Hardware-Based Data Value and Address Trace Filtering Techniques

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

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

ABSTRACT
Capturing program and data traces during program execution unobtrusively in real-time is crucial in debugging and testing of cyber-physical systems. However, tracing a complete program unobtrusively is often cost-prohibitive, requiring large on-chip trace buffers and wide trace ports. Whereas program execution traces can be efficiently compressed in hardware, compression of data address and data value traces is much more challenging due to limited redundancy. In this paper we describe two hardware-based filtering techniques for data traces: *cache first-access tracking for load data values* and *data address filtering using partial register-file replay*. The results of our experimental analysis indicate that the proposed filtering techniques can significantly reduce the size of the data traces (~5-20 times for the load data value trace, depending on the data cache size; and ~5 times for the data address trace) at the cost of rather small hardware structures in the trace module.

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

General Terms
Algorithms, Design, Verification.

Keywords
Debugging, Program Tracing, Compression.

1. INTRODUCTION
Ever increasing complexity of both hardware and software and tightening time-to-market impose a number of challenges to embedded system verification and debugging. According to one estimate, software developers spend 50%-75% of their development time in program debugging [1], yet it is estimated that the U.S. loses approximately $20-$60 billion a year due to software bugs and glitches. Software developers face a limited visibility of on-chip modules caused by limited I/O bandwidth, high internal complexity, and high operating frequencies. To meet these challenges and get reliable and high-performance products to the market on time, software developers increasingly rely upon on-chip resources for debugging and program tracing. However, even limited hardware support for debugging and tracing is associated with extra cost in chip area for capturing and buffering traces, for integrating these modules into the rest of the system, and for sending out the information through dedicated trace ports [2]. These costs often make system-on-a-chip (SOC) designers reluctant to invest in additional chip area solely devoted to debugging and tracing.

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

Many embedded processor vendors have developed modules with advanced tracing and debugging capabilities and integrated them into their embedded platforms, e.g., ARM's Embedded Trace Macrocell [4], MIPS's PDTrace [5], and OCDS from Infineon [6]. The trace and debug infrastructure on a chip typically includes logic that captures address, data, and control signals, logic to filter and compress the trace information, buffers to store the traces, and logic that emits the content of the trace buffer through a trace port to an external trace unit or host machine. In this paper we focus on data traces (Class 3 operation in Nexus). While program execution traces are very useful to pinpoint a bug location, often a full data trace is required to faithfully replay the program offline. Tracing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'10, October 24–29, 2010, Scottsdale, Arizona, USA.
Copyright 2010 ACM 978-1-60558-903-9/10/10...$10.00.

data writes only is useful to identify unexpected and erroneous writes to the memory from a specific core. Tracing load values only is, under certain conditions, sufficient to deterministically reconstruct the whole program offline. Data address traces captured in real-time are of special interest in multi-core systems as they offer valuable information about shared memory access patterns and possible data race conditions.

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

To illustrate challenges associated with data tracing, we profile seventeen representative benchmarks from the MiBench suite [7] assuming the ARM instruction set. Table 1 shows the instruction count (IC), the frequency of loads (LD.f) and stores (ST.f), as well as the trace port bandwidth when tracing load and store data values (LD.DVT and ST.DVT), and load and store data addresses (LD.DAT and ST.DAT). The bandwidth is expressed in the average number of bits required per each instruction executed (bits/ins). The average trace port bandwidth for tracing load data values is 8.2 bits/ins, ranging from 3.2 to 14.8 bits/ins. Note: the average is the weighted arithmetic mean; a benchmark weight is proportional to its number of load instructions divided by the total number of loads in all benchmarks. For example, an 8 KB trace buffer could capture load values for less than 1,000 instructions (8 KB/8.2 bits), which is often insufficient to locate software bugs. This estimation is based on the average bandwidth and does not accommodate for possible bursts in trace events (e.g., a benchmark may have a number of consecutive load instructions that needs to be traced). Tracing data addresses also requires significant trace port bandwidth: the average bandwidth is 7.84 bits/ins for load addresses and 3.37 bits/ins for store addresses. For some benchmarks, data address traces may require lower bandwidth than data value traces, which is counter-intuitive. However, this is due to ARM's multiple load/store instructions, where we trace a single address and multiple data values, and due to double-word loads and stores.

Filtering and compressing data traces at runtime in hardware can reduce requirements for on-chip trace buffers and trace port communication bandwidth. However, existing commercial solutions offer very rudimentary trace compression. For example, Nexus [3] and ARM's ETM implement differential compression of instruction and data addresses, where only a difference between consecutive addresses is recorded. Data values are traced uncompressed.

Trace reduction has been widely studied in academia. A number of trace-specific compression techniques have been proposed [8-10]. However, they are mainly focused on software-based compression, combining trace-specific compression with a general-purpose compression algorithm, such as *gzip* or *bzip2*. Such solutions would be impractical and cost-prohibitive for real-time tracing. Several proposals address reduction of trace messages captured on SOCs buses, but they provide fairly limited compression ratios [11]. Whereas several academic proposals have addressed real-time hardware-based compression of program execution traces [12-14], the more challenging problem of real-time hardware-based reduction of data address and value traces has not been directly addressed so far.

Table 1. Data value and address trace characteristics for MiBench programs.

	IC [mil.]	LD.f [%]	ST.f [%]	LD.DVT bits/ins	ST.DVT bits/ins	LD.DAT bits/ins	ST.DAT bits/ins
adpcm_c	732.5	12.7	0.9	3.21	0.08	4.07	0.29
bf_e	544.1	20.6	11.4	8.24	7.12	6.58	3.65
cjpeg	104.6	26.7	7.8	7.40	2.52	8.54	2.50
djpeg	23.4	33.1	12.4	7.48	2.51	10.60	3.97
fft	631.0	18.2	8.5	7.63	4.96	5.83	2.72
ghostscript	708.1	23.3	13.1	7.64	5.80	7.45	4.20
gsm_d	1299.3	15.9	10.1	4.63	3.23	5.08	3.22
lame	1285.1	30.0	16.5	14.77	6.77	9.60	5.29
mad	287.1	27.5	7.5	8.26	2.49	8.79	2.41
rijndael_e	320.0	39.1	8.2	11.18	3.51	12.51	2.62
rsynth	824.9	40.2	13.9	13.12	5.19	12.85	4.43
sha	140.9	16.1	8.4	4.77	1.79	5.17	2.69
stringsearch	3.7	11.9	16.1	4.20	6.14	3.82	5.15
tiff2bw	143.3	26.8	13.4	3.85	2.76	8.58	4.28
tiff2rgba	151.7	39.5	26.8	8.16	8.63	12.65	8.57
tiffdither	833.0	20.6	7.1	5.13	1.82	6.58	2.27
tiffmedian	541.3	31.0	7.1	6.96	1.87	9.93	2.26
Average		**24.6**	**10.5**	**8.20**	**4.18**	**7.84**	**3.37**

In this paper, we describe two cost-effective filtering mechanisms that can significantly reduce the size of the data traces used in debugging. In Section 2, we introduce the *cache first-access tracking mechanism for filtering load data values*. This mechanism utilizes an approach similar to the first-load mechanism presented by Narayanasamy et al. [15], but it is modified to make it suitable for real-time tracing in embedded systems. It relies on on-chip data caches augmented by *first-load tracking bits* that determine whether a load value needs to be traced out of the chip, or it can be inferred by the software debugger (Section 2.1). Our experimental analysis (Section 2.2) shows that the proposed filtering mechanism reduces the size of the load data value trace 12.4 times in a system with a 16 KB data cache and 20.6 times in a system with 64 KB data cache.

In Section 3, we introduce a *data address filtering mechanism based on a partial program replay*. The trace module maintains a structure that tracks whether general-purpose registers can be inferred by the software debugger based on the traced data addresses and the program binary (Section 3.1); only data addresses that cannot be inferred are traced out. The proposed filtering is combined with a simple differential encoding to further reduce the size of the data address trace. Our experimental analysis (Section 3.3) shows that the proposed mechanism offers 5-fold data address trace reduction over the uncompressed address trace.

The main contributions of this work are as follows.

- We introduce a hardware-based mechanism for filtering load data values called the *cache first-access tracking mechanism*. When coupled with the corresponding changes of the software debugger, it enables cost-effective and unobtrusive data tracing in real-time.
- We introduce a novel hardware-based mechanism for *data address filtering that utilizes a partial program replay*. When coupled with the corresponding changes of the software

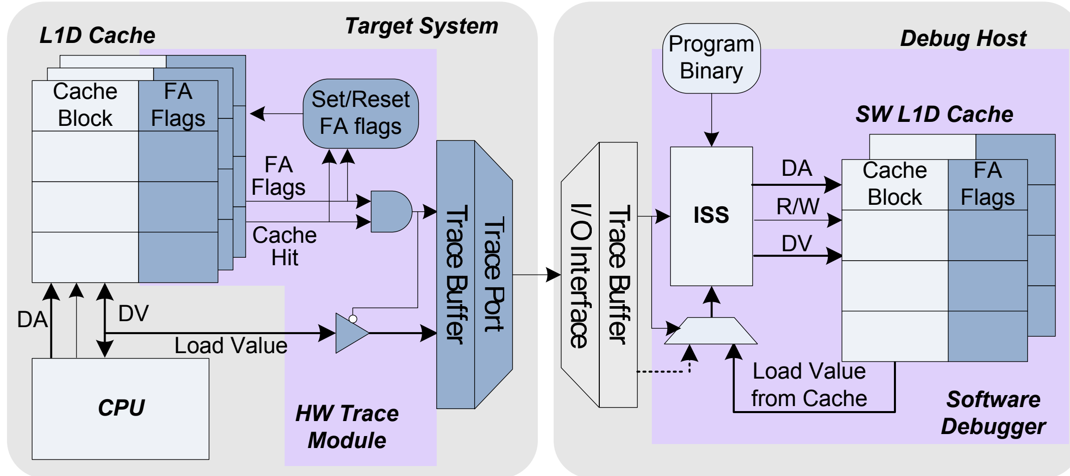

Figure 1. Cache first-access tracking mechanism: system view.

debugger, it significantly reduces the trace port bandwidth requirements.

- We perform a detailed experimental analysis that shows the proposed hardware-based filtering techniques outperform a software compressor when compressing load data value and address traces. The cache first-access tracking requires over ~5-20 times less bandwidth on the trace port than when tracing the uncompressed load data value trace (depending on the data cache size). The proposed data address filtering requires over 5 times less bandwidth on the trace port than when tracing raw data addresses. The proposed techniques require relatively little hardware resources dedicated to tracing, but instead require close integration of these resources with the processor core and advanced software debuggers.

2. LOAD VALUES FILTERING THROUGH CACHE FIRST-ACCESS TRACKING

A software debugger can replay program execution deterministically offline if the following four conditions are met: (a) it includes an instruction set simulator (ISS) for the target processor; (b) it has access to the program binary, (c) it has access to the load value trace captured on the target processor, and (d) it knows the initial state of general- and special-purpose registers. Consequently, capturing the load data value trace on the target processor and reading the trace out of chip are critical in program debugging. However, as shown in the previous section, tracing load data values in real-time may be cost-prohibitive or impossible. In addition, compressing load values using general-purpose compression algorithms will yield little benefits due to limited redundancy in data traces: e.g., the software *gzip* utility achieves the average compression ratio of only 3.5 for our benchmarks. It should be noted that implementing general-purpose compression algorithms in hardware would be cost prohibitive and infeasible for real-time compression.

Data caches are routinely used in mid- to high-end embedded processors to reduce latency of memory-referencing instructions by exploiting temporal and spatial locality. A data cache can be augmented to help reduce load value trace size. We do not need to trace a data value for each load instruction if the software debugger includes an exact model of the data cache[1] used in the target processor (with the same organization and update policies). Rather, the debugger can retrieve the load value from its software copy of the data cache. Thus, tracing load data values is required only for certain events in the data cache. For example, if a load causes a miss in the data cache, we need to trace its data value. In addition, if a load hits in the data cache, we still may need to trace it out to the debugger, if this is the first load access to that particular address. Consequently, we need to expand our data cache on the target processor so that for each data object we can keep track whether it has been already read (and thus can be inferred by the debugger) or not (it has to be traced out to the debugger).

We expect this filtering mechanism to significantly reduce the number of load values that needs to be traced out, thus reducing the required trace port bandwidth. We call this mechanism cache first-access tracking. It is based on a mechanism used in the BugNet [15] with some modifications to make it suitable for real-time tracing in embedded systems. The BugNet is designed to log relevant information about program execution on production runs (released software) and to communicate these logs back to the developer after system crashes. Its first-access track mechanism is used as an architectural extension to help reduce the amount of information that needs to be recorded in the log. The BugNet relies on a check-pointing mechanism and its first-load log requires hundreds of kilobytes of storage. The log is kept in main memory, and thus the logging itself is an obtrusive process. However, our goal is to examine whether a similar mechanism can ensure real-time continual and unobtrusive tracing of load data values in embedded systems.

[1] Without lack of generality we assume that our system include only first level data cache. The mechanism can be easily extended to systems with a multi-level cache hierarchy.

2.1 Cache First-Access Tracking Mechanism

Figure 1 shows the system view of the cache first-access mechanism. The target platform executes a program on a processor core. The processor has a data cache that is extended so that each cache block includes corresponding first-access flags. Generally, we need a flag for the smallest addressable unit, which is typically a byte. Consequently, a 32-byte cache block requires 32 single-bit first-access flags that are attached to the cache block. A trace module, coupled with the processor and its data cache, monitors cache events caused by load and store instructions (misses and hits) and the state of corresponding first-access tracking flags.

Figure 2 describes the trace module operation. For each load instruction, it checks whether it hits or misses in the data cache. If we have a load cache hit and the corresponding first-access flags are set[2] (an FA hit event), then we do not need to trace the load value because it can be inferred by the software debugger. In that case we send only a single header bit to a trace buffer that indicates that a data value can be inferred (line 4). Otherwise, if the corresponding first-access flags are cleared (or at least one of them is cleared), the requested load data value is traced out together with a header bit indicating a trace module miss event (lines 6-8). If we have a cache miss caused by a load instruction, the cache block is fetched from memory, and thus all first-access flags associated with that block need to be cleared. The load value is traced out and the FA flags are set accordingly. Please note that we could further reduce required trace port bandwidth by not sending the header bit on each FA hit event. Instead, we could use a counter that counts consecutive hit events and report the counter value only when we have a trace miss event. However, here we opted for a simpler approach that guarantees easier synchronization between the trace module and the software debugger at the cost of the slightly increased trace port bandwidth.

The cache first-access flags are also updated on store instructions and on signals triggered outside of the processor core, e.g., cache block invalidations caused by the cache controller (Figure 2). Each store will set the corresponding first-access flags because its value in the cache becomes known (and can be inferred by the debugger). Note: here we assume a data cache with write-allocate, write-back policies. External signals can invalidate a cache block at any point of time. In that case, the trace module needs to clear all first-access flags that belong to that line. This action does not need to be synchronized with the software debugger – the debugger always checks the trace input first.

The software debugger running on the host machine reads and decodes the trace records and replays the program. The debugger relies on its ISS with the software model of the data cache, program binary, and the load data value trace received from the target for program replay. Its operation is described in Figure 3. For each load instruction the debugger checks the next trace record from the target. If it reads a header bit ‘1’ (indicating an FA hit), the debugger reads the data value from its software copy of the data cache. Otherwise, the load data value is read from the trace record and the cache is updated accordingly, including the FA bits as well. For a store instruction, the debugger sets the FA flags accordingly.

[2] In case of a load that reads a 32-bit word we need to check all four first-access flags (for each byte in the word).

```
1.  // For each load that reads N bytes
2.  if (CacheHit) {
3.     if (corresponding N FA flags are set)
4.       Emit('1') into Trace Buffer;
5.     else {
6.       Emit('0') into Trace Buffer;
7.       Emit Load Value into Trace Buffer;
8.       Set corresponding N FA flags;
9.     }
10. else { // cache miss event
11.    Clear FA bits for newly fetched cache block;
12.    Perform steps 6-8;
13. }

1.  // For each store that writes N bytes
2.  Set corresponding N FA bits;

1.  // For external invalidation request
2.  Clear FA bits for entire cache block;
```

Figure 2. Trace module operation.

```
1.  // For each load that reads N bytes
2.  Get the next trace record;
3.  if (header == '1')
4.     if (corresp. FA bits in SW cache are set)
5.       Retrieve data value from SW cache;
6.     else
7.       ERROR in Tracing; // illegal event
8.  else {
9.     Read N bytes from trace record;
10.    Update SW cache;
11.    Set corresponding N FA flags in SW cache;
12. }

13. // For each store that writes N bytes
14. Update SW cache;
15. Set corresponding N SW cache FA bits;
```

Figure 3. Software debugger operation.

The proposed mechanism requires modest additional hardware resources. The major complexity overhead comes from the storage needed for first-access flags (1/8th of the data cache capacity). However, in a system with 64 KB data cache, this overhead reaches 8 KB of storage devoted to first-access flags. To reduce this overhead, we may consider different granularity for first-access flags. For example, a single first-access bit can guard an entire 32-bit word, thus reducing the storage overhead to 1/32th of the data cache capacity. In benchmarks with a significant number of byte- or half-word loads this change will result in an increased trace port bandwidth (only a word access can set the corresponding first-access flag). In this paper we assume that each byte in the cache has its own first-access flag, but further exploration is needed to determine optimal granularity of first-access flags.

It should be noted that in this work we assume that first-access flags are attached to cache blocks. However, if the design of the data cache can not be changed, an alternative design can be used where the first-access flags are physically placed inside the trace module instead being attached to the data cache. A well-defined interface between the data cache and the trace module would ensure exchange of control signals. The former approach is less complex because we do not need a separate address decoding logic for the first-access flags, but requires changes in the data cache design; the later may better fit current design practices where the trace module includes all debug infrastructure.

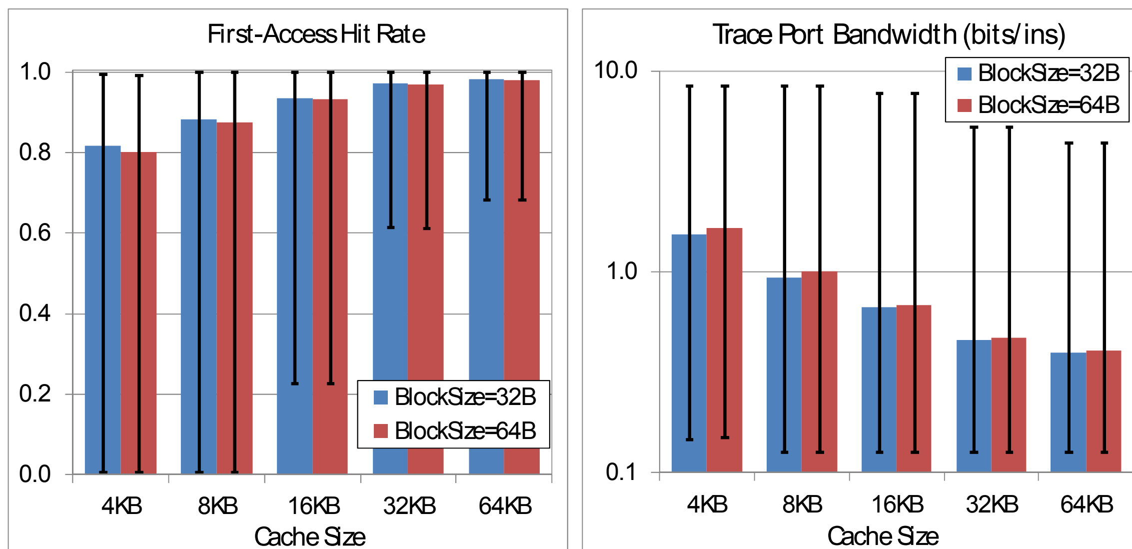

Figure 4. Cache first-access hit rate (a) and trace port bandwidth (b).

2.2 Experimental Evaluation

In this section we analyze the effectiveness of the cache first-access mechanism in *filtering load data values*. As a measure of effectiveness we use trace port bandwidth calculated in bits per executed instruction. The bandwidth depends on several parameters: data cache hit rate, first-access hit rate, load data size (byte, word, double word, and so on), and frequency of load instructions. We also measure the first-access hit rate. As workload we use seventeen benchmarks from the MiBench suite [7]. Our analysis is performed using a functional SimpleScalar ARM simulator [16]. The ARM instruction set includes multiple-load and multiple-store instructions. In calculating trace port bandwidth in bits/ins, a multiple load/store instruction is counted as multiple independent instructions.

Figure 4 shows the average first-access hit rate (left) and the trace port bandwidth (right) when tracing filtered load data values as a function of data cache size (varying from 4 KB to 64 KB) and cache block size (32 B and 64 B). Table 2 shows the first-access hit rate and the trace port bandwidth when tracing filtered data values for individual benchmarks as a function of data cache size for fixed 32 B cache blocks. In all cases we assume a 4-way set-associative data cache with write-allocate and write-back policies and a pseudo-LRU replacement policy. The average first-access hit rate is rather high even with very small caches. With a 4 KB data cache, on average 82% of loads require no tracing. With larger caches this number is over 90%, being 94% and 98% on average for 16 KB and 64 KB caches, respectively. However, we can see that some benchmarks, most notably *tiff2bw* and *tiff2rgba* (Table 2), have rather low first-access hit rates (~0% for very small caches), in spite of relatively high data cache hit rates. These two benchmarks process a significant number of byte-size operands that are accessed sequentially. Thus, traversing a 32 B cache block may result in a single data cache miss followed by 31 cache hits (the data cache hit rate is thus rather high, ~96% for this example). However, the first-access flags are all cleared on the miss, resulting in 32 first-access misses (the first-access hit rate is thus 0% for this example). With larger caches the hit rate goes up, and thus the number of first-access hits increases (the cache blocks with first-access flags set are not evicted from the cache).

The trace port bandwidth varies from 0.15 bits/ins (*adpmc_c*) to 8.47 bits/ins (*tiff2rgba*) for a 4 KB data cache, and from 0.13 to 4.39 bits/ins for a 64 KB cache. For all benchmarks except three (*tiff2rgba*, *tiff2bw*, and *tiffmedian*), the required trace port bandwidth is less than 1 bits/ins for a system with 16 KB data cache. For a system with 32 KB data cache, all benchmarks except one (*tiff2rgba*) require less than 1 bits/ins on the trace port, promising real-time, continual, and unobtrusive tracing of load data values using a very narrow trace port. For example, for a processor that executes one instruction per cycle (*IPC=1*), a single-pin trace port working at the processor speed would enable

Table 2. First-load hit rate and trace port bandwidth.

	First Load Hit Rate					Trace Port Bandwidth (bits/ins)					
	4KB	**8KB**	**16KB**	**32KB**	**64KB**	**4KB**	**8KB**	**16KB**	**32KB**	**64KB**	**gzip**
adpcm_c	0.98	1.00	1.00	1.00	1.00	0.15	0.13	0.13	0.13	0.13	0.78
bf_e	0.93	0.98	1.00	1.00	1.00	0.74	0.32	0.27	0.27	0.27	1.99
cjpeg	0.73	0.80	0.83	0.85	0.86	1.38	0.85	0.69	0.62	0.59	1.18
djpeg	0.77	0.86	0.93	0.99	1.00	1.37	0.86	0.57	0.39	0.34	1.34
fft	0.99	0.99	0.99	0.99	0.99	0.29	0.27	0.27	0.27	0.27	1.61
ghostscript	0.98	0.99	0.99	0.99	0.99	0.33	0.32	0.30	0.30	0.29	0.62
gsm_d	1.00	1.00	1.00	1.00	1.00	0.17	0.17	0.16	0.16	0.16	1.34
lame	0.73	0.88	0.95	0.97	0.98	4.34	2.02	1.13	0.79	0.66	5.66
mad	0.80	0.95	0.98	1.00	1.00	2.02	0.70	0.43	0.30	0.29	3.58
rijndael_e	0.64	0.94	1.00	1.00	1.00	4.98	1.16	0.45	0.40	0.40	4.73
rsynth	0.97	0.98	0.98	0.98	0.98	0.67	0.52	0.47	0.46	0.46	3.90
sha	0.97	0.99	1.00	1.00	1.00	0.35	0.21	0.17	0.17	0.17	1.94
stringsearch	0.96	0.97	0.99	0.99	0.99	0.33	0.29	0.21	0.19	0.19	0.70
tiff2bw	0.01	0.05	0.63	1.00	1.00	4.05	3.84	2.33	0.29	0.27	1.17
tiff2rgba	0.01	0.01	0.23	0.61	0.68	8.47	8.47	7.77	5.27	4.39	2.48
tiffdither	0.83	0.87	0.94	1.00	1.00	0.78	0.68	0.45	0.22	0.21	1.29
tiffmedian	0.51	0.53	0.74	0.91	0.98	2.48	2.38	1.69	0.78	0.41	1.78
Average	*0.82*	*0.88*	*0.94*	*0.97*	*0.98*	*1.53*	*0.93*	*0.66*	*0.46*	*0.40*	*2.40*

unobtrusive program tracing for all except one benchmark. The proposed mechanism reduces the trace port bandwidth even for *tiff2rgba* (e.g, over 50% in a system with 32 KB cache). Unfortunately, tracing this benchmark would still require a wider trace port.

We also determine a compression ratio achieved by the proposed filtering: it is calculated as the size of the uncompressed load data value trace divided by the size of the filtered output trace (including the header bit). We conclude that the proposed filtering achieves excellent total compression ratio: it is 5.3 for a system with 4 KB cache, 12.4 with 16 KB, and 20.6 with 64 KB. To illustrate the effectiveness of the proposed filtering we compare it with the software *gzip* utility when using it to compress the raw load data value trace. The fast *gzip* (gzip -1) achieves the average trace port bandwidth of 2.40 bits/ins (Table 2, last column). The proposed filtering mechanism outperforms the fast *gzip* for over 3.6 times in a system with 16 KB data cache, over 5.2 times in a system with 32 KB data cache, and 6 times in a system with 64 KB data cache. Please note that hardware implementation of the software *gzip* would be cost prohibitive in both required additional on-chip area and the compression latency.

The trace buffer in the proposed trace module (Figure 1) serves only to temporarily store trace records before they are read out through the trace port. The exact buffer size depends on the processor model (IPC), the number of data pins on the trace port, trace port speed, and benchmark characteristics (e.g., the frequency and density of load instructions and their locality). A detailed cycle-accurate simulation of the processor and trace module would be needed to determine the worst-case scenario for the trace buffer size. However, an ad-hoc analysis based on our functional simulation model indicates that a 64-byte buffer would be more than sufficient to amortize all possible bursts of first-access misses, enabling unobtrusive tracing in real-time (assuming a processor executing on average one instruction per processor clock cycle and a trace port working at the processor clock speed). This buffer would be several orders of magnitude smaller than buffers used to capture uncompressed load data value trace for a limited program segment.

3. DATA ADDRESS FILTERING THROUGH PARTIAL PROGRAM REPLAY

In the previous section we have shown that the load data value trace is sufficient to deterministically replay programs if the software debugger has a fully functional instruction set simulator. However, software designers may choose to capture only addresses of memory referencing instructions. They are valuable in memory hierarchy optimizations, providing insights in data access patterns, data sharing, and synchronization. In addition, we may not have readily available sophisticated debuggers augmented with the instruction set simulators and data cache simulators as discussed in the previous section.

Unfortunately, tracing of uncompressed data addresses requires a rather high trace port bandwidth as shown in Table 1, LD.DAT and ST.DAT columns. Here we introduce a filtering method that identifies data addresses that can be inferred by the software debugger during a partial program replay offline. Data addresses that can be inferred are not traced out of the chip, thus significantly reducing trace port bandwidth requirements. In inferring data addresses, the software debugger relies only on information *contained in the program binary* and on the *previously traced data addresses*. Note: consequently, in this section we assume tracing data addresses only.

Figure 5 shows a system view of the proposed data address filtering. A trace module is connected to the target processor through an interface that includes the following information: Instruction Type, Source Operands, Destination Operands, and an Exception control signal. The exception control signal serves to disable the control logic for register validation. If the software debugger is unable to trace the system calls or exception service routines, the trace module should disable the register validation unit in the trace module by configuring corresponding control registers.

The trace module maintains a valid bit vector associated with general-purpose registers. A register is flagged as valid only if its content can be inferred from the binary and the previously encountered data addresses; otherwise it is flagged as invalid. A

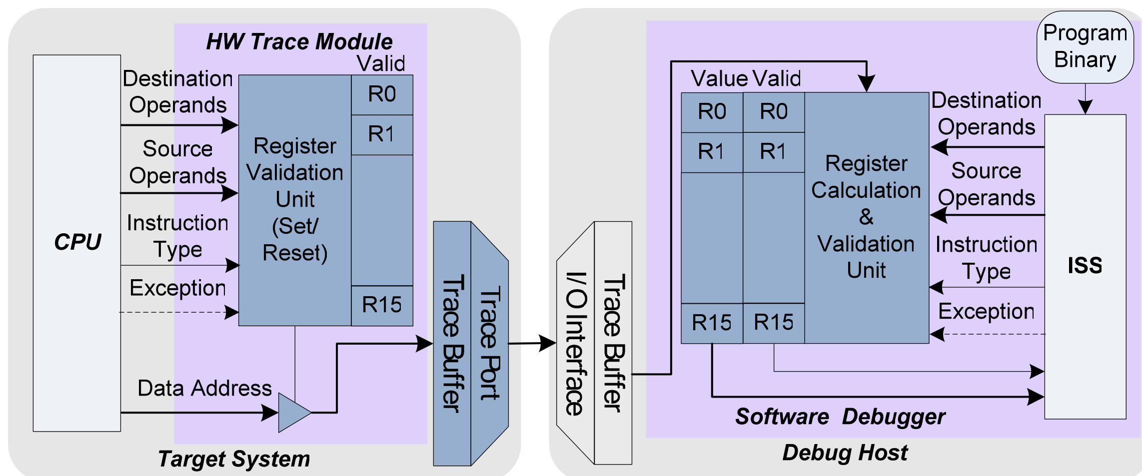

Figure 5. Data address filtering through partial replay: System view.

Register Validation Unit (RVU) is responsible for setting or resetting individual register valid flags based on its current state and the instruction execution information received from the processor. For each memory referencing instruction, the RVU also decides whether to enable or disable tracing of its data address. If tracing is required, a data address is stored in a trace output buffer and traced out of the chip.

Depending on the instruction type, the RVU performs the following operations.

- For a non memory-referencing instruction, it marks destination register(s) valid if all source registers are valid. The ARM instruction set architecture used in our study allows for up to 4 source registers, and up to 3 destination registers.
- For a memory referencing instruction, source registers are used to calculate the data address. If all source registers are valid, the data address is not traced, as it can be inferred by the software debugger; otherwise the data address needs to be traced out. If all but one source register is valid, the invalid register can be marked as valid, as its value can be calculated by the software debugger based on the data address to be traced and the content of other known source registers. For a load instruction, its destination register is always invalidated as its value is unknown after the load.

The software debugger on its side needs to partially replay the program binary[3]. It includes an instruction set simulator (ISS) with a software copy of the register file. The debugger encompasses a Register Calculation and Validation unit that is responsible to maintain and update the content and status (valid/invalid) of each general-purpose register. This unit performs reverse operations to those that are performed inside the processor for address calculation; it takes the traced data addresses as an input and tries to determine the content of individual registers using the information available from the program binary.

3.1 Register Validation

Register validation principles are applied to both the trace module and the software debugger. A destination register is set as valid in the following cases.

- An instruction specifies an immediate source operand. E.g., {`MOV R4, #2`} initializes the register R4; therefore, it is marked as valid and can be inferred by the software debugger.
- An instruction performs an operation on valid source registers. E.g., {`MOV R4, R5`} copies value in R5 to R4; if R5 is valid, R4 is marked as valid too, and R4 is loaded with the value of R5, *R4=R5*.
- A register is validated using a received data address, DA. E.g., {`LDR R5, [R4,#2]`} loads a value into register R5 from the memory address *DA=R4+2*. If the register R4 is not valid, the data address is traced. However, once the data address DA is sent out, R4 can be inferred by the software debugger as *R4=DA-2* and is thus set as valid.

[3] Partial replay capability is much simpler to implement than a fully functional instruction set simulator.

Table 3. Data address trace filtering example.

		Trace Module				Debug Host		
	Instruction	*Status*	*MR/TDA*	*DA*	*Status*	*Regs.*	*Status*	*DA*
1	MOV R1,#0	-	N/N	-	R1.V	R1=0	R1.V	-
2	LDR R5,R2,#5	R2.I	Y/Y	R2+5	R2.V, R5.I	R2=DA-5	R2.V, R5.I	DA
3	ADD R4,R2,R1	R1.V, R2.V	N/N	-	R4.V	R4=R1+R2	R4.V	-
4	LDR R8,R2,R1	R1.V, R2.V	Y/N	-	R8.I		R8.I	R2+R1
5	MOV R2,R7	R7.I	N/N	-	R2.I		R2.I	-

Table 3 illustrates the proposed data address filtering using a partial replay with register validation. We consider a sequence of events and the state of general-purpose registers in both the trace module and the software debugger while executing five instructions as shown in the second column of Table 3. We assume that all registers are initially invalid (both in the trace module and the software debugger). The first instruction is a non memory-referencing (*MR=N*), and thus no data address is traced (*TDA=N*). It loads an immediate constant #0 to register R1. Both the trace module and the software debugger set R1 as valid (R1.V) and the software debugger sets R1 to zero.

The second instruction is a load (*MR=Y*) that reads an operand from the memory location at the address DA, *DA=(R2+5)*. The register R2 is flagged as invalid and thus the data address needs to be traced out (*TDA=Y*). On the other side, the software debugger cannot calculate the data address and it expects it from the trace port; upon receiving the data address, DA, it calculates the value of register R2, *R2=DA-5*, and sets its valid bit (R2.V). The register R5 is loaded from memory, so its content is not known, and its valid bit is flagged as invalid (R5.I).

The third instruction is a non memory-referencing one that calculates R4, *R4=R2+R1*; the valid bit of register R4 (R4.V) is set because both the input operands R2 and R1 are valid. The fourth instruction loads the register R8 with an operand from memory at the address *DA=R2+R1*. The trace module does not need to trace out the DA, because the software debugger can calculate it based on R2 and R1 that are both valid. The register R8 is marked as invalid (R8.I). Finally, the last instruction moves the content of R7 to R2; the source register is invalid, and the register R2 is thus marked as invalid (R2.I).

3.2 Effects of Addressing Modes on Register Validation

Register validation is architecture dependant. In our analysis we use the ARM instruction set architecture (ISA), which supports a very rich set of addressing modes. Thus, in this section we specifically address the effects of the ARM addressing modes on the register validation process.

Effects of offset type. In general, a data address is calculated using a base and an offset. The base corresponds to the value of a specified base register. The offset can be an immediate value, the value of a specified index register, or the result of an operation performed on a specified index register. For example, in the instruction {`LDR R5, [R2, R4, lsl #2]`}, the base corresponds to the value of register R2 (R2 is the base register),

while the offset corresponds to the value in register R4 that is shifted left for two bits (R4 is an offset register). Below we discuss the register validation process depending on the type of the address offset.

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

To validate the offset register, the process is similar, but may involve additional steps in case of a calculated offset (the offset is the result of an operation on the offset register). For example, if the data address is calculated as $\{DA = R2 + f(R4)\}$, where R2 is the base and R4 is the offset register and f is an operation on the offset register, the software debugger can calculate R4 only if the function f has an inverse function f^{1}, $\{R4 = f^{1}(DA-R2)\}$.

For ARM architecture, function f is usually a shift operation on an offset register. Shift operations are as follows: shift left, shift right, rotate right, no shift, and zero. Some shift operations are not invertible. For example, the shift right or left operation can drop some bits and the inverse shift operation cannot recreate the original register values. The rotate operation uses information from the status register, thus validation process must be performed on the status register too. Memory referencing instructions in our benchmarks do not use the rotate operation, thus we do not consider them in the validation process.

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

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

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

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

The ARM instruction set also includes multiple-load and multiple-store instructions. These instructions specify a set of general-purpose registers that are loaded or stored from multiple consecutive memory locations defined by the starting address. These instructions use the immediate offset addressing mode. Once the starting data address is known, the remaining data addresses are calculated relative to this address (fixed stride). Thus, only the starting address is traced if it cannot be inferred by the debugger.

A fairly sophisticated set of addressing modes in the ARM instruction set results in somewhat complex register validation logic. However, other architectures may not present such challenges. For example, the MIPS instruction set features a fairly small set of simple addressing modes (register-displacement) and the register validation logic is quite simple.

3.3 Experimental Evaluation

The goal of our experimental evaluation is to determine the percentage of memory referencing instructions whose data addresses can be filtered out using the proposed mechanism. We consider both load and store data addresses. We also report the average trace port bandwidth requirements when tracing data addresses and the total compression ratio.

Table 4 shows the filtering rates for loads (*LD*), stores (*ST*), and all memory referencing instructions (*Total*). The average filtering rate for loads[4] is 70%, ranging from 28% (for *bf_e*) to 100% (for *tiff2bw*, *tiff2rgba*). The filtering rates for store instructions are slightly higher. The forth column shows the filtering rates for all memory referencing instructions: the average is 74%, ranging from as low as 36% (*bf_e*) to 100% (*tiff2bw* and *tiff2rgba*). We can expect that these relatively high filtering rates significantly reduce trace port bandwidth requirements.

Table 4, DAF column, shows the trace port bandwidth requirements when tracing only non-filtered data addresses for each individual benchmark. We consider data addresses for all memory referencing instructions (load and stores). The required bandwidth depends on both the address filtering rate and the frequency of memory-referencing instructions. It ranges from as low as 0.05 bits/ins (*tiff2bw*) to relatively high 6.56 bits/ins (*bf_e*) and it is 2.94 bits/ins on average. Some benchmarks see a small gain from the proposed address filtering; for example, *bf_e* filters only 36% of all data addresses and requires 6.56 bits/ins instead of 10.23 bits/ins with uncompressed tracing. However, for some benchmarks (*tiff2rgba*, *tiffbw*) the filtering reduces the required trace port bandwidth for several orders of magnitude.

---

[4] We use weighted average where a benchmark weight is directly proportional to the number of load instructions in that benchmark, relative to the total number of loads in all benchmarks.

We may further reduce the size of the data addresses that need to be traced (i.e., non-filtered data addresses). The upper address bits of data addresses rarely change, so we can apply a differential encoding. The incoming data address is XOR-ed with the previous data address and the difference is divided into 8-bit chunks. If the upper difference bits are all zeros, we do not trace them and use a single-bit header bit to indicate whether an 8-bit address chunk is the terminating chunk of an address or not. The column (DAFDiff) gives the trace port bandwidth after this simple enhancement is applied. It further reduces the required trace port bandwidth: the average is reduced from 2.94 to 2.26 bits/ins.

To stress the effectiveness of the proposed filtering method, we give compression ratios for four data address reduction techniques (the last four columns of Table 4). We consider our address filtering mechanism (DAF and DAFDiff), a Nexus-like differential encoding method, and the software utility gzip (with -1 switch for fast compression) when compressing a complete data address trace. We can see that the proposed filtering method outperforms even the software utility *gzip* for majority of the benchmarks: it achieves the average compression ratio of ~5 when combined with differential encoding. The gzip achieves compression ratio of ~3.5, and the Nexus-like encoding only ~1.6.

A drawback of the proposed filtering mechanism is that it requires a close integration with the processor core to be able to carry out register validation in hardware. However, the proposed mechanism does not require any additional compression structures and its implementation cost is relatively small (logic in RVU). In addition, the proposed filtering scheme could be combined with other compressor structures, e.g., predictors that recognize regular strides [17].

4. CONCLUSIONS

Modern embedded systems rely on on-chip resources to enable and expedite software debugging and testing. Data traces collected on the target system are often required during debugging for deterministic program replay. However, capturing and tracing out full data traces at program speeds require large on-chip trace buffers and wide trace ports.

In this paper we introduce and analyze two filtering techniques aimed at reducing load data value traces and data address traces. Our cache first-access tracking mechanism significantly reduces the size of the load value trace: from 5.3 times for a system with a 4 KB data cache to 20.6 times for a system with 64 KB data cache. Our data address filtering based on a partial program replay reduces the size of the data address trace for ~5 times when combined with a simple differential encoding.

These results indicate that trace modules implementing the proposed filtering techniques would make possible a continual real-time and unobtrusive program tracing. Even better reduction ratios are desired and possible when these filtering mechanisms are combined with cost-effective hardware trace compressors; however, examining these approaches is left to future research.

5. ACKNOWLEDGMENTS

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

Table 4. Data address filtering evaluation.

	Filtering Rates			TP Bandwidth		Compression Ratio			
	LD	ST	Total	DAF	DAF Dif	DAF	DAF Dif	Nexus	gzip-1
adpcm_c	0.71	1.00	0.73	1.17	0.98	3.75	4.44	1.64	3.46
bf_e	0.28	0.50	0.36	6.56	3.93	1.56	2.60	2.19	4.84
cjpeg	0.63	0.67	0.64	4.02	3.24	2.75	3.41	1.34	4.48
djpeg	0.65	0.94	0.73	3.94	3.49	3.70	4.17	1.34	3.77
fft	0.64	0.68	0.65	2.97	1.93	2.88	4.44	2.14	19.9
ghostscript	0.64	0.87	0.72	3.25	2.70	3.59	4.31	1.76	18.14
gsm_d	0.82	0.84	0.83	1.40	1.22	5.94	6.78	1.54	23.3
lame	0.69	0.90	0.76	3.50	3.16	4.25	4.70	1.49	5.7
mad	0.83	0.79	0.82	1.98	1.49	5.66	7.54	1.43	3.54
rijndael_e	0.53	0.78	0.57	6.46	4.43	2.34	3.41	1.4	3.2
rsynth	0.65	0.79	0.68	5.46	4.00	3.17	4.32	1.41	21.53
sha	0.97	0.97	0.97	0.23	0.15	33.79	52.29	2.27	8.35
stringsearch	0.55	0.91	0.76	2.15	1.78	4.18	5.05	1.9	8.83
tiff2bw	1.00	1.00	1.00	0.05	0.04	239.14	304.53	1.66	2.55
tiff2rgba	1.00	1.00	1.00	0.07	0.05	305.23	387.74	1.02	2.79
tiffdither	0.73	0.94	0.78	1.91	1.69	4.63	5.25	1.39	4.41
tiffmedian	0.88	0.75	0.85	1.80	1.36	6.77	8.98	1.73	3.49
Average	0.70	0.83	0.74	2.94	2.26	3.82	4.97	1.59	3.46

6. REFERENCES

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

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

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

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

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

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

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

[8] Milenkovic, A. and Milenkovic, M., "Stream-Based Trace Compression," *IEEE Computer Architecture Letter* 2, 1 (Jan. 2003), 9-12. DOI= http://dx.doi.org/10.1109/L-CA.2002.9.

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

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

[11] Kao, C.-F., Huang, I.-J., and Lin, C.-H., "An Embedded Multi-resolution AMBA Trace Analyzer for Microprocessor-based SoC Integration," in *Proceedings of the 44th annual Design Automation Conference* (San Diego, California 2007). DAC '07. ACM, 477-482. DOI= http://doi.acm.org/10.1145/1278480.1278604

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

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

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

[15] Narayanasamy, S., Pokam, G., and Calder, B., "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," *SIGARCH Comput. Archit. News* 33, 2 (2005), 284-295. DOI= http://doi.acm.org/10.1145/1080695.1069994.

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

[17] Milenković, M., Milenković, A., and Burtscher, M., "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, 27-29 Mar. 2007). DCC '07. IEEE Computer Society, 55-65. DOI= http://dx.doi.org/10.1109/DCC.2007.10