Caches and Predictors for Real-time, Unobtrusive, and Cost-Effective Program Tracing in Embedded Systems

Aleksandar Milenković, Vladimir Uzelac, Milena Milenković, Martin Burtscher

Abstract—The increasing complexity of modern embedded computer systems makes software development and system verification the most critical steps in the system development. To expedite verification and program debugging, chip manufacturers increasingly consider hardware infrastructure for program debugging and tracing, including logic to capture and filter traces, buffers to store traces, and a trace port through which the trace is read by the debug tools. In this paper, we introduce a new approach to capture and compress program execution traces in hardware. The proposed trace compressor encompasses two cost-effective structures, a stream descriptor cache and a last stream predictor. Information about the program flow is translated into a sequence of hit and miss events in these structures, thus dramatically reducing the number of bits that need to be sent out of the chip. We evaluate the efficiency of the proposed mechanism by measuring the trace port bandwidth on a set of benchmark programs. Our mechanism requires only 0.15 bits/instruction/CPU on average on the trace port, which is a six-fold improvement over state-of-the-art commercial solutions. The trace compressor requires an on-chip area that is equivalent to one third of a 1 kilobyte cache and it allows for continual and unobtrusive program tracing in real-time.

Index Terms—Compression Technologies, Real-time and Embedded Systems, Testing and Debugging, Tracing.

1 Introduction

TODAY'S society relies more than ever upon embedded computing, which drives modern communication, transportation, medicine, entertainment, and national security. The number of embedded processors by far surpasses the number of processors used for desktop and server computing. For example, a 2009 smartphone typically includes several processor cores [1], [2]; a 2009 luxury car may have over 70 different processors and microcontrollers [3]. The emerging ubiquitous computing and wireless sensor networks will lead to even higher proliferation and diversification of embedded processors. Current economic and technology trends present unique challenges to the design and the operation of embedded computer systems. Ever-increasing hardware complexity, limited visibility of internal signals due to increased integration, and reduced component reliability due to aggressive semiconductor technologies are only some of the most important challenges system designers of embedded computing platforms face. The time spent in post-silicon debug and verification has grown steadily as we move from one technology generation to the next [4]. Software designers of embedded systems face a number of challenging issues too, since increased hardware complexity enables more sophisticated applications. The software stack includes many layers, from hardware bring-up, low-level software, OS/RTOS porting, and application developing to system integration and performance tuning and optimization, production tests, in-field maintenance, and failure analysis. Growing software complexity often leads to lost revenue or even project failures if very tight time-to-market goals are not met. Software developers typically spend 50%-75% of their development time in program debugging [5]. This time is likely to grow with the shift from single- to multi-threaded applications. Hence, debugging and testing is already recognized as one of the most critical steps in the design and operation of modern embedded computer systems.

Ideally, system designers would like to be able to answer the simple question "What is my system doing?" at any point in the design and test cycle. However, achieving complete visibility of all signals in real time in modern embedded platforms is not feasible due to limited I/O bandwidth and high internal complexity. Moreover, even limited hardware support for tracing and debugging is associated with extra cost in on-chip area for capturing and buffering traces, for integration of these modules into the rest of the system, and for getting out the information through dedicated trace ports [6].

Debugging and testing of embedded processors is traditionally done through a JTAG port that supports two

- *A. Milenković is with the Department of Electrical and Computer Engineering, the University of Alabama in Huntsville, Huntsville, AL 35899. E-mail: milenka@uah.edu.*
- *V. Uzelac is with the Department of Electrical and Computer Engineering, the University of Alabama in Huntsville, Huntsville, AL 35899. E-mail: uzelacv@eng.uah.edu.*
- *M. Milenković is with IBM, Austin, Texas. E-mail: mmilenko@us.ibm.com.*
- *M. Burtscher is with the ICES, the University of Texas at Austin, Austin, TX 78712. E-mail: burtscher@ices.utexas.edu*

Manuscript received 14-Oct-2009; revised 15-Feb-2010; accepted 30-Apr-2010.

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

Recognizing these issues, many vendors have developed modules with tracing capabilities and have integrated them into their embedded platforms, e.g., ARM's Embedded Trace Macrocell [7], MIPS's PDTrace [8], and OCDS from Infineon [9] with a corresponding trace module from Freescale [10]. The IEEE's Industry Standard and Technology Organization has proposed a standard for a global embedded processor debug interface (Nexus 5001) [11]. Nexus defines four classes of operation: Class 1, Class 2, Class 3, and Class 4. Higher numbered classes progressively support more complex debug operations but require more on-chip resources. Class 1 provides basic debug features for run-control debugging (single stepping, breakpoints, watchpoints, and access to registers and memory while the processor is halted), typically implemented over IEEE 1149.1 (JTAG). Class 2 provides debug support for capturing program execution traces in real-time. Class 3 provides debug support for memory and I/O read/write traces. Class 4 provides resources for direct control of the processor over the trace port, where the instructions and data are supplied through the trace port rather than by the memory. The trace and debug infrastructure on a chip typically includes logic that captures the trace information, logic to filter and compress the captured information, buffers to store the traces, and logic that emits the content of the trace buffer through a trace port to an external trace unit or host machine. In this paper we focus on the capturing and compression of program execution traces (Class 2 operation in Nexus). Program execution traces consist of the addresses of all executed instructions. Such traces are crucial for both hardware and software debugging as well as for program optimization and tuning.

Many existing trace modules employ program trace compression and buffering to achieve a bandwidth of about one bit/instruction/CPU on the trace port at the cost of roughly 7,000 gates [12]. They rely on large on-chip buffers to store execution traces of sufficiently large program segments and/or on relatively wide trace ports that can transfer a large amount of trace data in real-time. However, large trace buffers and wide trace ports significantly increase the system complexity and cost. Moreover, they do not scale well (the I/O bandwidth does not increase at the same pace as the on-chip logic), which is a significant problem in the era of multicore chips. Whereas commercially available trace modules typically implement only rudimentary forms of hardware compression, several recent research efforts in academia propose compression techniques tailored to program execution traces that can achieve much higher compression ratios. For example, Kao et al. [13] 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 (51,678 gates), and it is unclear how effective it would be in tracing more diverse programs. Milenkovic et al. [14] propose new hardware structures to compress program execution traces that achieve 0.35 bits/instruction/CPU at a relatively modest hardware cost of two kilobytes of state.

In this paper, we introduce a very cost-effective mechanism for capturing and compressing program execution traces unobtrusively in real-time. The proposed trace compressor relies on two new structures, called stream cache and last stream predictor, to translate a sequence of program streams into a sequence of encoded hit and miss events in these structures. We also introduce several enhancements that reduce trace port bandwidth or compressor complexity and size. We explore trade-offs in the design of the proposed trace compressor and evaluate its overall effectiveness by measuring the average trace port bandwidth. Our experimental evaluation based on a set of representative benchmarks from the MiBench suite [15] shows that our best-performing trace compressor configuration with complexity equivalent to one third of a 1 kilobyte cache achieves a trace port bandwidth of 0.15 bits/instruction/CPU -- over six times lower than state-of-the-art commercial solutions. The proposed compression mechanism is orthogonal to data tracing required by Nexus' Class 3 operation. In fact, a good compressor for program execution traces benefits data tracing. First, reducing the trace port bandwidth consumed by program execution traces allows more bandwidth for data traces. Second, the internal trace buffers that would be used for storing of program execution traces compressed with a less sophisticated approach than ours can instead be devoted to storing data traces.

The reminder of the paper is organized as follows. Section 2 gives a system view of the proposed tracing mechanism and describes the trace compressor structures and their operation. Section 3 presents the results of the experimental evaluation, which includes a design space exploration and trace port bandwidth analysis. In addition, we describe several low-cost enhancements and explore their effectiveness. Section 4 compares the proposed schemes with existing solutions by analyzing the required trace port bandwidth and hardware complexity. Section 5 concludes the paper.

2 Trace Compression Using a Stream Descriptor Cache and a Last Stream Predictor

Program execution traces are created by recording the addresses of executed instructions. However, to replay a program flow offline we do not need to record each instruction address. Instead, we only have to record program flow changes, which can be caused by either control-flow instructions or exceptions. When a change in the program flow occurs, we need to know the address of the next instruction in the sequence; it is either the target address of the current branch[1] instruction or the starting address of an exception handler. Consequently, the program execution can be recreated from the recorded information about program streams, also known as dynamic basic blocks. An instruction stream is defined as a sequential run of instructions, from the target of a taken branch to the first taken branch in the sequence. Each instruction stream can be uniquely represented by its starting address (SA) and its length (SL). The pair (SA, SL) is called stream descriptor and replaces the complete trace of instruction addresses in the corresponding stream. Hence, the program execution can be replayed offline using the sequence of stream descriptors recorded during program execution.

A sequence of trace records with complete stream descriptors (SA, SL) includes some redundant information that can be directly inferred from the program binary during program replay. For example, if an instruction stream ends with a direct conditional branch, the next stream's starting address can be inferred from the binary and does not need to be traced out; instead, the next stream descriptor can include only information about its length (-, SL). Next, if an instruction stream ends with a direct unconditional branch, we can infer both the outcome and target from the binary. Thus, we do not need to terminate the current stream at such an instruction; rather, the current stream continues at the branch target. We use these two modifications to further reduce the size and the number of the trace records that need to be communicated through the trace port. Let us first examine characteristics of program streams as defined here by profiling a set of representative benchmarks targeting embedded computer systems.

Most programs have only a small number of unique program streams, with just a fraction of them responsible for the majority of program execution [16], [17], [18]. Table 1 shows some important characteristics of MiBench [15] benchmarks collected using SimpleScalar [19] running ARM binaries. The columns (a-d) show the number of executed instructions in millions (*IC - instruction count*), the number of unique streams (*USC – unique stream count*), and the maximum (*maxSL*) and the average stream length (*avgSL*), respectively. The total number of unique streams traversed during program execution is fairly limited – it ranges from 341 (*adpcm_c*) to 6871 (*ghostscript*), and the average dynamic stream length is between 5.9 (*bf_e*) and 54.7 instructions (*adpcm_c*). The fifth column (e) shows the number of unique program streams that constitute 90% of the dynamically executed streams. This number ranges between 1 (*adpcm_c*) and 235 (*lame*), and is 78 on average. Note that all averages throughout the paper use a weighted average. A benchmark weight is directly proportional to the number of executed instructions in that benchmark divided by the total number of executed instructions in all benchmarks. We use 8 bits to represent the stream length, SL, because program streams in our benchmarks never exceed 256 instructions. However, the streams may be longer than 256 instructions for other programs. In this case we would use the maximum stream length as an additional condition for terminating a stream – when the SL reaches the limit, the stream is automatically terminated and a new stream started.

TABLE 1
MiBench Program Characteristics

	IC (mil.)	*USC*	*max SL*	*avg SL*	*CDF 90%*
adpcm_c	732.5	341	71	54.7	1
bf_e	543.9	403	70	5.9	22
cjpeg	104.6	1590	239	12.3	47
djpeg	23.4	1261	206	25.1	31
fft	631.0	846	94	10.5	209
ghostscript	707.9	6871	251	10.0	67
gsm_d	1299.3	711	165	19.5	33
lame	1285.0	3229	237	32.4	235
mad	286.9	1528	206	20.7	42
rijndael_e	319.7	513	77	21.0	45
rsynth	824.9	1238	180	17.6	49
stringsearch	3.7	436	65	6.0	48
sha	140.9	519	65	15.4	10
tiff2bw	143.3	1038	43	12.8	2
tiff2rgba	151.7	1131	75	27.7	2
tiffmedian	541.3	1335	92	22.3	5
tiffdither	833.0	1777	67	14.3	63
Average	**816.3**	**1791**	**145**	**21.6**	**77.8**
	(a)	*(b)*	*(c)*	*(d)*	*(e)*

The proposed tracing mechanism, illustrated in Fig. 1, is designed to exploit the observed program characteristics. The target platform executes a program on a processor core. The trace module is coupled with the processor core through a simple interface including the program counter (PC), branch type information (direct/indirect, conditional/unconditional), and an exception control signal. The trace module consists of relatively simple hardware structures and logic dedicated to capturing, compressing, and buffering program traces. The recorded trace is read out of the chip through a trace port. The trace records can be collected on an external trace unit for later analysis or forwarded to a host machine running a software debugger.

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

[1] For simplicity, we refer to all control-flow changing instructions as branch instructions.

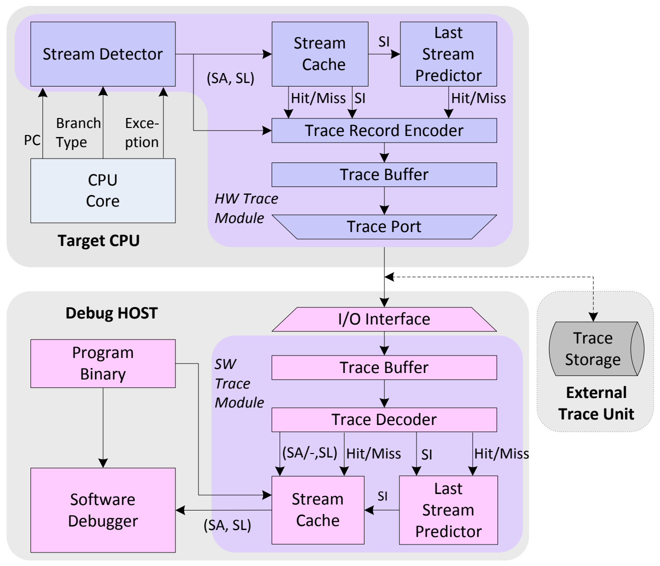


Fig. 1. System view of the proposed tracing mechanism.

conjunction with the program binary, provides enough information for a complete program replay off-line. The software debugger can control the trace module operation by initializing its configuration registers appropriately; for example, these registers determine how to handle interrupt service routines – tracing can continue if the software debugger is able to replay the code.

The proposed mechanism performs the capture and compression of program execution traces in three stages [20]. In the first stage, a stream detector module captures program streams. Instead of sending stream descriptors directly to the trace port, they are forwarded to compressor structures in the second and third stage. The second stage performs a stream cache transformation using a structure called stream descriptor cache (SDC). This transformation translates a stream descriptor into a stream index (SI). The stream index is used as an input to a predictor structure called a last stream predictor (LSP) in the third stage. Consequently, a sequence of stream descriptors coming from the stream detector is translated into a sequence of hit and miss events at the output of the SDC and LSP compressor structures. These events are efficiently encoded, thus significantly reducing the size of trace records that are stored in the trace buffer before they are sent to the trace port (Fig. 1). The following subsections describe the operations carried out in the stream detector, the stream descriptor cache, and the last stream predictor, respectively.

2.1 Stream Detector

The stream detector detects the end of the current stream and captures it by storing its descriptor (SA, SL) into a buffer whenever the new stream signal is asserted. To perform this task, the stream descriptor consists of an SA register that holds the starting address of the current stream, an SL register that tracks the current stream length, and a stream descriptor buffer (SDB). The SDB serves to amortize bursts of short program streams, and its depth should be large enough to prevent loss of stream descriptors when the compressor structures cannot keep up with the arrival rate of incoming program streams. Fig. 2 describes the steps performed by the stream detector and Fig. 3 (top, right) shows its organization.

The stream detector tracks the current program execution by monitoring the program counter and control signals coming from the CPU core. The SL register is incremented for each new instruction. The new stream signal is asserted when one of the following conditions is met: (a) the processor executes a control-flow instruction of a particular type, namely direct conditional, indirect conditional, indirect unconditional, or return; (b) an exception signal is asserted causing the program flow to depart from sequential execution, or (c) the maximum stream length has been reached. After forwarding the stream descriptor into the stream descriptor buffer, the stream detector prepares itself for the beginning of a new program stream by recording the starting address in the SA register and zeroing out the SL register.

```
// Stream detector operation; for each retired inst.
1.  if (NewStream) {
2.      SA = PC; SL = 0;
3.      NewStream = False;
4.  }
5.  if ((not ControlFlowChange) or (ControlFlowChange
        && (BranchType==DirectUncond))) {
6.      SL++;
7.      if (SL == MaxSL) {
8.           Terminate Stream;
9.           Place (SA, SL) into the Stream Descr. Buffer;
10.          NewStream = True; }
11. } else {
12.     SL++;
13.     Terminate Stream;
14.     Place (SA, SL) into the Stream Descr. Buffer;
15.     NewStream = True;
16. }

// Stream descriptor cache (SDC) operation
1.  Get the next stream descriptor, (SA, SL), from the
    Stream Descriptor Buffer;
2.  Lookup in the SDC with iSet = F(SA, SL);
3.  if (SDC hit)
4.    SI = (iSet concatenate iWay);
5.  else {
6.    SI = 0;
7.    if (SA is reached via an indirect branch)
8.       Prepare stream descriptor (SA, SL) for output;
9.    else
10.      Prepare stream descriptor (-, SL) for output;
11.   Select an entry (iWay) in the iSet to be replaced;
12.   Update stream descriptor cache entry:
      SDC[iSet][iWay].Valid = 1; SDC[iSet][iWay].SA = SA;
      SDC[iSet][iWay].SL = SL;
13. }
14. Update replacement indicators in the selected set;

// Last stream predictor (LSP) operation
1.  Get the incoming index, SI;
2.  Calculate the LSP index:
    pIndex = G(indices in the History Buffer);
3.  Perform lookup in the LSP with pIndex;
4.  if(LSP[pIndex] == SI)
5.    Emit('1');
6.  else {
7.    Emit('0' + SI);
8.    LSP[pIndex] = SI;
9.  }
10. Shift SI into the History Buffer;
```

Fig. 2. Stream detector, stream descriptor cache, and last stream predictor operation.

2.2 Stream Descriptor Cache

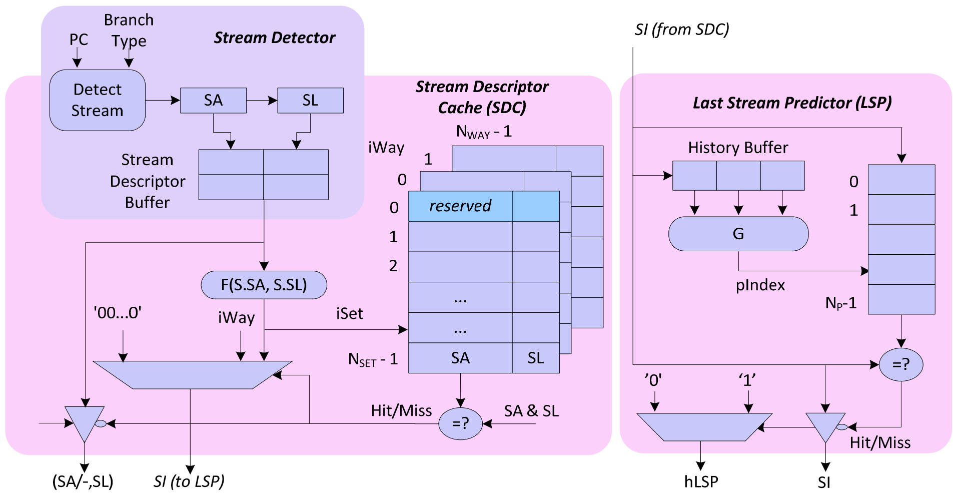


Fig. 3. Trace module structures: stream detector, stream descriptor cache, and last stream predictor.

The second stage consists of a cache-like structure called stream descriptor cache (SDC) that translates a stream descriptor into a relatively short stream index (SI). The stream cache is organized into N_{WAY} ways and N_{SET} sets as shown in Fig. 3. An entry in the stream descriptor cache holds a complete stream descriptor (SA, SL). Fig. 2 describes the sequence of steps carried out during the stream cache transformation by the SDC controller. The next stream descriptor is read from the stream descriptor buffer and an SDC lookup is performed. A set in the stream descriptor cache is calculated as a simple function of the stream descriptor, e.g., bit-wise XOR of selected bits from the SA and SL fields. If the incoming stream descriptor matches an entry in the selected set, we have an SDC hit event; otherwise we have an SDC miss event. In case of an SDC hit, the corresponding stream index, determined by concatenating the set and way indices (SI = {iSet, iWay}), is forwarded to the LSP. In case of an SDC miss, the reserved index zero is forwarded (SI = 0). If all entries of the selected set are occupied, an entry is evicted based on the replacement policy (e.g., LRU, FIFO), and it is updated with the incoming stream descriptor.

The compression ratio achieved by our stream detector and stream descriptor cache, CR(SDC), is defined as the ratio of the raw instruction address trace size, calculated as the number of instructions multiplied by the address size in bytes (IC*4), and the size of the SDC output (1). The SDC output size is a function of the number of executed program streams (calculated as IC/avgSL, where avgSL is the average dynamic stream length), the SDC hit rate (hrSDC), the SDC size (N_{SET}*N_{WAY}), and the probability that a stream starts with a target of an indirect branch (p_{IND}). For each program stream, 0.125*$\log_2$(N_{SET}*N_{WAY}) bytes are emitted to the SI output. On each SDC miss, a 5-byte (SA, SL) or 1-byte (-, SL) stream descriptor is output, depending on whether the corresponding stream starts with the target of an indirect or direct branch, respectively. The parameters avgSL and p_{IND} are benchmark dependent and cannot be changed except maybe through program optimization – e.g., favoring longer streams using loop unrolling or trace scheduling. Smaller stream caches require shorter indices but likely have a lower hit rate, which negatively affects the compression ratio. Thus, a detailed exploration of the SDC design space is necessary to determine a good hash function as well as a good size and organization (N_{SET} and N_{WAY}).

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

2.3 Last Stream Predictor

The third stage uses a simple last value predictor as shown in Fig. 3 to exploit redundancy in the SI trace component caused by repeating sequences of stream indices. A linear predictor table with N_P entries is indexed by a hash function that is based on the history of previous stream indices. If the selected predictor entry matches the incoming stream index, we have an LSP hit. Otherwise, we have an LSP miss, and the selected predictor entry is updated with the incoming stream index. The hit/miss information (one bit, a '0' for a miss and a '1' for a hit) and, in case of an LSP miss, SI ($\log_2$(N_{SET}*N_{WAYS}) bits) are forwarded to the trace encoder.

The compression ratio achievable by the LSP stage alone, CR(LSP), can be calculated as shown in (2). It depends on the stream index size and the LSP hit rate, hrLSP. The maximum compression ratio that can be achieved by this stage is $\log_2$(N_{SET}*N_{WAY}). The design space exploration for the last stream predictor includes determining a good hash function and a number of entries in the predictor N_P.

$$CR(LSP) = \frac{Size(SI)}{Size(LSP\ Output)} = \frac{1}{(1 - hrLSP) + 1/\log_2(N_{SET} \cdot N_{WAYS})} \quad (2)$$

2.4 Trace Record Encoder

The trace encoder assembles output trace messages based on the events in SDC and LSP as shown in Table 2. We distinguish three combinations of events in the compressor structures: (a) an LSP hit with an SDC hit, (b) an LSP miss with an SDC hit, and (c) an LSP miss with an SDC miss. The LSP cannot hit if the SDC misses. In case of an LSP hit with an SDC hit, the single-bit trace record '1' is placed into the trace buffer. In case of an LSP miss with

TABLE 2
TRACE RECORD ENCODINGS.

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

an SDC hit, the trace record starts with a '0' single-bit header and is followed by the value of the stream index that missed in the LSP. Finally, in case of an LSP miss with an SDC miss, the trace record consists of a '0' single-bit header, followed by a zero stream index that indicates a miss in the SDC, and a 40-bit (SA, SL) or 8-bit (-, SL) stream descriptor, depending on the type of branch that led to the beginning of the stream.

2.5 A Compression/Decompression Example

Let us consider the code snippet in Fig. 4 that includes a simple loop executing 100 iterations. The loop body consists of only one instruction stream. When the first iteration completes on the target platform, the stream detector captures the stream descriptor (SA, SL) = (0x020001f4, 9). Let us assume a 64-entry 4-way associative stream descriptor cache (NSET=16, NWAYS=4). The SDC indices are calculated as a function of certain bits of the stream descriptor; let us assume we calculate the iSet as follows: iSet = SA[7:4] xor SL[3:0]; in our case iSet=0x6. A lookup in the SDC set with index iSet=0x6 results in an SDC miss because the SDC entries are initially invalid. The least recently used entry in the selected SDC set is updated by the stream descriptor information (let us assume it is iWay=0), and the reserved 6-bit index 0 is output to the next stage (SI='000000'). A lookup in the LSP entry with index pIndex=0 results in an LSP miss because the LSP entries are also initially invalid. The LSP entry with index 0 is updated with the SI value. A complete trace record for the first occurrence of the stream includes a header bit '0' followed by the 6-bit index '000000' and the 40-bit stream descriptor (47 bits in total; here we assume that we need to output the starting address of the stream in spite of the fact that it can be inferred from the program binary). When we encounter the second iteration of the loop, the stream descriptor is found in the selected SDC set (an SDC hit). The SI is iSet concatenated with iWay, resulting in SI='011000' (iSet='0110' and iWay='00'). If we assume that the LSP predictor access is solely based on the previous SI (0 in our case), we will have another LSP miss. A trace record '0.011000' (h='0', SI='011000') is output to the trace buffer, and LSP's entry 0 is now updated with the value '011000'. The third loop iteration results in an SDC hit, and SI='011000' is forwarded to the LSP stage. The LSP will again miss (the entry pointed to by the previous SI is not initialized yet), and another trace record '0.011000' is sent to the trace buffer. The LSP entry with index '011000' is updated with the value '011000'. The fourth iteration hits in both the SDC and the LSP and only a single bit '1' is sent to the trace buffer. The next 95 iterations will also have only a single bit trace record to indicate both SDC and LSP hits. The final iteration does not hit because the loop end branch falls through and the stream length will therefore be larger than that of the previous streams.

The de-compressor on the debugger side reads the incoming bit stream from its trace buffer. The first bit in the trace is h='0', indicating an LSP miss event. The de-compressor then reads the next 6 bits from the trace that carry the SI='000000'. This index is reserved to indicate an SDC miss, and the de-compressor reads the next 40 bits from the trace to obtain the stream descriptor. The debugger updates the software copies of the SDC and LSP accordingly and replays 9 instructions starting at address 0x020001f4. The next step is to read the next trace record. It also starts with h='0', indicating an LSP miss. The next 6 bits are non-zero, which means that we have an SDC hit. The debugger retrieves the next stream descriptor from the SDC's entry SI='011000' and updates the SDC and LSP structures accordingly. The second iteration of the loop is replayed. Similarly, the debugger replays the third loop iteration. The fourth trace record starts with a header h='1'. This single-bit message is sufficient to replay the current stream. The software debugger retrieves the stream index from the LSP maintained in software (SI='011000') and, using this stream index, it retrieves the stream descriptor from the software copy of the stream cache. The debugger maintains its software copies of the compressor structures by updating the LSP's history buffer and SDC's replacement bits using the same policies as the hardware trace module does. The process continues until all iterations of the loop have been replayed.

```
// Code Snippet
        for(i=0; i<100; i++) {
          c[i] = s*a[i] + b[i];
          sum = sum + c[i];
        }
// Assembly listing of the code snippet for the ARM ISA
        @ 0x020001f4: mov  r1,r12, lsl #2
        @ 0x020001f8: ldr  r2,[r4, r1]
        @ 0x020001fc: ldr  r3,[r14, r1]
        @ 0x02000200: mla  r0,r2,r8,r3
        @ 0x02000204: add  r12,r12,#1 (1 >>> 0)
        @ 0x02000208: cmp  r12,#99 (99 >>> 0)
        @ 0x0200020c: add  r6,r6,r0
        @ 0x02000210: str  r0,[r5, r1]
        @ 0x02000214: ble  0x20001f4
// Trace records emitted per loop iteration
1.      h='0'; SI='000000'; (SA,SL)=(0x020001f4, 9)
2.      h='0'; SI='011000';
3.      h='0'; SI='011000';
4.      h='1';
5.      h='1';
6.      . . .
99.     h='1';
100.    h='0'; SI='000000'; (SA,SL)=(0x020001f4, ?)
```

Fig. 4. An example.

3 Experimental Evaluation

The goal of the experimental evaluation is twofold. First, we explore the design space to find good parameters for the proposed compressor structures and access functions (Section 3.1). As a measure of performance we use the average number of bits emitted per instruction on the trace port, which is equivalent to 32/(Compression Ratio), assuming 4-byte addresses. We also report the hit rates of the stream descriptor cache and the last stream predictor, hrSDC and hrLSP, because they directly influence the size of the output trace as explained in (1) and (2). Second, we introduce several enhancements to the original mechanism and explore their effectiveness in further improving the compression ratio at minimal added complexity (Section 3.2) or in reducing the trace module complexity (Section 3.3).

3.1 Design Space Exploration

SDC Access Function. A good hash access function should minimize the number of collisions in the SDC. Its efficacy depends on the program characteristics and SDC organization. We have evaluated a number of access functions while varying the SDC size and organization. We have found that access functions that combine the SA and SL portions of the stream descriptor in general outperform those based solely on the SA, because multiple streams can have the same starting address. Our experiments indicate that the hash function shown in (3) performs the best for different SDC sizes and configurations. The SA is shifted by *shift* bits and then the result is XOR-ed with the SL. The lower $\log_2 N_{SET}$ bits of the result are used as the set index, iSet. The optimal value for *shift* was found to be 4 for our benchmark suite. N_{SET} has to be a power of two.

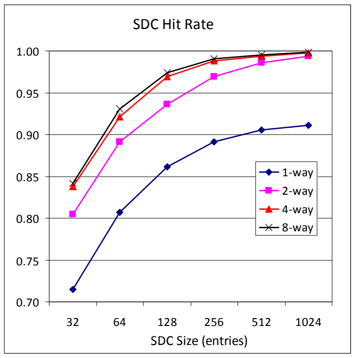
$$iSet = ((SA >> shift)\, XOR\, SL)\, AND\, (N_{SET} - 1) \tag{3}$$

SDC Size and Organization. Fig. 5 shows the SDC hit rate and the average trace port bandwidth required by the SDC alone and by the SDC in combination with LSP (SDC-LSP), when varying the number of entries and the number of ways (N_{WAYS} = 1, 2, 4, 8). The results reflect the weighted average for the whole benchmark suite. For SDC-LSP, the trace port bandwidth is calculated assuming an LSP with the same number of entries as the SDC and a simple hash access function that uses the previous stream index to access the LSP.

Let us first consider the SDC hit rate. The results show that increasing the SDC associativity improves the hit rate; for example, a 2-way SDC with 32 entries achieves the same hit rate as the direct-mapped SDC with 64-entries. However, increasing the associativity beyond 4 ways yields little or no benefit.

The results for the trace port bandwidth of the SDC-only method indicate that even relatively small stream descriptor caches with as few as 32 entries (8 x 4ways) perform well, achieving 0.49 bits/instruction (bits/ins). However, increasing the size of the SDC beyond 128 entries increases the trace port bandwidth, in spite of an increased hit rate. From (1) we can see that larger SDCs require longer stream indices, which outweigh the benefits of an increased hit rate. Thus, the compression ratio of SDC alone is fairly limited: the minimum trace port bandwidth (maximum compression) is ~0.45 bits/ins on our benchmarks. The SDC-LSP outperforms the SDC-only scheme for almost all sizes and organizations (except for a 32-entry direct-mapped SDC). Unlike SDC-only, it benefits from larger structures. Increasing the SDC and consequently the LSP size beyond 256 entries is not beneficial, as it only yields diminishing returns. Based on these results, we choose a 128-entry 4-way associative SDC and a 128-entry LSP as a good configuration for our trace compressor. This configuration represents a sweet spot in the trade-off between trace port bandwidth and design complexity; with our benchmarks, it yields under 0.2 bits/ins at a modest cost. We have also evaluated several SDC replacement policies, including pseudo-Random, First-In First-Out (FIFO), Least Recently Used (LRU), and several pseudo-LRU implementations. The results indicate that a pseudo-LRU replacement policy based on using Most Recently Used (MRU) bits performs best, outperforming even the full LRU policy.

Last Stream Predictor. We have considered several LSP organizations. The number of entries in the LSP may exceed the number of entries in the stream descriptor cache. In such a case, the LSP access function should be based on the program path taken to a particular stream. The path information may be maintained in a history buffer as a function of previous stream cache indices. However, our experimental analysis indicates that such an approach yields fairly limited improvements in trace port bandwidth. The reason is that our workload has a relatively small number of indirect branches, and those branches mostly have a very limited number of targets taken dur-



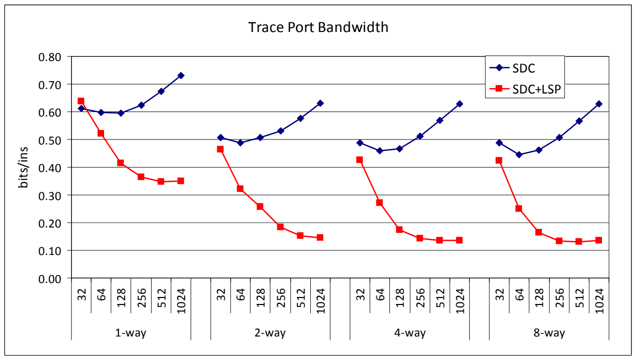


Fig. 5. SDC hit rate and trace port bandwidth as functions of SDC size and organization.

TABLE 3
HIT RATES AND TRACE PORT BANDWIDTH FOR THE BSDC+LSP SCHEME.

	hrSDC						hrLSP						bits/ins					
Program/Size	**32**	**64**	**128**	**256**	**512**	**1024**	**32**	**64**	**128**	**256**	**512**	**1024**	**32**	**64**	**128**	**256**	**512**	**1024**
adpcm_c	1.000	1.000	1.000	1.000	1.000	1.000	0.997	0.997	0.997	0.997	0.997	0.997	0.019	0.019	0.019	0.019	0.019	0.019
bf_e	0.985	0.996	1.000	1.000	1.000	1.000	0.824	0.838	0.843	0.843	0.843	0.843	0.405	0.359	0.357	0.384	0.410	0.437
cjpeg	0.952	0.991	0.998	0.999	1.000	1.000	0.894	0.916	0.921	0.922	0.922	0.922	0.204	0.138	0.131	0.134	0.140	0.146
djpeg	0.935	0.971	0.991	0.997	0.998	0.999	0.887	0.896	0.909	0.915	0.916	0.917	0.125	0.093	0.075	0.070	0.072	0.075
fft	0.482	0.685	0.859	0.952	0.985	1.000	0.522	0.674	0.762	0.827	0.850	0.870	1.492	1.007	0.616	0.354	0.256	0.219
ghostscript	0.456	0.778	0.987	0.993	0.997	0.999	0.518	0.696	0.865	0.868	0.869	0.870	1.585	0.823	0.232	0.227	0.229	0.234
gsm_d	0.972	0.980	0.989	0.996	0.999	1.000	0.946	0.946	0.947	0.951	0.952	0.954	0.103	0.094	0.086	0.077	0.076	0.075
lame	0.903	0.938	0.954	0.964	0.972	0.987	0.807	0.820	0.823	0.827	0.829	0.833	0.129	0.108	0.102	0.101	0.100	0.093
mad	0.833	0.972	0.984	0.993	0.998	1.000	0.715	0.825	0.830	0.832	0.835	0.836	0.295	0.136	0.129	0.124	0.124	0.128
rijndael_e	0.542	0.866	0.929	1.000	1.000	1.000	0.697	0.846	0.809	0.867	0.867	0.867	0.743	0.284	0.192	0.099	0.105	0.111
rsynth	0.848	0.923	0.966	0.997	1.000	1.000	0.843	0.843	0.860	0.883	0.887	0.887	0.382	0.245	0.175	0.116	0.115	0.122
sha	0.952	0.999	1.000	1.000	1.000	1.000	0.893	0.922	0.922	0.922	0.922	0.922	0.178	0.097	0.101	0.106	0.111	0.116
stringsearch	0.759	0.868	0.971	0.991	0.993	0.999	0.829	0.807	0.857	0.870	0.872	0.872	1.369	0.938	0.472	0.387	0.401	0.382
tiff2bw	0.971	0.979	0.992	0.998	1.000	1.000	0.996	0.993	0.992	0.992	0.994	0.994	0.151	0.135	0.104	0.087	0.083	0.084
tiff2rgba	0.935	0.969	0.996	1.000	1.000	1.000	0.991	0.978	0.989	0.989	0.989	0.989	0.114	0.077	0.045	0.040	0.040	0.040
tiffdither	0.824	0.904	0.963	0.988	0.997	1.000	0.834	0.823	0.848	0.864	0.870	0.873	0.332	0.249	0.190	0.164	0.157	0.160
tiffmedian	0.975	0.983	0.992	0.997	1.000	1.000	0.976	0.973	0.971	0.973	0.976	0.976	0.085	0.077	0.066	0.058	0.055	0.056
Average	**0.838**	**0.921**	**0.969**	**0.988**	**0.994**	**0.998**	**0.819**	**0.856**	**0.881**	**0.893**	**0.897**	**0.899**	**0.426**	**0.272**	**0.174**	**0.142**	**0.136**	**0.135**

ing program execution. Consequently, we chose the simpler solution of always having the same number of entries in the LSP and the SDC. The LSP access function is based solely on the previous stream cache index. We call this basic implementation of the proposed tracing mechanism bSDC-LSP.

Table 3 shows the stream descriptor cache hit rate (hrSDC), the last stream predictor hit rate (hrLSP), and the trace port bandwidth for individual benchmarks and for different sizes of the SDC and LSP. The average hit rate varies from 81.9% for 32-entry LSP to 89.9% for 1K-entry LSP. For some benchmarks (e.g., *fft*), capacity misses in the LSP limit the hit rate and they can benefit from larger structures. Another limitation comes from indirect branches with multiple targets that frequently change. The LSP predictor with its simple index function is not well-suited to handle them, but the number of such branches is typically small. The trace port bandwidth for the trace module configuration [32x4, 128] (4-way set-associative 128-entry SDC and 128-entry LSP) varies between 0.019 bits/ins for *adpcm_c* and 0.616 bits/ins for *fft*, and is 0.174 bits/ins on average for the whole benchmark suite. The *fft* benchmark significantly benefits from an increase in the SDC size and requires 0.354 bits/ins with the [64x4, 256] configuration. Many of the remaining benchmarks perform well even with very small configurations, e.g., *adpcm_c*, *tiffmedian*, and *tiff2rgba*.

3.2 Enhancements for Reducing Trace Port Bandwidth

The output trace records still contain a lot of redundant information that can be eliminated with low-cost enhancements. The three components of the output trace, as described in Table 2, are (i) LSP-hit records (hLSPt), (ii) LSP-miss with SDC-hit records (hSDCt), and (iii) LSP-miss and SDC-miss records (mSDCt). Table 4 shows distributions of the individual trace components for two trace module configurations: [16x4, 64] and [64x4, 256]. The mSDCt component dominates the output trace in smaller configuration; e.g., it is responsible for 41.3% of the total output for the [16x4, 64] configuration. By analyzing the sequence of mSDCt records, we observe that the upper address bits of the starting address (SA) field rarely change. We can use this property to reduce the number of bits that needs to be traced out. Moreover, with larger configurations, the hLSPt component dominates the output trace with long runs of consecutive ones; e.g., the hSLPt represents 48.5% of the total output trace for the [64x4, 256] configuration. We can use counters to encode these long runs of consecutive ones.

To take advantage of the redundancy in the mSDCt, we slightly modify our bSDC-LSP compressor as follows. An additional *u*-bit register called LVSA (Last Value Starting Address) is added to record the *u* upper bits of the SA field from the last miss trace record. The upper *u*-bit field of the SA of the incoming miss trace record is compared to the LVSA. If there is a match, the new miss trace record will include only the lower (32-*u*) address bits. Otherwise, the whole address is emitted and the

TABLE 4
DISTRIBUTION OF INDIVIDUAL TRACE COMPONENTS FOR TWO TRACE MODULE CONFIGURATIONS.

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

LVSA register is updated. To distinguish between these two cases, an additional bit in the trace record is needed to indicate whether all (SA[31:0]) or only the lower address bits (SA[31-*u*:0]) are emitted. The format of the trace record for an LSP miss with SDC miss event is modified to include this additional bit that precedes the stream descriptor field. Note that SA[1:0] is always '00' for the ARM ISA and is omitted from the mSDCt. For the ARM Thumb ISA only SA[0] can be omitted. These two bits do not need to be kept in the stream descriptor cache. In addition, we can also omit the address bits that can be inferred from the SDC index (this enhancement will be discussed further down).

Increasing the width of the LVSA register reduces the number of bits in the miss trace in case of LVSA hits; however, it also reduces the number of LVSA hit events. Table 5 shows the fraction of the original miss trace component for various values of the parameter *u* for the [32x4, 128] configuration. For example, we find that the LVSA enhancement reduces the miss trace component by 18% when *u*=14. It should be noted that the reduction in the total output trace is more significant for smaller trace module configurations and relatively insignificant for larger configurations, because the miss trace component is relatively small in the latter case.

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

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

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

Table 6 shows the trace port bandwidth of the eSDC-LSP scheme for individual benchmarks and for different sizes of the SDC and LSP. We observe relatively high improvements for small trace module configurations, mainly due to a reduction in the mSDCt size; for example, the average trace port bandwidth for the [8x4, 32] configuration is 0.35 bits/ins, down from 0.43 bits/ins in the bSDC-LSP scheme (18% lower). Similarly, for large trace module configurations the hLSPt size is significantly reduced; for example, the trace port bandwidth for the [64x4, 256] configuration is 0.12 bits/ins, versus 0.142 bits/ins in the bSDC-LSP scheme (a 15% reduction). Some benchmarks benefit significantly from this enhancement, especially those with a high LSP hit rate, such as *adpcm_c* (over 14 times lower bandwidth), *tiff2bw* (3.45), and *tiff2rgba* (3.67).

TABLE 5
FRACTION OF THE ORIGINAL MISS TRACE COMPONENT USING LVSA.

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

TABLE 6
TRACE PORT BANDWIDTH OF THE *eSDC-LSP* SCHEME.

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

3.3 Enhancements for Reducing Hardware Complexity

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

We can further reduce the number of bits kept in the stream descriptor cache without any negative impact on the trace module performance. The bits of the starting address SA[shift+$\log_2(N_{SET})$-1:shift] that are used in the calculation of the SDC index function (3) do not need to be kept in the SDC. This information can be inferred based on the known index function and SL bits that are stored in the SDC. (Alternatively, we can keep all address bits in the stream cache and eliminate the portion of the SL bits that are used for the SDC index.) For example, in the [32x4, 128] configuration, the iSet is calculated as the XOR result of SA[8:4] and SL[4:0]. Consequently, we can infer the value of SA[8:4] as SA[8:4] = iSet XOR SL[4:0]. The eSDC-LSP scheme with the modified LVSA enhancement and the reduced complexity SDC is called rSDC-LSP scheme.

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

TABLE 7
TRACE PORT BANDWIDTH OF THE *RSDC-LSP* SCHEME.

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

4 PUTTING IT ALL TOGETHER

In this section we evaluate the performance of the proposed schemes bSDC-LSP, eSDC-LSP, and rSDC-LSP relative to several alternative approaches. We measure the average trace port bandwidth requirements for our benchmark suite. While the average trace port bandwidth is a useful metric for comparison of different approaches to program tracing, it is important to determine the size of the trace buffer and the stream descriptor buffer so that the program tracing can be performed unobtrusively in real-time. We extend the SimpleScalar sim-outorder processor simulator with a model of the proposed trace module and use it to determine the minimum size of each buffer needed to guarantee unobtrusive tracing. Finally, we estimate the complexity of the proposed scheme and compare it to similar solutions available in the literature.

4.1 Trace Port Bandwidth Analysis

Fig. 6 shows the average, minimum, and maximum trace port bandwidths for the proposed schemes and alternative approaches, including base implementations of the trace module (fBASE and BASE), a Nexus-like implementation (NEXS) [11], and two recently proposed hardware-based trace compression schemes [13], [14]. For reference, we also show the results obtained by gzip, a widely used software compression utility (SW-GZIP). Table 8 shows the average trace port bandwidth for each scheme and each individual benchmark.

The fBASE scheme assumes sending a sequence of full stream descriptors (SA, SL) directly to the trace port, regardless whether the SA field can be inferred by the software debugger, whereas the BASE assumes the SA field is traced out only when it cannot be inferred (SA/-, SL). The sequence of stream descriptors is equivalent to the one captured at the output of our stream detector unit. The required average trace port bandwidth for fBASE is 2.51 bits/ins, ranging from 0.73 for *adpcm_c* to 6.80 bits/ins for *bf_e*. The trace port bandwidth for BASE is 1.06 bits/ins, ranging from 0.15 bits/ins for *adpcm_c* to 4.91 bits/ins for *bf_e*. These results indicate that using partial stream descriptors in the BASE scheme is indeed highly beneficial. Still, assuming a processor 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 in the worst case (for benchmark *bf_e*). However, having wide trace ports significantly increases system cost, especially in the presence of multiple processor cores because I/O bandwidth does not scale at the same pace as the on-chip resources. This result further underscores the need to have a trace module that supports effective compression.

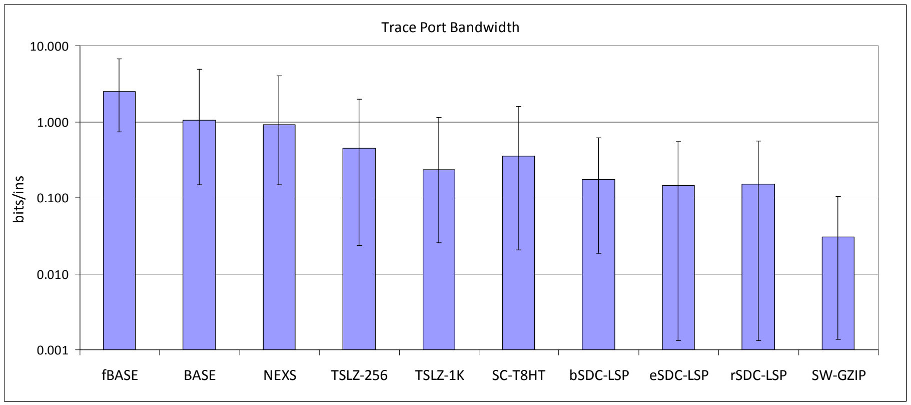


Fig. 6. Trace port bandwidth evaluation for all proposed and some related schemes.

The NEXS approach implements a simple trace reduction technique inspired by the NEXUS standard [11]. The starting address from the incoming stream descriptor is XORed with the starting address from the previous stream descriptor, producing DiffSA = Incoming.SA[31:0] xor Previous.SA[31:0]. The difference is split into groups of 6 bits, DiffSA[5:0], DiffSA[11:6], DiffSA[17:12], etc. The leading zeros in the DiffSA are not sent to the trace port, thus reducing the trace port bandwidth. For example, if the DiffSA[31:6] consists only of zeros, then only the DiffSA[5:0] is sent to the trace port, together with a 2-bit header indicating that this is the terminating byte for the stream address field. The SL field is always sent to the trace port without further reduction. The average trace port bandwidth required for the NEXS scheme is 0.907 bits/ins, ranging from 0.149 bits/ins for *adpmc_c* to 4.01 bits/ins for *bf_e*. This relatively small improvement compared to the BASE scheme is due to the fact that the number of indirect branches is small, so we have a small number of trace records that include a full stream descriptor. Another reason is the relatively high overhead in header bits.

Next, we analyze a recent adaptation of the Lempel-Ziv compression algorithm by Kao et al. that is specifically tailored to program execution traces [13]. The 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 average trace port bandwidth is 0.446 bits/ins for a compressor with a sliding window of 256 12-bit entries (TSLZ-256) with a maximum pattern length of 128. The compressor's complexity is estimated to be 51,678 logic gates [13]. For this configuration the worst performing benchmark (*stringsearch*) requires more than 1.9 bits/ins of trace port bandwidth. The compressor can recognize long repeating patterns, but it relies on relatively long fixed 27-bit trace records that consist of a 12-bit word, a 7-bit pattern length, and an 8-bit index in the sliding window. With even larger sliding windows of 1024 or 8192 12-bit entries, it requires 0.233 bits/ins and 0.1 bits/ins on the trace port, respectively. However, the implementation cost of such large sliding windows would be prohibitive.

We also evaluate one of our earlier compression methods [14]. This technique relies on the stream descriptor cache in the first stage and a tuple history table (THT) in the second stage. The tuple history table is a fully-associative structure that keeps the *m* most recent *n*-tuples of stream indices. An incoming *n*-tuple is searched in the THT; in case of a hit, the incoming *n*-tuple is replaced by a single index in the THT. In case of a THT miss, a reserved index 0 followed by the whole *n*-tuple is traced out; in case of an SDC miss, the full stream descriptor is traced

TABLE 8
TRACE PORT BANDWIDTH EVALUATION: A COMPARATIVE ANALYSIS.

	fBASE	BASE	NEXS	TSLZ-256	TSLZ-1K	SC-T8HT	bSDC-LSP	eSDC-LSP	rSDC-LSP	SW-GZIP
adpcm_c	0.731	0.150	0.149	0.024	0.025	0.021	0.019	0.001	0.001	0.001
bf_e	6.798	4.913	4.010	0.354	0.367	0.325	0.357	0.345	0.345	0.038
cjpeg	3.261	0.790	0.752	0.431	0.138	0.219	0.131	0.088	0.088	0.050
djpeg	1.605	0.390	0.366	0.230	0.176	0.173	0.075	0.053	0.054	0.019
fft	3.810	1.895	1.554	1.921	1.036	1.590	0.616	0.542	0.554	0.065
ghostscript	4.018	1.814	1.578	1.394	0.187	0.629	0.232	0.216	0.216	0.038
gsm_d	2.052	0.621	0.567	0.152	0.151	0.183	0.086	0.051	0.052	0.009
lame	1.234	0.452	0.391	0.171	0.148	0.203	0.102	0.090	0.109	0.040
mad	1.931	0.785	0.668	0.268	0.144	0.180	0.129	0.116	0.117	0.042
rijndael_e	1.911	1.013	0.840	0.043	0.038	0.071	0.192	0.183	0.185	0.013
rsynth	2.278	0.883	0.747	0.271	0.247	0.349	0.175	0.147	0.149	0.018
sha	2.597	0.602	0.567	0.441	0.036	0.073	0.101	0.074	0.074	0.005
stringsearch	6.644	2.157	1.932	1.962	1.135	1.402	0.472	0.412	0.416	0.104
tiff2bw	3.124	0.668	0.654	0.146	0.137	0.137	0.104	0.030	0.031	0.006
tiff2rgba	1.447	0.349	0.330	0.160	0.095	0.079	0.045	0.012	0.012	0.005
tiffdither	2.804	0.692	0.659	0.573	0.073	0.489	0.190	0.158	0.159	0.080
tiffmedian	1.795	0.380	0.374	0.081	0.301	0.078	0.066	0.027	0.028	0.007
Average	**2.510**	**1.055**	**0.907**	**0.446**	**0.233**	**0.353**	**0.174**	**0.146**	**0.150**	**0.031**

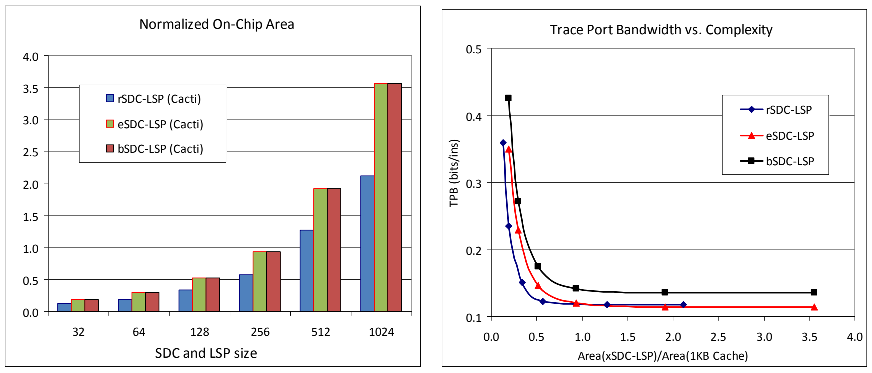


Fig. 7. Normalized on-chip area (left) and trace reduction ratio vs. complexity (right).

out. We evaluate a configuration with a 128x4 SDC and a 255-entry 8-tuple history table (*m*=255, *n*=8) on our benchmark suite (which differs slightly from the one we used previously [14]). This approach requires 0.353 bits/ins, which is over two times more than for any of the schemes proposed in this article.

Our most cost-effective scheme, rSDC-LSP, yields an average trace port bandwidth of 0.15 bits/ins, ranging from 0.001 bits/ins for *adpcm_c* to 0.554 bits/ins for *fft*. Consequently, the worst performing benchmark requires less than one bit per instruction, allowing us to trace the program execution through just a single bit on the trace port, i.e., a JTAG port would be sufficient. Finally, to underline the effectiveness of the proposed mechanism, we compare it with the software compression utility gzip that implements the Lempel-Ziv compression algorithm. This algorithm uses large memory buffers and its implementation in a hardware trace module would be cost-prohibitive. If we supply a sequence of filtered stream descriptors as an input to gzip with small buffers (gzip -1), it achieves 0.031 bits/ins, which is about 5 times better than the proposed schemes. While the result indicates that there is still room for improving the trace compression algorithm, we believe that the proposed mechanism achieves an excellent compression at minimal hardware cost.

4.2 Complexity Estimation

To estimate the size of the proposed trace module, we need to estimate the size of all structures, including the SDC, the LSP, the stream detector, the stream descriptor buffer, the LVSA register, the AOLC register with the training register, and the output trace buffer. Let us first discuss the complexity of the hardware structures using the number of storage bits as complexity metric. We focus on our most cost-effective scheme rSDC-LSP. The stream detector requires two registers, a 30-bit SA and an 8-bit SL. The SDC is a simple cache-like structure; for example our best performing configuration has 127 entries (32 sets x 4 ways). Entry 0 is non-existing since it is reserved to indicate a miss in the SDC. An entry in the SDC requires 13 bits for the SA (see Section 3.3), 8 bits for the SL, a valid bit, and one replacement bit (for the MRU-based replacement policy), so the total number of storage bits in the SDC is 2921. The LSP is a direct-mapped structure with 128 7-bit entries. The LSP is indexed by the previous stream index (also 7 bits), so the total number of storage bits is 903. The AOLC is an 8-bit counter and we also need a 4-bit training counter. Finally, we need to determine the minimum sizes of the stream descriptor buffer (Fig. 3) and the trace output buffer (Fig. 1). The size of these structures should be such that the processor is never stalled due to the finite capacity of the trace structures. To determine the size of these structures, we use a cycle-accurate processor model similar to Intel's XScale processor [21]. The trace module is modeled as follows. We assume that it requires one clock cycle to service an LSP with SDC hit or an LSP miss with SDC hit event and two clock cycles for an SDC miss event. The SDC and LSP work in parallel because the LSP is indexed by the previous stream index, so their access latencies are not additive. The trace records are stored in the trace output buffer. If the output buffer is not empty, a single bit is sent out through the trace data port each clock cycle. The processor is never stalled and no trace records are lost if the following conditions are met for our benchmarks: the number of entries in the stream descriptor buffer is at least two (2*38 = 76 bits), and the minimum trace output buffer size is 80 bits. Thus, the complexity of our rSDC-LSP scheme is estimated to be 4,042 storage bits.

The implementation complexity of the proposed schemes is predominantly determined by the size of the SDC and LSP structures. To quantitatively estimate their size, we use the Cacti tools (versions 4.0 and 6.0) [22] that report the area occupied by the cache tags and the data memory portions of the cache structures. To enable a comparative analysis with the complexity of known structures, we compare the total area of the SDC and LSP together with the total area occupied by a 1 kilobyte cache (including both the tags and the data portion) with the following parameters: a single read/write port, an 8-byte block size, 4 ways, and 32-bit addresses. Fig. 7 shows the normalized on-chip area for different compressor configurations assuming 90 nm technology. We find that our rSDC-LSP scheme (32x4 SDC, 128-entry LSP) requires an on-chip area of 15,640 μm^2 for 90 nm technology and 63,250 μm^2 for 180 nm technology. For 90 nm technology, the combined area for the SDC and the LSP is only 33.9% of the area required by the 1 kilobyte cache. The base scheme (bSDC-LSP) with the same configuration requires 51.7% of the area. This confirms our expectations that enhancements for reducing complexity are indeed beneficial. We also used the CACTI tools to estimate the access times of the SDC and LSP; the 128-entry (32x4) SDC requires 0.77 ns and the 128-entry LSP requires 0.86 ns assuming 90 nm technology. The estimated complexity of the compression method SC-T8HT [14] is 1.9 times the area of the

1 kilobyte cache. Kao et al. reported the on-chip area of their TSLZ-256 compressor to be 511,616 μm^2 using 180 nm technology, which is over 8 times larger than the rSDC-LSP using 180 nm technology.

Fig. 7, right, shows the trace port bandwidth as a function of the trace module complexity. The area represented on the x axis is normalized to the area of a 1 kilobyte cache. Different points represent different trace module configurations, varied from 32 entries [8x4, 32] to 1024 entries [256x4, 1024]. For example, rSDC-LSP [32x4, 128] requires only 0.15 bits/ins at the cost of 0.33*Area(1 KB Cache); rSDC-LSP [64x4, 256] requires only 0.123 bits/ins at the cost of 0.56*Area(1 KB cache). At the low end of complexity, which is what we are interested in, rSDC-LSP emerges as the best solution and is therefore our recommended implementation.

5 Conclusions

This paper describes a new low-cost mechanism for real-time tracing and compression of program executions. The mechanism exploits temporal and spatial locality of program streams using two new structures called stream descriptor cache and last stream predictor to achieve compression ratios that are over six times higher than commercial state-of-the-art solutions.

We have explored the design space of the proposed hardware structures including their access functions, size, and organization. We have introduced several low-cost enhancements to the initial scheme and demonstrated their effectiveness by analyzing the required trace port bandwidth and hardware complexity. Our best performing approach requires an average trace port bandwidth of only 0.15 bits/instruction on the MiBench programs. This enables very cost-effective tracing through a single-pin trace port at a cost in on-chip area that is equivalent to one third of a 1KB cache for the trace compression structures.

Glossary

Abbreviation	Definition
AOLC	Adaptive One-Length Counter
avgSL	Average Stream Length
hrLSP	Hit rate in Last Stream Predictor
hrSDC	Hit rate in Stream Descriptor Cache
IC	Instruction Count
LSP	Last Stream Predictor
LVSA	Last Value Starting Address
NEXS	Nexus-like Compressor
SA	Starting Address
SDB	Stream Descriptor Buffer
SDC	Stream Descriptor Cache
SI	Stream Index
SL	Stream Length
THT	Tuple History Table
TPB	Trace Port Bandwidth
TSLZ	Trace-Specific LZ Compressor

Acknowledgment

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.

References

[1] B. Dipert, "Inside Apple's iPhone: More Than Just a Dial Tone," <http://www.edn.com/article/CA6457065.html?nid=2551> (Available September 2009).

[2] J. Messina, "Multi-Core ARM Chips Slated for Smartphones Next Year," <http://www.physorg.com/news164386074.html> (Available September 2009).

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

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

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

[6] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores," *IEEE Transactions on Computers*, vol. 55, February 2006, pp. 174-184.

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

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

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

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

[11] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface," <http://www.nexus5001.org> (Available November 2009).

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

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

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

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

[16] A. R. Pleszkun, "Techniques for Compressing Program Address

Traces," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, 1994, pp. 32-39.

[17] A. Milenković and M. Milenković, "Exploiting Streams in Instruction and Data Address Trace Compression," in *Proceedings of IEEE 6th Annual Workshop on Workload Characterization*, Austin, TX, 2003, pp. 99-107.

[18] A. Milenković and M. Milenković, "An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams," *ACM Transactions on Modeling and Computer Simulation*, vol. 17, 2007, pp. 1-27.

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

[20] V. Uzelac, A. Milenković, M. Milenković, and M. Burtscher, "Real-Time, Unobtrusive, and Efficient Program Execution Tracing with Stream Caches and Last Stream Predictors," in *XXVII IEEE International Conference on Computer Design*, Resort at Squaw Creek, Lake Tahoe, CA, 2009, pp. 173-178.

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

[22] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A Tool to Model Large Caches," HP Laboratories, 2009.

Aleksandar Milenković received the Dipl. Ing., M.S., and Ph.D. degrees in computer engineering and science from the University of Belgrade, Serbia, in 1994, 1997, and 1999. He is currently 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/~lacasa). His research interests include computer architecture, embedded systems, VLSI, and wireless sensor networks. Prior to joining the University of Alabama he held academic positions at the University of Belgrade in Serbia and the Dublin City University in Ireland. He is a member of the IEEE, its Computer Society, the ACM, and Eta Kappa Nu.

Vladimir Uzelac received his B.S. degree in electrical engineering from the University of Belgrade in 2002 and his M.S. and Ph.D degree in computer engineering from the University of Alabama in Huntsville in 2008 and 2010. In the meantime he worked as a hardware design engineer for several years. He has recently joined Tensilica, Santa Clara, where he works as an R&D engineer for embedded software and debugging architecture and tools.

Milena Milenković Milena Milenkovic received her B.S. and M.S. degrees from the University of Belgrade and her Ph.D. degree from the University of Alabama in Huntsville. Her research interests include performance evaluation, secure computer architectures, data compression, and architecture-aware compilers. Milena 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 Ph.D. degree in computer science from the University of Colorado at Boulder in 2000. Since then, he has been an assistant professor in the School of Electrical and Computer Engineering at Cornell University and a Research Scientist in the Institute for Computational Engineering and Sciences at the University of Texas at Austin. His current research focuses on automatic parallelization of irregular programs for multicore and GPU architectures as well as on automatic performance assessment and optimization of HPC applications. He is an associate editor of the Journal of Instruction-Level Parallelism and a senior member of the IEEE, its Computer Society, and the ACM.