Real-time Unobtrusive Program Execution Trace Compression Using Branch Predictor Events

Vladimir Uzelac
Tensilica, Inc
255-6 Scott Blvd.
Santa Clara, CA 95054
vuzelac@tensilica.com

Aleksandar Milenković
The University of Alabama in Huntsville
301 Sparkman Dr.
Huntsville, AL 35899
+1 256 824-6830
milenka@uah.edu

Martin Burtscher
University of Texas at Austin
1 University Station C0200
Austin, TX 78712
+1 512 232-0808
burtscher@ices.utexas.edu

Milena Milenković
IBM
11501 Burnet Road
Austin, TX 78758
+1 512 286-5845
mmilenko@us.ibm.com

ABSTRACT

Unobtrusive capturing of program execution traces in real-time is crucial in debugging cyber-physical systems. However, tracing even limited program segments is often cost-prohibitive, requiring wide trace ports and large on-chip trace buffers. This paper introduces a new cost-effective technique for capturing and compressing program execution traces in real time. It uses branch predictor-like structures in the trace module to losslessly compress the traces. This approach results in high compression ratios because it only has to transmit misprediction events to the software debugger. Coupled with an effective variable encoding scheme, our technique requires merely 0.036 bits/instruction of trace port bandwidth (a 28-fold improvement over the commercial state-of-the-art) at a cost of roughly 5,200 logic gates.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.5: [**Testing and Debugging**]: Debugging aids, Tracing. E.4 [**Coding and Information Theory**]: Data Compaction and Compression.

General Terms

Algorithms, Design, Verification.

Keywords

Debugging, Program Tracing, Compression.

1. INTRODUCTION

Ideally, firmware and software developers of embedded systems would like to be able to answer the simple question "What is my system doing?" at any point in the development cycle. However, achieving complete visibility of all signals in real time in modern embedded systems is not feasible due to limited I/O bandwidth, high internal complexity, and high operating frequencies. Software developers face additional challenges caused by growing software complexity and ever tightening time-to-market pressures. According to an estimate, software developers spend 50%-75% of their development time in program debugging [1], yet the nation still loses approximately $20-$60 billion a year due to software bugs and glitches. The latest recalls in the automotive industry are a stark reminder of the need for improved software testing – a recent study found that 77% of all electronic failures in automobiles are due to software bugs [2]. To meet these challenges and get reliable and high-performance products on the market on time, software developers increasingly rely upon on-chip resources for debugging and tracing. However, even limited hardware support for tracing and debugging is associated with extra cost in chip area for capturing and buffering traces, for integration of these modules into the rest of the system, and for sending out the information through dedicated trace ports [3]. These costs often make system-on-a-chip designers reluctant to invest additional chip area for debugging and tracing.

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 real-time embedded systems. For instance, setting a breakpoint may be impossible or harmful in real-time systems such as a hard drive or 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.

Recognizing these issues, many vendors have developed modules with tracing capabilities and integrated them into their embedded platforms, e.g., ARM's Embedded Trace Macrocell [4], MIPS's PDTrace [5], and OCDS from Infineon [6]. The IEEE's Industry Standard and Technology Organization has proposed a standard for a global embedded processor debug interface called Nexus 5001 [7].

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. Hardware traces can be classified into three main categories depending on the type of information they capture: program (or instruction) traces, data traces, and interconnect traces. In this paper we focus on program execution traces, i.e., Class 2 operation in Nexus. They consist of the addresses of all executed instructions and are crucial for both hardware and software debugging, as well as for program optimization and tuning.

The existing commercially available trace modules rely either on hefty on-chip buffers to store execution traces of sufficiently large program segments or on wide trace ports that can sustain a large amount of trace data in real-time. However, large trace buffers and/or wide trace ports significantly increase the system's complexity and cost. Moreover, they do not scale well - the number of I/O pins dedicated to tracing cannot keep pace with the exponential growth in the number of on-chip logic gates – which is a substantial problem in the era of multicore systems.

Compressing program execution traces at runtime in hardware can reduce the requirements for on-chip trace buffers and trace port communication bandwidth. Whereas commercially available trace modules typically implement only rudimentary forms of hardware compression with a relatively small compression ratio (~1 bit per instruction) [8], several recent research efforts in academia propose trace compression techniques that reach much higher compression ratios [9, 10]. For example, Kao *et al*. [9] propose an LZ-based program trace compressor that achieves a good compression ratio for a selected set of programs. However, the proposed module has a relatively high complexity (50,000 gates). Uzelac and Milenkovic introduced a double move-to-front method that requires 0.12 bits/instruction on the trace port on average at the estimated cost of 24,600 logic gates [10]. A compressor using a stream descriptor cache and predictor structures requires a slightly higher trace port bandwidth of 0.15 bits/instruction, but at much lower hardware complexity [11].

In this paper we introduce a new technique for hardware compression of program traces in real-time (Section 2). The proposed technique relies on a trace module that incorporates a branch predictor-like structure to track the program execution. An identical structure is maintained in software by the debugger. To be able to replay the program execution off-line, we only need to record misprediction events in the trace module. These events are efficiently encoded (Section 3) and read out of the chip through a trace port. Our experimental analysis (Section 4) shows that the proposed technique requires only 0.036 bits/instruction of trace port bandwidth at an estimated cost of about 5,200 gates.

The main contributions of this work are as follows:

- We propose using branch predictor like structures in the trace module for cost-effectively and unobtrusively capturing and compressing program traces at run-time;
- We introduce an effective and low-complexity encoding scheme for the events that are captured at these branch predictor like hardware structures;
- We perform a detailed experimental analysis that shows the proposed trace compression scheme to achieve excellent compression ratios, outperforming existing hardware-based techniques for compression of program execution traces; it requires over *28 times less bandwidth on the trace port* than commercial state-of-the-art solutions and over *three times less* than recently published academic proposals at much lower hardware cost.

2. USING BRANCH PREDICTOR EVENTS TO CAPTURE PROGRAM EXECUTION TRACES

A program's execution path can be replayed off-line by recording changes in the program flow caused by control-flow instructions or exceptions during execution. When a change in the program flow occurs, we need to capture (a) the program counter (PC) of the currently executing instruction and (b) the branch target address (BTA) in case of a control-flow instruction or the exception-handler target address (ETA) in case of an exception. Thus, the program's execution path can be recreated by recording a sequence of (PC, BTA) and (PC, ETA) pairs. To reduce the amount of data, the program counter values can be replaced by the number of instructions executed in a sequential run since the last change in the control flow (SL). However, even this type of trace may contain redundant information that can be inferred by the software debugger from the program binary, such as the target address of a direct branch (the BTA is known at compile time). Similarly, there is no need to report unconditional direct branches as control-flow events; their outcomes and targets are known at compile time. However, in spite of these optimizations, the number of bits that needs to be traced out of the processor core is still relatively large. Depending on the frequency and type of control-flow instructions, this number ranges from 0.2 to 5 bits/instruction for typical benchmarks, requiring a deep trace buffer and a wide trace port. For example, in the worst-case scenario (5 bits/instruction), a trace buffer of 8 KB will capture a program trace for a program segment with slightly over 1,600 instructions. However, such a small number of instructions is insufficient in locating software errors in modern programs, where distances between bugs and their manifestations may be millions of instructions.

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 early in the pipeline, thus reducing the number of wasted clock cycles due to control hazards. The target of a branch is predicted using a branch target buffer (BTB), a cache-like structure indexed by a portion of the branch address [12], that keeps target addresses of taken branches. A separate hardware structure named indirect branch target buffer (iBTB) can be used to better predict indirect branches that may have multiple targets [13]. A dedicated stack-like hardware structure called return address stack (RAS) is often used to predict return addresses [14]. Branch outcome predictors range from a simple linear branch history table (BHT) with 2-bit saturating counters to very sophisticated hybrid branch outcome predictor structures found in recent commercial microprocessors [15]. Branch predictors are typically very effective, predicting branch outcomes and target addresses with over 95% accuracy.

Our key observation is that *program execution can be replayed off-line using a branch predictor trace* instead of a branch instruction trace. We propose a trace module that consists of branch predictor structures solely dedicated to real-time hardware trace compression. To distinguish it from the processor's branch predictor, we named it Trace Module Branch Predictor (TMBP). The TMPB includes structures for predicting branch targets and branch outcomes. Unlike regular branch predictors, the 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 like a regular branch predictor but later, i.e., 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*. These events are encoded and sent via a trace port to a 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 the TMBP is updated during program execution.

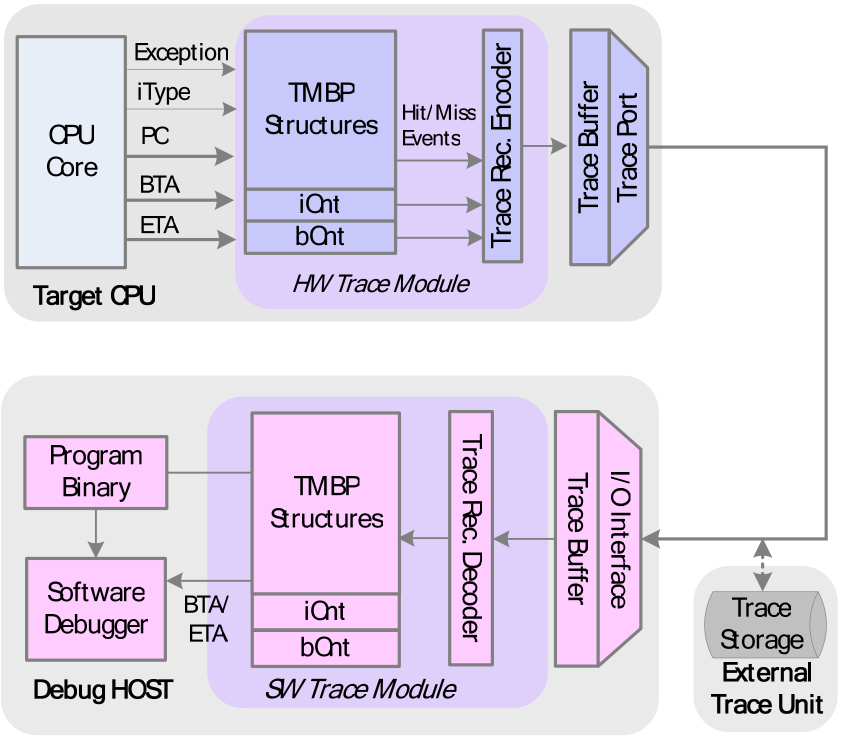


Figure 1. Trace module system view.

Figure 1 shows a system view of the proposed tracing mechanism. The trace module (TM) is coupled with the CPU core through an interface that carries the relevant information for each control-flow change: the branch target address (BTA), the exception target address (ETA), the program counter (PC), the instruction type (iType), and an exception control signal. The trace module monitors this information and updates its state accordingly. It 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 2 for the trace module operation). The counters are 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)[1]. These branch instructions may be either correctly predicted or mispredicted by the TMBP. In case of a correct prediction, the trace module does nothing beyond the counter updates. In case of a misprediction, the trace module generates a trace record that needs to be sent to the software debugger and clears the counters. The type and format of the trace record depends on the branch type and the misprediction event type (see Table 1). In case of a direct branch outcome misprediction, the trace record includes

```
// For each committed instruction
iCnt++;  // increment iCnt
if ((iType==IndBr)||(iType==DirCB)) {
  bCnt++;  // increment bCnt
  if (TMBP mispredicts) {
    Encode TMBP mispredicton 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 2. Trace module operation.

Table 1. Trace module branch prediction events and trace records: T – Taken, NT – Not Taken.

Branch Type	***TMBP Events***	***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**, **ETA**)

[1] Note: direct unconditional branches are not reported because their target can be inferred by the software debugger.

only the *bCnt* value so that the software debugger can replay the program execution until the mispredicted branch is reached. Then, it simply follows the not-predicted path. In case of an indirect branch misprediction, we can have an outcome misprediction, a target address misprediction, or both. For an indirect branch incorrectly predicted as taken, the trace record includes the *bCnt* and information specifying that the branch is not taken (NT bit). In case of a target address misprediction, the trace record includes the *bCnt*, the outcome taken bit (*T*), and the actual target address (*TA*). Finally, in case of an exception, the trace module emits a trace record that includes the *iCnt* and the starting address of the corresponding exception handler.

The software debugger replays all instructions updating the software copy of the branch predictor and the counters in the same way their hardware counterparts are updated (see Figure 3). 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 reaches zero, the software debugger processes the current instruction depending on its type. If the 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 program execution, and updates its predictor accordingly. Similarly, if the debugger processes an exception trace record, the *iCnt* counter is decremented on each instruction retirement until the instruction on which the exception has occurred is reached. If the software debugger can replay the exception handler, tracing can continue and the compressor structures are updated as usual. Alternatively, the tracing is stopped and resumed upon return from the exception handler. A developer needs to configure the trace module for one of these two options using configuration messages before the tracing starts; in addition, the software debugger also needs to know which of these two approaches is used.

An inevitable question is why we do not simply capture the necessary branch events in the regular branch predictor, thus eliminating the need to implement separate TMBP structures. While such an approach is possible and desirable for reducing complexity, it would require tight integration of the trace module with the CPU pipeline and would place debilitating restrictions on the branch predictor's design and operation. For example, we would need to reset the content of the branch predictor to a known state on each context switch to maintain consistency between the branch predictor in the CPU pipeline and the branch predictor in the software debugger. More importantly, we would need to disallow speculative updates of the branch predictor structures. These restrictions would result in an unacceptable loss of accuracy of the branch predictor and thus are not further considered in this paper.

2.1 Example

Let us first illustrate program tracing on the example of a code segment consisting of 4 basic blocks *W*, *X*, *Y*, and *Z* as shown in Figure 4 (left). Let us consider three iterations of the loop with the 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 initially predicts the branch `jle Y` to be not

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

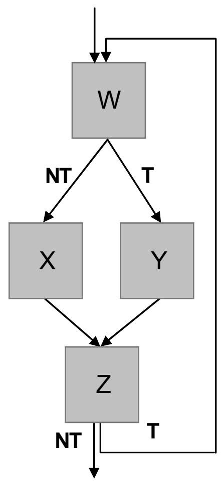
Figure 3. Execution replay in the software debugger.

taken (*P=NT*), and the branch `jge W` to be taken (*P=T*). Program execution starts with the first instruction in the block *W* (*i1*); the trace module increments the *iCnt* and *bCnt* counters as shown in the execution table in Figure 4. In the first two loop iterations, the 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 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 for this outcome misprediction is emitted with the counter value *bCnt=1*.

Assume the software debugger is ready to replay the program starting from instruction *i1*. It receives a trace record indicating that the program should be replayed until the fifth conditional branch is reached (replay table in Figure 4). 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 instruction `jle Y` in the third iteration), the debugger knows that the branch outcome of the current direct branch is different from the one suggested by the software TMBP. The debugger needs to update its predictor structures according to their update policies. It then reads the next trace record and continues program replay from the first instruction in block *Y*.

2.2 Related software-based trace compression techniques

A number of trace-specific software-based trace compression techniques have recently been introduced [16], [17]. The relationship between data compression and branch prediction was first noted by Chen *et al.* [18]. Several recent software-based trace compression techniques rely on branch predictors [19] or, more generally, on value predictors [20]. Many of these schemes include trace-specific compression in the first stage, combined with a general-purpose compressor in the second stage. For example, Barr and Asanović [19] have proposed a branch-predictor based trace compression scheme for improving architectural simulation. Similar to our scheme, they keep track of the number of correct predictions and emit entire trace records only in case of mispredictions. Whereas this scheme utilizes the



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/T*	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
Z: 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

Replay

Iteration 1		*Iteration 2*		*Iteration 3*	
P/A	*{i/b} Cnt*	*P/A*	*{i/b} Cnt*	*P/A*	*{i/b} Cnt*
	-/5		-/3		-/1
	-/5		-/3		-/1
NT/NT	-/4	NT/NT	-/2	NT/T*	-/0
	-/4		-/2		
	-/4		-/2		
	-/4		-/2		
	-/4		-/2		
					-/1
					-/1
					-/1
	-/4		-/2		-/1
	-/4		-/2		-/1
T/T	-/3	T/T	-/1	T/NT*	-/0

Figure 4. Tracing and replaying the sample execution.

same underlying program characteristics as our scheme, there are some notable differences, as discussed below.

First, the algorithm presented in [19] *compresses program traces in software* and is aimed at warming-up architectural simulators. It is designed to maximize the compression ratio assuming virtually unlimited storage and processing resources. Hence, it relies on large predictor structures that require megabytes of memory storage. More importantly, it utilizes the *gzip* compression algorithm for efficient encoding of the output trace. Such an approach would be cost-prohibitive or infeasible for real-time compression in hardware.

Moreover, the inner workings of the compression algorithm proposed in [19] are different from our approach. Whereas we use a subset of regular branch predictor structures in the trace module and encode regular misprediction events, their predictor structures behave differently. They use the incoming branch trace records as input into a range of branch predictor like software structures to predict the next trace record, rather than the next instruction.

In conclusion, our goal is to develop a hardware trace compressor that uses *a minimal subset of branch predictor structures* (e.g., we do not use a BTB) and employs an *efficient encoding scheme that ensures unobtrusive tracing in real-time at minimal hardware cost*.

3. TRACE RECORD ENCODING

Trace records should be encoded in a way that minimizes both the storage requirement in the trace buffer as well as the trace port bandwidth. The proposed mechanism uses four types of trace records as shown in Figure 5. The trace record length depends on the event type and can vary from several bits to several dozen bits. Using a fixed number of bits in the trace record for the *bCnt* and *iCnt* values would not be a good solution because the distance between two consecutive branch mispredictions may vary widely between programs as well as within a single program as it moves through different program phases. The typical values found in *bCnt* and *iCnt* are also heavily influenced by the misprediction rate, which is a function of the type and organization of the branch predictor. Thus, there is no general, optimal solution for encoding. Instead, we opt for an empirical approach in determining trace record formats. Our goal is to devise an effective yet easy-to-implement encoding scheme with minimal hardware complexity.

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

A trace record emitted on a direct branch outcome misprediction event consists of a header (*bh*) and a *bCnt* counter field (Figure 5a). A similar format is used for indirect conditional branches. If an indirect branch is predicted as taken but is actually not taken, a trace record with the format shown in Figure 5b is used. The additional one-bit field *O*=′0′ specifies that the outcome of the branch is not correct. The similar format shown in Figure 5c is used for indirect conditional branches that are predicted not taken but are actually taken. Indirect unconditional mispredictions are encoded as shown in Figure 5d. The trace records shown in Figure 5c and Figure 5d carry information about the *bCnt* counter as well as the branch target address *TA*. A simple approach would be to just append an additional 32-bit field holding the target address to the original trace record. An alternative approach is to encode the difference between subsequent target addresses. The trace module maintains the previous target address (*PTA*) – that is, the target

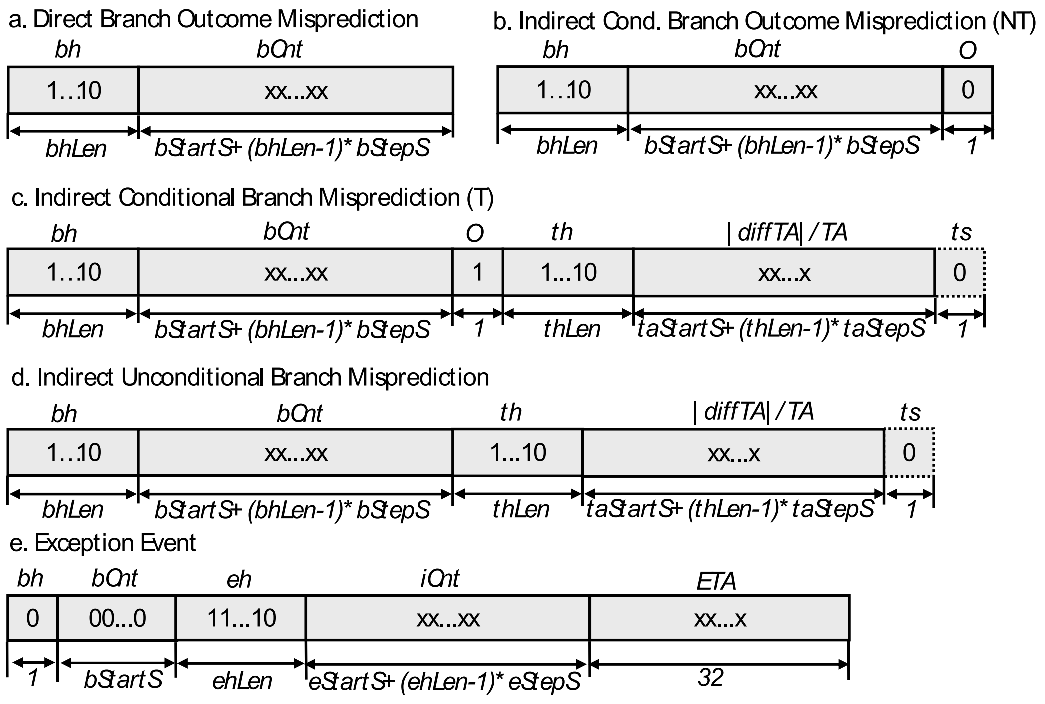


Figure 5. Trace record formats for branch misprediction events and exceptions.

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 benchmarks with a significant number of indirect branches, we found that we can indeed shorten the trace records by using difference encoding.

We employ variable encoding for the difference/target address field (|*diffTA*|/*TA*). Its length is specified by the number of header bits (*th*). We adopt the following scheme: a single header bit (*th* = '0') specifies *taStartS* bits in the |*diffTA*|/*TA* field. The two-bit header (*th* = '10') specifies *taStartS+1*taStepS* bits, the three-bit header (*th* = '110') specifies *taStartS+2*taStepS* bits, and so on. If the |*diffTA*|/*TA* field requires fewer than 32 bits, we also need to provide information about the sign bit (*ts*) of the difference; otherwise, the whole 32-bit address is included.

Figure 5e shows the format of the 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 executed since the last branch predictor misprediction event. We use the same variable encoding as before – the *ehLen*-bit header specifies the *eStartS+(ehLen-1)*eStepS* 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 simply encode the whole 32-bit exception address.

The optimal setting of the trace record parameters depends on the benchmarks and on the characteristics of the branch predictor. A detailed analysis aimed at finding good values for these parameters is provided in the next section.

4. EXPERIMENTAL EVALUATION

The goals of this section are as follows. First, we profile our benchmarks to determine good values for the trace record parameters, including *bStartS*, *bStepS*, *taStartS*, *taStepS*, *eStartS*, and *eStepS* (Section 0). After determining the trace record parameters, we evaluate the effectiveness of the proposed tracing mechanism by measuring the average trace port bandwidth for several branch predictor configurations (Section 4.3). We also compare the effectiveness of our mechanism with that of other academic proposals (Section 4.3). The trace port bandwidth requirements are expressed in bits per instruction (bits/ins) and averages are calculated using the weighted arithmetic mean; the weights are proportional to the number of executed instructions in programs. Note that the compression ratio for a program execution trace is 32/(Trace Port Bandwidth) for 32-bit architectures; e.g., a trace port bandwidth of 0.05 bits/ins is equivalent to a compression ratio of 640:1. Our analysis is performed using a functional and cycle-accurate SimpleScalar ARM simulator [21] with processor parameters modeled after the XScale processor.

As workload we use seventeen benchmarks from MiBench, a representative suite of benchmarks for embedded computers [22]. We consider three TMBP configurations, *bTMBP*, *sTMBP*, and *tTMBP*. The base TMPB configuration (*bTMPB*) includes a 64-

entry 2-way set associative iBTB and an 8-entry RAS for indirect branch target prediction, and a 512-entry GSHARE global outcome predictor [23]. Each entry in the iBTB includes a 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 of the last 9 conditional branches. The *sTMBP* configuration includes a smaller, 32-entry iBTB and the *tTMBP* does not include any iBTB.

In Section 4.2 we introduce an enhancement aimed at reducing the complexity and estimate the cost of the proposed configurations. Reducing the hardware complexity of the tracing infrastructure is especially important in mid- and low-end embedded processors where a complete processor cores require 50 – 100 Kgates. In addition, in multicore processors each core would need its own trace compression structures. Reducing the complexity of the required compressor structures may motivate processor vendors to more readily include debugging infrastructure that supports unobtrusive program tracing.

4.1 Encoding Profiles

To determine good values for the 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 found that our variable encoding scheme indeed reduces the size of the output trace and outperforms any fixed-length encoding scheme. Next, we analyzed several combinations of (*bStartS*, *bStepS*) pairs (*bStartS*∈[2..6] and *bStepS*∈[1..6]) to determine an optimal combination that results in the shortest program traces across our benchmark suite. The results of this analysis indicate that the total trace size is minimal when *bStartS*=3 and *bStepS*=2 for the *bTMBP* and *sTMBP* configurations and *bStartS=3* and *bStepS*=1 for the *tTMBP* configuration.

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

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

4.2 Hardware 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 the predictor structures is straightforward. For the iBTB and RAS, we include an enhancement to reduce their complexity. We find that the uppermost 12 bits of the indirect branch targets remain unchanged relative to the previous target in 99.99% of the cases. Consequently, we can use a last value 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 zero in the ARM architecture). A miss in the last value predictor causes the 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 and RAS hit rates. It should be noted that the number of bits that can be eliminated from the iBTB target address fields with negligible penalty for the prediction rates depends on benchmark characteristics. However, we believe that a certain number of upper address bits is likely to stay constant or change infrequently, even with dynamically loaded libraries, object-oriented code, and other modern software techniques.

To determine the size of the trace output buffer, we used a cycle-accurate processor model to find the maximum number of bits in this buffer at any point 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 TMBP warm-up when we experience a number of consecutive mispredictions in the benchmark *fft*. For the bTMBP configuration, we find that a buffer of 79 bits ensures that the processor is never stalled due to tracing and that no trace records are lost.

The estimates for the hardware complexity measured in logic gates for the three configurations are as follows: *tTMBP* requires 2,800 gates, *sTMBP* requires 4,000, and *bTMBP* requires slightly over 5,200 gates. These estimates confirm our expectations about the relatively small complexity of the proposed trace module compressor structures and support our decision to implement a separate TMBP outside of the processor pipeline.

4.3 Trace Port Bandwidth

Table 2 and Figure 6 show the results of the trace port bandwidth analysis for the three configurations of the proposed trace module (*tTMBP*, *sTMBP*, and *bTMBP*) and preexisting techniques (NEXS, TSLZ, and DMTF). We compare our technique with a Nexus-like trace module (NEXS) [7] and two trace-specific adaptations of general-purpose compression algorithms, namely the LZ scheme (TSLZ) [9] and the DMTF scheme [10]. To illustrate the effectiveness of the proposed technique, we also compare it to the software gzip utility when compressing a sequence of (SL, TA/-) pairs. Please note that implementing a *gzip* compressor in hardware would be cost-prohibitive in both the on-chip area and the compression latency.

The NEXS scheme assumes sending the minimum information needed to the trace port to replay the program off-line; it consists of a sequence of (SL, TA/-) pairs. The SL field records the number of sequentially executed instructions from the last taken branch and the TA field records the target address for indirect branches or exceptions. The TA field is differentially encoded and leading zeros are not emitted, which is similar to the Nexus standard. The TA field is XORed with the previous TA and the difference is split in groups of 6 bits. E.g., if *diffTA*[31:6] consists of zeros, then only *diffTA*[5:0] is sent to the trace port, together with a 2-bit header indicating that this is a terminating byte for the

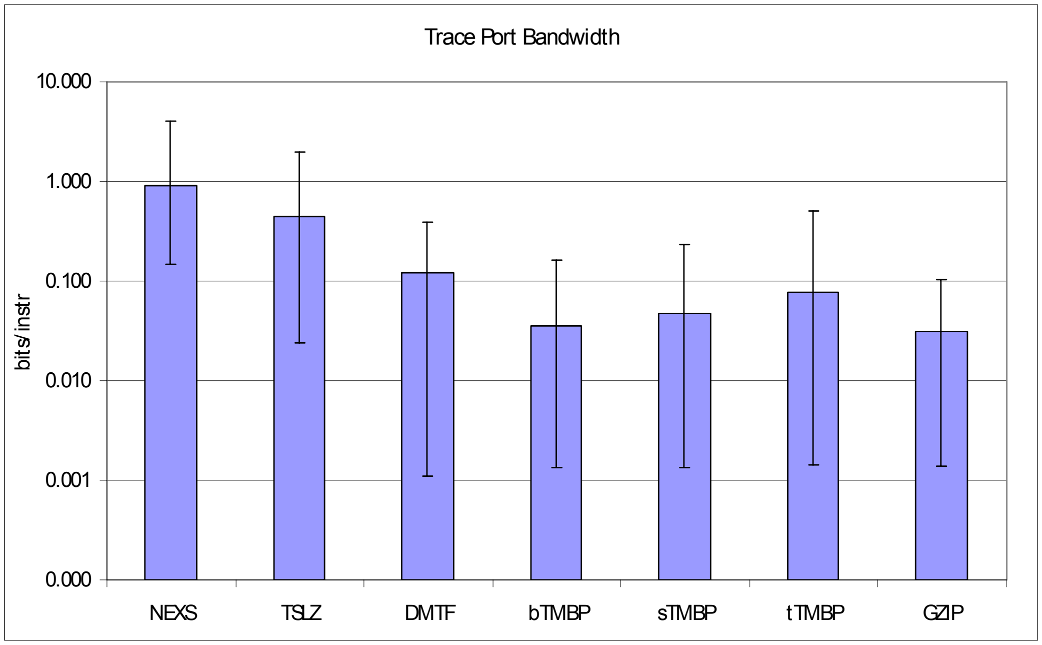


Figure 6. Trace port bandwidth comparison.

target address. The average trace port bandwidth required for the NEXS scheme is 0.907 bits/ins (close to the reporting bandwidths of commercial trace modules), ranging from 0.149 bits/ins for *adpmc_c* to 4.01 bits/ins for *bf_e*. Assuming a CPU core that can execute one instruction per clock cycle (IPC=1), and a trace port working at the processor clock speed, we would need at least 5 data pins on the trace port to trace the program execution unobtrusively (the worst case *bf_e* requires over 4 bits/ins).

The TSLZ compressor encompasses three stages: filtering of branch and target addresses, then difference-based encoding, and finally hardware-based LZ compression. We implemented this compressor and analyzed its performance on our set of benchmarks. The TSLZ configured with a sliding window of 256 12-bit entries requires 0.446 bits/ins on the trace port on average (ranging from 0.024 to 1.96 bits/ins). This compressor's complexity is estimated to be 51,678 logic gates [9]. The enhanced DMTF compressor encompasses two stages, each featuring a history table performing the move-to-front transformation. The compressor with a 192-entry first level and a 4-entry second level history table, eDMTF(192,4), requires on average 0.118 bits/ins on the trace port (ranging from 0.001 to 0.306 bits/ins). These two schemes reduce the trace port bandwidth, but they rely on fully-associative search tables that increase the cost of a hardware implementation and the compression latency. In addition, the worst performing benchmarks for TSLZ still require more than a single bit per instruction. Increasing the size of the search tables could alleviate this problem, but at a further increase in hardware complexity.

These results show that our technique requires a very small trace port bandwidth. The base configuration of our compressor (*bTMBP*) requires only 0.0356 bits/ins, which is a 28-fold improvement compared to the typical bandwidth of commercial state-of-the-art trace modules (~1 bits/ins [8]). It outperforms the best reported hardware compressor *eDMTF(192,4)* by over a factor of three (3.3) with an almost 5-fold reduction in complexity. We further observe that the compression ratio achieved by the *bTMBP* configuration is close to that of the software gzip utility when compressing a sequence of (SL,TA/-) pairs, which further underscores the strength of the proposed mechanism.

Our smallest configuration (*tTMBP*) requires only 0.0764 bits/ins on average (ranging from 0.0014 to 0.51 bits/ins) on the trace port, outperforming the enhanced DMTF scheme over 1.6 times with an order of magnitude lower complexity (2,800 vs. 24,600 logic gates). The *sTMBP* configuration benefits from the indirect branch target buffer and requires only 0.0467 bits/ins.

Table 2. Trace port bandwidth analysis [bits/ins].

	NEXS	***TSLZ***	***DMTF***	***bTMBP***	***sTMBP***	***tTMBP***	***GZIP***
adpcm_c	0.1486	0.0237	0.0011	0.0013	0.0013	0.0014	0.0014
bf_e	4.0102	0.3538	0.2840	0.0093	0.0094	0.0111	0.0377
cjpeg	0.7523	0.4312	0.0906	0.0420	0.0435	0.0623	0.0497
djpeg	0.3656	0.2298	0.0522	0.0205	0.0239	0.0354	0.0191
fft	1.5545	1.9208	0.2011	0.0909	0.0963	0.1664	0.0648
ghostscript	1.5776	1.3938	0.3060	0.1272	0.2298	0.5125	0.0381
gsm_d	0.5672	0.1518	0.0396	0.0129	0.0132	0.0132	0.0091
lame	0.3910	0.1706	0.1130	0.0288	0.0289	0.0282	0.0405
mad	0.6678	0.2678	0.1475	0.0331	0.0332	0.0349	0.0418
rijndael_e	0.8400	0.0426	0.0960	0.0158	0.0696	0.0779	0.0127
rsynth	0.7467	0.2707	0.1080	0.0208	0.0208	0.0227	0.0182
sha	0.5666	0.4414	0.3872	0.0218	0.0218	0.0297	0.0053
stringsearch	1.9319	1.9617	0.0489	0.1644	0.1829	0.3031	0.1044
tiff2bw	0.6543	0.1460	0.0114	0.0065	0.0106	0.0127	0.0063
tiff2rgba	0.3296	0.1597	0.0060	0.0075	0.0105	0.0153	0.0053
tiffdither	0.6588	0.5733	0.0118	0.0618	0.0621	0.0617	0.0801
tiffmedian	0.3740	0.0810	0.1656	0.0070	0.0078	0.0086	0.0068
Average	***0.9066***	***0.4462***	***0.1186***	***0.0356***	***0.0467***	***0.0764***	***0.0307***

Table 3 shows the hit rates for the predictor structures used in the three configurations of the proposed trace compressor. Coupled with the frequencies of each branch instruction type (direct/indicted, conditional/unconditional), the hit rates indicate the number of trace records that require tracing on the trace port. We can see that even the fairly small structures we use in this study achieve very high prediction rates. Even higher prediction rates may be achievable using more sophisticated predictors, thus further reducing the required trace port bandwidth. Designers of debugging infrastructure may configure their trace module predictor structures in such a way to minimize the complexity and trace port bandwidth while exploiting unique characteristics of the software under test.

Table 3. Branch predictor hit rates.

	Outcome Predictor	*Target Predictor*		
	GShare (512 ent.)	*RAS (8 ent.)*	*iBTB (16x2)*	*iBTB (32x2)*
adpcm_c	0.999	0.999	0.999	0.999
bf_e	0.984	1.000	1.000	1.000
cjpeg	0.921	0.599	0.931	0.957
djpeg	0.950	0.383	0.755	0.851
fft	0.906	0.807	0.931	0.945
ghostscript	0.948	0.285	0.664	0.835
gsm_d	0.973	0.983	0.986	0.991
lame	0.871	0.983	0.986	0.987
mad	0.908	0.973	0.978	0.978
rijndael_e	0.951	0.722	0.777	0.998
rsynth	0.945	0.996	0.998	0.999
sha	0.951	0.747	0.996	0.996
stringsearch	0.916	0.502	0.724	0.772
tiff2bw	0.996	0.467	0.588	0.781
tiff2rgba	0.993	0.486	0.696	0.797
tiffdither	0.909	0.979	0.983	0.987
tiffmedian	0.980	0.588	0.697	0.800
Average	***0.942***	***0.853***	***0.904***	***0.953***

5. CONCLUSIONS

This paper introduces a novel low-cost technique for real-time and unobtrusive tracing of program execution in embedded computer systems. The proposed trace module tracks the program execution by maintaining branch predictor-like structures that are updated during program execution akin to regular branch predictors. The debugger maintains a software version of these structures and employs the same policies as the trace module. The trace module needs to record only mispredictions in the predictor structures, which is why the proposed technique compresses traces well. We also introduce new, highly-effective variable encoding schemes for misprediction events.

The experimental evaluation shows that the proposed technique requires a very low trace port bandwidth, providing over an order of magnitude improvement over the commercial state-of-the-art and over a three-fold improvement over recent academic proposals at much lower cost. Our base configuration *bTMBP* requires only 0.0356 bits/ins on the trace port (i.e., a 898:1 compression ratio) at the cost of 5,200 logic gates and 0.0764 bits/ins (419:1 compression ratio) at the cost of 2,800 logic gates, allowing designers to perform trade-offs between the required trace port bandwidth and the trace module complexity.

6. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable suggestions. This work was supported in part by a National Science Foundation grant CNS-0855237.

7. REFERENCES

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

[2] McDonald-Maier, K. D. and Hopkins, A. B. T., "An Awakening Thought: Don't Let the Bug Bite While You Are Embedded," *Embedded Systems Engineering* (2004), 32-33.

[3] Hopkins, A. B. T. and McDonald-Maier, K. D., "Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores," *IEEE Trans. Comput.* 55, 2 (Feb. 2006), 174-184. DOI= http://dx.doi.org/10.1109/TC.2006.22

[4] ARM. Embedded Trace Macrocell Architecture Specification, ARM IHI 0014O (2007). http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014o/IHI0014O_etm_v3_4_architecture_spec.pdf

[5] MIPS. MIPS PDtrace Specification, MD00439 (2009). http://www.mips.com/products/product-materials/processor/mips-architecture/

[6] Infineon. TC1775 System Units 32-Bit Single-Chip Microcontroller, User's Manual, V2.0 (2001). www.infineon.com

[7] IEEE-ISTO. The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface, (2003). http://www.nexus5001.org/standard

[8] Orme, W. Debug and Trace for Multicore SoCs, (Sep. 2008). ARM White Paper. http://www.arm.com/pdfs/CoresightWhitepaper.pdf

[9] Kao, C.-F., Huang, S.-M., and Huang, I.-J., "A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors," *IEEE Transactions on Circuits and Systems* 54, 3 (Mar. 2007), 530-543.

[10] Uzelac, V. and Milenkovic, A., "A Real-Time Program Trace Compressor Utilizing Double Move-to-Front Method," in *Proceedings of the 46th Annual Design Automation Conference* (San Francisco, California 2009). DAC '09. ACM, 738-743. DOI= http://doi.acm.org/10.1145/1629911.1630102

[11] Uzelac, V., Milenković, A., Milenković, M., and Burtscher, M., "Real-time, Unobtrusive, and Efficient Program Execution Tracing with Stream Caches and Last Stream Predictors," in *International Conference on Computer Design* (Lake Tahoe, California, USA 2009). ICCD '09. IEEE Press, 173-178.

[12] Perleberg, C. H. and Smith, A. J., "Branch Target Buffer Design and Optimization," *IEEE Trans. Comput.* 42, 4 (1993), 396-412. DOI= http://dx.doi.org/10.1109/12.214687.

[13] Driesen, K. and Hölze, U., "Accurate indirect branch prediction," *SIGARCH Comput. Archit. News* 26, 3 (1998), 167-178. DOI= http://doi.acm.org/10.1145/279361.279380.

[14] Kaeli, D. R. and Emma, P. G., "Branch history table prediction of moving target branches due to subroutine returns," *SIGARCH Comput. Archit. News* 19, 3 (1991), 34-42. DOI= http://doi.acm.org/10.1145/115953.115957.

[15] Gochman, S., Ronen, R., Anati, I., Berkovits, A., Kurts, T., and Naveh, A., "The Intel Pentium M Processor: Microarhitecture and Performance," *Intel Technology Journal* 7, 2 (May 2003), 21-36.

[16] Milenković, A. and Milenković, M., "An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams," *ACM Trans. Model. Comput. Simul.* 17, 1 (2007), 1-27. DOI= http://doi.acm.org/10.1145/1189756.1189758.

[17] Burtscher, M., Ganusov, I., Jackson, S. J., Ke, J., Ratanaworabhan, P., and Sam, N. B., "The VPC Trace-Compression Algorithms," *IEEE Trans. Comput.* 54, 11 (2005), 1329-1344. DOI= http://dx.doi.org/10.1109/TC.2005.186.

[18] Chen, I.-C. K., Coffey, J. T., and Mudge, T. N., "Analysis of branch prediction via data compression," *SIGOPS Oper. Syst. Rev.* 30, 5 (1996), 128-137. DOI= http://doi.acm.org/10.1145/248208.237171.

[19] Barr, K. C. and Asanovic, K., "Branch trace compression for snapshot-based simulation," in *International Symposium on Performance Analysis of Systems and Software* (Austin, TX, Mar. 2006). ISPASS '06. IEEE Computer Society, 25-36.

[20] Burtscher, M. and Jeeradit, M., "Compressing Extended Program Traces Using Value Predictors," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (Sep. 2003). PACT '03. IEEE Computer Society, 159-168.

[21] Austin, T., Larson, E., and Ernst, D., "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer* 35, 2 (Feb. 2002), 59-67. DOI= http://dx.doi.org/10.1109/2.982917.

[22] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B., "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization* (Austin, TX, Dec. 2001). IEEE Computer Society, 3-14. DOI= http://dx.doi.org/10.1109/WWC.2001.15

[23] McFarling, S., "Combining Branch Predictors," Digital Equipment Corporation WRL Technical Note TN-36, 1993.