Exploiting Streams in Instruction and Data Address Trace Compression

Aleksandar Milenković, Milena Milenković
Electrical and Computer Engineering Dept., The University of Alabama in Huntsville
Email: {milenka | milenkm}@ece.uah.edu

Abstract

Novel research ideas in computer architecture are frequently evaluated using trace-driven simulation. The large size of traces incited different techniques for trace reduction. These techniques often combine standard compression algorithms with trace-specific solutions, taking into account the tradeoff between reduction in the trace size and simulation slowdown due to decompression. This paper introduces SBC, a new algorithm for instruction and data address trace compression based on instruction streams. The proposed technique significantly reduces trace size and simulation time, and can be successfully combined with general compression algorithms. The SBC technique combined with gzip reduces the size of SPEC CPU2000 traces 59-97930 times, and combined with Sequitur 65-185599 times.

1. Introduction

Trace-driven simulation has long been used in both processor and memory studies. Traces can accurately represent a system workload, and in the last decade there has been a lot of research efforts dedicated to trace issues, such as trace collection, reduction and processing [1]. In order to offer a faithful representation of a specific workload, traces are very large, encompassing billions of memory references and/or instructions. For example, an instruction trace with 1 billion instructions, where each trace record takes 10 bytes, requires almost 10 Gbytes of storage space. Yet, with a modern superscalar processor executing 1.5 instructions each clock cycle on average, and running at 3 GHz, it represents only 0.2 seconds of the simulated CPU execution time. To efficiently store and use even a small collection of traces, trace sizes must be reduced as much as possible. Although traditional compression techniques such as the Ziv-Lempel algorithm [2], used in the gzip utility, offer a good compression ratio, even better compression is possible when the specific nature of redundancy in traces is taken into account. On the other hand, since the ultimate purpose of traces is to be used in simulations, a trace compression should not introduce a significant decompression slowdown. An effective trace compression technique is loss-less, i.e., not introducing errors into the simulation, has high compression factor, short decompression time, and relatively short compression time.

Depending on the simulated system, a trace can contain different types of information. For example, control flow analysis needs only a trace of executed basic blocks or paths. Cache studies require address traces, and more complex processor simulations need instruction words as well. Branch predictors can be evaluated using traces with only branch-relevant information, such as branch and target addresses, and branch outcome, and ALU unit simulations require operand values. For example, the Dinero trace format record consists of the address of memory reference and the reference type – read, write, or instruction fetch [3], and BYU traces also include additional information, such as the size of the data transfer, processor ID, etc. [4]. In addition to addresses, the IBS trace format includes operation code and whether the instruction was executed in user or kernel mode [5].

Various trace compression techniques have been introduced, focusing on different trace information. One set of compression techniques, such as whole program path (WPP) [6] and timestamped WPP [7], relies on program instrumentation and concentrates on instruction traces only. In WPP, a trace of acyclic paths is compressed using a modified Sequitur algorithm [8]. In timestamped WPP, all path traces for one function are stored in one block, thus enabling fast access to function-related information.

Another set of compression techniques targets full address traces that include both instruction and data. Unlike instruction address or path traces, data address traces rarely have repeatable patterns and hence are more difficult to compress, although one memory referencing instruction may access addresses with a constant stride. One approach, applied in a one-pass algorithm called PDATS (Packed Differential Address

and Time Stamp), is to store address differences between successive references of the same type (load, store, instruction fetch) [9]. In PDATS, stored address differences can have variable length and an optional repetition count in cases when a constant difference is present in consequtive addresses of the same type. Another approach is to link information about the data addresses with a corresponding loop, but this requires previous control flow analysis to extract loop information and cannot be done in one pass [10]. The SIGMA trace compression algorithm takes advantage of nested loops [11]. However, this algorithm is based on the program instrumentation, and has some limitations, such as a constant iteration count for inner loops. A rather original approach regenerates original trace using a set of value predictors [12], but it has a relatively long decompression time.

Some techniques, such as PDI, compress combined address and instruction traces, i.e., traces consisting of instruction addresses + instruction words, and data addresses. In PDI, instruction words are compressed using a dictionary-based approach – each of the 256 most frequently used instruction words in the trace is replaced with its dictionary index while other words are left unchanged. Addresses are compressed as in PDATS, but without a repetition count. This algorithm can be one pass or two pass, depending on using a generic or a trace-specific instruction word dictionary. Specialized branch traces can be compressed by replacing an N-tuple of branch instruction trace records with its ID from the N-Tuple Record Table [13].

This paper proposes a new method for single-pass compression of combined address and instruction traces, Stream-Based Compression (SBC). The SBC algorithm relies on extracting instruction streams. An instruction stream is a sequential run of instructions, from the target of a taken branch to the first taken branch in sequence. A stream table keeps relevant information about streams: starting address, stream length, instruction words and their types. All instructions from a stream are replaced by its index in the stream table, creating a trace of instruction streams. Information about data addresses such as the data address stride and the number of stride repetitions is attached to the corresponding instruction stream and stored separately.

The proposed algorithm achieves a very good compression ratio and decompression time for both instruction and data address traces, yet it is simple to implement and does not require code augmentation nor lengthy several-passes control flow analysis. Furthermore, SBC can be successfully combined with general compression algorithms, such as Ziv-Lempel or Sequitur. We evaluated SBC on Dinero+ traces [9] of SPEC CPU2000 benchmark programs [14]. When combined with gzip, SBC reduces the trace size 78-97930 times, depending on the benchmark, and outperforms mPDI-gzip, gzipped combination of PDI and PDATS, 4-1231 times. SBC combined with Sequitur reduces the trace size even further, but at the price of a considerable decompression slowdown.

The rest of the paper is organized into four sections. The second section introduces the formats of traces and explains stream-based compression. The third section shows the compression ratio and decompression time for the compared compression techniques. The last section gives concluding remarks.

2. Stream-based compression

The SBC algorithm exploits several inherent characteristics of program execution traces. Instruction traces consist of a fairly limited number of different instruction streams, and most of the memory references exhibit strong spatial and/or temporal locality, for example, a load having a constant address stride across loop iterations. The stream-based compression of the combined address and instruction traces results in three files: Stream Table File (STF), Stream-Based Instruction Trace (SBIT), and Stream-Based Data Trace (SBDT).

In this paper SBC is demonstrated on Dinero+ traces, although it is applicable to any combined trace. A Dinero+ trace record has fixed length fields: the header field (0 – data read, 1 – data write, and 2 – instruction read), the address field, and the instruction word field for the instruction read type.

First we describe the decompression process, for the example in Figure 1– a short trace of a loop, where stream 1 is followed by 28 executions of stream 2, and one execution of stream 3. At the beginning of the trace decompression, the whole Stream Table File is loaded into a corresponding Stream Table structure, resident in the memory during decompression. One record in STF consists of a stream start address and a stream length -- i.e., the number of instructions in the stream -- followed by instruction words and their types -- load (0), store (1) or an instruction that does not access memory (2).

In addition to the pointer to the list of stream instruction words, each entry in the Stream Table structure in memory has a pointer to the list of stream data address references. One node in the stream data address list has the following fields: current data address, address stride, and repetition count. All fields are initialized to zero. This list is dynamically updated from the Stream-Based Data Trace during trace decompression, whenever the repetition count of an accessed node is 0.

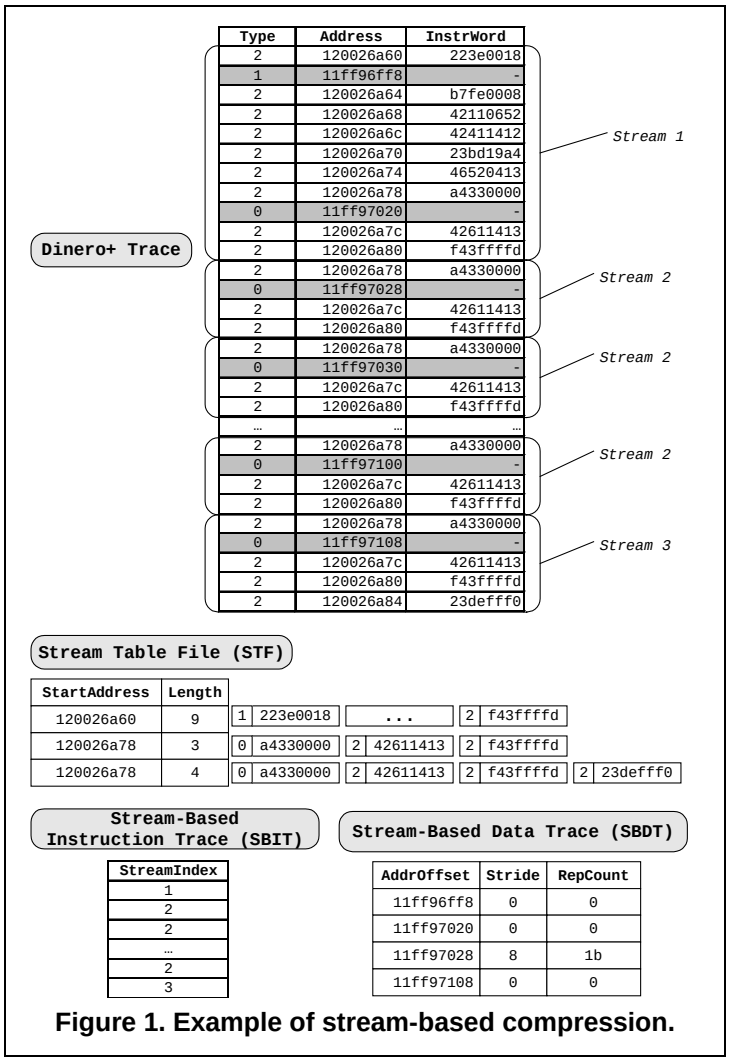

Figure 1. Example of stream-based compression.

Decompression proceeds as follows: a stream index is read from the Stream-Based Instruction Trace, i.e., stream index 1. Stream Table entry 1 is accessed, giving address, word and type of the first instruction in the stream. This instruction is a store (type 1), so the corresponding store address is needed. Since the repetition count for the first data reference in this stream is 0 after initialization, the decompression algorithm reads a record from the SBDT, consisting of the following fields: DataHeader, AddrOffset, Stride, and RepCount (Figure 2). The current data address in the node is calculated as the current address (0) plus the AddrOffset field, the stride is set to the value of the Stride field, in this case 0, and the repetition count is set to the RepCount value, again 0 since this stream executes only once. The pointer to the current instruction then moves along the stream instruction word list until all nine instructions are read. Each instruction address is obtained by incrementing the current instruction address for the instruction length, starting from the StartAddress. The SBDT is accessed once more, for the seventh instruction, which is a load. The next stream index in the SBIT is 2, so entry 2 is accessed. The first instruction is a load, so the corresponding node in the data address list is updated from SBDT; i.e., the current address is set to 0x11ff97028, the stride is set to 8, and the repetition count to 27. When the stream 2 is again encountered in the SBIT and its load instruction is read from the Stream Table, there is no need to access the SBDT – the load address is calculated as the previous address plus the stride, and the repetition count is decremented for all further 27 executions of the stream 2.

As can be seen on this simple example, the SBC algorithm handles instruction and data information separately. The SBIT is obtained by replacing each instruction stream by its index from the stream table. Since the stream table includes all streams and not just the most frequent ones, this is a one-pass algorithm – when the end of a stream is detected in the original input trace, SBC finds the corresponding stream in the table or, if necessary, adds a new entry to the table, and outputs the stream index to the SBIT. When compressing data addresses, SBC exploits frequent regularity of memory references produced by consecutive instances of the same load/store instruction. Ideally, during decompression one memory-accessing instruction should get new values from the SBDT only when its offset stride changes. However, we want to keep the compression algorithm one-pass, so the compression program keeps relevant values in a finite FIFO buffer. Clearly, the larger FIFO buffer will "catch" more data repetitions, thus increasing the compression ratio. Each entry in the FIFO buffer has a ready flag that is set at the change of the offset stride. The records are written to the SBDT when there is a sequence of ready records at the front of the FIFO buffer, or when the FIFO is full. The AddrOffset field records the offset from the last occurrence of a particular memory-accessing instruction and is equal to the memory address when such instruction occurs for the first time. The DataHeader field codes the length and the most frequent values of other fields in the Data Trace (Figure 2), thus achieving additional compression. The repetition count values 0 and 1, and the stride values 0, 1, 4 and 8 can be encoded in the DataHeader, and the proposed format allows variable length of AddrOffset

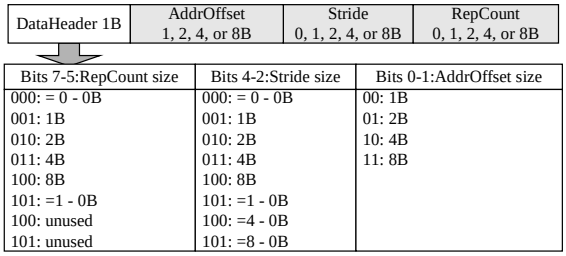

Figure 2. SBC data trace format.

(1, 2, 4 or 8 bytes), Stride and RepCount fields (0, 1, 2, 4, or 8 bytes).

One can ask why the SBC algorithm does not exploit the repetition of instruction streams. Such patterns in the SBIT are easily recognized by gzip or Sequitur, without an increase in complexity of SBC or any restrictions considering the number and the nature of nested loops.

3. Results

In order to evaluate the proposed compression mechanism, for each SPEC CPU2000 benchmark we traced two segments for reference data inputs: the first two billion instructions (F2B), and the two billion instructions after skipping 50 billion (M2B), thus making sure that the results do not overemphasize the program initialization. While the number of instructions is fixed, the number of memory-referencing instructions – loads and stores – varies greatly across benchmarks, from as low as 18.7% of all instructions for F2B segments of *lucas* and *fma3d*, to

Table 1. Statistics for SPEC2000 INT traces.

	Load+Store%		Dinero+ [GB]		Stream Number		AvrStreamLen		MaxStreamLen	
CINT	**F2B**	**M2B**	**F2B**	**M2B**	**F2B**	**M2B**	**F2B**	**M2B**	**F2B**	**M2B**
164.gzip	33.17	32.07	29.16	28.99	751	336	13.9	13.8	229	229
176.gcc	50.94	52.10	31.80	31.98	25416	22222	11.8	10.7	272	254
181.mcf	41.36	37.98	30.38	29.87	744	308	8.9	6.0	88	64
186.crafty	37.74	36.71	29.84	29.68	4122	1892	13.1	13.4	191	100
197.parser	37.94	35.06	29.87	29.44	4767	4200	9.4	9.9	157	157
252.eon	48.59	48.58	31.45	31.45	3486	588	13.8	14.1	169	168
253.perlbmk	45.02	46.88	30.92	31.20	9034	6344	10.1	12.0	84	868
254.gap	37.36	38.36	29.78	29.93	3218	476	24.3	10.3	284	75
255.vortex	44.40	38.95	30.83	30.02	5496	2644	11.1	11.2	126	110
300.twolf	33.77	33.00	29.25	29.13	2399	1014	12.3	14.5	163	185
Average	41.03	39.97	30.33	30.17	5943.3	4002.4	12.9	11.6	176.3	221.0

Table 2. Statistics for SPEC2000 FP traces.

	Load+Store%		Dinero+ [GB]		Stream Number		AvrStreamLen		MaxStreamLen	
CFP	**F2B**	**M2B**	**F2B**	**M2B**	**F2B**	**M2B**	**F2B**	**M2B**	**F2B**	**M2B**
168.wupwise	19.76	30.96	27.16	28.83	1563	234	23.9	27.5	229	229
171.swim	31.02	32.86	28.84	29.11	1582	496	93.6	132.3	707	707
172.mgrid	36.66	36.43	29.68	29.64	1457	875	240.1	159.6	1944	1944
173.applu	37.75	38.20	29.84	29.91	1470	506	411.5	448.9	3162	3162
177.mesa	37.53	38.09	29.81	29.89	1637	593	14.8	18.5	550	266
178.galgel	41.80	41.27	30.44	30.36	1818	81	18.4	23.0	264	206
179.art	37.81	34.12	29.85	29.30	435	341	10.3	8.7	168	561
183.equake	36.00	45.04	29.58	30.93	517	260	8.6	28.3	44	623
188.ammp	31.13	37.23	28.85	29.76	955	502	12.5	35.2	168	561
189.lucas	18.73	22.20	27.01	27.52	964	317	27.1	127.9	427	427
191.fma3d	18.71	45.70	27.00	31.02	2083	841	10.7	43.6	383	1158
200.sixtrack	32.09	24.69	29.00	27.89	3532	82	20.1	192.9	264	580
301.appsi	37.24	37.29	29.76	29.77	2439	389	34.0	51.5	729	729
Average	32.02	35.70	28.99	29.53	1573.2	424.4	71.2	99.8	695.3	857.9

over 50% for both *gcc* segments (Table 1 and Table 2). It is interesting to notice that the percentage of loads and stores does not significantly change between the two segments for all integer benchmarks (CINT) except *vortex*, and changes for more than 15% for a half of floating point benchmarks (CFP). The trace size is on average 30MB per segment, and the sum of uncompressed traces reaches 1366 GB, clearly showing the need for highly efficient compression techniques. Traces are generated using modified SimpleScalar environment [15], and precompiled Alpha SPEC2000 binaries.

Since the Stream Table is held in the memory during decompression, we first verified the assumption about the relatively limited number of different instruction streams, i.e., the Stream Table entries. Analysis of SPEC CPU2000 traces shows that all CFP and most CINT benchmarks have fewer than 5000 different instruction streams. The only benchmarks with larger number of unique streams are *171.gcc*, *253.perlbmk*, and the F2B segment of *255.vortex*. For all benchmarks the F2B segment has more streams than the M2B. The average stream length is less than 30 for all CINT and five CFP benchmarks, and it reaches the maximum for *173.applu*, 449 instructions in the M2B segment, but in that segment *applu* has only 506 unique streams. All these results indicate that a Stream Table has relatively modest memory requirements.

The compression ratio of SBC is compared to the compression obtained by gzip and by mPDI, a modified version of PDI. We merged PDI and PDATS into mPDI in order to compare SBC with the best possible combination of those techniques. The mPDI uses the PDATS algorithm for data, so it can benefit from address stride repetitions. The implemented PDATS algorithm does not make a difference between load and store references, as Johnson suggested in his later work [16]. Finally, in mPDI data and instruction references are separated into two files, making the regular patterns even more recognizable by gzip. The basis for comparison was the size of uncompressed Dinero+ traces, separated to instruction address + instruction word and data address components. The size of this format is slightly less than the size of the unified Dinero+, since the header field is not required in the data address trace. In addition to that, split components can be better compressed by gzip.

Table 3, Table 4, Table 5, and Table 6 show the compression ratio of compared algorithms, for CINT and CFP benchmarks and two considered program segments. The FIFO buffer size for SBC is 4000 entries. The SBC algorithm reduces the trace size for up to 60.5 times for CINT and 495.2 times for CFP benchmarks, outperforming gzip compression. On average, the SBC compression ratio is higher for CFP than for CINT applications, due to longer instruction streams and higher repetition counts for data references. As seen in Table 1 and Table 2, the average CFP instruction stream length is roughly an order of magnitude greater than in CINT benchmarks. One indicator of the successful SBC compression of data addresses is the ratio between the number of data addresses in original trace and the number of records in Stream-Based Data Trace file. These values are also significantly higher for CFP traces: on average 56.1 for F2B segment and 95.1 for M2B segment, while corresponding CINT values are 6.3 and 5.8. The compression ratio in two different trace segments depends on the application: it is on average higher in the "middle" of the execution than at the beginning, but not for all benchmarks.

Table 3. Compression ratio for SPEC2000 INT – F2B.

CINT: F2B	mPDI	SBC	Din.gz	mPDI.gz	SBC.gz	SBC.seq
164.gzip	4.3	60.2	39.7	46.9	210.1	193.4
176.gcc	3.2	30.9	9.4	19.4	168.8	193.0
181.mcf	3.4	46.5	24.3	55.5	500.6	597.2
186.crafty	3.0	40.0	7.0	22.2	228.3	247.8
197.parser	3.6	33.6	27.6	32.3	182.9	347.9
252.eon	3.4	21.9	6.1	26.6	396.9	775.3
253.perlbmk	3.1	30.6	5.9	16.4	340.2	318.5
254.gap	3.9	49.9	12.9	35.5	765.5	868.4
255.vortex	3.4	20.9	6.8	14.3	115.2	332.0
300.twolf	3.3	28.2	7.4	23.4	105.7	88.3
Average	3.46	36.26	14.70	29.25	301.41	396.17

Table 4. Compression ratio for SPEC2000 INT – M2B.

CINT: M2B	mPDI	SBC	Din.gz	mPDI.gz	SBC.gz	SBC.seq
164.gzip	3.7	60.5	41.6	48.2	217.8	200.3
176.gcc	3.0	40.3	14.7	20.9	260.4	291.1
181.mcf	2.3	16.2	20.9	20.3	58.5	82.8
186.crafty	2.9	44.0	7.0	25.0	257.2	278.8
197.parser	3.4	33.0	28.1	32.7	167.0	333.5
252.eon	3.4	21.4	6.0	28.1	384.5	753.2
253.perlbmk	2.9	42.0	34.8	46.9	735.0	1101.8
254.gap	2.9	34.9	33.6	38.3	1115.4	1912.0
255.vortex	3.3	26.7	11.8	24.8	228.6	402.1
300.twolf	3.2	24.4	6.4	19.4	78.3	64.9
Average	3.10	34.36	20.49	30.45	350.28	542.06

The SBC compression ratio is significantly higher than mPDI, due to the combined effect of replacing whole instruction streams with stream IDs instead of replacing instructions with dictionary indexes, and exploiting stride repetitions between the subsequent executions of the same load/store instruction instead of subsequent load/store instructions. The SBC-compressed CINT trace files are 6.1-16.2 times smaller then when compressed with mPDI, and CFP trace files are 11.2-180.4 times smaller. Notice that SBC also outperforms combination mPDI-gzip for most benchmarks.

Even better compression ratios can be obtained by further compressing an SBC trace by gzip. The combined SBC-gzip compression ratio goes up to 1115.4 for CINT (*254.gap* in M2B segment) and up to 97930.4 (*171.swim* in M2B segment) for CFP, with corresponding average values greater than 300 for CINT and greater than 95000 for CFP. Translated back to bytes, this means that instead of 30GB for an uncompressed trace or about 2GB for a gzip-only compressed trace, a combined SBC-gzip compressed trace occupies less than 200MB for all benchmarks except *300.twolf* in both segments, *255.vortex* in F2B and *176.gcc* in M2B, and the trace size is even less than 1MB for some CFP benchmarks, *171.swim* and *189.lucas*. Compared to combined mPDI-gzip, SBC-gzip achieves on average 10.8 and 10.7 times better compression in CINT F2B and M2B segments, and 153.6 and 301 times better for corresponding CFP segments, sometimes outperforming mPDI-gzip for more than two orders of magnitude.

The SBC technique can also be combined with other compression techniques, such as Sequitur, which proved to be a highly suitable technique for instruction address traces, and consequently, for SBIT files. As it could be expected, Sequitur cannot be as efficient for data address traces, nor for SBDT files. Overall, combined SBC-Sequitur has a better compression ratio than SBC-gzip, but at the price of significantly increased decompression time.

Figure 3 shows the decompression speedup relative to gzipped Dinero+ traces. Since traces are used during simulation, we measured decompression time in a program that reconstructs an entire trace, and uses pipes for gzip and Sequitur. Decompression speedup for SBC-gzip is proportional to the compression ratio, i.e., smaller files are faster to decompress. On average, SBC.gz traces are decompressed about 20 times faster than Dinero.gz for floating-point, and about 10 times for integer benchmarks. The decompression of traces compressed with SBC-Sequitur is slower than decompression of SBC.gz traces: 4.2-11.8 times for CINT, and 2.2-8.6 times for CFP benchmarks. For some benchmarks the SBC-Sequitur traces even have longer decompression times than gzipped Dinero+: for *300.twolf* in both segments and *181.mcf* in M2B.

The results presented do not include traces compressed with Sequitur only. While Sequitur on its own is very efficient for instruction address traces, it does not perform well on data traces, producing larger compressed files than gzip for most benchmarks.

Table 5. Compression ratio for SPEC2000 FP – F2B

CFP: F2B	mPDI	SBC	Din.gz	mPDI.gz	SBC.gz	SBC.seq
168.wupwise	3.9	78.0	33.8	98.3	2840.4	4746.9
171.swim	3.0	402.7	23.9	176.1	43083.6	42668.0
172.mgrid	2.8	73.3	11.9	37.5	8774.4	15962.3
173.applu	2.8	64.8	12.7	22.5	2646.3	30648.5
177.mesa	2.9	73.0	10.1	55.6	1210.3	1734.9
178.galgel	3.4	97.4	20.6	28.3	11534.7	43124.5
179.art	4.1	79.0	23.6	29.9	12315.9	24224.7
183.equake	3.7	53.2	30.0	149.8	1887.0	3278.8
188.ammp	4.5	78.0	24.4	48.2	2573.1	3501.5
189.lucas	3.6	149.8	68.7	179.7	30783.6	77058.6
191.fma3d	4.2	47.4	12.5	23.4	3571.6	17376.7
200.sixtrack	3.1	67.1	19.6	49.7	1265.9	1911.7
301.appsi	3.0	34.4	8.4	19.6	2242.8	11063.0
Average	3.5	99.8	23.1	70.7	9594.6	21330.8

Table 6. Compression ratio for SPEC2000 FP – M2B

CFP: M2B	mPDI	SBC	Din.gz	mPDI.gz	SBC.gz	SBC.seq
168.wupwise	2.7	42.1	17.6	37.2	2007.3	3668.4
171.swim	2.7	495.2	20.5	152.5	97930.4	185599.2
172.mgrid	2.9	75.2	12.3	37.7	9368.1	17133.0
173.applu	2.8	75.9	13.9	24.3	3441.8	44464.9
177.mesa	2.8	81.6	10.5	49.8	1056.4	1473.1
178.galgel	2.4	54.5	27.2	37.6	9188.9	74833.7
179.art	2.9	67.1	25.6	35.9	20452.0	92720.6
183.equake	2.4	33.9	26.5	26.3	364.6	425.3
188.ammp	2.5	40.8	22.2	27.9	434.9	432.7
189.lucas	2.5	266.4	37.3	76.1	28898.6	57234.9
191.fma3d	2.6	108.7	4.8	9.5	11667.4	33310.4
200.sixtrack	2.6	128.7	13.3	32.0	7312.5	15313.6
301.appsi	2.8	34.0	7.9	18.2	2238.6	13214.7
Average	2.7	115.7	18.4	43.5	14950.9	41525.0

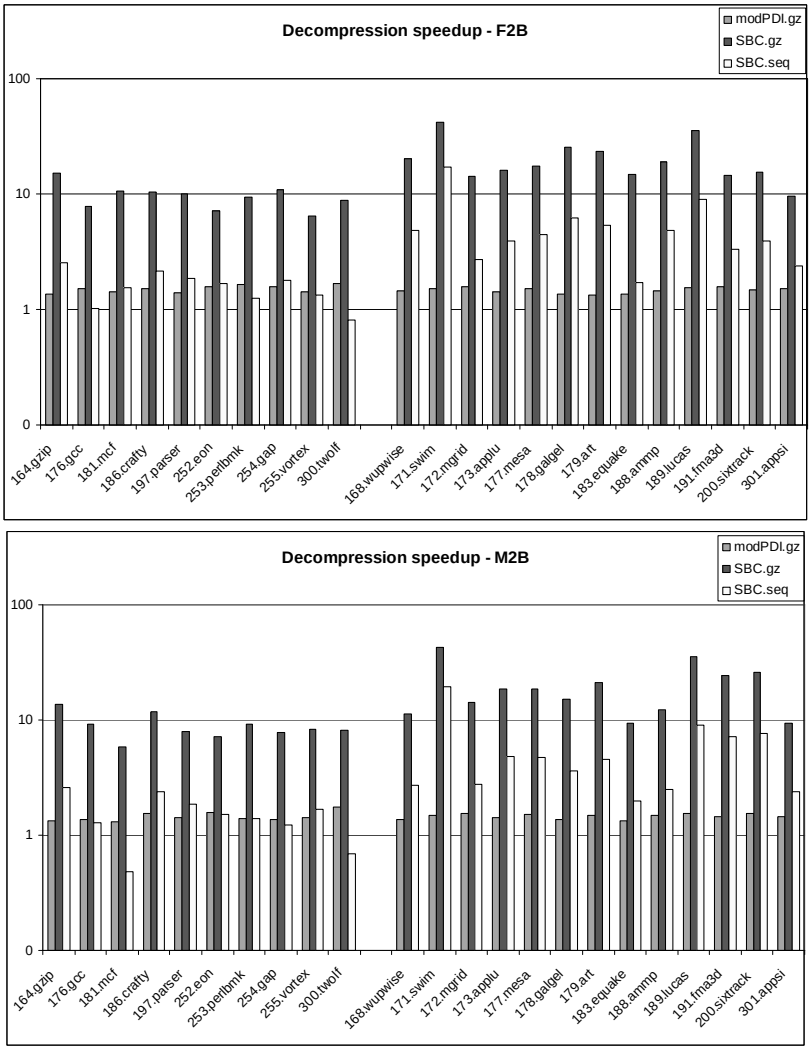

Figure 3. Decompression speedup relative to gzipped Dinero+ traces

Another interesting question is the general compressibility of instruction address + instruction word trace component, versus data address component. Figure 4 and Figure 5 show the compression ratio separately for different trace components in F2B segment, for mPDI-gzip, SBC-gzip, SBC-seq combinations, and gzip only compression. With both SBC-gzip and SBC-seq combination, the instruction component is compressed much better. The gzip utility alone compresses data better than instruction addresses in *186.crafty*, *252.eon*, *253.perlbmk*, *255.vortex*, *300.twolf*, *173.applu*, *183.equake*, and *191.fma3d*. Data address component compression with mPDI-gzip outperforms instruction component compression in two benchmarks, *197.parser* and *301.appsi*.

It should also be noted that for gzipped SBC traces, on average, SBIT.gz makes only 5% of total compressed trace for CINT, and 10% for CFP. Therefore, further improvement in the instruction trace compression would not significantly increase the overall compression ratio. This indicates that further research efforts should be aimed to improve only the data address compression.

A good indicator of the data address component compression ratio is the number of memory references in the trace divided by the number of records in SBDT file, N_{MEM}/N_{SBDT}. However, compression also depends on the length of repetition, stride, and address offset fields. For example, *176.gcc* and *300.twolf* in F2B segment have the similar N_{MEM}/N_{SBDT} ratio, 4.6 for *176.gcc* and 4.5 for *300.twolf*. The compression ratio is 10.7 for *176.gcc* and 6.9 for *300.twolf*, due to the different length of record fields: 29% of records in *300.twolf* SBDT file need more than two bytes for the

address offset, versus only 4.7% for *176.gcc*. Moreover, 53% *300.twolf* records need more than one byte for the stride, while the corresponding value for *176.gcc* is 22.2%. Both traces have very similar percentages for different lengths of the repetition count field. This example illustrates that the compression ratio is a complex function of N_{MEM}/N_{SBDT} and fields length, and it is even more difficult to estimate the compression gain if the SBC algorithm is combined with some other compression method.

Possible extensions of the SBC algorithm may include another trace information, such as the value of operands. The Stream Table memory requirements can be reduced, for example by using a two-level scheme where each entry in the Stream Table keeps the indexes of the basic blocks from the Basic Block Table.

4. Conclusion

The SBC algorithm offers a new technique for compressing combined instruction and address traces: data address information is linked to a corresponding instruction stream, and instruction stream is replaced by its index from the Stream Table. SBC compresses data without any loss of information, in a single-pass, significantly reduces the trace size and the time needed to read the trace during simulations, and can be successfully combined with other general compression techniques. The choice of the additional compression scheme depends on the end user requirements. The SBC-gzip combination has a very good compression ratio and a short decompression time, and SBC-Sequitur compresses even better, but with a slower decompression. As a single-pass algorithm, SBC can be easily modified for online tracing in real-time, and implemented in hardware.

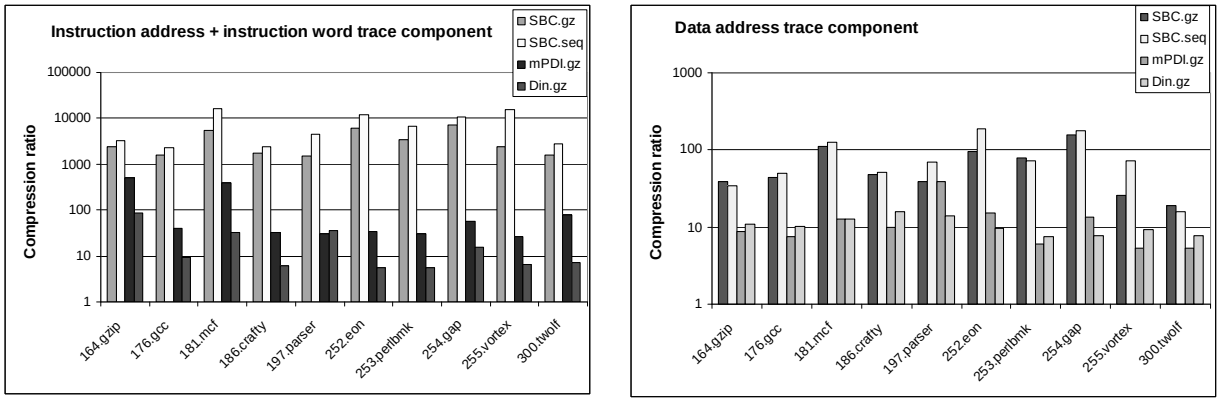

Figure 4 CINT F2B: Compression ratio relative to Dinero+, for instruction/data trace components

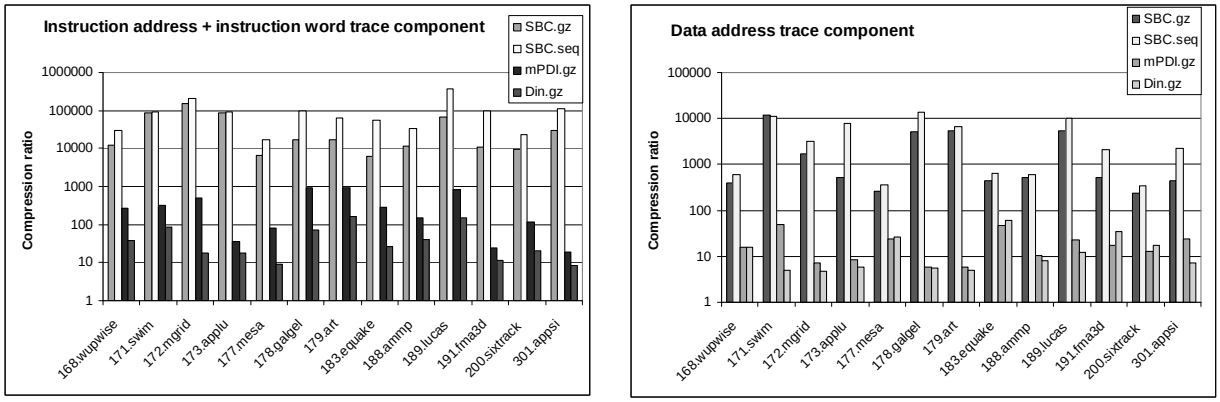

Figure 5 CFP F2B: Compression ratio relative to Dinero+, for instruction/data trace components

5. References

[1] R. Uhlig, T. Mudge, “Trace-driven memory simulation,” ACM Computing Surveys, Vol. 29, No. 2, June 1997.

[2] L. Ziv, A. Lempel, “A universal algorithm for sequential data compression,” IEEE Transaction on Information Theory, Vol. 23, No 3., 1977.

[3] J. Edler, M.D. Hill, Dinero IV Trace-Driven Uniprocessor Cache Simulator, http://www.cs.wisc.edu/~markhill/DineroIV/

[4] N.C. Thornock, J.K. Flanagan, “A national trace collection and distribution resource,” ACM SIGARCH Computer Architecture News, Vol. 29, No. 3, June 2001.

[5] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, J. Emer, “Instruction fetching: coping with code bloat,” Proceedings of the 22nd Annual Internationa Symposium on Computer Architecture, June 1995.

[6] J. Larus, “Whole Program Paths,” in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Atlanta, GA, 1999, pp. 259-269.

[7] Y. Zhang, R. Gupta, “Timestamped Whole Program Path Representation and Its Applications,” in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Snowbird, Utah, 2001, pp. 180-190.

[8] C. G. Nevill-Manning, I. H. Witten, “Linear-Time, Incremental Hierarchy Interference for Compression,” in Proc. IEEE Data Compression Conference, 1997, pp. 3-11.

[9] E. E. Johnson, J. Ha, M. B. Zaidi, “Lossless Trace Compression,” IEEE Transactions on Computers, Vol. 50, No. 2, February 2001.

[10] E. N. Elnozahy, “Address Trace Compression Through Loop Detection and Reduction,” ACM SIGMETRICS Performance Evaluation Review, v.27 n.1, p.214-215, June 1999.

[11] L. DeRose et al., “SIGMA: A Simulator Infrastructure to Guide Memory Analysis,” in Proc. SC 2002, Baltimore, MD, 2002.

[12] M. Burtscher, M. Jeeradit, “Compressing Extended Program Traces Using Value Predictors,” in Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), New Orleans, LA, 2003.

[13] A. Milenković, M. Milenković, J. Kulick, “N-Tuple Compression: A Novel Method for Compression of Branch Instruction Traces,” in Proc. of the PDCS 2003, Reno, Nevada, 2003.

[14] SPEC 2000 Benchmark Suite, http://www.spec.org.

[15] D. Burger, T. Austin, “The SimpleScalar Tool Set Version 2.0,” Technical Report CS-TR-97-1342, University of Wisconsin, 1997.

[16] E. E. Johnson, “PDATS II: Improved Compression of Address Traces,” in Proc. of the IEEE International Performance, Computing, and Communication Conference, 1999.