Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging

VLADIMIR UZELAC, Tensilica Inc.
ALEKSANDAR MILENKOVIĆ, The University of Alabama in Huntsville

Capturing program and data traces during program execution unobtrusively on-the-fly 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. This article describes a new hardware-based load data value filtering technique called Cache First-access Tracking. Coupled with an effective variable encoding scheme, this technique achieves a significant reduction of load data value traces, from 5.86 to 56.39 times depending on the data cache size, thus enabling cost-effective, unobtrusive on-the-fly tracing and debugging.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids; tracing*; E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*

General Terms: Algorithms, Design, Verification

Additional Key Words and Phrases: Debugging, program tracing, trace compression, load value filtering, trace module, software debugger, variable encoding

ACM Reference Format:
Uzelac, V. and Milenković, A. 2013. Hardware-based load value trace filtering for on-the-fly debugging. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article 97 (May 2013), 18 pages.
DOI: http://dx.doi.org/10.1145/2465787.2465799

1. INTRODUCTION

Ever-increasing hardware and software complexity, increased integration and miniaturization, diversification and proliferation of embedded systems, and tightening time-to-market impose a number of challenges to embedded system design and verification. To cope with growing sophistication and complexity of embedded systems, software developers need to be able to gain insight into the internal system state at any point in the design and test cycle. However, high internal complexity and limited I/O bandwidth prevent complete visibility of the internal state. According to an estimate, programmers spend between 50 to 75 percent of their development time in debugging [Tassey 2002], and this fraction will likely continue to grow with a current shift toward multi-core systems and parallel software. Yet, in spite of significant investments in software debugging and testing, it is estimated that the United States alone loses approximately between $20 and $60 billion a year due to software bugs and glitches [Tassey 2002]. For example, a study found that 77% of all electronic failures

This work was supported in part by National Science Foundation grants CNS-0855237 and CNS-1217470.
Authors' addresses: V. Uzelac, Tensilica Inc., 255-6 Scott Blvd., Santa Clara, CA 95054; A. Milenković, Department of Electrical and Computer Engineering, The University of Alabama in Huntsville, 301 Sparkman Dr., AL 35899; email: milenka@ece.uah.edu.

DOI: http://dx.doi.org/10.1145/2465787.2465799

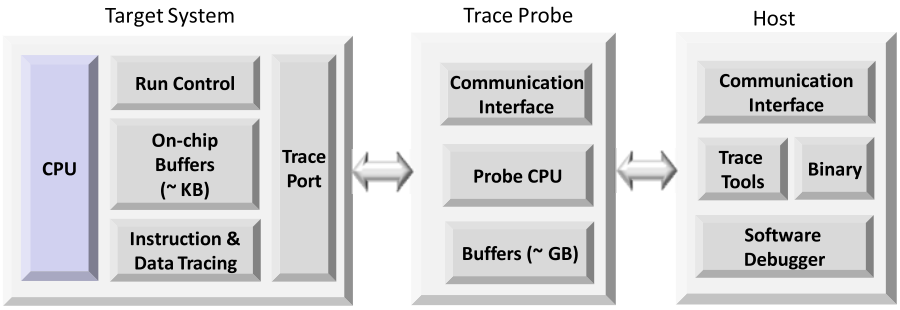

Fig. 1. Tracing and debugging in embedded systems: system view.

in automobiles are due to software bugs [McDonald-Maier and Hopkins 2004]. The recent recalls in the automotive industry are a stark reminder of the need for improved software testing and debugging. To shorten development time, reduce development cost, and minimize the number of bugs, programmers need better debugging tools.

Increasingly, programmers rely upon on-chip resources dedicated solely to program debugging. For instance, the IEEE's Industry Standard and Technology Organization has developed a standard named Nexus 5001 [IEEE-ISTO 2003] that defines functions and general-purpose interface for software development and debugging of embedded processors. Nexus 5001 specifies four classes of debug operations (Class 1–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, including single-stepping, breakpoints, and access to processor registers and memory while the processor is stopped. It is traditionally implemented through a JTAG interface [IEEE 2001]. Whereas Class 1 debug operations are widely deployed and routinely used, they are lacking in several important aspects. First, setting breakpoints and examining the processor state to locate difficult and intermittent bugs in large software projects is demanding and time-consuming for programmers. Second, setting a breakpoint is often not practical in debugging real-time embedded systems, for example, it may be harmful for hard drives or engine controllers. Third, debugging through breakpoints interferes with program execution. The order of events during debugging may deviate from the order of events when the program is running natively with no interference from debugging operations; this in turn can cause original bugs to disappear in the debug run.

Class 2 provides support for nearly unobtrusive capturing and streaming out program execution traces in near real time. Program execution traces record the program's control flow and are invaluable for hardware and software debugging, as well as for program profiling. However, for certain classes of software bugs (e.g., data races), program execution traces alone are insufficient and data traces are required too. Class 3 provides support for capturing and streaming out memory and I/O read and write (load and store) data values and addresses, in addition to the program's control flow. Whereas data traces are crucial in reconstructing program execution in single-core systems, they are critical in multi-core systems, as they offer valuable information about shared memory access patterns and possible data race conditions. Finally, Class 4 adds resources for direct processor control through the trace port: instructions and data are fetched from the trace port instead of main memory.

Figure 1 illustrates a typical embedded processor with its trace and debug module. It encompasses logic for run-control debugging (Class 1), logic to capture and filter program execution traces (Class 2) and data traces (Class 3), on-chip buffers for storing

traces (in order of kilobytes), and a trace port that connects the target system to an external trace unit (trace probe) or directly to a development workstation (host machine). The external trace probe typically includes a probe processor for control, a communication interface to the host (e.g., Ethernet or USB), and very large trace buffers (in order of gigabytes). The host machine runs a software debugger and other trace processing tools that can read and analyze traces, allowing programmers to step forward and backward through the program execution. This way, programmers are able to gain complete visibility into the target system and its behavior, while the target processor is running at full speed.

Many vendors have introduced modules with program tracing capabilities that can be integrated into their platforms. They usually support Class 1 operations, often Class 2, and optionally Class 3. Some examples include ARM's Embedded Trace Module [ARM 2004, 2007], MIPS's PDTrace [MIPS 2009], and Tensilica's TRAX-PC [Tensilica 2009]. Commercial trace modules require trace port bandwidth in the range of 1 to 4 bits per instruction per core for program execution traces, and 8 to 16 bits per instruction per core for data traces [Orme 2008]. Thus, an internal one-kilobyte trace buffer can capture the execution of a program segment of about 8,000 instructions on average (or about 2,000 instructions in the worst case), if a program trace is collected, or a program segment of about 400 to 800 instructions, if a data trace is collected. Such short segments are often insufficient in locating software errors in modern processors where distances between bug sources and their manifestations may be in millions or billions of instructions.

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

Filtering and compressing program execution traces at runtime in hardware can reduce the requirements for on-chip trace buffers and trace port communication bandwidth. Commercially available trace modules typically implement only rudimentary forms of hardware compression with a relatively small compression ratio for program execution traces and data address traces. Data traces are typically streamed out uncompressed. Whereas several academic proposals have addressed real-time hardware-based compression of program execution traces [Kao et al. 2007; Milenković et al. 2011; Uzelac and Milenković 2009], the more challenging problem of real-time hardware-based reduction of data value traces has not been directly addressed so far. This article focuses on load data value traces (Section 2) that are, under certain conditions, sufficient to deterministically reconstruct the whole program offline in the software debugger. Our work supports software debugging as specified by the Nexus 5001 and assumes a correct hardware design. The proposed method is not directly applicable to post-silicon debugging. A detailed treatment of capturing and real-time compression of debug data for post-silicon verification can be found in Daoud and Nicolici [2009].

In this article, we introduce a hardware filtering mechanism called *Cache First-access Tracking* mechanism (Section 3) that reduces trace port bandwidth requirements, thus enabling practical on-the-fly load data value tracing. Data caches are augmented by first-access tracking bits that determine whether a load value needs

to be streamed out of the chip, or whether it can be inferred by the software debugger (Section 3.1). The software debugger maintains its copy of the data cache with corresponding first-access tracking bits that are updated during program replay, using identical policies to those used in the trace module. This way, the trace module needs to send trace messages to the software debugger only on first-access miss events. Trace messages include the number of consecutive first-access hits and the load data value on which a first-access miss event has occurred. We discuss granularity of first-access tracking bits (Section 3.2) and introduce a variable encoding of the first-access hit counter (Section 3.3) to further reduce the trace port bandwidth.

Our experimental analysis confirms the excellent performance of the first-access tracking mechanism (Section 4). We explore the design space and evaluate effectiveness of the proposed mechanism as a function of the data cache size (Section 4.1). We also describe selection of good encoding parameters (Section 4.2) that yields a minimal trace size. The compression ratio, defined as the size of the raw load data value trace divided by the size of the filtered trace, ranges from 5.86:1 in a system with a 4 KB data cache to 56.39:1 in a system with a 64 KB data cache (Section 4.3). Finally, we discuss implementation issues and estimate complexity of the proposed mechanism (Section 4.4).

The main contributions of this work are as follows.

(1) We introduce a hardware-based mechanism for filtering load data values called the Cache First-access Tracking mechanism.
(2) We introduce an effective, low-complexity encoding scheme that adapts to benchmark behavior and data cache configurations to further minimize the size of the output trace, enabling cost-effective and unobtrusive load data value tracing in real time.
(3) We perform a detailed experimental analysis that shows that the proposed mechanism achieves excellent compression ratios. For example, a system with a 32 KB data cache requires bandwidth of only 0.211 bits per instruction on the trace port, which is a 38-fold improvement over the uncompressed load data value trace.

2. LOAD DATA VALUE TRACING

A software debugger can replay program execution deterministically offline if the following five 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 program execution trace containing information about exceptions; (d) it has access to the load data value trace captured on the target processor; and (e) it knows the initial state of general- and special-purpose registers. Consequently, capturing the load data value trace on the target processor and streaming it out of the processor chip are critical in program debugging. However, capturing and streaming out load data values in near real time may be cost-prohibitive because they require wide trace ports and large on-chip trace buffers.

To illustrate challenges associated with load data value tracing, we profile seventeen representative benchmarks from the MiBench suite [Guthaus et al. 2001] compiled for the ARM instruction set. Table I shows the instruction count (IC), an adjusted instruction count (IC*), and the dynamic frequency of load instructions (*ld-all*), classified into byte loads (*ldb*), half-word loads (*ldh*), word loads (*ldw*), and double-word loads (*lddw*). The ARM instruction set [ARM 2005] supports load-multiple and store-multiple instructions. A load-multiple instruction specifies a number of general-purpose registers that are loaded from a block of data in memory. However, a single load-multiple instruction in the processor pipeline appears as multiple single-load instructions [Intel 2004]. Thus, we report the adjusted instruction count (IC*), where each load-multiple

Table I. MiBench Program Statistics Related to Load Data Value Tracing

	IC	IC*	Frequency of load instructions					LD.DVT
	[mil.]	[mil.]	ldb [%]	ldh [%]	ldw [%]	lddw [%]	ld-all [%]	[bpi]
adpcm_c	732.52	732.77	3.64	0.00	9.11	0.00	12.75	3.21
bf_e	544.06	758.63	2.14	0.00	25.21	0.00	27.35	8.24
cjpeg	104.61	107.74	5.57	0.00	21.74	0.00	27.30	7.40
djpeg	23.39	23.92	13.35	0.00	20.02	0.00	33.37	7.48
fft	631.04	726.59	0.83	0.00	18.55	2.54	21.93	7.63
ghostscript	708.10	827.47	0.23	6.07	20.16	0.00	26.46	7.64
gsm_d	1299.27	1329.18	2.19	0.14	14.21	0.02	16.56	4.63
lame	1285.12	1386.35	0.38	2.16	19.66	10.98	33.18	13.69
mad	287.09	298.03	3.23	0.00	25.00	0.00	28.23	8.26
rijndael_e	319.98	352.12	6.40	0.00	33.34	0.00	39.74	11.18
rsynth	824.94	868.76	0.92	0.00	37.98	1.39	40.30	13.12
sha	140.89	142.89	2.27	0.00	14.33	0.00	16.60	4.77
stringsearch	3.68	4.10	3.20	0.00	12.30	0.00	15.50	4.19
tiff2bw	143.26	143.94	19.85	0.01	7.05	0.00	26.91	3.85
tiff2rgba	151.70	152.61	18.72	0.02	20.82	0.00	39.56	8.16
tiffdither	832.95	860.66	3.22	5.90	12.29	0.00	21.41	5.13
tiffmedian	541.26	542.49	12.28	0.16	18.61	0.00	31.05	6.96
Total			***3.13***	***1.44***	***19.88***	***1.98***	***26.43***	***8.11***

and store-multiple instruction is counted as multiple instructions (equal to the number of load or store operations). The last column of Table I (LD.DVT) shows the trace port bandwidth required to stream all load data values out of the processor chip. The trace port bandwidth of a benchmark is calculated as the size of all loaded values during program execution divided by the adjusted instruction count; it is expressed in the average number of bits per instruction executed (bpi).

The load data value trace port bandwidth ranges between 3.2 bpi (*adpcm_c*) and 13.69 bpi (*lame*), which is close to a range of 8 to 16 bpi reported for commercial modules [Orme 2008]. The required bandwidth depends on frequency of load instructions as well as on the size of loaded data. For example, *lame* has a relatively high frequency of load instructions, with almost 20 percent of word loads and 11 percent of double-word loads, which results in the required bandwidth of 13.69 bpi on the trace port. Similar observations can be made for the *rsynth* and *rijndael_e* benchmarks. The *tiff2rgba* benchmark has an even larger percentage of load instructions, but about half of them are byte loads (∼19 percent byte loads and ˜21 percent word loads), so the required bandwidth is 8.16 bpi. On the other side, the *adpcm_c* and *stringsearch* benchmarks have relatively small frequency of byte and word loads, resulting in the bandwidths of 3.2 and 3.8 bpi, respectively. The row marked as *Total* shows the total load frequencies and the trace port bandwidth for the entire benchmark suite. The total bandwidth is calculated as the total number of bits of all loaded values in the benchmark suite divided by the total adjusted number of instructions.

The total bandwidth of 8.11 bpi indicates a high cost of load data value tracing. For example, to unobtrusively capture a load data value trace for a program segment of 100,000 instructions, one would need a trace buffer of 800 kilobytes, which is cost-prohibitive. Unfortunately, load data values exhibit limited redundancy, so a straightforward approach to compressing load data values using general compression algorithms yields little benefit. For example, the software *gzip* utility achieves the total compression ratio of only 3.5:1 for our benchmark suite. In addition, implementing general-purpose compression algorithms in hardware would be cost-prohibitive

and infeasible for real-time compression. This underscores a need for alternative approaches to reduce the size of the load data value trace.

3. LOAD DATA VALUE FILTERING USING CACHE FIRST-ACCESS MECHANISM

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 also be augmented to help reduce load data value trace size. We do not need to stream out 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 data 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 stream its data value out to the debugger. In addition, if a load hits in the data cache, we still may need to stream 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 already been 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 (*c-fiat*). It is based on a mechanism used in the BugNet [Narayanasamy et al. 2005] 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 the system crashes. Its first-access tracking 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 unobtrusive tracing of load data values in real time in embedded systems, and thus help program debugging.

3.1. Cache First-Access Tracking Mechanism

Figure 2 shows the system view of the proposed Cache First-access Tracking 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. For the moment, we assume that a first-access flag is assigned to 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. However, the size of the object protected by a first-access flag is a design parameter, and a flag can protect a larger object, such as a half-word or a word. 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 3 describes the trace module operation for the Cache First-access Tracking mechanism. For each load instruction, the module 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], we say we have an *FA hit event*. In this case, we do not need to stream out the

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

[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).

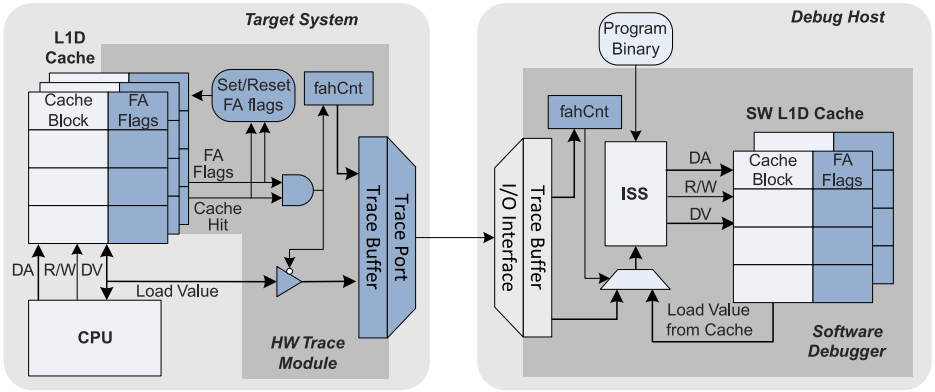

Fig. 2. Cache First-access Tracking mechanism: system view.

```
// For each retired load that reads n bytes
1.    if (CacheHit) {
2.      if (corresponding n FA flags are set)
3.              fahCnt++;
4.      else {
5.              Emit trace record into Trace Buffer (fahCnt, loadValue);
6.              Set corresponding n FA flags;
7.              fahCnt = 0;
8.      }
9.    } else { // cache miss event
10.     Clear FA bits for newly fetched cache block;
11.     Perform steps 5-7;
12.   }

// For each retired store that writes n bytes
13.   Set corresponding n FA bits;

// For external invalidation/update request
14.   Clear FA bits for entire cache block;
```

Fig. 3. Cache first access tracking: trace module operation.

load value because the software debugger can find it in its cache model. To synchronize the trace module and the software debugger, the trace module can report this event by sending a single-bit trace message [Uzelac and Milenković 2010]. A more efficient alternative is to keep track of the number of consecutive FA hits using a local register called *fahCnt* (first-access hit counter). In case of an FA hit event, we just increment *fahCnt*, and no trace message is streamed out (line 3 in Figure 3). 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 the current value of the counter *fahCnt* to indicate an *FA miss event* (lines 5–7 in Figure 3). 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 (line 10). The load value is traced out and the FA flags are set accordingly (line 11).

The cache first-access flags are also updated on store instructions and on signals triggered outside of the processor core, for example, cache block invalidations caused by the cache controller (Figure 3). Each store will set the corresponding first-access flags because its value in the cache becomes known (and can be inferred by the debugger) (line 13). Note: here we assume the data cache has write-allocate, write-back

```
// For each load that reads n bytes
1.  fahCnt --;
2.  if (fahCnt > 0) {
3.     Perform lookup in the SW data cache;
4.     if (corresponding n FA bits in SW cache are set)
5.            Retrieve data value from SW cache;
6.     else
7.            ERROR in Tracing; // illegal event
8.  }
9.  else { // FA miss event
10.    Read n bytes from trace record;
11.    Update SW cache;
12.    Set corresponding n FA flags in SW cache;
13.    Get the next trace record (fahCnt, LoadValue);
14. }

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

Fig. 4. Execution replay in the software debugger.

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 (line 14). In addition, if an external hardware module directly writes into the data cache (e.g., cache injection mechanism [Milenković 2000]), the corresponding first-access flags need to be cleared too. These actions do not need to be synchronized with the software debugger—the debugger always checks the trace input first during the program replay. Finally, in case of exceptions or interrupts, load data value trace alone is insufficient to replay the program offline. We assume that an exception control-flow (exception entry and exit) trace augments the load trace value. In our prior work, we showed that such a trace requires a minimal bandwidth on the trace port [Uzelac et al. 2010], and in this article, we do not further consider exception trace messages.

The software debugger running on the host machine reads and decodes the trace messages and replays the program. The debugger relies on its ISS with the software model of the data cache, a software copy of the first-access hit counter (*fahCnt*), the program binary, and the load data value trace received from the target platform for program replay (Figure 2). Its steady-state operation is described in Figure 4. For each load instruction, the debugger decrements a software copy of the *fahCnt* counter (line 2). If the *fahCnt* value is positive, that means that this as an *FA hit* event and that the load value should be retrieved from the local copy of the data cache (lines 3–7). Otherwise, this is an *FA miss* event. The load data value is retrieved from the trace message, and the software copy of the data cache and FA flags are updated accordingly (lines 10–13). For a store instruction, the debugger updates the software data cache and sets the FA flags accordingly (lines 15–16).

3.2. First-Access Flag Granularity

By profiling frequencies of load instructions with respect to their type (byte, half-word, word, and double word), we can see that a relatively small percentage of all instructions are byte and half-word loads: about 3.1 percent of instructions are byte loads and about 1.4 percent are half-word loads (see Table I). However, several benchmarks have a significant portion of such instructions (e.g., *tiff2bw*, *tiff2rgba*, and *tiffmedian*). One important question is what should be the size of a data object protected by a single first-access bit.

So far we have assumed that a single first-access bit guards the smallest addressable unit, which is a byte. In case of a multi-byte memory referencing instruction, multiple first-access bits are set or reset accordingly. If we further assume a data cache with 32-byte cache blocks, the complexity overhead caused by first-access bits is 32 bits or 4 bytes per one cache block, which is 1/8th of the data cache capacity. Thus, in a system with 64 kilobyte data cache, this overhead reaches 8 kilobytes of storage devoted to first-access flags. Although this extra complexity may well be justified by overall reduction in the trace buffer sizes, we may consider an alternative approach. 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. Not only does this approach reduce the complexity overhead of the proposed mechanism, it may lower the number of trace messages that need to be streamed out of the chip.

To illustrate trade-offs in selecting an optimal granularity of first-access flags, let us consider an example program that sequentially reads characters in a string aligned to a cache block boundary. The first load results in a data cache miss; the requested cache block is read from memory, and all first-access flags are cleared. When first-access flags protect each byte, we will see 32 first-access miss events, and consequently 32 trace messages will be streamed out. Each trace message carries *fahCnt* value and an 8-bit load value. Alternatively, if first-access flags protect each word, we will have only 8 miss first-access miss events, and consequently only 8 trace messages will be streamed out. Each trace message carries an *fahCnt* value and a 32-bit load data value. It should be noted that all load byte and load half-word instructions that miss on the first-access flags will trigger streaming out of the entire 32-bit word that they belong to, rather than 8-bit bytes or 16-bit half-words. The only drawback of using first-access flags to protect words rather than bytes is possible in programs where load byte instructions dominate and a memory referencing pattern is such that not all bytes within a word are used. However, we have found no such access patterns in our benchmark suite and observed only positive effects of using word-size granularity of first-access flags. Thus, in the rest of the article, we assume that a first-access flag is assigned to a 32-bit word in the data cache.

3.3. Encoding Trace Messages

Trace messages should be encoded in such a way that minimizes the trace port bandwidth requirements and enables a simple and efficient implementation. The trace module sends trace messages to the software debugger on first-access miss events. Each trace message consists of the first-access hit counter, *fahCnt*, and the actual load value. A straightforward approach to encoding trace messages is to use a fixed-length field for the *fahCnt* counter value. The length of the field that carries information about the load value depends on the type of load instruction (*ldb*, *ldh*, *ldw*, *lddw*) and the granularity of the first-access flag. Whereas the size of the load value field is defined as *max(sizeof(loadValue), FA granularity)*, the size of the *fahCnt* field is more challenging to determine.

The *fahCnt* carries the information about the number of consecutive first-access hits, and it is a function of the load data cache hit rate and the first-access hit rate. In turn, the load cache hit rate is a function of the data cache size and organization, and spatial and temporal locality for a given benchmark program. The first-access hit rate depends on the load data cache hit rate, load data type, load access patterns, and the granularity of first-access bits. For example, larger data caches will result in higher load data cache hit rates, and consequently in higher first-access hit rates. Next, higher first-access hit rates will result in longer runs of consecutive hits, requiring more bits to encode the value of the *fahCnt* counter in a trace message. This value is expected to vary across different data cache configurations and benchmarks, but also within a

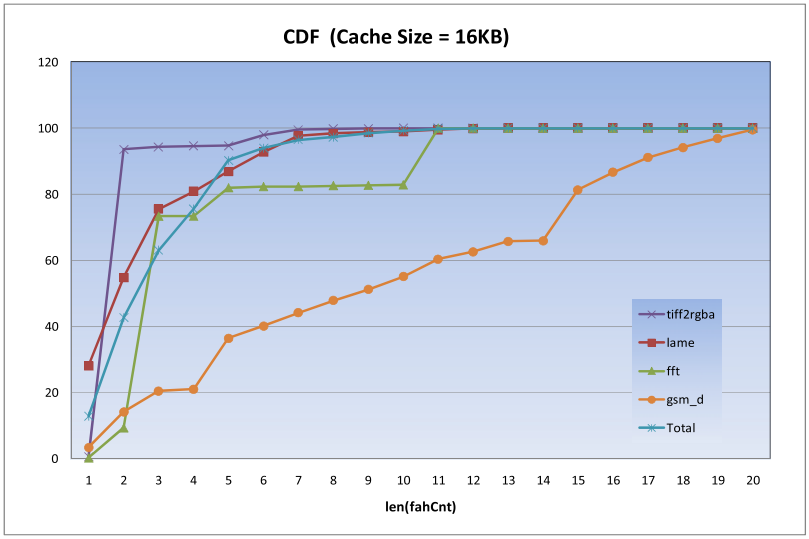

Fig. 5. Cumulative distribution function for the minimum *fahCnt* length.

benchmark, as it moves through different execution phases. For example, first-access miss events are more likely to occur in program warm-up phases, and the values found in the *fahCnt* counter are likely to be rather small.

To illustrate challenges in encoding of the *fahCnt* trace message field, we profile the values found in this field. We assume a 16 kilobyte four-way set associative data cache with 32-byte cache blocks. First-access bits are assigned to each word in the data cache. Figure 5 shows the cumulative distribution function (CDF) for the minimum length of the *fahCnt* field in trace messages, *len(fahCnt)*, for several characteristic benchmark programs. The *gsm_d* and *fft* benchmarks exhibit almost perfect load data cache hit and first-access hit rates. On the other hand, the *lame* benchmark exhibits relatively high load data cache hit rate and somewhat smaller first-access hit rate, whereas *tiff2rgba* exhibits a medium high load data cache hit rate, but rather low first-access hit rate. The line marked as *Total* shows the cumulative distribution function for the minimum length of the *fahCnt* field when all benchmarks in the suite (Table I) are considered together.

A fixed eight-bit field can encode the *fahCnt* values from 0 to 255. However, the total CDF shows that over 60 percent of all *fahCnt* values in trace record messages require no more than three bits, resulting in a significant waste of trace port bandwidth—at least five out of eight bits will be unused in 60 percent of trace messages. On the other hand, a significant number of trace messages require more than eight bits for the *fahCnt* value in the *fft* and *gsm_d* benchmarks. For example, more than 50 percent of all trace messages require more than eight bits for the *gsm_d* benchmark. The challenge is to devise an encoding scheme that will work well across different benchmarks and data cache configurations and yet minimize the number of bits streamed out through the trace port. To meet this challenge, we opt for a variable encoding scheme and an empirical approach to determine good encoding parameters.

In our encoding scheme, all trace messages start with the field that carries the *fahCnt* value. The length of this field is variable: after eliminating the leading zero bits, the *fahCnt* counter bits are divided into a certain number of chunks (chunks do not necessarily need to be of equal size) (Figure 6). Each chunk is followed by a so-called connect bit (*C*) that indicates whether it is a terminating chunk for the

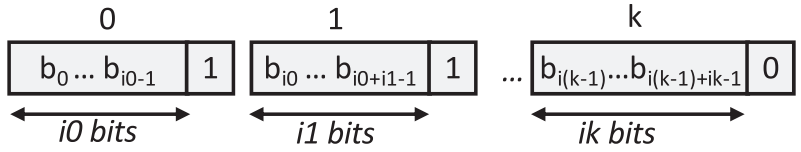

Fig. 6. Encoding *fahCnt* values in trace records.

fahCnt field (*C = 0*), or it is followed by more chunks carrying relevant bits for the *fahCnt* value (*C = 1*). For example, a trace message that includes a two-bit chunk '11' followed by a connect bit with value '0' indicates a first-access miss event that occurred after three consecutive first-access hits. If the first chunk ends with a connect bit *C = 1*, more relevant bits follow in the subsequent chunk. Let us assume that the following chunk is also two-bit long, and its value is '10' and *C = 0*. This trace record thus carries information that the *fahCnt* is '11 _10' or seven in the decimal number system.

The length of individual chunks (*i0*, *i1*, ... *ik*) is a design parameter that will be determined in an experimental analysis. In determining the length of individual chunks, we need to balance the overhead caused by the connect bits (shorter chunks will result in a relative increase in the overall number of the connect bits) and the number of wasted bits in individual chunks (longer chunks result in lower overhead, but possibly have more unused bits).

4. EXPERIMENTAL EVALUATION

The goal of our experimental evaluation is to thoroughly explore the effectiveness of the proposed first-access tracking mechanism in filtering load data value traces. We consider a range of data cache configurations, from 4 to 64 kilobyte data caches. As a measure of effectiveness, we use compression ratio, which is calculated as the size of the unfiltered load value trace divided by the size of the load data value trace after filtering. To illustrate suitability of the proposed method for unobtrusive tracing in real time, we also report the trace port bandwidth calculated in bits per instruction retired. The bandwidth depends on several parameters: data cache hit rate, first-access hit rate, load data size (e.g., byte, half-word, word, double word), and the frequency of load instructions. To illustrate the impact of these parameters on the overall compression ratio, we will report both load data cache hit rate and first-access hit rate (Section 4.1). In addition to these parameters, encoding of the *fahCnt* field in trace messages also impacts the overall performance. Consequently, we first discuss results of experimental evaluation aimed at finding good encoding parameters (Section 4.2). Next, for the chosen set of encoding parameters, we analyze compression ratio and trace port bandwidth (Section 4.3). Finally, we perform a complexity estimation of the proposed mechanism and discuss several implementation issues (Section 4.4).

The data cache subsystem is modeled after the XScale processor. Apart from the data cache size, other data cache parameters are selected to yield the maximum performance at minimal cost. Thus, all data caches are four-way set-associative structures with 32-byte cache blocks, use write-allocate and write-back policies, and a pseudo least recently used replacement policy that is based on the most-recently used bit [Al-Zoubi et al. 2004]. As a workload, we use 17 benchmarks from the MiBench suite shown in Table I. Our analysis is performed using a functional SimpleScalar ARM simulator [Austin et al. 2002]. In calculating hit rates, load-multiple and store-multiple instructions from the ARM ISA are considered as multiple load and store instructions, corresponding to IC* in Table I.

Table II. Load Hit Rate (LCHR) and First-Access Hit Rate (FAHR)

	LCHR					*FAHR*				
	4 KB	*8 KB*	*16 KB*	*32 KB*	*64 KB*	*4 KB*	*8 KB*	*16 KB*	*32 KB*	*64 KB*
adpcm_e	0.999	1.000	1.000	1.000	1.000	0.995	1.000	1.000	1.000	1.000
bf_e	0.983	0.999	1.000	1.000	1.000	0.946	0.995	1.000	1.000	1.000
cjpeg	0.930	0.986	0.993	0.994	0.994	0.872	0.933	0.951	0.959	0.962
djpeg	0.972	0.990	0.996	0.999	1.000	0.899	0.950	0.977	0.995	1.000
fft	0.997	0.999	0.999	0.999	0.999	0.991	0.993	0.993	0.993	0.993
ghostscript	0.997	0.998	0.999	0.999	0.999	0.991	0.994	0.995	0.996	0.997
gsm_d	0.999	1.000	1.000	1.000	1.000	0.998	1.000	1.000	1.000	1.000
lame	0.945	0.976	0.990	0.994	0.996	0.757	0.898	0.954	0.973	0.981
mad	0.975	0.994	0.997	1.000	1.000	0.808	0.954	0.983	0.998	0.999
rijndael	0.852	0.983	0.999	1.000	1.000	0.638	0.940	0.996	1.000	1.000
rsynth	0.995	0.998	0.999	0.999	0.999	0.981	0.992	0.995	0.995	0.996
sha	0.996	0.999	1.000	1.000	1.000	0.966	0.992	1.000	1.000	1.000
stringsearch	0.993	0.996	0.998	0.999	0.999	0.965	0.973	0.988	0.993	0.994
tiff2bw	0.943	0.946	0.968	0.999	1.000	0.560	0.585	0.761	0.997	1.000
tiff2rgba	0.919	0.919	0.926	0.951	0.961	0.362	0.362	0.417	0.615	0.685
tiffdither	0.988	0.991	0.995	1.000	1.000	0.916	0.932	0.965	0.999	1.000
tiffmedian	0.942	0.946	0.956	0.982	0.996	0.765	0.776	0.846	0.943	0.986
Total	0.969	0.986	0.992	0.996	0.998	0.871	0.929	0.957	0.979	0.986

4.1. Design Space Exploration

Table II shows load data cache hit rate (LCHR) and first-access hit rate (FAHR) for all benchmarks, while varying the data cache size from 4 to 64 kilobytes. The load data cache hit rate is determined as the number of load requests that hit in the data cache divided by the total number of load requests. Note that a load instruction that reads a double word (64-bit) from memory may span two cache blocks, resulting in four possible hit/miss scenarios. To capture this behavior faithfully, we count such loads as two requests. The first-access hit rate is determined as the number of loads that find all corresponding first-access flags set divided by the total number of load requests. The effectiveness of the proposed mechanism is directly influenced by the first-access hit rate—higher first-access hit rates mean fewer trace messages that need to be streamed out to the software debugger. The last row in the table marked as *Total* shows the hit rates for the entire benchmark suite (hit rates are calculated as the total number of hits in all benchmarks divided by the total number of load requests in all benchmarks).

The total load data cache hit rate is relatively high regardless of the data cache size. It ranges from 96.2 percent for a system with a 4-kilobyte data cache to 99.8 percent for a system with 64-kilobyte data cache. However, considering individual benchmarks, it ranges from 85.2 percent for *rijndael* in a system with 4-kilobyte data cache to 100 percent for a number of benchmarks (e.g., *adpcm_e*, *bf_e*, *gsm_d*) in a system with larger caches. The total first-access hit rate is rather high too, even with very small caches. It ranges from 87.1 percent for a system with 4-kilobyte data cache to 98.6 percent for a system with 64-kilobyte data cache. These results confirm our expectations that the proposed mechanism can indeed dramatically lower the number of load values that needs to be streamed out of the target processor. Whereas the total first-access hit rate is relatively high regardless of the data cache size, several benchmarks exhibit rather small first-access hit rates, in spite of having relatively large load data cache hit rates. For example, the first-access hit rate is only 36.2 percent for *tiff2rgba* in a system with a 4-kilobyte data cache (load data cache hit rate is 91.9 percent), and reaches 68.5 percent for a system with a 64-kilobyte data cache (load hit rate is 96.1

percent). Several benchmarks exhibit low first-access hit rates in systems with a small data cache, but benefit significantly from larger data caches (e.g., *tiff2bw* and *rijndael*). These diverse behaviors are caused by unique memory access patterns and the amount of spatial and temporal locality found in individual benchmarks.

One interesting question is how changes in data cache parameters other than the data cache size may impact our findings. For example, changes in cache replacement policy impact load hit rates and thus first-access hit rates. However, we found that these changes are not significant. Using write-no-allocate policy may also have small impact on first-access hit rate. Finally, increasing the data cache block size will result in higher load data cache hit rates, but somewhat smaller first-access hit rates. We repeated our experiments for 64-byte data cache blocks, but found that the total first-access hit rate remained rather high, ranging from 85.8 percent in a system with a 4-kilobyte data cache to 98.5 percent in a system with 64-kilobyte data cache.

4.2. Encoding Parameters Selection

To select good chunk sizes for the proposed variable encoding, we profile the behavior of MiBench benchmarks by analyzing the cumulative distribution function of the minimum length of the *fahCnt* field in the trace messages, while varying the data cache size (Figure 5 shows the CDF for a system with 16-kilobyte data cache). Each benchmark has its own set of parameters that yield minimal size of the output trace for a given data cache size. However, here we seek for a set of parameters that results in a minimal size of the output trace when all benchmarks are considered together. It should be noted that the proposed encoding makes benchmark-wise customization of chunk sizes practical—it can be done before tracing, by initializing trace module control registers based on typical program profiles. Accordingly, the software debugger should decode trace messages using the same set of parameters.

In search of good values for chunk sizes $i0$, $i1$, $i2$, $\ldots ik$ (Figure 6), we limit the design space by requiring that $i1 = i2 = \ldots = ik$. We vary the parameters $i0$, $i1 \in [1, 6]$ and search for a combination of parameters that yields the minimum size of the output trace. Figure 7 shows normalized compression ratio for the entire benchmark suite as a function of encoding parameters; the pair of parameters (i0, i1) = (1, 1) is used as the base in normalization. We consider a subset of nine best-performing encoding pairs, from (i0, i1) = (1, 1) to (i0, i1) = (3,3) for all data cache sizes. Somewhat surprisingly, we find that the (i0, i1) = (1, 1) pair yields the smallest output trace in systems with relatively small data caches (4–8 kilobytes). In systems with larger data caches (16–64 kilobytes), a pair (i0, i1) = (1, 2) yields the minimum output trace. These findings can be explained as follows. The total size of the filtered load data value trace for the entire benchmark suite is dominated by benchmarks that have a large number of instructions, a high frequency of load instructions, and relatively small first-access hit rates (e.g., *lame*, *tiff2rgba*, *tiffmedian*). In these benchmarks, first-access miss events are more frequent and clustered, thus favoring shorter chunk sizes. If we consider individual benchmarks, we find that optimal chunk sizes are, for example, (i0, i1) = (2, 3) for *gsm_d* and *tiffdither* for all data cache sizes. In general, chunk sizes yielding minimal output traces for individual benchmarks depend on the size of the data cache—smaller caches favor shorter chunks and larger caches favor longer chunks.

An interesting question is sensitivity of the output trace size to the selection of chunk size parameters. The results in Figure 7 show that encoding parameters (1,1), (1,2), (1,3), (2,1), (2,2), and (2,3) produce output traces of sizes within 2–3 percent of each other. This result suggests that variable encoding remains stable for a subset of good encoding parameters; however other tested pairs may result in larger differences. Again, it should be noted that here we discuss the compression ratio for

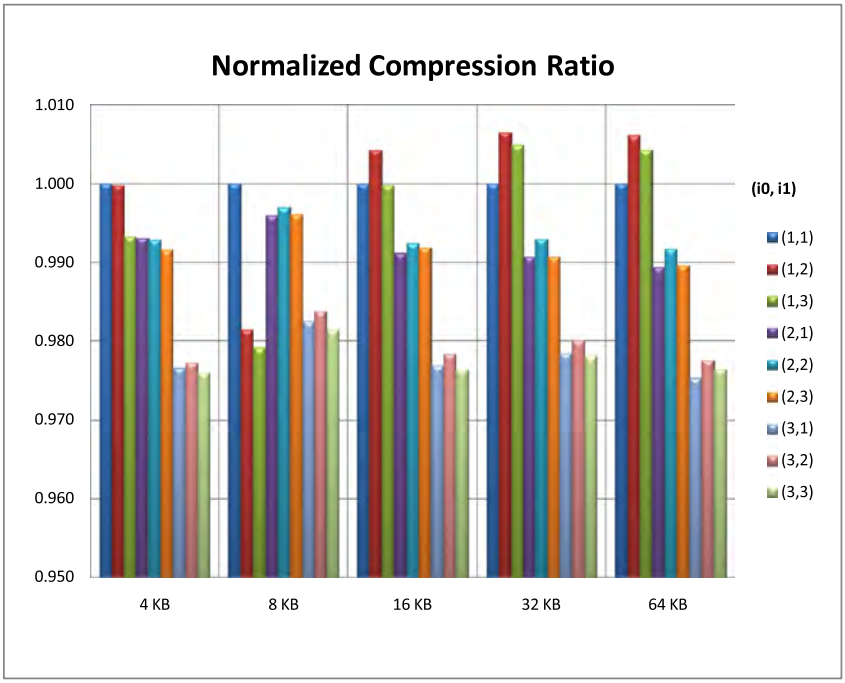

Fig. 7. Normalized compression ratio as a function of (i0,i1) encoding parameters.

the entire benchmark suite. Individual benchmarks may show larger sensitivity to changes in encoding parameters. Overall, a fraction of the total number of bits used to encode *fahCnt* values relative to the total trace size (*fahCnt* values and load data values) does not vary significantly with the data cache sizes; it ranges between 8.45 percent to 10.15 percent. This result confirms a significant stability of the proposed encoding.

Finally, an important question relates to the overall effectiveness of the proposed variable encoding. To shed more light on this question, we compare the size of the output trace with variable encoding for the best set of parameters with the size of the output trace where the trace module sends a single-bit header for each load instruction to indicate whether it is a first-access hit or miss event (no *fahCnt* counter is used). The results show that variable encoding provides higher compression for all data cache sizes—it outperforms the alternative encoding from 1.11 times in a system with a 4-kilobyte data cache to 2.57 times in a system with 64-kilobyte data cache.

4.3. Compression Ratio/Trace Port Bandwidth Analysis

Table III shows a compression ratio achieved by the proposed mechanism (*c-fiat*) as a function of the data cache size. It also shows the trace port bandwidth expressed in bits per instruction retired; it is calculated as the size of the filtered output trace divided by the number of retired instructions in a benchmark. To illustrate the effectiveness of the proposed filtering mechanism, we compare it with the software *gzip* utility when using it to compress the raw load data value trace. We use the gzip utility as a yard-stick because it is the most frequently used general-purpose compressor. In addition, a hardware implementation of LZ77 has been proposed for program trace compression, though at a hefty cost in additional complexity [Kao et al. 2007].

Table III. Compression Ratio and Trace Port Bandwidth: A Comparative Analysis

	Compression Ratio						Trace Port Bandwidth (bits/ins)					
	4KB c-fiat	8KB c-fiat	16KB c-fiat	32KB c-fiat	64KB c-fiat	gzip-1	4 KB c-fiat	8 KB c-fiat	16 KB c-fiat	32 KB c-fiat	64 KB c-fiat	gzip-1
adpcm_e	132.4	65742.6	66346.1	66346.1	66346.1	4.1	0.02	0.00	0.00	0.00	0.00	0.78
bf_e	15.2	133.9	5848.7	67479.6	67479.6	4.1	0.54	0.06	0.00	0.00	0.00	1.99
cjpeg	5.8	10.6	14.2	16.9	18.2	6.3	1.27	0.70	0.52	0.44	0.41	1.18
djpeg	6.0	11.7	24.9	111.9	1293.8	5.6	1.24	0.64	0.30	0.07	0.01	1.34
fft	93.9	133.6	137.6	138.8	139.3	4.7	0.08	0.06	0.06	0.05	0.05	1.61
ghostscript	84.7	114.4	152.5	181.0	206.9	12.3	0.09	0.07	0.05	0.04	0.04	0.62
gsm_d	292.9	2047.7	30489.7	113862.6	113862.6	3.4	0.02	0.00	0.00	0.00	0.00	1.34
lame	3.5	8.2	17.8	31.1	45.1	2.6	3.96	1.66	0.77	0.44	0.30	5.66
mad	4.4	17.7	47.2	420.6	937.2	2.3	1.90	0.47	0.18	0.02	0.01	3.58
rijndael	2.2	12.1	174.4	22100.5	30329.3	2.4	5.04	0.92	0.06	0.00	0.00	4.73
rsynth	42.4	94.8	179.8	191.8	203.4	3.4	0.31	0.14	0.07	0.07	0.06	3.90
sha	23.9	104.3	8228.0	8269.3	8269.3	2.5	0.20	0.05	0.00	0.00	0.00	1.94
stringsearch	21.1	28.2	61.9	108.5	120.4	6.0	0.20	0.15	0.07	0.04	0.03	0.70
tiff2bw	0.9	1.0	1.7	149.3	2815.0	3.3	4.12	3.90	2.23	0.03	0.00	1.17
tiff2rgba	0.9	0.9	1.0	1.6	1.9	3.3	8.68	8.68	7.92	5.18	4.24	2.48
tiffdither	7.5	9.2	18.0	1020.8	17149.7	4.0	0.69	0.56	0.28	0.01	0.00	1.29
tiffmedian	2.7	2.8	4.0	10.6	42.6	3.9	2.62	2.50	1.72	0.66	0.16	1.78
Total	***5.9***	***10.9***	***18.5***	***38.4***	***56.4***	***3.4***	***1.38***	***0.74***	***0.44***	***0.21***	***0.14***	***2.40***

The proposed mechanism proves highly effective in reducing the load data value trace size. The total compression ratio for the entire benchmark suite ranges from 5.86:1 in a system with a very small 4-kilobyte data cache, to 56.39:1 for a system with a large 64-kilobyte data cache. The fast gzip (*gzip -1*) software utility achieves compression ratio of 3.41:1. Please note that a hardware implementation of the software *gzip* utility would be cost-prohibitive in both required additional on-chip area and the compression latency. In a system with a 32-kilobyte data cache, the proposed mechanism outperforms *gzip* utility for over 11 times, which further underscores its strength.

Analyzing individual benchmarks, we can observe that almost all benchmarks benefit from the proposed filtering mechanism, even with very small data cache sizes. Notable exceptions are two benchmarks, *tiff2rgba* and *tiff2bw*, which perform poorly in systems with very small data caches (4 KB and 8 KB). The additional overhead caused by message encoding results in having the output trace slightly larger than the original load data value trace (compression ratio is below 0.93:1 and 0.94:1). This is not an unexpected result because these two benchmarks demonstrate low first-access hit rates in systems with small data caches. However, even these two benchmarks see benefits of the proposed mechanism for larger data cache sizes. As expected, the benchmarks with high load data cache hit and first-access hit rates experience very high compression ratios (e.g., *gsm_d*, *adpcm_e*). The compression ratio improves significantly in systems with larger data cache sizes.

The total trace port bandwidth (row *Total* in Table III, Figure 8) ranges from 1.38 bpi in a system with a 4-kilobyte data cache to 0.144 bpi in a system with a 64-kilobyte data cache. For all benchmarks except three (*tiff2rgba*, *tiff2bw*, and *tiffmedian*), the required trace port bandwidth is less than 1 bpi for a system with 16-kilobyte data cache. For a system with a 32-kilobyte data cache, all benchmarks except one (*tiff2rgba*) require less than 1 bpi on the trace port, promising real-time, continual, and unobtrusive tracing of load data values using a very narrow trace port (e.g., JTAG). Although the proposed mechanism reduces the size of the output trace of the *tiff2rgba* benchmark for configurations with larger caches, the compression ratio remains relatively small (1.6 times in a system with 32-kilobyte data cache). This unfortunately translates into trace port bandwidth that is well above 1 bpi.

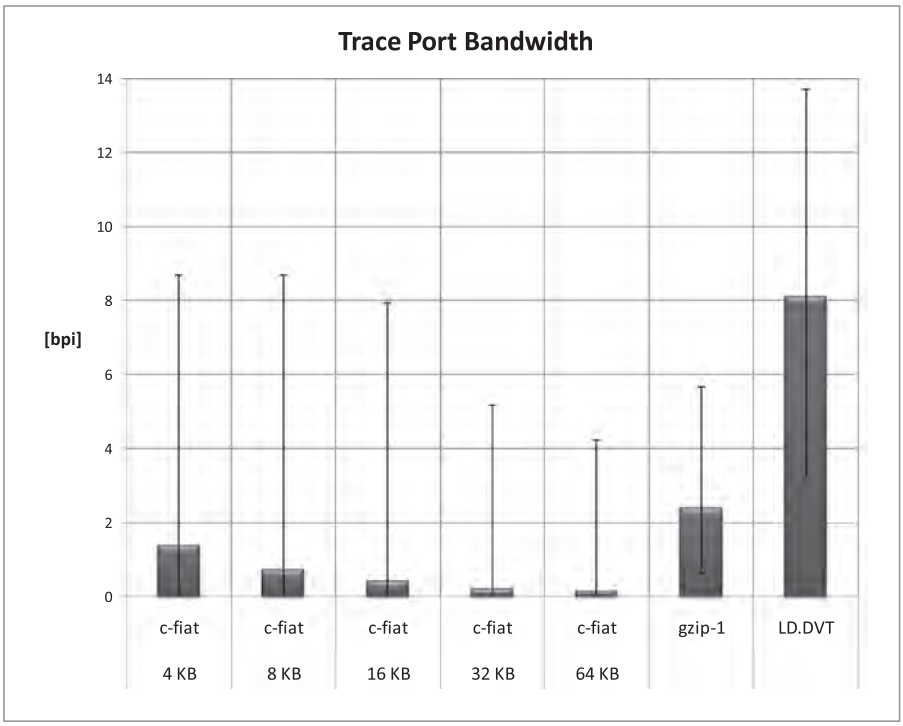

Fig. 8. Trace port bandwidth evaluation.

4.4. Hardware Complexity and Implementation Issues

The proposed mechanism requires modest additional hardware resources. The major complexity overhead comes from the storage needed for first-access flags ($1/32^{nd}$ of the data cache capacity). Thus, the overhead depends of the data cache size, and ranges from 128 bytes of extra storage in a system with a 4-kilobyte data cache to 2 kilobytes in a system with a 64-kilobyte data cache. Negligible overhead comes from logic that controls the first-access flags (set or reset) and trace encoding.

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 cannot be changed, an alternative design can be used where the first-access flags are physically placed inside the trace module instead of 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.

To quantitatively estimate complexity overhead caused by the first-access flags, we use Cacti tools (version 5.3) that report the area occupied by the tag and the data memory portions of the cache structures [Thoziyoor et al. 2008]. Table IV shows on-chip area for the cache configurations considered in our experimental evaluation, assuming 45 nm technology. We show the tag area, the data memory area, and the total cache area for the base cache configurations (columns 2–4). The next two columns show the data portion and the total area of the cache augmented with the first-access flags (columns 5–6). The results confirm our qualitative analysis and show that storage overhead

Table IV. Complexity Estimates in On-Chip Area

Size [KB]	BASE CACHE AREA [μm^2]				CACHE + FA flags AREA [μm^2]		Over-head		External FA flags	Over-head
	Tag	Data	Total		Data + FA	Total	Norm. to Base		Tag + FA flags	Norm. to Base
4	2,834	30,758	33,592		31,714	34,547	1.028		3,795	1.113
8	4,864	59,486	64,350		61,227	66,090	1.027		6,723	1.104
16	8,318	86,367	94,685		88,842	97,159	1.026		11,017	1.116
32	14,743	126,325	141,068		130,273	145,016	1.028		18,690	1.132
64	28,926	247,338	276,264		254,186	283,112	1.025		36,655	1.133

for the first-access flags ranges between 2.5 and 2.8 percents (column 7). In addition, we consider a configuration when the first-access flags are physically placed in the trace module. In addition to the first-access flags this structure requires replication of the cache tags. The results show that in this case on-chip area overhead ranges between 10.4 and 13.3 percent, confirming feasibility of this approach as well. In addition to on-chip area analysis, the Cacti tool reports estimates for access times to the cache structures. We found that additional latency, when flags are added to the cache, never exceeds 0.2% of the base cache configuration time. The trace buffer in the proposed trace module (Figure 2) serves only to temporarily store trace records before they are streamed 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 first-access miss events). 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.

5. CONCLUSIONS

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

In this article, we introduce and analyze a filtering mechanism called Cache First-access Tracking that significantly reduces the size of load data value traces at modest cost in additional hardware complexity and corresponding changes in the software debugger. When combined with a variable encoding scheme, the proposed method reduces the size of the load value trace from 5.86 times for a system with a 4 KB data cache to 56.39 times for a system with 64 KB data cache. These results indicate that trace modules implementing the proposed filtering technique would make possible 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.

REFERENCES

Al-Zoubi, H., Milenković, A., and Milenković, M. 2004. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd Annual Southeast Regional Conference*. 267–272. http://doi.acm.org/10.1145/986537.986601.

ARM. 2004. CoreSight On-chip Debug and Trace Technology. http://www.arm.com/products/solutions/CoreSight.html.

ARM. 2005. Architecture and Implementation of the ARM®Cortex™-A8 Microprocessor. http://www.arm.com/pdfs/TigerWhitepaperFinal.pdf.

ARM. 2007. Embedded Trace Macrocell Architecture Specification. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014o/IHI0014O_etm_v3_4_architecture_spec.pdf.

Austin, T., Larson, E., and Ernst, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Comput. 35*, 59–67.

Daoud, E. A. and Nicolici, N., 2009. Real-time lossless compression for silicon debug. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. 28*, 1387–1400.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization*. 3–14.

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

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

Intel. 2004. Intel XScale®Core Developer's Manual. http://download.intel.com/design/intelxscale/27347302.pdf.

Kao, C.-F., Huang, S.-M., and Huang, I.-J. 2007. A hardware approach to real-time program trace compression for embedded processors. *IEEE Trans. Circuits Syst. 54*, 530–543.

McDonald-Maier, K. D. and Hopkins, A. B. T., 2004. An awakening thought: Don't let the bug bite while you are embedded. *Embed. Syst. Eng. 12*, 32–33.

Milenković, A. 2000. Achieving high performance in bus-based shared-memory multiprocessors. *IEEE Concurrency 8*, 3, 36–44.

Milenković, A., Uzelac, V., Milenković, M., and Burtscher, M. 2011. Caches and predictors for real-time, unobtrusive, and cost-effective program tracing in embedded systems. *IEEE Trans. Comput. 60*, 992–1005.

MIPS. 2009. MIPS PDtrace Specification. http://www.mips.com/products/product-materials/processor/mips-architecture/.

Narayanasamy, S., Pokam, G., and Calder, B., 2005. BugNet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News 33*, 284–295.

Orme, W. 2008. Debug and trace for multicore SoCs. http://www.arm.com/files/pdf/CoresightWhitepaper.pdf.

Tassey, G. 2002. The economic impacts of inadequate infrastructure for software testing. http://www.rti.org/pubs/software_testing.pdf.

Tensilica. 2009. Non-intrusive Real-Time Trace Debug. http://www.tensilica.com/products/hw-sw-dev-tools/for-software-developers/real-time-trace-3.htm.

Thoziyoor, S., Muralimanohar, N., Ahn, J. H., and Jouppi, N. P. 2008. *CACTI 5.1*. http://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf?q=cacti.

Uzelac, V. and Milenković, A. 2009. A Real-time program trace compressor utilizing double move-to-front method. In *Proceedings of the 46th Annual Design Automation Conference*. 738–743.

Uzelac, V. and Milenković, A. 2010. Hardware-based data value and address trace filtering techniques. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 117–126. http://doi.acm.org/10.1145/1878921.1878940.

Uzelac, V., Milenković, A., Burtscher, M., and Milenković, M. 2010. Real-time unobtrusive program execution trace compression using branch predictor events. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 97–106. http://portal.acm.org/citation.cfm?doid=1878921.1878938 [Accessed October 16, 2011].

Received June 2011; revised October 2011; accepted December 2011