Using Branch Predictors and Variable Encoding for On-the-Fly Program Tracing

Vladimir Uzelac, Aleksandar Milenković, *Senior Member, IEEE* ,
Milena Milenković, *Member, IEEE* , and Martin Burtscher, *Senior Member, IEEE*

Abstract—Unobtrusive capturing of program execution traces in real-time is crucial for debugging many embedded 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 on-the-fly. It relies on branch predictor-like structures in the trace module and corresponding software modules in the debugger to significantly reduce the number of events that need to be streamed out of the target system. Coupled with an effective variable encoding scheme that adapts to changing program patterns, our technique requires merely 0.029 bits per instruction of trace port bandwidth, providing a 34-fold improvement over the commercial state-of-the-art and a five-fold improvement over academic proposals, at the low cost of under 5,000 logic gates.

Index Terms—Compression technologies, real time and embedded systems, testing and debugging, tracing

1 Introduction

The ever-increasing hardware and software complexity and the tightening time to market impose a number of challenges to embedded system verification and debugging. According to one estimate, software developers spend between 50 and 75 percent of their development time debugging [1], and this already high fraction is likely to grow due to the current shift towards multi-core systems and parallel software. Yet, in spite of significant investments in software debugging and testing, it is estimated that the United States alone lose between $20 and $60 billion a year due to software bugs and glitches [1]. For example, a study found 77 percent of all electronic failures in automobiles to be due to software bugs [2]. The recent recalls in the automotive industry are a stark reminder of the need for improved software testing and debugging. To shorten development time and reduce development cost, programmers need better debugging tools.

Increasingly, developers of embedded systems rely on on-chip resources for program debugging, as are already present in most higher-end embedded processors. Traditional approaches to software debugging using software instrumentation or run-control debugging are often not allowed in such systems (e.g., automotive, avionics, space, or military) because they are too intrusive. For instance, some errors appear only when subtle timing requirements are violated. Such errors are hard to reproduce using instrumentation or run-control debugging, which alter the real-time characteristics of the system and thus may cause the bugs to not manifest themselves in the debug runs. Furthermore, it is often not practical to instrument critical code sections such as interrupt service routines. In many high-reliability systems, the final code is required to be tested absent of any instrumentation and certified in the production system (e.g., avionics). Tracking down bugs in production versions of code thus cannot rely on software instrumentation, and hardware tracing is the most important tool available. Last but not least, hardware tracing is very helpful for performance analysis. It allows designers to monitor system performance in production software without rebuilding or modifying software or changing the timing of the code, which is inevitable with software instrumentation.

The IEEE's Industry Standard and Technology Organization has developed a standard named Nexus 5001 [3] that defines functions and a general-purpose interface for software development and debugging of embedded processors. Nexus 5001 specifies four classes of debug operations; higher numbered classes progressively support more complex operations but require more on-chip resources. Class 1 provides basic debug features for run-control debugging, including single stepping, breakpoints, and access to processor registers and memory while the processor is stopped. It is traditionally implemented through a JTAG interface [4]. Class 2 provides support for nearly unobtrusive capturing and streaming out program execution traces (PETs) (control-flow) in real-time. Class 3 provides support for capturing and streaming out memory and I/O read/write data values and addresses, in addition to the program execution trace. Finally, Class 4 adds resources to support emulated memory and I/O accesses through the trace port.

- *V. Uzelac is with Tensilica Inc., 3255 Scott Boulevard #6, Santa Clara, CA 95054. E-mail: vladimir.uzelac@gmail.com.*
- *A. Milenković is with the Department of Electrical and Computer Engineering, the University of Alabama in Huntsville, 301 Sparkman Drive, Huntsville, AL 35899. E-mail: milenka@uah.edu.*
- *M. Milenković is with IBM, 380 Weatherford Dr. NW, Madison, AL 35757. E-mail: mmilenko@us.ibm.com.*
- *M. Burtscher is with the Department of Computer Science, Texas State University-San Marcos, 601 University Drive, San Marcos, TX 78666-4684. E-mail: burtscher@txstate.edu.*

Manuscript received 3 June 2011; revised 31 Mar. 2012; accepted 18 Oct. 2012; date of publication 10 Nov. 2012; date of current version 5 Mar. 2014.
Recommended for acceptance by T. Conte.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2012.267

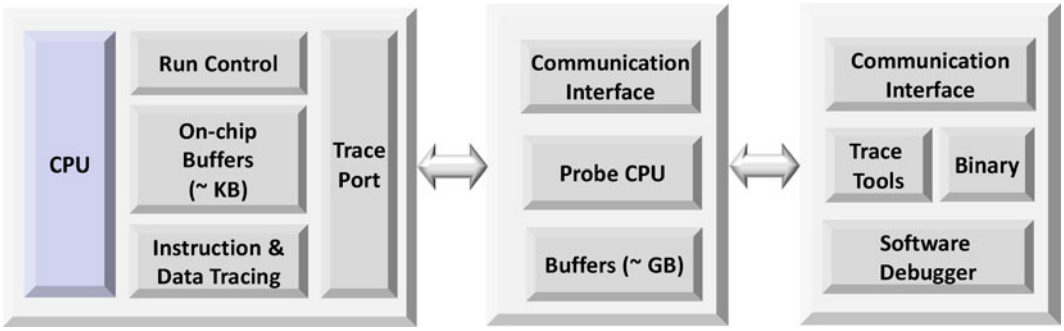


Fig. 1. Tracing and debugging in embedded systems: a system view.

Fig. 1 illustrates a typical embedded processor with its trace and debug module. It encompasses logic for run-control debugging (Class 1), logic to capture and filter program execution traces (Class 2) and data traces (Class 3), on-chip buffers for storing traces (on the order of kilobytes), and a trace port that connects the target system to an external trace unit (trace probe) or directly to a development workstation (host machine). The external trace probe typically includes a probe processor for control, a communication interface to the host (e.g., Ethernet or USB), and very large trace buffers (on the order of gigabytes). The host machine runs a software debugger and other trace processing tools that can read and analyze traces, allowing programmers to step forward and backward through the program execution. This way, programmers are able to gain complete visibility into the target system and its behavior while the target processor is running at full speed.

Whereas Class 1 debug operations are widely deployed and routinely used, they are lacking in several important aspects. First, setting breakpoints and examining the processor state to locate difficult and intermittent bugs in large software projects is demanding and time-consuming for developers. Second, setting a breakpoint is often not practical in debugging real-time embedded systems, e.g., it may be harmful for hard drives or engine controllers. Third, as discussed above, debugging through breakpoints interferes with program execution causing original bugs to disappear in the debug run.

Many vendors have recently introduced modules with program tracing capabilities that can be integrated into their platforms. They usually support Class 1 operations, often Class 2, and optionally Class 3. Some examples include ARM's Embedded Trace Module [5], [6], MIPS's PDTrace [7], and Tensilica's TRAX-PC [8]. Commercial trace modules require trace-port bandwidths in the range of 1 to 4 bits per instruction per core for program execution traces and 8 to 16 bits per instruction per core for data traces [9]. Thus, an internal 1 kilobyte trace buffer can capture the execution of a program segment of 2,000 to 8,000 instructions if a program execution trace is collected, or a program segment of 400 to 800 instructions if a data trace is collected. Such short segments are often insufficient for locating software errors in modern processors, where the distances between bug sources and their manifestations may be millions or even billions of instructions.

To support unobtrusive tracing in Class 2 and Class 3, the commercially available trace modules rely on hefty on-chip buffers and wide trace ports that can sustain streaming out large amounts of trace data in real-time. However, these resources significantly increase system complexity and cost, making embedded processor vendors reluctant to support higher classes of the Nexus 5001 standard. This problem is exacerbated in multi-core processors where the number of I/O pins dedicated to trace ports cannot keep pace with the exponential growth in the number of cores per chip. Hence, reducing the size of the output trace is critical to (a) lower the cost of on-chip debugging resources (smaller buffers and narrower trace ports), (b) enable unobtrusive tracing in real time, and (c) support debugging of processors with multiple cores.

In this paper we focus on program execution traces, i.e., on Class 2 operations in Nexus 5001. Program execution traces record the control flow of the program and are invaluable for hardware and software debugging as well as for program profiling. It should be noted that for certain classes of software bugs (e.g., data races), program execution traces alone are insufficient and data value traces are also required. However, program execution traces are still necessary in those cases, too—e.g., to capture exceptions. Because of the high costs, data tracing is typically done only on a limited program segment rather than on the entire program. Program execution traces and program check-pointing are used to pinpoint the program segment for which a full data trace is needed. Capturing and compression of data traces is thus out of the scope of this paper. More information on capturing and filtering of data traces in real-time can be found elsewhere [10].

Filtering and compressing program execution traces at runtime in hardware can reduce the requirements for on-chip trace buffers and trace port bandwidth. Whereas commercially available trace modules typically implement only rudimentary forms of hardware compression with a relatively small compression ratio (down to about 1 bit per instruction) [9], several recent research efforts in academia propose trace compression techniques that reach much higher compression ratios. For example, Kao et al. [11] 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 Milenković introduced a double move-to-front method that requires 0.12 bits per instruction on the trace port on average at an estimated cost of 24,600 logic gates [12]. A compressor using a stream descriptor cache and predictor structures requires a slightly higher

trace port bandwidth (0.15 bits per instruction) but has a much lower hardware complexity [13].

In this paper we introduce a new technique that combines hardware structures in the trace module and corresponding modules in the software debugger to enable very cost-effective compression of program execution traces in real-time (Section 3). The proposed trace module includes predictor structures that predict outcomes of conditional branches and target addresses of indirect branches (only those that cannot be inferred by the software debugger). Identical predictor structures are maintained in the software debugger. The key insight that leads to good compression is that to be able to replay the program execution offline, we only need to record the rare *misprediction* events in the trace module. We encode these events efficiently using a variable encoding scheme (Section 4) before they are streamed out of the chip through a trace port, thus further reducing the required bandwidth.

Our experimental evaluation shows that a trace module with a predictor configuration requiring fewer than 5,000 logic gates (a 512-entry outcome predictor, an eight-entry return address stack, and a 64-entry indirect branch target buffer (BTB)) results in only 0.0292 bits per instruction on the trace port (which is equivalent to a compression ratio of 1,098:1). We consider a range of branch predictor configurations and their impact on the trace port bandwidth. We also explore how to most effectively encode the trace messages and determine good encoding parameters that work well for a diverse set of benchmarks and for a range of trace module configurations (Section 5).

The main contributions of this work are as follows.

1. We propose using branch predictor like structures in the trace module for cost-effectively and unobtrusively capturing and compressing program traces at runtime.
2. We introduce an effective and low-complexity encoding scheme for the events that are captured at these hardware structures.
3. We perform a detailed experimental analysis that shows the proposed trace compression scheme to achieve excellent compression ratios, substantially outperforming existing hardware-based techniques for compression of program execution traces. It requires over 34 times less bandwidth on the trace port than commercial state-of-the-art solutions and over five times less than the best published academic proposal [13] at lower hardware cost.

Whereas the approach we propose in this paper shares some commonalities with the mechanism described in [13], such as utilizing cost-effective hardware structures in the processor's trace module and their counterparts in the software debugger, the proposed approach significantly outperforms the prior mechanism due to the following five reasons. (1) It employs smaller hardware structures akin to the processor's branch predictor to maintain the program's state instead of a stream descriptor cache and a last stream predictor. (2) The hardware structures work in parallel whereas the multi-level structures used in [13] work in series. (3) It reduces the number of trace messages that need to be communicated through the trace port by employing local hit counters. (4) It employs a variable trace record encoding scheme that further reduces the required trace port bandwidth for a range of benchmarks and predictor configurations. (5) It better handles benchmarks with a significant number of indirect branches.

The proposed method promises (a) to significantly reduce the cost of capturing and streaming out control-flow traces by eliminating the need for large on-chip trace buffers and wide trace ports, (b) to shorten the time software developers spend on debugging, and (c) to expedite certification and validation of production systems because entire program traces can be captured and later analyzed.

2 Program Execution Traces

Program execution traces are created by recording the program counter (PC) values of the committed instructions. However, to be able to replay a program offline in a software debugger with access to the program binary, we need to record only changes in the program flow, caused by control-flow instructions or exceptions during program execution. When a change in the program flow occurs, we need to capture (a) the 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. With this information, the program's execution path can be recreated from a sequence of <PC, BTA/ETA> pairs. To reduce the number of bits required to encode a <PC, BTA/ETA> pair, a program counter can be replaced by the number of instructions executed in a sequential run since the last change in the control flow (we call this number *stream length* or SL for short). Thus, a program's execution path can be represented by a sequence of <SL, BTA/ETA> pairs. However, even this already much smaller trace still contains redundant information that can be omitted. For example, the target address of a direct branch is known statically and can be inferred by the software debugger from the program binary. Consequently, in such cases only the stream length needs to be reported <SL, ->. Similarly, there is no need to report unconditional direct branches; their outcomes and targets are also known statically. However, in spite of these optimizations the number of bits that need to be streamed out through the trace port remains relatively large.

To illustrate the challenges associated with program execution tracing, we profiled 17 representative benchmarks from the MiBench suite [14]. The benchmarks are compiled for the ARM instruction set [15] and statistics are collected on the SimpleScalar functional simulator [16]. Table 1 shows the benchmark statistics of interest for capturing program flow information. The relatively low frequency of branch instructions (only *stringsearch* has more than 20 percent) is due to ARM's ISA support for conditional (predicated) instructions, which allows the compiler to reduce the number of control flow instructions. The last row shows statistics for the entire benchmarks suite.

The last column shows the required trace port bandwidth for the program execution trace when enough information necessary to reconstruct program execution is streamed out, i.e., the stream length and target address for indirect branches and exceptions. The bandwidth is

TABLE 1
MiBench Program Statistics Related to Instruction Tracing

	IC [mil.]	DirUB [%]	DirCB [%]	IndUB [%]	IndCB [%]	SWI [%]	PET bits/ins
adpcm_c	732.52	0.01	3.64	0.01	0.00	0.00	0.150
bf_e	544.06	5.66	6.73	5.55	0.00	0.00	4.913
cjpeg	104.61	1.19	9.11	0.17	0.15	0.00	0.790
djpeg	23.39	0.64	5.15	0.14	0.01	0.00	0.390
fft	631.04	2.30	11.27	1.78	0.49	0.00	1.895
ghostscript	708.10	2.22	12.15	2.18	0.34	0.00	1.814
gsm_d	1299.27	4.23	5.49	0.33	0.00	0.00	0.621
lame	1285.12	0.83	4.45	0.32	0.00	0.00	0.452
mad	287.09	0.76	5.57	0.57	0.20	0.00	0.785
rijndael_e	319.98	1.33	3.81	1.08	0.13	0.00	1.013
rsynth	824.94	1.31	5.70	0.67	0.00	0.00	0.883
sha	140.89	0.22	6.60	0.11	0.04	0.00	0.602
stringsearch	3.68	2.11	17.41	1.56	0.48	0.00	2.157
tiff2bw	143.26	0.06	7.76	0.08	0.02	0.01	0.668
tiff2rgba	151.70	0.08	3.53	0.11	0.03	0.01	0.349
tiffdither	832.95	1.15	14.29	0.21	0.00	0.00	0.692
tiffmedian	541.26	0.04	4.64	0.04	0.00	0.00	0.380
Total		*1.82*	*7.06*	*0.92*	*0.08*	*0.00*	*1.055*

IC—instruction count, DirUB—frequency of direct unconditional branches, DirCB—frequency of direct conditional branches, IndUB—frequency of indirect unconditional branches, IndCB—frequency of indirect conditional branches, SWI—frequency of software exceptions, and PET—program execution trace.

expressed as the average number of bits per executed instruction (bits/ins). The total trace port bandwidth is 1.05 bits/ins, which closely matches the bandwidth reported for commercial state-of-the-art trace modules [9]. The trace port bandwidth ranges from 0.15 to 4.91 bits/ins, depending on the frequency and type of control-flow instructions executed by the benchmarks. For example, in the *bf_e* benchmark, over 5.5 percent of all instructions are indirect branches, resulting in a high trace port bandwidth of 4.91 bits/ins. To be able to capture a program execution trace without stopping the target processor, we need either trace buffers that can hold a significant portion of the program execution or a wide trace port so the trace can be streamed out on the fly. An alternative is to further reduce the required bandwidth, as shown in the next section.

3 Program Tracing Using Branch Predictor Structures

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, a cache-like structure indexed by a portion of the branch PC [17] 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 [18]. A dedicated stack-like hardware structure called return address stack (RAS) is often used to predict return addresses [19]. 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 [20] found in recent commercial microprocessors [21]. Branch predictors are typically very effective, predicting branch outcomes and target addresses with high accuracy.

The concept of branch prediction can be used to dramatically reduce the amount of trace information that needs to be streamed out of the target platform. Assuming a software debugger that can replay control-flow instructions and includes a software model of the target machine's branch predictor (with the same organization and functionality), it is possible to replace the control-flow trace <SL, -/BTA/ETA> with a branch predictor trace. By maintaining its own copy of the branch predictor structures, the software debugger only requires trace messages when the target platform's branch predictor mispredicts. With the typically high prediction accuracy of branch predictors, this approach promises a dramatic reduction in the number of trace messages that needs to be communicated.

Ideally, we would be able to use the CPU's branch predictor and augment it with additional logic to compose trace messages, enabling control-flow tracing almost for free. Unfortunately, such an approach poses several challenges. First, tracing functionality is typically implemented in trace modules offered as intellectual property cores that connect to a processor core through a well-defined interface. Thus, tight integration of tracing infrastructure with the critical portion of the processor pipeline is not desirable. More importantly, support for tracing would place debilitating restrictions on the branch predictor's design and operation. For example, if the software debugger cannot replay kernel code (e.g., because the software debugger does not have a complete image of the system), 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. Next, we would need to disallow speculative updates of the branch predictor structures because they cannot be recreated on the software debugger side unless a detailed cycle accurate simulator of the target machine is available, which is impractical and/or economically infeasible. These restrictions would result in an unacceptable loss of accuracy of the branch predictor and therefore CPU performance, and are not further considered in this paper. Instead, we propose a trace module that incorporates branch predictor structures that are solely devoted to tracing. To distinguish it from the processor's branch predictor, we named it *T*racing B*ra*nch *P*redic*tor*, or T-raptor for short.

T-raptor includes structures for predicting branch targets and branch outcomes. Unlike regular branch predictors, T-raptor does not need to include a large BTB because direct branch targets can be inferred from the binary. Instead, it may include an iBTB for predicting targets of indirect branches, and a RAS for predicting return addresses. T-raptor structures are updated like regular branch predictors, but later in the pipeline, i.e., only when a branch instruction is retired. As long as the prediction from T-raptor corresponds to the actual program flow, the trace module does not need to send any trace records. *It reports only misprediction events*. These events are encoded and sent via a trace

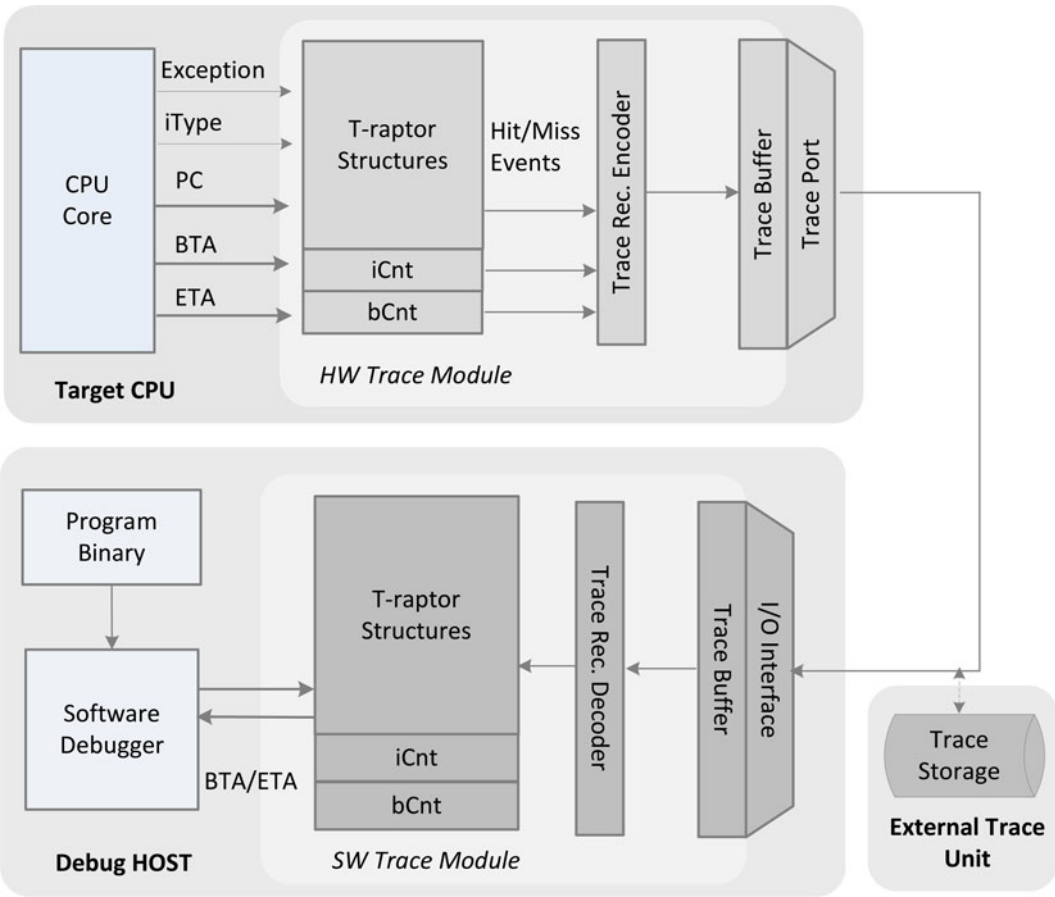


Fig. 2. System view of program-execution tracing using T-raptor.

port to a software debugger. The software debugger maintains an exact software copy of the T-raptor structures. It reads the branch predictor trace records, replays the control-flow instructions, and updates its branch predictor structures in the same way T-raptor is updated on the target platform during program execution. We assume that only retired instructions are passed to the trace module, allowing the software debugger to replay program execution utilizing a fast functional simulator for the given architecture. An alternative would be to capture speculative instructions as well, in which case we would need additional trace message to invalidate the prior trace messages and re-synchronize the state of the hardware structures.

Fig. 2 shows a system view of the proposed tracing mechanism. The trace module is coupled with the CPU core's instruction retirement unit through an interface that carries the relevant information for each instruction (PC, BTA, ETA, instruction type, exception). 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 retired 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 Fig. 3 for the trace module operation). The *iCnt* counter is incremented upon retirement of 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 (IndUB and IndCB). These branch instructions may be either correctly predicted or mispredicted by T-raptor. In case of a correct prediction, i.e., the frequent case, the trace module does nothing beyond the counter updates. In case of a misprediction, i.e., the infrequent case, the trace module generates a trace message that needs to be sent to the software debugger and clears the counters.

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

Fig. 3. T-raptor operation. iType—instruction type, PC—program counter, BTA—branch target address, ETA—exception target address, iCnt—instruction counter, bCnt—branch counter.

TABLE 2
Trace Module Branch Prediction Events and Content of Trace Records

Control-flow Type	*T-raptor Events*	*Trace Record Format*
DirCB	Outcome mispred.	<**bCnt**>
IndCB (NT)	Outcome mispred.	<**bCnt, NT**>
IndCB (T) or IndUB	Target mispred.	<**bCnt**, **T, BTA**>
Exception	--	<**iCnt**, **ETA**>

The type and format of the trace message depends on the branch type and the misprediction event type (Table 2). In case of a direct branch outcome misprediction, the trace record includes 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 (*BTA*). 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 T-raptor and the counters in the same way their hardware counterparts are updated (see Fig. 4); in particular, all branch instructions update the predictors. The debugger reads a trace message and then replays the program instruction-by-instruction. If it processes a non-exception trace message, the counter *bCnt* is decremented on 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. Then a new trace message is read to continue program replay. If the current instruction is an indirect branch, the debugger reads the outcome bit and possibly the target address from the trace message and redirects program execution 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.

Although this paper focuses on single-threaded benchmarks, the proposed method can be extended to support multi-threaded workloads through the addition of a 'thread switch' trace message. Akin to the exception trace record, this message would encompass an *iCnt* counter field and a thread identification number. In case of self-modifying code, T-raptor would require synchronization trace records that are emitted whenever a new region of the code is dynamically compiled. This synchronization message will need to be accompanied by the newly generated code from the Virtual Machine.

```
1.  // For each instruction
2.  Replay the current instruction;
3.  if (exception rec. is being processed) {
4.    iCnt--;
5.    if (iCnt == 0) {
6.      Goto Exception Handler Routine;
7.      Get the next trace record;
8.    }
9.  }
10. if (iType==AnyBranch) {
11.   Update software copy of T-raptor;
12.   if ((iType==IndBr) || (iType==DirCB)) {
13.     bCnt--;
14.     if (bCnt==0) Get the next trace rec.;
15.   }
16. }
```

Fig. 4. Execution replay in the software debugger.

3.1 Related Software-Based Trace Compression Techniques

A number of software-based trace compression techniques have been introduced [22], [23], [24], [25]. The relationship between data compression and branch prediction was first noted by Chen et al. [26]. Several recent *software*-based trace compression techniques rely on branch predictors [27] or, more generally, on value predictors [28]. 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 [27] 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 same underlying program characteristics as our scheme, there are some notable differences. First, their algorithm *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 Barr and Asanović's compression algorithm are different from our approach. Whereas we use a subset of regular branch predictor structures in the trace module and encode regular misprediction events, 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 summary, 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

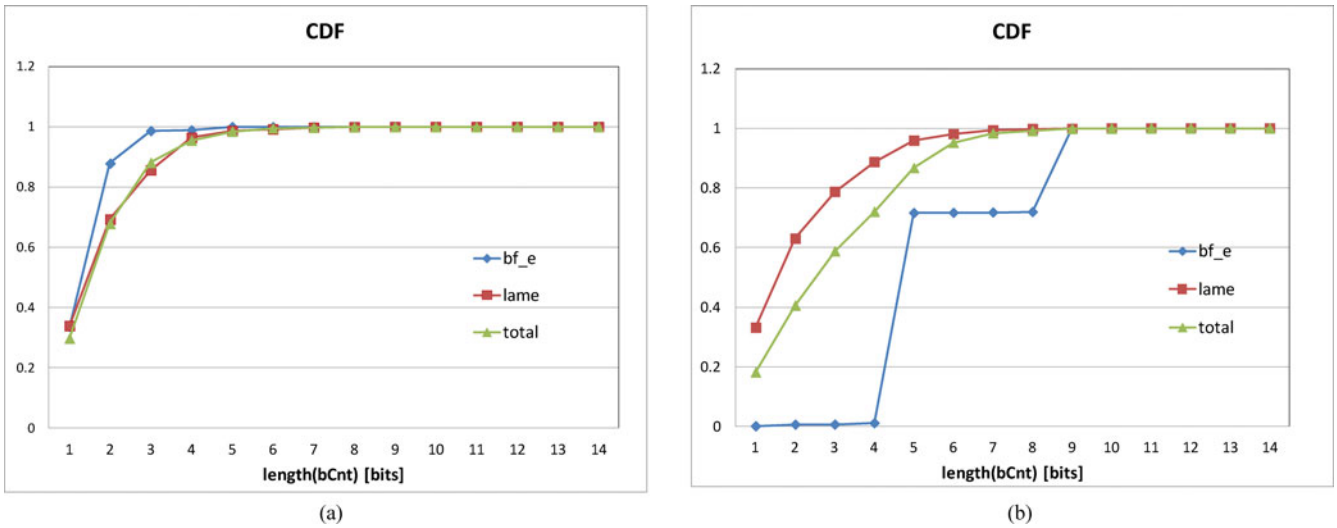


Fig. 5. Cumulative distribution function for the minimum *bCnt* length for the S0 (a) and B4 (b) configurations.

efficient encoding scheme that ensures unobtrusive tracing in real-time at minimal hardware cost. Our work strives to answer key questions concerning (a) the organization and implementation of the predictor structures in the trace module, (b) the efficient tracking of program behavior, (c) efficient adaptive encoding of trace messages, and (d) the overall performance of the proposed method.

4 Variable Encoding of Trace Messages

Trace messages should be encoded in a way that minimizes the trace port bandwidth requirements and enables simple and efficient implementation. A straightforward approach to encode the trace messages shown in Table 2 is to use fixed length fields for the counter values (*bCnt*, *iCnt*), the prediction bit (T/NT), and the target address field (BTA or ETA). However, using fixed-field formats is a suboptimal solution. The *bCnt* values in trace messages vary widely between programs and even within a program as it moves through different program phases (e.g., mispredictions are more likely during initialization due to cold misses in the predictor structures). Moreover, the *bCnt* values in the trace messages are heavily influenced by the T-raptor misprediction rate, which in turn is a function of the type and organization of the predictor structures. For example, a fixed eight-bit field can encode *bCnt* values from 1 to 255. However, we may have *bCnt* values that require more than 8 bits to encode. Moreover, a number of upper bits will often be unused, resulting in unnecessary waste in trace port bandwidth (e.g., when reporting $bCnt = 3$, the six upper bits would effectively be unused). Similarly to the *bCnt* values, the *iCnt* values also vary widely and are influenced by the frequency and the distribution of exception events. Thus, the challenge is to devise an encoding scheme that will work well across different benchmarks and configurations while minimizing the number of bits streamed out through the trace port.

To illustrate some of these encoding challenges, we profiled our benchmarks on a range of T-raptor configurations, starting from those that include only a small branch outcome predictor to those that include a larger outcome predictor, an indirect branch target buffer, and a return address stack. Fig. 5 shows the cumulative distribution function (CDF) for the minimum number of bits needed to encode the values found in the *bCnt* counter, referred to as *length(bCnt)*, for several characteristic benchmarks (*bf_e* and *lame*). The line marked *total* shows the CDF when all benchmark programs in our suite are taken into account. We consider two extreme T-raptor configurations, S0 (Fig. 5a) and B4 (Fig. 5b). The S0 configuration includes only a 256-entry gshare outcome predictor, whereas B4 includes a 1,024-entry gshare outcome predictor, a 64-entry iBTB, and an eight-entry RAS. The relatively low prediction rate for S0 results in frequent trace messages with small *bCnt* values that can be encoded with a small number of bits (see Fig. 5a). For example, considering the *total* CDF for the benchmark suite, we see that over 30 percent of all *bCnt* values can be encoded with a single bit ($bCnt = 1$, indicating a large number of consecutive misses in the branch predictor), over 70 percent can be encoded with two bits ($bCnt = 1, 2,$ *or* 3), and over 90 percent can be encoded with 3 bits. In contrast, the B4 configuration achieves a higher prediction rate, resulting in fewer trace messages with larger *bCnt* values (Fig. 5b). For example, 40 percent of all *bCnt* values can be encoded with two bits, over 70 percent can be encoded with four bits (*bCnt* values 1 to 15), and over 95 percent can be encoded with six bits. This shows that the predictor configuration impacts the *bCnt* profiles, and in some cases the change is quite significant. For example, *bf_e* with the *B*4 configuration has a very small number of mispredictions and over 70 percent of the *bCnt* values require exactly 5 bits. The remaining 30 percent of the *bCnt* values require exactly nine bits. This is quite a significant change relative to the profile of this program when using the S0 configuration. We further observe that different benchmarks exhibit very different profiles for the same configuration (e.g., compare *bf_e* and *lame* with the B4 configuration). Thus, to meet the encoding challenge, we opt for a variable encoding scheme and an empirical approach to determine good encoding parameters.

In our encoding scheme, all trace messages start with the field that contains the *bCnt* value. The length of this field is variable: after eliminating the leading zero bits, the *bCnt*

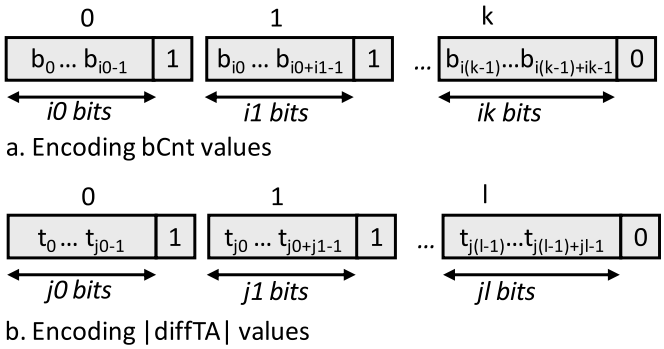


Fig. 6. Variable encoding of branch counters and target addresses.

counter bits are divided into a certain number of chunks, which do not necessarily need to be of equal size (see Fig. 6a). Each chunk is followed by a so-called connect bit (C) that indicates whether it is the terminating chunk for the *bCnt* field $(C = 0)$, or whether it is followed by more chunks with additional bits from the *bCnt* value $(C = 1)$. For example, a trace message that includes a 3-bit chunk '110' (the least significant bit of the chunk goes first) followed by a connect bit with value '0' indicates a misprediction event occurred on the third control-flow instruction from the previous event $(bCnt = 3)$. If the first chunk ends with a connect bit $C = 1$, more bits follow in the next chunk. Let us assume that the following chunk is also three bits long and its value is '010' and $C = 0$. This trace record thus specifies a *bCnt* value of '010_011' or 19 in decimal.

The length of individual chunks $(i0, i1, \ldots, ik)$ is a design parameter that should be determined empirically. In determining the length of individual chunks, we need to balance the overhead caused by the connect bits (shorter chunks result in a relative increase in the overall number of connect bits) and the number of wasted bits in individual chunks (longer chunks result in lower overhead due to connect bits, but may have more unused leading zero bits).

The trace records for mispredicted indirect branches contain information about the correct target address, in addition to the *bCnt* value. An alternative to sending an entire 32-bit address is to encode 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*, $|diffTA|$, we found that we can, indeed, shorten the trace records by using difference encoding. Fig. 7 shows the cumulative distribution function for the minimum number of bits needed to encode the $|diffTA|$ called $length(|diffTA|)$, for two benchmarks with a significant number of indirect branch instructions (*fft* and *ghostscript*). The line marked *total* represents the CDF for the entire benchmark suite. Similarly to the *bCnt* profiles, we again consider the two T-raptor configurations $S0$ (Fig. 7a) and $B4$ (Fig. 7b).

The results from Fig. 7 indicate that we rarely need more than 18 bits to encode the $|diffTA|$ field for our benchmarks, regardless of the T-raptor configuration. The slowly rising slopes of CDFs indicate that a variable encoding should be applied to encode the $|diffTA|$ field, too. Whereas the CDFs for individual benchmarks change with the T-raptor configuration, they generally follow similar trends. One may wonder about the source of a certain number of $|diffTA|$ values that can be encoded by a single bit as illustrated in Fig. 7b. This seeming anomaly stems from a certain number of hard to predict indirect branches that are consecutively mispredicted, that is, $|diffTA| = 0$. With the $B4$ configuration, the number of such branches is relatively high because the majority of the remaining indirect branches are correctly predicted by the iBTB and RAS structures. With the $S0$ configuration, all indirect branch target addresses are mispredicted and need to be streamed out. To implement variable encoding for $|diffTA|$ we can use a similar scheme as shown for the *bCnt* value (see Fig. 6b for illustration). The trace messages carrying the $|diffTA|$ field are followed by a sign bit that specifies whether the difference is a positive or a negative number.

An exception trace record starts with a single chunk where $bCnt = 0$, followed by a field that holds the value of the *iCnt* counter and the starting address of the exception handler (*ETA*). We employ variable encoding for both of

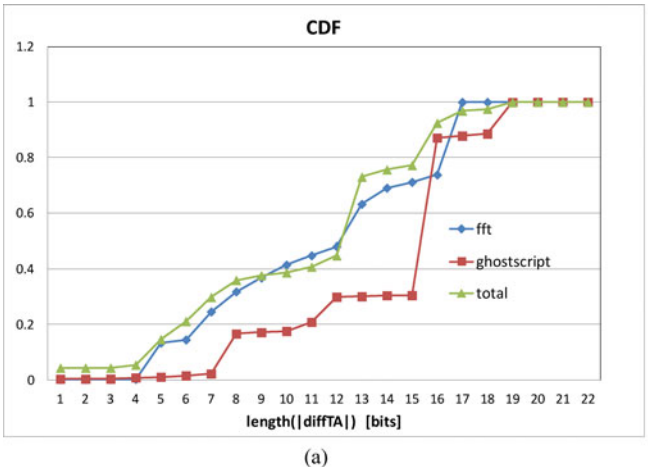
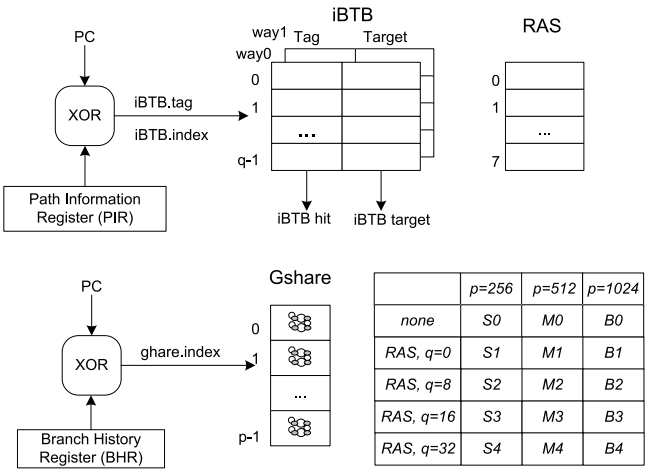


Fig. 7. Cumulative distribution function for minimum $|diffTA|$ length for the S0 (a) and B4 (b) configurations.



	p=256	p=512	p=1024
none	S0	M0	B0
RAS, q=0	S1	M1	B1
RAS, q=8	S2	M2	B2
RAS, q=16	S3	M3	B3
RAS, q=32	S4	M4	B4

Fig. 8. T-raptor organization and configurations.

these fields. A detailed analysis aimed at finding good values for chunk sizes is provided in the next section.

5 Experimental Evaluation

The goal of our experimental evaluation is to thoroughly explore the design space for the proposed trace module. We want to identify a T-raptor configuration that achieves maximum compression (or minimum trace port bandwidth) at minimal cost in hardware complexity. As a measure of performance, we use the average number of bits emitted on the trace port per instruction, which is equivalent to 32/(Compression Ratio) for the 32-bit ARM ISA. The trace port bandwidth is a function of (a) the prediction rates of the T-raptor structures, which in turn depend on the benchmark characteristics and predictors' size and organization, and (b) the encoding parameters. We explore a wide range of T-raptor configurations (15 in total), starting from those that include only a small outcome predictor, to those with large outcome predictors, a RAS, and an iBTB (Section 5.1). The selection of the encoding parameters and their impact on the trace port bandwidth is discussed in Section 5.2. We compare the trace port bandwidth of the proposed mechanism to the best pre-existing techniques in Section 5.3. Finally, we perform a complexity estimation and provide recommendations for configurations that strike a balance between complexity and compression ratio (Section 5.4).

5.1 T-Raptor Organization

Fig. 8 shows the T-raptor block diagram. For outcome prediction, T-raptor uses a global *gshare* outcome predictor in three sizes with $p = 256$ (configuration marked with S), 512 (M), and 1,024 (B) entries, each entry with a 2-bit counter. The index function is $gshare.index = BHR[log_2(p) : 0]xorPC[4 + log_2(p) : 4]$, where the BHR register holds the outcome history of the last $log_2(p)$ conditional branches.

For target address prediction of indirect branches, we consider five configurations, marked 0 through 4, which are: (0) no predictor structures dedicated to target address prediction; (1) an eight-entry RAS only, (2) an eight-entry RAS and a 16-entry iBTB, (3) an eight-entry RAS and a 32-entry iBTB, and (4) an eight-entry RAS and a 64-entry iBTB. All iBTB predictors are 2-way set-associative structures. Each entry in the iBTB includes a tag field and the target address. The tag and iBTB index are calculated based on information from a path information register (PIR) [29]. For example, a 64-entry iBTB uses a 13-bit PIR that is updated by relevant branch instructions as follows: $PIR[12 : 0] = ((PIR[12 : 0] \ll 2)xorPC[16 : 4])| \ Outcome$, where PC is the program counter and *Outcome* is the outcome bit of conditional branches. The iBTB tag and index are calculated as follows: $iBTB.tag = PIR[7 : 0]xorPC[17 : 10]$, and $iBTB.index = PIR[12 : 8]xorPC[8 : 4]$.

Table 3 shows outcome prediction rates for the three sizes of outcome predictor (S, M, and B) and target address prediction rates (M0-M4). We can see that only a few benchmarks benefit from an increased size of the outcome

TABLE 3
Outcome and Target Address Prediction Rates

	Outcome Prediction Rate			Target Address Prediction Rate				
	S	M	B	M0	M1	M2	M3	M4
adpcm_c	0.999	0.999	0.999	0.000	0.999	0.999	0.999	0.999
bf_e	0.982	0.984	0.984	0.000	1.000	1.000	1.000	1.000
cjpeg	0.916	0.923	0.928	0.000	0.599	0.923	0.945	0.967
djpeg	0.940	0.950	0.954	0.000	0.383	0.674	0.756	0.852
fft	0.860	0.908	0.937	0.000	0.807	0.914	0.916	0.949
ghostscript	0.896	0.948	0.959	0.000	0.285	0.409	0.607	0.974
gsm_d	0.965	0.973	0.976	0.000	0.983	0.984	0.991	0.993
lame	0.855	0.871	0.879	0.000	0.983	0.984	0.986	0.986
mad	0.888	0.914	0.926	0.000	0.973	0.975	0.975	0.977
rijndael_e	0.951	0.950	0.967	0.000	0.722	0.777	0.998	0.999
rsynth	0.938	0.945	0.947	0.000	0.996	0.998	0.998	0.999
sha	0.951	0.951	0.956	0.000	0.747	0.991	0.997	0.996
stringsearch	0.889	0.919	0.931	0.000	0.502	0.555	0.728	0.786
tiff2bw	0.996	0.997	0.997	0.000	0.467	0.467	0.612	0.901
tiff2rgba	0.992	0.993	0.994	0.000	0.485	0.595	0.763	0.881
tiffdither	0.900	0.909	0.918	0.000	0.979	0.979	0.981	0.986
tiffmedian	0.979	0.980	0.982	0.000	0.588	0.609	0.643	0.812

TABLE 4
Chunk Sizes for the *bCnt* and $|diffTA|$ Values

bCnt chunk sizes	Configurations		$\|diffTA\|$ chunk sizes	Configurations
i0 = 2, i1 = 1	S0, M0, B0		j0 = 8, j1 = 6, j2 = 6, j3 = 12	S0, M0, B0
i0 = 3, i1 = 1	S1, M1, M2		j0 = 1, j1 =7, j2 = 10, j3 = 14	S1, S2, S3, S4
i0 = 2, i1 = 2	S2, S3		j0 = 1, j1 = 11, j2 = 6, j3 = 14	M1, M2, M3, M4, B1, B2, B3, B4
i0 = 3, i1 = 2	S4, M3, M4, B1,B2, B3, B4			

predictor (*fft*, *ghostscript*, *mad*, and *stringsearch*); one benchmark has a relatively low prediction rate (*lame*) of about 86 percent, regardless of the predictor size. For indirect target address predictors, the RAS captures a large number of indirect target addresses (e.g., *bf_e*), and several other benchmarks benefit significantly from the iBTBs (*fft*, *ghostscipt*, *rinjndael_e*, and *tiff2bw*). It should be noted that more sophisticated predictors may be considered that would provide even higher prediction rates, but here we opted for a simple and straightforward design to prove that branch predictor structures are practical in compressing program execution traces.

5.2 Encoding Parameters Selection

To select good encoding parameters, i.e., chunk sizes, for the proposed variable encoding, we profiled the MiBench benchmarks using all T-raptor configurations (as shown in Fig. 5 for the S0 and B4 configurations). Whereas each benchmark has its own set of parameters that yields the minimal size of the output trace, we searched for parameters that minimize the size of the output trace when all benchmarks are considered. However, it should be noted that the proposed encoding makes benchmark-wise customization of chunk sizes practical – it can be accomplished before tracing through initialization of trace module control registers based on typical program profiles.

In the search for good values for chunk sizes $i0, i1, i2...ik$ (Fig. 6), we limited the design space by requiring that $i1 = i2 = \cdots = ik$. We vary the parameters $i0, i1 \in [1, 6]$ and $jm \in [1, 14], m = 0 \ldots l$. Table 4 lists the parameters that yield the minimal output trace sizes for all configurations. For the *bCnt* encoding, we can see that configurations that predict only outcomes (*S0*, *M0*, *B0*) favor shorter chunks $(i0 = 2, i1 = 1)$. Configurations with an iBTB and a RAS favor larger chunk sizes $(i0 = 2, i1 = 2; i0 = 3, i1 = 1;$ and $i0 = 3, i1 = 2)$. One interesting question is how important it is to use the encoding parameters that give the minimal output trace. All four pairs shown in Table 4 lay within 10 percent of each other, so selecting any of them will not cause dramatic changes in the trace port bandwidth. Other encoding parameters not listed in Table 4 may result in larger differences, though. In general, the sensitivity to variation in the encoding parameters is more pronounced for configurations with no predictor structures for target prediction (*S0*, *M0*, and *B0*).

Similarly, we analyzed the minimum bit length of the $|diffTA|$ field in search of a good set of encoding parameters (Fig. 7). Again, the selection is somewhat influenced by the T-raptor configuration. The three sets of parameters shown in Table 4 emerge as the most effective. The selected parameters result in output target address trace sizes that are within 12 percent of each other, indicating a certain level of stability in the variable encoding. To quantify the impact of the proposed encoding of the $|diffTA|$, we compare it to an encoding where entire 32-bit addresses are streamed out for target address mispredictions. We find that the proposed encoding almost halves the number of target address bits that needs to be streamed out through the trace port.

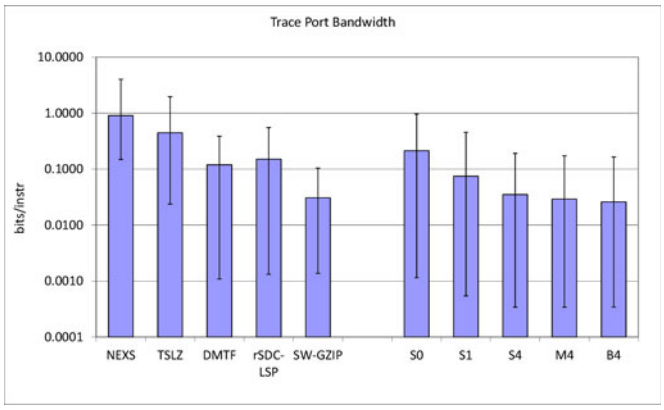


Fig. 9. Trace port bandwidth evaluation.

In spite of the relatively low frequency of exception events, we also analyzed the profiles for the *iCnt* counters to determine good encoding parameters. The profiles for software exceptions indicate that all *iCnt* values can be encoded using 2-bit chunk sizes.

5.3 Trace Port Bandwidth Analysis

Fig. 9 shows the total trace port bandwidth on our benchmark suite for several configurations of the proposed trace module ($S0$, $S1$, $S4$, $M4$, and $B4$) and several preexisting techniques (NEXS, TSLZ, DMTF, and SDC-LSP). Table 5 provides more detail by showing the average trace port bandwidth for each benchmark. We compare our technique with a Nexus-like trace module (NEXS) [3] and two trace-specific adaptations of general-purpose compression algorithms, namely the LZ scheme (TSLZ) [11] and the DMTF scheme [12], and a trace-specific compression method that uses stream cache and last stream predictor structures (rSDC-LSP) [13]. To illustrate the effectiveness of the proposed technique, we also compare it to the software gzip utility when compressing a sequence of <SL, -/BTA/ETA> pairs (SW-GZIP). 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 offline; it consists of a sequence of <SL, -/TA> pairs. 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 into groups of 6 bits. For example, 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 target address. The average trace port bandwidth required for the NEXS scheme is 0.907 bits/ins

TABLE 5
Trace Port Bandwidth: A Comparative Analysis

	NEXS	TSLZ	DMTF	rSDC-LSP	SW-GZIP	S0	S1	S4	M4	B4
adpcm_c	0.1486	0.0237	0.0011	0.0013	0.0014	0.0011	0.0005	0.0003	0.0003	0.0003
bf_e	4.0102	0.3538	0.2840	0.3452	0.0377	0.9628	0.0118	0.0099	0.0093	0.0093
cjpeg	0.7523	0.4312	0.0906	0.0884	0.0497	0.0936	0.0690	0.0426	0.0409	0.0382
djpeg	0.3656	0.2298	0.0522	0.0536	0.0191	0.0507	0.0428	0.0230	0.0209	0.0199
fft	1.5545	1.9208	0.2011	0.5538	0.0648	0.4658	0.1777	0.1118	0.0874	0.0711
ghostscript	1.5776	1.3938	0.3060	0.2161	0.0381	0.6620	0.4546	0.0856	0.0565	0.0483
gsm_d	0.5672	0.1518	0.0396	0.0515	0.0091	0.0837	0.0156	0.0152	0.0128	0.0122
lame	0.3910	0.1706	0.1130	0.1092	0.0405	0.0796	0.0318	0.0322	0.0283	0.0267
mad	0.6678	0.2678	0.1475	0.1170	0.0418	0.1277	0.0351	0.0355	0.0313	0.0273
rijndael_e	0.8400	0.0426	0.0960	0.1849	0.0127	0.2274	0.0964	0.0156	0.0155	0.0111
rsynth	0.7467	0.2707	0.1080	0.1488	0.0182	0.1419	0.0243	0.0225	0.0208	0.0203
sha	0.5666	0.4414	0.3872	0.0745	0.0053	0.0550	0.0267	0.0219	0.0218	0.0204
stringsearch	1.9319	1.9617	0.0489	0.4163	0.1044	0.4510	0.3154	0.1906	0.1727	0.1644
tiff2bw	0.6543	0.1460	0.0114	0.0308	0.0063	0.0203	0.0147	0.0046	0.0045	0.0042
tiff2rgba	0.3296	0.1597	0.0060	0.0124	0.0053	0.0266	0.0178	0.0060	0.0059	0.0056
tiffdither	0.6588	0.5733	0.0118	0.1589	0.0801	0.0895	0.0667	0.0668	0.0619	0.0572
tiffmedian	0.3740	0.0810	0.1656	0.0278	0.0068	0.0135	0.0094	0.0072	0.0070	0.0067
Total	**0.9066**	**0.4462**	**0.1196**	**0.1505**	**0.0307**	**0.2139**	**0.0748**	**0.0352**	**0.0292**	**0.0261**

(close to the reporting bandwidths of commercial trace modules), ranging from 0.149 bits/ins for *adpcm_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 five data pins on the trace port to trace the program execution unobtrusively (the worst case *bf_e* requires over 4 bits/ins on average).

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. 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 [11]. 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 four-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. The rSDC-LSP trace compressor with a 128-entry stream cache and a 128-entry last stream predictor requires on average 0.15 bits/ins, at much lower complexity of ~6,100 logic gates.

T-raptor demonstrates superior performance with even lower complexity. For example, configuration $S1$ (256 entries outcome predictor and an eight-entry RAS) requires only 0.0748 bits/ins on average on the trace port, which is a half of the bandwidth required by the rSDC-LSP. Configurations with no target address predictors for indirect branches ($S0$, $M0$, $B0$) perform poorly for all benchmarks with a significant number of indirect branches and are considered here only as border configurations. Our most complex configuration $B4$ requires only 0.0261 bits/ins on average, ranging from 0.0003 bits/ins (*adpcm_c*) to 0.16 bits/ins (*stringsearch*). It outperforms eDMTF(192,4) over 4.5 times and rSDC-LSP(128,128) over 5.7 times. We further observe that the compression ratio achieved by the M4 and B4 configurations even outperforms the software gzip utility when compressing a sequence of <SL,TA/-> pairs, which further underscores the strength of the proposed mechanism.

5.4 Hardware Complexity and Implementation Issues

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 percent of the cases in our benchmarks. Consequently, we can use a last value predictor for the upper 12 bits of the target address and keep only the lower 18 bits in the iBTB and RAS entries (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 iBTB and RAS prediction 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 the 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.

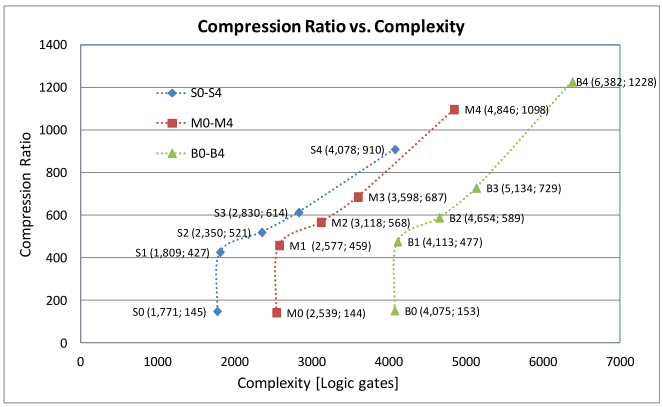


Fig. 10. Compression Ratio versus Complexity.

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 warm-up, when we experience a number of consecutive mispredictions in the *fft* and *ghostscript* benchmarks. For the $M4$ 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; this number is higher for configurations without target address predictors (up to 384 bits).

The estimates for the hardware complexity measured in logic gates are given in Fig. 10 and range between 1,771 logic gates for $S0$ to 6,382 for $B4$. These estimates confirm our expectations about the relatively small complexity of the proposed trace module compressor structures and support. All T-raptor configurations have much lower complexity than other competitive solutions, except $B4$, which has approximately same complexity as the rSDC-LSP(128,128).

Fig. 10 shows the compression ratio as a function of the complexity for all configurations. It allows us to trade compression ratio for complexity and thus meet design requirements. Clearly, configurations S0, $M0$, and $B0$ are not attractive design points. Since they require deeper trace buffers, they are almost as complex as configurations with an eight-entry RAS ($S1$, $M1$, $B1$). If we want to minimize complexity and trace port bandwidth (maximize compression ratio) and both are equally important, configurations $S1$ and $M4$ are good options. Similarly, if we value lower complexity more than trace port bandwidth, configuration $S1$ emerges as a top choice. Finally, if we want to minimize trace port bandwidth and do not worry about additional complexity, then $B4$ is the top candidate, followed by $M4$ and $S4$.

We do not expect the proposed mechanism to increase the overall energy expenditures caused by tracing. Whereas lookups in the predictor structures result in an additional energy overhead, these structures are relatively small (less than 5 Kgates). In contrast, preexisting solutions rely on large on-chip trace buffers and wide trace ports. Reads and writes into large on-chip buffers and streaming out a large amount of trace data through the trace port are by far the most expensive operations in terms of energy consumed. By dramatically reducing the size of the trace that needs to be streamed out and eliminating the need for large on-chip trace buffers, we expect the proposed method to reduce overall energy expenditures due to tracing activities.

6 Conclusions

This paper introduces a new 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 updates them during program replay using the same policies as in the trace module. The trace module needs to stream out only mispredictions in the predictor structures. Given the generally low misprediction rates of the predictor structures, the number of trace messages that needs to be reported is small, thus dramatically reducing the number of bits that needs to be traced out. We also introduce a highly-effective variable encoding scheme and optimize its parameters to further reduce the number of bits that needs to be streamed out.

Our experimental evaluation explores the design space of the proposed module, considering a range of predictor configurations and variable encoding parameters. For example, we find that a configuration with a 512-entry gshare outcome predictor, an eight-entry RAS, and a 64-entry iBTB requires a trace port bandwidth of only 0.0292 bits per committed instruction, which corresponds to a compression ratio of 1,098:1, at a hardware cost of only 4,846 logic gates. This bandwidth represents an over 34-fold improvement over the commercial state-of-the-art and an over 5-fold improvement over the best academic proposals at a much lower hardware cost.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable suggestions. This work was supported in part by US National Science Foundation (NSF) grants CNS-0855237, CNS-1217231, and CNS-1217470.

References

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

[2] K.D. McDonald-Maier and A.B.T. Hopkins, "An Awakening Thought: Don't Let the Bug Bite while You Are Embedded," *Embedded Systems Eng.*, vol. 12, p. 32-33, 2004.

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

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

[5] ARM, "Embedded Trace Macrocell Architecture Specification," http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014o/IHI0014 O_etm_v3_4_architecture_spec.pdf, 2007.

[6] ARM, "CoreSight On-Chip Debug and Trace Technology," http://www.arm.com/products/solutions/CoreSight.html, 2004.

[7] MIPS, "MIPS PDtrace Specification," http://www.mips.com/products/product-materials/processor/mips-architecture/, 2009.
[8] Tensilica, "Non-Intrusive Real-Time Trace Debug," http://www.tensilica.com/products/hw-sw-dev-tools/for-software-developers/real-time-trace-3.htm, 2009.
[9] W. Orme, "Debug and Trace for Multicore SoCs," http://www.arm.com/files/pdf/CoresightWhitepaper.pdf, 2008.
[10] V. Uzelac and A. Milenković, "Hardware-Based Data Value and Address Trace Filtering Techniques," *Proc. Int'l Conf. Compilers, Architectures and Synthesis for Embedded Systems*, pp. 117-126, 2010.
[11] C.-F. Kao, S.-M. Huang, and I.-J. Huang, "A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors," *IEEE Trans. Circuits and Systems*, vol. 54, no. 3, pp. 530-543, Mar. 2007.
[12] V. Uzelac and A. Milenkovic, "A Real-Time Program Trace Compressor Utilizing Double Move-to-Front Method," *Proc. 46th Ann. Design Automation Conf.*, pp. 738-743, 2009.
[13] A. Milenkovic, V. Uzelac, M. Milenkovic, and M. Burtscher, "Caches and Predictors for Real-Time, Unobtrusive, and Cost-Effective Program Tracing in Embedded Systems," *IEEE Trans. Computers*, vol. 60, no. 7, pp. 992-1005, July 2011.
[14] 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," *Proc. Fourth Ann. Workshop Workload Characterization*, pp. 3-14, 2001.
[15] ARM, "Architecture and Implementation of the ARM Cortex-A8 Microprocessor," http://www.arm.com/pdfs/TigerWhitepaperFinal.pdf, 2005.
[16] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp. 59-67, Feb. 2002.
[17] C.H. Perleberg and A.J. Smith, "Branch Target Buffer Design and Optimization," *IEEE Trans. Computers*, vol. 42, no. 4, pp. 396-412, Apr. 1993.
[18] K. Driesen and U. Hölze, "Accurate Indirect Branch Prediction," *ACM SIGARCH Computer Architecture News*, vol. 26, pp. 167-178, 1998.
[19] D.R. Kaeli and P.G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *ACM SIGARCH Computer Architecture News*, vol. 19, pp. 34-42, 1991.
[20] S. McFarling, "Combining Branch Predictors," Digital Equipment Corporation, 1993.
[21] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, and A. Naveh, "The Intel Pentium M Processor: Microarhitecture and Performance," *Intel Technology J.*, vol. 7, pp. 21-36, 2003.
[22] A.R. Pleszkun, "Techniques for Compressing Program Address Traces," *Proc. 27th Ann. Int'l Symp. Microarchitecture*, pp. 32-39, 1994.
[23] A. Milenkovic and M. Milenkovic, "An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams," *ACM Trans. Modeling and Computer Simulation*, vol. 17, pp. 1-27-39, 2007.
[24] M. Burtscher, I. Ganusov, S.J. Jackson, J. Ke, P. Ratanaworabhan, and N.B. Sam, "The VPC Trace-Compression Algorithms," *IEEE Trans. Computers*, vol. 54, no. 11, pp. 1329-1344, Nov. 2005.
[25] Y. Luo and L. K. John, "Locality-Based Online Trace Compression," *IEEE Trans. Computers*, vol. 53, no. 6, pp. 723-731, June 2004.
[26] I.-C.K. Chen, J.T. Coffey, and T.N. Mudge, "Analysis of Branch Prediction via Data Compression," *SIGOPS Operating Systems Rev.*, vol. 30, pp. 128-137, 1996.
[27] K.C. Barr and K. Asanovic, "Branch Trace Compression for Snapshot-Based Simulation," *Proc. Int'l Symp. Performance Analysis of Systems and Software*, pp. 25-36, 2006.
[28] M. Burtscher and M. Jeeradit, "Compressing Extended Program Traces Using Value Predictors," *Proc. 12th Int'l Conf. Parallel Architectures and Compilation Techniques*, pp. 159-169, 2003.
[29] V. Uzelac and A. Milenkovic, "Experiment Flows and Microbenchmarks for Reverse Engineering of Branch Predictor Structures," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software*, pp. 207-217, 2009.

Vladimir Uzelac received the BS degree in electrical engineering from the University of Belgrade in 2002 and the MS and PhD degrees in computer engineering from the University of Alabama in Huntsville in 2008 and 2010, respectively. In the meantime, he was a hardware design engineer for several years. He is currently an R&D engineer for embedded software and debugging architecture and tools working for Tensilica, Santa Clara.

Aleksandar Milenković received the DipIIng, MS, and PhD degrees in computer engineering and science from the University of Belgrade, Serbia, in 1994, 1997, and 1999, respectively. He is an associate professor of electrical and computer engineering at the University of Alabama in Huntsville, where he leads the LaCASA Laboratory (http://www.ece.uah.edu/∼milenka). His research interests include computer architecture, embedded systems, VLSI, and wireless sensor networks. Prior to joining the University of Alabama in Huntsville, he held academic positions at the University of Belgrade in Serbia and the Dublin City University in Ireland. He is a senior member of the IEEE, its Computer Society, the ACM, and Eta Kappa Nu.

Milena Milenković received the BS and MS degrees from the University of Belgrade and the PhD degree from the University of Alabama in Huntsville. Her research interests include performance evaluation, secure computer architectures, data compression, and architecture-aware compilers. She joined IBM in June 2005 as an advisory software engineer. She is a member of the IEEE, its Computer and Women in Engineering Societies, and the ACM.

Martin Burtscher received the combined BS/MS degree in computer science from the Swiss Federal Institute of Technology (ETH) Zurich in 1996 and the PhD degree in computer science from the University of Colorado at Boulder in 2000. He is an associate professor in the Department of Computer Science at Texas State University-San Marcos. His research interests include efficient parallelization of programs for GPUs and multicore CPUs, automatic performance assessment and optimization of HPC applications, and high-speed data compression. He has coauthored over 65 peer-reviewed research publications. He is a senior member of the IEEE, its Computer Society, and the ACM.