Stream-Based Trace Compression

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

***Abstract*— Trace-driven simulation has long been used in both processor and memory studies. The large size of traces motivated 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 it is orthogonal to general compression algorithms. When combined with gzip, SBC reduces the size of SPEC CPU2000 traces 94-71968 times.**

***Index Terms* —simulation, instruction and address trace, trace compression.**

I. Introduction

Novel research ideas in computer architecture are frequently evaluated using trace-driven simulation. Traces can accurately represent a system workload, and in the last decade there has been a lot of research effort dedicated to trace issues, such as trace collection, reduction and processing [7]. 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 10GB of storage space. Yet, with a modern superscalar processor executing 1.5 instructions each clock cycle on average and running at 3 GHz, it will represent 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 [9], 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 the simulation, trace compression should not introduce a significant decompression slowdown. Hence, ideal trace reduction would be fast, with a high reduction factor, and loss-less, i.e., not introducing errors into the simulation.

Depending on the simulated system, a trace can contain different kinds of information. For example, for control flow analysis we need only a trace of executed basic blocks or paths. Cache studies require address traces, and more complex processor simulations also need instruction words. 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 need operand values.

Various trace compression techniques have been introduced, focusing on different trace information. One set of compression techniques, such as whole program path (WPP) and timestamped WPP, relies on program instrumentation and concentrates on instruction traces only [4], [8]. In WPP, a trace of acyclic paths is compressed using a modified Sequitur algorithm [5]. 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 (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 the one-pass PDATS algorithm, is to store address differences between successive references of the same type (load, store, instruction fetch) [3]. In PDATS, stored address differences can have variable length and an optional repetition count in cases when a constant difference is present in consecutive 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 [2]. A rather original approach regenerates original trace using a set of value predictors [1], 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.

In this paper, we propose a new method for single-pass compression of combined address and instruction traces,

Manuscript submitted: 1 August 2003. Manuscript accepted: 9 Sept. 2003. Final manuscript received: 11 Sept. 2003.

Stream-Based Data Compression (SBC). Our approach 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 data address stride and number of 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, our technique is orthogonal to general compression algorithms, such as gzip or Sequitur. We evaluated SBC on Dinero+ traces [3] of SPEC CPU2000 benchmark programs [6]. When combined with gzip, SBC reduces the trace size 94-71968 times, depending on the benchmark, and outperforms mPDI-gzip, gzipped combination of PDI and PDATS, 4.3-497.7 times. SBC combined with Sequitur reduces the trace size even further, but at the price of considerable decompression slowdown.

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 results. The last section gives concluding remarks.

II. Stream-Based Compression

SBC exploits several inherent characteristics of program execution traces. Instruction traces consist of a fairly limited number of different instruction streams while most of the memory references exhibit strong spatial and/or temporal locality, for example, a load having a constant address stride across loop iterations. 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: header field (0 – data read, 1 – data write, and 2 – instruction read), address field, and instruction word field for instruction read type.

First we will describe the decompression process, for the example in Fig. 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 STF 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).

Analysis of SPEC CPU2000 traces shows that a Stream Table has relatively modest memory requirements since almost all benchmarks have fewer than 5000 different instruction streams, and 9 out of 23 benchmarks have fewer than 1000 streams, while the average stream length is fewer than 30 instructions for 18 benchmarks. 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 SBDT during trace decompression, whenever the repetition count of an accessed node is 0.

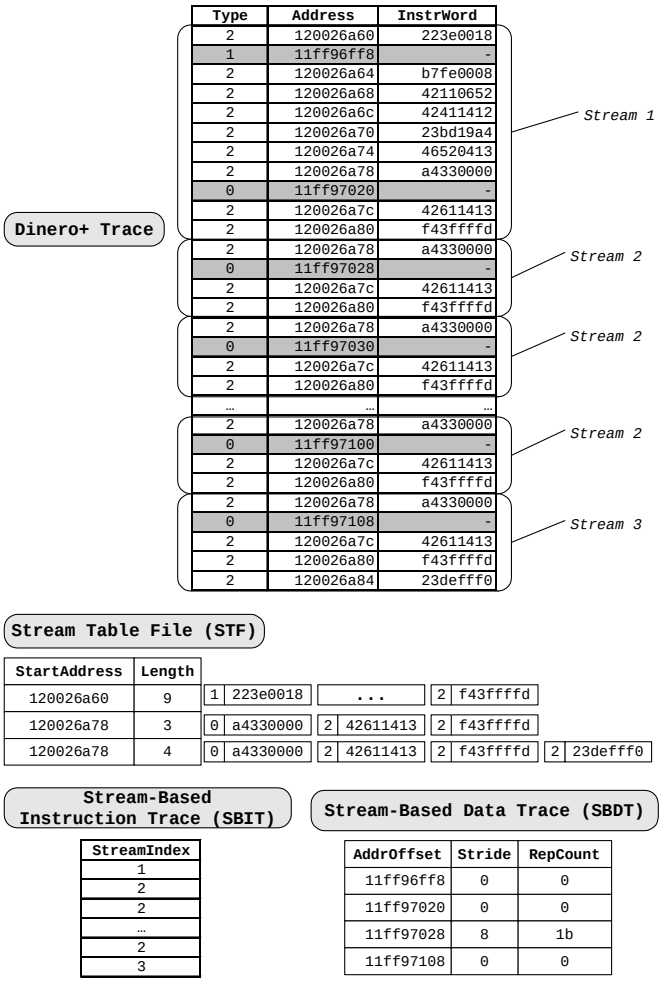

Fig. 1. Example of Stream-Based Compression.

Decompression proceeds as follows: a stream index is read from the SBIT, 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 we need the corresponding store address. 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 a 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, stride is

set to the value of the Stride field (in this case 0), and 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 an end of stream is detected in the original input trace, our compression program 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. On the other hand, our algorithm 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. 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. Hence we need the field AddrOffset, which 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 (Fig. 2), thus achieving additional compression. Clearly, the larger FIFO buffer will "catch" more data repetition, thus increasing the compression ratio.

DataHeader 1B	AddrOffset 1, 2, 4, or 8B	Stride 0, 1, 2, 4, or 8B	RepCount 0, 1, 2, 4, or 8B

Bits 7-5:RepCount size	Bits 4-2:Stride size	Bits 0-1:AddrOffset size
000: = 0 - 0B	000: = 0 - 0B	00: 1B
001: 1B	001: 1B	01: 2B
010: 2B	010: 2B	10: 4B
011: 4B	011: 4B	11: 8B
100: 8B	100: 8B	
101: =1 - 0B	101: =1 - 0B	
100: unused	100: =4 - 0B	
101: unused	101: =8 - 0B	

Fig. 2. SBC Data Trace Format.

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.

III. Results

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 our results do not overemphasize the program initialization. We compared the compression ratio of SBC with the compression obtained by gzip, and by mPDI, an improved version of PDI that uses the PDATS algorithm for data. In mPDI, data and instruction references are separated into two files, making the regular patterns even more recognizable by gzip.

Table I and Table II show the compression ratio of the compared algorithms relative to an uncompressed Dinero+ trace. Due to restricted space we show the combined compression ratio – the average between F2B and M2B for each benchmark, and Table III shows average values for F2B and M2B. Full set of results can be found at www.ece.uah.edu/~lacasa/sbc/sbc.htm. The FIFO buffer size for SBC is 4000 entries. The SBC algorithm reduces the trace size for up to 61.6 times for integer (CINT) and 458.2 times for floating point (CFP) benchmarks, outperforming gzip compression. On average, the SBC compression ratio for CINT is 36.2, and for CFP is 110. The very high compression ratio for some CFP benchmarks is due to longer instruction streams and higher repetition counts for data references. When SBC is compared to mPDI, the trace size is on average reduced 10.7 times for CINT and 36.1 times for CFP.

Even better compression ratios can be obtained by further compressing an SBC trace by gzip. The combined SBC-gzip compression ratio goes up to 962.7 times for CINT (*254.gap*) and up to 71968.1 (*171.swim*) for CFP, with corresponding average values of 334.1 and 12520.4. 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 and for some benchmarks, even less than 1MB. Compared to combined mPDI-gzip, SBC-gzip achieves on average 10.8 times better compression for CINT and 203.6 times better for CFP, 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. Fig. 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 19.9 times faster than Dinero.gz for floating-point, and 8.9 times for integer benchmarks.

TABLE I. COMBINED COMPRESSION RATIO FOR SPEC2000 INT.

CINT	mPDI	SBC	Din.gz	mPDI.gz	SBC.gz	SBC.seq
164.gzip	4.1	61.6	41.5	48.6	218.4	201.0
176.gcc	3.2	36.7	12.4	20.7	221.0	249.4
181.mcf	2.9	32.1	23.1	38.8	286.6	348.6
186.crafty	3.0	43.0	7.1	24.2	248.4	269.4
197.parser	3.6	34.1	28.5	33.3	179.0	348.5
252.eon	3.5	22.3	6.2	28.1	401.9	786.2
253.perlbmk	3.1	37.3	20.9	32.5	552.5	729.9
254.gap	3.5	43.4	23.8	37.8	962.7	1423.1
255.vortex	3.4	24.4	9.5	20.0	176.2	376.4
300.twolf	3.3	26.9	7.1	21.9	94.0	78.2
Average	3.4	36.2	18.0	30.6	334.1	481.1

TABLE II. COMBINED COMPRESSION RATIO FOR SPEC2000 FP

CFP	mPDI	SBC	Din.gz	mPDI.gz	SBC.gz	SBC.seq
168.wupwise	3.4	61.0	26.1	68.8	2463.2	4276.5
171.swim	2.9	458.2	22.7	167.6	71968.1	116512.0
172.mgrid	2.9	75.9	12.4	38.4	9279.4	16927.4
173.applu	2.9	72.0	13.6	24.0	3116.1	38446.8
177.mesa	2.9	79.1	10.5	53.9	1160.1	1641.8
178.galgel	3.0	77.9	24.5	33.8	10625.6	60477.7
179.art	3.5	74.7	25.2	33.7	16751.1	59764.1
183.equake	3.1	44.6	28.9	90.1	1152.1	1895.0
188.ammp	3.5	60.7	23.8	38.9	1534.9	2007.3
189.lucas	3.1	211.1	53.8	129.7	30257.0	68074.3
191.fma3d	3.5	79.8	8.8	16.7	7802.6	25912.7
200.sixtrack	2.9	99.6	16.8	41.6	4362.6	8758.6
301.appsi	3.0	35.0	8.3	19.3	2293.0	12421.9
Average	3.1	110.0	21.2	58.2	12520.4	32085.9

TABLE III. AVERAGE COMPRESSION RATIO FOR FIRST TWO BILLION (F2B) AND TWO BILLION AFTER SKIPPING 50 BILLION INSTRUCTIONS (M2B)

	mPDI	SBC	Din.gz	mPDI.gz	SBC.gz	SBC.seq
CINT F2B	3.5	37.1	15.1	30.0	309.0	406.3
CINT M2B	3.2	35.2	21.0	31.2	359.2	555.9
CFP F2B	3.5	101.8	23.5	72.0	9778.9	21745.1
CFP M2B	2.7	118.1	18.8	44.4	15261.9	42426.7

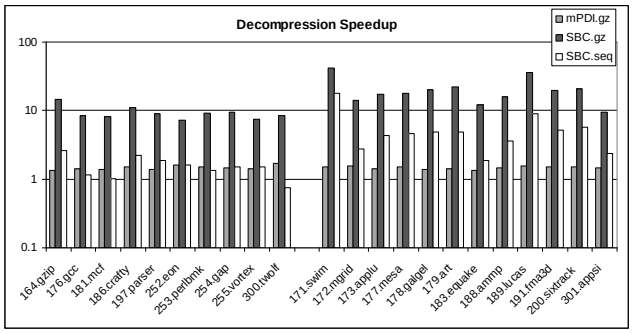

Fig. 3. Decompression Speedup relative to gzipped Dinero+ traces.

The results presented do not include traces compressed with Sequitur only. While Sequitur on its own is very efficient for instruction address traces, it doesn't perform as well on data traces, producing larger compressed files than gzip for most benchmarks. 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 instruction trace compression would not significantly increase the overall compression ratio.

IV. CONCLUSION

The SBC algorithm offers a new technique for compressing data address references in combined instruction and address traces: data address information is linked to a corresponding instruction stream. It 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 one-pass algorithm, SBC can be easily modified for online tracing in real-time, and implemented in hardware.

We are currently investigating possible extensions of the SBC algorithm, such as further refining the algorithm and its applicability to another trace information such as the value of operands. For example, a two-level scheme can reduce the size of the Stream Table: instead of instructions, each entry in the Stream Table keeps the indexes of the basic blocks from the Basic Block Table.

ACKNOWLEDGMENT

The authors are grateful to the editors and anonymous referees for their insights and suggestions for strengthening this paper.

REFERENCES

[1] 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.

[2] 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.

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

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

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

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

[7] R. Uhlig, T. Mudge, "Trace-driven memory simulation," *ACM Computing Surveys*, Vol. 29, No. 2, June 1997.

[8] 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.

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