Algorithms and Hardware Structures for Unobtrusive Real-Time Compression of Instruction and Data Address Traces

Milena Milenkovic[§], Aleksandar Milenkovic[†], and Martin Burtscher[¥]
[§] IBM, Austin, Texas
[†]Electrical and Computer Engineering Department, University of Alabama in Huntsville
[¥] Computer Systems Laboratory, Cornell University, Ithaca, New York

Abstract. Instruction and data address traces are widely used by computer designers for quantitative evaluations of new architectures and workload characterization, as well as by software developers for program optimization, performance tuning, and debugging. Such traces are typically very large and need to be compressed to reduce the storage, processing, and communication bandwidth requirements. However, preexisting general-purpose and trace-specific compression algorithms are designed for software implementation and are not suitable for runtime compression.

Compressing program execution traces at runtime in hardware can deliver insights into the behavior of the system under test without any negative interference with normal program execution. Traditional debugging tools, on the other hand, have to stop the program frequently to examine the state of the processor. Moreover, software developers often do not have access to the entire history of computation that led to an erroneous state. In addition, stepping through a program is a tedious task and may interact with other system components in such a way that the original errors disappear, thus preventing any useful insight. The need for unobtrusive tracing is further underscored by the development of computer systems that feature multiple processing cores on a single chip.

In this paper, we introduce a set of algorithms for compressing instruction and data address traces that can easily be implemented in an on-chip trace compression module and describe the corresponding hardware structures. The proposed algorithms are analytically and experimentally evaluated. Our results show that very small hardware structures suffice to achieve a compression ratio similar to that of a software implementation of *gzip* while being orders of magnitude faster. A hardware structure with slightly over 2 KB of state achieves a compression ratio of 125.9 for instruction address traces, whereas *gzip* achieves a compression ratio of 87.4. For data address traces, a hardware structure with 5 KB of state achieves a compression ratio of 6.1, compared to 6.8 achieved by *gzip*.

1. Introduction

Instruction and data address traces are invaluable for quantitative evaluations of new architectures as well as for workload characterization, performance tuning, testing, and debugging. Two major issues are trace collection and storage. To offer a faithful representation of the system workload or to capture program behavior in real-world conditions, traces are needed from programs that run for seconds or even minutes on real machines. Hence, trace files tend to be very large and difficult to use and distribute. To reduce their size, they are typically compressed using general-purpose compression algorithms such as Ziv-Lempel (*gzip*) [1], the Burroughs-Wheeler transformation (*bzip2*) [2], or Sequitur

[3]. Whereas these algorithms offer good compression ratios, more efficient compression is possible when the specific nature of redundancy in traces is taken into account.

Trace-specific compression techniques can be broadly classified in two groups, depending on whether they compress only instruction traces or traces including both instruction and data address information. Instruction traces can be compressed either by replacing an execution sequence by its identifier [4-7] or by exploiting control-flow graph information [8, 9]. Combined instruction and data address traces can be compressed by recording only offsets from previous trace records of the same type [4, 10], by linking data addresses to the corresponding dynamic basic blocks or loops [11-14], or by regenerating values using abstract execution [9, 15] or prediction [16, 17]. Compression of more complex trace records can exploit trace locality by storing relevant values in a cache-like structure so that a compressed trace consists of cache hit and miss information [18].

Virtually all trace compression techniques target compression in software. However, some computer systems could greatly benefit from hardware support for trace collection and compression, such as emerging systems-on-a-chip with multiple embedded RISC and DSP processor cores. They present a formidable challenge to efficient debugging and performance tuning. For instance, ARM offers a module for tracing the complete pipeline information [19]. However, the existing compression techniques that can be efficiently implemented in hardware have poor compression ratios. For example, the ARM emulator compresses traces by replacing sequences of the same record by their repetition count [20].

In this paper, we present a set of trace compression algorithms targeting on-the-fly compression of instruction and data address traces. The proposed algorithms strive to provide a good compression ratio while minimizing the required chip area for the trace compressor and the number of pins on the trace port. For the compression of instruction address traces we propose two new structures: stream caches and N-tuple history buffers. For the compression of data address traces we propose novel data address stride caches. Detailed experimental analyses based on full system simulations (i) prove the feasibility of runtime compression, (ii) show the proposed instruction address trace compressor to outperform *gzip* with minimal hardware cost, and (iii) demonstrate that the proposed data address trace compressor performs as well as *gzip* with relatively small structures. The compression ratio over all considered instruction address traces is 87.4 with *gzip* and 125.9 with a 128-entry stream cache and a 255-entry trace history buffer. The compression ratio over all considered data address traces is 6.78 with *gzip* and 6.16 with a 1024-entry data address stride cache. The total size of the stream compressor corresponds to 7629 bytes of on-chip memory.

The rest of this paper is organized as follows. Section 2 describes the architecture of the trace compressor and presents algorithms for instruction and data address trace compression. Section 3 discusses the results of the experimental analysis. Section 4 concludes the paper.

2. Instruction and Data Address Trace Compression

The proposed algorithms for instruction and data address trace compression are suitable for both software and hardware implementations. A software implementation may be used as an operating system plug-in for on-line compression or as a separate application

for compressing already generated trace files. In this paper, we focus on hardware implementations. Our goals are (i) to minimize the size of the structures to reduce the chip area required for trace compression, (ii) to provide real-time compression so that the processor is never stalled, and (iii) to achieve a good compression ratio so that the trace port requires only a few external pins.

Figure 1 shows the structure of the proposed trace compressor. The trace compressor receives instruction addresses (the program counter, PC), data addresses (DA), and task switch information from the processor core. The first level of the trace compressor encompasses an *instruction stream cache* (SC) and a *data address stride cache* (DASC).

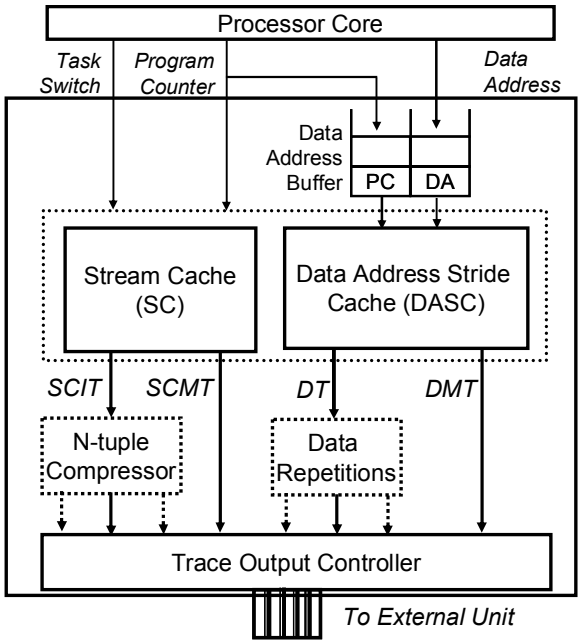

Figure 1. Trace compressor

The output from this level consists of four components: the *stream cache index trace* (SCIT), the *stream cache miss trace* (SCMT), the *data address trace* (DT), and the *data address miss trace* (DMT). Redundancy in the output traces can be further exploited with an optional second-level compressor that features *N-tuple compression* for the SCIT trace component and *data repetitions* – a simple finite state machine that compresses repetitions in the DT stream. The final streams are forwarded to a *trace output controller* that manages the output of the logical trace streams (synchronize, pack, add header) and interfaces with the external trace unit through the trace port pins akin to the ARM trace funneling [19]. Internal buffers ensure that the trace compression proceeds without stalling the processor and without dropping data.

2.1. Instruction Address Trace Compression

Instruction trace compression exploits temporal and spatial locality in instruction streams [14]. An *instruction stream* is defined as a sequential run of instructions, from the target of a taken branch to the first taken branch in the sequence. Previous studies show that most programs generate only a small number of unique instruction streams. For example, the average instruction stream length is about 12 instructions for the SPEC CPU2000 integer applications and about 117 instructions for the floating-point applications, with a maximal length of 3162 instructions and a minimal length of one instruction [14]. The starting address (SA) and length (SL) uniquely identify an instruction stream.

To compress an instruction address trace, we detect instruction streams and replace each of them with an identifier, which is similar to the SBC trace compression technique [14, 21]. Instruction streams are detected as described in Figure 4 using very simple hardware (Figure 2). SA and SL are placed in the instruction stream buffer, which is a FIFO structure that buffers possible bursts of short instruction streams. S.SA and S.SL are read from the instruction stream buffer and a stream cache lookup is performed (Figure 4). The stream cache has N_{WAY} ways and N_{SET} sets (Figure 2). A set is selected using a simple function of S.SA and S.SL, such as bit-wise XOR of selected bits and/or bit concatena-

tion. In case of a stream cache hit, the corresponding stream cache index (concatenated *iSet* and *iWay* indices) is emitted to the SCIT. In case of a cache miss, the reserved index 0 is emitted to the SCIT, and the stream descriptor (S.SA and S.SL) is emitted to the SCMT. The algorithm then deterministically selects a cache entry to be replaced, and the selected entry is updated with the stream descriptor.

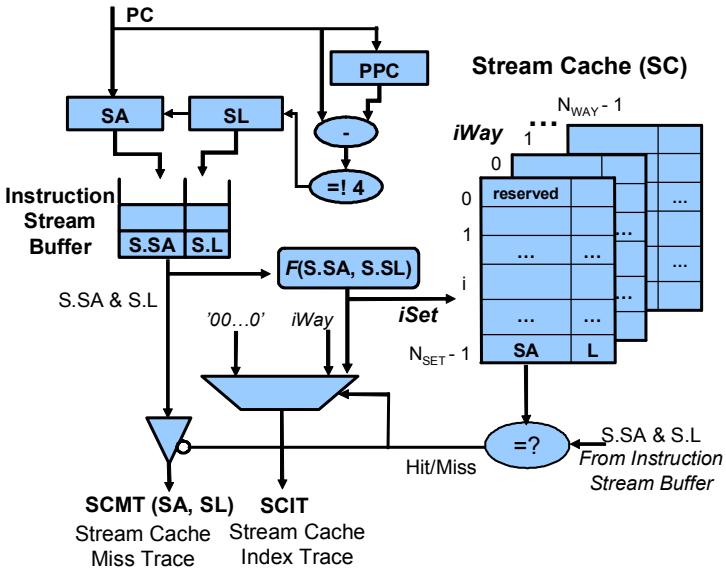

Figure 2. Stream cache

The compression ratio achieved by the stream cache compression, *CR(SC.I)*, is defined as the ratio of the raw instruction address trace (Itrace) size, calculated as the number of instructions multiplied by the address size, and the sum of the sizes of the output traces SCIT and SCMT (Eq. 1). It can be expressed analytically as a function of the average dynamic stream length (*SL.Dyn*), the stream cache hit rate (*SC.Hit*), and the stream cache size (N_{SET}*N_{WAY}) (Eq. 2). For each instruction stream, $\log_2(N_{SET}*N_{WAY})$ bits are emitted to the SCIT output. On each miss in the stream cache, 5 bytes are emitted to the SCMT output, assuming 1-byte stream lengths and 4-byte addresses.

Eq. 1
$$CR(SC.I) = \frac{Size(Itrace)}{Size(SCIT) + Size(SCMT)}$$

Eq. 2
$$CR(SC.I) = \frac{4 \cdot SL.Dyn}{0.125 \cdot \log_2(N_{SET} \cdot N_{WAYS}) + 5 \cdot (1 - SC.Hit_{N_{SET} \cdot N_{WAYS}})}$$

Typically, we see high stream cache hit rates due to the small number of unique instruction streams and the high temporal locality of the streams. Consequently, the size of the compressed trace is predominantly determined by the size of the SCIT output. The SCIT output trace is highly redundant because the majority of the runtime is spent in critical portions of the code that often encompass short sequences of instruction streams. To further exploit this redundancy with small hardware resources, we employ N-tuple compression. Figure 3 shows the structure of the N-tuple compressor, and Figure 4 details the N-tuple compression of the SCIT output trace. A se-

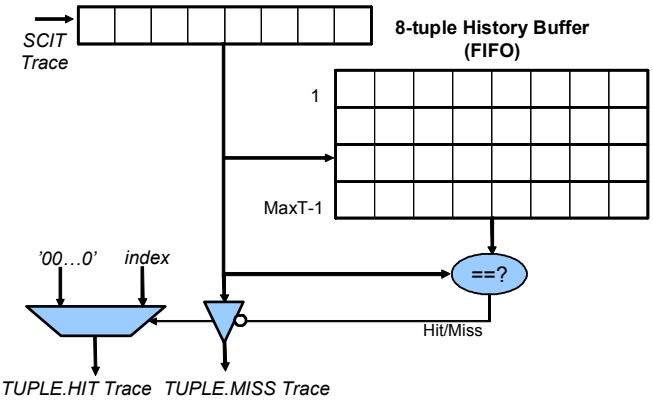

Figure 3. N-tuple compressor (N=8)

quence of *N* indices in the SCIT trace makes an *N*-tuple. The SCIT trace is replaced by Tuple.Hit and Tuple.Miss traces. We maintain a tuple history buffer (THB) of the most recent *N*-tuples. This THB is searched for a match with an incoming *N*-tuple. In case of a hit, an index in the THB is emitted to the Tuple.Hit trace. Otherwise, the whole N-tuple is emitted to the Tuple.Miss trace.

```
//   Detect a new instruction stream
1.   Get next PC;
2.   ndiff = PC - PPC; // PPC is the previous PC
3.   if (ndiff != 4 or SL == MaxStreamLength) { // a new stream is detected
4.          Place <SA, SL> into the instruction stream buffer;
5.          SL = 1;
6.          SA = PC;
7.   } else SL++;
8.   PPC = PC;
//   Compress an instruction stream
1.   Get the next stream from the instruction stream buffer (S.SA, S.SL);
2.   Perform lookup in the stream cache with iSet = F(S.SA, S.SL);
3.   if (hit)
4.          Emit <iSet, iWay> to SCIT;
5.   else {
6.          Emit reserved value <0> to SCIT;
7.          Emit stream descriptor <S.SA, S.SL> to SCMT;
8.          Select an entry (iWay) in the iSet set to be replaced;
9.          Update stream cache entry: SC[iSet][iWay].Valid = 1;
            SC[iSet][iWay].SA = S.SA; SC[iSet][iWay].SL = S.SL; }
10.  Update stream cache replacement indicators;
//   N-tuple compression
1.   Get the next index from the SCIT stream
2.   if (N-tuple incoming stream buffer is full) {
3.          Perform lookup in the Tuple History Buffer (THB);
4.          if (hit) {
5.                 Emit <index in the THB> to the Tuple.Hit trace;
6.                 // emit the first index found in the buffer
7.          } else {
8.                 Emit <0> to Tuple.Hit trace;
9.                 Emit <N-tuple> to Tuple.Miss trace; }
10.         Update the Tuple History Buffer; }
```

Figure 4. Pseudo code for stream detection, stream compression, and N-tuple compression

2.2. Data Address Trace Compression

Unlike instruction addresses, data addresses (of memory referencing instructions) rarely stay constant during program execution [22]. However, they often have a regular stride. Our proposed algorithm for runtime data address trace compression exploits temporal locality of memory referencing instructions and regularity in data address strides.

The data address trace compression utilizes a data address stride cache (DASC). The DASC is a tagless direct mapped cache-like structure, where each entry consists of two fields: a last data address (LDA) and a stride field (Figure 5). The data address trace compression algorithm is

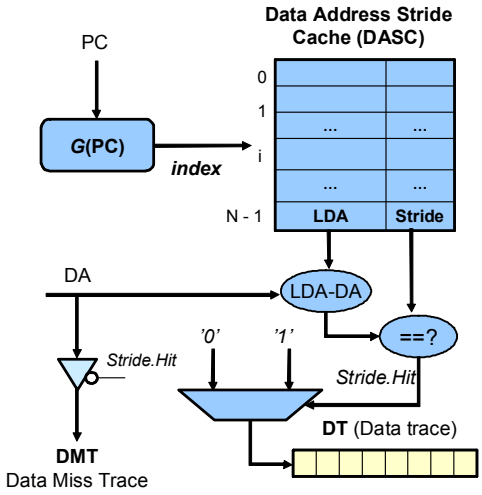

Figure 5. Data address stride cache

described in Figure 6. A memory reference descriptor, i.e., a <PC, DA> pair, is read from the data address FIFO buffer. An entry in the DASC is selected using a portion of the PC. A new stride (cStride) is calculated and compared to the Stride field read from the selected entry. If they match, only a single bit '1' is emitted to the DT output, indicating a DASC hit. Otherwise, a '0' bit is emitted to the DT and the DA field is emitted to the DMT output.

```
1.   Get the next <PC, DA> pair from the data buffers
2.   Perform lookup in the data address stride cache, index = G(PC);
3.   cStride = DA - DASC[index].LDA;
4.   if (cStride == DASC[index].Stride) {
5.          Emit <1> to DT; // 1-bit info
6.   } else {
7.          Emit <0> to DT;
8.          Emit <DA> to DMT;
9.          DASC[index].Stride = lsb(cStride); }
10.  DASC[index].LDA = DA;
```

Figure 6. Data address trace compression

The compression ratio achieved by the data address trace compression, *CR(DASC.D)*, is defined as the ratio of the raw data address trace (Dtrace) size, calculated as the number of memory referencing instructions multiplied by the address size, and the sum of the sizes of the output traces DT and DMT (Eq. 3). It can be expressed analytically as a function of the data address stride cache hit rate (Eq. 4). For each memory referencing instruction a single bit is emitted to the DT. On each miss a 4-byte address is emitted to the DMT.

Eq. 3 $$CR(DASC.D) = \frac{Size(Dtrace)}{Size(DT) + Size(DMT)}$$

Eq. 4 $$CR(DASC.D) = \frac{1}{1.03125 - DASC.StrideHit}$$

A generalized set-associative organization of DASC promises even better stride hit rates and consequently better compression ratios. However, the set-associative DASC requires address tags to be kept, which increases hardware complexity. Hence, we do not consider such DASCs in this paper. A simple state machine detects repetitions in the DT output and replaces repeating patterns with a <*pattern, number of repetitions*> pair.

3. Experimental Evaluation and Results

The goals of the experimental evaluation are (i) to assess the effectiveness of the proposed compression algorithms and (ii) to explore the feasibility of the proposed hardware implementations. We compare the compression ratio of the proposed algorithms to the compression ratio achieved by the general-purpose compression algorithms in the *gzip* (fast, default, best)

Table 1. Benchmark characteristics

	IC	NUS	max.SL	SL.Dyn
cjpeg	104,607,812	1636	239	10.89
djpeg	23,391,628	1324	206	21.81
lame	1,285,111,635	3410	252	27.81
tiff2bw	143,254,646	1058	43	12.79
tiff2rgba	151,691,275	1146	75	27.54
tiffmedian	541,260,067	1431	75	22.22
tiffdither	832,951,018	1831	51	12.57
mad	286,974,899	1659	1055	20.09
sha	140,885,982	495	62	15.15
bf_e	544,053,846	413	300	5.85
rijndael_e	319,977,971	542	254	18.94
ghostscript	708,090,638	6900	187	8.70
rsynth	824,942,227	1323	180	15.77
stringsearch	3,675,745	439	62	5.61
adpcm_c	732,513,651	347	71	54.63
gsm_d	1,299,270,245	845	401	11.07

and *bzip2* (best) software utility programs. To explore the design space of the hardware trace compressor, we extended the SimpleScalar simulator [23] to support the proposed runtime trace compression algorithms.

As workload we use complete runs of 16 MiBench programs. Table 1 shows the benchmark characteristics, including the number of instructions executed (IC), the number of unique streams (NUS), the maximum stream length (max.SL), and the average dynamic stream length (SL.Dyn). This table reveals that the number of unique instruction streams is relatively small. The average stream length ranges from 5.61 in *stringsearch* to 54.6 in *adpcm_c*.

Table 2. Stream cache hit rate and total compression ratio

SC.Hit	Ways			
Entries	*1*	*2*	*4*	*8*
8	55.47	59.67	61.06	59.54
16	67.35	71.22	74.58	73.60
32	73.99	79.51	82.45	82.82
64	80.75	88.28	91.44	93.08
128	84.62	94.27	97.26	98.33
256	85.98	97.05	99.08	99.08
CR(SC.I)	Ways			
Entries	*1*	*2*	*4*	*8*
8	16.33	17.59	16.99	15.79
16	21.10	22.15	27.81	26.61
32	23.88	28.02	34.40	33.96
64	27.54	36.89	44.12	47.07
128	28.95	47.57	54.14	57.43
256	28.05	47.81	53.60	54.24

3.1. Instruction Address Trace Compression

The compression ratio for instruction address traces depends on application characteristics (such as the average stream length and the temporal locality of the instruction streams) and the stream cache parameters. To evaluate the impact of the stream cache size and organization, we vary the number of entries from 8 to 256, and the number of ways from 1 to 8. Table 2 shows the average stream cache hit rate and the total compression ratio (the sum of the raw instruction traces for all applications divided by the sum of all compressed traces). The results indicate that even very small stream caches can achieve a good compression ratio. For example, the 16x4 (16-set and 4-way) stream cache achieves an overall compression ratio of 44.1, i.e., about 80% of the compression ratio achieved with the 32x4 stream cache, which is twice as complex. Increasing the associativity of the stream cache improves the compression ratio. Even though the 16x8 stream cache yields the best overall compression ratio of 57.4, the 32x4 represents the best price-performance tradeoff. We have tested several mapping functions and *S.SA<5+ne:6> xor S.L<ne-1:0>* performs the best, where $ne=\log_2(N_{SET}*N_{WAY})$. The chosen stream cache organization achieves a better compression ratio than *gzip* with the "fast" option on the raw instruction traces (Table 3).

N-tuple compression can further compress the SCIT trace. We consider a 32x4 stream cache and a 255-entry 8-tuple history buffer. Table 3 shows the compression ratio for the following algorithms: stream cache compression only (SC.I), combined stream cache and N-tuple compression (SC.I+Ntup), *gzip* (default, fast, best), and *bzip2* (best). The combined SC.I+Ntup outperforms *gzip* even with the "best" option, yet it can be performed in real time with small on-chip hardware structures. It only requires a bandwidth of 0.25 bits per executed instruction on the trace port.

3.2. Data Address Trace Compression

The compression ratio for data address traces depends on program behavior (the number of memory referencing instructions and their locality) and the size and organization of the DASC structure. We vary the size of the DASC from 128 to 1024 entries. Table 4 shows the compression ratios for data address trace compression for different DASC structures as well as the compression ratio achieved by *gzip* (fast, default, best) and *bzip2* (best) on

the raw data address traces. The results indicate that increasing the number of entries is beneficial. The 1024-entry DASC achieves a compression ratio of 6.12, which is higher than that of fast *gzip*, but slightly lower than that of default and best *gzip*. The tagged DASC with the same number of entries, organized as a set-associative structure with 256 sets and 4 ways, achieves a compression ratio of 6.6, which is as good as default *gzip*. This translates into a bandwidth of 0.26 bits per executed instruction on the trace port. A 256-entry DASC requires 0.4 bits/instruction.

Table 3. Compression ratio for instruction address traces

			FAST	**DEF.**	**BEST**	**BEST**
	SC.I	SC.I+Ntup	I.GZ	I.GZ	I.GZ	I.BZ2
cjpeg	47.98	147.56	54.53	109.58	124.45	341.96
djpeg	87.35	188.53	39.85	71.78	73.70	201.98
lame	100.68	158.10	128.53	60.46	333.88	87.61
tiff2bw	54.91	235.05	83.94	114.11	114.42	376.83
tiff2rgba	117.53	407.14	20.26	121.30	121.98	529.62
tiffmedian	95.91	414.37	92.32	152.81	155.47	472.93
tiffdither	43.45	65.48	46.35	91.09	99.84	170.88
mad	81.52	177.84	37.82	73.46	78.52	94.31
sha	69.24	440.35	54.42	211.43	221.75	656.53
bf_e	25.57	98.46	40.95	170.38	182.25	352.02
rijndael_e	85.17	454.63	12.56	143.82	150.62	141.77
ghostscript	26.57	50.91	39.68	100.64	111.24	212.54
rsynth	56.42	91.83	30.61	46.71	48.02	143.22
stringsearch	16.92	24.22	32.34	82.06	100.63	202.47
adpcm_c	249.71	1583.96	107.34	233.12	233.63	1862.63
gsm_d	46.79	174.57	59.22	85.37	87.17	165.58
TOTAL	54.14	125.90	47.24	87.45	112.91	171.97

Table 4. Compression ratio for data address traces

	32	*64*	*128*	*256*	*512*	*1024*	*FAST*	*DEF.*	*BEST*	*BEST*
	DASC	*DASC*	*DASC*	*DASC*	*DASC*	*DASC*	*D.GZ*	*D.GZ*	*D.GZ*	*D.BZ2*
cjpeg	3.35	4.60	5.14	5.77	6.54	7.11	4.50	5.98	6.11	18.20
djpeg	2.81	3.57	4.28	4.96	5.22	5.29	3.78	4.22	4.22	8.62
lame	1.20	1.52	2.81	3.82	4.49	4.88	4.01	6.56	6.63	8.80
tiff2bw	76.31	78.04	84.28	105.04	128.84	134.23	2.55	2.14	2.10	14.28
tiff2rgba	5.98	79.81	91.24	107.49	127.05	139.57	2.79	2.10	2.09	4.06
tiffmedian	8.64	8.70	8.74	8.81	8.87	8.89	4.37	4.40	4.53	11.16
tiffdither	2.61	6.08	7.21	8.69	9.65	10.06	4.41	4.51	4.51	7.87
mad	1.30	1.59	1.96	2.07	2.35	2.64	3.60	4.08	4.22	13.47
sha	6.58	7.94	9.38	10.79	11.36	11.36	8.36	44.91	45.61	172.71
bf_e	1.58	1.95	2.38	2.61	2.75	2.91	4.86	7.58	7.83	16.35
rijndael_e	1.10	1.10	1.10	1.13	1.29	2.06	3.22	4.24	4.27	7.31
ghostscript	1.07	1.19	1.56	2.19	2.93	5.27	18.58	27.21	27.46	47.42
rsynth	1.22	1.36	1.76	3.81	8.30	32.43	21.46	24.44	25.27	57.40
stringsearch	1.80	2.04	2.70	4.13	4.44	5.16	8.57	11.12	11.23	15.03
adpcm_c	3.13	3.13	3.13	3.13	3.13	3.13	3.64	6.57	7.15	12.27
gsm_d	2.67	4.48	11.30	13.60	14.81	16.78	18.05	21.60	23.29	63.53
TOTAL	1.66	2.04	2.80	3.77	4.67	6.12	5.51	6.78	6.90	13.29

3.3. Hardware Complexity

So far we have shown that the proposed algorithms indeed achieve a good compression ratio ensuring that a small trace port would suffice. In addition, the compressed output traces are suitable for further compression in software, which allows the design of external trace units that can capture traces over prolonged periods of time for experimental systems (the results are not shown due to page limitation).

The simple hardware structures guarantee low latency of the proposed compression. To verify that we can perform runtime compression without stalling the processor, we extended the SimpleScalar full system simulator to support our runtime compressor. In addition to verifying the feasibility of the proposed system, this simulator is used to determine the minimal necessary depth of the instruction stream buffer (Figure 2) and the data address buffer (Figure 1). We assume that the stream cache latency is 1 clock cycle for hits and 2 clock cycles for misses. The DASC latency is 2 clock cycles for both hits and misses. The modeled processor corresponds to the XScale processor. The results indicate that the instruction stream buffer needs only 2 entries, while the data address buffer needs 8 entries.

Table 5 provides an estimate of the hardware complexity of the proposed structures. The overall size corresponds to 7629 bytes, which is several times smaller than L1 processor caches, giving further evidence that the structures can operate at CPU clock frequencies.

Table 5. Hardware complexity estimation

Component	*Entries*	*Complexity*	*Bytes*
Instruction stream buffer	2	2x5	10
Stream detector	2	2x4	8
Stream cache	32x4	128x5	640
N-tuple history buffer	255	255x8*(7/8)	1785
Data address buffer	8	8x8	64
Data address stride cache	1024	1024x5	5120
Data repetitions state machine	-	2	2

4. Conclusion

This paper presents a set of algorithms for runtime compression of instruction and data address traces. Based on these algorithms we propose an on-chip hardware compressor capable of unobtrusive real-time instruction and data address trace compression. It achieves excellent compression ratios, comparable to general-purpose compression in software, at minimal hardware complexity.

References

[1] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transaction on Information Theory*, vol. 23, May 1977, pp. 337-343.

[2] M. Burrows and D. J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," Digital SRC, Report 124, May 10, 1994.

[3] C. G. Nevill-Manning and I. H. Witten, "Linear-Time, Incremental Hierarchy Interference for Compression," in *Proceedings of the IEEE Data Compression Conference*, Snowbird, UT, 1997, pp. 3-11.

[4] E. E. Johnson, J. Ha, and M. B. Zaidi, "Lossless Trace Compression," *IEEE Transactions on Computers*, vol. 50, February 2001, pp. 158-173.

[5] J. R. Larus, "Whole Program Paths," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999, pp. 259-269.

[6] A. Milenkovic, M. Milenkovic, and J. Kulick, "N-Tuple Compression: A Novel Method for Compression of Branch Instruction Traces," in *Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems*, Reno, NV, 2003, pp. 49-55.

[7] Y. Zhang and R. Gupta, "Timestamped Whole Program Path Representation and Its Applications," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, 2001, pp. 180-190.

[8] A. Hamou-Lhadj and T. C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces," in *Proceedings of the 10th International Workshop on Program Comprehension*, Paris, France, 2002, pp. 159 -168.

[9] J. R. Larus, "Efficient Program Tracing," *IEEE Computer*, vol. 26, May 1993, pp. 52-61.

[10] A. D. Samples, "Mache: No-Loss Trace Compaction," in *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Oakland, CA, 1989, pp. 89 - 97.

[11] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia, "Sigma: A Simulator Infrastructure to Guide Memory Analysis," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, 2002, pp. 1-13.

[12] E. N. Elnozahy, "Address Trace Compression through Loop Detection and Reduction," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, June 1999, pp. 214-215.

[13] A. R. Pleszkun, "Techniques for Compressing Program Address Traces," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, 1994, pp. 32-39.

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

[15] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," in *Proceedings of the 1990 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, 1990, pp. 37 - 47.

[16] M. Burtscher, "VPC3: A Fast and Effective Trace-Compression Algorithm," in *Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2004, pp. 167-176.

[17] M. Burtscher and N. B. Sam, "Automatic Generation of High-Performance Trace Compressors," in *2005 International Symposium on Code Generation and Optimization*, San Jose, CA, USA, 2005, pp. 229-240.

[18] Y. Luo and L. K. John, "Locality-Based Online Trace Compression," *IEEE Transaction on Computers*, vol. 53, June 2004, pp. 723-731.

[19] "Coresight On-Chip Debug and Trace Technology," <http://www.arm.com/products/solutions/CoreSight.html> (Available July 2004).

[20] D. I. McCullough and L. A. Traylor, "Trace Reporting Method and System," United States Patent 6,615,371, American Arium, 2003.

[21] A. Milenkovic and M. Milenkovic, "Stream-Based Trace Compression," *Computer Architecture Letters*, vol. 2, September 2003.

[22] A. Milenkovic and M. Milenkovic, "An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams," *ACM Transactions on Modeling and Computer Simulation* January 2007.

[23] D. Burger and T. Austin, "The Simplescalar Tool Set Version 2.0," University of Wisconsin, Technical Report CS-TR-97-1342, 1997.