N-TUPLE COMPRESSION:
A NOVEL METHOD FOR COMPRESSION OF BRANCH INSTRUCTION TRACES

Aleksandar Milenković, Milena Milenković, and Jeffrey Kulick
Electrical and Computer Engineering, The University of Alabama in Huntsville
Email: milenka@ece.uah.edu, URL: http://www.ece.uah.edu/~milenka

Abstract

Branch predictors and processor front-ends have been the focus of a number of computer architecture studies. Typically they are evaluated separately from other components using trace-driven simulation based on instruction traces. To offer a faithful representation of processor's workload the traces are very large, and hence difficult to manage if kept in uncompressed form. In order to reduce simulation overhead due to the processing of non-branch instructions, we propose a new form of instruction trace, the Branch Instruction Trace (BIT), suitable for simulation of dynamic branch prediction mechanisms, fetch engines, and trace caches. A novel method for lossless trace compression, which can be applied to both ASCII and binary BIT traces, is also introduced. The proposed method relies on the trace record table (TRT) consisting of unique trace records. The trace size can be reduced by replacing each trace record by its ID in the TRT, since the number of unique trace records is much less than the trace length. We further extend this idea and replace an entire N-tuple of BIT records with its ID from the N-Tuple Record Table (N-TRT). The analysis shows that for a subset of SPEC CPU2000 benchmarks 8-tuple replacement yields significant compression ratio (40 for binary traces and 32-43 for ASCII traces), while keeping N-TRT size reasonable. When combined with the common compression tools such as gzip the compression ratio is 195-3888 for binary, and 306-4604 for ASCII traces, while gzipped-only traces achieve compression ratio 20-201 for binary, and 20-216 for ASCII traces.

Keywords*: trace-driven simulation, branch prediction, trace compression.*

1. INTRODUCTION

The dynamic branch predictor is frequently used to improve instruction level parallelism in processor control flow. In order to estimate the limits and the benefits of a novel front-end architecture for a large set of design parameters, the fetch-engine, branch prediction mechanisms, and trace caches are frequently evaluated independently of other processor components. If trace-driven simulation is used with some common form of full instruction traces, the simulator has to deal with a lot of data unrelated to branch instructions, thus increasing the simulation time.

In order to have only the information necessary for front-end processor simulation (branch predictor, fetch engine, trace cache, etc.), we propose a format for branch instruction traces. One record in this trace contains data about branch instruction address, target address, branch type, and the length of the corresponding basic block.

To offer a faithful representation of a specific workload, traces must be 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 10GB of storage space. Yet, with a modern superscalar processor executing 1.5 instructions each clock cycle and running at 3 GHz, a 10 GB file will represent only 0.2 seconds of the CPU execution time. To efficiently store and use even a small collection of traces, the trace size must be reduced as much as possible. Since traces have a lot of redundant information [1], compression tools based on Ziv-Lempel algorithm [2], such as the gzip program, can achieve a very good compression ratio. However, even better compression is possible by taking advantage of the redundancy in address and instruction traces. On the other hand, since the ultimate purpose of traces is to be used in a simulation, the trace compression should not introduce a significant decompression slowdown. Hence, the ideal trace reduction would be fast, with a high reduction factor, and lossless, i.e., not introducing errors into the simulation [3].

In this paper, we propose a novel method for compression of branch instruction traces. In the proposed method, a trace is divided into N-tuples, with each n-tuple corresponding to a sequence of events. An N-Tuple Record Table (N-TRT) is generated, consisting of unique N-tuples. Each N-tuple in the trace is then replaced with its index in the N-TRT. When the trace is accessed during a simulation, each tuple can be easily and rapidly converted back to N original trace records. We evaluated this method on traces generated using modified SimpleScalar environment [4], selected SPEC CPU2000 benchmarks [5], and reduced input dataset [6]. Each trace consists of smaller partial traces that can be individually compressed and decompressed. The proposed compression method can be applied to both ASCII and binary traces, and can be easily applied for any full instruction trace format.

When combined with gzip, N-tuple compression achieves significant improvement in comparison to traces compressed only by gzip. For 1-tuple compression, this

gain (over gzip only) ranges from 2.19 to 9.74 times for binary, and from 2.65 to 7.16 for ASCII traces, while for 8-tuple compression the gain is 2.47-29.63 for binary, and 3.12-32.82 for ASCII traces, achieving combined compression rate of up to 4600 times.

The remainder of the paper is organized into five sections. The second section gives some details about related trace compression work. The third section explains the branch instruction trace format and gives details about preliminary trace analysis. The fourth section explains the proposed compression mechanism, and shows the compression ratio obtained for N-tuple compression. The last section gives concluding remarks.

2. RELATED WORK

Simulations based on the use of program execution traces have been widely employed in computer architecture research. Trace related issues form a research area on their own, including trace collection, reduction and processing [3]. Traces must faithfully represent targeted workloads, and different software and hardware methods must be developed for complete, detailed and undistorted trace collection. In order to be representative, traces must include a large number of references, thus requiring enormous amount of storage. Since keeping traces in an unreduced form is infeasible, some method of trace reduction must be applied, ranging from trace sampling and filtering, to loss-less trace compression.

Program execution traces can include only addresses of memory references (address traces), if they are used in simulations of memory hierarchy. For example, Dinero trace format record [7] consists of the address of memory reference and the reference type - read, write, or instruction fetch, while BYU traces [8] also include additional information, such as the size of the data transfer, processor ID, etc. If we want to evaluate performance of different CPU designs, we also need instruction traces, i.e., traces including at least the operation code for an instruction along with its address. One such example is the IBS trace format [9], which also includes information about user or kernel activity. Another example is the PDI trace format, an extension of PDATS [10], which includes the instruction word and address. The branch instruction trace proposed in this paper is a subset of the instruction trace family.

PDI instruction words are compressed using a dictionary-based approach [10]. A dictionary of 256 most frequently used instruction words is stored at the beginning of the trace file, and each of those words in the trace is replaced with its index in the dictionary. The words that are not in the dictionary are left unchanged. The dictionary can be created for each trace file separately, and then compression requires two passes through the trace. Another approach is to use the same generic dictionary for all traces. The generic approach has a lower compression ratio, but requires only a single pass trace processing. PDI addresses are compressed using the PDATS method, where an address in the trace stream is replaced by its offset. One stream consists of the same type of addresses, such as load, store, or instruction fetch streams. On ASCII traces of the SPEC92 programs, PDI has compression ratio 5-8. When a trace compressed by PDI method is further compressed using gzip, the combined compression ratio is in the range 23-426.

Analysis based on basic blocks, rather then separate instructions, is employed for address traces stored in two compressed formats, MTRACE [11] and RPS (Recovered program structure) [12]. The MTRACE format stores the address of the first instruction of each basic block, and data address references within that block. An auxiliary file describing basic blocks is used to reconstruct the trace. The RPS format also stores trace data in multiple files: one file stores information about basic block length and positions of load and store instructions within blocks, another file keeps invocation sequence of the basic blocks, and separate files store data address references.

3. BRANCH INSTRUCTION TRACES

The branch instruction traces for this paper were collected using the modified SimpleScalar system [4], a widely used set of simulation tools developed at the University of Wisconsin. Each record in the trace file consists of the following fields: BranchAddress, TargetAddress, OpCode, BBlockLength (Table 1). The same fields exist in both ASCII and binary trace format. The basic block length is the number of instructions executed between two consecutive branches, including the last branch. Preliminary trace analysis shows that one byte is enough for the basic block length field in the binary format, while a special value of the OpCode field can be used in the case of larger basic blocks, and such data can be stored in two trace records instead of one. Figure 1 shows an excerpt from an ASCII trace file. While the binary format requires less storage space, the ASCII format is convenient for trace examining, editing, and simulator debugging, as well as in the educational purposes [10]. We applied our compression method to both ASCII and binary traces, in order to show that it is independent of the trace format.

We collected the branch instruction traces for a subset of SPEC CPU2000 benchmarks [1], including five integer (*gcc – scilab.s* input, *gzip - graphic* input, *mcf*, *parser*, and *vortex*) and four floating-point benchmarks (*ammp*, *art*, *equake* and *mesa*). We used large inputs from the reduced datasets [6], developed specifically for simulation purposes at the University of Minnesota. All benchmarks were compiled for the MIPS-like PISA architecture, using the Gnu C compiler with O3 optimization flag. The SimpleScalar's modified *bpred* simulator was run under RedHat 7.2 Linux OS.

All considered benchmarks were run to completion, and Table 2 shows the number of trace records, i.e., the

executed branches. In order to have trace files of manageable size, each complete trace consists of a set of partial trace files, all but the last (foreshortened) trace file containing 1,000,000 trace records. The size of an uncompressed partial trace file is about 20MB for an ASCII trace, and 10MB for a binary trace. Each partial trace file is compressed using the standard gzip program, and all partial traces are stored in a tar file. The size of a gzipped partial trace can vary from less than a 50K (*mcf* benchmark), to more than 1MB (*gcc*). This storage structure is similar to the structure of IBS traces [9], and enables the use of traces even with relatively modest computer configurations that have low disk space.

Table 1 Branch trace record fields.

Field	**Description**	**Length [bytes]**
BranchAddress	Branch instruction address	4
TargetAddress	Branch target address	4
OpCode	Branch opcode	1
BBlockLength	Length of a basic block ending with that branch	1

Branch Address	Target Address	OpCode	BBlock Length
4001a8	41a3d0	3	e
41a3e0	41faa0	3	3
41fac0	41fac8	6	5
41fad0	41fad8	6	2
41fae0	41b890	3	2
41b8b8	41b8d8	7	6
41b8e8	423a00	3	3

Figure 1 Branch Instruction Trace example

Table 2 Trace records statistics (whole trace)
Legend: #TRs – number of trace records in a trace, #UTRs – number of unique trace records

Benchmark	**#TRs**	**#UTRs**	**#UTRs making 90% of TRs**
164.gzip.graphic	251,460,618	1874	110
176.gcc	759,484,369	52961	5494
177.mesa	325,124,149	2602	74
179.art	310,425,057	1093	44
181.mcf	135,550,269	1822	66
183.equake	112,253,537	2027	258
188.ammp	369,474,018	1621	83
197.parser	823,886,062	7367	360
255.vortex	168,611,654	15237	821

In order to explore the potential for compression, we counted the number of unique trace records (#UTR), both for whole trace (Table 2) and per partial trace file, i.e., per one million branch instructions (Figure 2). Table 3 shows some additional statistics: the average, minimum, and maximum number of UTRs per partial trace file. Compared to the trace length, the number of UTRs for the whole trace is very small, ranging from just about 1000 unique records (*art*) to about 53000 (*gcc*). If we consider the number of unique trace records per partial trace file, these numbers are even lower, ranging from only 1 (*mcf*) to 25024 (*gcc*). The number of UTRs accounting for 90% of the trace is in the range of 44 (*art*) to 5494 (*gcc*). All this data indicates very good compression potential.

Table 3 Statistics for unique trace records per partial trace file.

Benchmark	avr	min	max
164.gzip	335	95	1063
176.gcc	4410	9	25024
177.mesa	176	165	1759
179.art	121	57	565
181.mcf	170	1	917
183.equake	183	62	1021
188.ammp	422	335	1131
197.parser	378	142	4042
255.vortex	2593	276	5868

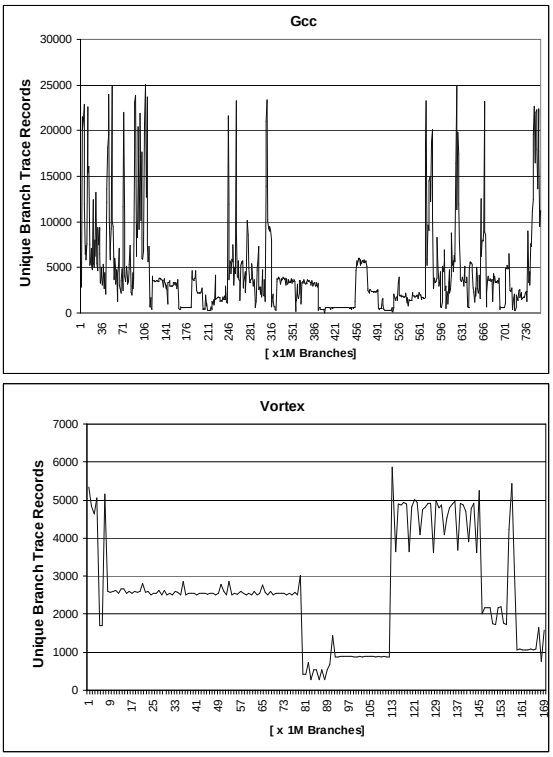

Figure 2 Number of unique trace records per 1M branch instructions for two SPEC CPU2000 benchmarks

4. BRANCH TRACE COMPRESSION

The storage problem of traces has several different aspects. First, the trace access time can take a major portion of the simulator execution time. Second, trace archives require a significant server storage space and download time, or a significant storage space on

removable media, such as compact disks. Finally, uncompressed partial traces occupy a considerable amount of hard disk space. To alleviate these problems, we propose a simple, yet efficient compression algorithm, based on the preliminary trace analysis in Section 3. Since the number of unique trace records is relatively small, we can create a table of unique trace records, and replace each trace record with the corresponding table record ID, similarly to the PDI dictionary approach. This idea can be further extended to unique N-tuples of consecutive trace records, providing that the size of the N-tuple record table (N-TRT) does not increase too much, compared to the trace record table consisting of unique trace records. One alternative approach is to provide an N-TRT table per each partial trace. This approach gives smaller partial table sizes, so there is less pressure on the memory during simulation, but requires more storage space and longer access time. This approach has not been studied in this paper.

Figure 3 illustrates N-tuple compression method for N=1 and N=2, where the BIT (Branch Instruction Trace) is compressed to cBIT-T1 (compressed BIT using 1-tuple compression) and cBIT-T2 (compressed BIT using 2-tuple compression). For example, trace record *4f66b8 4f66f0 6 4* is replaced with its table ID, 529, using 1-tuple compression, or two trace records, *4f66e8 4f66c0 7 3* and *4f66c8 4f66d0 6 2* are replaced by a respective 2-TRT ID if 2-tuple compression is used.

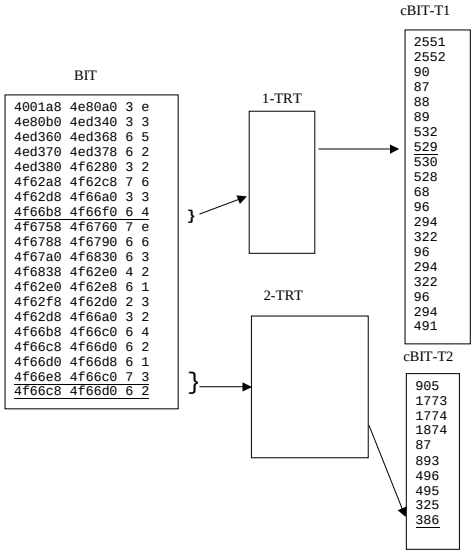

Figure 3 N-tuple compression example, for N=1 and N=2

One example improvement of the N-tuple compression method over raw (uncompressed) traces is shown in (Figure 4). For all binary traces except gcc, each N-tuple in the trace is replaced by its' corresponding 2-byte N-TRT ID, yielding compression ratios of 5, 10, 20, 40 and 60, for 1-, 2-, 4-, 8-, and 12-tuple compression, respectively. Since *gcc* has a larger TRT table, we used 4 bytes for its table ID for N>1, which gives half of the compression ratio of other benchmarks. The ASCII compression ratio varies, since the records have variable length, but it stays close to the binary compression ratio for all considered benchmarks.

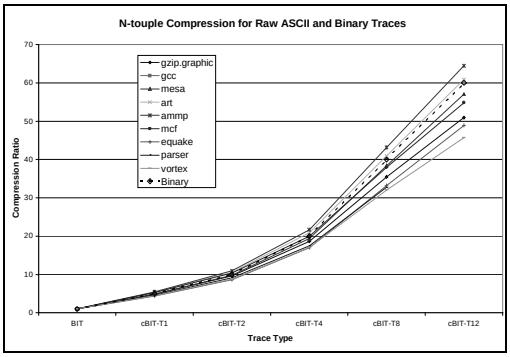

Figure 4 N-tuple compression ratio for raw ASCII and binary traces

The N-tuple compression shows its real benefit when combined with the gzip compression, i.e., when each partial trace is first compressed using the N-tuple algorithm, and then gzipped. In an N-tuple compressed trace, patterns are better exposed to the gzip algorithm, since a trace record or a group of records is replaced with its table ID. Table 4 shows the combined compression ratio for binary traces. The gzip-only compression is in the range of 20 for *vortex* to 201 for the *art* benchmark. Even a 1-tuple compression improves this ratio, from 127 times for *gcc*, to 582 for *art*, while for 8-tuple compression ratio is 195-3888. The benchmark *parser* reaches the point of diminishing returns for 4-tuple, with 8-tuple combined compression slightly worse than 4-tuple. Figure 5 shows the N-tuple+gzip compression gain relative to the gzip-only compression for binary traces. We can see that for *mesa*, *ammp*, and *vortex* this gain increases significantly when the tuple size increases, even for 12-tuple, where *mesa* is compressed over 35 times. Another group of benchmarks, *art*, *gzip*, *parser*, and *mcf*, does not benefit much from the tuple size increase, with a gain of about 4. The rest of the benchmarks, *gcc* and *equake*, show moderate improvement when the tuple size increases from 8 to 12 – from 7 to 7.8 times for *gcc*, and from 11.2 to 11.5 for *equake*. Hence, we suggest 8 or 4 as the optimal tuple size. Table 5 and Figure 5 show combined compression ratio and compression gain for ASCII traces. Again, combined compression ratio is impressive, reaching over 4500 for *mesa* benchmark and 8-tuple compression.

During a simulation, the simulator now needs to read just one N-TRT ID instead of N trace records, and to access the corresponding N-TRT entry. The most frequently accessed parts of the TRT table will be stored in the memory, thus reducing the overhead of trace reading during a trace-driven simulation. Table 6 shows the access time for a simulation using binary traces and corresponding access time speedup for compressed traces, both for the total user+system time, and for the system

time component, while Figure 6 shows the average speedup. We measured access time using a simple program that reads the whole trace, on an AMD Athlon PC at 2.2GHz, with 256MB memory, again under Linux Red Hat 7.2 OS. The trace access time will be dependant on the computer configuration where the simulation is performed, but we can expect the similar access times speedup.

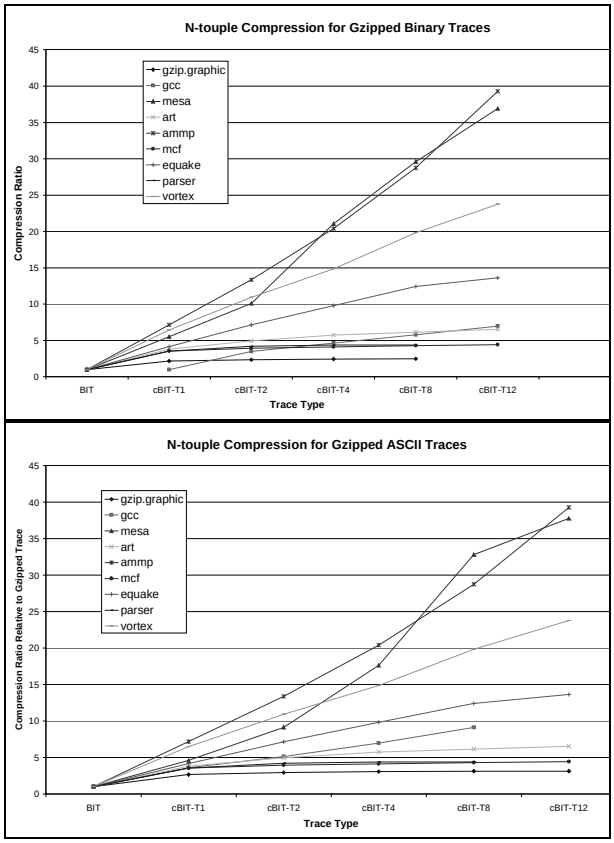

Figure 5 Compression gain relative to gzipped trace archive for binary and ASCII traces.

Table 4 Combined N-tuple-gzip compression ratio of binary traces (BIT – original gzipped trace)

Bin	BIT	cBIT-T1	cBIT-T2	cBIT-T4	cBIT-T8	cBIT-T12
164.gzip	79	173	185	194	195	197
176.gcc	36	127	168	209	252	284
177.mesa	131	724	1323	2762	3888	4845
179.art	201	582	695	775	819	864
181.mcf	72	219	242	256	271	280
183.equake	110	525	812	1006	1232	1260
188.ammp	56	513	878	1191	1576	1950
197.parser	66	215	236	240	236	-
255.vortex	20	190	245	311	417	476

The access time speedup is not linear, since some time must be spent on accessing the tuple record table. While TRT does not add much to the size of the compressed trace, its size may pose a problem during simulation on a system with relatively low available memory. Figure 7 shows the number of unique trace records for different tuple compressions. The table may store raw tuples, or two-level tables may be used, thus reducing the overall size. In a two-level implementation, an N-TRT table holds IDs of the 1-TRT table, instead of whole trace records.

Table 5 Combined N-tuple-gzip compression ratio of ASCII traces (BIT – original gzipped trace)

ASCII	BIT	cBIT-T1	cBIT-T2	cBIT-T4	cBIT-T8	cBIT-T12
164.gzip	98	259	287	300	306	306
176.gcc	20	70	100	136	178	210
177.mesa	140	641	1282	2473	4604	5296
179.art	216	822	1070	1243	1331	1417
181.mcf	92	328	363	381	397	410
183.equake	115	478	822	1129	1430	1570
188.ammp	54	388	725	1106	1559	2130
197.parser	73	260	305	320	318	-
255.vortex	20	126	213	290	387	464

Table 6 Binary traces access time and N-tuple speed-up

	binary BIT [s]		cBIT-T1 speedup	
	total	system	total	system
164.gzip	29.76	14.93	1.11	3.00
177.mesa	33.80	19.80	1.02	3.17
179.art	45.24	18.75	1.32	3.28
181.mcf	17.92	8.16	1.25	3.26
183.equake	12.55	6.69	1.07	2.99
188.ammp	54.73	23.23	1.35	3.35
197.parser	99.90	55.52	1.14	3.49
255.vortex	20.76	10.60	1.17	3.20

	cBIT-T4 speedup		cBIT-T8 speedup	
	total	system	total	system
164.gzip	3.88	8.90	6.43	12.20
177.mesa	3.65	9.52	6.52	14.51
179.art	4.80	9.73	8.48	14.55
181.mcf	4.37	9.13	7.39	13.59
183.equake	3.85	9.38	6.65	13.12
188.ammp	4.84	9.93	8.74	14.49
197.parser	3.90	9.02	5.99	10.02
255.vortex	3.99	8.50	6.59	11.68

5. CONCLUSION

Trace-driven simulation is a very useful tool in computer architecture research, especially for exploring different implementations of front-end processor components used to increase available instruction level parallelism, such as the dynamic branch predictor. Since traces of complete program runs require rather large storage space, traces are usually stored in compressed form, using some of the widely available compress tools such as gzip.

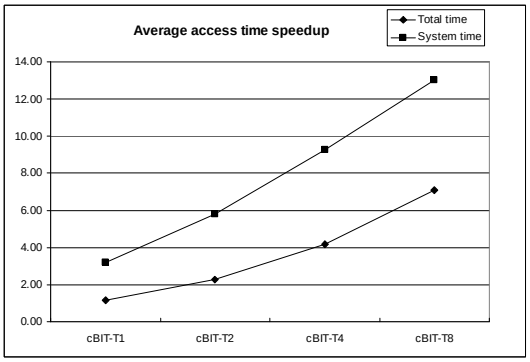

Figure 6 Average access time speedup for compressed binary traces

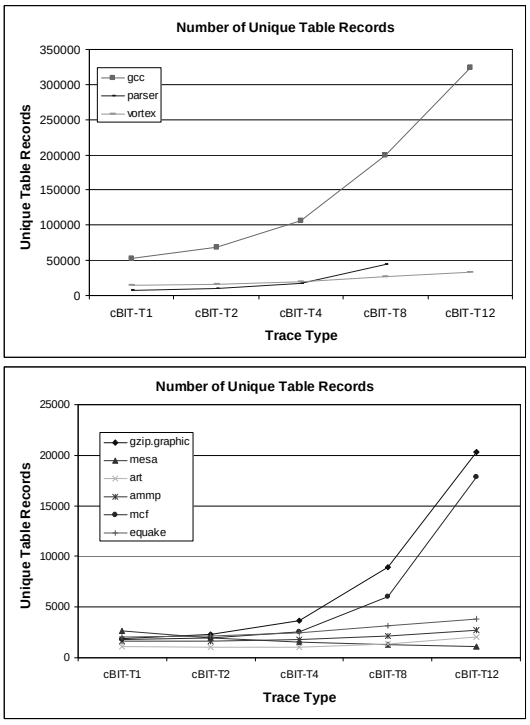

Figure 7 Number of unique table records

The contributions of this paper are the following: first, we propose a branch instruction trace format, containing only information needed for the front-end processor simulation. Then, we propose a novel trace compression which does not aim to replace gzip compression, but rather to modify original trace so that trace structure is preserved, while further reducing the size of both uncompressed and compressed trace. Consecutive N-tuples (sequences of N trace records) are replaced with corresponding IDs in the N-Tuple Record Table. Since the number of unique N-tuples is relatively small compared to the number of N-tuples in the trace, this method is applicable for all considered benchmarks. The compressed BIT trace together with the compressed trace record table is significantly smaller than the original compressed trace, up to almost 30 times for some benchmarks and 8-tuple compression. Trace access time is also reduced. Finally, we collected and compressed state-of-the-art SPEC CPU2000 traces in the SimpleScalar environment, and made our trace database available to the research community.

Future work includes extending the proposed method to full instruction and address trace. In this case, the tuple record table will hold data about tuples of basic blocks. The memory reference trace can be stored in the separate file, using some other compression method such as PDATS.

Another future direction is to extract path information from the TRT, and replace each repeatable sequence in the trace with its repetition count and the Path Record Table ID. Again, the conversion of each PRT record into corresponding sequence of trace records can be done easily during a trace-driven simulation. We expect this compression method to achieve even better size reduction.

Acknowledgements

This work has been partly supported by the Software Engineering Directorate of the U.S. Army Missile Command.

REFERENCES

[1] Becker, J., Park, A. "An analysis of the information content of address and data reference streams," *Proc. 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems.*

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

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

[4] Burger, D., Austin, T., "The SimpleScalar Tool Set Version 3.0," University of Wisconsin Madison Computer.

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

[6] KleinOsowski, AJ., Flynn, J., Meares, N., Lilja, D.J., "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research," *Proc. Workshop on Workload Characterization (ICCD)*, September, 2000.

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

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

[9] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., Emer, J., "Instruction fetching: coping with code bloat," *Proc. 22nd ISCA*, June 1995.

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

[11] Elnozahy, E.N., "Address Trace Compression Through Loop Detection and Reduction," *Proc. of the 1999 ACM SIGMETRICS conference on Measurement and modeling of computer systems*.

[12] Fox, A., Grün, T., "Compressing Address Trace Data for Cache Simulations," *Proc. of International Data Compression Conference*, February 1997.