ALGORITHMS AND HARDWARE STRUCTURES FOR REAL-TIME COMPRESSION OF PROGRAM TRACES

by

VLADIMIR UZELAC

A DISSERTATION

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

HUNTSVILLE, ALABAMA

2010

DISSERTATION APPROVAL FORM

Submitted by Vladimir Uzelac in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the 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 dissertation. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy 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	Doctor of Philosophy	College/Dept.	Engineering/Electrical and Computer Engineering

Name of Candidate Vladimir Uzelac

Title Algorithms and Hardware Structures for Real-Time Compression of Program Traces

Rising complexity of both hardware and software of modern embedded systems-on-a-chip makes software development and system verification the most critical step in a system development. To expedite system verification and program debugging chip manufacturers increasingly pay attention to hardware infrastructure for program debugging and tracing. The program tracing infrastructure includes logic to capture program execution traces, buffers to store traces on the chip and a trace port through which the trace is read by the debug tools. In order to cope with the size and bandwidth requirements for the real-time tracing, hardware compression of traces is required. In this dissertation, we present and analyze methods for on-the-fly compression of three components of the program (software) trace: addresses of executed instructions, addresses of memory referencing instructions and results of memory load instructions – the data brought to the system. We introduce a number of new algorithms for compression of individual trace components and explore their design space. We demonstrate that the proposed hardware trace compressors enable unobtrusive program tracing in real-time at the minimal additional hardware cost.

Abstract Approval:	Committee Chair	
	Department Chair	
	Graduate Dean	

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Dr. Aleksandar Milenković, for his excellent guidance, care and patience from the very first day of my graduate studies. He was a great advisor in my professional and private life, and he helped me easily overcome all the hurdles and challenges during the four years of my studies. He guided me on how to write conference and journal papers and he spent countless hours proofreading my research papers and discussing the results of my research.

Also, I would like to thank Dr. Emil Jovanov and Dr. Reza Adhami, for their financial help and support I received from the very first day of my graduate studies. While working as a teaching assistant for Dr. Jovanov, I had an opportunity to learn many valuable skills in embedded system design. I would like to thank Dr. Jeffrey Kulick who showed great interest in my education and research and helped me with his expertise and advice. During my studies at The University at Alabama in Huntsville, I have had contacts with many students, faculty and staff. Their friendly and supportive attitude was of great help, and I thank all of them.

In the past four years my wife was responsible for managing all other aspects of our life, enduring with me through the bad and good. Her patience and emotional stability made my commitment to this research possible. Thank you Tijana for all your support and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Embedded computing drives important aspects of our life including the modern communications, transportation, medicine and entertainment. The number of embedded processors by far surpasses the number of processors used for desktop and server computing. For example, a 2009 smartphone typically includes several processor cores [1] and a modern luxury car may have over 70 different processors and microcontrollers [2]. With emergence of ubiquitous computing and wireless sensor networks, we expect further diversification of embedded processors and their applications.

Current technology and economic trends pose unique challenges to the design and the operation of embedded computer systems. Semiconductor technology continues to provide cheaper, smaller, and faster transistors with each new technology generation, and we can integrate more and more transistors on a single chip. However, more aggressive technologies suffer from lower component reliability. Next, increased sophistication of hardware blocks and an increased level of integration limit the observability of internal signals. In result, the time spent in post-silicon debug and verification has grown steadily as we move from one technology generation to the next [3]. On the other side, software designer's challenges are also on the rise: increased hardware complexity enables more sophisticated applications. The software stack includes many layers, from hardware bring-up, low-level software, OS/RTOS porting, application developing, system integration and performance tuning & optimization, production tests, in-field maintenance, and failure analysis. Growing software complexity often leads to project failures or lost revenue if very tight time-to-market goals are not met. According to a study published by the National Institutes of Standards, software developers typically spend 50%-70% of their development time in program debugging [4]. This time is likely to continue growing with a shift from single- to multi-threaded applications -- developing parallel programs is known to be a more challenging task than developing sequential programs. Hence, debugging and testing becomes one of the most critical steps in the design and operation of modern embedded computer systems.

1.1 Software Debugging Challenges

Ideally, system designers and software developers would like to be able to answer the simple question "What is my system doing?" at any point in the design and test cycle. However, achieving complete visibility of all signals in real time in modern embedded platforms is not feasible due to limited I/O bandwidth and high internal complexity.

Modern embedded computer systems are built as Systems-On-a-Chip (SOCs) as illustrated in Figure 1.1. SOCs now replace traditional designs that consist of multiple integrated circuits connected on a printed circuit board (PCB). This transition from systems-on-a-board to systems-on-a-chip prevents system designers from observing and controlling internal signals. One approach to addressing this problem is the development of a dedicated In-Circuit-Emulator (ICE) with additional support for debugging. However, this approach is cost-prohibitive. Furthermore, the ICE's physical characteristics such as chip floorplan, pin layout, and timing characteristics, differ from the targeted SoC. An alternative approach is to incorporate a trace module on the chip – a dedicated hardware resource solely devoted to debugging. The trace module captures, buffers, and sends out a hardware trace – a recorded sequence of events related to program execution, including program execution trace, data trace, and interconnect signals.

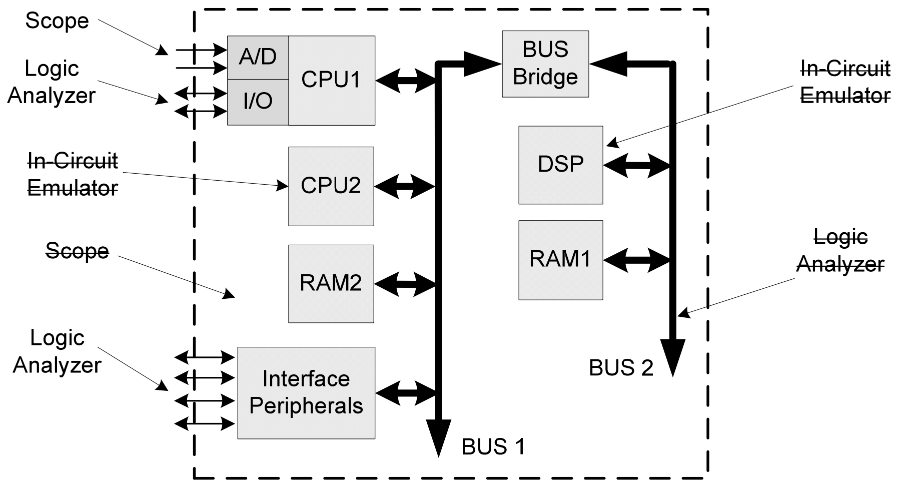


Figure 1.1 System-on-Chip Observabililty
(Strikethroughs represent a change in observability when the identical system is built on a printed circuit board)

Debugging and testing of embedded processors is traditionally done through a JTAG port that supports two basic functions: stopping the processor at any instruction or data access and examining the system state or changing it from outside. The problem with this approach is that it is obtrusive – the order of events during debugging may deviate from the order of events during “native” program execution when no interference from debugging operations is present. These deviations can cause the original problem to disappear in the debug run. For example, debugging operations may interfere with program execution in such a way that the data races we are trying to locate disappear. Moreover, stepping through the program is time-consuming for programmers and is simply not an option for debugging real-time embedded systems. For example, setting a breakpoint may be impossible or harmful in real-time systems such as a hard drive or a vehicle engine controller. A number of even more challenging issues arise in multi-core systems. They may have multiple clock and power domains, and we must be able to support debugging of each core, regardless of what the other cores are doing. Debugging through a JTAG port is not well suited to meet these challenges.

1.2 The Case for Hardware Tracing

Recognizing debugging challenges and issues, many vendors have developed modules with tracing capabilities and integrated them into their embedded platforms, e.g., ARM’s Embedded Trace Macrocell [5], MIPS’s PDTrace [6] and OCDS from Infineon [7] with a corresponding trace module from Freescale [8]. The IEEE’s Industry Standard and Technology Organization has proposed a standard for a global embedded processor debug interface (Nexus 5001) [9]. 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.

Beside resolving debugging challenges and issues in single-core system, tracing is essential in debugging of multi-core systems; various interdependencies and interactions cannot be observed with the simple snapshot of the system state, but continual tracing is needed in order to find bugs and other unwanted situations.

Hardware traces can be classified into three categories depending on the type of information they contain: program (or instruction) traces, data traces (from the memory bus) and system traces (various signals of interest in debugging the implemented hardware or observing the inter-cores dependencies and so on). Figure 1.2 illustrates how the tracing helps us to regain the observably into the system-on chip-behavior; real-time traces from different subsystems can be analyzed offline in order to debug and develop the system and compensate for the lack of observability through logic-scopes.

In this dissertation we focus on program traces which consist of instruction address (program execution) traces, data address traces (addresses of memory referencing instructions) and data value traces (data brought to the CPU using load instructions). These traces are widely used for both hardware and software debugging as well as for program optimization and tuning. While data value traces are sufficient to replay the program offline and analyze its behavior, this type of tracing is often not favored due to its high bandwidth requirements. On contrary, program execution traces have much lower bandwidth requirements while allowing for simple debugging tasks which are sufficient for many testing, debugging and verification purposes.

Tracing requires huge amount of data to be sent out of a chip. For example, a processor running at 1GHz produces gigabytes of trace information for just one second of execution time. Thus, the tracing process must rely on very large on-chip buffers to store the traces of large program segments and/or on relatively wide trace ports that can transfer a large amount of trace data in real-time.

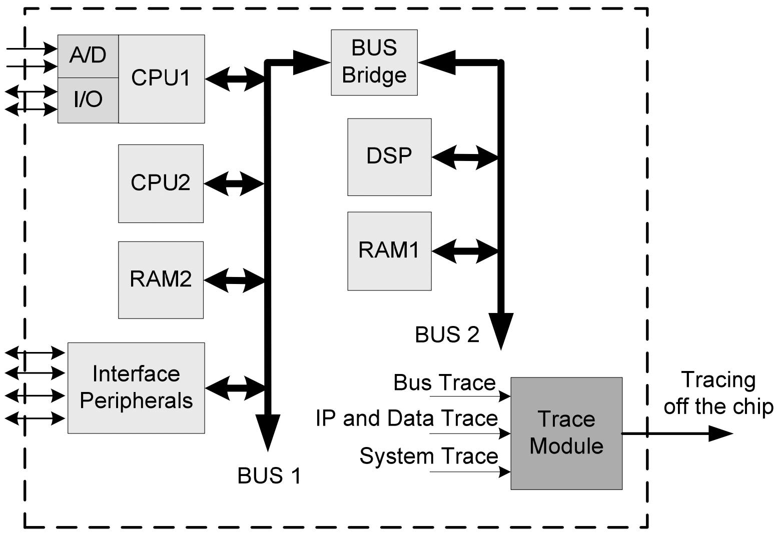


Figure 1.2 Observability of SoC operation using different types of traces

1.3 The Case for Trace Compression

Many existing trace modules employ program trace compression and buffering to achieve a bandwidth of about one bit/instruction/CPU on the trace port at the cost of roughly 7,000 gates (for instruction addresses only) [10]. These modules often use very rudimentary compression, such as differential encoding of consecutive instruction and data addresses. However, they still rely on large on-chip buffers that can capture the program trace in real-time for a relatively short program segment and wide trace ports to read out trace information. If we want to capture program traces of larger program segments, we will need larger trace buffers and/or wider trace ports. However, large trace buffers and wide trace ports significantly increase the system complexity and cost. Moreover, they do not scale well, which is a significant problem in the era of multi-core chips, especially considering the rising need for data tracing, which has much larger bandwidth requirements than the program execution traces. For example, the number of package pins does not increase linearly with the increase of processor speed, or the number of transistors on the chip, as we move from one generation to the next-one.

In this dissertation we argue that tracing can be significantly improved and the debugging costs decreased by using cost-effective hardware structures that compress program traces. The existing trace modules cannot guarantee unobtrusive tracing, rather they strive to provide minimally obtrusive program tracing often providing incomplete traces. However, this limits applicability of the trace modules in debugging real-time embedded systems where interference from debugging operations often cannot be tolerated. The proposed trace compression algorithms and hardware structures are designed to reduce the cost of debugging infrastructure through reducing requirements for large on-chip trace buffers and wide trace ports. In addition, the proposed trace compressors allow for unobtrusive tracing in real-time.

1.4 Contributions

In this dissertation we introduce a number of cost-effective hardware-based and real-time trace compression techniques. We characterize program and data traces of embedded processors that run typical benchmark programs. Based on these characteristics we develop several novel algorithms and trace compression structures that can be used in trace modules of future embedded systems. We explore the design space including trace port bandwidth requirements and implementation complexity and find the optimal design strategies that guarantee unobtrusive tracing in real-time at minimal hardware cost. More specifically, our contributions are as follows.

- We have performed characterization of program execution traces in order to draw conclusions about the best approach for compression of these types of traces.

- We have developed a Double Move-To-Front method for compression of program execution traces.
- We have developed a Stream Cache and Last Stream Predictor method for compression of program execution traces.
- We have developed a Branch Predictor based method for compression of program execution traces.
- We have developed a cost-effective filtering technique for reducing the size of data address traces. The technique relies on reuse of register values to re-generate data addresses in a software debugger.
- We have developed a technique for compression of high order data address bits. The compression of higher-order bits requires a simple compressor while the compressed traces have low variation in required bandwidth, eliminating the possibility of dropping traces when the available bandwidth is saturated.
- We have characterized the behavior of data values in typical embedded systems applications to gain insight into the compressibility of these types of traces.
- We modified and improved an existing scheme that relies on reusing data values already residing in the processor's cache to re-generate register file values during the program execution in a software debugger.

1.5 Outline

The remainder of this dissertation is organized as follows. In Chapter 2 we discuss the existing approaches and standards for software debugging and tracing in embedded systems. In Chapter 3 we survey general-purpose compression algorithms as well as a number of software-based and hardware-based trace-specific compression algorithms. In Chapter 4 we describe the proposed mechanism for capturing and compression of program execution traces. We introduce three methods for compression of program execution traces, namely, (a) Double Move-To-Front (b) Stream Descriptor Cache and Last Value Predictor, and (c) Branch Predictor Method. For each method we describe algorithms and compressor structures and perform a detailed design space exploration including trace port bandwidth and design complexity. In Chapter 5 we introduce and explore the effectiveness of two methods for capturing and compressing data address traces, namely, (a) Data Address Trace Filtering through a partial replay of the register file, and (b) Data Address Trace Compression through exploiting low variability of higher address bits. In Chapter 6 we analyze typical load value traces and explore their amiability to compression. We propose several low-complexity approaches to the compression of load value traces and explore their effectiveness. Finally, in Chapter 7 we give concluding remarks.

CHAPTER 2

PROGRAM TRACING: BACKGROUND

Program traces encompass recorded sequences of instruction addresses, data addresses, and data values captured during program execution. Program execution or instruction traces are instrumental for software debugging and are also used in profiling for tuning and optimization. However, some software bugs require a complete re-creation of memory values. In such cases collection of data address and data value traces is needed.

In Section 2.1 we discuss types of program tracing depending on how traces are collected: software traces (Section 2.1.1) and hardware traces (Section 2.1.2). In Section 2.2 we focus on hardware supported tracing and debug infrastructure needed to support it because it is crucial in debugging embedded systems. Section 2.3 describes several representative examples of commercial trace modules.

2.1 Types of Tracing

2.1.1 Software Tracing

The most common form of program traces are software traces. Software traces are traditionally created using a trace collector process that runs on the same platform as the traced program. Exception handlers with different granularity (e.g., single-step, step-on-branch) are typically used to capture relevant trace data. The traces are then stored to dedicated trace buffers in system memory. A separate process is responsible for emptying the buffers to an external storage (e.g., hard disk) using available communication channels. A software debugger then reads and analyzes the collected trace. Software tracing has a relatively low implementation cost and offers flexibility in selecting the type of events that are captured during tracing. The main drawback is its obtrusiveness – the trace collector and trace transfer processes consume the CPU time, system memory, and communication channel bandwidth. Thus, software tracing often slows down the program execution by an order of magnitude or more.

A number of software tools for trace collection and analysis have been developed. The best known tools are Shade [11] for instrumentation of SPARC and MIPS architectures; Performance Inspector [12] for x86, x86_64 and PowerPC64 architectures; and PIN [13, 14] for x86 and ARM architectures. Microsoft has developed a tool for instruction level tracing [15] that also includes a compressor to reduce the trace storage requirements. Here we will briefly discuss the PIN tool.

The PIN tool [13] performs run-time binary instrumentation of Linux and Windows applications through a just-in-time compiler (JIT). PIN starts by intercepting the execution of the first instruction of the executable and generating ("compiling") new code for the straight-line code sequence starting at the current instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but PIN ensures that it regains control when a branch exits the sequence. After regaining control, PIN generates more code for the branch target and continues execution. Every time the JIT compiler fetches a code segment, a PIN analysis tool (called *Pintool*) has the opportunity to instrument it before it is translated for execution.

PIN implements different optimizations to reduce the number of handshaking operations between the JIT and actual program executable, resulting in reduced tracing overhead. For example, a branch predictor for indirect branches in a JIT compiler allows for less frequent switching to the original code and obtaining the target from there [14]. In spite of these optimizations it imposes a significant slowdown (4 to 12 times just for a simple counting of the number of executed instructions). This slowdown is acceptable for tracing of different general-purpose applications but is not feasible for tracing of real-time embedded systems.

2.1.2 Hardware Tracing

Hardware program traces are collected using dedicated on-chip logic that can capture and store address, data, and control signals. The implementation cost of debug infrastructure for hardware tracing varies depending on trace types that are being collected and the level of obtrusiveness. With obtrusive hardware tracing, on-chip resources are used to perform tracing, compression and qualification of traces, but ultimately, the traces are transferred to system memory using system buses (thus interfering with system operation). With minimally obtrusive or completely unobtrusive tracing, dedicated on-chip buffers are used to store traces and a separate trace port is used to transfer the collected traces off-chip.

Figure 2.1 gives a system view of a typical on-chip debug infrastructure for unobtrusive hardware tracing. A target processor provides necessary internal signals to the trace module (*Trace Signals* block). The trace module then selects what signals are of interest in the current trace cycle based on debug control and configuration commands received from the software debugger, it starts and stops tracing based on specified external or internal triggers. The trace module optionally performs compression of traces. Finally, traces are output through a trace port to the external trace storage memory, using a trace communication protocol (*Trace Protocol* block). The trace protocol block is used not only for tracing data off the chip, but it also serves as an access point to the on-chip trace related resources from the software debugger side.

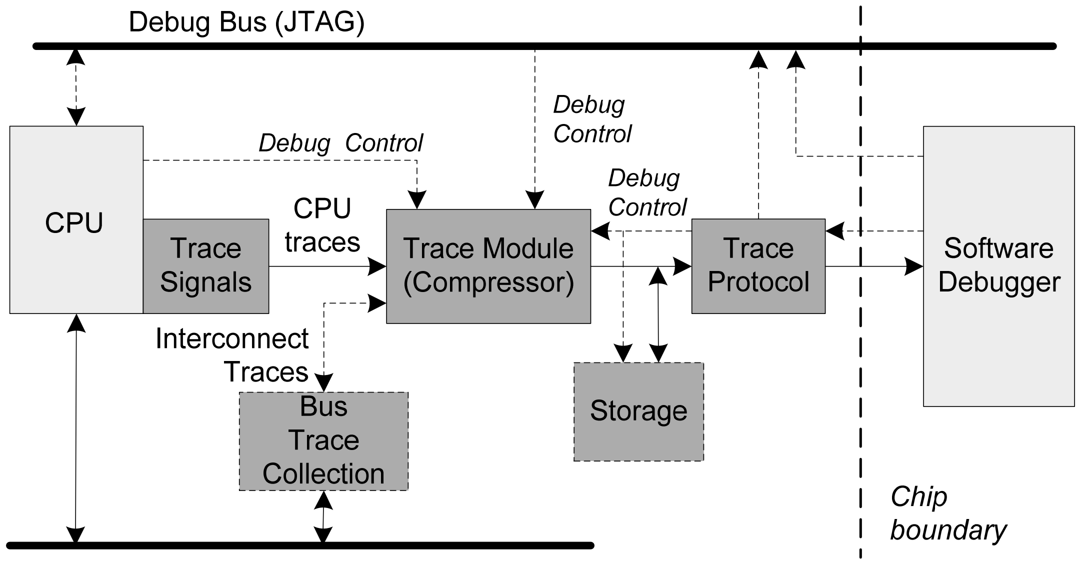


Figure 2.1 System view of the hardware support for unobtrusive program tracing

The trace module is controlled by *Debug Control* signals. *Debug Control* signals can come from the trace protocol subsystem if the protocol implements control features besides tracing the data only. *Debug Control* signals can also come from the CPU side. For example, software interrupt routines can be used to setup the trace module operation. The control signals can come from the system level access protocol, JTAG or *Debug Bus* as shown in Figure 2.1, which allows for more flexible control of the trace module operation (the unified system level debug bus accepts control signals from different processors on the chip or from the software debugger). When reading out already stored traces, the same set of *Debug Control* signals instruments the storage memory to output traces out of the chip.

A *Bus Trace Collection* block serves as an additional source of traces for the trace module. For example, in a multi-core system, traces collected directly from the shared bus give better insight into inter-processor dependencies and allow for better correlation of different processors' execution.

2.2 Hardware Supported Unobtrusive Tracing

In this section we give more details on the debug infrastructure that supports minimally obtrusive or unobtrusive hardware tracing as illustrated in Figure 2.1. Section 2.2.1 gives examples of the existing implementations of the *Trace Signals* block. Section 2.2.2 discusses typical operations of trace modules. Section 2.2.3 discusses protocols used to access trace and debug resources on the chip from the software debugger side, most notably the one specified by the IEEE-ISTO 5001 (Nexus) standard.

2.2.1 CPU Trace Signals

A processor's trace signals reflect the processor's internal state. They typically include signals that carry information about the currently executing instruction (e.g., instruction word and address) and values on the memory buses (data addresses and data values). In addition, various pipeline and system state information may be available.

For example, the Tensilica LX2 processor family implements a configurable trace port with different number of trace signals [16]. The simplest processor trace port is 72-bit wide and includes a 32-bit program counter, 8-bit debug status field, and a 32-bit debug data field. The debug status field contains information about the validity of the current instruction and its type and size. Together with the debug data field, various events can be encoded, for example, causes of pipeline bubbles (e.g., read-after-write dependency, structural hazard, I-cache or D-cache miss), pipeline dependency information, and stall information. Next, the debug status and data fields can signal whether an instruction is not executed using the CPU but from the on-chip debugger hardware (instrumented from software). If the data tracing option is enabled, a processor's port width ranges from 140 to 400 bits. Additional bits include the load/store units address, data values, and status signals. Load/store data field size depends on the memory bus width; Load/store status specifies the type of a memory or cache access and the size of the memory referencing request. If the CPU implements two load/store units, the load/store trace signals exist for each unit.

Similarly to the Tensilica's processor trace port, the corresponding trace ports for MIPS and Freescale include status signals that inform the trace module about what type of data is presented on the unified trace port. The MIPS trace port [6] supports tracing of out-of-order load and store instructions by sending out a signal which contains the information of the load's actual position in the load store queue (e.g., received load value corresponds to the Nth oldest load instruction). In addition to memory addresses and data, the Freescale MPC565 [8] trace port includes the ownership information (which device on the bus issued a memory referencing instruction).

The Xilinx Microblaze processor [17] implements a trace port with approximately 200 signals [18]. In addition to detailed information on instruction traces and data traces from different memory modules, it includes information about the executing process, unmasked interrupt occurrences, pipeline halted information, exception type, information of pipeline advance for each pipeline stage, debug mode operation, etc.

2.2.2 Trace Module Details

Trace modules today offer various features. The most important ones are related to the qualification of traced data. Trace qualification assumes control on what to trace and when to trace (implemented using various triggers). Other features include a certain level of automatic processing of traces to reduce the amount of unnecessary data presented to the user, compression of traces, and overall a tracing mechanism implementation which allows for correlation of traces in a multi-core system (this may also include timestamping of traces).

The minimum set of tracing triggers allows starting and stopping tracing when certain instruction addresses occur. More advanced triggers may come from conditioning other signals coming from the processor. Finally, the most advanced triggers may come from the other processors or components on the chip (this is often referred to as cross-triggering). The cross-triggering often requires additional considerations on synchronization of traces coming from different processers and their correlation in time.

For example, Tensilica offers TRAX [16], a tracing module which connects to the Tensilica CPU's trace port and outputs the traced data to the embedded trace memory according to Nexus format. Traced data can be read out through the JTAG. The current TRAX version implements instruction tracing only. This indicates the importance of the program execution tracing only, due to its small bandwidth and storage requirements while enabling basic debugging tasks. In addition, TRAX implements cross-triggering for stopping and starting tracing initiated from other cores in the system.

Xilinx offers the MDM and XTRACE debugging tracing modules [17-19]. The XTRACE module encodes and multiplexes signals from the Microblaze trace port into 22 signals, which are output off the chip, or the trace session can be saved to the on-chip trace memory. Internal cross-trigger functionality is used to setup the trace sessions. The MDM module is used to control the XTRACE module. It is accessible through the JTAG port. For basic debugging purposes, MDM can output information through the UART and it also has access to the Microblaze fast main bus fabric (FSL).

Freescale offers READI [8], a trace and debug module which provides real-time tracing capabilities for different Freescale processors. The module implements a Nexus Class 3 interface which allows for instruction and data tracing and also allows the control of various processor registers and on-chip memories from an external software debugger.

The Embedded Trace Macrocell (ETM) module provides debug and trace facilities for ARM processors [5]. They allow information on the processor's state to be captured both before and after a specific event, unobtrusive to the processor execution. The ETM can be configured in software to capture only select trace information and only after a specific sequence of conditions. A trace port and FIFO, both configurable, allow the compressed trace data to be read from the chip by an external trace port analyzer, again unobtrusive to the processor execution. The trace port can be configured from a 1 to 32-bit data bus, with a trace clock independent to the core clock.

The functionality of any of the ETMs can be extended by the addition of an Embedded Trace Buffer (ETB) [20]. The ETB, an on-chip memory, stores the captured traces so that later they can be read out at a reduced clock rate. This can be done through the JTAG port of the device without using expensive trace port pins.

2.2.3 Tracing and Debugging Protocols

The most widely used protocol to access on-chip resources is JTAG. JTAG connects on-chip registers into a scan chain which allows the software debugger to modify their values, which in turn, drive the logic internal signals according to predefined test patterns. Each device within a chip has registers already included into the device's scan chain. In a system-on-a-chip with multiple logic devices all individual scan chains are connected together allowing the software debugger to access any device on the chip. Because of long scan chains and a serialized operation, JTAG is suitable only for run/stop based debugging, where the software debugger halts the processor, or the whole system, and then reads out registers in the scan chain to obtain information about the system state. JTAG has been improved over time to offer more debugging capabilities. The IEEE 1149.7 standard [21] was introduced to improve JTAG debug by implementing a system level bypass which allows for quicker accesses to a selected device using much shorter scan chains. The IEEE P1687 (IJTAG) standard focuses on standardizing the way an external system communicates with any form of internal design debug and test logic via the JTAG port. IJTAG offers a standard way of connecting, accessing, analyzing, and describing embedded instrumentation hardware. Instrumentation hardware includes not only debug and test logic but any circuitry used for device characterization, monitoring, configuration or functional use. The IJTAG infrastructure resides at the deeper level than the JTAG and is accessible through JTAG. The standard defines access mechanisms to the instruments and also defines their hardware and software description. This way, the actual instrument details are hidden from the user allowing for better portability of instruments across different systems and their use with different testing tools.

MIPI Test & Debug. A Mobile Industry Processor Interface Test and Debug working group adds the high bandwidth unidirectional trace interface to the IEEE P1149.7 interface for use in smart phones and other network devices [22, 23]. The tracing is implemented through a System Trace Module (STM), which collects software and debug data from internal buses, encodes the data and sends it out of the chip through a Parallel Trace Interface (PTI). The PTI is an external port of configurable width. A protocol, System Trace Protocol (STP) is used for encoding trace information. MIPI aims to offer a single debug and trace port for multi-core systems; thus, it allows tracing from up to 256 sources (either OS processes or hardware sources, e.g., different processors), offers automatic timestamping and has a configurable PTI width to conform various tracing requirements.

IEEE 5001 (Nexus). The Nexus standard defines trace and debug interface, including associated protocols and infrastructure that can serve tracing and controlling of multiple cores on a chip from the software debugger. Communication between the Nexus internal logic and a software debugger can be achieved through either a JTAG port already included on-chip or through the dedicated and extensible AUX port. An AUX output interface is typically used for tracing data out of chip. The Nexus employs a packet-based messaging tracing scheme with packet headers providing information about data source and destination and type of payload. An AUX input interface allows for bypassing the JTAG control sequences, thus achieving faster response when configuring and calibrating the system. Trace collection mechanisms are not addressed by the Nexus.

Nexus allows vendors to adopt the standard at four levels or classes of operation. Higher classes include support for more complex debug operations, but in turn require more on-chip resources:

- **Class1: Basic Run Control**. System implements read/write to user registers and memories, single step debug through break-/watch- points and access to registers and memory locations when the execution is halted.
- **Class2: Instruction Trace**. In addition to class 1, the system includes support for capturing processor execution trace. Also, it allows for monitoring of process ownership which is useful to correlate simultaneously executed threads in time.
- **Class 3: Data Trace.** In addition to class 2, the system includes support for monitoring and read/write to data and memory locations, including the I/O reads and writes. This mainly relates to implementation of full tracing capabilities.
- **Class 4: Remote processor control and advanced trace**. In addition to class 3, the system includes support for direct processor control using memory substitution. Memory substitution allows the processor to execute instructions

from the trace port rather than from the memory. Memory substitution is implemented using address remapping for the I/O space (where the trace port is located) and for the memory (where the original executing program resides). An advanced trace includes tracing of parameters that are not only used to determine program execution but also performance parameters and trace of interconnect signals.

The Nexus packet-based messaging tracing scheme includes different types of messages which follow different implementation classes. There are five main types of messages:

- **Status**. Indicate status information messages from the target.
- **General register read/write**. Memory mapped reads and writes between software tools and Nexus registers. These messages can be used for run control and configuration of watchpoint/breakpoint operations.
- **Program Trace**. Trace of instruction addresses reduced to trace of branches only (or other program discontinuities).
- **Data Trace**. Trace of data addresses and values. Nexus also supports data acquisition instructions for streaming export of larger amounts of system information (data from an on-chip trace buffer).
- **Memory Access.** Non-intrusive access to internal memory blocks.

Figure 2.2 shows the system level organization of the Nexus infrastructure. It includes various FSM’s to access internal registers, registers to control the Nexus operation and the TCODE formatting logic, which performs type-based packaging of trace messages coming from the underlying system using TCODE header.

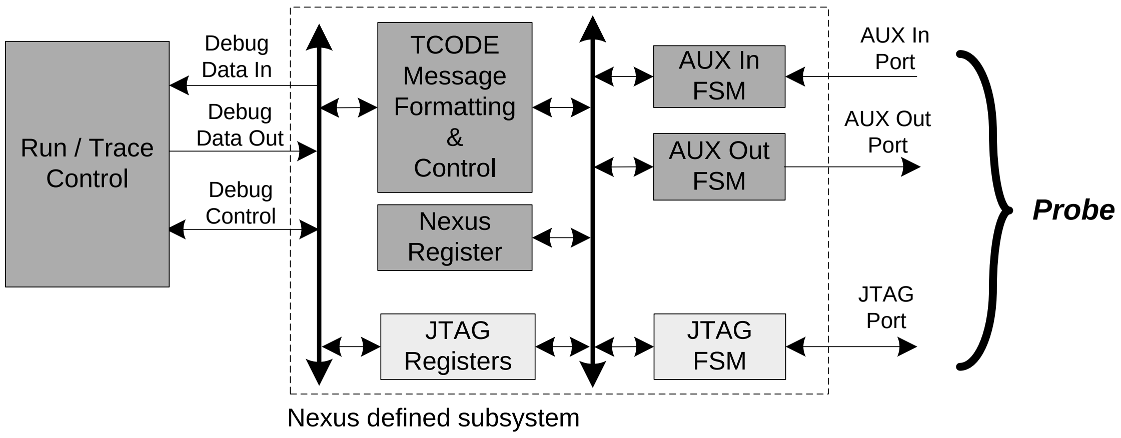


Figure 2.2 Nexus system level organization

2.3 Hardware Trace Infrastructure Examples

In this section we focus on infrastructures that allow for collection of traces inside of a chip and transferring the traces out of the chip. The examples do not necessarily follow the unobtrusive tracing system description from Figure 2.1, but they describe different important intricacies of trace collection and storage in real systems. For example, the OCP-IP Debug in Section 2.3.1 focuses on collection of traces from the main system bus in a multi-core system. The BugNet system in Section 2.3.2 is an obtrusive mechanism which explains different mechanisms when tracing the shared memory and presents some solutions to reduce the size of trace information carried out. Finally, Section 2.3.3 describes the CoreSight infrastructure, a complete unobtrusive system for real-time tracing of all cores in a multi-core environment.

2.3.1 OCP-IP Debug

Open Core Protocol (OCP) is a widely used standard IP core interface [24, 25]. OCP facilitates an IP cores plug-and- play approach by decoupling the cores from other parts of the chip using a clearly specified core interface protocol. In an OCP system, any proprietary component (e.g., processors, memory modules, and I/O devices) can become OCP compliant by using a wrapper which implements the OCP interface. The wrapper works as a socket, allowing any component to be easily attached and detached to/from an existing port on the OCP interconnect. The socket-based approach also speeds up the verification and optimization of a design.

Part of the OCP standard related to the debug has been developed by the OCP Debug Working Group [26]. This group aims to provide a socket based debug standard for OCP compliant SoC. OCP debug efforts focus on defining the set of debug signals, at the OCP fabric level, which can be used by the external software tools for debugging, testing and other verification purposes. The OCP debug socket is attached to the OCP bus and thus, resides at the deepest level of the debugging infrastructure, and the standard does not addresses how the debug data is traced out of the system (MIPI, Nexus or other protocols can be used for that purpose). The OCP debug socket actual implementation can be adapted to different debugging needs: software centric debugging requires basic run control for cores and program and data tracing. Hardware centric debugging exposes hardware logic signals used for hardware verification purposes. System centric debugging allows for observation of interactions of different components on the chip, such as comparative debugging of any two cores on the chip and operates independently of implemented hardware in order to capture pre-reset and post-crash events and to perform. A multi-core system debug greatly benefits from using the OCP bus and an OCP debug socket. A debug socket, attached to the OCP fabric, has predefined functionalities, such as monitoring of all transactions and responses on the bus in a cycle-accurate manner and it also implements automatic trace qualification and performance analysis. Various levels of support for debug are defined through basic and extended debug signals:

Basic:

- Debug and run control for cores (run/stop/halt, watch/break- points…) as seen in JTAG only debug solutions.
- Cross-triggering between multiple cores and events to allow global and distributed event recognition across the multi-core system.
- Synchronized run control supports clock synchronized program execution of two cores that run asynchronously in the normal case, allowing for time alignment of their instruction streams in order to study interdependencies.
- Basic bus traffic observation through system trace with control of trace start/stop using triggers. Tracing includes filtering based on OCP operations (e.g., Initiator, thread, address range, DMA).

Extended:

- Performance Counters (single and multi-core enable observation of selected parameters through summary only, which reduces the output bandwidth.
- Time-stamping allows for correlation of events in a distributed system or events coming from asynchronous parts of system.
- Power Management Monitoring includes observation of gated clock domains and voltage domains in aware architectures, where a debugging process must be performed across all IPs even those that are disconnected at the time.

- Security monitoring requires selective enabling debug of sensitive locations at different times and with different levels of authorization.

2.3.2 Hardware/Software (Hybrid) Tracing

Hybrid techniques rely on hardware structures to perform compression and reduction of traces, but ultimately, they require the use of system memory as trace storage. This also has an impact on processor execution due to the sharing of memory resources and on-chip buses. One such technique is BugNet [27] which uses the main memory to store traces.

BugNet specifically addresses inter-core dependencies by attaching a modest hardware to a shared cache coherence protocol. By recording the outcomes of all memory races, a debugger is supplied with sufficient information to correlate different processes in time. BugNet additionally reduces the amount of data coming from this coherence monitor by introducing a hardware algorithm that filters out the memory race outcomes that can be inferred from other races that are already traced [28].

BugNet aims to reduce the amount of traced data needed for a debugger to find the bug by recording only the last one second of program execution before the failure. This is because usually program failure is often due to a bug just a few million instructions away from the failure. To support this, BugNet uses *Checkpoints* – a snapshot of system state – at each second and tracing of the system within last second.

Regular data traces from/to each of the processors are compressed using a simple dictionary based compressor. Additionally, data traces are reduced by logging only the first load into the cache; a load accessing a memory location needs to be logged only if it is the first access to that memory location. The values of other loads can be re-generated during replay in a software debugger that models caches and memories. To implement this optimization, hardware extensions to L1 and L2 caches are necessary.

The basic architecture of BugNet is shown in Figure 2.3. Shaded parts of the system are part of BugNet. BugNet attaches a logger to each shared cache to record memory races outcomes and saves the traces to the Memory Race Buffer. The Checkpoint buffer records system state. Both buffers are dumped to main memory on error detection. BugNet limitations are the need for integration with the operational system, detection of only the bugs that lead to the system failure and obtrusive tracing due to usage of main memory.

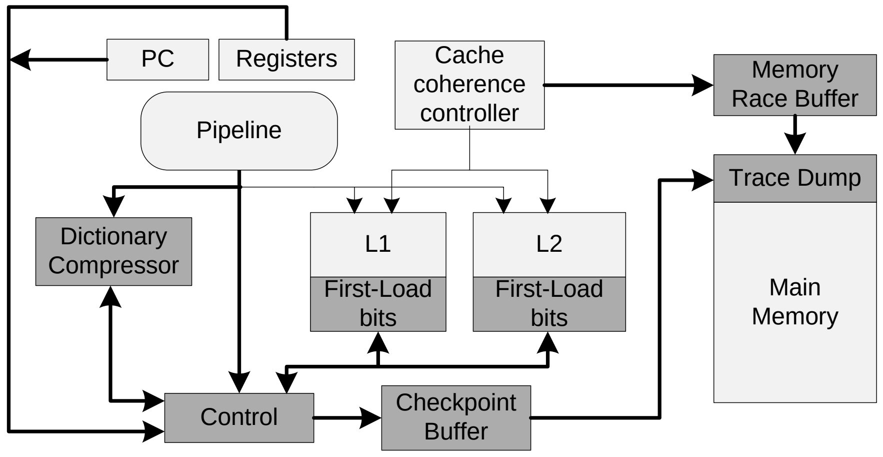


Figure 2.3 System view of BugNet architecture

2.3.3 Hardware Based Debugging Infrastructure

CoreSight [29] provides a SoC architecture which enables a software engineer to fully debug and trace an entire SoC in real-time (Figure 2.4). CoreSight specifically targets multiprocessor systems where two or more cores have to be halted simultaneously and tracing is required to be cycle accurate. The cross trigger matrix, similar to other triggering mechanisms, enables simultaneous triggering of multiple cores including breakpoints and interrupts. As a result, when a single core is halted (or triggered), all connected cores and peripherals are halted (or triggered), thus providing the user with both control and observability of the system state. To provide the user with correctly correlated traces from different processors, a trace funnel puts data coming from different ETMs in order to the AMBA Trace Bus (ATB). From here, data can be saved to the Embedded Trace Buffers or can be driven directly to the output port. Together with the RealView software analysis tool, ARM aims to provide users with full support for correlations and synchronization of multi-processor executions.

CoreSight hardware is accessible through the Debug access port (DAP), which can be also connected to the standard JTAG port. DAP accesses the AMBA APB bus, which can communicate with all the debugging and tracing devices within the system (ETM, Cross-triggering, directly with the CPUs). CoreSight also includes an Instrumentation block, which is a simple hardware support for software based tracing (e.g., *printf* style debugging, tracing of OS and application events and emitting diagnostic system information).

While offering complete infrastructure for debugging complex systems on a chip, CoreSight does not address the compression of trace data. With a growing number of cores on a single chip, it's a question how CoreSight can scale in a multi-core environment due to the huge bandwidth requirements for tracing.

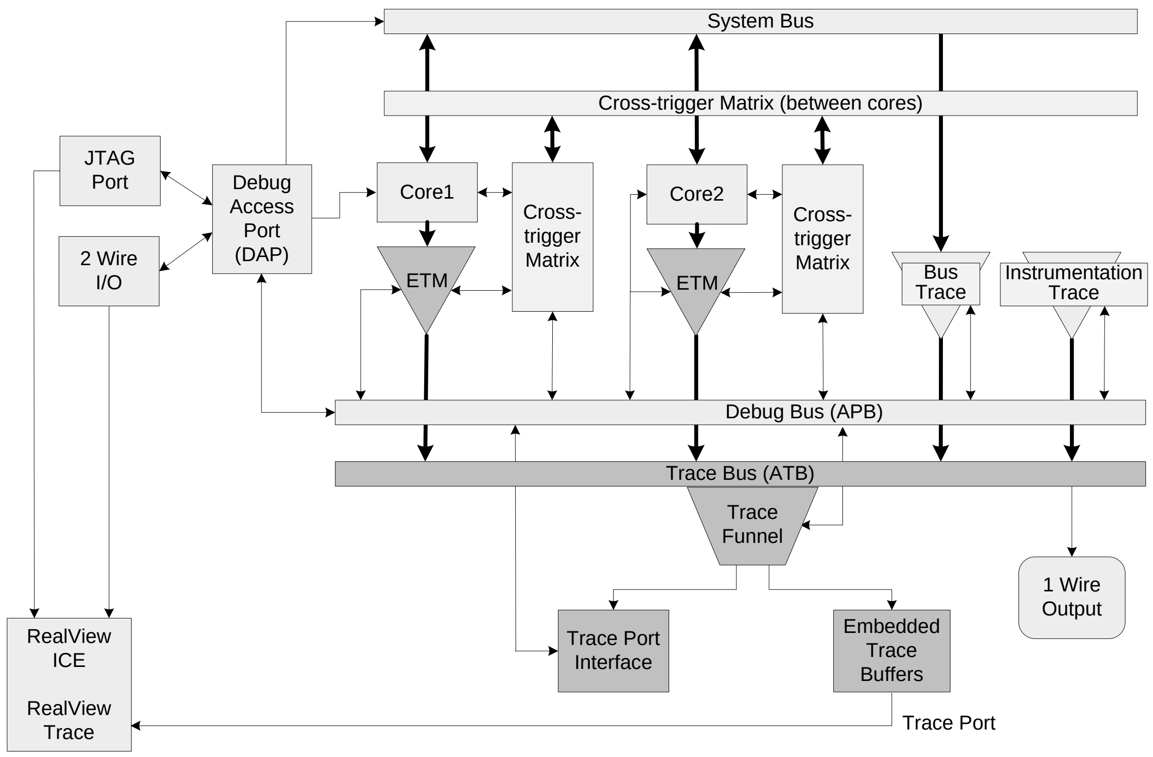


Figure 2.4 System view of ARM's CoreSight architecture

CHAPTER 3

TRACE COMPRESSION ALGORITHMS

Trace compression algorithms are designed to exploit redundancies found in program traces. Many of these redundancies can be extracted with general purpose compressors too, but trace-specific compressors can achieve similar or better compression ratios with less compute and storage resources. Hardware implementation of either general purpose or trace specific compressors is a very challenging task. To achieve high compression ratios, the compressors need large buffers, but they tend to be costly. Efficient compression mechanisms with modest hardware resources exploit unique features of program traces. For example, including architectural support for tracing in a processor (e.g., caches, execution units, information on program execution) and instruction set simulator in a software debugger enables significant reduction in the number of required trace records.

The operation and principles behind the most widely used general-purpose software compression applications are described in Section 3.1. Section 3.2 describes specialized techniques and approaches for compression of program traces. In Section 3.3

we discuss hardware implementations of various trace compression techniques and describe several existing compression techniques which can be implemented in hardware, including those that rely on processor architectural state.

3.1 General-Purpose Compression Algorithms

In this section we briefly describe the most widely used software compression applications, *gzip* , *bzip2* and *Sequitur*. They are based on distinctly different approaches to the compression of general data sets. *gzip* finds repeating patterns within a data set and replaces them with identifiers which are then efficiently encoded [30]. *bzip2* is based on Burrow-Wheeler transformation [31], and it reorganizes data before compression to reduce the entropy within the individual data sets. Both mechanisms also use different encoding techniques (Huffman coding [32, 33], arithmetic coding [34]) to produce a more compact description of input data set values. We also describe the operation of *Sequitur*, a relatively new mechanism with a novel approach to compression which builds hierarchies of patterns.

3.1.1 gzip

gzip is based on a variant of the *lz77* pattern match mechanism [30]. The *lz77* algorithm tries to find repeated patterns in the data set by comparing the incoming, yet to be processed, data with already processed data. Whenever a pattern match occurs, the repeating pattern is replaced with an identifier consisting of two values: *distance*, which indentifies the start position where the previous occurrence of the same pattern starts, and *length*, which indentifies the length of repeated pattern. To reduce the size of (*distance, length*) pairs, *gzip* uses two Huffman trees separately for *distance* and *length* in order to encode more frequent values with less bits. To improve efficiency, adaptive Huffman coding is used, where a new Huffman tree is built for each new block of data. A new block starts when the internal logic determines that the current Huffman tree is no longer the optimal one.

The *gzip* is important in this dissertation as the pattern repetitions are common within instruction and data addresses and a relatively simple pattern matching algorithm can be employed in hardware to compress the traces.

3.1.2 bzip2

bzip2 is a complex compression algorithm which uses different concepts from data encoding and the compression field. In its core, *bzip2* implements a reversible block-sort algorithm, or Burrow-Wheeler transformation [31]. The transformation does not compress the data but re-organizes it in such a way that it reduces the entropy of data in each separately processed block. For example, if the frequency of the different and most frequent data values, A and B, is approximately the same in one data set, the data set can be reorganized so that A values appear mostly in one part of the program (block 1) while B values appear mostly in other part of the program (block 2). This leads to lower entropy within the data set, when compressing block 1 and block 2 separately. However, an increased compression ratio must be high enough to compensate for the additional storage required to record the information about the reverse transformation. *bzip2* uses Move-to-Front (MTF) encoding in the second stage. MTF is a frequency based table, where recent and most frequent data are in the front of the table. MTF does not reduce the size of the input block, but adds additional reordering of data to decrease the entropy even more. Huffman coding is the final stage in the *bzip2* algorithm. A complex scheme implements multiple Huffman tables and dynamically decides on the most optimal one for currently processed data.

bzip is less amiable to hardware implementations because it relies on frequent memory operations to reorganize data and it also works with extremely large data sets (up to 900 KB). However, certain concepts, such as MTF, are used in this dissertation.

3.1.3 Sequitur

Sequitur [35] replaces two repeating symbols with a new one. For example, repeating symbols *a* followed by *b* can be replaced by a new symbol *A*. Further on, if the *A* is followed by symbol *c* repeatedly, the *Ac* pattern is replaced by, e.g., symbol *D*. In this way a hierarchy of patterns is captured into a smaller number of symbols. *Sequitur* has unique feature in that it preserves the context of original data; an original symbol or a pattern is easily found and reconstructed (e.g., symbol *D* from the compressed file is simply converted into *a*, *b*, *c* using the grammar file built during execution). *Sequitur* is highly unsuitable for hardware implementation as the algorithm saves the context information (grammar) for each new symbol built during the execution. Thus, the memory requirements can increase linearly with the processed trace and are in fact unbounded.

3.2 Specialized Algorithms for Instruction and Data Trace Compression

Various mechanisms exist which try to identify and exploit redundancies typical for instruction and data traces. Several approaches are described here. Many of them use one of the compression principles exploited by general purpose compressors and just adapts them to the expected behavior of data or instruction traces. Others exploit the correlation between data addresses (or values) with the instruction address in order to achieve greater compression of data traces.

In this dissertation, the most important compressors are those built from value predictors. Value predictors are used in a processor pipeline to improve latencies of memory operations by predicting data addresses and load values of fetched, and not yet processed load instruction.

3.2.1 Instruction Address Trace Compression

Instruction address compression techniques work by compressing only the information from control-flow changing instructions. This is possible as a simple decompressor, which has the basic information from program binary, can easily reconstruct all other instructions.

The simplest approach to compression is to replace an address with the offset from the last address, which is employed in Nexus [9]. This approach exploits the locality of instruction addresses, where consecutive branching often changes just lower instruction address bits, and thus, upper ones do not have to be traced.

Several techniques exist which try to replace the execution sequence with its identifier. For example, Whole Program Path (WPP) [36] instruments a program to produce a trace of acyclic paths. Then, a modification of the Sequitur algorithm is used to compress acyclic paths, represented by unique identifiers. WPP successfully finds the most frequent program paths, which is very useful for various types of analysis. However, since the WPP is not a single-pass mechanism, as it requires code instrumentation, its implementation in hardware is very limited. An advanced WPP implementation, time stamped WPP [37], enables fast access to the trace produced by a particular function. The mechanism separately performs WPP on traces belonging to different functions. PDI [38] associates dictionary entries with the most frequent instruction words. N-TUPLE [39] saves a number of control flow change identifiers per dictionary entry. In this way, a whole group of repeated identifiers can be replaced with the pointer to the dictionary entry. N-TUPLE exploits frequent patterns that exist among dynamic basic blocks in common programs.

A more complex approach to compression is through building a control flow graph, which together with information on transitions between graph nodes, describes the control flow changes. One such a technique is QPT [40]. QPT records only information about significant events and only transitions between basic blocks where a program chooses between alternative paths and only those transitions that are not part of a maximum spanning tree of a control flow graph. The actual instruction addresses can be regenerated using the control flow graph.

3.2.2 Data Address Compression

The Nexus standard [9] uses the same offset based encoding for both instruction addresses and data addresses. The compression benefits from spatial and temporal locality of memory references where the consecutive memory addresses differ by only a small value. Mache [41] replaces a data address with the offset from the last address of the same type. Types are instruction reference, data read and data write. PDATS [38] adds more complexity by introducing the variable length encoding of the offset field and includes an optional repetition counter.

Data addresses can be compressed by linking them to the program loops. Various mechanisms [42] [43] require two-pass mechanisms where first program loops are detected either using the control-flow analysis or program instrumentation. In a second pass, many of the data references are easily identified as being always constant per loop or always having the same offset from loop to loop. However, all other data addresses, with so-called chaotic behavior, are not compressed.

A special type of compressors tries to link the data addresses with the instruction block. A technique described in [44] records all possible data offsets and repetitions for a data address which is linked to an instruction block. Also, instruction blocks are compressed by indentifying all instruction blocks' successors and their repetition count. The memory requirements for this mechanism are very high, and the mechanism can be considered as only the first-pass reordering of memory and instruction addresses for an easier compression using a general purpose compressor. Stream Based Compression (SBC) [45] exploits the inherent characteristics of instruction and data address traces. Instruction traces consist of a fairly low number of different streams (streams are dynamic basic blocks described by their descriptor - starting address and the length, in number of instructions). Newly occurred stream descriptors are allocated in a table and the table search is performed upon each new stream occurrence. An index in a table replaces the whole stream descriptor. Data addresses traces have strong spatial/local locality which is exploited through a mechanism aware of frequent constant strides between consecutive memory referencing addresses.

Value predictors for data addresses. Value predictors for data addresses or data address predictors are cache like structures that store recently executed data addresses (last several data addresses per instruction address). Data address predictors are used in the processor's pipeline for data pre-fetching [46] (prediction of future memory references) and early load address calculation [47] (to enable issuing a load instruction early in the pipeline instead of waiting for the address calculation). Data address predictors can also be used for compression of data addresses. The compression is achieved by replacing a data address with an identifier which points to the cache entry which stores the correctly predicted address.

Data address predictors are usually organized as instruction pointer-indexed (IP-indexed) caches or tables, which store different information on data addresses recently issued by the load instruction. For example, last value (or address) predictors [48, 49] store the last several addresses seen by a load instruction (predictors are named LV*x* where *x* denotes the number of stored addresses). Stride based predictors are frequently used for both load address prediction and hardware pre-fetching due to their modest hardware requirements and the fact that many data addresses appear in strides (both consecutively in program or per instruction). This is particularly true for scientific applications where programs traverse large arrays of data [46, 50]. Context based predictors [51-53] are trying to predict the next data address based on the value of several last addresses. Context based predictors usually employ two caches (tables) where the first one stores the hashed history of the previously seen addresses which is used to access the second cache where the addresses are saved. Hybrid predictors [53, 54] combine last value stride and context predictors and implement a chooser which tries to determine which of the implemented predictors should give the final prediction for a given load.

A special context predictor named correlated predictor [55] tries to predict addresses which are not easily predictable by last value or stride predictors. The correlated address predictor splits the upper and lower part of each data address. Upper bits are stored in an IP-indexed table (a table of so-called *base* addresses), while the lower bits of several data addresses corresponding to the same *base* address are saved in the second table. A hashed history of data addresses, for each *base* addresses, is used to index into the second table. The lower data address bits change more frequently, and reducing their storage requirements (32 to 8 bits in this case) allows for more information to be added to a cache.

The compression of data addresses using address predictors is not a developed field of research. One of the existing mechanisms, data address stride cache [56], focuses on IP addressed, tagged and non tagged, caches of acceptable sizes. The technique achieves good compression rates but only for programs with a significant amount of data addresses which appear in strides.

For software based compression, the VPC mechanism [57, 58], combinations of various predictors each covering different behaviors of data addresses, yields excellent compression ratios that surpass the ones achieved by a general purposes compressor. However, this approach is not suitable for hardware implementation as it is not scalable to the point where implementable cache structures can perform any good.

3.2.3 Data Value Prediction

Mechanisms for compression of results of memory referencing instructions often compress load values only. The load values are sufficient to replay the program in a software debugger which employs an instruction set simulator while requiring less design effort and hardware support. Fully functional simulators are often available, as they are also used for system instrumentation, profiling and optimization purposes, and require modest changes to include support to obtain information from traces.

Compression of load value traces is not a developed research field. Load values are often considered as any other, random, data set which presumably requires a very complex and expensive compressor. However, certain types of redundancies exist and are typical for load value traces only. For example, the Value-Centric Data Cache [59] focuses on statistically proven frequent values within the data set. It shows that certain small values, such as -1, 0, 1, 4 …, are very frequently used in programs. An efficient encoding achieves compression by encoding small values in the predetermined number of bits (e.g., 4, 6, 8).

Value predictors (for load values). The research in compression of load values benefits from the research on load value predictors [47]. Value predictors have similarity with the techniques used in the processor's pipeline for efficient data prefetching and early load address calculation described in the previous section. Load value predictors bring the whole prediction concept to another level and try to speculate on the value of addressed memory location. If successful, the program execution can continue without waiting for the time consuming memory access. The speculatively executed instructions are confirmed once the predicted value is verified by the fetched one.

The driving force behind research in value predictors is the concept of *value locality*. The concept of value locality is introduced in [48]. The value locality describes the likelihood a load instruction will load a value that was recently loaded by the same instruction. The value locality exists because of various reasons and several are mentioned here: a) data redundancies, which exist due to small variations in input data set, b) constant values, which are usually loaded from memory as opposed to encoding them into the instruction word, c) branches, which are frequent instruction, often load very predictable branch target address information, d) data address operands (registers used to calculate data addresses), which are loaded from memory, often exhibit the same strong spatial and temporal locality as the data addresses, e) register spill, which occurs when a compiler runs out of registers, causes repeated reloading of the same value. According to an analysis in [48], up to 50% of load instructions in SPEC92 programs load the same value (unbound LV*1* predictor) or up to 80% when the instruction loads one of the last 16 loaded values (unbound LV*16* predictor).

Value predictors are proven successful in the software compression of load values. For example, the VPC3 mechanism [58] which employs 17 different value predictors with overall memory requirements of 27MB achieves a compression ratio of 1.7*x* more than the *Sequitur*, the best performing general purpose mechanism analyzed.

3.3 Compression Algorithms in Hardware

Very few compression algorithms are proposed for compression of either instruction or data traces. A hardware implementation of LZ77 pattern match algorithm [60], specialized for the compression of branch addresses and their targets, achieves relatively good compression rates under the enormous hardware cost. For example, a configuration with 50,000 logic gates requires a trace port bandwidth of 0.45 bits/ins for MiBench [61] programs. The mechanism suffers from the lack of any encoding of (*distance, length*) pairs (see Section 3.1.1 on *gzip*) as it would require even more costly implementation.

Stream cache (SC) is a mechanism that uses cache structures to store recently executed streams. Stream cache achieves very high compression ratios. The indices from the cache (information about the SC way and set where the hit occurred) are further grouped into tuples of 8 and stored in a buffer (the buffer keeps several groups of tuples). Each time a new tuple is found in the tuple buffer, an index in the buffer is output, effectively replacing 8 stream descriptors. This mechanism suffers from relatively low hit rates in the tuple history buffer which also requires expensive CAM memory.

Data addresses are not a focus of research in trace compression due to their smaller significance when compared to both instruction trace and data value trace and because of the lack of a practical need for unobtrusive tracing of data addresses in the past. In single-core systems, real-time tracing of data addresses was not important, as the memory access patterns, needed to simulate a program and optimize the memory, can be extracted using functional simulators. Furthermore, in multi-core general purpose systems, actual real-time unobtrusive tracing is usually not required. However, in the rising field of multi-core embedded real-time systems, data address tracing is crucial to observe correlations between cores and their communication with the shared memory.

Value predictors are proven to increase the processors performance, but due to their hardware constraints, the use of value predictors as the compressor's structures is very limited. The differentiating point between value predictors (either data address or load value) used for performance improvements and for the trace compression is the usable hit rate or the prediction rate. For example, the maximum achievable predictability for the data address for the SPEC95 test suite is between 21% and 79% using the last value address predictor and 3% and 54% using the stride predictor [62].

Average correct prediction rates are 40% for data addresses reported in [55] for the last value address predictor and 14% for the stride predictor. For prediction of load values, a theoretical limit for a LV*4* predictor is less than 70% for the SPEC95 test suite [48]. A realistic, and improved LV*4* load value predictor with 21KB of state achieves a prediction rate of 32% for the SPEC95 test suite [49]. Reported prediction rates improve a processor's performance to the point where the use of value and address predictors is justified. However, these prediction rates are very low concerning the compression. For example, a prediction rate of approximately 50% yields very low maximum compression ratios of only 2*x*. As mentioned earlier, VPC compressors [57, 58], a combination of different predictors with very large structures, can achieve high compression rates, but they are not suitable for hardware implementation.

3.3.1 Architectural Support for Trace Compression

Program traces have a unique feature that they can often be inferred by a software debugger augmented by an architectural functional simulator. The results of any instruction during program replay can be reconstructed as long as the target platform provides load value traces. This approach can be taken one step further where the functional simulator also simulates the state of memory. In that case, the software debugger needs to know only the values which enter the memory for the first time. From that point, any processor operation which stores and loads the known memory location can be reconstructed by the software debugger. One such a scheme is implemented in BugNet [27] (see Section 2.3.2), a debugging and tracing infrastructure for multi-core systems. A similar mechanism is implemented in a Flight Data Recorder (FDR), a multi-processor debugging and tracing infrastructure [63], which logs only the results of store instructions. The FDR system tracks memory stores originating from the DMA and I/O devices, which bring the new, and unknown, data to the system. Thus, any load instruction that a processor issues, loads a known value from the memory.

The techniques described require an instruction set simulator at the software debugger side and the architectural extensions on the target platform so that the implemented tracing mechanism knows whether to trace results of memory referencing instruction or not.

Both FDR and BugNet focus on general purpose systems built by one vendor. These mechanisms include the support for tracing in the L1 cache, L2 cache and the main memory, which is often not feasible in embedded and application specific systems where the whole chip is built out of components provided by different vendors. However, these mechanisms are important in this dissertation as they can be adapted for different purposes, including the required compression of data traces in embedded systems.

CHAPTER 4

COMPRESSION OF PROGRAM EXECUTION TRACES

In this chapter we describe the proposed mechanism for program execution tracing and introduce three algorithms and the corresponding hardware structures for compression of program execution traces. The chapter is organized as follows. Section 4.1 describes a system view of the proposed tracing mechanism. Section 4.2 discusses the program execution trace characteristics and different approaches for capturing program flow changes. In the next three sections we introduce three program execution trace compression techniques, namely, Double Move To Front (DMTF) (Section 4.3), Stream Descriptor Cache and Last Stream Predictor (SDC-LSP) (Section 4.4), and Branch Predictor Based Trace Compressor (Section 4.5). Section 4.6 gives the results of a comparative analysis of the proposed compression techniques and several recent academic proposals from the open literature.

4.1 System View of the Proposed Tracing Mechanism

Program execution traces are created by recording the addresses of executed instructions. However, to replay a program flow offline, we do not need to record each instruction address. Instead, we only have to record program flow changes, which can be caused by either control-flow instructions or exceptions. When a change in the program flow occurs, we need to know the address of the next instruction in the sequence; it is either the target address of the current branch instruction or the starting address of an exception handler.

Figure 4.1 shows a system view of the proposed tracing mechanism. The target platform encompasses a CPU core running a program and a trace module capturing the changes in the program control flow. Therefore, the trace module is coupled with the CPU core through a simple interface that consists of the program counter, branch type information (direct/indirect, conditional/unconditional), and an exception control signal. The trace module consists of relatively simple hardware structures and logic dedicated to capturing and compressing program execution traces. A trace encoding block receives a set of events from the trace compressor structures and encodes them in such a way to reduce the size of trace records. The trace records are stored into a trace buffer. The trace buffer serves to store trace records for significant portions of the traced program, or to amortize sudden bursts of trace records in case of real-time tracing. The recorded trace is read out of the chip through a trace port. The trace records can be collected on an external trace unit for later analysis or forwarded to a host machine running a software debugger.

The software debugger, running on a host machine, reads, decodes, and decompresses the trace records. To decompress the trace records, the debugger maintains exact software copies of the state in the trace module compressor structures. They are updated during program replay by emulating the operation of the hardware trace module.

Decompression produces a sequence of control flow descriptors that, in conjunction with the program binary, provide enough information for a complete program replay off-line.

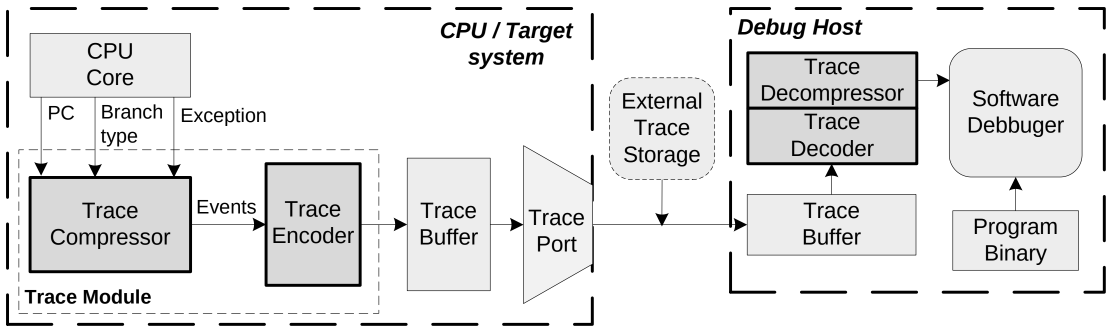


Figure 4.1 System view of the proposed program tracing and replay mechanism

The goal of the trace module designer is to minimize the size of the trace buffer and to minimize trace port bandwidth requirements. Thus, as a measure of performance of compression techniques, we use the average number of bits emitted per instruction on the trace port, which is equivalent to 32/(Compression Ratio), assuming 4-byte addresses. The compression ratio (CR) is defined as the ratio of the raw instruction address trace size, calculated as the number of instructions multiplied by the address size, and the size of the total compressor output during program execution. To summarize trace port bandwidth for multiple benchmarks, we use a weighted average mean. A benchmark weight is directly proportional to the number of instructions executed in the benchmark.

4.2 Characterization of Control Flow Changes

Capturing program flow changes can be done using stream (dynamic basic block) descriptors or by recording branch/target pairs. Both techniques are used to describe the very same changes in the control flow but in a different way. An instruction stream is a sequential run of instructions, from the target of a taken branch to the first taken branch in the sequence. Each instruction stream can be uniquely represented by its starting address (SA) and its length (SL). Thus, the complete trace of instruction addresses from an instruction stream can be replaced by the corresponding stream descriptor, i.e., the (SA, SL) pair.

A sequence of stream descriptors is sufficient to enable program reply off-line. However, the sequence of stream descriptors (SA, SL) includes some redundant information that can be omitted in order to reduce trace port bandwidth or storage requirements. If a control flow is changed as a result of an unconditional direct branch (target encoded within the instruction word), we do not need to end the current instruction stream. This is because the software debugger is able to follow the program execution across the control flow changes in case of unconditional direct branches. This modification reduces the number of program streams and thus the number of trace records that needs to be sent out through the trace port. In addition, when an instruction stream starts with the target of a direct branch, the SA field does not have to be traced out because it can be inferred by the software debugger. Figure 4.2 shows the distribution of branch types in the MiBench benchmark suite [61], collected using SimpleScalar [64] while running ARM binaries. The results indicate that only a small fraction of control flow changes require the SA field to be traced: on average only 1% of the total number of instructions are indirect taken branches.

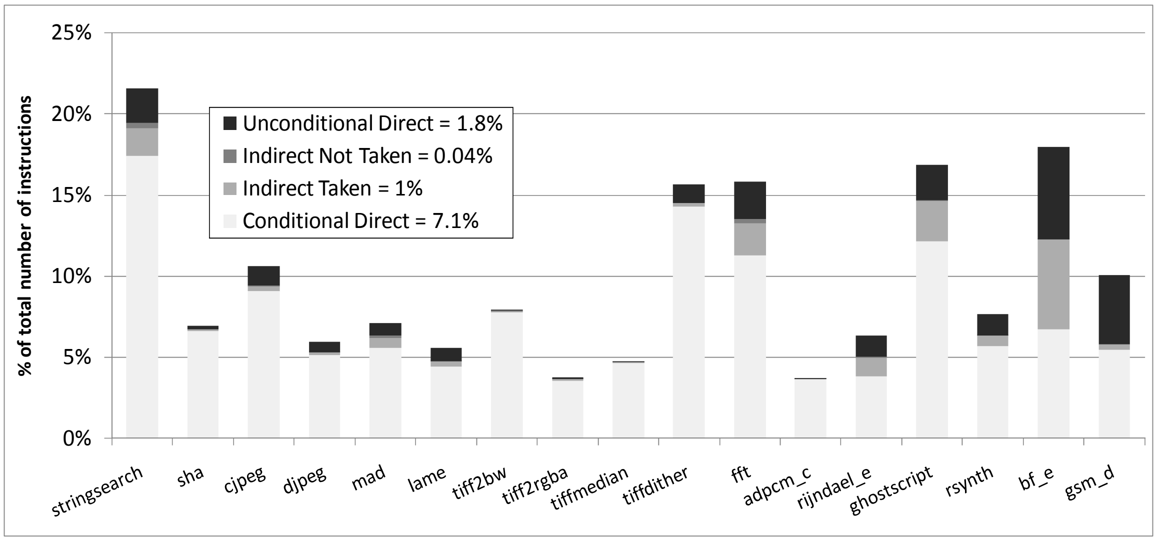


Figure 4.2 MiBench branch related statistics

Figure 4.3 gives a sequence of steps carried out by a stream detector logic that captures instruction streams. The stream detector tracks the current program execution by monitoring the program counter and control signals coming from the CPU core. For each new instruction, the SL register is incremented. The new stream signal is asserted when one of the following conditions is met: (a) the processor executes a control-flow instruction of a particular type, namely, direct conditional, indirect conditional, indirect unconditional, or return; or (b) an exception signal is asserted causing the program flow to depart from sequential execution. After forwarding the stream descriptor into the trace compressor, the stream detector prepares itself for the beginning of a new program stream by recording the starting address in the SA register and zeroing out the SL register.

```
if (NewStream) {
     SA = PC; SL = 0; NewStream = False;
}
if ((not ControlFlowChange)or(ControlFlowChange &&
(BranchType==DirectUncond)){
     SL++;
     if (SL == MaxSL) {
         Terminate Stream;
         Send (SA, SL) to the trace compressor;
         NewStream = True;
     }
} else {
     SL++;
     Terminate Stream and place (SA, SL) to the Stream compressor;
     NewStream = True;
}
```

Figure 4.3 Stream Detector operation

Most programs have only a small number of unique program streams, with just a fraction of them responsible for majority of program execution. Table 4.1 shows some important characteristics of MiBench. The columns *(a-d)* show the number of executed instructions (in millions), the number of unique streams, and the maximum and average stream length, respectively. The number of unique streams ranges from 341 to 6871, and the average dynamic stream length is between 5.9 (*bf_e*) and 54.7 (*adpcm_c*) instructions. The fifth column *(e)* shows the number of unique program streams that constitute 90% of dynamically executed streams. This number ranges between 1 and 235, and it is 78 on average. Note: all calculations assume weighted average, where weights are determined based on the number of executed instructions. The maximum stream length never exceeds 256, thus we choose to use 8 bits to represent SL. In addition to this, it can be shown that these frequently executed program streams create repeating patterns with strong local correlation.

Table 4.1 MiBench benchmark characteristics
[Legend: IC – Instruction Count, SC – Stream Count, SL – Stream Length
CDF – Cumulative Distribution Function]

Test	IC (mil.)	SC	Max SL	Avg SL	CDF 90%
adpcm_c	733	341	71	54.7	1
bf_e	544	403	70	5.9	22
cjpeg	105	1590	239	12.3	47
djpeg	23	1261	206	25.1	31
fft	631	846	94	10.5	209
ghostscript	708	6871	251	10.0	67
gsm_d	1299	711	165	19.5	33
lame	1285	3229	237	32.4	235
mad	287	1528	206	20.7	42
rijndael_e	320	513	77	21.0	45
rsynth	825	1238	180	17.6	49
stringsearch	4	436	65	6.0	48
sha	141	519	65	15.4	10
tiff2bw	143	1038	43	12.8	2
tiff2rgba	152	1131	75	27.7	2
tiffmedian	541	1335	92	22.3	5
tiffdither	833	1777	67	14.3	63
Average	**816**	**1791**	**145**	**21.6**	**77.8**
	(a)	***(b)***	***(c)***	***(d)***	***(e)***

4.3 Double Move-To-Front Method

In this section we discuss a method for program execution trace compression called the Double Move to Front Method (DMTF). First we describe the Move-to-Front transformation used in general-purpose compression algorithms (Section 4.3.1). Then, we introduce the proposed DMTF method (Section 4.3.2) and several enhancements designed to increase its effectiveness and/or reduce its implementation complexity (Section 4.3.3). Finally, we discuss the results of our trace port bandwidth analysis (Section 4.3.4) and the results of our hardware implementation cost analysis (Section 4.3.5).

4.3.1 Move-to-Front Transformation

Move-to-Front (MTF) [65] is an encoding of data designed to reduce the entropy of symbols in a data message by exploiting the local correlation between symbols. It is used in conjunction with the Burrows-Wheeler transform in the *bzip2* utility program [31]. The MTF algorithm encodes an input data message as follows. If an incoming input symbol is found in a history table *ht*, it is replaced with its index *i* in the *ht*, and the symbol is moved at the top of the table (the entry with index 0). The *ht* is updated by shifting down first *i-1* entries by one position, such that *ht*[*i*]=*ht*[*i*-1], ..., *ht*[1]=*ht*[0]. To illustrate the MTF operation, let us consider an input message *AABC*, and a history table *ht*=[<u>*C*</u>, *B*, *A*] (symbol *C* is at the position 0). The MTF transforms the 3-symbol input message into a new 2-symbol message *2022*.

The original MTF transformation can be easily extended to allow operation starting from an empty history table. The history table is searched for an incoming input symbol. If the symbol is not found in the table (we call this event an *ht* miss), the original symbol is output and the table is updated by shifting its content by one position and by placing the incoming symbol in the *ht*[0]. If the symbol is found in the history table (an *ht* hit event), its index is output and the table is updated as described above. The MTF allows for an effective encoding of frequently executed program sections. Let us consider several typical examples of program loops where symbols A, B, and C represent unique instruction streams characterized by their respective (SA, SL) pairs, for example, a program loop consisting of a sequence of two streams A and B repeating many times, illustrated as {AB}, is transformed into a hit pattern {11}; similarly, a loop with a repeating pattern {ABC} is transformed into {222}.

A relatively small history table will suffice to achieve a good hit rate due to a strong temporal locality of instruction streams in common programs. When a stream descriptor (SA, SL) is found in the history table, it is replaced with its index in the history table. Otherwise, the full stream descriptor of 40 bits is output in case the SA is a target of an indirect branch. If the SA is a target of a direct branch, it can be inferred from the program binary, and we output only 8 bits for SL. The effectiveness of the MTF transformation on program execution traces consisting of a sequence of stream descriptors is shown in Table 4.2*a*. We measure the frequency of the output symbols after the MTF transformation is applied. The average number of unique MTF output symbols that constitute 90% of all dynamically executed program streams is only 14.5, down from 78 before the MTF transformation in Table 4.1*e*, ranging from 1 (*adpcm_c*) to 38 (*stringsearch*). Note: the experiments are conducted assuming a history table with 128 entries; the hit rate is over 97%, so a very small number of streams are not transformed with the MTF.

A perfect trace compression without stream pattern recognition would replace each stream with just a single bit. As described above, the MTF transformation significantly reduces the number of trace symbols. To come close to a one bit per stream goal, we need to identify the most frequent entry and encode it with a single bit. Table 4.2*b* shows the percentage of the hit events in the most frequent *ht* entry. Although this percentage is fairly high for many benchmarks (e.g., *adpcm_c*, *tiff2bw*), it is relatively modest for others (e.g, 17% for *fft*, and 46% on average across all benchmarks). An additional problem is how to identify the most frequent entry in the *ht* because it varies across benchmarks.

Table 4.2 Move-to-Front transformation effect on program streams in MiBench

Test	*ht CDF 90%*	*ht[most freq.] HR*	*ht2[0] HR*
adpcm_c	1	0.99	1.00
bf_e	5	0.41	0.69
cjpeg	11	0.46	0.90
djpeg	11	0.69	0.87
fft	32	0.17	0.66
ghostscript	22	0.20	0.76
gsm_d	6	0.48	0.90
lame	21	0.23	0.74
mad	25	0.30	0.75
rijndael_e	2	0.57	0.79
rsynth	10	0.26	0.77
stringsearch	38	0.62	0.81
sha	2	0.86	0.92
tiff2bw	1	0.97	0.99
tiff2rgba	1	0.92	0.99
tiffmedian	1	0.91	0.97
tiffdither	38	0.45	0.78
Average	**14.5**	**0.46**	**0.82**
	(a)	***(b)***	***(c)***

In order to resolve these two problems, we introduce an additional, second level move-to-front transformation. Let us consider the following repeating stream pattern {ABAC}. The hit pattern at the output from the first-level MTF is {1212}. If we supply this pattern to the second-level MTF history table, the hit pattern at the output is {1111}, with even lower entropy of information. Because the MTF transformation lowers the number of frequent symbols, the level 2 history table can be significantly smaller.

This approach can be further extended by introducing another level of MTF transformation; in general, we could introduce a hierarchy of MTF history tables. The size of the MTF history tables will exponentially decrease as we move toward the upper levels. However, an increase in the number of MTF levels will reach the point of diminishing returns, and will not yield expected gains. In general, the optimal configuration is application specific. Our analysis shows that a 2-level MTF configuration appears to be optimal. Table 4.2*c* shows a high percentage of program streams that end up in the entry '0' of the level 2 history table (*ht2*), from 66% to 100%. By encoding this entry with a single bit, we approach the goal of having one bit per instruction stream. Note: the experiments are conducted using *ht2* with 16 entries achieving 94% hit rate.

4.3.2 DMTF Method

The analysis from the previous section suggests the use of a 2-level move-to-front transformation as optimal in compressing program instruction traces. Consequently, we design an instruction trace compressor with two history tables in sequence. We name this scheme Double Move-to-Front (DMTF). The first- and the second-level history tables are named *mtf1* and *mtf2*, respectively.

Figure 4.4 illustrates the operation of the proposed trace compressor. When a new stream is detected, its descriptor (SA, SL) is forwarded to *mtf1*. As described before, the *mtf1* table is searched for the stream descriptor. If we find a match, we have an *mtf1* hit; the index of the matching entry is output to the next stage, and *mtf1* is updated accordingly. Otherwise, we have an *mtf1* miss; the *mtf1* content is shifted down by one position, and *mtf1[0]* is loaded with the stream descriptor. In the case of an *mtf1* hit, the index *i1* is sent to the *mtf2* history table; *mtf2* is searched for the index *i1*. If we find a match in the entry 0, *mtf2*[0], we have an *mtf2* zero entry hit. If we find a match in the remaining *mtf2* entries, we have an *mtf2* non-zero entry hit. Otherwise, we have an *mtf2* miss event.

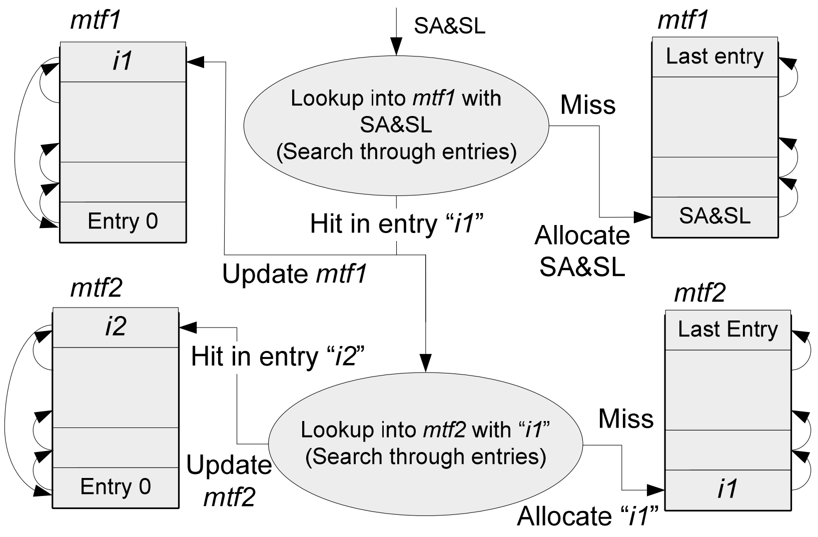


Figure 4.4 DMTF operation

We can distinguish four different events in the DMTF scheme, and they are encoded as follows. An *mtf2[0]* hit is encoded with a single bit '0'. An *mtf2* non-zero entry hit is encoded with a one-bit header '1' and the *mtf2* index *i2* ('1'+*i2*). An *mtf1* hit with a miss in *mtf2* is encoded with ('1'+i2miss+*i1*); note that the last index in the *mtf2* table, *i2miss*, is reserved to indicate a miss event in the *mtf2*. Finally, a miss in *mtf1* is encoded with a header ('1'+i2miss+i1miss) followed by a full or a partial stream descriptor ([SA], SL) – 40 or 8 bits. Note: the last index in the *mtf1* table, i1miss, is reserved to indicate a miss in the *mtf1*.

The compression ratio that can be achieved using the DMTF scheme can be expressed analytically as follows. Equation (4.1) shows the number of bits needed to encode a single stream after DMTF compression, as a function of five parameters: *mtf2* zero-entry hit rate, *mtf2.zhr*; *mtf2* non-zero entry hit rate, *mtf2.ohr*; *mtf1* hit rate, *mtf1.hr*; *mtf1* size, *mtf1.size*; and *mtf2* size, *mtf2.size*. Note: *mtf2.hr=mtf2.zhr* + *mtf2.ohr*. Equation (4.2) shows the compression ratio as a function of the number of instructions in a program, the number of executed instruction streams, and the number of bits per one instruction stream.

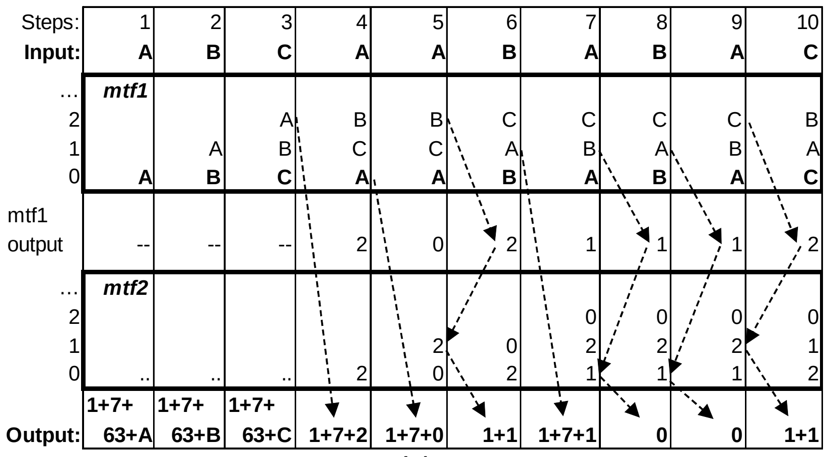
$$\begin{aligned} BitsPerStream = {} & mtf2.zhr + \\ & (mtf2.ohr * (1 + \log_2(mtf2.size) + \\ & (mtf1.hr - mtf2.hr) * (1 + \log_2 mtf2.size + \log_2 mtf1.size) + \\ & (1 - mtf1.hr) * (1 + \log_2 mtf2.size + \log_2 mtf1.size + 8 + [32]) \end{aligned} \tag{4.1}$$

$$CR = \frac{32 * InstructionCount / StreamCount}{BitsPerStream} \tag{4.2}$$

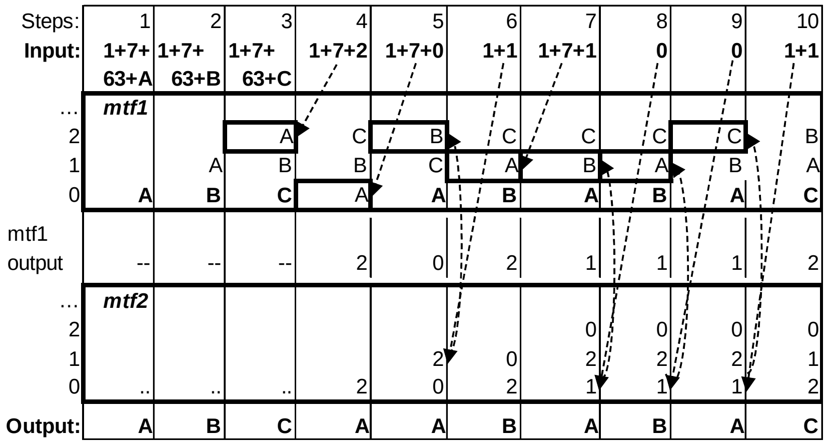
An Example Compression/Decompression. Let us illustrate the compression flow using an example from Figure 4.5*a*. We consider the following sequence of instruction streams ABCAABABAC, where A, B, and C denote 3 instruction streams with distinct stream descriptors. Let us assume a 64-entry *mtf1* and an 8-entry *mtf2*.

Note that the actual number of entries is 63 and 7, respectively, since the last indices are reserved to indicate miss events. The first three instruction streams are not found in the *mtf1* and are output with the header '1', followed by a 3-bit index in the *mtf2* reserved for miss events ('111'), a 6-bit index in the *mtf1* reserved for miss events ('111111'), and individual stream descriptors ([SA], SL). Next, the stream A is found in *mtf1[2]*, but index 2 is not found in the *mtf2* resulting in an *mtf2* miss with *mtf1* hit event; we emit a header '1'+'111' followed by the mtf1 index '000010'. The next stream in sequence is A, resulting again in an *mtf2* miss with *mtf1* hit event; this event is encoded with '1+'111'+'000000'. The rest of the compression flow continues as illustrated in Figure 4.5*a.*

The decompression flow is a reversed compression flow, and it requires the same configuration of the history tables. The compressed trace is read, headers are analyzed and the history tables updated according to the DMTF method described above. The decompression flow is illustrated in Figure 4.5*b*. The first item starts with the header '1'+'111'+'111111', which indicates that the next 40 bits represent the first stream descriptor (SA, SL). The stream descriptor is loaded into *mtf1[0]*. The de-compressor now can recreate a complete instruction trace for this stream. The next two items in the trace are decompressed in the same way. The next trace record '1+'111'+'000010' directs the de-compressor to find the original stream descriptor in *mtf1[2]* (instruction stream A). The following trace record '1'+'111'+'000000' directs the de-compressor to find the stream in *mtf1[0]*, which is stream A. The rest of the decompression process is illustrated in Figure 4.5*b.*


Steps: 1 2 3 4 5 6 7 8 9 10
Input: A B C A A B A B A C
mtf1
2 A B B C C C C B
1 A B C C A B A B A
0 A B C A A B A B A C
mtf1 output -- -- -- 2 0 2 1 1 1 2
mtf2
2 0 0 0 0
1 2 0 2 2 2 1
0 .. .. .. 2 0 2 1 1 1 2
Output: 1+7+63+A 1+7+63+B 1+7+63+C 1+7+2 1+7+0 1+1 1+7+1 0 0 1+1

(a)


Steps: 1 2 3 4 5 6 7 8 9 10
Input: 1+7+63+A 1+7+63+B 1+7+63+C 1+7+2 1+7+0 1+1 1+7+1 0 0 1+1
mtf1
2 A C B C C C C B
1 A B B C A B A B A
0 A B C A A B A B A C
mtf1 output -- -- -- 2 0 2 1 1 1 2
mtf2
2 0 0 0 0
1 2 0 2 2 2 1
0 .. .. .. 2 0 2 1 1 1 2
Output: A B C A A B A B A C

(b)

Figure 4.5 DMTF compression (a) and decompression (b) flow examples

Performance Analysis. Table 4.3*a* shows the trace port bandwidth for MiBench benchmarks; the size of the *mtf1* is fixed to 128 entries and the *mtf2* size is varied from 4 to 16 entries. The last row (*Average*) shows the total bits per instruction calculated as the weighted mean of individual program results. The results show that we are able to achieve an excellent compression with bandwidth ranging from 0.714 to 0.018. The results also indicate that a DMTF configuration with only 4-entry *mtf2* will outperform configurations with larger *mtf2*.

Table 4.3 Trace bandwidth for DMTF(128,X), X= 4-16 (a), distribution of the individual trace components (b)

	bits/ins (mtf1 size = 128)			***Distribution per component for DMTF(128,4)***			
Test \ mtf2 size	***4***	***8***	***16***	***zht***	***mtf2ht***	***mtf1ht***	***mtf1mt***
adpcm_c	0.018	0.018	0.019	99%	1%	0%	0%
bf_e	0.292	0.340	0.380	40%	52%	8%	0%
cjpeg	0.128	0.127	0.127	57%	10%	31%	2%
djpeg	0.074	0.074	0.074	47%	12%	30%	11%
fft	0.714	0.734	0.749	9%	3%	10%	78%
ghostscript	0.299	0.301	0.299	25%	7%	55%	13%
gsm_d	0.089	0.093	0.098	52%	13%	4%	32%
lame	0.098	0.097	0.099	23%	12%	28%	37%
mad	0.153	0.153	0.151	24%	8%	48%	20%
rijndael_e	0.099	0.078	0.088	38%	17%	45%	0%
rsynth	0.153	0.161	0.172	29%	18%	13%	41%
stringsearch	0.394	0.411	0.426	34%	7%	58%	2%
sha	0.075	0.080	0.085	79%	20%	0%	0%
tiff2bw	0.082	0.082	0.082	95%	2%	3%	0%
tiff2rgba	0.038	0.038	0.038	95%	3%	0%	1%
tiffmedian	0.061	0.062	0.063	71%	5%	2%	22%
tiffdither	0.189	0.199	0.207	29%	9%	44%	18%
Average	**0.176**	**0.183**	**0.190**	**42.8%**	**12.3%**	**20.0%**	**25.0%**
		(a)			***(b)***		

4.3.3 Enhanced DMTF Method

The output of the DMTF trace compressor contains a lot of redundant information. We introduce two low-cost enhancements that exploit this redundancy and/or reduce the complexity of the compressor implementation. The four components of the output trace, *mtf2* zero hit trace (*zht*), *mtf2* non-zero hit trace (*mtf2ht*), *mtf2* miss with *mtf1* hit trace (*mtf1ht*), and *mtf1* miss trace (*mtf1mt*) are analyzed separately. Table 4.3*b* shows the contribution of each component to the total trace size for DMTF(128,4) (128-entry *mtf1* and 4-entry *mtf2*). The results show the *mtf1mt* component is responsible for 25% of the total size, in spite of high hit rates in the *mtf1*.

Fortunately, the redundant information in this trace can be easily exploited using a simple last-value predictor on upper address bits that stay constant during program execution. Next, the zero trace occupies 43% of the total trace. We expect it to contain long runs of '0's, and its size can be reduced by replacing them by a counter.

Last-Value Predictor for Upper Address Bits. The upper address bits of the starting address (SA) field in the stream descriptor rarely change during program execution. We analyzed the locality of the stream starting addresses; the SA field of the incoming stream is compared bit by bit to the SA of the previous instruction stream or to SAs of the several last instruction streams. The results indicate that the upper 12 address bits, SA[31:20], stay constant during program execution in 99% of cases. Therefore, we divide the SA field into two parts: the lower 20 address bits SA[19:0] that are compressed through the DMTF, and the upper 12 bits that are handled using a simple last value predictor (*HLV*). Note: SA[1:0] is '00' for the ARM ISA and could be omitted; SA[0]= '0' for the ARM Thumb ISA, so the SA[0] could be omitted. Here we keep the whole address.

A 12-bit last value (LV) register keeps the upper 12 bits of the last stream's SA. The upper 12 address bits of an incoming stream are compared to the LV. If they match, we have an *HLV* hit. The lower 20 address bits SA[19:0] and SL are used in a *mtf1* lookup. We adopt a scheme where *mtf1* hits are conditional upon the corresponding *HLV* hits. An *HLV* miss will cause a miss trace record to be emitted regardless of *mtf1* hits. When we have an *HLV* hit with *mtf1* miss event, the upper address bits are not emitted (in case a full stream descriptor is required). Finally, in cases when both *mtf1* and *HLV* have a hit, a regular DMTF record is emitted. The miss trace format is consequently extended to support these modifications.

The effectiveness of this enhancement is analyzed below. In general, it is beneficial in DMTF configurations with a relatively small *mtf1* and lesser so with a larger *mtf1*. The performance benefits are somewhat limited because direct branches dominate in the MiBench suite (92% of all branches on average) and all stream descriptors that start with targets of direct branches do not require the SA field. However, it significantly reduces the complexity of the DMTF implementation, as we do not need to keep the upper 12 address bits in the *mtf1* history table.

Zero Hit Trace Counters. We now show that the DMTF method ensures that the *mtf2* zero hit event is the most frequent one, and thus it is encoded with a single bit '0'. In many benchmarks the output trace will consist of long runs of zeros. The redundancy in this trace can be exploited by utilizing a zero-length counter (ZLC for short); it counts the consecutive zeros and replaces them with a counter value preceded by a new header. The number of bits used to encode this trace component is determined by the counter size. A longer counter can capture longer runs of zeros, but a counter which is too long results in wasted bits. Our analysis of the *zht* trace component shows a fairly large variation in the average number of consecutive zeros, ranging from 5 in *ghostscript* and *fft* to hundreds in *adpcm_c* and *tiff2bw*. In addition, zero runs in a program may vary across different program phases. This implies that an adaptive ZLC length method would be optimal.

The adaptive zero-length counter (AZLC) tries to dynamically adjust the ZLC size to the program flow characteristics. An additional 4-bit saturating counter monitors the *zht* component, and it is updated as follows. It is incremented by 3 when the number of consecutive zeros in the trace (*mtf2*[0] hits) exceeds the current size of the ZLC. The monitoring counter is decremented by 1 when a detected zero sequence is smaller than the ZLC counter maximum value. When the monitoring counter reaches the maximum (15) or minimum (0) values, a change in the ZLC size occurs.

The AZLC requires a slight modification of the trace output format. A header bit '0' is followed by $\log_2$(AZLC Size) bits. The counter size is automatically adjusted as described above. The decompressor needs to implement the same adaptive algorithm.

Figure 4.6 shows a modified trace format that supports two enhancements, HLV and AZLC.

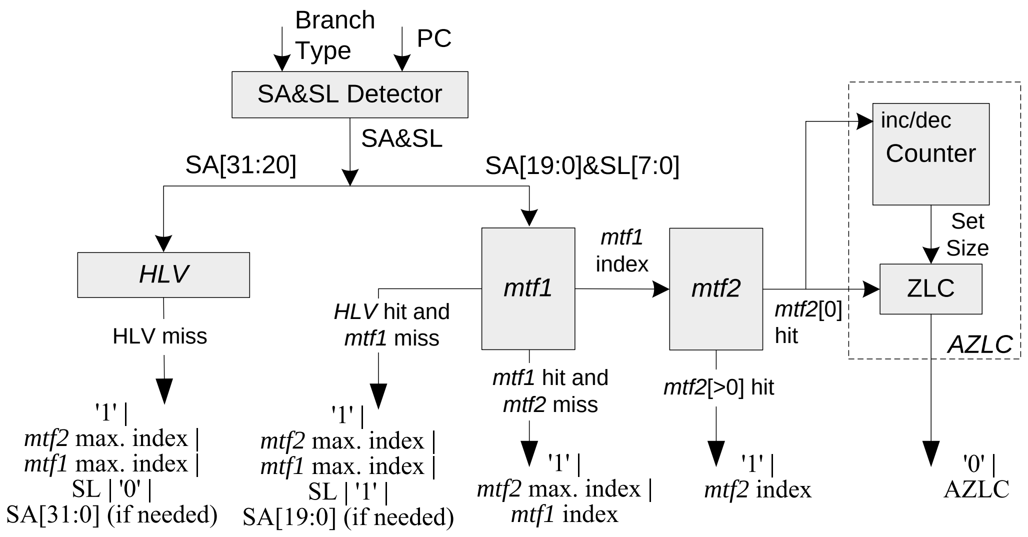


Figure 4.6 An enhanced DMTF trace format

4.3.4 Trace Port Bandwidth Analysis

Figure 4.7 shows the average trace port bandwidth (bits/ins) of several DMTF configurations as a function of the *mtf1* size (64-320). The basic DMTF (bDMTF) with *mtf2*=4 performs better than with *mtf2*=8 for any *mtf1* size as previously indicated in Figure 4.4. The DMTF with HLV predictor and *mtf2*=4 (hDMTF) performs better than bDMTF only for small *mtf1* sizes. When *mtf1*=192, bDMTF slightly outperforms hDMTH primarily due to significant performance degradation for the *lame* benchmark. Finally, the enhanced DMTF with *mtf2*=4 (eDMTF) with both improvements performs the best. For all configurations, the compression ratio saturates for the *mtf1* with 256 entries, and the *mtf1* with 192 entries strikes an optimal balance between the complexity and compression ratio. Figure 4.7 gives a design guideline and shows how one can trade compression ratio for complexity (the most complex resource in the enhanced DMTF module is the *mtf1* table).

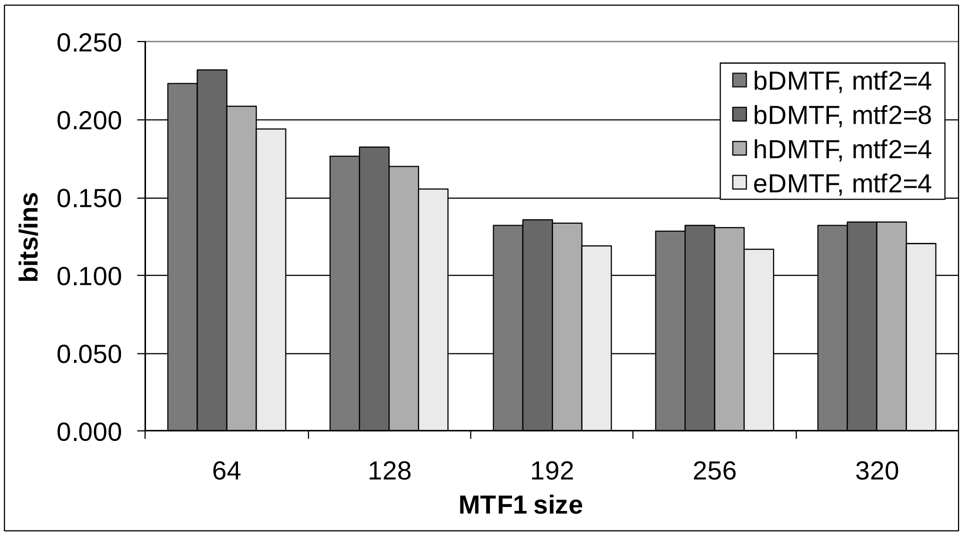


Figure 4.7 Trace port bandwidth as a function of the *mtf1* size

Table 4.4 Compression ratios for xDMTF (x=b,h,e)

	mtf1=192, mtf2=4				*mtf1=64, mtf2=4*			
	CR	***CR***	***CR***	***bits/ins.***	***CR***	***CR***	***CR***	***bits/ins***
Test	***bDMTF***	***hDMTF***	***eDMTF***	***eDMTF***	***bDMTF***	***hDMTF***	***eDMTF***	***eDMTF***
adpcm_c	1738	1738	29389	0.001	1738	1738	29441	0.001
bf_e	108.8	108.8	112.7	0.284	110.5	110.5	114.5	0.279
cjpeg	245.3	245.7	353.1	0.091	245.6	248.7	360.5	0.089
djpeg	448.9	451.6	612.9	0.052	421.9	433.6	582.4	0.055
fft	151.7	151.9	159.1	0.201	26.9	31.1	31.6	1.012
ghostscript	104.7	106.1	104.6	0.306	104.8	108.4	106.9	0.299
gsm_d	495.4	495.4	808.4	0.040	363.2	381.2	547.8	0.058
lame	317.1	274.2	283.2	0.113	318.6	279.7	289.8	0.110
mad	202.7	208.8	217.0	0.147	219.0	227.6	237.4	0.135
rijndael_e	308.9	308.9	333.5	0.096	333.7	334.5	363.4	0.088
rsynth	307.2	307.4	296.3	0.108	159.1	176.6	173.8	0.184
stringsearch	77.0	77.2	82.7	0.387	32.9	37.2	38.8	0.825
sha	424.9	425.1	654.3	0.049	425.1	425.2	654.7	0.049
tiff2bw	390.2	390.4	2807.6	0.011	221.3	241.0	521.8	0.061
tiff2rgba	852.5	853.8	5334.6	0.006	348.8	393.5	637.7	0.050
tiffmedian	653.1	653.4	2712.9	0.012	386.1	418.7	819.1	0.039
tiffdither	167.8	169.6	193.2	0.166	140.9	144.8	162.7	0.197
Average	**242.5**	**239.5**	**268.1**	**0.119**	**143.1**	**153.1**	**165.1**	**0.193**

Table 4.4 shows a detailed evaluation for bDMTF, hDMTF, and eDMTF with two configurations (192,4) and (64,4), x={b,h,e}. The hDMTF configuration achieves 7% higher compression ratio than bDMTF for *mtf1*=64. This improvement is due to reducing the size of the miss trace. For *mtf1*=192, we see a decrease in hDMTF performance over bDMTF as explained above. The eDMTF configuration achieves 15% higher CR over bDMTF for *mtf1*=64 and 11% for *mtf1*=192. This improvement is unevenly distributed over benchmarks and is useful for tests such as *adpcm*_c (17x) or *tiff2rgba* (6x). The best performing configuration (eDMTF with *mtf1*=192) achieves the total weighted average bandwidth on the trace port of only 0.12 bits per instruction.

4.3.5 Hardware Implementation and Complexity

The *mtf1* and *mtf2* history tables can be implemented as custom fully associative structures with a single-clock cycle lookup and additional hardware needed to support the move-to-front update operation. Instead, we propose a cost-effective implementation that combines a standard content addressable memory (CAM) and a most-recently used (MRU) stack (Figure 4.8). The MRU stack has the same number of entries as the history table, but its contents are indices in the CAM memory. Each MRU stack entry points to a particular CAM location, and thus has [$\log_2$(MTF_Size)] bits.

The *mtf* lookup operation encompasses a lookup into the CAM with (SA, SL) pair and a lookup into the MRU stack. In the case of a CAM hit, the corresponding CAM index is forwarded to the MRU stack and the MRU lookup is performed. The selected entry is moved to the top of the MRU stack, and the top *(i-1)* locations are shifted down. In the case of a CAM miss, the MRU stack provides the address of the CAM location where the new stream is going to be stored (the index at the bottom of the MRU stack), and the MRU stack is updated accordingly. Figure 4.8 shows a block diagram of a single level MTF history buffer. The lookup and update together require two processor clock cycles and are performed only when a new instruction stream is detected. Hence, the compression can be done at the full processor speed without ever slowing the processor.

To estimate the complexity of the proposed implementation, we consider enhanced DMTF(192,4) configuration. The *mtf1* CAM memory has 191 entries, each with 28 bits (20 for SA, and 8 for SL). With 3 gates per CAM bit [66], the CAM complexity is estimated at 3x28x191 ~ 16000 logic gates. The *mtf1* MRU stack has 191 8-bit entries, plus comparators attached to each of them. Registers use latches that occupy approximately 2.5 logic gates per bit, comparators use 2.5 logic gates per bit while a tri-state buffer uses 0.5 logic gates per bit. The *mtf1* MRU stack size is estimated at approximately 8400 gates. Similarly the *mtf2* size is estimated to be approximately 150 logic gates. Together with the LV predictor (12-bit register + comparator) and the AZLC counter (4 bits), the total complexity of the DMTF(192,4) is about 24,600 gates.

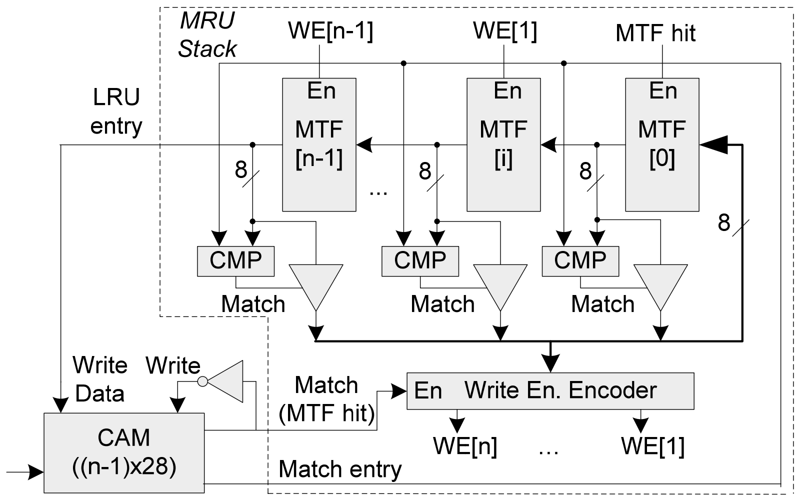


Figure 4.8 MTF hardware implementation

4.4 Stream Descriptor Cache and Last Stream Predictor

In this section we describe a trace compression method based on hardware structures called a stream descriptor cache (SDC) and a last stream predictor (LSP). First, we describe the trace compressor structure and compression algorithms (Section 4.4.1). Next, we explore the design space including the size and organization of the proposed compressor structures (Section 4.4.2). In Section 4.4.3 we describe several enhancements designed to further reduce the trace port bandwidth and reduce the complexity of the compressor structures. Finally, we give the results of the trace port bandwidth analysis in Section 4.4.4 and estimate hardware implementation cost in Section 4.4.5.

4.4.1 Compressor Organization

The proposed mechanism performs the compression of program execution traces in two stages. In the first stage, we perform a stream cache transformation using a structure called the stream descriptor cache (SDC). This transformation translates a stream descriptor into a stream index (SI). The stream index is used as an input to a predictor structure called a last stream predictor (LSP) in the second stage. Consequently, a sequence of stream descriptors coming from the stream detector is translated into a sequence of hit and miss events at the output of the SDC and LSP compressor structures. These events are efficiently encoded, thus significantly reducing the size of trace records that are stored in the trace buffer before they are sent to the trace port. A high level architecture of the compressor is given in Figure 4.9.

Stream descriptor cache. The stream cache is organized into N_{WAY} ways and N_{SET} sets as shown in Figure 4.10. An entry in the stream descriptor cache holds a complete stream descriptor (SA, SL).

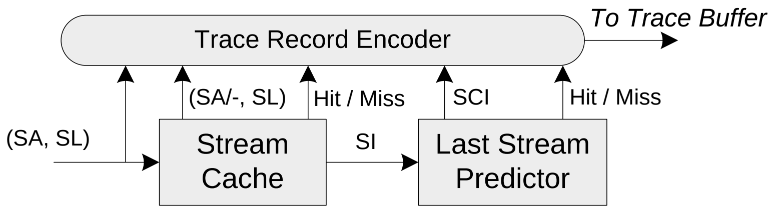


Figure 4.9 System view of Stream Cache with Last Stream Predictor compressor

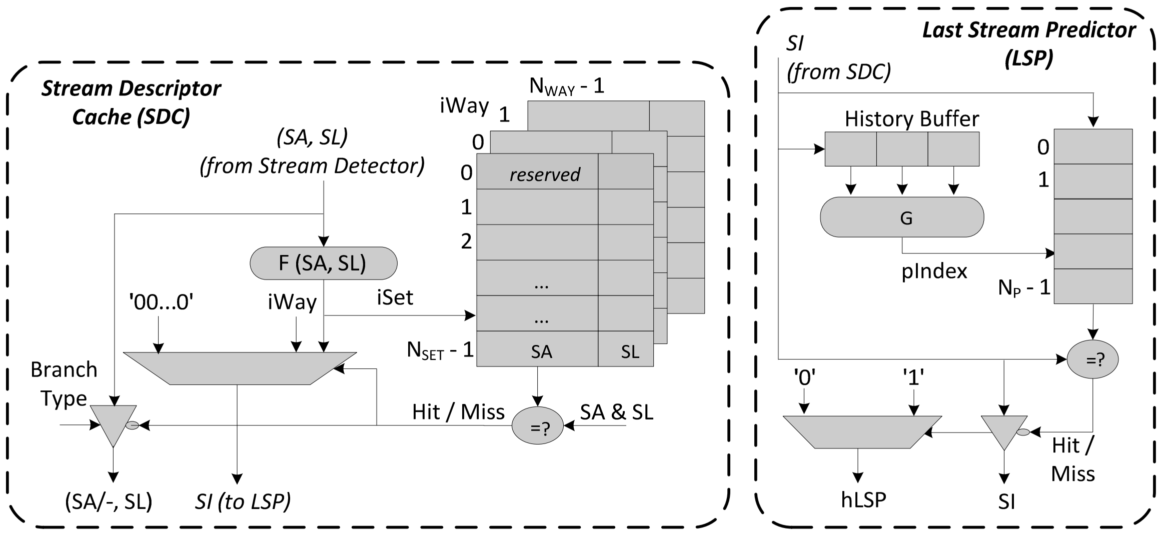


Figure 4.10 Trace module structures: Stream Descriptor Cache and Last Stream Predictor

Figure 4.11 describes the sequence of steps carried out during the stream cache transformation by the SDC controller. The next stream descriptor is received from the Stream Detector and an SDC lookup is performed. A set in the stream descriptor cache is calculated as a simple function of the stream descriptor, e.g., bit-wise XOR of selected bits from the SA and SL fields. If the incoming stream descriptor matches an entry in the selected set, we have an SDC hit event; otherwise, we have an SDC miss event. In the case of an SDC hit, the corresponding stream index, determined by concatenating the set and way indices (SI = {iSet, iWay}), is forwarded to the LSP. In the case of an SDC miss, the reserved index zero is forwarded (SI = 0). If all entries of the selected set are occupied, an entry is evicted based on the replacement policy (e.g., LRU, FIFO), and it is updated with the incoming stream descriptor.

The compression ratio achieved by our stream detector and stream detector cache, CR(SDC), is defined as the ratio of the raw instruction address trace size, calculated as the number of instructions multiplied by the address size, and the size of the SDC output (Equation (4.3)). It can be expressed analytically as a function of the average dynamic stream length (*avgSL*), the SDC hit rate (*hrSDC*), the SDC size ($N_{SET}*N_{WAY}$), and the probability that a stream starts with a target of an indirect branch (p_{IND}). For each program stream, $log_2(N_{SET}*N_{WAY})$ bits are emitted to the SI output. On each SDC miss, a 5-byte (SA, SL) or 1-byte (-, SL) stream descriptor is outputted, depending on whether the corresponding stream starts with the target of an indirect or direct branch, respectively. The parameters *avgSL* and p_{IND} are benchmark dependent and cannot be changed except maybe through program optimization – e.g., favoring longer streams using loop unrolling or trace scheduling. Smaller stream caches require shorter indices but likely have a lower hit rate, which negatively affects the compression ratio. Thus, a detailed exploration of the stream cache design space is necessary to determine a good hash function as well as a good stream cache size and organization (N_{SET} and N_{WAY}).

```
Get the next stream descriptor, (SA, SL), from the Stream Detector
Lookup in the stream descriptor cache with iSet = F(SA, SL);
if (SDC hit)
  SI = (iSet concatenate iWay);
else {
  SI = 0;
  if (SA is reached via an indirect branch)
      Prepare stream descriptor (SA, SL) for output;
  else
      Prepare stream descriptor (-, SL) for output;
  Select an entry (iWay) in the iSet set to be replaced;
  Update stream descriptor cache entry:
  SDC[iSet][iWay].Valid = 1;
  SDC[iSet][iWay].SA = SA,
  SDC[iSet][iWay].SL = SL;
}
Update replacement indicators in the selected set;
```

Figure 4.11 Stream Descriptor Cache operation

$$CR(SDC) = \frac{4 \cdot IC}{Size(SDC\ Output)} = \frac{4 \cdot avgSL}{0.125 \cdot \log_2(N_{SET} \cdot N_{WAYS}) + (1 - hrSDC) \cdot [p_{IND} \cdot 5 + (1 - p_{IND}) \cdot 1]} \tag{4.3}$$

Last Stream Predictor. The second stage uses a simple last value predictor as shown in Figure 4.10 and described in Figure 4.12, to exploit redundancy in the SI component caused by repeating sequences of stream indices. A linear predictor table with N_P entries is indexed by a hash function that is based on the history of previous stream indices. If the selected predictor entry matches the incoming stream index, we have an LSP hit. Otherwise, we have an LSP miss, and the selected predictor entry is updated with the incoming stream index. The hit/miss information (1 bit) and, in the case of an LSP miss, SI ($log_2(N_{SET}*N_{WAYS})$ bits) are forwarded to the trace encoder.

```
Get the incoming index, SI;
Calculate the LSP index: pIndex = G(indices in the History Buffer);
Perform lookup in the Last Stream Predictor with pIndex;
if(LSP[pIndex] == SI)
      Emit('1');
else {
      Emit('0' + SI);
      LSP[pIndex] = SI;
}
Shift SI into the History Buffer;
```

Figure 4.12 Last Stream Predictor operation

The compression ratio achievable by the LSP stage alone, CR(LSP), can be calculated as shown in Equation (4.4). It depends on the stream index size and the LSP hit rate, *hrLSP*. The maximum compression ratio that can be achieved in this stage is $log_2(N_{SET}*N_{WAY})$. The design space exploration for the last stream predictor includes determining a good hash function and a good number of entries in the predictor N_P.

$$CR(LSP) = \frac{Size(SI)}{Size(LSP\ \ Output)} = \frac{1}{(1-hrLSP)+1/\log_2(N_{SET}\cdot N_{WAYS})} \tag{4.4}$$

Trace Record Encoder. The trace encoder assembles output trace messages based on the events in SDC and LSP as shown in Table 4.5. We distinguish three combinations of events in the compressor structures: (a) an LSP hit with a SDC hit, (b) an LSP miss with an SDC hit, and (c) an LSP miss with an SDC miss. The LSP cannot hit if the SDC misses. In the case of an LSP hit with an SDC hit, the single-bit trace record ′1′ is placed into the trace buffer. In the case of an LSP miss with an SDC hit, the trace record starts with a ′0′ single-bit header and is followed by the value of the stream index that missed in the LSP. Finally, in the case of an LSP miss with an SDC miss, the trace record consists of a ′0′ single-bit header, followed by a zero stream index that indicates a miss in the SDC, and a 40-bit (SA, SL) or 8-bit (-, SL) stream descriptor, depending on the type of branch that led to the beginning of the stream.

Table 4.5 Trace records encoding

Event Encoded	*Trace Record*			*Trace Record Bit Width*
	H	*SI*	*Stream Descriptor*	
LSP miss, SDC miss (SA is the target of an indirect branch)	0	0	(SA, SL)	1+ $\log_2(N_{SET}*N_{WAY})$ + 40
LSP miss, SDC miss (SA is the target of a direct branch)	0	0	(-, SL)	1+ $\log_2(N_{SET}*N_{WAY})$ + 8
LSP miss, SDC hit	0	non-zero	-	1+ $\log_2(N_{SET}*N_{WAY})$
LSP hit, SDC hit	1	-	-	1

An Example Compression/Decompression. Code snippet in Figure 4.13 includes a simple loop executing 100 iterations. The loop body consists of only one instruction stream. When the first iteration completes on the target platform, the stream detector captures the stream descriptor (SA, SL) = (0x020001f4, 9). Let us assume a 64-entry 4-way associative stream descriptor cache (N_{SET}=16, N_{WAYS}=4). The SDC indexes are calculated as a function of certain bits of the stream descriptor; let us assume we calculate the *iSet* as follows: *iSet = SA[7:4] xor SL[3:0]*; in our case *iSet=0x6*. A lookup in the SDC set with index *iSet=0x6* results in an SDC miss because the SDC entries are initially invalid. The least recently used entry in the selected SDC set is updated by the stream descriptor information (let us assume it is *iWay*=0), and the reserved 6-bit index *0* is output to the next stage (SI=′000000′). A lookup in the LSP entry with index *pIndex =0* results in an LSP miss because the LSP entries are also initially invalid. The LSP entry with index *0* is updated with the SI value. A complete trace record for the first occurrence of the stream includes a header bit ′0′ followed by the 6-bit index ′000000′ and the 40-bit stream descriptor (47 bits in total; here we assume that we need to output the starting address of the stream in spite of the fact that it can be inferred from the program binary).

When we encounter the second iteration of the loop, the stream descriptor is found in the selected SDC set (an SDC hit). The SI is *iSet* concatenated with *iWay*, resulting in SI=′011000′ (*iSet*=′0110′ and *iWay*=′00′). If we assume that the LSP predictor access is solely based on the previous SI (0 in our case), we will have another LSP miss. A trace record ′0.011000′ (h=′0′, SI=′011000′) is output to the trace buffer, and LSP’s entry 0 is now updated with the value ′011000′. The third loop iteration results in an SDC hit, and SI=′011000′ is forwarded to the LSP stage. The LSP will again miss (the entry pointed to by the previous SI is not initialized yet), and another trace record ′0.011000′ is sent to the trace buffer. The LSP entry with index ′011000′ is updated with the value ′011000′. The fourth iteration hits in both the SDC and the LSP and only a single bit ′1′ is sent to the trace buffer. The next 95 iterations will also have only a single bit trace record to indicate both SDC and LSP hits. The final iteration does not hit because the loop end branch falls through and the stream length will therefore be larger than that of the previous streams.

```
// Code Snippet
28.    for(i=0; i<100; i++) {
29.            c[i] = s*a[i] + b[i];
30.            sum = sum + c[i];
31.    }
// Assembly listing of the code snippet for the ARM ISA
32.    @ 0x020001f4: mov  r1,r12, lsl #2
33.    @ 0x020001f8: ldr  r2,[r4, r1]
34.    @ 0x020001fc: ldr  r3,[r14, r1]
35.    @ 0x02000200: mla  r0,r2,r8,r3
36.    @ 0x02000204: add  r12,r12,#1 (1 >>> 0)
37.  @ 0x02000208: cmp  r12,#99 (99 >>> 0)
38.  @ 0x0200020c: add  r6,r6,r0
39.  @ 0x02000210: str  r0,[r5, r1]
40.  @ 0x02000214: ble  0x20001f4
// Trace records emitted per loop iteration
41.  h='0'; SI='000000'; (SA,SL)=(0x020001f4, 9)
42.  h='0'; SI='011000';
43.  h='0'; SI='011000';
44.  h='1';
45.  h='1';
46.  . . .
99.  h='1';
100. h='0'; SI='000000'; (SA,SL)=(0x020001f4, ?)
```

Figure 4.13 A compression/decompression example

The de-compressor on the debugger side reads the incoming bit stream from its trace buffer. The first bit in the trace is h=′0′, indicating an LSP miss event. The de-compressor then reads the next 6 bits from the traces that carry the SI=′000000′. This index is reserved to indicate an SDC miss, and the de-compressor reads the next 40 bits from the trace to obtain the stream descriptor. The debugger updates the software copies of the SDC and LSP accordingly and replays 9 instructions starting at address 0x0x020001f4. The next step is to read the next trace record. It also starts with h=′0′, indicating an LSP miss. The next 6 bits are non-zero, which means that we have an SDC hit. The debugger retrieves the next stream descriptor from the SDC's entry SI=′011000′ and updates the SDC and LSP structures accordingly. The second iteration of the loop is replayed. Similarly, the debugger replays the third loop iteration. The fourth trace record starts with a header h=′1′. This single-bit message is sufficient to replay the current stream. The software debugger retrieves the stream index from the LSP maintained in software (SI=′011000′) and, using this stream index, it retrieves the stream descriptor from the software copy of the stream cache. The debugger maintains its software copies of the compressor structures by updating the LSP's history buffer and SDC's replacement bits using the same policies as the hardware trace module does. The process continues until all iterations of the loop have been replayed.

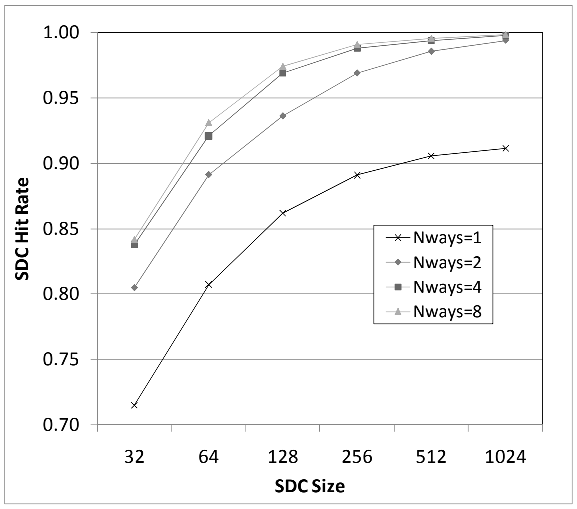
4.4.2 Design Space Exploration

The goal of this design space exploration is twofold. First, we explore the design space to find good parameters for the proposed compressor structures and access functions. As a measure of performance, we use the average number of bits emitted per instruction on the trace port. We also report the hit rates of the stream descriptor cache and the last stream predictor, *hrSDC* and *hrLSP*, because they directly influence the size of the output trace as explained in Equation (4.3) and Equation (4.4). Second, we introduce several enhancements to the original mechanism and explore their effectiveness in further improving the compression ratio at minimal added complexity or in reducing the trace module complexity.

SDC Access Function. A good hash access function should minimize the number of collisions in the SDC. Its efficacy depends on the program characteristics and SDC organization. We have evaluated a number of access functions while varying the SDC size and organization. We have found that access functions that combine the SA and SL portions of the stream descriptor in general outperform those based solely on the SA because multiple streams can have the same starting address. Our experiments indicate that the hash function shown in Equation (4.5) performs the best for different SDC sizes and configurations. The SA is shifted by *shift* bits and then the result is XORed with the SL. The lower $\log_2 N_{SET}$ bits of the result are used as the set index, *iSet*. The optimal value for *shift* was found to be 4 for our benchmark suite. N_{SET} has to be a power of two.

$$iSet = ((SA << shift)\ xor\ SL)\ and\ (N_{SET} - 1) \tag{4.5}$$

SDC Size and Organization. Figure 4.14 shows the average SDC hit rate and trace port bandwidth for the whole benchmark suite when varying the number of entries and the number of ways (N_{WAYS} = 1, 2, 4, 8). It should be noted that considering the SDC hit rate alone is not sufficient to make a definite design decision about the SDC organization. Sometimes a suboptimal organization can result in a lower trace port bandwidth because of the last stream predictor and its access function. The trace port bandwidth is calculated assuming an LSP with the same number of entries as the SDC and a simple hash access function that uses the previous stream index to access the LSP.



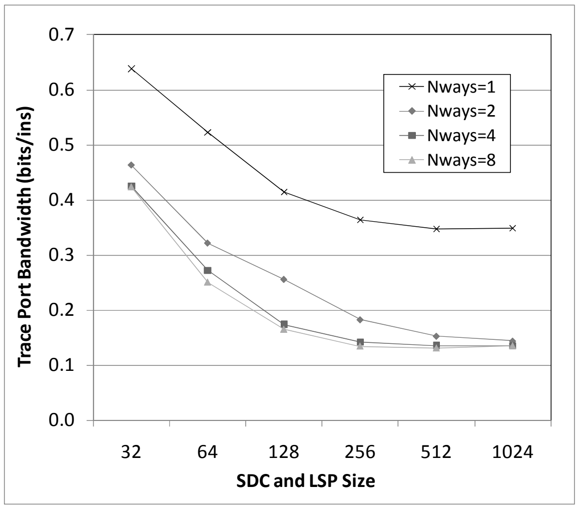


Figure 4.14 SDC hit rate and trace port bandwidth as functions of its size and organization

The results show that increasing the stream cache associativity helps reduce the trace port bandwidth and thus improves the compression ratio, but only up to a point. Increasing the associativity beyond 4 ways yields little or no benefit. The results further indicate that even relatively small stream caches with as few as 32 entries perform well, achieving less than 0.5 bits/ins on the trace port. Increasing the SDC and consequently the LSP size beyond 256 entries is not beneficial as it only yields diminishing returns in trace port bandwidth. Based on these results, we choose a 4-way associative SDC with 128 entries and a 128-entry LSP as a good configuration for our trace module. This configuration represents a sweet spot in the trade-off between trace port bandwidth and design complexity; with our benchmarks, it yields under 0.2 bits/ins at a modest cost. Table 4.6 shows the 4-way SDC hit rate as function of its size, for each benchmark.

We have also evaluated several SDC replacement policies including Pseudo-Random, FIFO (First-In First-Out), LRU (Least Recently Used), and several pseudo-LRU implementations. Our findings indicate that a pseudo-LRU replacement policy based on using MRU (Most Recently Used) bits performs the best, outperforming even the full LRU policy.

Table 4.6 SDC hit rate as functions of its size and organization

	hrSDC					
Test \ Size	**32**	**64**	**128**	**256**	**512**	**1024**
adpcm_c	1.000	1.000	1.000	1.000	1.000	1.000
bf_e	0.985	0.996	1.000	1.000	1.000	1.000
cjpeg	0.952	0.991	0.998	0.999	1.000	1.000
djpeg	0.935	0.971	0.991	0.997	0.998	0.999
fft	0.482	0.685	0.859	0.952	0.985	1.000
ghostscript	0.456	0.778	0.987	0.993	0.997	0.999
gsm_d	0.972	0.980	0.989	0.996	0.999	1.000
lame	0.903	0.938	0.954	0.964	0.972	0.987
mad	0.833	0.972	0.984	0.993	0.998	1.000
rijndael_e	0.542	0.866	0.929	1.000	1.000	1.000
rsynth	0.848	0.923	0.966	0.997	1.000	1.000
sha	0.952	0.999	1.000	1.000	1.000	1.000
stringsearch	0.759	0.868	0.971	0.991	0.993	0.999
tiff2bw	0.971	0.979	0.992	0.998	1.000	1.000
tiff2rgba	0.935	0.969	0.996	1.000	1.000	1.000
tiffdither	0.824	0.904	0.963	0.988	0.997	1.000
tiffmedian	0.975	0.983	0.992	0.997	1.000	1.000
Average	**0.838**	**0.921**	**0.969**	**0.988**	**0.994**	**0.998**

Last Stream Predictor. We have considered several LSP organizations. The number of entries in the LSP may exceed the number of entries in the stream descriptor cache. In such a case, the LSP access function should be based on the program path taken to a particular stream. The path information may be maintained in a history buffer as a function of previous stream cache indices. However, our experimental analysis indicates that such an approach yields fairly limited improvements in trace port bandwidth. The reason is that our workload has a relatively small number of indirect branches, and those branches mostly have a very limited number of targets taken during program execution. Consequently, we chose the simpler solution of always having the same number of entries in the LSP and the SDC. The LSP access function is based solely on the previous stream cache index. We call this basic implementation of the proposed tracing mechanism *bSDC-LSP*.

Table 4.7 LSP hit rates (a), and LSP trace port bandwidth (b), for the bSDC-LSP scheme

	hrLSP						**bits/ins**					
Test \ Size	**32**	**64**	**128**	**256**	**512**	**1024**	**32**	**64**	**128**	**256**	**512**	**1024**
adpcm_c	0.997	0.997	0.997	0.997	0.997	0.997	0.019	0.019	0.019	0.019	0.019	0.019
bf_e	0.824	0.838	0.843	0.843	0.843	0.843	0.405	0.359	0.357	0.384	0.410	0.437
cjpeg	0.894	0.916	0.921	0.922	0.922	0.922	0.204	0.138	0.131	0.134	0.140	0.146
djpeg	0.887	0.896	0.909	0.915	0.916	0.917	0.125	0.093	0.075	0.070	0.072	0.075
fft	0.522	0.674	0.762	0.827	0.850	0.870	1.492	1.007	0.616	0.354	0.256	0.219
ghostscript	0.518	0.696	0.865	0.868	0.869	0.870	1.585	0.823	0.232	0.227	0.229	0.234
gsm_d	0.946	0.946	0.947	0.951	0.952	0.954	0.103	0.094	0.086	0.077	0.076	0.075
lame	0.807	0.820	0.823	0.827	0.829	0.833	0.129	0.108	0.102	0.101	0.100	0.093
mad	0.715	0.825	0.830	0.832	0.835	0.836	0.295	0.136	0.129	0.124	0.124	0.128
rijndael_e	0.697	0.846	0.809	0.867	0.867	0.867	0.743	0.284	0.192	0.099	0.105	0.111
rsynth	0.843	0.843	0.860	0.883	0.887	0.887	0.382	0.245	0.175	0.116	0.115	0.122
sha	0.893	0.922	0.922	0.922	0.922	0.922	0.178	0.097	0.101	0.106	0.111	0.116
stringsearch	0.829	0.807	0.857	0.870	0.872	0.872	1.369	0.938	0.472	0.387	0.401	0.382
tiff2bw	0.996	0.993	0.992	0.992	0.994	0.994	0.151	0.135	0.104	0.087	0.083	0.084
tiff2rgba	0.991	0.978	0.989	0.989	0.989	0.989	0.114	0.077	0.045	0.040	0.040	0.040
tiffdither	0.834	0.823	0.848	0.864	0.870	0.873	0.332	0.249	0.190	0.164	0.157	0.160
tiffmedian	0.976	0.973	0.971	0.973	0.976	0.976	0.085	0.077	0.066	0.058	0.055	0.056
Average	**0.819**	**0.856**	**0.881**	**0.893**	**0.897**	**0.899**	**0.426**	**0.272**	**0.174**	**0.142**	**0.136**	**0.135**
			a)						***b)***			

Table 4.7*a* shows the last stream predictor hit rate (*hrLSP*), and Table 4.7*b* shows trace port bandwidth for individual benchmarks and for different sizes of the SDC and LSP. The trace port bandwidth for the trace module configuration [32x4, 128] (4-way set-associative 128-entry SDC and 128-entry LSP) varies between 0.019 bits/ins for *adpcm_c* and 0.616 bits/ins for *fft*, and is 0.174 bits/ins on average for the whole benchmark suite. The *fft* benchmark significantly benefits from an increase in the SDC size and requires 0.354 bits/ins with the [64x4, 256] configuration. Many of the remaining benchmarks perform well even with very small trace module configurations, e.g., *adpcm_c*, *tiffmedian*, and *tiff2rgba*.

4.4.3 Enhancements for Base SDC-LSP Scheme

The output trace records still contain a lot of redundant information that can be eliminated with low-cost enhancements. The three components of the output trace are (i) LSP-hit records (*hLSPt*), (ii) LSP-miss with SDC-hit records (*hSDCt*), and (iii) LSP-miss and SDC-miss records (*mSDCt*). Table 4.8 shows distributions of the individual trace components for two trace module configurations: [16x4, 64] and [64x4, 256]. The *mSDCt* component dominates the output trace in smaller configuration; e.g., it is responsible for 41.3% of the total output for the [16x4, 64] configuration. With larger configurations, the *hLSPt* component dominates the output trace with long runs of consecutive ones; e.g., the *hSLPt* represents 48.5% of the total output trace for the [64x4, 256] configuration.

By analyzing the *mSDCt* records we observe that the upper address bits of the starting address (SA) field rarely change. To take advantage of this property, we slightly modify our *bSDC-LSP* compressor as follows. An additional *u*-bit register called LVSA is added to record the *u* upper bits of the SA field from the last miss trace record. The upper *u*-bit field of the SA of the incoming miss trace record is compared to the LVSA. If there is a match, the new miss trace record will include only the lower (32-*u*) address bits. Otherwise, the whole address is emitted and the LVSA register is updated. To distinguish between these two cases, an additional bit in the trace record is needed to indicate whether all (SA[31:0]) or only the lower address bits (SA[31-u:0]) are emitted.

Table 4.8 Distributions of individual trace components for two trace module configurations

[SDC, LSP] Size	**[16x4, 64]**			**[64x4, 256]**		
Test	**mSDCt**	**hSDCt**	**hLSPt**	**mSDCt**	**hSDCt**	**hLSPt**
adpcm_c	0.1%	1.9%	98.1%	0.1%	2.4%	97.5%
bf_e	6.9%	53.6%	39.6%	0.0%	62.7%	37.3%
cjpeg	12.0%	34.5%	53.5%	1.2%	42.9%	55.9%
djpeg	32.2%	30.4%	37.5%	4.8%	43.3%	51.9%
fft	80.8%	14.8%	4.4%	39.0%	39.8%	21.2%
ghostscript	73.2%	20.2%	6.6%	9.5%	52.3%	38.1%
gsm_d	29.4%	20.1%	50.5%	8.2%	29.0%	62.7%
lame	44.4%	33.7%	21.9%	29.5%	46.0%	24.4%
mad	29.1%	42.4%	28.5%	9.4%	58.4%	32.2%
rijndael_e	72.0%	15.7%	12.3%	0.1%	58.0%	41.9%
rsynth	58.3%	23.6%	18.1%	5.3%	51.5%	43.3%
sha	1.6%	36.6%	61.8%	0.1%	43.1%	56.7%
stringsearch	66.9%	20.7%	12.4%	13.4%	49.7%	36.9%
tiff2bw	41.2%	2.8%	56.1%	5.3%	6.1%	88.6%
tiff2rgba	48.5%	7.1%	44.4%	0.8%	8.7%	90.5%
tiffdither	47.4%	31.6%	21.0%	11.6%	51.9%	36.6%
tiffmedian	33.1%	10.8%	56.0%	6.6%	18.4%	74.9%
Average	**41.3%**	**23.7%**	**34.9%**	**11.9%**	**39.6%**	**48.5%**

The format of the trace record for an LSP miss with SDC miss event is modified to include this additional bit that precedes the stream descriptor field. Note that SA[1:0] is always '00' for the ARM ISA and is omitted from the *mSDCt*. For the ARM Thumb ISA, only SA[0] can be omitted. These two bits do not need to be kept in the stream descriptor cache. In addition, we can also omit the address bits that can be inferred from the SDC index (this enhancement will be discussed further down).

Increasing the width of the LVSA register reduces the number of bits in the miss trace in the case of LVSA hits; however, it also reduces the number of LVSA hit events. Table 4.9 shows the fraction of the original miss trace components for various values of the parameter *u* for the [32x4, 128] configuration. For example, we find that the LVSA enhancement reduces the miss trace component by 18% when $u=14$. It should be noted that the reduction in the total output trace is more significant for smaller trace module configurations and relatively insignificant for larger configurations because the miss trace component is relatively small in the latter case.

The redundancy in the *hLSPt* component can be reduced using a counter that counts the number of consecutive bits with value '1'. The counter is called *one length counter* or OLC for short. Long runs of ones are replaced by the counter value preceded by a new header. The number of bits used to encode this trace component is determined by the counter size. A longer counter can capture longer runs of ones, but a counter which is too long results in wasted bits. Our analysis of the *hLSPt* components shows a fairly large variation in the average number of consecutive ones, ranging from 5 in *ghostscript* and *fft* to hundreds in *adpcm_c* and *tiff2bw*. In addition, these sequences of consecutive ones may vary across different program phases, meaning that an adaptive OLC length method would yield better results.

Table 4.9 Fraction of the original miss trace component using LVSA

Test \ u	21	20	19	18	17	16	15	14	13	12
adpcm_c	0.83	0.82	0.83	0.83	0.83	0.82	0.79	0.79	0.81	0.82
bf_e	0.87	0.85	0.74	0.72	0.73	0.75	0.76	0.77	0.79	0.81
cjpeg	0.87	0.86	0.86	0.86	0.85	0.82	0.81	0.82	0.84	0.85
djpeg	0.87	0.84	0.84	0.84	0.84	0.81	0.81	0.82	0.83	0.84
fft	0.89	0.89	0.87	0.86	0.86	0.85	0.83	0.81	0.83	0.84
ghostscript	0.87	0.86	0.86	0.85	0.86	0.85	0.84	0.85	0.84	0.84
gsm_d	0.87	0.86	0.86	0.82	0.82	0.81	0.81	0.82	0.83	0.85
lame	0.87	0.86	0.85	0.85	0.86	0.87	0.86	0.86	0.87	0.88
mad	0.82	0.81	0.81	0.81	0.79	0.78	0.78	0.79	0.81	0.82
rijndael_e	0.96	0.96	0.83	0.84	0.85	0.82	0.83	0.84	0.86	0.87
rsynth	0.88	0.89	0.87	0.83	0.83	0.75	0.77	0.78	0.80	0.82
sha	0.87	0.88	0.79	0.71	0.72	0.74	0.75	0.76	0.78	0.80
stringsearch	0.83	0.84	0.84	0.85	0.78	0.77	0.79	0.80	0.82	0.83
tiff2bw	0.88	0.85	0.83	0.84	0.83	0.80	0.82	0.78	0.80	0.82
tiff2rgba	0.96	0.96	0.96	0.90	0.85	0.86	0.82	0.77	0.79	0.80
tiffdither	0.95	0.94	0.93	0.93	0.93	0.92	0.93	0.93	0.93	0.94
tiffmedian	0.88	0.88	0.83	0.83	0.83	0.81	0.81	0.78	0.80	0.82
Average	**0.88**	**0.87**	**0.85**	**0.84**	**0.84**	**0.83**	**0.82**	**0.82**	**0.84**	**0.85**

The adaptive one-length counter (AOLC) dynamically adjusts the OLC size to the program flow characteristics. An additional 4-bit saturating counter monitors the *hLSPt* entries and is updated as follows. It is incremented by 3 when the number of consecutive ones in the *hLSPt* trace exceeds the current size of the OLC. The monitoring counter is decremented by 1 whenever the number of consecutive ones is smaller than half of the maximum OLC counter value. When the monitoring counter reaches its maximum (15) or minimum (0), a change in the OLC size occurs. If the maximum is reached, the OLC size is increased by one bit (if possible). If the minimum is reached, the OLC size is decreased by one bit (if possible).

Using an AOLC necessitates a slight modification of the trace output format. We use a header bit '1' that is followed by $log_2(AOLC\ Size)$ bits. The counter size is automatically adjusted as described above. Of course, the software decompressor needs to implement the same adaptive algorithm. We call this scheme, which includes the LVSA and AOLC optimizations, *eSDC-LSP*.

Reducing Hardware Complexity by eliminating upper bits. The LVSA enhancement could be slightly modified to reduce the overall size of the trace module implementation. For example, the uppermost 12 bits of the stream starting address do not change with a probability of 0.99 in our benchmarks. Consequently, we may opt not to keep the upper address bits SA[31:20] in the stream descriptor cache, thus reducing its size. Instead, the upper address bits are handled entirely by a last value predictor in a manner similar to the LVSA enhancement discussed above. The mechanism used in the *eSDC-LSP* scheme can be modified as follows. In the *eSDC-LSP* scheme, we only considered trace records in the miss trace (*mSDCt*), updating the LVSA register only when an LSP miss with SDC miss event occurs. Here we need to continuously update the LVSA register, regardless of whether we have a hit or a miss in the SDC and LSP structures. Moreover, a miss in the LVSA register results in sending a stream descriptor to the output trace; the SDC and LSP are updated accordingly. To determine the optimal number of upper bits that should be handled by the LVSA predictor, we need to consider the SDC performance. Reducing the number of address bits that are stored in the SDC reduces its size, but may result in an increased miss rate and thus increase the trace port bandwidth. A modified *eSDC-LSP* with the uppermost 12 address bits handled by the LVSA appears optimal on our benchmarks.

Reducing Hardware Complexity by eliminating indexing bits. We can further reduce the number of bits kept in the stream descriptor cache without any negative impact on the trace module performance. The bits of the starting address *SA[shift+log$_2$(N$_{SET}$)-1:shift]* that are used in the calculation of the SDC index function (Equation 3) do not need to be kept in the SDC. This information can be inferred based on the known index function and SL bits that are stored in the SDC. (Alternatively, we can keep all address bits in the stream cache and eliminate the portion of the SL bits that are used for the SDC index.) For example, in the [32x4, 128] configuration, the *iSet* is calculated as the XOR result of *SA[8:4]* and *SL[4:0]*. Consequently, we can infer the value of *SA[8:4]* as *SA[8:4] = iSet* XOR *SL[4:0]*. The *eSDC-LSP* scheme with modified LVSA enhancement and reduced complexity of the SDC is called *rSDC-LSP* scheme.

4.4.4 Trace Port Bandwidth Analysis

Table 4.10 shows the trace port bandwidth of the *eSDC-LSP* scheme for individual programs and for different sizes of the SDC and LSP. We observe relatively high improvements for small trace module configurations, mainly due to a reduction in the *mSDCt* size; for example, the average trace port bandwidth for the [8x4, 32] configuration is 0.35 bits/ins, down from 0.43 bits/ins in the *bSDC-LSP* scheme (18% lower). Similarly, for large trace module configurations, the *hLSPt* size is significantly reduced; for example, the trace port bandwidth for the [64x4, 245] configuration is 0.12 bits/ins, versus 0.142 bits/ins in the *bSDC-LSP* scheme (a 15% reduction). Some programs benefit significantly from this enhancement, especially those with a high LSP hit rate, such as *adpcm_c* (over 14 times lower bandwidth), *tiff2bw* (3.45), and *tiff2rgba* (3.67).

Table 4.11 shows the trace port bandwidth of the *rSDC-LSP* scheme for different sizes of the SDC and LSP. The upper twelve address bits *SA[31:20]* are predicted using the last value predictor and an entry in the stream cache consists of the lower 13 address bits SA[19:9] and SA[3:2] and the stream length field SL[7:0]. The *rSDC-LSP* scheme requires slightly higher bandwidth on the trace port than *eSDC-LSP*. For example, the trace module configuration [32x4, 128] achieves 0.15 bits/ins at the trace port versus 0.146 bits/ins for the *eSDC-LSP* scheme. However, this degradation due to aliasing in the SDC is less than 3%, which is probably an acceptable loss for a significant reduction in the size of the stream descriptor cache (we keep 13 instead of 30 bits for stream starting addresses).

Table 4.10 Trace port bandwidth of the *eSDC-LSP* scheme

eSDC-LSP	**bits/ins**					
Test \ Size	**32**	**64**	**128**	**256**	**512**	**1024**
adpcm_c	0.001	0.001	0.001	0.001	0.001	0.001
bf_e	0.378	0.342	0.345	0.372	0.398	0.425
cjpeg	0.154	0.095	0.088	0.092	0.098	0.104
djpeg	0.092	0.068	0.053	0.048	0.050	0.053
fft	1.235	0.851	0.542	0.327	0.237	0.196
ghostscript	1.358	0.760	0.216	0.214	0.217	0.224
gsm_d	0.062	0.057	0.051	0.045	0.043	0.042
lame	0.110	0.094	0.090	0.090	0.090	0.085
mad	0.254	0.120	0.116	0.114	0.115	0.120
rijndael_e	0.599	0.239	0.183	0.090	0.096	0.103
rsynth	0.297	0.200	0.147	0.097	0.097	0.103
sha	0.141	0.070	0.074	0.079	0.084	0.089
stringsearch	1.082	0.789	0.412	0.344	0.358	0.345
tiff2bw	0.062	0.052	0.030	0.016	0.011	0.012
tiff2rgba	0.066	0.041	0.012	0.007	0.008	0.008
tiffdither	0.279	0.213	0.158	0.135	0.129	0.133
tiffmedian	0.039	0.035	0.027	0.021	0.017	0.018
Average	**0.349**	**0.230**	**0.146**	**0.120**	**0.114**	**0.114**

Table 4.11 Trace port bandwidth of the *rSDC-LSP* scheme

rSDC-LSP	**bits/ins**					
Test \ Size	**32**	**64**	**128**	**256**	**512**	**1024**
adpcm_c	0.001	0.001	0.001	0.001	0.001	0.001
bf_e	0.381	0.343	0.345	0.372	0.398	0.425
cjpeg	0.156	0.096	0.088	0.092	0.098	0.104
djpeg	0.094	0.068	0.054	0.048	0.050	0.053
fft	1.276	0.874	0.554	0.331	0.237	0.196
ghostscript	1.377	0.758	0.216	0.214	0.217	0.224
gsm_d	0.064	0.058	0.052	0.045	0.043	0.042
lame	0.128	0.113	0.109	0.109	0.110	0.106
mad	0.259	0.121	0.117	0.114	0.115	0.120
rijndael_e	0.623	0.246	0.185	0.090	0.096	0.103
rsynth	0.308	0.205	0.149	0.097	0.097	0.103
sha	0.143	0.070	0.074	0.079	0.084	0.089
stringsearch	1.118	0.807	0.416	0.346	0.359	0.345
tiff2bw	0.064	0.054	0.031	0.016	0.011	0.012
tiff2rgba	0.069	0.042	0.012	0.007	0.008	0.008
tiffdither	0.282	0.214	0.159	0.135	0.129	0.133
tiffmedian	0.040	0.035	0.028	0.021	0.017	0.018
Average	**0.359**	**0.235**	**0.150**	**0.123**	**0.117**	**0.117**

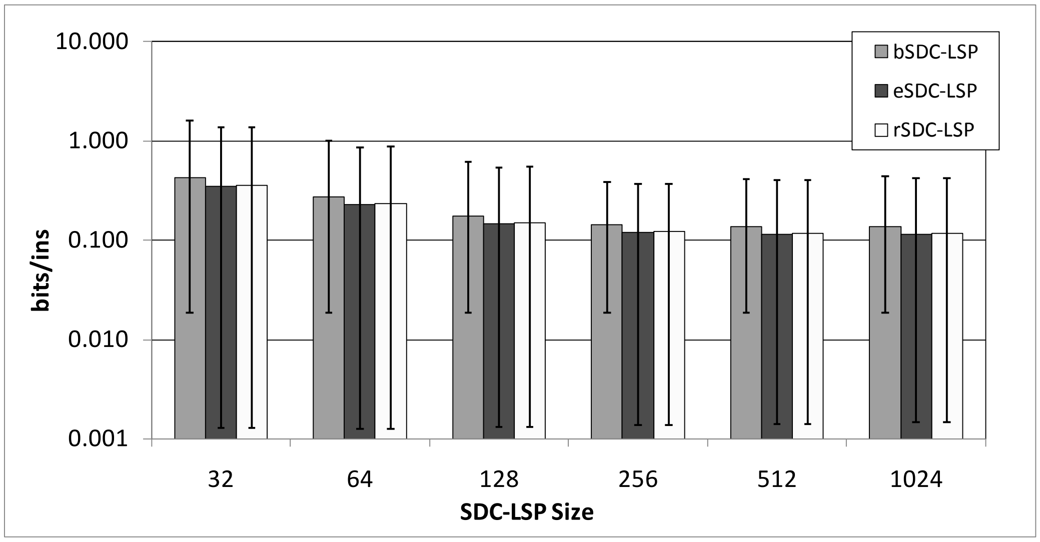


Figure 4.15 Trace port bandwidth of all proposed schemes.

Figure 4.15 shows the trace port bandwidth for a range of trace module configurations for the three proposed schemes, *bSDC-LSP*, *eSDC-LSP*, and *rSDC-LSP*. As we pointed out before, the low-cost enhancements discussed in the previous sections indeed reduce the required trace port bandwidth. The *rSDC-LSP* scheme trails a little behind the *eSDC-LSP* scheme, but offers a significant cost reduction. Consequently, we consider *rSDC-LSP* to be the most cost-effective scheme striking a good balance between trace port bandwidth and implementation complexity.

4.4.5 Hardware Implementation and Complexity

Both SDC and LSP use simple cache-like structures (multi-way or direct-mapped). Cache implementation techniques are well known and are not addressed here. We focus on finding the complexity, in number of logic gates, of different proposed SDC-LSP configurations.

To estimate the size of different proposed configurations, we first need to determine the minimum sizes of the trace output buffer. The size of this structure should be such that the traces are never dropped due to the finite capacity of the trace structures and that no trace record is lost. To determine the size, we use a cycle-accurate processor model similar to Intel's XScale processor [67]. The trace module is modeled as follows. We assume that it requires one clock cycle to service an LSP with SDC hit or an LSP miss with an SDC hit event and two clock cycles for an SDC miss event. The trace records are stored in the trace output buffer. If the output buffer is not empty, a single bit is sent out through the trace data port each clock cycle. The processor is never stalled and no trace records are lost if the following conditions are met: the number of entries in the stream descriptor buffer is at least 2 and the minimum trace output buffer size is 80 bits.

The estimation of the SDC and LSP compressing structures is straightforward. As an example, let us assess the complexity of the *rSDC-LSP* scheme for the [32x4, 128] configuration. The stream descriptor includes a register pair for the starting address and stream length with a total of 38 bits of storage (30 bits for the SA and 8 bits for the SL). An entry in the stream descriptor cache requires 13 bits for the SA (30 - 12 - 5), 8 bits for the SL, a valid bit, and one replacement bit (MRU-based LRU). The total amount of storage is thus (128 - 1)*(13 + 8 + 2) = 2921 bits. Similarly, we find that the LSP requires 903 bits of storage. Assuming 1.5 logic gates per memory bit, we estimate the complexity of our *rSDC-LSP* scheme to be fewer than 6,100 gates.

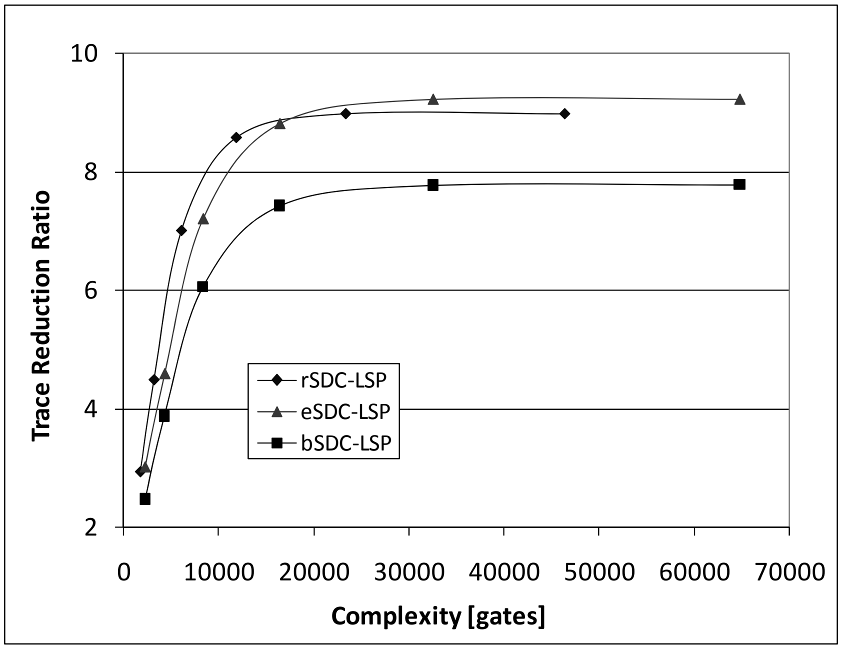


Figure 4.16 Trace reduction ratio vs. complexity for the three proposed schemes (higher is better)

Figure 4.16 shows the trace reduction ratio as a function of trace module complexity. The trace reduction ratio is calculated as the ratio of the average trace port bandwidth for the uncompressed stream descriptors ((SA/-, SL) pairs coming from the stream detector) and the average trace port bandwidth for the proposed schemes *xSDC-LSP*, where x = (b, e, r). Different points represent different trace module configurations, varied from 32 entries [8x4, 32] to 1024 entries [256x4, 1024]. For example, *rSDC-LSP* reduces the trace port bandwidth over 7 times, at the cost of less than 6,100 gates (configuration [32x4, 128]) or 8.6 times at the cost of 11,900 gates. At the low end of complexity, which is what we are interested in, *rSDC-LSP* emerges as the best solution and is therefore our recommended implementation.

4.5 Program Execution Tracing using Branch Predictors

Almost all modern mid- to high-end embedded processors include branch predictors in their front-ends. Branch predictors detect branches and predict the branch target address and the branch outcome in the early pipeline stages, thus reducing the number of wasted clock cycles in the pipeline due to control hazards. The target of a branch is predicted using a branch target buffer (BTB) – a cache structure indexed by a portion of the branch address [68] that keeps target addresses of taken branches. A separate hardware structure named an indirect branch target buffer (iBTB) can be used to better predict indirect branches that may have multiple target addresses [69]. A dedicated stack-like hardware structure called the Return Address Stack (RAS) is often used to predict return addresses. The branch outcome predictors range from a simple linear branch history table (BHT) with 2-bit saturating counters (2bc) to very sophisticated hybrid branch outcome predictor structures found in recent commercial microprocessors

[70]. Branch predictors proved to be very effective in predicting branch outcomes and target addresses with high probability (over 95%).

As described in the introduction to this chapter, a program flow change can be captured by recording branch address/target pairs and this is the exactly what the branch predictor does. However, we do not need to send information about all the branches to the trace port to be able to replay the program offline. If the software debugger maintains the exact software copy of the Trace Module branch predictor and uses the same policies for the branch predictor update, the control flow information can come from the local predictor, rather than traced out from the target platform. The trace module thus needs to report only misprediction events which are relatively rare.

The rest of this section is organized as follows. In Section 4.5.1 we describe operation of the trace module based on branch predictor. Section 4.5.2 describes our approach to encoding of branch predictor related trace events. In Section 4.5.3 we give the results of our trace port bandwidth analysis, and Section 4.5.4 estimates hardware implementation cost.

4.5.1 Trace Module Branch Predictor

Our key observation is that the program execution can be replayed off-line using a branch predictor trace instead of a branch instruction trace. We implement a trace module that consists of branch predictor structures that are solely dedicated to trace compression. To distinguish it from the processor branch predictor, we name it Trace Module Branch Predictor (TMBP). TMPB includes structures for predicting branch targets and branch outcomes. Unlike regular branch predictors, TMBP does not need to include a large BTB because direct branch targets can be inferred from the binary, but it may include an iBTB for predicting targets of indirect branches and a RAS for predicting return addresses.

The TMBP structures are updated similarly to those in a regular branch predictor, but only when a branch instruction is retired. As long as the prediction from the TMBP corresponds to the actual program flow, the trace module does not need to send any trace records. It records only misprediction events and they are encoded and sent via a trace port to the software debugger. The software debugger maintains an exact software copy of the TMBP structures. It reads the branch predictor trace records, replays the program instruction-by-instruction, and updates the software structures in the same way TMBP is updated during program execution.

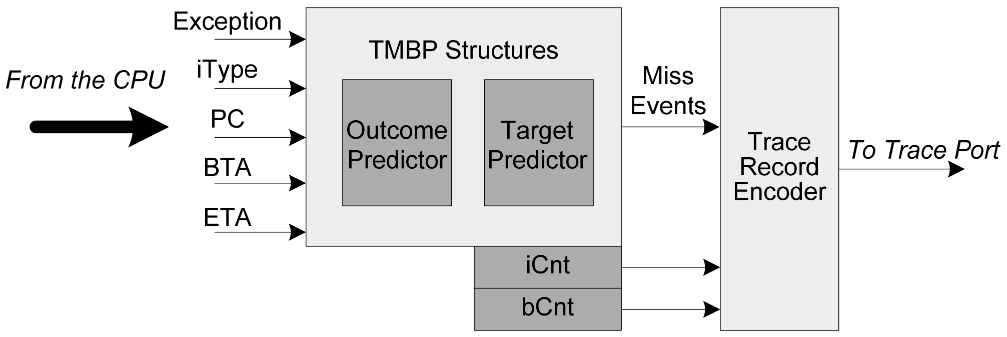


Figure 4.17 Trace module: a system view

Figure 4.17 shows a system view of the proposed tracing mechanism. The trace module is coupled with the CPU core through an interface that carries relevant branch related information. The trace module includes two counters: an instruction counter (*iCnt*) that counts the number of instructions executed since the last trace event has been reported, and a branch counter (*bCnt*) that counts the number of relevant control-flow instructions executed since the last trace event has been reported (see Figure 4.18 for TMBP operation).

```
// For each committed instruction
iCnt++; // increment the iCnt
if ((iType==IndBr)||(iType==DirCB)) {
     bCnt++;// increment bCnt
     if (TMBP mispredicts) {
        Encode an BPM event;
        Place record into the Trace Buffer;
        iCnt = 0; bCnt = 0;
     }
}
if (Exception event) {
    Encode an exception event;
    Place record into the Trace Buffer;
    iCnt = 0; bCnt = 0;
}
```

Figure 4.18 Trace module operation

The counters are being updated upon completion of an instruction in its retirement phase; *iCnt* is incremented after each instruction and *bCnt* is incremented only upon retirement of control-flow instructions of certain types, namely, after direct conditional branches (DirCB) and all indirect branches (IndB). (Unconditional direct branches are not of our interest as their target can be inferred by the software debugger.) These branch instructions may be either correctly predicted or mispredicted by TMBP. In the case of a correct prediction, the trace module does nothing beyond the counter and branch structures updates. In the case of a misprediction, the trace module generates a trace record that needs to be traced out to the software debugger. The type and format of the trace record depends on the misprediction event type (Table 4.12). An outcome misprediction trace record includes the *bCnt* value only, so that the software debugger can replay the program execution until the mispredicted branch is reached. In the case of an indirect branch misprediction, we can have an outcome misprediction or the target address misprediction (or both). For an indirect branch incorrectly predicted as taken, a trace record includes the *bCnt* and the information that the branch is not taken (NT bit). In the case of a target address misprediction, a trace record includes the *bCnt*, the outcome bit taken (*T*), and the actual target address (*TA*). Finally, in the case that an exception occurs, the trace module prepares a trace record that includes the *iCnt* and the starting address of the corresponding exception handler.

Table 4.12 Trace module branch pred. events and trace records.
[T – Taken, NT – Not Taken]

Branch Type	***TMBP Event***	***Trace Record***
DirCB	Outcome mispred.	(header, **bCnt**)
IndB (NT)	Outcome mispred.	(header, **bCnt, NT**)
IndB (T) /Uncond.	Target mispred.	(header, **bCnt**, **T, TA**)
Exception	--	(header, **iCnt**, **ET**)

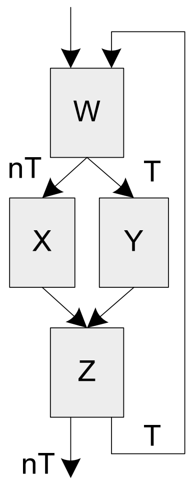
The software debugger replays all instructions updating its software copy of the branch predictor and the trace module counters in the same way the hardware counterparts are updated (Figure 4.19). The debugger reads a trace record and then replays the program instruction-by-instruction. If it processes a non-exception trace record, the counter *bCnt* is decremented on retirement of direct conditional and indirect branch instructions. When the counter *bCnt* reaches a zero, the software debugger processes the instruction depending on its type. If the current instruction is a direct conditional branch, the debugger takes the opposite outcome from the one provided by the predictor. The predictor is updated and a new trace record is read to continue program replay. If the current instruction is an indirect branch, the debugger reads the target address from the trace record, redirects the program execution, and updates its predictor accordingly. Similarly, if the debugger processes an exception trace record, the *iCnt* counter is decremented on each instruction until the instruction on which the exception has occurred is reached.

```
// For each instruction
Replay the current instruction;
if (An ETR is being processed) {
   iCnt--;
   if (iCnt == 0) {
       Goto Exception Handler Routine;
       Read the next trace record;
   }
}
if (iType = AnyBranch) {
   Update software copy of the TMBP;
   if ((iType==IndBr)||(iType==DirCon)) {
      bCnt--;
      if (bCnt==0)
           Read the next trace record;
       }
}
```

Figure 4.19 Program execution replay in software debugger

An Example. Let us first illustrate the program tracing using an example. A code segment consists of 4 basic blocks *W*, *X*, *Y*, and *Z* as illustrated in Figure 4.20*a*. Let us consider three iterations of the program loop with the following execution pattern $\{WXZ\}^2\{WYZ\}$. The code sequence includes only direct branches and only two basic blocks *W* and *Z* end with conditional branches (`jle Y` and `jge W`). Let us assume that the branch predictor initial state predicts the branch `jle Y` to be not taken (*P*=*NT*), and the branch `jge W` to be taken (*P*=*T*). The program execution starts from the first instruction in the block *W* (*i1*); the trace module keeps incrementing the *iCnt* and *bCnt* counters as shown in Figure 4.20*b* (the execution table). In the first two loop iterations, conditional branches are correctly predicted. In the third iteration, the branch predictor predicts the branch `jle Y` as not taken when it is actually taken (*A*=*T*), so we have an outcome misprediction event. The trace module emits a trace record that includes information about the misprediction type (outcome misprediction) and the number of branches that have been correctly predicted since the last trace event, *bCnt*=5. The counters are cleared and the program execution continues with block Y. In the last iteration, the instruction `jge W` is predicted taken (*P*=*T*), but it is actually not taken (*A*=*NT*). A new trace record due to outcome misprediction is emitted and the counter value is *bCnt*=1.

The software debugger on its side is ready to replay the program starting with the instruction *i1*. It receives a trace record that indicates that the program should be replayed until the fifth conditional branch is reached (Figure 4.20*c*, the replay table). The program is replayed instruction-by-instruction and the software copy of the TMBP and the replay counters are updated accordingly. When the counter *bCnt* reaches zero (at the instruction `jle Y` in the 3rd iteration), the debugger knows that the branch outcome of the currently replayed direct branch is different from the one suggested by the software TMBP. The debugger needs to update its predictor structures according to the update policies. It then reads the next trace record and continues program replay from the first instruction in block Y.



(a)

Execution

Instr.	*Iteration 1*		*Iteration 2*		*Iteration 3*	
	P/A	***{i/b} Cnt***	***P/A***	***{i/b} Cnt***	***P/A***	***{i/b} Cnt***
W: i1		1/0		11/2		21/4
i2		2/0		12/2		22/4
jle Y	nT/nT	3/1	nT/nT	13/3	nT/nT*	23/5
X: i4		4/1		14/3		
i5		5/1		15/3		
i6		6/1		16/3		
jmp Z		7/1		17/3		
Y: i8						1/0
i9						2/0
i10						3/0
i11		8/1		18/3		4/0
i12		9/1		19/3		5/0
jge W	T/T	10/2	T/T	20/4	T/nT*	6/1

(b)

Replay

	Iteration 1		*Iteration 2*		*Iteration 3*	
	P/A	***{i/b} Cnt***	***P/A***	***{i/b} Cnt***	***P/A***	***{i/b} Cnt***
W: i1		-/5		-/3		-/1
i2		-/5		-/3		-/1
jle Y	nT/nT	-/4	nT/nT	-/2	nT/nT*	-/0
X: i4		-/4		-/2		
i5		-/4		-/2		
i6		-/4		-/2		
jmp Z		-/4		-/2		
Y: i8						-/1
i9						-/1
i10						-/1
i11		-/4		-/2		-/1
i12		-/4		-/2		-/1
jge W	T/T	-/3	T/T	-/1	nT/T*	-/0

(c)

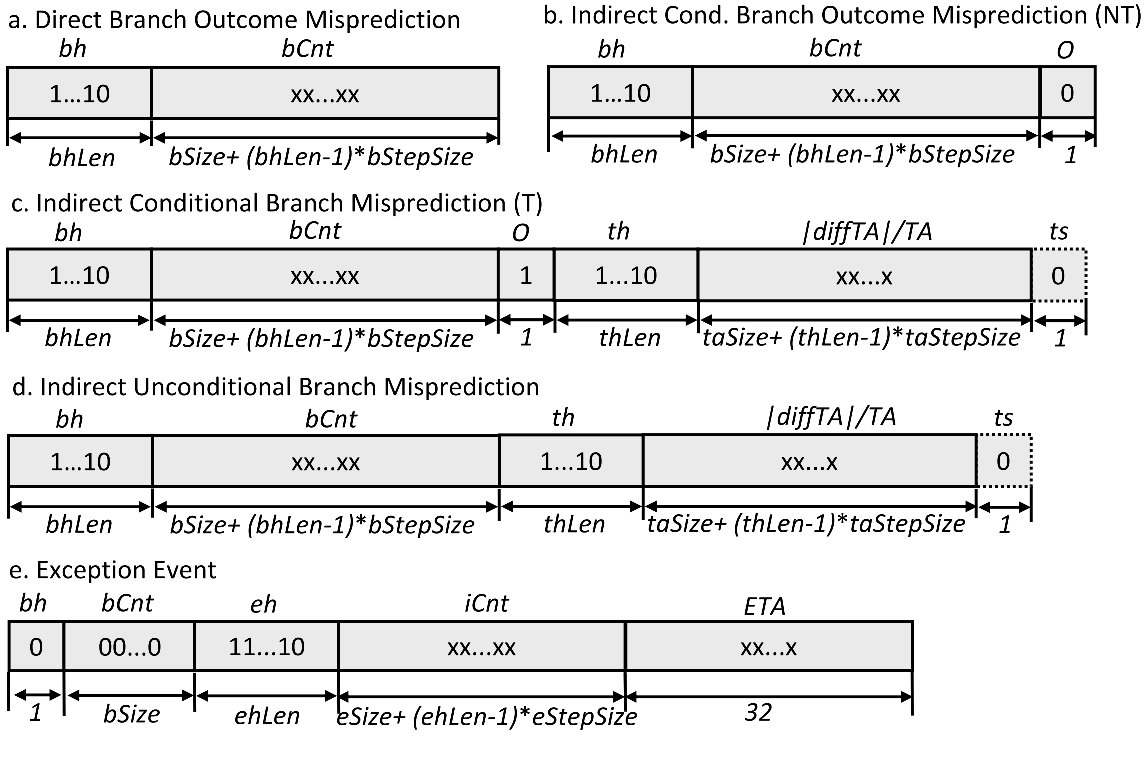
Figure 4.20 Tracing and replaying program execution example

4.5.2 Trace Record Encoding

Trace records should be encoded in such a way to minimize storage requirements due to buffering inside the trace module and trace port bandwidth requirements. The proposed mechanism requires four main types of trace records as shown in Table 4.12 and the software debugger has to be able to distinguish between them. The trace record length depends on the event type and can vary from several bits to several dozen of bits. Having a fixed number of bits in the trace record for *bCnt* and *iCnt* counter values would not be an optimal solution because the distance between two consecutive branch predictor mispredictions may vary widely between programs, as well as within a single program as it moves through different program phases. Typical values found in *bCnt* and *iCnt* counters are also heavily influenced by the misprediction rate, which is a function of the type and organization of the branch predictor. Thus, finding a theoretical optimal solution for encoding is not feasible. Rather, we opt for an empirical approach in determining trace record formats.

We employ a variable-length encoding scheme that minimizes trace record lengths for the most frequent *bCnt* and *iCnt* values. All trace records start with a header field, followed by a variable length field that carries the *bCnt* counter value. The header (*bh*) has variable length (*bhLen*) and it always ends with a zero bit, i.e., *bh*=′111...10′. Its length determines the length of the *bCnt* counter field as follows: (*bSize+(bhLen-1)*bStepSize*). The single-bit header, *bh*=′0′, specifies *bSize* bits in the *bCnt* counter field (encoding values from 0 to 2^{bSize}-1). The two-bit header, *bh*=′10′, specifies *bSize+1*bStepSize* bits in the *bCnt* counter field (encoding values from 0 to $2^{bSize+bStepSize}$ - 1), three-bit header, *bh*=′110′, specifies *bSize+2*bStepSize* bits (0 to $2^{bSize+2*bStepSize}$ - 1), and so on.

A trace record emitted on a direct branch outcome misprediction event consists of the header (*bh*) and the *bCnt* counter field (Figure 4.21*a*). A similar trace record format is used for indirect conditional branches. If an indirect branch is predicted as taken, but it is actually not taken, a trace record with the format shown in Figure 4.21*b* is used. An additional one-bit field *O* carries information that the outcome of the branch is not correct. A similar format shown in Figure 4.21*c* is used for indirect conditional branches that are predicted as not taken, but are actually taken. Indirect unconditional mispredictions are encoded as shown in Figure 4.21*d*. The last two trace records shown in Figure 4.21*c* and Figure 4.21*d* carry information about the *bCnt* counter and the branch target address *TA*. A naïve approach would be to just append an additional 32-bit field with the target address to the original trace record. An alternative approach is to encode only the difference between subsequent target addresses. The trace module maintains the previous target address (*PTA*) – that is, the target address of the last mispredicted indirect branch. When a new target misprediction event is detected, the trace module calculates the difference *diffTA* as follows: *diffTA* = *TA* – *PTA*, where the *TA* is the target address of the current branch. The trace module then updates the *PTA*, *PTA*=*TA*. By profiling the absolute value of the *diffTA*, |*diffTA*|, for several programs with a significant number of indirect branches, we find that we can indeed shorten trace records by using difference encoding.



a. Direct Branch Outcome Misprediction
bh
bCnt
1...10
xx...xx
bhLen
bSize+ (bhLen-1)*bStepSize
b. Indirect Cond. Branch Outcome Misprediction (NT)
bh
bCnt
O
1...10
xx...xx
0
bhLen
bSize+ (bhLen-1)*bStepSize
1
c. Indirect Conditional Branch Misprediction (T)
bh
bCnt
O
th
|diffTA|/TA
ts
1...10
xx...xx
1
1...10
xx...x
0
bhLen
bSize+ (bhLen-1)*bStepSize
1
thLen
taSize+ (thLen-1)*taStepSize
1
d. Indirect Unconditional Branch Misprediction
bh
bCnt
th
|diffTA|/TA
ts
1...10
xx...xx
1...10
xx...x
0
bhLen
bSize+ (bhLen-1)*bStepSize
thLen
taSize+ (thLen-1)*taStepSize
1
e. Exception Event
bh
bCnt
eh
iCnt
ETA
0
00...0
11...10
xx...xx
xx...x
1
bSize
ehLen
eSize+ (ehLen-1)*eStepSize
32

Figure 4.21 Trace record formats for branch misprediction events and exceptions

We employ variable encoding for the difference/target address field (|diffTA|/TA). Its length is specified by the number of header bits. We adopt the following scheme: a single header bit (th = ′0′) specifies taSize bits in the |diffTA|/TA filed. The two-bit header (th = ′10′) specifies taSize+1*taStepSize bits, three-bit header (th = ′110′) specifies taSize+2*taStepSize bits, and so on. If the |diffTA|/TA field requires less than 32 bits, we also need to provide information about the sign bit (*ts*) of the difference; otherwise, the whole 32-bit address is sent (without the sign bit).

Figure 4.21*e* shows the format of trace records used to report exception events. An exception trace record includes information about the *iCnt* counter and the starting address of the exception handler (ETA). It is an extension of the base format used for direct conditional branch mispredictions. The *bh* field indicates the shortest *bCnt* field of *bSize* bits. The *bCnt* field consists of all zeros indicating that this is an exception event trace record. The next two fields, the exception header (*eh*) and the instruction counter (*iCnt*), are used to specify the number of instructions since the last branch predictor misprediction event. We use the same variable encoding as before - the *ehLen*-bit header specifies the *eSize+(ehLen-1)*eStepSize* bits in the *iCnt* field. Finally, the last portion of the message includes the whole exception address (ETA). Note: we could use the same differential encoding described for indirect branches (eth and diffETA/ETA fields), but because of the low frequency of exception events, we opt for encoding the whole exception address of 32-bits**.**

Determining the trace record parameters such as *bSize*, *bStepSize*, *taSize*, *taStepSize*, *eSize*, and *eStepSize* depends on benchmark profiles and characteristics of the branch predictor. To determine suitable values for trace record parameters, we profiled the behavior of MiBench benchmarks and analyzed the probability density function for the minimum bit-length of the *bCnt* counter. We show that our proposed variable encoding scheme indeed reduces the size of the output trace and outperforms any fixed-length encoding scheme. Next, we analyzed several combinations of (*bSize*, *bStepSize*) pairs ($bSize \in [2..6]$ and $bStepSize \in [1..6]$) to determine an optimal combination that results in the shortest program trace across our benchmark suite. The results of this analysis indicate that the total trace size has a minimum when *bSize*=3 and *bStepSize*=2 for *bTMBP* and *sTMBP* configurations and *bSize=3* and *bStepSize*=1 for *tTMBP* configuration.

Similarly we analyzed the minimum bit-length of the |*diffTA*| field. The results clearly indicate that upper address bits of the subsequent mispredicted indirect branches rarely change, thus we can encode the difference *diffTA* instead of the whole target address. We analyze several combinations for parameters *taSize* and *taStepSize*. The results indicate that *taSize*=8 and *taStepSize*=6 give the shortest trace for the *tTMBP* configuration, and *taSize*=12, *taStepSize*=4 for *sTMBP* and *bTMBP*.

In spite of a relatively low frequency of exception events, we analyze the profile for *iCnt* counters in order to determine the parameters *eSize* and *eStepSize*. The profiles for software exceptions indicate that all *iCnt* values can be encoded with a 2-bit field. Thus, we adopt that *eSize=2* and *eStepSize=4*.

4.5.3 Trace Port Bandwidth Analysis

In this section we evaluate the effectiveness of the proposed tracing mechanism by measuring the average trace port bandwidth (TPB) for several branch predictor configurations. We consider three TMBP configurations, *bTMBP*, *sTMBP*, and *tTMBP*.

The base TMPB configuration (*bTMPB*) includes a 64-entry 2-way set associative iBTB, an 8-entry RAS for indirect branch target prediction, and a 512-entry GSHARE global outcome predictor. Each entry in the iBTB includes the tag field and the target address. The tag and iBTB index are calculated based on the information contained in a path information register (PIR). We assume a 13-bit PIR that is updated by relevant branch instructions as follows: PIR[12:0]=((PIR[12:0]<<2) xor PC[16:4]) | Outcome. The iBTB tag and index are calculated as follows: iBTBTag = PIR[7:0] xor PC[17:10] and iBTBIndex = PIR[12:8] xor PC[8:4]. The outcome predictor index function is GSHAREIndex = BHR[8:0] xor PC[12:4], where the BHR register keeps the outcome history for last 8 conditional branches. The *sTMBP* configuration includes a smaller, 32-entry, iBTB and the *tTMBP* does not include iBTB at all.

Table 4.13 Trace port bandwidth for different TMBP organizations

Test	**tTMBP**	**sTMBP**	**bTMBP**
adpcm_c	0.001	0.001	0.001
bf_e	0.011	0.009	0.009
cjpeg	0.062	0.044	0.04
djpeg	0.035	0.024	0.021
fft	0.166	0.096	0.091
ghostscript	0.513	0.230	0.127
gsm_d	0.013	0.013	0.013
lame	0.028	0.029	0.029
mad	0.035	0.033	0.033
rijndael_e	0.078	0.070	0.016
rsynth	0.023	0.021	0.021
sha	0.029	0.02	0.022
stringsearch	0.303	0.183	0.164
tiff2bw	0.013	0.011	0.007
tiff2rgba	0.015	0.011	0.008
tiffdither	0.062	0.062	0.062
tiffmedian	0.009	0.008	0.007
Average	**0.076**	**0.047**	**0.036**

Table 4.13 shows the results of trace port bandwidth analysis for three configurations of the proposed trace module (*tTMBP*, *sTMBP*, and *bTMBP*). We can see that the proposed technique requires a very small trace port bandwidth. The smallest *tTMBP* configuration requires only 0.0764 bits/ins. The *sTMBP* and *bTMBP* configurations benefit from the indirect branch target buffer requiring only 0.0467 bits/ins and 0.0356 bits/ins, respectively. When compared to 1 bit/ins that is the typical bandwidth of the commercial state-of-the-art trace modules, the proposed technique with the *bTMBP* configuration provides improvement of over 28 times.

4.5.4 Hardware Implementation and Complexity

To estimate the size of the proposed trace module, we need to estimate the size of all structures inside the trace module, including the outcome predictor, RAS, iBTB, PIR, BHR, the trace encoder, and the trace output buffer. The estimation of the size of predictor structures is straightforward. For iBTB and RAS, we implement an enhancement to reduce their complexity. We find that uppermost 12 bits of the indirect branch targets remain unchanged relative to the previous target in 99.99% of cases. Consequently, we can use a last value (LV) predictor for the upper 12 bits of the target address, and keep only the lower 18 address bits in the iBTB address entry (the last two bits are always 0 in ARM architecture). A miss in the LV predictor causes a whole target address to be included in the trace record. This way we reduce the complexity significantly with negligible degradation in the TMBP's iBTB hit rates.

To determine the size of the trace output buffer, we use a cycle-accurate processor model to find the maximum number of bits in this buffer at any point in time during benchmark execution. We assume the trace buffer is emptied through the trace port at the rate of a one bit per processor clock cycle. The worst case happens during the TMBP warm-up when we experience a number of consecutive mispredictions. We find that a buffer of 128 bits ensures that the processor is never stalled due to tracing and that no trace records are lost.

The estimates for hardware complexity measure in logic gates for three configurations are as follows (assuming 1.5 logic gates per memory bit): *tTMBP* requires 2,800 gates, *sTMBP* requires 4,000, and *bTMBP* requires slightly over 5,200 gates.

4.6 Comparative Analysis

In this section we compare the performance of the proposed compression techniques (DMTF, SDC-LSP, and TMBP) with several alternative approaches. We show the average trace port bandwidth requirements and complexity. For trace port bandwidth comparison, we use four different schemes described below. **BASE** is the output from the stream detector, and it shows us the compression rate after transforming the instruction addresses into partial stream descriptors (SA/-, SL). **HWLZ** is a hardware implementation of Lempel-Ziv compression algorithm that is specifically tailored to program execution traces [60]. **SW-GZIP** is the software compression utility *gzip* that implements the Lempel-Ziv compression algorithm, and is used here to underline the effectiveness of all proposed schemes. This algorithm uses large memory buffers and its implementation in a hardware trace module would be cost-prohibitive. **NEXS** implements a simple trace reduction technique inspired by the NEXUS standard [9]. The starting address from the incoming stream descriptor is XORed with the starting address from the previous stream descriptor, producing *DiffSA = Incoming.SA[31:0] xor*

Previous.SA[31:0]. The difference is split into groups of 6 bits, *DiffSA[5:0]*, *DiffSA[11:6]*, *DiffSA[17:12]*, etc. The leading zeros in the *DiffSA* are not sent to the trace port, thus reducing the trace port bandwidth. For example, if the *DiffSA[31:6]* consists only of zeros, then only the DiffSA[5:0] is sent to the trace port, together with a 2-bit header indicating that this is the terminating byte for the stream address field. The SL field is always sent to the trace port without further reduction.

Trace port bandwidth results are given in Table 4.14 and a summary for several configurations of proposed compression methods is given in Figure 4.22. The complexity of these configurations measured in the number of logic gates is given in Figure 4.23.

Table 4.14 Trace port bandwidth evaluation: A comparative analysis

Test	BASE	NEXS	HWLZ 256	HWLZ 1024	eDMTF (64,4)	eDMTF (192,4)	bSDC 128	eSDC 128	rSDC 128	tTMBP	sTMBP	bTMBP	SW-GZIP
adpcm_c	0.150	0.149	0.024	0.025	0.001	0.001	0.019	0.001	0.001	0.001	0.001	0.001	0.001
bf_e	4.913	4.010	0.354	0.367	0.279	0.284	0.357	0.345	0.345	0.011	0.009	0.009	0.038
cjpeg	0.790	0.752	0.431	0.138	0.089	0.091	0.131	0.088	0.088	0.062	0.044	0.04	0.050
djpeg	0.390	0.366	0.230	0.176	0.055	0.052	0.075	0.053	0.054	0.035	0.024	0.021	0.019
fft	1.895	1.554	1.921	1.036	1.012	0.201	0.616	0.542	0.554	0.166	0.096	0.091	0.065
ghostscript	1.814	1.578	1.394	0.187	0.299	0.306	0.232	0.216	0.216	0.513	0.230	0.127	0.038
gsm_d	0.621	0.567	0.152	0.151	0.058	0.040	0.086	0.051	0.052	0.013	0.013	0.013	0.009
lame	0.452	0.391	0.171	0.148	0.110	0.113	0.102	0.090	0.109	0.028	0.029	0.029	0.040
mad	0.785	0.668	0.268	0.144	0.135	0.147	0.129	0.116	0.117	0.035	0.033	0.033	0.042
rijndael_e	1.013	0.840	0.043	0.038	0.088	0.096	0.192	0.183	0.185	0.078	0.070	0.016	0.013
rsynth	0.883	0.747	0.271	0.247	0.184	0.108	0.175	0.147	0.149	0.023	0.021	0.021	0.018
sha	0.602	0.567	0.441	0.036	0.049	0.049	0.101	0.074	0.074	0.029	0.02	0.022	0.005
stringsearch	2.157	1.932	1.962	1.135	0.825	0.387	0.472	0.412	0.416	0.303	0.183	0.164	0.104
tiff2bw	0.668	0.654	0.146	0.137	0.061	0.011	0.104	0.030	0.031	0.013	0.011	0.007	0.006
tiff2rgba	0.349	0.330	0.160	0.095	0.050	0.006	0.045	0.012	0.012	0.015	0.011	0.008	0.005
tiffdither	0.692	0.659	0.573	0.301	0.197	0.166	0.190	0.158	0.159	0.062	0.062	0.062	0.080
tiffmedian	0.380	0.374	0.081	0.073	0.039	0.012	0.066	0.027	0.028	0.009	0.008	0.007	0.007
Average	**1.055**	**0.907**	**0.446**	**0.233**	**0.194**	**0.119**	**0.174**	**0.146**	**0.150**	**0.076**	**0.047**	**0.036**	**0.031**

The average trace port bandwidth required for the NEXS scheme is 0.907 bits/ins, ranging from 0.149 bits/ins (*adpmc_c*) to 4.01 bits/ins (*bf_e*). This relatively small improvement compared to the BASE scheme is due to the fact that the number of indirect branches is small, resulting in a small number of trace records that include a full stream descriptor. Another reason is the relatively high overhead in header bits.

The average trace port bandwidth required for the NEXS scheme is 0.907 bits/ins, ranging from 0.149 bits/ins (*adpmc_c*) to 4.01 bits/ins (*bf_e*). This relatively small improvement compared to the BASE scheme is due to the fact that the number of indirect branches is small, resulting in a small number of trace records that include a full stream descriptor. Another reason is the relatively high overhead in header bits.

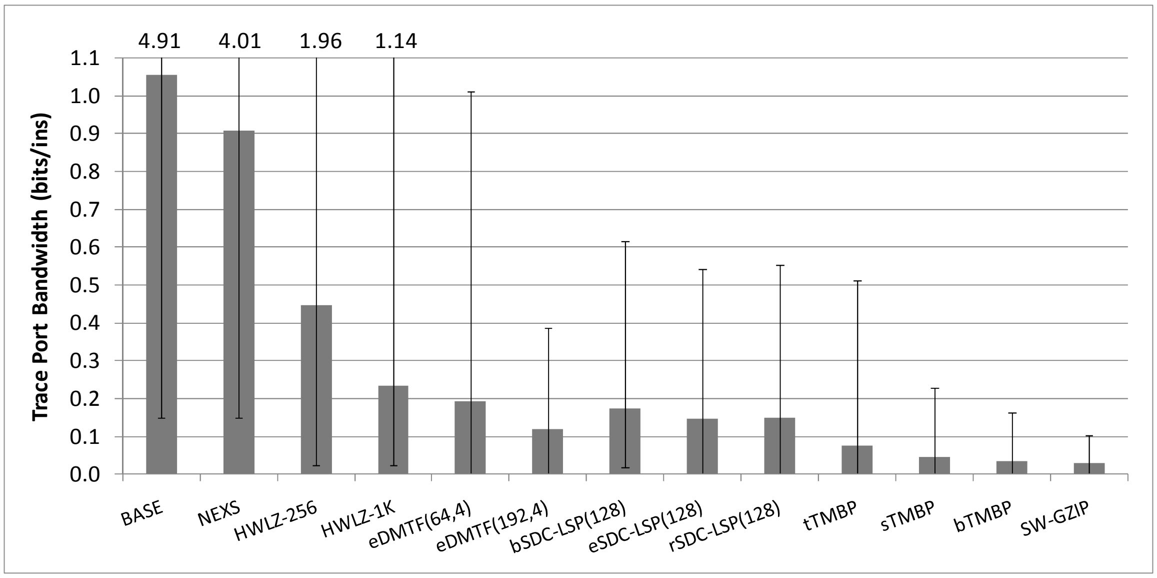


Figure 4.22 Trace port bandwidth evaluation for proposed and related mechanisms

HWLZ with a search buffer with 256 12-bit entries (HWLZ-256) achieves average trace port bandwidth of 0.446 bits/ins, ranging from 0.024 bits/ins to 1.96 bits/ins. The drawback of this scheme is a relatively high cost in both trace module complexity and compression time. Also, this scheme has very poor compression on certain programs (e.g., *fft* and *stringsearch* require more than 1.9 bits/ins). HWLZ-1K shows the results for the same scheme when the search buffer size is increased to 1024 entries. We see that certain programs are still poorly compressed (e.g., *fft* and *stringsearch* require more than 1 bits/ins), requiring wider trace ports.

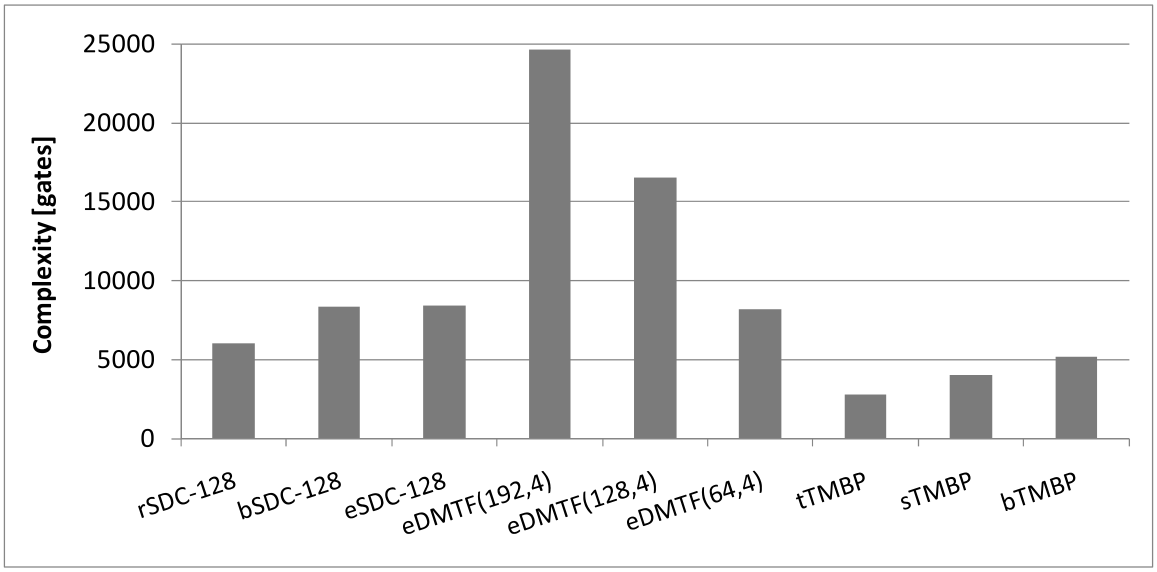


Figure 4.23 Implementation complexity of proposed tracing mechanisms

The best DMTF configuration, eDMTF (192, 4), achieves trace port bandwidth of 0.12 bits/ins on average, ranging from 0.001 to 0.39 bits/ins. Its estimated complexity is equivalent to 25,000 logic gates, which is half of the complexity reported for the HWLZ trace compressor (HWLZ-256). The smaller configuration, DMTF (64, 4), achieves 0.2 bits/ins on average (ranging from 0.001 to 1) at the cost of 8,200 logic gates.

The rSDC with 128 entries requires 0.15 bits/ins of trace port bandwidth, ranging from 0.0013 to 0.551 bits/ins at the cost of 6,061 logic gates. The bSDC requires 0.174 bits/ins, ranging from 0.019 to 0.616 bits/ins at the cost of 8,311 logic gates. The eSDC requires 0.146 bit/ins, ranging from 0.001 to 0.542 bits/ins at the cost of 8311 logic gates. The worst performing benchmark requires less than 1 bit/ins in the rSDC-128 scheme, allowing us to trace the program execution through just a single bit on the trace port (e.g., a JTAG port is sufficient).

The *tTMBP* configuration requires 0.0764 bits/ins on average at the trace port, ranging from 0.0013 to 0.551 bits/ins. The *sTMBP* and *bTMBP* configurations benefit from the indirect branch target buffer requiring only 0.0467 bits/ins. and 0.0356 bits/ins. on average, respectively. When compared to 1 bit/ins that is the typical bandwidth of the commercial state-of-the-art trace modules, the proposed technique with the *bTMBP* configuration provides improvement of over 28 times. We can also observe that the compression ratio achieved by the *bTMBP* configuration is close to that achieved by the software *gzip* utility, which further underscores the strength of the proposed mechanism. Note: The SW-GZIP compresses the partial stream descriptors only, not the whole stream of instruction addresses. This is more realistic for comparison purposes as the proposed compressor mechanisms also compress stream descriptors, although this approach favors the *gzip.* The estimates for hardware complexity for these three configurations are as follows: *tTMBP* requires 2,800 gates, *sTMBP* requires 4,000, and *bTMBP* requires slightly over 5,200 gates.

CHAPTER 5

COMPRESSION OF DATA ADDRESSES

Data address traces are widely used in trace-driven computer architecture simulators for evaluation and optimization of a processor's caches or the whole memory subsystem. In multi-core systems, data address traces offer valuable information about shared memory performance. Correlated traces offer insights into the order of memory accesses and into the number of collisions on the memory bus, and thus allow system designers to optimize the memory subsystem.

The Nexus standard [9] includes data addresses as a part of data trace. Although data values, already included in the data trace, are sufficient to reconstruct the program execution in a fully functional offline simulator, data addresses offer an additional level of security into correct reconstruction and can be used for trace synchronization purposes. Trace synchronization is needed when a part of the data trace is dropped due to the saturation of available bandwidth or due to errors in the communication channel. When a fully functional simulator is not available in the software debugger, data address traces are necessary to capture memory access patterns, which are especially important in the debugging of multi-core systems. The importance of data address traces is further emphasized by ever-tightening time-to-market pressures – system designers may not have enough time to develop sophisticated software debuggers that include fully functional simulators.

Data address trace compression is crucial for reducing the overall cost of the data tracing. Unfortunately, data addresses are much harder to compress than instruction address traces. The existing compressor structures are often very complex [57, 58, 71] as they include large caches for storing data addresses of recently executed memory referencing instructions. In this section, we introduce two cost-effective techniques to reduce the size of data address traces. Section 5.1 describes our data address filtering method which identifies and traces out only data addresses that cannot be inferred by a software debugger. The proposed trace module and the software debugger both maintain the list of general-purpose registers with known content in order to minimize the number of data addresses that are traced out. Section 5.2 presents our data address trace compression technique that focuses on the compression of high-order address bits that exhibit low variability.

5.1 Data Address Filtering

The proposed data address filtering method tries to identify data addresses that can be inferred by the software debugger during the program replay. Data addresses that can be inferred are thus not traced out of the chip which helps reduce trace port bandwidth requirements. Let us consider a memory referencing instruction specifying a *memory direct* addressing mode. In this case the data address is directly encoded in the instruction, and thus can be inferred by the software debugger during program replay.

However, the memory direct addressing modes are not that frequently used in modern RISC architectures. Fortunately, data addresses can be inferred in case of other addressing modes, e.g., *register indirect with displacement* which is the most frequently used in modern RISC architectures [72]. For example, the software debugger can maintain the content of general-purpose registers in the register file; the content is updated during a partial program replay offline using the data addresses that cannot be inferred and are thus traced out by the trace module. Similarly to the program execution tracing, both the trace module and the software debugger maintain similar structures that are synchronized during program tracing. It should be noted that here we do not consider a complete program replay (we do not trace data values, rather only data addresses). The proposed method requires a close integration of the trace module and processor, but in turn requires no additional storage resources. In Section 5.1 we describe a register validation mechanism that is used to identify what addresses need to be traced out and what addresses can be inferred by the software debugger that supports the proposed mechanism. In Section 5.2, we describe a mechanism for compression of only the high-order data address bits. Section 5.3 gives the results of a comparative analysis of the proposed compression techniques and several other data address compression techniques.

5.1.1 Data Address Filtering Details

The software debugger can perform a full program replay if all program inputs are made available. The software debugger maintains its local copy of the processor's register file that is updated during program execution. To be able to replay the program execution offline, the trace module needs to record and trace out all the data values brought to the processor chip from memory or input peripheral devices (loads from memory or peripheral devices). If we opt to trace only data addresses, the software debugger cannot perform a full program replay, rather only a partial replay is feasible. Similarly to the full program replay, the software debugger maintains its own local copy of the register file. However, in this case we may not know the content of individual registers; the content of some registers can be inferred from the previously traced data addresses, while the content of others is not known. Thus, the software debugger needs to know the list of registers with the known content and the list of registers with the unknown content. Thus, each register can be associated with a flag that indicates whether its value is known (Valid) or not known (Invalid). Based on this information, the software debugger determines whether a particular data address can be calculated locally or if data address trace records are required. Both the trace module and the software debugger need to maintain the status information about individual registers (Valid or Invalid). We call this mechanism a register validation.

There exists three possible ways for a register content to become valid (known):

- From a received data address, *DA*, e.g., an instruction {*LDR, R5, [R4,#2]*} loads a value into register *R5* from the memory address calculated *DA = R4+2*. Once the data address *DA* is received, *R4* can be set valid and its value is {*DA-2*}. Oppositely, if *R4* is valid, *DA* can be calculated locally by the software debugger, DA=R4+2, and thus needs not be traced out.
- From an instruction that specifies an immediate operand, e.g., an instruction {*MOV R4, #2*} sets value of register *R4* to *2*. Therefore, register *R4* is marked as Valid, and initialized *R4*=2.

- From an instruction that performs an operation on valid source registers, e.g., an instruction {*MOV R4, R5*} copies value from *R5* to *R4*; if *R5* is Valid, *R4* is marked as Valid too, and *R4* is loaded with the value of *R5*, *R4*=*R5*. This rule can be extended to instructions with multiple source operands (up to four in the ARM's instruction set). In general, an instruction destination register can be marked as Valid if it is the result of an operation on source registers that are all marked as Valid.

The software debugger is responsible for maintaining and updating its local copy of the register file during program replay and maintaining and updating the status of each register (Valid/Invalid). To keep the register file up to date, the software debugger performs operations that are reverse to those of address calculations performed inside the processor. It takes traced data addresses as an input and tries to determine the content of individual registers using the information available in the program binary. To be able to perform this task, the software debugger should be able to decode instructions from the binary.

Table 5.1 illustrates this process using a sequence of instructions. We assume that all registers are initially invalid (both in the trace module and software debugger). The first instruction loads an immediate constant #0 to register *R1*. Both the trace module and the software debugger set *R1* as valid and the software debugger sets *R1* to zero. The next instruction loads an operand from memory from the address calculated as *DA*=*R2*+5. The register *R2* is not valid and thus the data address, *DA,* is traced out of the processor. On the other side, the software debugger cannot calculate the data address and it expects it from the trace port; upon receiving the data address, *DA*, it calculates the value of register *R2*, *R2*=*DA*-5, and sets its valid bit. The register *R5* is loaded from memory, so its content is not known, so its valid bit has to be reset. The third instruction calculates *R4*, *R4*=*R2*+*R1*; the valid bit of register *R4* is set because both the input operands *R2* and *R1* are valid. The fourth instruction loads the register *R8* with an operand from memory at the address *DA*=*R2*+*R1*. The trace module does not need to trace out the *DA* because the software debugger can calculate it based on *R2* and *R1* that are both valid. The register *R8* is marked as invalid. Finally, the last instruction moves the content of *R7* to *R2*; the source register is invalid, and the register *R2* is thus marked as invalid.

Table 5.1 Partial software replay filtering example

Instruction	**CPU: Reg. File**	**CPU: Trace Output**	**Debugger: Trace Input**	**Debugger: Reg. File**
MOV R1, #0	R1: Valid	—	—	R1: Valid, R1=0
LDR R5, R2, #5	<u>R2: Valid</u> <u>R5: Invalid</u>	Trace DA (DA=R2+5)	DA expected	R2: Valid, R2=DA–5 R5: Invalid
ADD R4, R2, R1	R4: Valid (R1 and R2 are Valid)	—	—	R4: Valid, R4=R1+R2
LDR R8, R2, R1	R8: Invalid	— (no trace)	-- (DA is inferred, DA=R2+R1)	R8: Invalid
MOV R2, R7	R2: Invalid		—	R2: Invalid

Figure 5.1 shows a system view of the proposed data address filtering. The trace module (Figure 5.1 left) receives the relevant information from the processor core about the instruction type (memory referencing ins., load/store), the destination and source registers, and the data address itself. It also receives a signal indicating an entrance into an exception or system call routine. The register validation process depends on whether the instruction is a memory-referencing or a non memory-referencing instruction.

- For a non-memory referencing instruction, the trace module marks destination registers valid only if all source registers are valid. For the ARM architecture set, we can have up to 4 source registers and up to 3 destination registers.
- For a memory referencing instruction, source registers are used to calculate the data address. In the case that all source registers are valid, the data address is not traced as it can be inferred by the software debugger (the data address is filtered out); otherwise, the data address is traced out. In the case that all but one source register is valid, the only invalid register can be marked as valid in both the trace module and the software debugger (it can be calculated based on the data address and the content of other known source registers). For load instruction, the destination register is always invalidated as its value is unknown after the load.

The exception control signal serves to disable the control logic for register validation. If the software debugger is unable to trace the system calls or exception service routines, the trace module should disable the register validation unit in the trace module.

Figure 5.1 right shows the block diagram of the software debugger and its corresponding data address trace structures. It maintains a software copy of the register file status bits (valid/invalid) as well as the software copy of the register file. The debugger replays the program by fetching the instructions from the source binary. A replayed instruction is decoded in the functional simulator for the given architecture; it provides an identical set of signals as the processor to the trace module. The functional simulator is extended with the register calculation and validation unit (RCVU). The RCVU maintains and updates the content and the status of the general-purpose register based on the received data address and control signals from the functional simulator. As the final data address, RCVU selects one calculated in the instruction set simulator if all the source registers are valid, or takes the traced one if the data address cannot be calculated. To calculate a register value, the functional simulator replays all instructions that do not require traced data addresses. For register value calculations that require data addresses, RCVU implements the needed functionality.

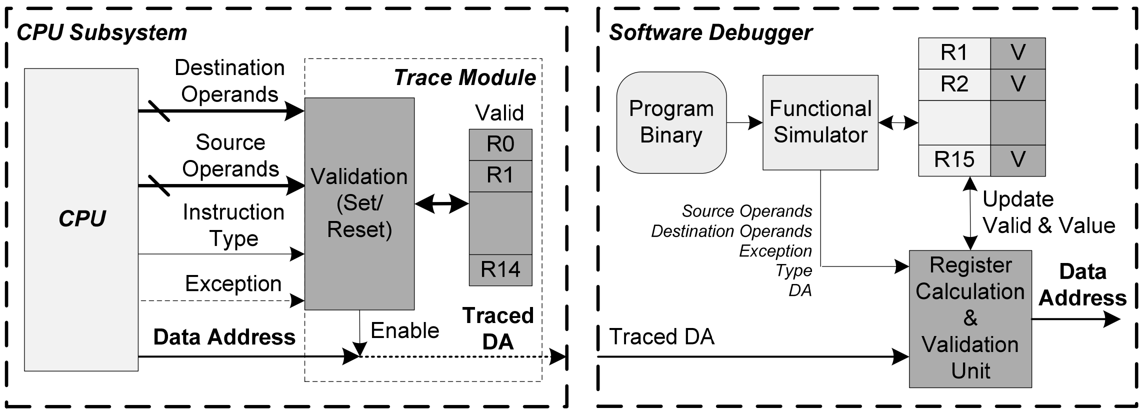


Figure 5.1 Data address trace filtering: A system view

5.1.2 Addressing Modes and Register Validation

The process of register validation is architecture dependant. In this section we specifically address the implications of the ARM addressing modes on the register validation process.

Effects of offset mode. In general, a data address is calculated as a sum of two source operands specified by the instruction word: a *base* and an *offset*. The *base* operand corresponds to the value of a specified base register. The *offset* operand can be an immediate value, the value of a specified index register, or the result of an operation performed on a specified index register. For example, in the instruction *{LDR R5, [R2, R4, lsl #2]}*, the base corresponds to the value of register *R2* (*R2* is a base register), while the offset corresponds to the value in register *R4* that is shifted for two bits left (*R4* is an offset register). Below we discuss the register validation process depending on the type of the address offset.

- In the case of an immediate *offset* (the *offset* field is specified in the instruction itself), the *base* register is always marked as valid as its value is known since the data address is known. For example, the validation process for the instruction {*LD R2, [R4, #5]*}, which loads *R2* from the memory address {*R4+5*}, will mark *R4* as valid; the software debugger will set its value to {*DA-5*}.
- In the case of a register *offset* (the *offset* field specifies a register and an operation on it), the *base* register can be validated only if the *offset* register is valid. For example, the register validation for the instruction {*LD R2, [R4, R5]*}, which loads *R2* from the memory address {*R4+R5*}, will mark *R4* as valid and the software debugger will set its value to {*DA-R5*} only if *R5* is valid.

To validate the *offset* register, the process is similar, but may involve additional steps in the case of a calculated offset (the offset is the result of an operation on the offset register). For example, if the data address is calculated as{$DA = R2 + f(R4)$}, where *R2* is the *base* and *R4* is the *offset* register and *f* is an operation on the index register, the software debugger can calculate *R4* as $f^{-1}(DA - R2)$. Thus, function *f()* must be reversible. For the ARM architecture, function *f()* is usually a shift operation on an offset register. Shift operations are as follows:

- *Shift left:* Shift to the left for a certain number of bits.
- *Shift right:* Shift to the right for a certain number of bits.
- *Ror:* Rotate register. Shift for one bit to the right is performed while the most significant bit of the register gets the value of the carry flag from the status register.
- *No Shift:* Original value is returned.
- *Zero:* This type specifies that the actual shift value should be zero.

Some shifting operations are not reversible. For example, the shift right or left operation can drop some bits and the reversed shift operation cannot recreate the original register values. In several special cases, it is possible to preserve these values, but for simplicity of implementation, we do not consider them. The rotate operation uses information from the status register, thus the validation process must be performed on the status register too. Memory referencing instructions in our benchmarks do not use the rotate operation, thus we do not consider them in the validation process. Note: The filtering mechanism is actually unusually complex in the ARM instruction set due to a rich set of addressing modes. In other architectures, such as MIPS [73], the implementation of the register validation is quite straightforward.

Effects of indexing. The ARM instruction set architecture uses three basic indexing modes: *Pre-*, *Post-* and *Auto- Indexing.* Both data address calculation and the register validation depend on the indexing type.

In the *Pre-Indexing* mode, the data address is calculated from the *base* register and the result of an operation on an *offset* register, e.g., for instruction {*LD, R5, [R4, R3]*}, which uses *Pre-Indexing* mode, the DA is calculated as {*DA = R4 + R3*}. Thus, the *DA* can be calculated only if both the *base* and *offset* registers are valid. The *base* register R4 is marked as valid if the register R3 is valid, and vice versa, the *offset* register R3 is marked valid if the register R4 is valid.

In the *Auto-Indexing* mode, the data address is also created from the *base* register and the result of an operation on the *offset* register. Upon completion of the instruction, the *base* register is updated, e.g., in instruction {*LD, R5, [R4, R3, lsl #2]!*}, which uses an *Auto-Indexing* mode, the data address DA is {*DA = R4 + (R3<<2)*} and *R4* takes the value of *DA,* {*R4 = DA*}. Thus, the *DA* can be calculated only if both the *offset* and the *base* registers are valid. The base register is always marked as valid.

In the *Post-Indexing* mode, the data address is calculated from the *base* register only. The *base* register is updated with information from both the *base* and the *offset* registers, e.g., in instruction {*LD, R5, [R4], R3, lsl #2*}, which uses the *Post-Indexing* mode, the data address is {*DA = R4 + (R3<<2)*} and *R4* is updated as follows: {*R4 <= R4 +(R3<<2)*}. Thus, the *DA* can be calculated only if the *base* register is known (*R4*). The *base* register can be validated only if both the *base* and *offset* registers are valid.

The ARM instruction set also includes multiple load and store instructions. These instructions specify a set of general-purpose registers that are loaded or stored from multiple consecutive memory locations defined by the starting address. These instructions use the immediate offset addressing mode. Once the starting data address is known, the remaining data addresses are calculated relative to this address (fixed stride).

5.1.3 Experimental Evaluation

The goal of our experimental evaluation is to determine the percentage of memory referencing instructions whose data addresses can be filtered out using the proposed mechanism. Our first step is to analyze the distribution of memory-referencing instruction types. We consider the following types of memory-referencing instructions:

- NotValid – the instruction condition is not satisfied and instruction is not executed (Note: ARM ISA supports conditional execution of all instructions).
- PreImm, PostImm, AutoImm – *Pre*-, *Post*-, *Auto*-indexing with immediate offset.
- PreReg, PostReg, AutoReg – *Pre*-, *Post*-, *Auto*-indexing with register offset.
- LDM – multiple loads.
- PC relative – PC as the *base* register with immediate offset; the resulting data address is always known.

Table 5.2 Distribution of load instruction types

Test	NotValid	PreImm	PostImm	AutoImm	PreReg	PostReg	AutoReg	LDM	PC relative
adpcm_c	0.0%	71.4%	0.0%	0.0%	28.5%	0.0%	0.0%	0.0%	42.9%
bf_e	0.0%	29.7%	8.0%	0.0%	29.4%	0.0%	0.0%	32.8%	5.8%
cjpeg	0.2%	67.1%	1.1%	0.0%	30.8%	0.0%	0.0%	0.9%	3.2%
djpeg	0.4%	52.2%	7.6%	0.0%	38.7%	0.0%	0.0%	1.1%	1.2%
fft	1.1%	75.1%	5.2%	0.1%	5.7%	0.0%	0.0%	12.8%	9.8%
ghostscript	2.1%	80.1%	4.9%	0.1%	5.6%	0.0%	0.0%	7.2%	8.2%
gsm_d	0.0%	97.3%	0.0%	0.0%	0.6%	0.0%	0.0%	2.1%	18.2%
lame	0.0%	71.4%	4.1%	0.0%	8.3%	0.0%	0.0%	16.2%	4.7%
mad	0.0%	87.8%	1.7%	0.0%	8.2%	0.0%	0.0%	2.3%	2.3%
rijndael_e	0.2%	52.5%	2.6%	0.2%	41.6%	0.0%	0.0%	2.9%	23.2%
rsynth	0.1%	94.1%	1.5%	0.0%	1.1%	0.0%	0.0%	3.2%	29.0%
sha	0.0%	20.1%	0.0%	0.0%	78.7%	0.0%	0.0%	1.1%	0.9%
stringsearch	4.1%	51.3%	14.7%	15.3%	5.5%	0.0%	0.0%	9.2%	5.4%
tiff2bw	0.0%	25.4%	74.3%	0.0%	0.1%	0.0%	0.0%	0.2%	0.0%
tiff2rgba	0.0%	85.8%	0.0%	0.0%	0.1%	0.0%	13.9%	0.2%	0.0%
tiffdither	0.0%	78.6%	11.1%	0.0%	9.3%	0.0%	0.0%	1.0%	13.0%
tiffmedian	0.0%	51.3%	34.2%	0.0%	14.4%	0.0%	0.0%	0.1%	0.5%
Average	**0.3%**	**73.5%**	**7.3%**	**0.0%**	**11.4%**	**0.0%**	**0.4%**	**7.1%**	**13.2%**

Table 5.2 shows the distribution of load instruction types. The information about load instruction types helps us identify the most common types that have the potential to benefit from the proposed filtering the most. The results indicate that the *Pre-Indexing* loads with an immediate offset dominate – overall 73.5% of the total number of loads. For several programs, the *Post-Indexed* with an immediate offset addressing is significant (e.g., *tiff2bw*, *tiffmedian*), and it is responsible for 7.3% loads on average. The *Pre-Indexing* mode is significant in the register offset loads (11.4% of the total number of loads). Finally, for some benchmarks (e.g., *adpcm_c*), a significant number of data addresses, 13.2% on average, use the PC as its base register, so they can always be validated.

Table 5.3 Data address filtering rates for load instructions

Test	PreImm	PostImm	AutoImm	PreReg	PostReg	AutoReg	LDM	Total
adpcm_c	40.0%	66.1%	86.1%	0.0%	nan	nan	0.0%	71.4%
bf_e	58.0%	26.4%	86.2%	8.6%	nan	nan	1.3%	28.1%
cjpeg	75.5%	82.8%	92.0%	25.0%	nan	nan	0.2%	62.4%
djpeg	95.3%	91.1%	90.6%	17.5%	nan	nan	0.4%	64.7%
fft	60.1%	78.5%	44.5%	6.1%	nan	nan	31.4%	63.4%
ghostscript	63.8%	61.6%	81.3%	0.4%	nan	nan	1.0%	62.5%
gsm_d	65.9%	69.0%	97.2%	0.6%	nan	nan	0.0%	82.3%
lame	75.9%	88.2%	96.3%	55.0%	nan	100.0%	12.6%	69.1%
mad	89.9%	2.3%	98.4%	24.4%	nan	nan	0.3%	83.2%
rijndael_e	51.9%	93.7%	100.0%	0.0%	nan	nan	0.0%	53.0%
rsynth	34.8%	92.7%	99.7%	42.9%	nan	nan	33.0%	64.7%
sha	92.1%	64.7%	85.4%	98.0%	nan	nan	19.7%	96.8%
stringsearch	56.5%	71.7%	87.6%	12.2%	nan	nan	1.0%	59.1%
tiff2bw	99.2%	99.9%	1.9%	4.5%	nan	0.0%	0.0%	99.5%
tiff2rgba	99.8%	30.4%	1.1%	2.9%	nan	nan	0.0%	99.5%
tiffdither	62.3%	99.9%	97.2%	0.0%	nan	nan	0.0%	73.1%
tiffmedian	98.7%	99.9%	97.1%	15.9%	nan	nan	0.0%	87.6%
Average	**65.2%**	**77.4%**	**87.7%**	**21.4%**	**–**	**–**	**9.6%**	**69.8%**

Table 5.3 shows the percentage of loads that can be filtered out, that is, the percentage of load instructions that do not require being traced out. The overall filtering rate is approximately 70%, which is an excellent result. However, it varies widely depending on the benchmark, e.g., from 28.1% for *bf_e* to 99.5% for *tiff2bw* and tiff2rgba. It also depends significantly depending on the load type. For example, the *Pre*- and *Post*- immediate offset indexing modes have a hit rate of 65.2% and 77.4% respectively, while the *Pre*- register indexing success rate is only 21.4% on average.

Table 5.4 shows the distribution of store instruction types. From left to right, the following columns are shown:

- NotValid – instruction condition is not satisfied and the instruction is not executed.
- PreImm, PostImm, AutoImm – *Pre*-, *Post*-, *Auto*-indexing with an immediate offset.
- PreReg, PostReg, AutoReg – *Pre*-, *Post*-, *Auto*-indexing with register offset.
- *STM* – multiple stores.

The results indicate that the *Pre-Indexed* with an immediate offset store type dominates – the overall 68.5% of the total number of stores belong to this type. Several benchmarks have a significant number of the *Post-Indexed* with the immediate offset store types (e.g., *tiff2bw* and *adpcm_c*), and it is responsible for 9.2% stores on average. Multiple stores are significant for *bf*_e and *fft* benchmarks. Surprisingly, several benchmarks have a significant number of stores that are not executed – 60% for *tiffmedian* and 50% for *adpcm_c*.

Table 5.4 Distribution of types of store instructions

Test	NotValid	PreImm	PostImm	AutoImm	PreReg	PostReg	AutoReg	STM
adpcm_c	49.9%	0.2%	49.8%	0.0%	0.0%	0.0%	0.0%	0.1%
bf_e	0.0%	25.9%	14.4%	0.0%	11.1%	0.0%	0.0%	48.6%
cjpeg	1.0%	58.1%	6.7%	0.0%	32.0%	0.0%	0.0%	2.3%
djpeg	0.1%	75.1%	18.1%	0.0%	2.4%	0.0%	0.0%	4.3%
fft	2.2%	50.9%	8.9%	6.9%	8.5%	0.0%	0.0%	22.5%
ghostscript	4.2%	73.7%	6.4%	4.1%	0.0%	0.0%	0.0%	11.6%
gsm_d	0.0%	94.8%	0.9%	0.0%	1.1%	0.0%	0.0%	3.3%
lame	0.2%	88.1%	2.4%	0.9%	2.9%	0.0%	0.0%	5.6%
mad	0.0%	72.7%	6.2%	0.0%	16.0%	0.0%	0.0%	5.1%
rijndael_e	0.0%	63.5%	12.4%	0.0%	12.4%	0.0%	0.0%	11.6%
rsynth	0.3%	84.1%	1.6%	5.0%	1.3%	0.0%	0.0%	7.7%
sha	0.0%	64.4%	0.0%	0.0%	34.3%	0.0%	0.0%	1.3%
stringsearch	0.8%	28.2%	6.8%	0.0%	56.7%	0.0%	0.0%	7.4%
tiff2bw	0.0%	49.9%	49.7%	0.0%	0.1%	0.0%	0.0%	0.4%
tiff2rgba	0.0%	76.3%	23.4%	0.0%	0.0%	0.0%	0.0%	0.3%
tiffdither	0.0%	63.2%	33.6%	0.0%	0.0%	0.0%	0.0%	3.1%
tiffmedian	60.0%	17.9%	10.0%	0.0%	11.9%	0.0%	0.0%	0.1%
Average	**7.2%**	**68.5%**	**9.2%**	**1.6%**	**4.7%**	**0.0%**	**0.0%**	**8.8%**

Table 5.5 Filtering results for store instructions

Test	PreImm	PostImm	AutoImm	PreReg	STM	Total
adpcm_c	99.8%	99.8%	70.0%	50.0%	0.4%	49.9%
bf_e	93.4%	22.7%	80.0%	48.2%	34.7%	49.7%
cjpeg	93.8%	95.0%	90.9%	16.5%	17.5%	66.5%
djpeg	95.7%	99.6%	75.0%	16.8%	90.8%	94.3%
fft	75.1%	64.7%	81.7%	22.5%	65.7%	66.4%
ghostscript	96.4%	52.5%	99.5%	44.1%	40.2%	83.2%
gsm_d	88.0%	0.0%	80.0%	0.5%	31.7%	84.5%
lame	90.3%	99.7%	66.4%	75.1%	89.0%	89.6%
mad	97.1%	47.1%	53.5%	25.7%	27.9%	79.0%
rijndael_e	96.3%	93.7%	80.0%	0.0%	40.1%	77.5%
rsynth	80.3%	99.2%	45.9%	100.0%	77.6%	78.7%
sha	98.0%	68.9%	86.7%	98.8%	33.5%	97.4%
stringsearch	96.0%	67.7%	80.0%	98.1%	45.3%	90.7%
tiff2bw	99.9%	99.9%	83.0%	67.7%	58.1%	99.7%
tiff2rgba	99.9%	99.9%	85.7%	51.3%	56.6%	99.8%
tiffdither	96.6%	95.9%	33.3%	48.9%	10.2%	93.6%
tiffmedian	99.9%	99.8%	36.4%	15.9%	58.7%	29.9%
Average	**91.3%**	**72.6%**	**67.6%**	**47.8%**	**55.3%**	**76.9%**

Table 5.5 shows the filtering rates for different benchmarks and store types. The overall filtering rate is approximately 77% ranging from 29.9% for *tiffmedian* to 99.8% for *tiff2rgba*. Again, the filtering rate varies widely depending on the benchmark and store type. For example, the *Pre*- and *Post-indexing* with an immediate offset addressing achieve filtering rates of 91.3% and 72.6% respectively, while the *Pre-indexing* with the register offset achieves only 47.8% on average. The main contributor to the low filtering rates for some benchmarks is the relatively small number of the filtered data addresses for multiple store instruction and the *Pre-indexing* with the register offset addressing.

Table 5.6 shows the total filtering rate for all memory referencing instructions and the number of data address bits per each instruction. The filtering rates range from 35.8% to 99.6%, and it is 72% on average. Some benchmarks (e.g., *bf_e* and *tiffmedian*) do not benefit much from the proposed filtering and still require a significant bandwidth on the trace port (6.57 bits/ins for *bf_e* and 5.19 bits/ins for *tiffmedian*). On other hand, several benchmarks benefit greatly from the proposed filtering (e.g., *tiff2bw* and *tiffrgba*) and require a very small trace port bandwidth of 0.058 bits/ins. The proposed filtering mechanism requires close integration with the processor core to be able to carry out register validation in hardware. However, it does not require any additional compression structures and its implementation cost is negligible. In addition, the proposed filtering scheme could be combined with the additional compression structures – e.g., the data addresses that need to be traced can be encoded using a variant of differential encoding similar to the Nexus standard.

Table 5.6 Filtering results and trace port bandwidth for all memory referencing instructions

Test	Total	bits/ins
adpcm_c	68.7%	1.457
bf_e	35.8%	6.565
cjpeg	63.4%	4.061
djpeg	72.7%	3.986
fft	64.4%	3.092
ghostscript	70.1%	3.591
gsm_d	83.2%	1.397
lame	76.4%	3.511
mad	82.3%	1.981
rijndael_e	57.2%	6.479
rsynth	68.3%	5.491
sha	97.0%	0.233
stringsearch	75.0%	2.204
tiff2bw	99.6%	0.058
tiff2rgba	99.6%	0.075
tiffdither	78.4%	1.913
tiffmedian	66.7%	5.189
Average	**72.0%**	**3.566**

5.2 Adaptive Data Address Cache

The existing hardware compressors for data address traces rely on either predictor structures [57] or cache-like structures that can detect regular strides [56]. A system designer typically faces a challenging task in making design trade-offs between the size of the compressor structures and the compression ratio. Larger structures help achieve higher compression ratios, but may be costly or not practical to implement. The existing techniques demonstrate good compression ratios for benchmarks with memory-referencing instruction that exhibit regular data address patterns (fixed data addresses or data addresses with fixed address strides). However, the compression of data addresses for memory-referencing instructions with irregular patterns remains a challenging task.

In designing data address trace compressors, we should exploit common characteristics of memory access patterns. For example, the simplest approach is to apply a differential encoding of the stream of data addresses. Instead of tracing out a complete data address, we could trace only the difference between subsequent data addresses. It can help reduce the number of traced bits at a minimal implementation cost. To better exploit program characteristics, we can use caches or predictors that try to capture data address patterns and/or temporal and spatial locality of data for each memory referencing instruction. For example, a data address stride cache can be trained to detect a fixed data address stride. However, it is very challenging to design a compressor structure that will exhibit a stable performance for all benchmark programs or throughout the entire execution of a particular benchmark. Compressor structures often perform well when a desirable type of memory access patterns prevails; however, they perform poorly on other types of memory access patterns.

A nearly constant compression ratio is an important feature of trace compressors. Designers use this information to determine minimum buffering and trace port bandwidth requirements so that no trace records are lost and tracing is done unobtrusively in real-time. Figure 5.2*a* illustrates trace port bandwidth requirements for two hypothetical compressors A and B. Compressor A requires higher trace port bandwidth on average but demonstrates a nearly constant compression ratio. On the other side, compressor B requires lower trace port bandwidth than compressor A, but its bandwidth varies widely over time. Compressor B thus will require much larger trace buffers to smooth out these peaks in the bandwidth; without these buffers some trace records are likely to be lost. Figure 5.2*b* illustrates the behavior of compressors A and B: while compressor A reduces the number of address bits for all instructions, compressor B performs very well for some data addresses while for the others require complete data addresses.

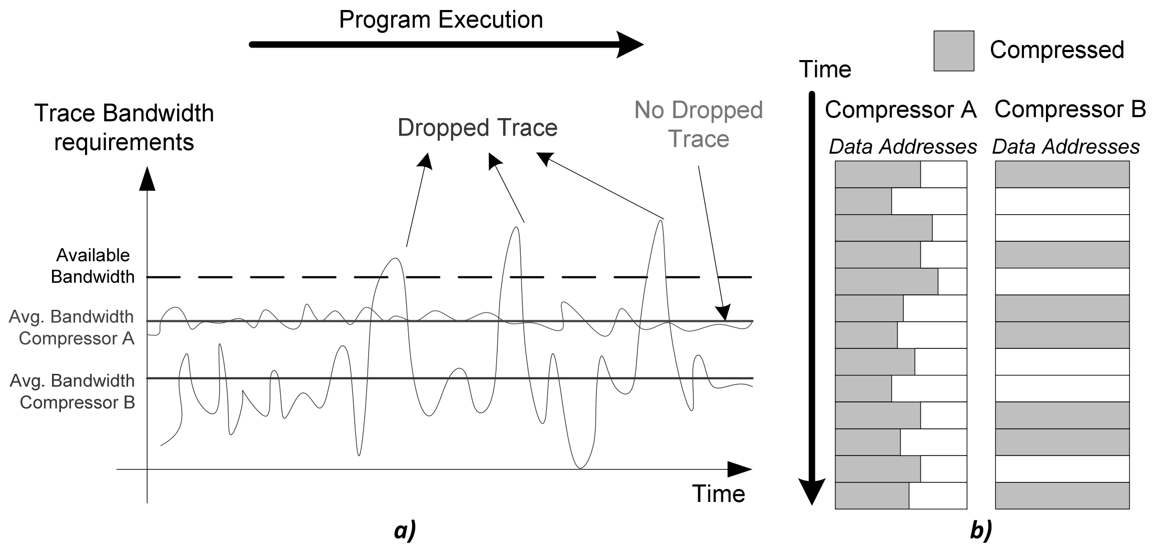


Figure 5.2 Trace bandwidth requirements (a) and compressor effectiveness (b)

In this section we introduce a data address trace compression technique that tries to exploit redundancy in programs caused by temporal and spatial locality. The proposed technique shares the common tracing mechanism with previously described compressors. The trace module and software debugger maintain identical compressor structures (the trace module in the hardware and the software debugger in the software) that are updated during program execution and program replay, respectively, using the same set of update policies. We propose a cache-like compressor that keeps the most recently used data addresses. The data address cache (DAC) is indexed based on the program counter. If we find an incoming data address in the DAC, we need to trace out only several bits to

indicate a DAC hit event. In the case of a DAC miss, the whole data address needs to be traced out. However, data addresses rarely stay constant, so the DAC would have a relatively low hit rate. To alleviate this problem, we modify the DAC to allow lookups that consider only upper address bits of the data addresses that rarely change.
Section 5.2.1 discusses redundancy in high-order address bits. Section 5.2.2 introduces the adaptive data address cache, while Section 5.2.3 gives the results of experimental evaluation.

5.2.1 Variability of High-Order Data Address Bits

In this section we explore the variability of high-order data address bits. We can identify two types of variability of high-order address bits: a global and a local. The global variability considers the sequence of data addresses as they appear during program execution. The local variability considers variability of data addresses for individual memory-referencing instructions.

Let us first discuss the global locality of high-order data address bits. We consider a history buffer that keeps *n* most recent unique data addresses encountered during program execution. We perform a lookup in this buffer with an incoming data address to find if there is a data address match. The lookup is performed on high-order address bits only. For example, we consider upper address bits DA[32:SHIFT], assuming 32-bit data addresses. Table 5.7 shows the hit rate in the history table for each benchmark as a function of the history table size (16, 32, and 64 entries) and the parameter SHIFT (0, 4, and 8).

Table 5.7 Hit rate in a data address history table

	SHIFT = 0			SHIFT = 4			SHIFT = 8		
Test \ Size	**16**	**32**	**64**	**16**	**32**	**64**	**16**	**32**	**64**
adpcm_c	0.765	0.839	0.887	0.967	0.976	0.978	0.999	1.000	1.000
bf_e	0.751	0.753	0.761	0.816	0.826	0.850	0.976	1.000	1.000
cjpeg	0.563	0.668	0.694	0.871	0.900	0.954	0.990	0.998	0.999
djpeg	0.552	0.697	0.718	0.820	0.901	0.928	0.983	0.991	0.996
fft	0.575	0.750	0.801	0.852	0.887	0.943	0.965	0.993	1.000
ghostscript	0.444	0.592	0.841	0.727	0.899	0.988	0.945	0.997	0.999
gsm_d	0.920	0.956	0.966	0.986	0.989	0.995	0.999	0.999	1.000
lame	0.279	0.393	0.549	0.710	0.774	0.862	0.974	0.987	0.997
mad	0.332	0.434	0.570	0.796	0.876	0.883	0.991	0.998	0.999
rijndael_e	0.285	0.439	0.526	0.608	0.652	0.680	0.886	0.944	1.000
rsynth	0.417	0.535	0.571	0.813	0.849	0.984	0.996	0.998	1.000
sha	0.448	0.684	0.689	0.910	0.949	0.994	1.000	1.000	1.000
stringsearch	0.343	0.406	0.513	0.730	0.825	0.860	0.955	1.000	1.000
tiff2bw	0.498	0.499	0.499	0.872	0.872	0.872	0.991	0.991	0.993
tiff2rgba	0.215	0.216	0.216	0.802	0.802	0.802	0.987	0.987	0.987
tiffdither	0.800	0.871	0.886	0.925	0.967	0.973	0.998	0.998	0.999
tiffmedian	0.682	0.691	0.699	0.886	0.897	0.907	0.971	0.982	0.988
Average	**0.523**	**0.578**	**0.64**	**0.817**	**0.863**	**0.916**	**0.977**	**0.991**	**0.998**

The results indicate that the hit rate increases with an increase of the SHIFT parameter, as expected. For a very small history table with 16 entries, the average hit rate increases from 52.3% with SHIFT=0 to 97.7% with SHIFT=8. Some benchmark programs see a significant increase in the hit rate with an increase of the SHIFT parameter (e.g., *rijndeal_e*), while others see more modest improvements (e.g., *gsm_d*). The results thus clearly indicate that there exists a lot of redundant information in a sequence of data addresses, and that the redundancy is higher if we focus on upper address bits only (larger SHIFT parameter).

The local variability of the upper-address bits is evaluated using a tagless cache structure that is addressed using the program counter (PC). We call this structure a data address cache or DAC. The DAC's index function should minimize the probability of having multiple memory-referencing instructions mapping into a single DAC entry. While PC-based tags in the DAC could reduce the problem of aliasing, they would significantly increase the implementation cost, and are thus not considered in our proposal. The DAC is a multi-way cache structure, and in our work we use 4-way set-associative DAC because it showed a good performance. A DAC lookup is performed as follows. The DAC controller receives the program counter and data address for the currently executing memory referencing instruction. A DAC set is calculated based on the program counter. We compare the incoming data address (DA) with the entries in the selected DAC set. If there is a match of the upper address bits, we say we have a DAC hit event. Otherwise, we have a DAC miss event.

Table 5.8 Hit rate in a data address cache

	SHIFT=0			SHIFT=4			SHIFT=8		
Test\Size	**32**	**64**	**128**	**32**	**64**	**128**	**32**	**64**	**128**
adpcm_c	0.507	0.507	0.521	0.944	0.944	0.949	0.997	0.997	0.998
bf_e	0.593	0.575	0.634	0.728	0.715	0.783	0.820	0.824	0.872
cjpeg	0.437	0.473	0.512	0.742	0.775	0.810	0.936	0.956	0.974
djpeg	0.342	0.368	0.386	0.700	0.726	0.739	0.920	0.915	0.918
fft	0.302	0.430	0.531	0.523	0.598	0.675	0.857	0.849	0.891
ghostscript	0.120	0.193	0.506	0.267	0.334	0.645	0.759	0.835	0.944
gsm_d	0.728	0.784	0.790	0.961	0.966	0.967	0.995	0.995	0.994
lame	0.147	0.320	0.539	0.273	0.412	0.615	0.848	0.861	0.880
mad	0.200	0.269	0.311	0.402	0.415	0.436	0.867	0.850	0.816
rijndael_e	0.050	0.047	0.046	0.265	0.236	0.218	0.509	0.470	0.421
rsynth	0.260	0.274	0.448	0.373	0.354	0.521	0.784	0.776	0.858
sha	0.088	0.101	0.124	0.716	0.711	0.750	0.978	0.987	0.995
stringsearch	0.058	0.112	0.224	0.465	0.525	0.631	0.774	0.786	0.841
tiff2bw	0.001	0.000	0.001	0.580	0.559	0.559	0.968	0.947	0.948
tiff2rgba	0.001	0.000	0.002	0.143	0.066	0.020	0.936	0.888	0.882
tiffdither	0.397	0.447	0.436	0.822	0.859	0.884	0.920	0.956	0.981
tiffmedian	0.392	0.389	0.390	0.682	0.677	0.678	0.914	0.907	0.907
Average	**0.301**	**0.363**	**0.471**	**0.515**	**0.548**	**0.650**	**0.853**	**0.859**	**0.887**

Table 5.8 shows the DAC hit rate for different sizes of the DAC (32, 64, and 128 entries) and different values of the SHIFT parameter (0, 4, and 8). Similarly to the global locality analysis, we can observe that the DAC hit rate increases with an increase of the SHIFT parameter. Several benchmark programs significantly benefit from increases of the SHIFT parameter (e.g., *tiff2bw* hit rate increases from almost 0% with SHIFT=0 to ~52% with SHIFT=4, and to 96.8% for SHIFT=8, assuming a DAC with 32 entries). Several other benchmark programs have little or no benefit when the SHIFT parameter increases from 0 to 4 (e.g., *tiff2rgba* hit rate increases from almost 0% when SHIFT=0 to 14% when SHIFT=4, and then to 93.6% when SHIFT=8).

The results for the global and local variability of the upper address bits indicate that different benchmark programs benefit from different values of the SHIFT parameter. An additional analysis indicates that an optimal value of the SHIFT parameter varies during the execution of a single benchmark, as the program moves through different execution stages. This conclusion motivates us to consider a DAC compressor structure that allows for adaptive value of the SHIFT parameter. We extend the DAC structure to include the field for the SHIFT parameter and an adaptive mechanism for its training. The compression ratio can be expressed as a function of the DAC hit rate and a value of the SHIFT parameter. The higher the DAC hit rate, the smaller number of DAC misses that require tracing out of an entire data address. Higher DAC hit rates are usually achieved for higher values of the SHIFT parameter. However, the larger value of the SHIFT parameter means the larger portion of the uncompressed data address, which in turn lowers the compression ratio. Thus, a detailed exploration of the design trade-offs is required to find good values for the DAC size, organization, and SHIFT parameter.

5.2.2 *Adaptive Data Address Cache*

The data address cache (DAC) keeps the most recent data addresses encountered during program execution. It is a set-associative structure, indexed by a hash function based on the program counter of the corresponding memory-referencing instructions. The DAC is a tagless structure to reduce the implementation complexity.

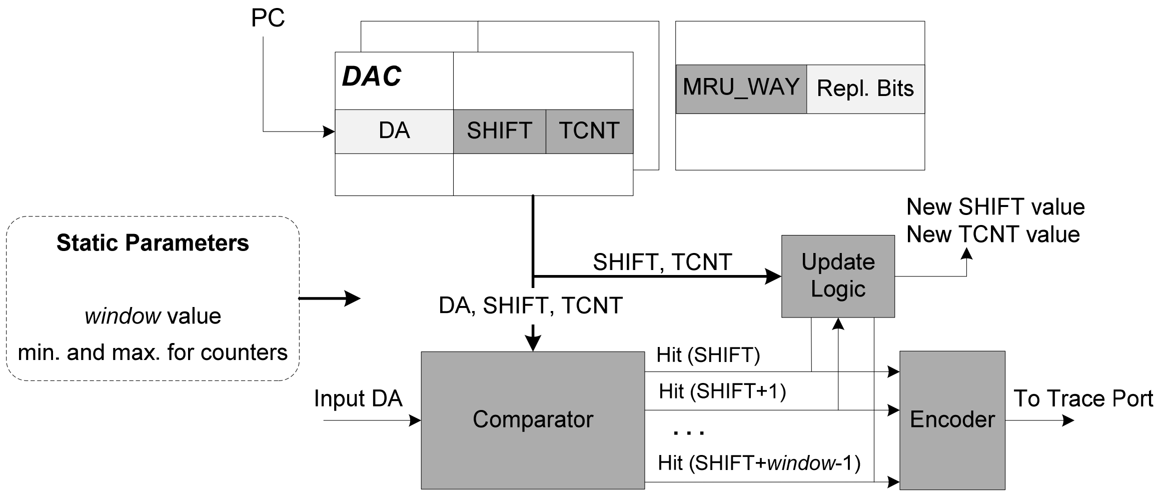


Figure 5.3 Adaptive Data Address Cache organization

Figure 5.3 illustrates a block diagram of the adaptive data address cache (ADAC). Each entry in the DAC keeps a full data address (DA). Additional fields such as SHIFT and TCNT are used to control DAC lookups. The SHIFT field keeps the current value of the SHIFT parameter which determines the number of lower data address bits that will be traced out with no compression. The TCNT field is a training counter that determines whether the current SHIFT value should be increased in order to increase the DAC hit rate, or decreased in order to reduce the number of bits that is traced out uncompressed. The TCNT is incremented by a miss and decremented by a hit in the DAC. When it saturates, the external logic decides whether to increase or decrease the SHIFT parameter. The DAC with these extensions is called the Adaptive Data Address Cache (ADAC).

Figure 5.4 describes the adaptive data address cache operation. The ADAC controller receives a (PC, DA) pair and calculates a set index iSet based on the program counter. It then performs a lookup into the selected ADAC. The selected bits of the incoming data address DA[31:SHIFT] are compared with the corresponding data addresses in the selected entry of the ADAC. The ADAC lookup differs from a regular comparison of DA[31:SHIFT] address bits. In addition to this comparison, it also involves the comparison of upper address bits with different lengths (DA[31:SHIFT+1], DA[31:SHIFT+2], ... DA[31:SHIFT+*window*-1]). The comparator gives the results of the comparison for *window* data addresses (SHIFT corresponds to j=0, SHIFT+1 to j=1, and so on). The variable *window* is a pre-determined fixed parameter. The outputs from the comparator are used to control the update mechanism for the SHIFT parameter.

The outcome of the multi-try comparison is used to update the TCNT (training counter) as follows. When an ADAC hit occurs with a *window* value of zero, the TCNT is decremented by TCNT_DEC. When an ADAC hit occurs also on a data address field with (SHIFT+j, where j>0), it indicates that the current SHIFT field could possibly be increased. Of course, making a premature decision about the SHIFT length may be detrimental for the compression ratio. Consequently, a training TCNT is used. When such a hit occurs, the TCNT is incremented by TCNT_INC (it is a miss regarding the current SHIFT). When the TCNT saturates, the value of the SHIFT is incremented (decrease the portion of the data address that will be compressed). When we have an ADAC miss (no *j* value gives a hit), the TCNT and SHIFT fields take the maximum possible values, TMAX and SHIFT_MAX.

```
Get the next (PC, DA)pair;
iSet = f(PC)
Lookup in the data addres cache with iSet (*)
if(DAC hit){
        if (WHit == 0){
            Decrement TCNT by TCNT_DEC
            if (TCNT == 0){
                DAC[iset][iway].SHIFT--; #Decrement SHIFT value
                DAC[iset][way].TCNT = TMAX;}
        } else {
            Increment TCNT by TCNT_INC
            if (TCNT == TMAX){
                DAC[iset][iway].SHIFT++; #Increment SHIFT value
                DAC[iset][way].TCNT = 0;}
        }
    if(DAC[iset].MRU_WAY == way)
        MRU Way Hit = 1;
    DAC[iset].MRU_WAY = way;
    } else{
        DAC[iset][way].SHIFT = SHIFT_MAX;
        DAC[iset][way].TCNT = TMAX; }
    //Encode outputs
    // (*) DAC Lookup
        for (i=0; i<Nways; i++)
            for (j=0; j<window; j++)
                if (DA >>(SHIFT + j) == DAC[iset][iway].DA >> (SHIFT + j)){
                    DAC hit = 1;
                    WHit = j;
                    way = i;
                    exit;}
```

Figure 5.4 Adaptive Data Address Cache operation

Finally, let us discuss encoding of the compressed trace records. Table 5.9 describes relevant events on ADAC and encoding of these events. The ADAC is a tagless cache, so to be able to recreate the trace record at the software debugger side, we need to trace out the way identifier in the case of an ADAC hit. Like in regular caches, we observe that data addresses that are most recently accessed tend to be more likely accessed again. We can exploit this property by relying on the ADAC set replacement bits to identify the most recently used entry in an ADAC set. Thus, a hit in the most recently used entry of an ADAC set can be encoded using just a single header bit ('1'). A hit in a non-MRU entry is encoded with a header bit '0' followed by a way.id field. In the case of an ADAC hit in a non-MRU entry with *j*>0, the trace record header includes a '0', followed by the most recently used way (to indicate that *j*>0), followed by the actual way identifier on which the hit is observed and the value of j (this information is used by the software debugger to train its own copy of the SHIFT parameter). Finally, a miss is encoded by the reserved unique header followed by a full data address (32 bits in our case).

Table 5.9 Data address trace record encoding for ADAC compressor

Hit/Miss	***MRU Way***	***MIN {j}***	***Trace Record***		***Trace Length (bits)***
			Header	***Data Address***	***Output Header***
Hit	Yes	Zero	"1"	DA[SHIFT-1:0]	1 + shift
Hit	No	Zero	"0" + "way.id"	DA[SHIFT-1:0]	1 + log2(Nway) + shift
Hit	--	> 0	"0" + "MRU.way.id" + "way.id" + "j"	DA[SHIFT-1:0]	1 + 2*log2(Nway) + log2(window) + shift
Miss	--	--	"0" + "MRU.way.id" + "way.id=0" + "j=0"	DA[31:0]	1 + 2*log2(Nway) + log2(window) + 32

5.2.3 Design Space Exploration and Trace Port Bandwidth Analysis

The design of the adaptive data address cache requires a thorough design space exploration to determine good values for the following parameters: the SHIFT field maximum and minimum limits (SHIFT_MAX and SHIFT_MIN), the training counter

TCNT maximum limit (TMAX), the TCNT increment and decrement steps (TCNT_INC and TCNT_DEC), and the value of the *window* parameter. Each of these parameters influences the compression ratio and design complexity to a certain degree and discerning an impact of each of them independently from the other parameters is a challenging task. We have explored the design space by considering trace port bandwidth (compression ratio) as well as the stability of the trace port bandwidth. We strive to find a configuration that will provide the minimum trace port bandwidth and yet exhibit very low variability at the trace port.

The total number of possible configurations is very large and here we present the results for the overall best performing configuration: *window*=4, SHIFT_MAX=12, SHIFT_MIN=0, TMAX=8, TCNT_DEC=1. For TCNT_INC, an ADAC miss actually sets TCNT to TMAX (which in return increments the SHIFT value). This TCNT_INC value favors a higher ADAC hit rate over having smaller values for the SHIFT parameter. This approach balances the trade-offs between the hit rate (the higher the hit rate, the less the number of costly misses) and the number of uncompressed bits (SHIFT value may be slightly higher than absolutely necessary). Figure 5.5 shows the average ADAC hit rates for different compressor schemes and configurations as a function of the data address cache size. We consider several compressors: four compressors based on the DAC with the fixed SHIFT parameters (SHIFT=0, SHIFT=4, SHIFT=8, SHIFT=12), and two compressors with adaptive data address cache for different values of the parameter *window* (*window*=1 and *window*=4). The number of entries in the DAC/ADAC is varied between 32 and 512 entries. The number of ways is fixed to four. The results for ADAC with *window*=1 are crucial to determine the effectiveness of the training mechanism for the SHIFT parameter. With the *window*=1, the hit rate is not inflated by possible matches on DA[31:SHIFT+*j*], where *j*>0. Figure 5.5 also shows the average value of the SHIFT parameter during the program execution. For example, the ADAC with 32 entries has the average SHIFT of 8.7, which is just slightly above the SHIFT value of 8. This configuration achieves the hit rate of 92.8% on average (ranging from 80% for *ghostscript* to 99% for *adpcm_c*). It should be noted that the DAC with the fixed SHIFT=8 achieves a lower hit rate of 85.3% on average (ranging from 50% for *rijndael_e* to 99% for *adpcm_c*). Hence, we may conclude that the proposed adaptive mechanism is indeed beneficial for the overall performance of the ADAC compressor.

The effects of the proposed mechanism are even more visible by allowing multi-try comparison. For example, the ADAC with *window*=4 achieves higher hit rates. The ADAC with 128 entries achieves the hit rate of 98.2% on average with the average SHIFT value of just 6.5. This result clearly illustrates the benefit of adaptive mechanisms – more high-order data address bits can be compressed on average, without reducing the ADAC hit rate. In addition, this approach is applicable to almost all data addresses because we achieve consistently high hit rates in the ADAC. Consequently, the ADAC compressor achieves stable trace port bandwidth requirements with relatively small variations.

Figure 5.6 gives the compression ratio and the trace port bandwidth for the ADAC compressor. While the average trace port bandwidth steadily declines when increasing the data address cache size, its maximum value, of approximately 7.5 bits/ins., does not change much. This is caused by *tiff2rgba* test, which has over 60% of memory referencing instructions and the compression rate does not improve much when increasing the ADAC size.

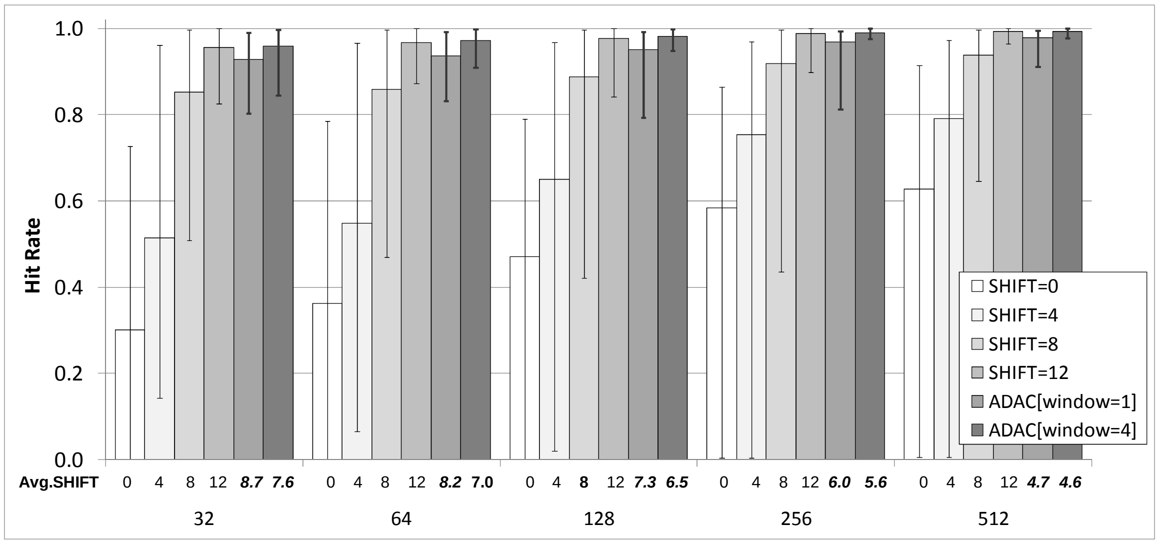


Figure 5.5 Average hit rates for different DAC-based compressors

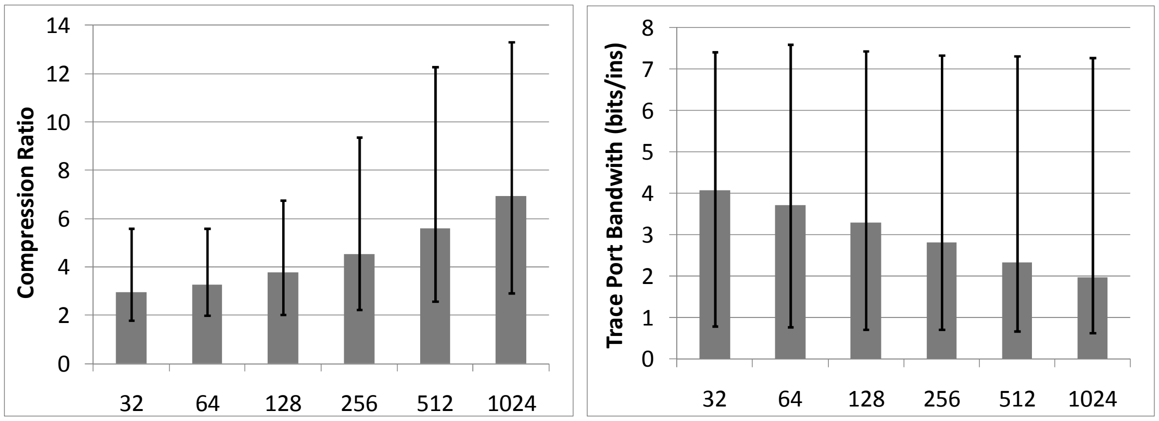


Figure 5.6 Average compression ratio and trace port bandwidth for ADAC

The ADAC compressor has a good characteristic in that it is highly configurable. Its configurability allows users to set compressor parameters for each benchmark individually in order to further minimize trace port bandwidth requirements. However, we limited our analysis to finding a good configuration that works well for the whole benchmark suite. A number of enhancements could be considered for the ADAC targeting further reducing the trace port bandwidth requirements or the compressor complexity. For example, the highest address bits may be handled separately by a global value predictor or a per-set value predictor. Another direction is to combine the proposed compressor with the filtering method proposed in Section 5.1.

5.3 Comparative Analysis

In this section we discuss the trace port bandwidth requirements for two proposed data address compression techniques: the data address filtering and the adaptive data address cache. For estimating quality of the proposed compressors, we compare their performance with the software utility program *gzip* that is supplied with a stream of raw data addresses. Please note that the *gzip* relies on very large memory buffers and highly sophisticated encoding which is not practical for implementation in hardware.

Table 5.10 shows the compression ratios for several data address trace compressors. We consider the following configurations of the ADAC compressors: ADAC 16x4 and ADAC 128x4, configured according to findings in Section 5.2.3. The bADAC 128x4 shows the results assuming the best possible configuration for each program.

We can see that the method based on filtering outperforms the ADAC method on average by a large margin. The filtering method outperforms the software *gzip* compressor for many programs. However, its performance varies widely for different benchmarks: the compression ratios range from 1.56 for *bf_e* to 250 for *tiff2bw* and *tiff2rgba*.

Table 5.10 Data address trace compression

Test	Filtering	ADAC 16x4	ADAC 128x4	bADAC 128x4	Nexus	gzip -1
adpcm_c	3.19	5.60	6.14	7.04	1.64	3.46
bf_e	1.56	4.02	7.03	7.69	2.19	4.84
cjpeg	2.73	4.31	6.25	6.53	1.34	4.48
djpeg	3.66	3.24	4.57	5.18	1.34	3.77
fft	2.81	2.74	4.14	6.53	2.14	19.9
ghostscript	3.34	2.21	5.62	8.96	1.76	18.14
gsm_d	5.95	4.69	12.27	12.78	1.54	23.3
lame	4.24	2.71	4.2	4.62	1.49	5.7
mad	5.65	2.89	3.49	3.59	1.43	3.54
rijndael_e	2.34	1.99	2.59	2.70	1.4	3.2
rsynth	3.15	2.61	5.26	7.50	1.41	21.53
sha	33.33	3.99	4.49	4.75	2.27	8.35
stringsearch	4.00	2.26	4.17	5.08	1.9	8.83
tiff2bw	250.00	3.38	3.44	4.07	1.66	2.55
tiff2rgba	250.00	2.85	2.9	3.11	1.02	2.79
tiffdither	4.63	3.70	5.55	6.75	1.39	4.41
tiffmedian	3.00	4.21	4.44	4.87	1.73	3.49
Average	**3.56**	**3.25**	**5.593**	**6.71**	**1.59**	**11.16**

Interestingly, the filtering method performs worse than the ADAC for two benchmarks *cjpeg* and *djpeg*. The ADAC method achieves a more modest compression ratio from 1.99 for *rijndael_e* to 5.6 for *adpcm_c*, but its variations between benchmarks are much smaller. With an increased size of the ADAC, the compression ratio increases, from 3.25 for 16x4 to 5.59 for 128x4. The NEXS (column 6) technique implements a simple differential encoding of data addresses as described in the Nexus standard [9]. The incoming data address is XORed with the previous data address and the difference is split into groups of 6 bits which are sent to the trace port in bytes (2 bits are header). The leading zeros are not sent. The NEXS compression ratio is rather small; it is 1.6 on average, ranging from 1 to 2.27.

Assuming 1.5 logic gates per memory bit, we estimate the complexity of the ADAC configuration with 16 sets and 4 ways to be to less than 4000 logic gates, while the configuration with 128 sets and 4 ways occupies approximately 30,000 logic gates.

CHAPTER 6

LOAD VALUES COMPRESSION

When tracing data values we are primarily concerned with load values – data read from main memory by load instructions. These values coupled with a check-pointing mechanism are sufficient to the software debugger to replay a program under test offline. The software debugger includes a functional processor simulator and maintains its software copies of the processor context (general and special purpose registers). The check-pointing mechanism ensures that the software debugger synchronizes the content of general- and special-purpose registers maintained in the software with the content of the corresponding registers in the processor core. Once the contexts are synchronized, the software debugger can replay the program execution based on only load values that are traced from the target processor. Note: in general this applies to all reads either from main memory or input peripheral devices.

Tracing load values in real-time poses a very challenging task. Programs may have a high frequency of load instructions (e.g., 40% of all instructions for *tiff2rgba* benchmark) and the amount of redundancy in load values is not nearly as close to the redundancy observed in instruction and data address traces. Thus, real-time tracing of load values would require both large on-chip trace buffers and wide trace ports for reading them out of the chip. For example, an on-chip buffer of 64 KB would be sufficient to store data traces of only 20 microseconds of execution assuming a processor running at 1 GHz with only 10% of the load instructions. To cope with excessive trace port bandwidth or storage requirements, full data tracing is applied only to short program segments. The program execution trace with a checkpoint mechanism is used to run a program close to the place where a software bug occurs, and then the trace module triggers, capturing the data traces for a short period of time. However, the size of the traced segment is limited by the size of on-chip trace buffers and may not be enough to capture the actual bug. In addition, software developers need to spend a lot of time locating the likely location of the bug, and this process is likely to be obtrusive, and thus not appropriate for debugging real-time embedded systems.

In this chapter we first explore the entropy of load values in the MiBench benchmark suite. In Section 6.1 we analyze the entropy of load values using theoretical approaches and architectural approaches when we consider the compression of a certain portion of load values. In considering load values, we may extend the processor resources or the trace module to track only the first occurrence of a data value. In that case we need to trace only the first load instruction from a given memory location and thus dramatically reduce the number of loads that need to be traced. In Section 6.2 we introduce two hardware implementations of a first-load track mechanism and evaluate its effectiveness. In Section 6.3 we put everything together and discuss the effectiveness of different approaches.

6.1 Data Values Compressibility

In this section we examine the compressibility of load values by finding the entropy of the load values data set. We also discuss the compression ratios of individual benchmarks achieved using general-purpose compressors.

6.1.1 Entropy of Load Values

The amount of redundancy in a data set can be expressed through the measure of entropy or the Shannon entropy [74]. Shannon entropy quantifies the information contained in a data set and expresses it as a single number. If the data set is considered a Markov process, meaning that the outcome (or a value) of the next data is completely random and independent of previously outcomes (values) in the data set, the Shannon entropy actually expresses the maximum possible compression achievable by a lossless compression algorithm. Such entropy is also named Markov 0th order entropy [75].

Since we examine the set of load values as a Markov process, or a set of independent values, we calculate the Markov 0th order entropy for MiBench programs in order to find the overall compressibility of load values. The Markov 0th order entropy accounts for the probability of values within the data set and assigns fewer bits to more frequent values. The entropy thus expresses the average number of bits that can be used to represent a value in the data set. The total entropy, H_{M0}, is expressed as $H_{M0} = -\sum_i [\log(p_i) \cdot p_i]$, where p_i is the probability of occurrence of an unique data i in the data set. To calculate entropy, we do not evaluate all program generated load values as one data set, but rather divide them into smaller sets of several kilobytes each. Each set is then examined independently from the other sets. The goal of this approach is to better represent the changes in load values and their occurrence frequencies through different program phases. In addition, the actual evaluation of a data set which can contain millions of different values would not be practical.

Table 6.1 shows the results for the Markov 0th order entropy. We consider four block sizes: 1 KB, 8 KB, 64 KB and 128 KB. For each block size, four columns are shown: column *min* is the minimum observed entropy across all blocks in a program, column *max* is the maximum observed entropy across all blocks in a program, and column *avg* is the average entropy across all blocks and the measure of entropy for a program (represents the measure of entropy). Column *#val* shows the average number of unique data values per block during the program execution. The results indicate large variations of the entropy between programs and within a program as it moves through different program stages. For example, for 64KB block size, the average entropy varies between 0.2 and 13 and its overall average is 8.1. The maximum variation for a single benchmark is observed in *tigg2rgba*, ranging between 0.2 (min) and 11 (max). In addition to the entropy variation, we expectedly observe lower entropy values for smaller block sizes: the average entropy is 6.3 for 1 KB, 7.2 for 8 KB, 8.1 for 64 KB and 8.3 for 128 KB blocks.

The variations observed above indicate that the number of more frequent values and their occurrence frequencies are changing from block to block, and the compression process would benefit from an adaptive compression algorithm.

To calculate compression ratios, we assume tracing H_{M0} bits for a data value already logged in the table and 32 bits for its first occurrence; we have *#val* occurrences per one block. We have already observed lower entropy for smaller blocks; in addition, smaller blocks mean more frequently tracing entire sets of data values (32-bit for word operands). Table 6.2 shows the compression ratios for different block sizes of 1 KB, 8 KB, 64 KB, and 128 KB. The compression ratio averages (last row in the table) are calculated using a weighted arithmetic mean, where a program weight is directly proportional to the number of loads in that program relative to the total number of loads in all programs. The compression ratios range between 2.07 for *lame* to 5.65 for *adpcm_c* assuming the large 128 KB blocks. It ranges between 1.78 for *lame* to 4.43 for *cjpeg* for blocks of 1 KB. The overall results of this exercise indicate a limited compressibility of data values, and thus information-theoretic approaches to load value compression are not likely to be cost-effective.

Table 6.1 Markov 0th order entropy for MiBench tests

	Block = 1KB				**Block = 8KB**				**Block = 64KB**				**Block = 128KB**			
Test	***min***	***avg***	***max***	***#val***	***min***	***avg***	***max***	***#val***	***min***	***avg***	***max***	***#val***	***min***	***avg***	***max***	***#val***
adpcm_c	3.0	5.2	7.0	174	5.0	5.5	6.8	303	5.1	5.6	5.7	320	5.1	5.6	5.7	333
bf_e	4.7	6.4	8.9	298	7.1	7.3	9.6	1284	7.5	7.6	10.2	3977	7.6	7.7	9.4	6549
cjpeg	0.2	4.1	9.5	114	0.5	4.6	9.1	322	2.7	5.1	9.1	1155	2.7	5.2	9.0	1896
djpeg	0.1	4.1	8.2	127	2.7	4.8	8.0	493	3.0	5.2	7.5	1991	3.0	5.5	7.6	3559
fft	3.6	6.3	8.3	196	3.9	6.9	10.7	738	4.3	7.4	13.0	4482	4.4	7.6	13.5	8521
ghostscript	0.9	5.5	9.3	105	1.0	5.8	9.3	296	4.6	6.1	10.0	1621	5.6	6.1	10.2	3061
gsm_d	3.6	6.4	7.2	231	3.7	7.4	8.1	976	5.0	8.2	8.7	2805	4.9	8.3	8.8	3670
lame	0.0	7.7	10.0	434	3.4	9.3	11.2	2123	4.7	11.2	12.6	12407	5.7	11.9	13.1	23261
mad	2.9	7.7	9.4	420	4.4	8.8	10.9	1692	7.2	10.2	11.0	9154	7.8	10.5	11.2	15788
rijndael_e	6.5	8.1	8.3	542	9.1	9.1	9.2	1793	9.5	9.5	9.6	4864	9.6	9.6	9.6	7776
rsynth	0.0	6.8	7.6	223	0.1	7.9	8.5	1190	2.9	8.7	9.1	5937	3.3	8.9	9.3	10343
sha	5.6	8.0	8.1	302	9.5	10.2	10.3	1696	11.9	12.2	12.2	9699	12.2	12.5	12.6	14646
stringsearch	6.1	6.5	7.5	165	7.0	7.2	8.6	562	7.6	7.6	7.9	2859	7.6	7.7	7.8	4032
tiff2bw	0.0	4.1	10.0	215	0.0	4.9	10.4	1362	0.2	5.3	7.5	8592	0.4	5.5	7.5	15923
tiff2rgba	0.0	4.9	10.0	292	0.0	5.8	10.3	1626	0.2	6.2	11.1	7874	0.3	6.4	11.3	13556
tiffdither	0.9	4.8	9.8	116	1.5	5.3	8.3	397	2.1	5.9	7.0	2058	2.2	6.0	6.8	3490
tiffmedian	0.0	3.8	10.0	182	1.2	4.3	9.1	1053	1.6	4.8	8.8	6986	1.6	4.9	9.0	13108
Average	**1.8**	**6.3**	**8.9**	**270**	**3.1**	**7.2**	**9.5**	**1197**	**4.5**	**8.1**	**10.0**	**5887**	**4.9**	**8.3**	**10.1**	**10483**

Table 6.2 Compression ratio achieved by measuring Markov 0th order entropy

	Compression Ratio			
Test	**1KB**	**8KB**	**64KB**	**128KB**
adpcm_c	3.28	4.91	5.60	5.65
bf_e	2.31	2.87	3.53	3.60
cjpeg	4.43	5.62	5.79	5.72
djpeg	4.22	4.99	5.30	5.17
fft	2.85	3.48	3.52	3.49
ghostscript	3.88	4.74	4.78	4.74
gsm_d	2.63	3.10	3.48	3.58
lame	1.78	2.11	2.11	2.07
mad	1.81	2.35	2.41	2.44
rijndael_e	1.54	2.26	2.86	2.93
rsynth	2.60	2.80	2.96	2.98
sha	2.13	2.18	2.12	2.18
stringsearch	3.01	3.61	3.68	3.79
tiff2bw	3.22	3.41	3.62	3.68
tiff2rgba	2.54	2.92	3.43	3.55
tiffdither	4.06	4.88	4.73	4.79
tiffmedian	3.65	4.06	4.17	4.22
Average	**2.47**	**2.97**	**3.16**	**3.17**

6.1.2 Variability of High-order Bits

In this section we explore the potential of an approach that compresses only the upper bits of data values; the lower bits are sent to the trace port uncompressed. The rationale behind this approach is that short constants may dominate program data sets; the upper bits are often zeros and thus need not be traced. To evaluate the compressibility of high-order load value bits, we find the Markov 0th order entropy for load values when lower bits are masked out. The MASK parameter determines the number of lower bits that is masked out. We consider data values as a sequence of 32-bit words (byte size and half-word size operands are expanded to 32-bits).

Table 6.3 shows the entropy for different values of the MASK parameter (0, 4, 8, and 12), the average number of the unique non-masked data values (DV[31:MASK]), and the compression ratio. We assume a large block size of 128 KB in this experiment. To calculate compression ratios, we assume tracing H_{M0} + MASK bits if the non-masked portion of the data value is already logged in the table and the entire 32-bits if it is not in the table (first occurrence of a unique value).

Table 6.3 Entropy of high-order load value bits.

	Entropy				***#val* (Number of unique values)**				**Compression Rate**			
Test \ MASK	**0**	**4**	**8**	**12**	***0***	***4***	***8***	***12***	***0***	***8***	***4***	***12***
adpcm_c	5.6	4.4	3.1	2.1	333	93	60	32	5.65	3.82	2.88	2.27
bf_e	7.7	6.6	5.0	4.4	6549	2383	2106	2079	3.60	2.92	2.41	1.92
cjpeg	5.2	3.9	2.6	2.0	1896	1276	1017	762	5.72	3.93	2.97	2.27
djpeg	5.5	4.4	3.1	2.4	3559	2762	2486	2036	5.17	3.60	2.78	2.18
fft	7.6	6.4	5.5	4.9	8521	7630	6652	4964	3.49	2.74	2.21	1.84
ghostscript	6.1	5.0	3.7	3.2	3061	744	295	132	4.74	3.51	2.73	2.10
gsm_d	8.3	6.5	5.2	4.3	3670	1470	952	798	3.58	2.98	2.40	1.95
lame	11.9	11.4	10.8	10.0	23261	21603	19632	13708	2.07	1.77	1.54	1.39
mad	10.5	9.1	7.0	4.8	15788	10996	9060	7519	2.44	2.18	1.98	1.81
rijndael_e	9.6	8.9	7.9	7.5	7776	7499	6922	6566	2.93	2.29	1.91	1.59
rsynth	8.9	8.2	7.0	6.5	10343	9243	8834	8381	2.98	2.36	1.99	1.65
sha	12.5	12.0	11.7	11.1	14646	14346	13792	10057	2.18	1.80	1.52	1.35
stringsearch	7.7	6.4	5.0	4.0	4032	1304	693	428	3.79	3.01	2.44	1.99
tiff2bw	5.5	4.2	2.8	2.5	15923	13148	10926	6785	3.68	3.02	2.54	2.08
tiff2rgba	6.4	5.0	3.7	2.9	13556	9448	6200	3267	3.55	2.99	2.54	2.09
tiffdither	6.0	4.1	2.6	1.7	3490	1593	1318	1012	4.79	3.80	2.96	2.31
tiffmedian	4.9	3.8	2.1	1.5	13108	6042	4646	3430	4.22	3.59	2.93	2.28
Average	**8.3**	**7.2**	**6.0**	**5.3**	**10483**	**8220**	**7276**	**5600**	**3.17**	**2.58**	**2.15**	**1.79**

The results show that by increasing the size of the MASK parameter, the entropy goes down: it is 8.3 bits for MASK=0, 7.2 bits for MASK=4, 6.0 bits for MASK=8, and 5.3 bits when MASK=12. Also, we can observe that the number of unique non-masked values goes down (*#val* columns). The lower *#val* and the lower entropy both lead to a smaller number of bits needed to represent a value within a block which has a positive effect on the compression ratio. However, this improvement is not sufficient to compensate an increase in the number of bits that is sent uncompressed (MASK bits). For example, an increase of the MASK parameter from 0 to 4 leads to a decrease of the entropy for approximately 1.2 bits, while the *#val* parameter goes from 10483 to 8220. This is not sufficient to compensate for sending MASK=4 lower value bits for each load instruction. This fact is clearly illustrated by the compression ratios that decrease with an increase in the number of MASK bits. Consequently, we conclude that this selective masking of lower load value bits is not a practical solution for trace compression.

6.1.3 Compression of Small Load Values Only

Some data values are proven to occur more frequently than others in regular programs [59]. These values are -1, 0, 1, 2, 4 and other small numbers. Thus, a compressor can implement a simple encoding that uses fewer bits to encode smaller values. Table 6.4 shows the fraction of load values that fit into SIZE bits, for different values of the parameter SIZE (=4, 5, ..., 12 bits). The results indicate high variations among programs. For example, *cjpeg* and *djpeg* have more than 40% and *tiff2bw* has almost 60% of the load values that fit into 4 bits, while *sha* has only 14% that can fit into 12 bits. The results also indicate that the number of data values that can fit into a relatively small number of bits quickly saturates. For example, 32.2% of load values can fit into 8 bits on average; the percentage of load values that can fit into 12 bits is only negligibly higher and it is 36.7%. This result indicates a limited potential of this approach in exploiting load value redundancy. Figure 6.1 shows the compression ratio as a function of the parameter SIZE. The best compression ratio is achieved when SIZE=8 (~1.4*x*) and has maximum for *tiff2bw* benchmark (~2.5*x*). Consequently, we conclude that this approach would not be practical for the compression of load values.

Table 6.4 The fraction of load values that fit in SIZE bits

Test \ SIZE	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
adpcm_c	0.069	0.100	0.148	0.192	0.324	0.327	0.335	0.356	0.403
bf_e	0.053	0.069	0.130	0.173	0.197	0.197	0.197	0.197	0.197
cjpeg	0.428	0.469	0.659	0.671	0.704	0.713	0.723	0.725	0.728
djpeg	0.406	0.420	0.440	0.488	0.625	0.631	0.669	0.671	0.671
fft	0.285	0.291	0.307	0.315	0.316	0.316	0.317	0.329	0.330
ghostscript	0.291	0.301	0.378	0.400	0.451	0.454	0.460	0.463	0.466
gsm_d	0.214	0.233	0.257	0.285	0.302	0.305	0.314	0.326	0.343
lame	0.082	0.092	0.100	0.105	0.117	0.122	0.130	0.138	0.149
mad	0.179	0.205	0.242	0.256	0.270	0.300	0.323	0.348	0.392
rijndael_e	0.029	0.055	0.073	0.130	0.200	0.200	0.200	0.201	0.202
rsynth	0.107	0.111	0.120	0.159	0.189	0.219	0.226	0.228	0.230
sha	0.000	0.000	0.001	0.137	0.137	0.138	0.138	0.138	0.140
stringsearch	0.107	0.119	0.215	0.387	0.387	0.389	0.392	0.394	0.397
tiff2bw	0.595	0.674	0.755	0.801	0.818	0.818	0.820	0.823	0.827
tiff2rgba	0.374	0.424	0.476	0.506	0.516	0.517	0.518	0.519	0.521
tiffdither	0.307	0.361	0.435	0.493	0.569	0.572	0.579	0.589	0.763
tiffmedian	0.446	0.656	0.700	0.730	0.790	0.792	0.795	0.799	0.861
Average	**0.189**	**0.220**	**0.256**	**0.286**	**0.322**	**0.330**	**0.335**	**0.342**	**0.367**

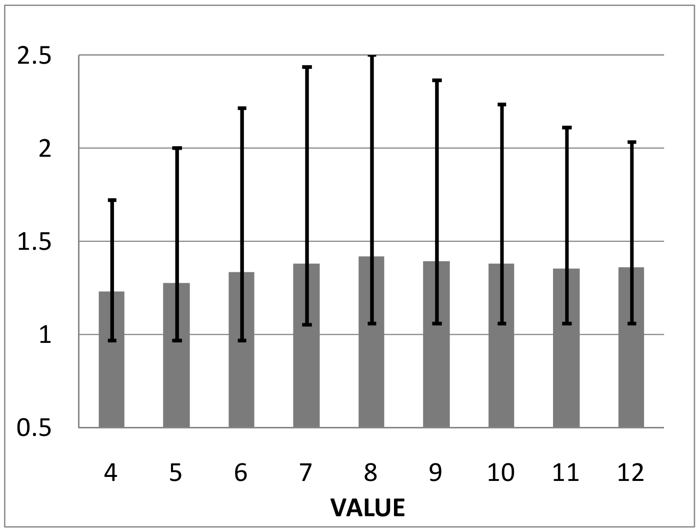


Figure 6.1 Compression ratio when using optimal length for load values encoding

6.2 Compression Using First-Access Cache Track Mechanism

Data caches are routinely used to reduce latency of memory-referencing instructions. Data caches exploit temporal and spatial locality of data with more recently used data kept in fast and small cache structures. Here we argue that they can be augmented to help load value trace compression too. A software debugger with a full functional simulator can also maintain a software copy of a data cache. Thus, our trace module does not need to trace all load values. Instead, it can only trace load values that miss in the processor data cache and are brought from the memory. To replay the program off-line, the software debugger inspects its software copy of the data cache. If it finds the requested data in the data cache, it can continue the program reply using the value stored in the cache. Otherwise, the software debugger expects a trace record from the target machine. Data caches have high hit rates, and consequently we can expect that they can help significantly reduce the number of load values that need to be traced.

Narayanasamy et al. propose a mechanism [27] that traces only those loads that access a memory location for the first time. They assume a software debugger that maintains a software copy of the entire memory. Only first accesses to memory locations need to be traced on the target machine. For all subsequent accesses, data can be retrieved from the memory maintained by the software debugger. A data in a memory location is known (valid) from the time the processor loads or stores the data to that memory location until it is overwritten by some other device accessing the memory (e.g., DMA controller, cache controller that maintain coherence protocols, or from the secondary storage). A flag is added to each memory location to mark whether it is valid or not. The proposed mechanism is named *first-load trace* as only the first loads to the memory location have to be traced.

We expand the mechanism proposed in [27] and name it *first-access track mechanism*. Our first-access track scheme differs from the one in [27] as we track the validity of L1 cache lines only, not the whole memory. The tracking of the L1 cache only is more suitable for embedded systems and it also makes the mechanism hardware implementation feasible. A vendor usually develops a processor's subsystem which includes the L1 cache. Also, the corresponding trace module is developed in parallel. Other parts of the whole system-on-a-chip, such as the L2 cache or the main memory are developed independently of the processor core. Developing a trace module for a processor core (including L1 cache) allows developers a modular design and design reuse. From the software debugger perspective, it is also easier to extend the functional simulator with the L1 cache functionality only, rather than with the whole memory simulator.

6.2.1 First-Access Track Mechanism Details

The key operation in the first access track scheme is updating (set/reset) first-access flags attached to each L1 cache line. An L1 cache line can be replaced from the upper memory hierarchy whenever the cache has a miss or some other device (cache coherence controller, DMA) replaces a cache line with new data. When this happens, all first-access flags in the selected cache line have to be reset.

A first-access mechanism system view is shown in Figure 6.2 and the summary of various events of interest and corresponding action is given in Table 6.5. A program executes on a target processor core. The processor includes an L1 data cache that supports a first access track mechanism, and it is connected to a trace module. The L1 data cache is extended with first-access flags, one per minimum instruction set addressable unit (e.g., byte, word). When a memory referencing instruction has a hit in the cache, the first-access flag for a selected word is set (if not already set), indicating that the value in the cache is known. When a memory referencing instruction has a miss in the cache, a block of data is brought from the upper memory hierarchy and all first-access flags are reset. Next, the selected word is traced out and its first-access flag is set. The invalidation of a cache line from an external device also invalidates all first-access flags in that cache line.

The trace module operation is rather simple. It looks for events in the L1 cache and decides whether to trace the load value out of the chip or not. The trace module suppresses the tracing of load value only if the cache reports a hit and the flag-access bit for the addressed word is originally set. For each load value, the trace module also traces a header (e.g., hit=1, miss=0) bit. The software debugger knows whether to expect a load value or not based on this header bit. Note: here we assume that a data cache line can be invalidated using external events (those that are not caused by program execution). If such events are not possible, we may avoid tracing the header bit.

On the software debugger side, a fully functional simulator is extended to accept load values either from the L1 cache simulator or the trace port; for each load instruction, the software debugger reads the header bit from the trace file. If the header bit is set, the software debugger reads the selected word from its model of the L1 cache. If the header bit is reset, the debugger reads a load value from the trace port. Also, the data cache is updated with the incoming load value.

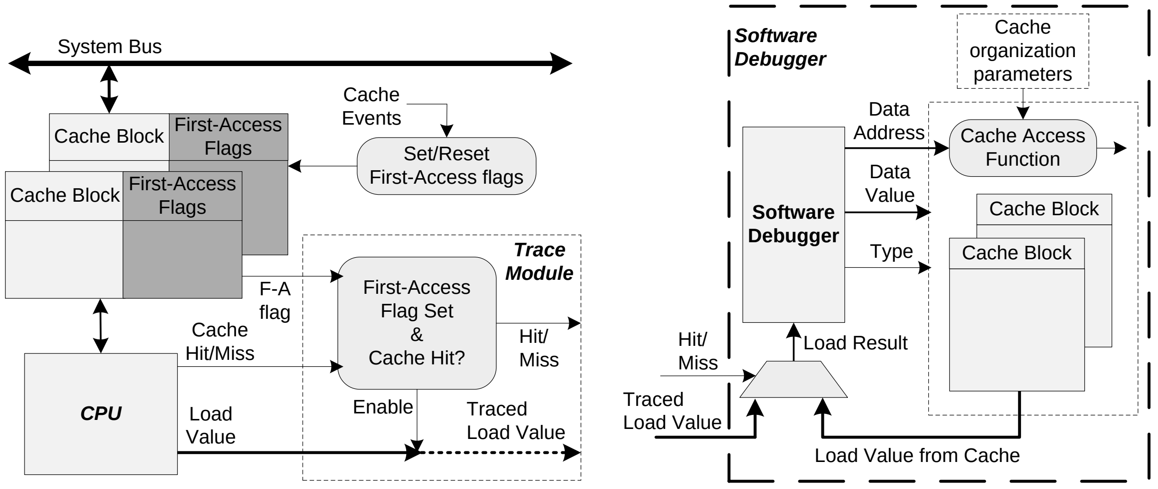


Figure 6.2 System view of the first-load track mechanism

Note: While this mechanism supports the external invalidation of cache lines, the single-threaded MiBench programs we use generate L1 cache invalidations only on cache misses. This may produce skewed results when comparing to a realistic multi-core system. However, we do not expect significant changes due to the following: The majority of load values in a program is produced by the program itself and is not received from other sources. For example, let us consider a MiBench *cjpeg* program that processes an input picture that is 800 KB is size. Assume that this picture is brought into the system memory using a DMA. The *cjpeg* program includes 28 million load instructions for a total amount of load data brought to the chip by all load instructions over 100 MB. Consequently, the input picture would account for less than 1% of the all load data values. This practically means that the external device invalidations of cache lines cannot have a noticeable impact on overall cache invalidation rate.

Table 6.5 First-access mechanism operation: L1 cache (a), trace module (b), software debugger (c)

L1 events (on load or store instruction)	***L1 action***
Cache Hit	Set First-Access flag
Cache Miss	Reset all First-Access flags in selected line Set First-Access flag for the selected word in line
External Invalidation	Reset all First-Access flags in addressed cache line

(a)

Trace Module Events (on load instruction)	***Trace Module Action***	***Trace Record***
Cache Hit & First-Access flag set	Trace hit bit only	*"1"*
All others	Trace miss bit followed by data value	"0" + "N-bit load value" (N = size of transfer, e.g. 8, 16, 32 bits)

(b)

Software Debugger Events (on load instruction)	***Software Debugger Action***
Trace hit/miss bit set	Take load value from the cache
Trace hit/miss bit not set	Take load results from the trace file Allocate traced load value to the cache

(c)

6.2.2 Results and Analysis

The results for the first-access track mechanism are given in Figure 6.3. Figure 6.3*a* shows the cache hit rate while the Figure 6.3*b* shows the percentage of load instructions for which the corresponding first-access flag is set (first-access flag set hit rate). The results are given for three block sizes (4, 16 and 64 bytes) and for cache sizes from 1 KB to 128 KB; the number of cache ways is fixed to 4. We can observe that the cache hit rate benefits from larger block sizes, while the first-access flag set hit rate benefits from smaller ones, as the number of flags invalidated after cache line invalidation is lower.

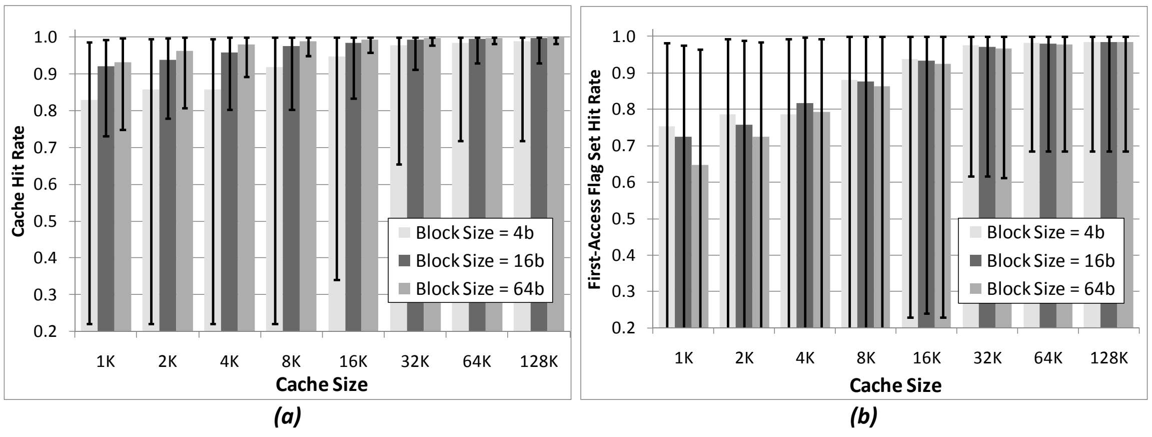


Figure 6.3 Cache Hit rate (a) and first-access flag set hit rate (b) for the first-access track mechanism

Table 6.6 shows detailed results for first-access set hit rate when the block size is 64, which is more realistic in modern processors. Both the cache hit rate and the first-access flag set hit rate are overall high, but some programs have poor performance even for very large cache structures. For example, *tiff2rgba* achieves a first-access flag set hit rate of only 68% for a cache size of 64 KB and approximately 0% for a cache size of 8 KB. The zero hit rate is caused by the sequential access to the cache, which represents the worst-case scenario regarding the first-access mechanism.

Table 6.6 First-access set hit rate for cache with 64-byte block

	First-access set hit rate							
Test \ Size	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**
adpcm_c	0.711	0.740	0.981	1.000	1.000	1.000	1.000	1.000
bf_e	0.773	0.801	0.905	0.983	0.999	1.000	1.000	1.000
cjpeg	0.595	0.669	0.728	0.798	0.831	0.851	0.858	0.871
djpeg	0.551	0.634	0.735	0.835	0.921	0.983	0.998	0.999
fft	0.841	0.910	0.979	0.993	0.993	0.993	0.993	0.993
ghostscript	0.758	0.928	0.982	0.986	0.988	0.990	0.991	0.991
gsm_d	0.964	0.984	0.993	0.999	1.000	1.000	1.000	1.000
lame	0.552	0.615	0.708	0.868	0.942	0.964	0.975	0.993
mad	0.592	0.641	0.780	0.948	0.977	0.995	0.997	0.999
rijndael_e	0.499	0.524	0.564	0.891	0.994	1.000	1.000	1.000
rsynth	0.704	0.868	0.961	0.976	0.982	0.982	0.983	0.985
sha	0.961	0.966	0.966	0.990	1.000	1.000	1.000	1.000
stringsearch	0.717	0.851	0.920	0.972	0.987	0.992	0.993	0.993
tiff2bw	0.007	0.007	0.007	0.045	0.626	0.996	1.000	1.000
tiff2rgba	0.007	0.007	0.007	0.007	0.229	0.612	0.685	0.685
tiffdither	0.668	0.815	0.825	0.869	0.941	0.999	1.000	1.000
tiffmedian	0.478	0.497	0.508	0.520	0.698	0.881	0.971	0.999
Average	**0.648**	**0.726**	**0.793**	**0.863**	**0.924**	**0.966**	**0.978**	**0.985**

To calculate the trace port bandwidth requirements, we assume tracing of the first-access flag value as a hit or miss information, followed by an N-bit load value if the flag is not set (N=8, 16 or 32 depending on size of the memory transfer). The results for the 4-way cache with the 64-byte block are shown in Table 6.7. The results indicate high variations in the required trace port bandwidth. While some benchmarks require small tracing overhead with small cache sizes (*gsm_d* requires 0.27 bits/ins for 1 KB cache), some other tests have high trace bandwidth requirements (*tiff2rgba* achieves 4.41 bits/ins at lowest). Also, some programs (*tiffrgba*, *tiffmedian* , *rijndael_e* and *tiff2bw*) require large cache sizes (8KB and above) for a noticeable decrease in required trace port bandwidth.

Table 6.7 Trace port bandwidth for first-load track mechanism

Test \ Size	Trace Port Bandwidth (bits/ins)							
	1K	2K	4K	8K	16K	32K	64K	128K
adpcm_c	0.43	0.40	0.15	0.13	0.13	0.13	0.13	0.13
bf_e	2.75	2.42	1.31	0.45	0.38	0.38	0.38	0.38
cjpeg	2.56	1.91	1.45	0.90	0.72	0.64	0.61	0.57
djpeg	2.89	2.23	1.61	0.98	0.62	0.40	0.35	0.34
fft	1.46	0.92	0.42	0.31	0.31	0.31	0.31	0.31
ghostscript	2.26	0.89	0.41	0.38	0.36	0.35	0.34	0.34
gsm_d	0.27	0.21	0.19	0.17	0.17	0.17	0.17	0.17
lame	5.08	4.37	3.32	1.71	0.93	0.70	0.59	0.43
mad	3.89	3.61	2.32	0.74	0.47	0.31	0.30	0.29
rijndael_e	7.35	7.05	6.49	1.96	0.52	0.44	0.44	0.44
rsynth	4.24	2.02	0.77	0.57	0.49	0.49	0.48	0.48
sha	0.38	0.35	0.35	0.22	0.17	0.17	0.17	0.17
stringsearch	1.48	0.78	0.46	0.33	0.24	0.21	0.21	0.21
tiff2bw	4.07	4.07	4.07	3.87	2.34	0.30	0.27	0.27
tiff2rgba	8.52	8.52	8.52	8.52	7.81	5.34	4.41	4.41
tiffdither	1.62	0.85	0.81	0.71	0.48	0.23	0.22	0.22
tiffmedian	2.72	2.58	2.50	2.43	1.86	0.95	0.48	0.32
Average	**3.34**	**2.58**	**1.96**	**1.22**	**0.85**	**0.59**	**0.51**	**0.46**

6.2.3 Dedicated First-Access Tracking Scheme

We can extend the first-access track mechanism to require hardware changes in the trace module only while the data cache remains unchanged. The trace module includes a cache that stores recently seen load and store values. By using this approach, first-access flags are no longer needed. Instead, a load value is traced only when the incoming load value is not found in the trace module cache. If it is found in the trace module cache, the load value is not traced. This mechanism offers greater flexibility over the first-access flag mechanism for an application specific processor as it allows changing the configuration of the cache and its size to the application requirements. For example, a *gsm_d* benchmark will achieve excellent compression rates for the cache size of only 1 KB, which actually requires less hardware than implementing first-access flags if, for example, a 64 KB L1 cache is used (8 KB of first-access flags required).

Figure 6.4 compares the trace port bandwidth for an N-way cache with a 64-byte block while the N takes values of 1, 2 and 4. We see a very small increase in the required trace port bandwidth when we decrease the number of ways. This indicates that the trace module may include a simple and smaller caches solely dedicated to tracing.

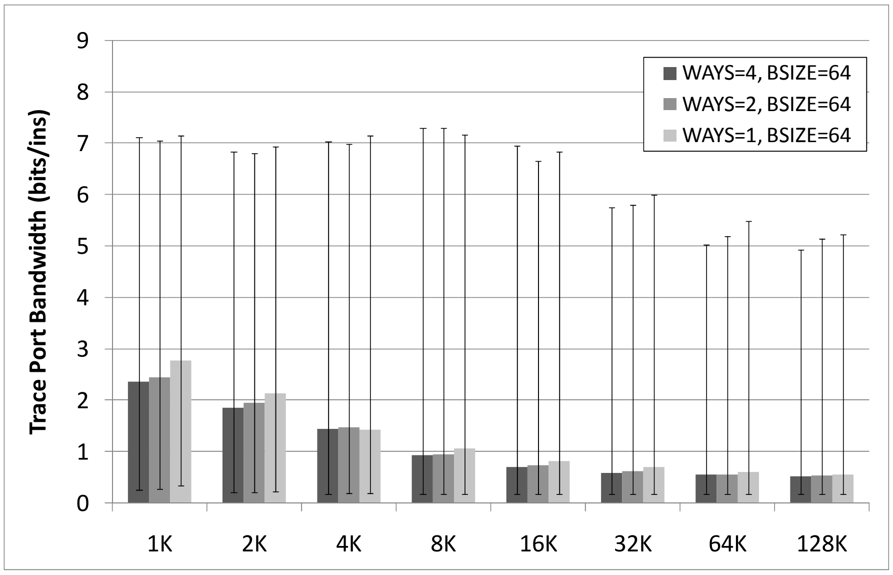


Figure 6.4 Trace port bandwidth for dedicated first-load track mechanism

6.3 Putting It All Together

In this chapter we examined the compressibility of load values by finding the entropy of the load value data set. The results indicate a compression algorithm would benefit from working with large sets of load values, which is not feasible for hardware implementation. Also, by measuring the Markov 0^{th} order entropy, we found the compression limit is just over three times. Further, we examined the entropy of high-order load value bits and found that the entropy does not decrease to compensate for the lower bits that are traced without compression. This leads to a conclusion that the compression of high-order bits only is not an efficient mechanism. We also examined the possibility of compression of small load values as they are very frequent (values such as -1, 0, 1, 4 …). We found that the compression ratios achieved using this approach are very small (less than 2 times for almost all program).

The greater level of compressibility can be achieved by including the processor's architectural state to augment compression. We examined the first-access track mechanism which tracks the state of the level one data cache in both the processor and the software debugger. A load value is traced out of the chip only if the debugger cannot find the correct value in its own cache. The main drawback of the first-access scheme is its dependency on processor internals, and it must be implemented together with the processor subsystem which is often not welcome by system architects. Also, it requires the implementation of the L1 data cache, which may not be available in low-end embedded processors. Fortunately, integration with the L1 cache offers a key advantage: the designers may opt to increase the cache size (or to include one) to benefit both the load value compression and the processor performance.

The first-access track scheme can be decoupled from the processor operation. In this implementation, first-load flags are not attached to the cache but reside in the trace module while all the necessary signals to access the cache come from the processor trace port as usual. This is possible because the trace module has all the parameters needed to organize first-load flags to follow the cache organization and program execution; organization of a cache is a design parameter available to the trace module designer, while the program execution (addressing into the cache, type of reference, hits/miss…) is available on the trace port. However, additional implementation details are necessary; the cache access decoder has to be duplicated to access first-access flags in the trace module operation and the trace module must track the invalidations of cache lines from an external device.

Finally, the first-access mechanism can be fully decoupled from the cache operation; the trace module can implement the dedicated cache, which stores results of load and store operations and traces a load value only if it does not match the one in the addressed cache line. This mechanism can be useful for application specific systems, where the static profiling of the application can help find the most efficient cache size and parameters.

We compare compression ratios achieved using the first-access mechanism with compression rate limits extracted from the Markov 0th order entropy and the compression ratio achieved by the general purpose compressors (*gzip, bzip*). Table 6.8 shows the results for the first-access mechanism with 8 KB and 64 KB cache sizes (64-byte block and 4 ways) in the first two columns, compression ratios achieved using the Markov 0th order entropy for 128 KB data sets and the fast *gzip* (*-1* option). The results indicate the first-access mechanism with the 64 KB cache outperforms the other schemes for all programs except for *tiff2rgba*. Compression using *bzip-1* greatly improves the compression for certain tests (e.g., *cjpeg*, *djpeg*, *ghostscript* and *stringsearch*) as it uses an algorithm to reorder data within a large data set. However, even then, low predictability of data set values leads to a decrease of compression rate for the *sha* benchmark. In result, we can say that the first-access track mechanism is an excellent approach for the compression of load values. It outperforms other mechanisms, even the software ones, for almost all benchmarks.

Table 6.8 Load values compression rate comparison

	First-Access		**Markov 0th**		
Test \ Size	**8KB**	**64KB**	**128KB**	***gzip-1***	***bzip -1***
adpcm_c	25.14	25.14	5.65	4.14	8.83
bf_e	25.56	30.11	3.60	3.81	5.04
cjpeg	8.47	12.55	5.72	6.54	14.11
djpeg	7.79	22.02	5.17	6.47	10.41
fft	25.16	25.41	3.49	4.73	6.32
ghostscript	23.59	25.95	4.74	13.63	20.40
gsm_d	27.53	28.12	3.58	3.64	5.36
lame	6.42	18.58	2.07	2.52	2.51
mad	11.52	28.58	2.44	2.44	2.70
rijndael_e	6.29	28.11	2.93	2.36	2.69
rsynth	23.34	27.65	2.98	3.32	3.77
sha	21.86	28.62	2.18	2.48	1.99
stringsearch	14.25	22.39	3.79	6.34	10.85
tiff2bw	1.00	14.23	3.68	3.47	4.61
tiff2rgba	0.96	1.86	3.55	3.43	4.21
tiffdither	7.52	23.96	4.79	4.55	8.21
tiffmedian	2.87	14.55	4.22	4.47	7.06
Average	**6.69**	**17.20**	**3.17**	**4.12**	**5.72**

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Our society relies on embedded computer systems that drive transportation, communication, medicine, and all other aspects of our daily life. Semiconductor trends have enabled embedded computer systems with ever-increasing functionality and sophistication that are less costly and smaller. Developing and testing of software in modern embedded systems becomes one of the most critical issues. At the time of writing this dissertation, our society is going through costly recalls in the automotive industry, some of which are caused by software bugs in cars. Software developers of real-time embedded systems need appropriate hardware and software tools that will enable them to gain insight into what their system is doing at any point in time. These tools should not interfere with normal system operation and should be easy to use and possibly reduce the time software developers spend on debugging (over 80% of their development time according to some estimates).

Trace modules are crucial components of modern embedded systems that help software debugging. Trace modules capture program traces and send them through trace ports to a host machine running a software debugger. However, program traces are extremely large (an embedded processor running at 1 GHz can generate more than 1 GB of trace data per second, making program tracing in real-time impractical or impossible). The current approaches to debugging require costly on-chip trace buffers and wide trace ports that can capture program traces unobtrusively for short program segments. However, these resources are rarely made available by processor vendors. These problems are further exaggerated by the current trends toward complex multi-core systems on a single chip and a scalability discrepancy between the on-chip logic (growing exponentially with each new technology generation) and available I/O bandwidth (does not grow exponentially). This dissertation is aimed at developing trace modules that will dramatically reduce the amount of trace data by means of trace compression.

This dissertation proposed a number of algorithms for the cost-effective compression of program traces, including instruction address traces, data address traces, and load data values. We explore characteristics of individual trace components, introduce new compression algorithms, and explore the design space that includes trace port bandwidth requirements and implementation cost. Our goal is to develop techniques that will minimize the required trace port bandwidth (maximize the compression ratio) and minimize the implementation cost, while allowing for unobtrusive program tracing in real-time. To achieve this goal we characterize program traces and exploit their inherent characteristics together with cost-effective architectural solutions. Typically, we apply filtering to minimize the number of trace records needed for program replay at the software debugger. Compression is carried out using architecture-inspired resources (stream caches, predictors, adaptive data address caches) where a sequence of trace records is translated into a sequence of a few hit/miss events on these structures. These events are then encoded in cost-effective encoding methods.

For the compression of instruction address traces, we introduce three new techniques, namely, Double Move-to-Front (DMTF), Stream Caches with Last Stream Predictors (SC-LSP), and a Branch Predictor-based compression. All three techniques guarantee unobtrusive tracing of instruction address traces with less than 0.15 bits/instruction on the trace port bandwidth. The best performing and the least costly technique based on Branch Predictors requires 0.036 bits/ins at the trace port at the cost of 5,200 additional gates. This is almost 28-fold improvement in the required bandwidth over the commercial state-of-the-art at negligible hardware costs.

For compression of data addresses we introduce two new techniques. The first technique uses new Data Address Filtering to reduce the number of data addresses that need to be traced. The second technique performs compression using a new hardware structure called Adaptive Data Address Cache. The data address filtering achieves average compression ratios of 3.56, outperforming the software compression utility *gzip* for many programs. The filtering method requires minimal hardware complexity providing that certain signals from the processor core are made available. The adaptive data address cache achieves a compression ratio of 5.6 at the cost of 30,000 logic gates. State-of-the-art solutions achieve compression ratios on data addresses of only 1.6. Consequently, the data address filtering provides 10-fold, and the adaptive data address cache provides over 3.5-fold improvement over the currently used compression method in the Nexus standard.

Finally for data values, we show that tracing only load values is sufficient assuming sophisticated software debuggers. We expanded previously proposed techniques for software trace capturing and made them suitable for embedded systems and hardware trace capturing. Our technique is called first-access load tracking and we offer two implementations – one closely coupled with the processor cache and the other that is implemented in the trace module. Depending on cache size, the proposed technique achieves compression ratios of 6.9 (assuming 8 KB data cache) and 17.2 (assuming 64 KB data cache). This method also outperforms the software compression of all load values at the cost of relatively small additional complexity.

Our results indicate that real-time program tracing is possible in embedded systems if our trace compression techniques are used. They significantly reduce the required trace port bandwidth and eliminate the need for deep on-chip trace buffers to capture traces before they are read out through narrow trace ports.

While we focused on developing compression algorithms and cost-effective hardware implementations in uni-processor systems, the proposed compression algorithms are applicable to multi-core systems too. Future research efforts should address tracing of on-chip interconnect and trace synchronization in multi-core systems. One direction for future research would be to explore combining different compressors: for example, our data address filtering method can be combined with a second-level compressor to further reduce the size of data addresses. In spite of excellent average compression ratios, we noticed that certain benchmarks still pose high bandwidth requirements. They may also benefit from further investigation focusing on combining different compressors.

REFERENCES

[1] B. Dipert, "Inside Apple's Iphone: More Than Just a Dial Tone," <http://www.edn.com/article/CA6457065.html?nid=2551> (Available September 2007).

[2] C. J. Murray, "Automakers Aim to Simplify Electrical Architectures," <http://www.designnews.com/article/316784-Automakers_Aim_to_Simplify_Electrical_Architectures.php> (Available September 2009).

[3] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A Reconfigurable Design-for-Debug Infrastructure for SoCs," in *43rd Design Automation Conference*, 2006, pp. 7-12.

[4] RTI-International, "The Economic Impacts of Inadequate Infrastructure for Software Testing," <http://www.nist.gov/director/prog-ofc/report02-3.pdf> (Available July 2009).

[5] ARM, "Embedded Trace Macrocell, Architecture Specification," <http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014o/IHI0014O_etm_v3_4_architecture_spec.pdf> (Available November 2009).

[6] MIPS, "The PDtrace™ Interface and Trace Control Block Specification " <http://www.mips.com/products/product-materials/processor/mips-architecture/> (Available November 2009).

[7] Infineon, "TC1775 System Units 32-Bit Single-Chip Microcontroller," <http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b7535083b> (Available November 2009).

[8] Freescale, "MPC555 / MPC556 User's Manual," <http://www.freescale.com/files/microcontrollers/doc/user_guide/MPC555UM.pdf> (Available November 2009).

[9] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface," IEEE- Industry Standards and Technology Organization (IEEE-ISTO), 2003.

[10] W. Orme, "Debug and Trace for Multicore SoCs," ARM White Paper, 2008.

[11] B. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," in *Joint International Conference on Measurement and Modeling of Computer Systems*, 1994, pp. 128-137.

[12] 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*, 2008.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Conference on Programming Language Design and Implementation*, 2005, pp. 190-200.

[14] K. Hazelwood and A. Klauser, "A Dynamic Binary Instrumentation Engine for the ARM Architecture," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[15] S. Bhansali, W.-K. Chen, S. d. Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau, "Framework for Instruction-Level Tracing and Analysis of Program Executions," in *2nd International Conference on Virtual Execution Environments*, 2006, pp. 154-163.

[16] Tensilica, *Xtensa LX2 Microprocessor Data Book for Xtensa® LX2 Processor Cores*, 2008.

[17] Xilinx, "Microblaze Processor Reference Guide Embedded Development Kit EDK 11.4," UG081 (v10.3), 2009.

[18] Xilinx, "Xilinx Microblaze Trace Core (XMTC) Product Specification," DS642, 2009.

[19] Xilinx, "Microblaze Debug Module (MDM) Product Specification," DS641, 2009.

[20] ARM, "Embedded Trace Buffer Technical Reference Manual," ARM DDI 0242B, 2002.

[21] "IEEE P1149.7, Draft Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary Scan Architecture, ," IEEE Computer Society, 2008.

[22] "'MIPI Test and Debug Interface Framework,'' V3.2, White Paper," MIPI Test and Debug Working Group, 2006.

[23] "MIPI Alliance Test and Debug - NIDnT-Port,'' V1.0, White Paper, MIPI Test and Debug Working Group," MIPI Test and Debug Working Group, 2007.

[24] OCP-IP, "The Importance of Sockets in SoC Design, White Paper," <http://www.ocpip.org/uploads/documents/sockets_socdesign.pdf> (Available 2010).

[25] OCP-IP, "Socket-Centric IP Core Interface Maximizes IP Applications, White Paper," <http://www.ocpip.org/uploads/documents/wp_pointofview_final.pdf> (Available 2010).

[26] N. Stollon, B. Uvacek, and G. Laurenti, "Standard Debug Interface Socket Requirements for OCP-Compliant SoC, White Paper," 2007.

[27] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," in *International Symposium on Computer Architecture*, 2005, pp. 284-295.

[28] R. H. B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[29] ARM, "Coresight on-Chip Debug and Trace Technology," <http://www.arm.com/products/solutions/CoreSight.html> (Available July 2004).

[30] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory* 1977, pp. 337-343.

[31] M. Burrows and D. J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," Digital SRC Research Report, 1994.

[32] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," in *Proceedings of the I.R.E*, 1952, pp. 1098-1102.

[33] J. S. Vitter, "Design and Analysis of Dynamic Huffman Codes," *Journal of the ACM*, vol. 34, 1987, pp. 825-845.

[34] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, 1987, pp. 520-540.

[35] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm," *Journal of Artificial Intelligence Research*, vol. 7, 1997, pp. 67-82.

[36] J. R. Larus, "Whole Program Paths," *ACM SIGPLAN Notices*, vol. 34, 1999, pp. 259-269.

[37] Y. Zhang and R. Gupta, "Timestamped Whole Program Path Representation and Its Applications," *ACM SIGPLAN Notices*, vol. 36, 2001, pp. 180-190.

[38] E. E. Johnson, J. Ha, and M. B. Zaidi, "Lossless Trace Compression," *IEEE Transactions on Computers*, vol. 50, 2001, pp. 158-173.

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

[40] J. R. Larus, "Efficient Program Tracing," *IEEE Computer*, vol. 26, 1993, pp. 52-61.

[41] A. D. Samples, "Mache: No-Loss Trace Compaction," *ACM SIGMETRICS Performance Evaluation Review*, vol. 17, 1989, pp. 89-97.

[42] E. N. Elnozahy, "Address Trace Compression through Loop Detection and Reduction," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, 1999, pp. 214-215.

[43] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia, "Sigma: A Simulator Infrastructure to Guide Memory Analysis," in *ACM/IEEE Supercomputing Conference*, vol. 1-13, 2002.

[44] A. R. Pleszkun, "Techniques for Compressing Program Address Traces," in *Proceedings of the 27th International Symposium on Microarchitecture*, 1994.

[45] A. Milenkovic and M. Milenkovic, "Stream-Based Trace Compression," *IEEE Computer Architecture Letters*, vol. 2, 2003, pp. 4.

[46] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective Stream-Based and Execution-Based Data Prefetching," in *Proceedings of the 18th International Conference on Supercomputing*, 2004, pp. 1-11.

[47] G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation," in *Proceedings of the 31st International Symposium on Microarchitecture*, 1998, pp. 127-137.

[48] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *International Conference on Architectural Support for Programming Languages and Operatings Systems*, 1996.

[49] M. Burtscher and B. G. Zorn, "Exploring Last N Value Prediction," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, 1999, p. 66.

[50] R. J. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit for Pipelined Processors," *IBM Journal of Research and Development archive*, vol. 37, 1993.

[51] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[52] Y. Sazeides and J. E. Smith, "Modeling Program Predictability," *International Symposium on Computer Architecture* 1998 pp. 73-84.

[53] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction Using Hybrid Predictors," *International Symposium on Microarchitecture* 1997, pp. 281-290.

[54] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai, and J. P. Shen, "Load Execution Latency Reduction," in *International Conference on Supercomputing* ACM, 1998, pp. 29-36.

[55] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," *ACM SIGARCH Computer Architecture News*, vol. 27, 1999, pp. 54-63.

[56] M. Milenković, A. Milenković, and M. Burtscher, "Algorithms and Hardware Structures for Unobtrusive Real-Time Compression of Instruction and Data Address Traces," in *Proceedings of the 2007 Data Compression Conference*, 2007, pp. 55-65.

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

[58] M. Burtscher, "VPC3: A Fast and Effective Trace-Compression Algorithm," in *Joint International Conference on Measurement and Modeling of Computer Systems*, 2004, pp. 167-176.

[59] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *ACM SIGARCH Computer Architecture News*, vol. 28, 2000, pp. 150-159.

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

[61] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the IEEE 4th Workshop on Workload Characterization*, 2001.

[62] E. Morancho, J. M. Llabería, and A. Olivé, "Split Last-Address Predictor," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, 1998, pp. 230-239.

[63] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" For Enabling Full-System Multiprocessor Deterministic Replay," *ACM SIGARCH Computer Architecture News*, vol. 31, 2003, pp. 122-135.

[64] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, 2002, pp. 59-67.

[65] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A Locally Adaptive Data Compression Scheme," *Communications of the ACM*, vol. 29, 1986, pp. 320-330.

[66] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE Journal of Solid-State Circuits*, vol. 41, 2006.

[67] Intel, "Intel Xscale® Core Developer's Manual," 2004.

[68] J. K. F. Lee and A. J. Smith, "Branch Target Buffer Design and Optimization," *IEEE Transactions on Computers*, vol. 42, 1993, pp. 396-412.

[69] K. Driesen and U. Hölze, "Accurate Indirect Branch Prediction," in *Proceedings of the 25th International Symposium on Computer Architecture*, 1998, pp. 167-168.

[70] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, and A. Naveh, "The Intel Pentium M Processor: Microarhitecture and Performance," *Intel Technology Journal*, vol. 07, 2003, pp. 21-36.

[71] M. Burtscher, "TCgen 2.0: A Tool to Automatically Generate Lossless Trace Compressors," *SIGARCH Computer Architecture News*, vol. 34, 2006, pp. 1-8.

[72] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 2007.

[73] "MIPS32 34k Processor Core Family Software User's Manual Rev. 01.11," MIPS Technologies, Inc., 2008.

[74] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, 1948, pp. 623-656.

[75] S. P. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability*: Springer-Verlag, London, 1993.