Real-time, Unobtrusive, and Efficient Program Execution Tracing with Stream Caches and Last Stream Predictors *

Vladimir Uzelac[§], Aleksandar Milenković[§], Milena Milenković[¥], Martin Burtscher[†]
[§]*ECE Department, The University of Alabama in Huntsville*
[¥]*IBM, Austin, Texas,* [†]*ICES, The University of Texas at Austin*

***Abstract*—This paper introduces a new hardware mechanism for capturing and compressing program execution traces unobtrusively in real-time. The proposed mechanism is based on two structures called stream cache and last stream predictor. We explore the effectiveness of a trace module based on these structures and analyze the design space. We show that our trace module, with less than 600 bytes of state, achieves a trace-port bandwidth of 0.15 bits/instruction/processor, which is over six times better than state-of-the-art commercial designs.**

I. Introduction

Debugging and testing of embedded processors is traditionally done through a JTAG port that supports two basic functions: stopping the processor at any instruction or data access, and examining the system state or changing it from outside. This approach is obtrusive and may cause the order of events during debugging to deviate from the order of events during "native" program execution without interference from debug operations. These deviations can cause the original problem (e.g., a data race) to disappear in the debug run. In addition, stepping through the program is time-consuming for programmers and is simply not an option for debugging real-time embedded systems, where setting a breakpoint may be impossible or harmful. A number of even more challenging issues arise in multi-core systems. They may have multiple clock and power domains, and we must be able to support debugging of each core, regardless of what other cores are doing. Debugging through a JTAG port is not well suited to meet these challenges.

Recognizing these issues, many vendors have developed modules with tracing capabilities and integrated them into their systems on a chip (SoCs), e.g., ARM's ETM [1], MIPS's PDTrace [2], and Infinion's OCDS [3]. The IEEE's Industry Standard and Technology Organization has proposed a standard for a global embedded processor debug interface (Nexus 5001) [4].

The trace and debug infrastructure on a chip typically includes logic that captures address, data, and control signals within SoCs, logic to filter and compress the trace information, buffers to store the traces, and logic that emits the content of the trace buffers through a trace port to an external trace unit or host machine. Traces can be classified into three categories, depending on the type of information they contain: program traces, data traces, and interconnect traces.

In this paper we focus on program execution traces that record which instructions were executed. Such traces are widely used for hardware and software debugging as well as for program optimization and tuning. In their basic form, these traces consist of the addresses of all committed instructions. However, they can be more compactly represented by a trace that reports only changes in the program's control flow.

Many existing trace modules employ trace compression and buffering to achieve about 1 bit/instruction/CPU bandwidth at the trace port at the cost of about 7,000 gates [5]. They rely on large on-chip buffers to store execution traces of sufficiently large program segments and/or on relatively wide trace ports that can transfer a large amount of trace data in real-time. However, large trace buffers and/or wide trace ports substantially increase the system complexity and cost. Moreover, they do not scale well, which is a significant problem in the era of multicore chips. Whereas commercially available trace modules typically implement only rudimentary forms of hardware compression, several recent research efforts in academia propose compression techniques tailored to program execution traces that can achieve much higher compression ratios [6-8]. For example, Kao et al. [6] propose an LZ-based program trace compressor that achieves a good compression ratio for a selected set of programs. However, the proposed module has high complexity (over 50,000 gates), and it is unclear how effective this module would be in tracing more diverse programs.

In this paper we introduce a very cost-effective mechanism for capturing and compressing program execution traces unobtrusively in real-time. The proposed trace module relies on two new structures called stream cache and last stream predictor to translate a sequence of program streams into a sequence of hit or miss events in these structures (Section 2). We also introduce several enhancements of the original mechanism that reduce the trace port bandwidth or module complexity and size. We explore tradeoffs in the design of the proposed trace module and evaluate its effectiveness (Section 3). The experimental evaluation based on a set of benchmarks from the MiBench suite [9] shows that our trace module with less than 600 bytes of state can achieve a trace

* *This is a corrected version of the original paper.*

port bandwidth of only 0.15 bits/instruction/CPU, which is over six times lower than state-of-the-art commercial solutions.

II. Program Tracing and Debugging with Stream Caches and Last Stream Predictors

A program execution can be replayed offline by recording changes in the program flow caused by control-flow instructions and exceptions (including interrupts). When a change in the program flow occurs, we need to capture the starting address of a new instruction stream, which is either the target address of the currently executing branch instruction or the starting address of an exception handler. Consequently, the program execution can be recreated by recording information about program streams, also known as dynamic basic blocks. Each instruction stream can be uniquely represented by its starting address (SA) and its length (SL). The pair (SA, SL) is called stream descriptor and it replaces the complete trace of instruction addresses in the corresponding stream.

A sequence of full stream descriptors (SA, SL) is suboptimal for recording program executions, because it includes redundant information that can be inferred directly from the program binary during program replay. For example, if an instruction stream starts at the target of a direct conditional branch, a partial stream descriptor (-, SL) will suffice because its starting address can be inferred from the binary. Next, if an instruction stream ends with a direct unconditional branch, we may opt not to terminate the current instruction stream at that instruction. Rather, the stream continues with the next instruction, which is located at the target address of the branch. Our approach includes these modifications to minimize the size of the trace records.

Most programs have only a small number of unique program streams, with just a fraction of them responsible for the majority of the program execution. Table 1 shows some important characteristics of the MiBench [9] programs collected using SimpleScalar [10] running ARM binaries. The columns (a-d) show the number of executed instructions in millions (*IC*), the number of unique streams (*USC*), and the maximum (*maxSL*) and average stream length (*avgSL*), respectively. The total number of unique streams traversed during program execution is fairly limited – it ranges from 341 (*adpcm_c*) to 6871 (*ghostscript*), and the average dynamic stream length is between 5.9 (*bf_e*) and 54.7 (*adpcm_c*) instructions. The fifth column (e) shows the number of unique program streams that constitute 90% of the dynamically executed streams. This number ranges between 1 (*adpcm_c*) and 235 (*lame*) and is 78 on average. Note that all calculations use a weighted average, where the weights are based on the number of executed instructions, since the size of the raw instruction trace is directly proportional to the number of executed instructions. The maximum stream length in our benchmarks never exceeds 256, thus we may choose to use 8 bits to represent SL (MaxSL=255).

Table 1. MiBench Program Characteristics.

	IC (mil.)	*USC*	*max SL*	*avg SL*	*CDF 90%*
adpcm_c	733	341	71	54.7	1
bf_e	544	403	70	5.9	22
cjpeg	105	1590	239	12.3	47
djpeg	23	1261	206	25.1	31
fft	631	846	94	10.5	209
ghostscript	708	6871	251	10.0	67
gsm_d	1299	711	165	19.5	33
lame	1285	3229	237	32.4	235
mad	287	1528	206	20.7	42
rijndael_e	320	513	77	21.0	45
rsynth	825	1238	180	17.6	49
stringsearch	4	436	65	6.0	48
sha	141	519	65	15.4	10
tiff2bw	143	1038	43	12.8	2
tiff2rgba	152	1131	75	27.7	2
tiffmedian	541	1335	92	22.3	5
tiffdither	833	1777	67	14.3	63
Average	**816**	**1791**	**145**	**21.6**	**77.8**
	(a)	*(b)*	*(c)*	*(d)*	*(e)*

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

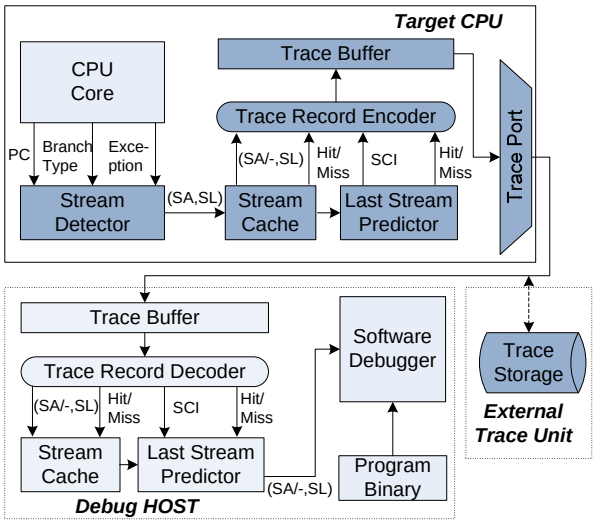

Fig. 1. System view of program tracing and replay.

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

tors that, in conjunction with the program binary, provide enough information for a complete program replay off-line.

The proposed mechanism performs the capture and compression of program execution traces in three stages. In the first stage, a stream detector detects instruction streams and forwards their descriptors to the second stage. In the second stage, a stream cache (SC) translates the stream descriptors into stream cache indices (SCI). In the third and final stage, a last stream predictor (LSP) compresses the SCI trace into trace records that are stored into a trace buffer (Fig. 1).

Stream Detector. Whenever a control-flow instruction of a certain type (direct conditional, indirect, or return) or an exception causes the program flow to depart from sequential execution, the stream detector captures the current stream descriptor (SA, SL) and prepares for the beginning of a new instruction stream by recording its starting address in the SA register and zeroing out the SL register (Fig. 2). The captured stream descriptors (SA, SL) are placed into a stream buffer that serves to smoothen out possible bursts of short streams.

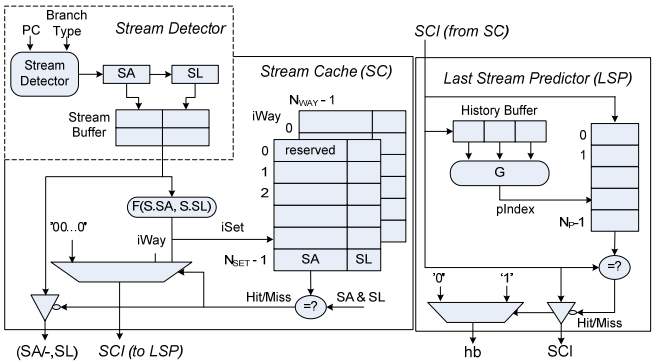

Fig. 2. The three trace module structures.

Stream Cache. To exploit the observed program characteristics, the second stage consists of a cache-like structure called stream cache with N_{WAY} ways and N_{SET} sets (Fig. 2). An entry in the stream cache keeps a complete stream descriptor (SA, SL). The SC translates a stream descriptor into a relatively short stream-cache index as follows. A stream descriptor is read from the stream buffer and a stream cache lookup is performed. A set in the stream cache is calculated as a simple function of the stream descriptor, e.g., bit-wise XOR of selected bits from SA and SL. If the incoming stream descriptor matches an entry in the selected set, we have an SC hit; the corresponding stream cache index SCI, determined by concatenating the set and way indices (SCI={*iSet*, *iWay*}), is forwarded to the next stage. In case of an SC miss, a reserved index zero is emitted (SCI=0). If necessary, an entry in the selected set is evicted based on the replacement policy (e.g., least recently used) and updated with the incoming stream descriptor.

The compression ratio achieved by our stream detection and stream cache compression, CR(SC), is defined as the ratio of the raw instruction address trace size, calculated as the number of instructions multiplied by the address size (*IC*4* bytes), and the total size of the SCI output (Eq. 1). It can be expressed analytically as a function of the *avgSL*, the stream cache hit rate (*hrSC*), the stream cache size (N_{SET}*N_{WAY}), and the probability that a stream starts with a target of an indirect branch (p_{IND}). For each instruction stream (the number of dynamic streams is equal to *IC/avgSL*), $\log_2(N_{SET}*N_{WAY})$ bits are emitted to the SCI output. On each stream cache miss, a 5-byte (SA, SL) or 1-byte (-, SL) stream descriptor is output. Eq. 1 is useful in exploring the design space with the goal to maximize the compression ratio. The parameters *avgSL* and p_{IND} are benchmark dependent and cannot be changed except maybe through program optimization. Smaller stream caches require shorter indices but likely have a lower hit rate, which negatively affects the compression ratio. Thus, a detailed exploration of the stream cache design space is necessary to determine a good hash access function and stream cache size and organization.

Eq. 1 $$CR(SC) = \frac{4 \cdot avgSL}{0.125 \cdot \log_2(N_{SET} \cdot N_{WAYS}) + (1 - hrSC) \cdot [p_{IND} \cdot 5 + (1 - p_{IND}) \cdot 1]}$$

Last Stream Predictor. The third stage is designed to exploit redundancy in the SCI output. Perfect trace compression without stream pattern recognition would replace each stream with just a single bit. We can approach this goal by using a simple last value predictor as shown in Fig. 2. A linear predictor table with N_P entries is indexed by a hash function that is based on the history of previous stream cache indices. If the selected predictor entry contains a match for the incoming stream index, we have an LSP hit. Otherwise, we have an LSP miss, and the selected predictor entry is updated by the incoming stream cache index.

In case of an LSP hit, a single-bit trace record with header bit (hb) ′1′ is placed into the trace buffer. Upon seeing this bit, the debugger's decompressor retrieves the current stream descriptor from its software structures that mirror those in the trace module. In case of an LSP miss, a trace record with a single-bit header ′0′, followed by the value of the stream cache index is placed in the trace buffer. Finally, in case of an SC miss, a trace record with a header bit ′0′ followed by SCI=0 and the stream descriptor (SA/-, SL) is placed in the trace buffer.

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

Eq. 2 $$CR(LSP) = \frac{1}{(1 - hrLSP) + 1/\log_2(N_{SET} \cdot N_{WAYS})}$$

III. EXPERIMENTAL EVALUATION

The goal of our experimental evaluation is threefold. First, we explore the design space to find good parameters for the proposed structures and access functions. As a measure of performance we use the average number of bits emitted per instruction on the trace port, which is equivalent to 32/(compression ratio). Second, we introduce several enhancements and explore their effectiveness in further improving the compression ratio at minimal added complexity or in reducing the trace module size. Finally, we compare the effectiveness of the proposed mechanism to several recent proposals from the literature.

A. Design Space Exploration

Stream Cache Access Function. We have evaluated a number of stream cache access functions. Access functions combining the SA and SL portions of the stream descriptor outperform those based solely on the SA because multiple streams can have the same starting address. Our experiments indicate that the hash function shown in Eq. 3 performs the best for different sizes and configurations of the trace module. The SA is shifted by *Shift* bits and then the result is XOR-ed with the SL. The lower bits of this result are used as the set index, *iSet*. With our benchmarks and inputs, the optimal value for *Shift* was found to be 4.

Eq. 3 $$iSet = (SA << Shift)\ xor\ SL$$

Stream Cache Size and Organization. Fig. 3 shows the average trace port bandwidth when varying the total number of entries in the stream cache as well as the number of ways (N_{WAYS}=1, 2, 4, 8). The results reflect the weighted average for the whole benchmark suite. The trace port bandwidth is calculated assuming an LSP with the same number of entries as the SC and a simple hash access function that uses the previous stream cache index to access the LSP.

The results show that increasing the stream cache associativity helps reduce the bandwidth on the trace port and thus improves the compression ratio, but only up to a point. Increasing the associativity beyond 4 ways yields little or no benefit. The results further indicate that even relatively small stream caches with as few as 32 entries perform well, achieving less than 0.5 bits/instruction (bits/ins) on the trace port. Increasing the SC and consequently the LSP size beyond 256 entries is not beneficial as it only yields diminishing returns in compression ratio. Based on these results, we believe a 4-way associative stream cache with 128 entries and a 128-entry LSP to be a good choice for our trace module. This configuration represents a sweet spot in the trade-off between trace port bandwidth and design complexity; on our benchmarks, it yields under 0.2 bits/ins at a modest cost.

Last Stream Predictor. We have evaluated several LSP organizations. The number of entries in the LSP may exceed the number of entries in the stream cache. In this case, the LSP access function should be based on the program path taken to a particular stream. The path information may be maintained in a history buffer as a function of previous stream cache indices. However, our results indicate that such configurations provide fairly limited improvements in trace port bandwidth. The reason is that our workload has a relatively small number of indirect branches, and those branches mostly have a very limited number of targets taken during program execution. Consequently, we chose the simpler solution of always having the same number of entries in the LSP and the SC, which only requires an access function that is solely based on the previous stream cache index. We call this scheme BASE.

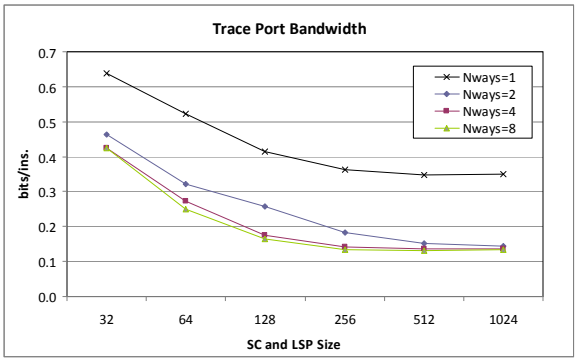

Fig. 3. Trace port bandwidth as a function of SC size and organization.

Table 2 shows the trace port bandwidth for individual benchmarks and for different sizes of the SC and LSP. The trace port bandwidth for a trace module configuration <32x4, 128> (i.e., 4-way associative 128-entry SC and 128-entry LSP) varies between 0.019 bits/ins for *adpcm_c* and 0.616 bits/ins for *fft* and is 0.174 bits/ins on average. The *fft* benchmark significantly benefits from an increase in SC size and requires 0.354 bits/ins for the <64x4, 256> configuration. Many of the remaining benchmarks perform well even with very small trace module configurations, e.g., *adpcm_c*, *tiff-median*, and *tiff2rgba*.

B. Enhancements for Reducing Trace Port Bandwidth

The output trace records have a lot of redundant information that can be eliminated with low-cost enhancements. The three components of the output trace are (i) the LSP hit trace (*hLSPt*), (ii) the LSP miss with SC-hit trace (*hSCt*), and (iii) the LSP miss and SC miss trace (*mSCt*). With small configurations, the *mSCt* component dominates the output trace; e.g., it is responsible for 41.3% of the total output trace for the <16x4, 64> configuration. For larger configurations, the *hLSPt* component dominates the output trace with long runs of consecutive ones; e.g., the *hSLPt* represents 48.5% of the total output trace for the <64x4, 256> configuration.

By analyzing the *mSCt* we observe that the upper address bits of the starting address (SA) field rarely change. To take

advantage of this property we slightly modify our BASE compressor as follows. An additional *u*-bit register called LVSA is added to record the *u* upper bits of the SA field from the last miss trace record. The upper *u*-bit field of the SA of each incoming miss trace record is compared to the LVSA. If there is a match, the new miss trace record will include only the lower (32-*u*) address bits. Otherwise, the whole 32-bit address is emitted and the LVSA register is updated accordingly. To distinguish between these two cases, an additional bit is needed in the trace record to indicate whether all or only the lower address bits are emitted. Note that SA[1:0] is always '00' for the ARM ISA and is omitted from the *mSCt*. For the ARM Thumb ISA only SA[0] can be omitted.

Table 2. Trace port bandwidth requirements for the BASE scheme.

Program/Size	32	64	128	256	512	1024
adpcm_c	0.019	0.019	0.019	0.019	0.019	0.019
bf_e	0.405	0.359	0.357	0.384	0.410	0.437
cjpeg	0.204	0.138	0.131	0.134	0.140	0.146
djpeg	0.125	0.093	0.075	0.070	0.072	0.075
fft	1.492	1.007	0.616	0.354	0.256	0.219
ghostscript	1.585	0.823	0.232	0.227	0.229	0.234
gsm_d	0.103	0.094	0.086	0.077	0.076	0.075
lame	0.129	0.108	0.102	0.101	0.100	0.093
mad	0.295	0.136	0.129	0.124	0.124	0.128
rijndael_e	0.743	0.284	0.192	0.099	0.105	0.111
rsynth	0.382	0.245	0.175	0.116	0.115	0.122
sha	0.178	0.097	0.101	0.106	0.111	0.116
stringsearch	1.369	0.938	0.472	0.387	0.401	0.382
tiff2bw	0.151	0.135	0.104	0.087	0.083	0.084
tiff2rgba	0.114	0.077	0.045	0.040	0.040	0.040
tiffdither	0.332	0.249	0.190	0.164	0.157	0.160
tiffmedian	0.085	0.077	0.066	0.058	0.055	0.056
WAverage	0.426	0.272	0.174	0.142	0.136	0.135

The experimental analysis shows that this enhancement reduces the *mSCt* component by 18% for the <32x4, 128> configuration when *u*=14. It should be noted that the reduction in the total output trace size is much more significant for smaller configurations and insignificant for larger configurations (where the miss trace component is relatively small).

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

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

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

C. Enhancements for Reducing Size

The LVSA enhancement could be slightly modified to reduce the overall size of the trace module implementation. For example, the uppermost 12 bits do not change with a probability of 0.99 in our benchmarks. Consequently, we may opt not to keep the upper address bits SA[31:20] in the stream cache, thus reducing its size. The upper address bits are handled entirely by the LVSA register. This requires one additional modification. In the LVSA enhancement, we only considered trace records in the miss trace (*mSCt*), whereas here we need to continuously update the LVSA register, regardless of whether we have a hit or a miss in the SC and LSP structures. Moreover, a miss in the LVSA register results in sending a stream descriptor to the output trace; the SC and LSP are updated accordingly. To determine the optimal number of upper bits that should be handled by the LVSA predictor, we need to consider the stream cache performance. Reducing the number of address bits that are stored in the stream cache reduces its size but may result in an increased stream cache miss rate and thus increase the trace port bandwidth. A modified eBASE with the uppermost 12 address bits handled by the LVSA appears optimal on our benchmarks. We call this final scheme rBASE.

D. Putting it all Together

Fig. 4 shows the trace port bandwidth for a range of trace module configurations and the three presented schemes. The eBASE scheme reduces the trace port bandwidth for small trace module configurations, predominantly due to a reduction in the miss trace size. For example, the average trace port bandwidth for eBASE with the <8x4, 32> configuration is 0.35 bits/ins, compared to 0.43 bits/ins for BASE. Similarly, for large trace module configurations the hit trace is significantly reduced. For example, eBASE with the <128x4, 512> configuration requires 0.11 bits/ins instead of 0.14 bits/ins for BASE. Some benchmarks benefit significantly from the

AOLC enhancement, especially those with a high *hLSP* rate, such as *adpc_c* (over 14 times higher compression), *tiff2bw* (3.45), and *tiff2rgba* (3.67).

The rBASE scheme requires slightly higher bandwidth on the trace port than eBASE. For example, the trace module configuration <32x4, 128> achieves 0.15 bits/ins at the trace port versus 0.146 bits/ins for the eBASE scheme. However, this degradation is less than 3%, an acceptable tradeoff for significant savings in the size of the stream cache.

To underline the effectiveness of the proposed mechanism, we compare it with the software compression utility *gzip* that implements the Lempel-Ziv compression algorithm. It should be noted that *gzip* uses large memory buffers and implementing it in hardware would be cost-prohibitive. We use the stream descriptor sequences as an input for *gzip*. The *gzip* utility achieves a trace port bandwidth of 0.031 bits/ins with small buffers (gzip -1) taking stream descriptors as inputs. We also evaluated a recent adaptation of the LZ algorithm for compressing program execution traces [6]. It achieves 0.47 bits/ins on our benchmark suite with a sliding dictionary of 256 12-bit entries at a total cost of 51,678 gates. Using a very large buffer of 8192 12-bit entries, this scheme approaches 0.1 bits/ins, which is comparable to our proposed mechanism, but requires much larger resources

To estimate the size of the proposed trace module, we need to include the size of all structures, including the SC, the LSP, the stream buffer, and the output trace buffer. The output trace buffer capacity should be such that the CPU never has to stall due to program tracing and that no trace records are lost. We assume a single-bit trace port that can output one bit per CPU clock cycle. Using a cycle-accurate processor model that corresponds to Intel's XScale CPU we find that the worst case in our benchmark suite is *ghostscript* (80 bits in the trace output buffer). The estimates of the total storage requirements in the trace module with the <32x4, 128> configuration are as follows: it is equivalent to 771 bytes of storage for BASE, 775 bytes for eBASE, and 582 bytes for rBASE.

IV. Conclusion

This paper describes a new low-cost hardware mechanism for the real-time compression of program execution traces. The mechanism exploits temporal and spatial locality of program streams using two new structures called stream cache and last stream predictor to achieve compression ratios that are over six times higher than commercial state-of-the-art solutions'. We introduce several enhancements to the BASE scheme and explore their effectiveness. Our smallest scheme rBASE achieves compression ratios between 32,000:1 and 58:1 on MiBench, with an average of 213:1 (which translates into 0.15 bits/ins on the trace port) at the cost of 582 bytes of state and some extra logic.

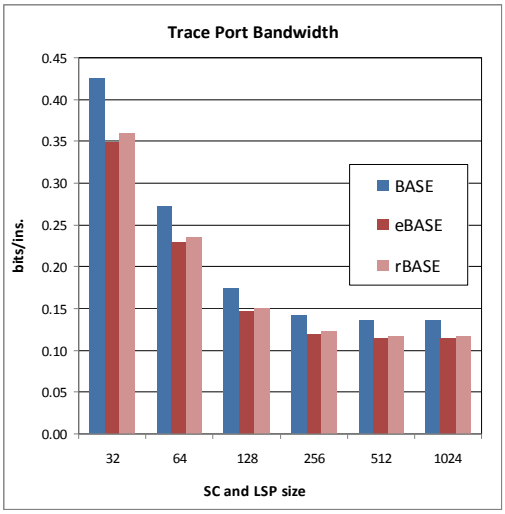

Fig. 4. Trace port bandwidth requirements for BASE, eBASE, and rBASE.

References

[1] ARM, "Embedded Trace Macrocell Architecture Specification."

[2] MIPS, "The Pdtrace™ Interface and Trace Control Block Specification," 2008.

[3] Infineon, "Tc1775 System Units 32-Bit Single-Chip Microcontroller," 2001.

[4] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface," IEEE- Industry Standards and Technology Organization (IEEE-ISTO), 2003.

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

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

[7] M.-C. Hsieh and C.-T. Huang, "An Embedded Infrastructure of Debug and Trace Interface for the DSP Platform," *45th Design Automation Conference*, 2008.

[8] M. Milenkovic, A. Milenkovic, and M. Burtscher, "Algorithms and Hardware Structures for Unobtrusive Real-Time Compression of Instruction and Data Address Traces," *Data Compression Conference*, pp. 283-292, 2007.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Workshop on Workload Characterization*, 2001.

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