An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams

ALEKSANDAR MILENKOVIĆ and MILENA MILENKOVIĆ[‡]

The University of Alabama in Huntsville

[‡] IBM, Austin, TX

Trace-driven simulations have been widely used in computer architecture for quantitative evaluations of new ideas and design prototypes. Efficient trace compression and fast decompression are crucial for contemporary workloads, as representative benchmarks grow in size and number. This paper presents Stream-Based Compression (SBC), a novel technique for single-pass compression of address traces. The SBC technique compresses both instruction and data addresses by associating them with a particular instruction stream, i.e., a block of consecutively executing instructions. The compressed instruction trace is a trace of instruction stream identifiers. The compressed data address trace encompasses the data address stride and the number of repetitions for each memory-referencing instruction in a stream, ordered by the corresponding stream appearances in the trace. SBC reduces the size of SPEC CPU2000 Dinero instruction and data address traces from 18 to 309 times, outperforming the best trace compression techniques presented in the open literature. SBC can be successfully combined with general-purpose compression techniques. The combined SBC-gzip compression ratio is from 80 to 35,595, and the SBC-bzip2 compression ratio is from 75 to 191,257. Moreover, SBC outperforms other trace compression techniques when both decompression time and compression time are considered. This paper also shows how the SBC algorithm can be modified for hardware implementation with very modest resources and only a minor loss in compression ratio.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**] Performance Analysis and Design Aids; C.0 [**General**] *Modeling of computer architectures*; C.4 [**Performance of Systems**]: *Modeling techniques*; *Design studies*; E.4 [**Coding and Information Theory**]: *Data compaction and compression*.

General Terms: Algorithms, Design, Experimentation, Measurement, and Performance

Additional Key Words and Phrases: instruction and data traces, trace compression, instruction streams.

Authors' addresses: A. Milenkovic, Department of Electrical and Computer Engineering, The University of Alabama in Huntsville, 301 Sparkman Drive, Huntsville, Alabama 35899.
M. Milenkovic, IBM, 11501 Burnet Rd., Austin, Texas 78758.

1. INTRODUCTION

Software simulations have been a vital tool in computer architecture for quantitative evaluations of new ideas and design prototypes. Both industry and academia rely extensively on simulation because it is the easiest and least expensive way to characterize and explore design space [Skadron et al. 2003]. A common simulation technique is trace-driven simulation, where the input to the simulator is a trace of relevant events, collected during execution of a realistic workload. Though execution-driven simulators may provide faster simulation and more flexibility, they are often not readily available for a computer system of interest. Building a full-scale execution-driven simulator is often impractical and unnecessary, and certainly it is very expensive. Moreover, in performance-tuning efforts for real-world server applications, recreating conditions and inputs on an execution-driven simulator is often impossible. Consequently, program execution traces for trace-driven simulation remain an important resource for computer engineers.

In the last decade many research efforts have been dedicated to trace issues, such as trace collection, reduction, and processing [Uhlig and Mudge 1997]. Depending on its purpose, a trace can contain different types of information. For example, control flow analysis needs only a trace of the executed basic blocks or paths. Cache studies require address traces, and more detailed 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; ALU unit simulations require operand values. For example, a Dinero trace record consists of the address of the memory reference and the reference type (read, write, or instruction fetch) [Edler and

Hill 1998]. BYU traces include additional information, such as the size of the data transfer and processor ID [Thornock and Flanagan 2001]. ATUM traces (Address Tracing Using Microcode) also include the process ID, and encompass information about system activity, such as mapping between physical and virtual memory at each translation look-aside buffer miss [Agarwal, Sites and Horowitz 1986]. An IBS trace record contains the operation code and the user/kernel indicator [Uhlig et al. 1995]. Traces collected using the pixie tool differentiate between records for load/store memory references of different size, such as byte, double, or word [Smith 1991].

To efficiently store, transfer, and use even a small collection of traces, the traces must be compressed as much as possible. Each new generation of the industry-recognized SPEC benchmark suite has a longer run-time, larger resource requirements, and a larger set of benchmarks [SPEC 2000]. The number of executed instructions in SPEC CPU2000 benchmarks with reference input sets ranges from 62 to 547 billion [Cantin and Hill 2003]. If each benchmark is executed with only one reference input set, the sum of the executed instructions is about 7.5 trillion. We would need almost 70 terabytes of disk space to store these traces, assuming 10-byte trace records.

An effective trace compression technique is lossless (i.e., it does not introduce any errors into the simulation), has a high compression factor, and has a small decompression overhead. Although traditional compression techniques, such as the Ziv-Lempel algorithm [Ziv and Lempel 1977] used in the *gzip* utility or Burroughs-Wheeler transformation [Burrows and Wheeler 1994] used in the *bzip2* utility, offer a good compression ratio, even better compression is possible when the specific nature of redundancy in traces is taken into account. Better compression techniques not only enable more representative input and faster execution of trace-driven simulation, but also help in other uses of traces, such as the dynamic analysis of program behavior, debugging, and daily system maintenance [Zhou and Smith 2000].

This paper introduces Stream-Based Compression (SBC), a new method for single-pass compression of address traces and various extended trace formats. The SBC algorithm relies on extracting instruction streams. 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. A stream table created during compression encompasses all relevant information about streams, such as the starting address, stream length, and instruction types. All instructions from a stream are replaced by its index in the stream table, creating a trace of instruction streams.

SBC features an efficient on-line algorithm for compression of data address references. Unlike instruction addresses, data addresses for a memory-referencing instruction rarely stay constant during program execution, but they can have a regular stride. The SBC-compressed data address trace encompasses a data address stride and the number of repetitions for each memory-referencing instruction in a stream. A change of the data address stride results in another record in the compressed trace. The records are ordered by the corresponding stream appearances in the original trace.

In this paper we also show how a slightly modified SBC algorithm can be used in a hardware resource called a trace compressor that will allow on-line trace compression with only modest resource usage. Such a new resource can be very useful for system debugging and verification, one of the most significant problems in emerging SOCs

(Systems-On-the-Chip) where physical processor pins are not available [Fisher, Faraboschi and Young 2005].

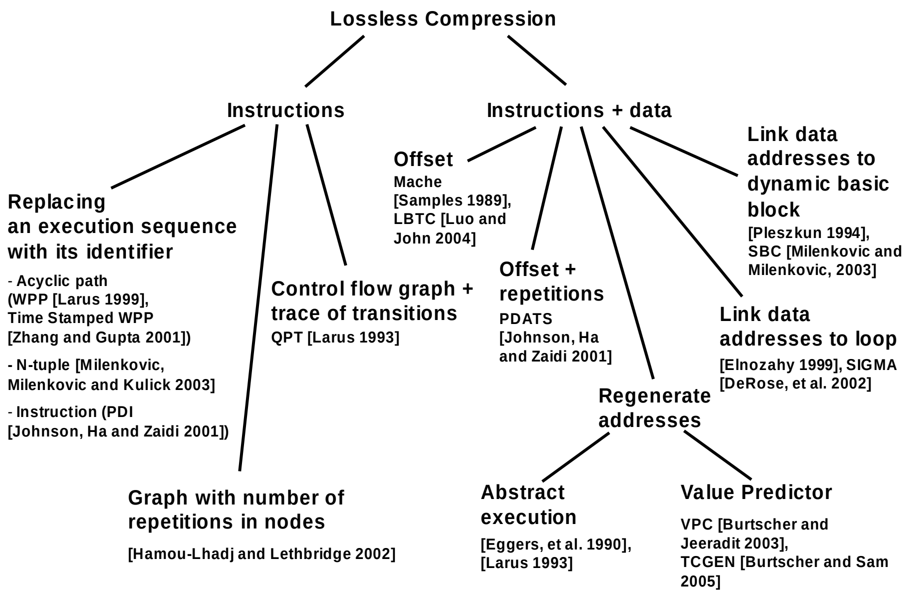
The proposed algorithm achieves a very good compression ratio and decompression/compression time for instruction and data address traces, yet it is straightforward to implement and does not require code augmentation or lengthy multi-pass trace analysis. Furthermore, SBC can be successfully combined with general compression algorithms, such as Ziv-Lempel or Burroughs-Wheeler.

We evaluate the proposed technique's efficiency by measuring compression ratio, decompression, and compression time and comparing its performance versus the best existing techniques for trace compression, such as PDATS [Johnson, Ha and Zaidi 2001], TCGEN [Burtscher and Sam 2005], and LBTC [Luo and John 2004]. We also consider these techniques when they are combined with *gzip* and *bzip2*. We use full instruction and data address traces [Edler and Hill 1998] of SPEC CPU2000 benchmark programs as the input. Detailed experimental analysis shows that SBC outperforms other techniques, achieving a good balance between compression ratio and decompression time. Using the trace compression ratio of the sum of all traces as a metric, SBC reduces the trace size 35.9 times, versus 7.2 times reduction with PDATS, 4.9 with TCGEN, and 7.6 with LBTC. When compression techniques are combined with *gzip*, the compression ratio with SBC.gz is 326.6 versus 62.3 with PDATS.gz, 271.1 with TCGEN.gz, and 84.7 with LBTC.gz. Additional compression with *bzip2* reduces the trace size even more: the compression ratio with SBC.bz2 is 390, 84.8 with PDATS.bz2, 722.3 with TCGEN.bz2, and 122.8 with LBTC.bz2. Although TCGEN.bz2 has a higher compression ratio than SBC.bz2 for all but five benchmarks, SBC.bz2 has from 2 to 7 times shorter decompression time. In addition, we show that SBC-compressed traces provide the shortest simulation time with a real cache memory hierarchy simulator.

The rest of this paper is organized as follows. The next section gives an overview and a new classification of the existing techniques for trace compression. Section 3 explains stream-based compression and decompression processes and discusses implementation of the SBC algorithm in hardware. Section 4 evaluates the effectiveness of SBC versus other trace compression techniques, comparing compression ratio, compression time, and decompression time. The last section gives concluding remarks.

2. RELATED WORK

We can broadly classify traces in two categories. The first category encompasses traces that have information only about the instruction flow: traces of instruction addresses, basic blocks, executed branches, or procedure calls. The traces in the second category also include records with highly variable values such as data addresses or operand values. Consequently, trace compression techniques can also be divided into two groups: techniques targeting only instruction flow traces, and those targeting traces including both instruction and data information. Fig. 1 shows this classification for lossless trace compression techniques.



Lossless Compression
Instructions
Instructions + data
Replacing an execution sequence with its identifier
- Acyclic path (WPP [Larus 1999], Time Stamped WPP [Zhang and Gupta 2001])
- N-tuple [Milenkovic, Milenkovic and Kulick 2003]
- Instruction (PDI [Johnson, Ha and Zaidi 2001])
Graph with number of repetitions in nodes
[Hamou-Lhadj and Lethbridge 2002]
Control flow graph + trace of transitions
QPT [Larus 1993]
Offset
Mache [Samples 1989], LBTC [Luo and John 2004]
Offset + repetitions
PDATS [Johnson, Ha and Zaidi 2001]
Regenerate addresses
Abstract execution
[Eggers, et al. 1990], [Larus 1993]
Value Predictor
VPC [Burtscher and Jeeradit 2003], TCGEN [Burtscher and Sam 2005]
Link data addresses to dynamic basic block
[Pleszkun 1994], SBC [Milenkovic and Milenkovic, 2003]
Link data addresses to loop
[Elnozahy 1999], SIGMA [DeRose, et al. 2002]

Fig. 1. Lossless trace compression techniques.

The underlying idea for several instruction trace compression techniques is to replace an execution sequence with its identifier. For example, in the *whole program path* technique (WPP), a program is instrumented to produce a trace of acyclic paths [Larus 1999]. Acyclic paths are then compressed using a modified Sequitur algorithm [Nevill-Manning and Witten 1997]. WPP yields a very good compression ratio, and it is convenient for certain types of analysis, such as finding the most frequently executed paths. The downside is that it requires the code to be instrumented, so it is not directly applicable for already-recorded traces. Time-stamped WPP [Zhang and Gupta 2001] enables fast access to the trace of a particular function: WPP is broken into path traces corresponding to individual function calls, and all path traces for one function are stored together as a block. Another compression technique replaces an *N*-tuple of original trace records with its identifier from the *N*-Tuple record table [Milenkovic,

Milenkovic and Kulick 2003]. The reduced trace is then compressed using *gzip*. The redundancy in a trace is better exposed to *gzip*, so this technique achieves up to 30 times better compression than with *gzip* alone, for traces of branch instruction records and *N* as small as 8.

In a trace that includes instruction words, one can replace the most frequent trace records with their identifiers in the dictionary (PDI, [Johnson, Ha and Zaidi 2001]). Another option is to replace a sequence of repeating trace records by the corresponding repetition count. Hamou-Lhadj and Lethbridge propose one such technique for traces of procedure calls [Hamou-Lhadj and Lethbridge 2002]: a trace is first preprocessed to replace repeating sequences of calls with a number of repetitions, and then it is represented as an ordered labeled tree, where nodes are procedure calls and tree levels correspond to nesting levels. In a third phase the tree is transformed into a dynamic acyclic graph.

QPT is a tracing and compression technique that records only information about significant events [Larus 1993]. A QPT trace includes only transitions between basic blocks where a program chooses between alternative paths, and only those transitions that are not part of a maximum spanning tree of a control flow graph. The actual instruction addresses can be regenerated using the control flow graph and a trace of transitions.

Traces including data information are often full address traces, i.e., instruction and data address traces. The simplest way to reduce the size of an address trace is to replace an address with the offset from the last address of the same type (instruction reference, data read, or data write reference) [Samples 1989]. This single-pass algorithm is called Mache. The *packed differential address and time stamp* (PDATS) algorithm takes the Mache approach one step further [Johnson, Ha and Zaidi 2001]. PDATS also stores address offsets between successive references of the same type, but the records in the trace of offsets can have variable lengths, specified in a one-byte record header, and an optional repetition count. For full address traces including instruction words, PDATS can be combined with a dictionary approach into PDI. PDATS and Mache have very small compression/decompression overhead, but they do not take into account the underlying structure of the executed program.

Luo and John propose *locality-based online trace compression* (LBTC) that targets more complex trace records [Luo and John 2004]. Information about an executed instruction is kept in a small direct-mapped compression cache. A compressed trace record consists of a record type (instruction or data), a cache hit/miss flag, and possibly an offset from the previous record of the same type. Other fields of the original trace record, such as instruction word or virtual address, are emitted to the compressed trace only in the case of a cache miss. This technique keeps only the last data address together with information about the corresponding memory referencing instruction. If a data address is repeated, a cache-hit flag is emitted; otherwise, the compressed trace includes the offset from the previous data address in the trace, similar to the PDATS technique.

Another approach is to link information about the data addresses with a corresponding loop [Elnozahy 1999]. In the first pass, loops in the trace are detected using a control flow analysis technique. In the second pass, data address references inside each loop are classified as chaotic, constant, or loop varying, i.e., with a constant offset between loop iterations. Constant and loop-varying addresses need to be encoded only once in the compressed trace, but all chaotic addresses must be stored separately. Control flow analysis to extract loop information can be avoided if a program is instrumented before tracing [DeRose et al. 2002]. However, the limitation of this technique is that the iteration count for inner loops must be constant.

Information about data addresses can also be linked to an instruction block [Pleszkun 1994]. For each memory-referencing instruction in an instruction block, the possible data offsets and numbers of repetitions are recorded. This technique may have very large memory requirements, since information about all possible data address offsets for one load or store instruction is kept in a linked list. However, for data references without a fixed offset or with a large number of different offsets, this approach may not be feasible. Pleszkun also proposes compression for the instruction component: an intermediate compressed trace encompasses the identifiers of successors for each instruction block and the number of repetitions. During a second pass, static basic blocks are fused into larger, dynamic basic blocks.

Data address traces can be regenerated by using so-called *abstract execution* [Larus 1993]. For each data address, a set of instructions computing that address is identified, and these instructions are re-executed during trace regeneration. Instructions are classified as easy, calculating a constant value; hard, computing a value that depends on previously computed values; and impossible, with dependencies too complex to recalculate. Instruction addresses are regenerated using the QPT technique. Abstract execution requires access to the source code, and it generates a significant overhead during trace decompression. A similar technique is used for tracing on a shared memory multiprocessor [Eggers et al. 1990].

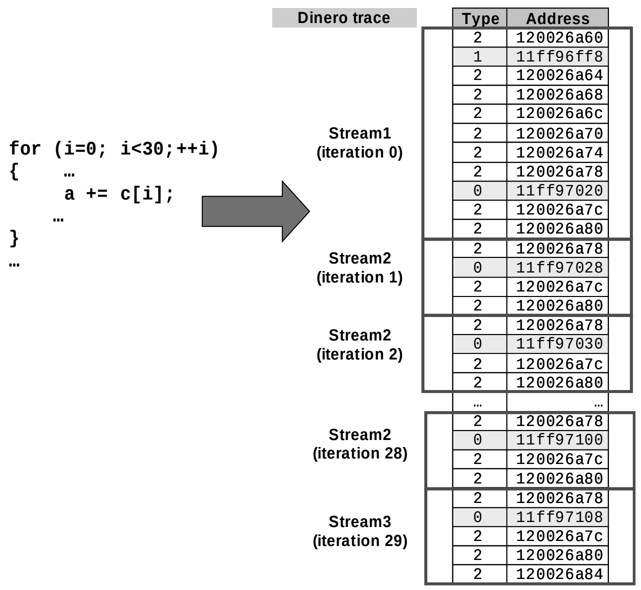
Traces can also be regenerated using *value predictors* (VPC [Burtscher and Jeeradit 2003], VPC3 [Burtscher 2004] [Burtscher et al. 2005], and TCGEN [Burtscher and Sam 2005]). The latest implementation, TCGEN, automatically generates optimized compression and decompression code, featuring a configurable set of value predictors for each component of a trace record. During compression, each component of a trace record is fed to a separate set of value predictors, indexed by the instruction addresses. Each trace component is compressed into two subtraces. If one of the predictors in a set is able to predict the component value, the identifier of that predictor is written to the value predictor code subtrace. If the value is mispredicted by all predictors, it is written into the mispredicted value subtrace, and a reserved code is written into the predictor code subtrace. Each subtrace is further compressed using *bzip2*. This technique has a very good compression ratio, but it requires a relatively long decompression time.

3. STREAM-BASED COMPRESSION ALGORITHM

The Stream-Based Compression algorithm (SBC) bridges the gap between simple trace compression algorithms and techniques requiring several passes, complex code analysis, and vast memory resources. SBC exploits inherent characteristics of instruction and data components in program execution traces. Instruction traces consist of a fairly limited number of different instruction streams [Milenkovic and Milenkovic

2003a] [Milenkovic and Milenkovic 2003b], and a large number of memory references exhibit strong spatial and/or temporal locality: for example, a load instruction having a constant address stride across loop iterations.

We demonstrate SBC on Dinero traces, although it is applicable to any address or extended address trace format. A Dinero trace record has two fixed-length fields: the header field (0 – data read, 1 – data write, and 2 – instruction read) and the address field. Fig. 2 shows a short excerpt from a Dinero trace in which stream 1 is followed by 28 executions of stream 2 and one execution of stream 3; this trace segment is used in explaining the SBC compression and decompression.



```
for (i=0; i<30;++i)
{    …
     a += c[i];
    …
}
…
```

Type	Address
2	120026a60
1	11ff96ff8
2	120026a64
2	120026a68
2	120026a6c
2	120026a70
2	120026a74
2	120026a78
0	11ff97020
2	120026a7c
2	120026a80
2	120026a78
0	11ff97028
2	120026a7c
2	120026a80
2	120026a78
0	11ff97030
2	120026a7c
2	120026a80
…	…
2	120026a78
0	11ff97100
2	120026a7c
2	120026a80
2	120026a78
0	11ff97108
2	120026a7c
2	120026a80
2	120026a84

Fig. 2. An example of instruction streams in a Dinero trace. Clear rows represent instruction addresses (type = 2) and shaded rows represent data addresses (type = 0 for reads and type = 1 for writes).

3.1 SBC Compression

The compression flow is illustrated in Fig. 3. The SBC input is a Dinero trace, and the output consists of three files: *stream table file* (STF), *stream-based instruction trace* (SBIT), and *stream-based data trace* (SBDT). The instruction types (IT) of a currently processed instruction stream are buffered in the IBuffer, and the corresponding data addresses (DA) in the DBuffer. An IT can be load (IT = 0), store (IT = 1), or non-memory referencing (IT = 2). The starting address of the current stream is kept in the S.SA variable, and the S.L variable keeps the current length of the stream. When the program flow changes -- that is, when the first instruction of the following stream is identified -- the information about current stream is processed. The SBC algorithm first determines the index of the current instruction stream, and then it processes data addresses from the DBuffer.

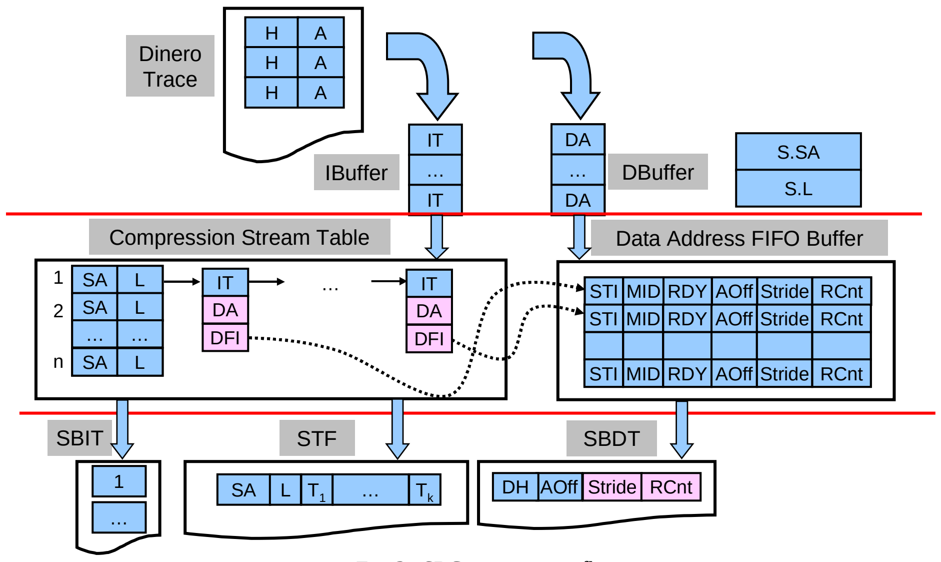


Fig. 3. SBC compression flow.

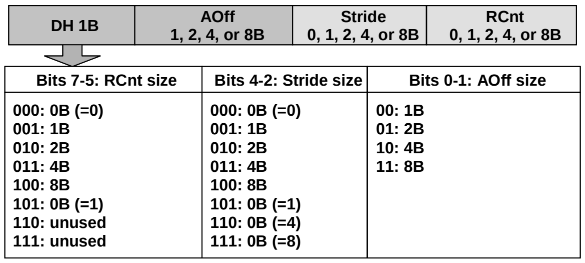
The compression stream table, residing in memory, is searched for an entry with the matching stream starting address and length. A simple hash table can be used to speed up the search. For each stream, this table keeps the starting address (SA), length (L), and a linked list of ITs (Fig. 3). A node in the list that corresponds to a load or a store instruction also has the DA and a pointer to the corresponding data address FIFO buffer entry (DFI). If there is no match in the compression stream table, a new stream entry is allocated, its linked list is filled with information from the IBuffer, and the corresponding information is also written into the STF file. The stream index in the compression stream table is written in the SBIT file, so the whole stream of instructions is replaced by its index. Finally, S.SA is set to the current instruction address -- the start of a new stream -- and S.L is set to 1.

The data address FIFO buffer and DA fields in the compression stream table are updated for each load/store instruction in the currently processed stream. One entry in the data address FIFO buffer holds information about one memory-referencing instruction in a stream. It consists of the following fields: *stream table index* (STI), *memory reference index* inside the stream (MID), *ready flag* (RDY), *address offset* (AOff), *data stride* (Stride), and *repetition count* (RCnt). The STI points to the corresponding compression stream table entry, and RDY indicates that the FIFO entry is ready to be flushed.

The data address FIFO buffer is updated as follows. If a DFI field for a *k*-th memory-referencing instruction in the stream is valid, the SBC algorithm calculates the difference between the *k*-th DA field in the compression stream table (previous data address for that instruction) and the DA in the input data address buffer DBuffer (current data address). This value is the current stride. If the current stride is equal to the value of the stride field in the data address FIFO entry determined by the DFI, the RCnt is incremented. If RCnt = 0, this is the second execution of that instruction, so the stride field is set to the current stride value. If the stride has changed and RCnt is not 0, the RDY is set to 1 and a new entry is added to the data address FIFO buffer. A new entry is also added if the DFI in the compression stream table entry has an invalid value, which is always the case with instructions in a new stream.

Before adding a new data address FIFO buffer entry, the SBC algorithm first verifies if the FIFO is full. If yes, the oldest FIFO record is written to the SBDT. If a FIFO record with RDY = 0 has to be thrown out of the buffer, the corresponding DFI value in the compression stream table is set to invalid. In a new data address FIFO buffer entry, STI and MID are set to the values of the compression stream table index and memory reference index, AOff is set to the current stride value, and the RCnt and RDY fields are set to 0.

A record in the SBDT file contains information about address offset, stride, and repetition count, and it can have variable length and a variable number of fields. The data header field (DH) encodes the length and the most frequent values of other fields (Fig. 4), 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 data header; and the proposed format allows the variable length of the AOff (1, 2, 4, or 8 bytes), Stride, and RCnt fields (0, 1, 2, 4, or 8 bytes). For example, the store instruction in stream 1 needs 8 bytes for AOff, while RCnt and Stride are equal to 0 and are encoded in the data header. The load instruction in stream 2 has the repetition count equal to 0x1b; thus it needs one byte for the RCnt field. The stride is equal to 8, so it is encoded in the data header. The content of SBC components for the trace in Fig. 2 is shown in Fig. 5.

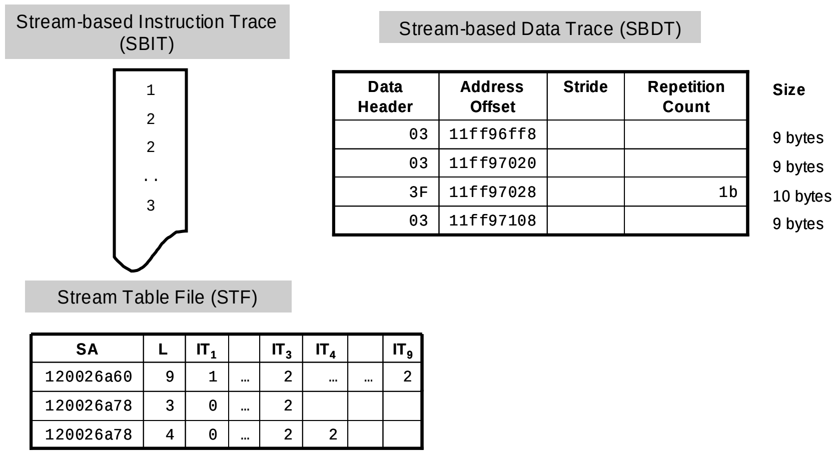


DH 1B	AOff 1, 2, 4, or 8B	Stride 0, 1, 2, 4, or 8B	RCnt 0, 1, 2, 4, or 8B

Bits 7-5: RCnt size	Bits 4-2: Stride size	Bits 0-1: AOff size
000: 0B (=0)	000: 0B (=0)	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: 0B (=1)	101: 0B (=1)	
110: unused	110: 0B (=4)	
111: unused	111: 0B (=8)	

Fig. 4. Format of the SBC data trace.

The SBC components can be further compressed using some of the general-purpose compression algorithms. The SBC records can be piped to a general-purpose compression process, reducing time and memory requirements needed for further compression. One can ask why the SBC algorithm does not exploit the repetition of instruction streams. Such patterns in SBIT are easily recognized by *gzip*/*bzip2*, without an increase in complexity of SBC or any restrictions considering the number and nature of nested loops.



Stream-based Instruction Trace (SBIT)
1
2
2
. .
3
Stream-based Data Trace (SBDT)
Stream Table File (STF)

Data Header	Address Offset	Stride	Repetition Count	Size
03	11ff96ff8			9 bytes
03	11ff97020			9 bytes
3F	11ff97028		1b	10 bytes
03	11ff97108			9 bytes

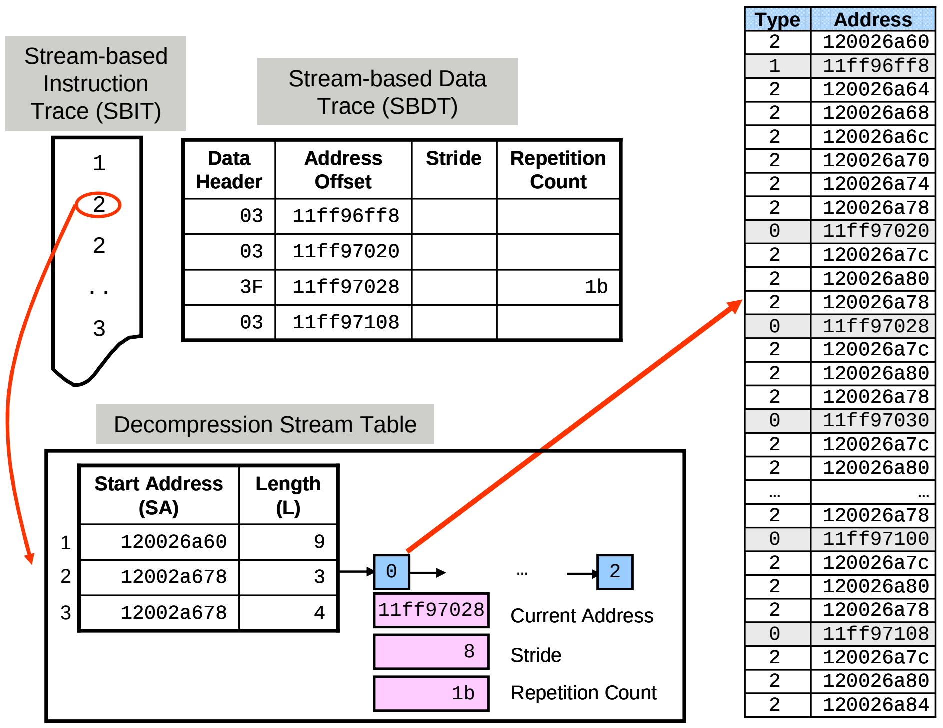
SA	L	IT_1		IT_3	IT_4		IT_9
120026a60	9	1	…	2	…	…	2
120026a78	3	0	…	2			
120026a78	4	0	…	2	2		

Fig. 5. An example of SBC trace components.

3.2 SBC Decompression

At the beginning of decompression, the whole STF file is loaded into the decompression stream table (Fig. 6). Like in the compression stream table, an entry in this table encompasses the starting address (SA) and length (L) fields, and the pointer to the list of stream instruction types. Each node in the list that corresponds to a load or a store instruction has the following fields: current data address, address stride, and repetition count. These fields are initialized to zero and dynamically updated from the SBDT file during trace decompression, whenever the repetition count of an accessed node is zero.

Decompression proceeds as follows: a stream index is read from SBIT, e.g., a stream index 1. Entry 1 of the decompression stream table is accessed, giving address 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 SBDT. The current data address in the node is calculated as the previous current address (0) plus the AOff field from SBDT; the stride is set to the value of the Stride field, in this case 0; and the repetition count is set to the RCnt value, again 0 since this stream executes only once. The pointer to the current instruction then moves along the stream instruction type list until all nine instructions are read. Each instruction address is obtained by incrementing the current instruction address for the value of the instruction length, beginning from the starting address field. The SBDT file is accessed once more, for the seventh instruction in stream 1, a load instruction.


Stream-based Instruction Trace (SBIT)
1
2
2
..
3
Stream-based Data Trace (SBDT)
Decompression Stream Table
0
...
2
Current Address
Stride
Repetition Count
11ff97028
8
1b

Data Header	Address Offset	Stride	Repetition Count
03	11ff96ff8		
03	11ff97020		
3F	11ff97028		1b
03	11ff97108		

	Start Address (SA)	Length (L)
1	120026a60	9
2	12002a678	3
3	12002a678	4

Type	Address
2	120026a60
1	11ff96ff8
2	120026a64
2	120026a68
2	120026a6c
2	120026a70
2	120026a74
2	120026a78
0	11ff97020
2	120026a7c
2	120026a80
2	120026a78
0	11ff97028
2	120026a7c
2	120026a80
2	120026a78
0	11ff97030
2	120026a7c
2	120026a80
…	…
2	120026a78
0	11ff97100
2	120026a7c
2	120026a80
2	120026a78
0	11ff97108
2	120026a7c
2	120026a80
2	120026a84

Fig. 6. An example of SBC decompression.

The next stream index read from the SBIT file is 2, so entry 2 is accessed. The first instruction is a load, so the corresponding node information 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 (0x1b). When stream 2 is again encountered in the SBIT file and its load instruction is read from the decompression stream table, there is no need to access the SBDT file – the load address is calculated as the previous address plus the stride, and the repetition count is decremented. The same process is repeated for all 26 remaining executions of stream 2.

3.3 SBC Compression in Hardware

Virtually all trace compression techniques target compression in software. On the other hand, computer systems could greatly benefit from hardware support for trace collection and compression, especially emerging systems-on-a-chip with multiple embedded RISC and DSP processor cores. For example, ARM already offers a module for tracing the complete pipeline information [ARM 2004]. However, the existing compression techniques that can be efficiently implemented in hardware have poor compression ratio: e.g., an ARM emulator compresses ARM traces by replacing the sequence of the same records by their repetition count [McCullough et al. 2003].

The SBC implementation described in the previous sections cannot be directly implemented in hardware: although the SBC algorithm uses a FIFO buffer of limited size for data address compression, it keeps information about instruction streams in an unbounded stream table. However, the SBC algorithm can be modified to use fixed-size resources for stream information: instead of the stream table, it can use a stream cache resource that has a fixed number of entries. The modified SBC, SBChw, generates three trace components: a stream cache hit component and a stream cache miss component for instruction addresses, and a stream-based data trace for data addresses. In the case of a stream cache hit, only the corresponding cache index is emitted into the stream cache index trace, similar to the stream index in the basic SBC. In the case of a stream cache miss, an instruction stream descriptor (encompassing the stream starting address, stream length, and instruction types) is emitted into the stream cache miss trace, and a reserved index is emitted into the stream cache index trace.

To make SBC suitable for hardware implementation, we also need to limit the maximal stream length and the maximal number of memory-referencing instructions in a stream. Hence, we introduce two new conditions for stream termination: in addition to a change in control flow, an instruction stream also ends when it reaches the maximal allowed length or the maximal number of memory-referencing instructions.

Fig. 7 shows an implementation of the stream cache and the corresponding data address FIFO buffer. The stream cache has N_{WAY} ways and N_{SET} sets. 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 concatenation. The S.SA and S.L serve as tags in the stream cache. Similar to the stream table entry, a stream cache entry also includes an IT field for each instruction in the corresponding stream, and the data address (DA) and the data address FIFO buffer index fields (DFI) for each memory-referencing instruction in the stream.

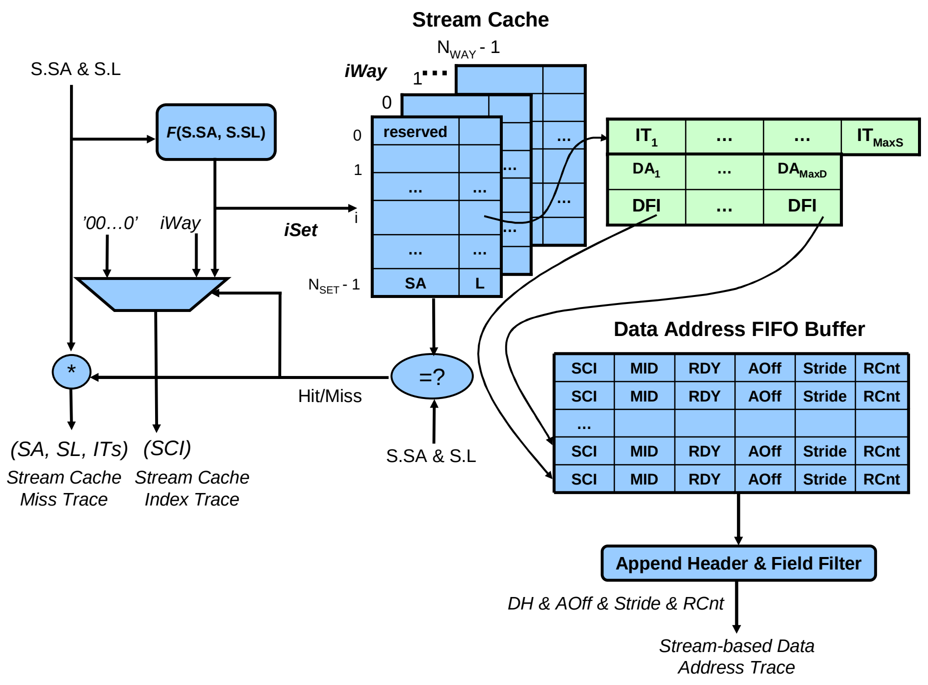


Fig. 7. Stream Cache and Data Address FIFO Buffer in SBC hardware implementation.

4. RESULTS

In this section we first describe our evaluation environment and determine the feasibility of the SBC technique using stream statistics. For all evaluated techniques, we compare the trace compression ratio, compression and decompression times, and simulation speedup. We also evaluate the compression ratio of SBC modified for hardware implementation. Finally, we discuss the compressibility of instruction and data components.

4.1 Evaluation Environment and Stream Statistics

In order to evaluate the proposed compression mechanism, we use Dinero address traces of 10 SPEC CPU2000 integer (INT) and 13 floating-point (FP) benchmarks (Table I). We trace two segments for each benchmark: the first billion instructions (F1B) and a billion instructions after skipping 50 billion (M1B), thus making sure that the results do not overemphasize program initialization. Traces are generated using a modified SimpleScalar environment [Burger and Austin 1997], precompiled Alpha binaries, and SPEC CPU2000 reference inputs. Although the number of instructions in each segment is fixed, the number of memory-referencing instructions – loads and stores – varies greatly across benchmarks, from as low as 18.8% of all instructions for F1B segment of *189.lucas*, to over 60% for M1B segment of *176.gcc*. The average segment size of Dinero traces is about 11GB, and the sum of uncompressed traces reaches 510GB, clearly showing the need for highly efficient compression techniques.

Since the compression and decompression stream tables are stored in memory, we first verify that the number of unique instruction streams is indeed fairly limited. The size of these tables depends on the number of unique streams and the number of instructions in each stream. Trace analysis shows that all FP benchmarks have fewer than 4,000 unique instruction streams in a segment, and all INT benchmarks except *176.gcc* and *253.perlbmk* have fewer than 6,000 unique streams (Table I). When complete execution of benchmarks is considered, the average number of unique streams is 7,686 for INT and 2,568 for FP applications, with a maximum 30,162 in *176.gcc*. The average stream length is less than 30 instructions for all INT and 5 FP benchmarks, and it reaches the maximum for *173.applu*, 449 instructions in the M2B segment, but in that segment *173.applu* has only 502 unique streams. All these results indicate that the stream table has relatively modest memory requirements.

Table I. Stream statistics for the first billion instructions (F1B), for the billion after skipping 50 billion (M1B), and for complete program execution (All)

	# of Streams			MaxStreamLen			AverageStreamLen		
INT	F1B	M1B	All	F1B	M1B	All	F1B	M1B	All
164.gzip	744	315	1437	100	90	229	14.1	13.7	13.6
176.gcc	25251	1518	30162	272	103	315	10.3	11.6	11.4
181.mcf	480	306	1181	88	64	88	9.4	8.2	7.4
186.crafty	4067	1802	5347	191	100	191	13.0	13.4	13.3
197.parser	3436	4176	6116	130	157	189	8.9	10.0	10.0
252.eon	3468	574	4389	169	168	169	13.8	14.1	13.7
253.perlbmk	9031	2738	11542	84	84	868	10.0	12.3	11.8
254.gap	3154	473	3530	284	75	284	25.0	10.3	11.1
255.vortex	5441	1615	8254	126	110	126	11.1	11.2	11.0
300.twolf	2235	1013	4902	163	185	185	10.7	14.5	14.4
Average	5730.7	1453	7686.0	160.7	113.6	264.4	12.6	11.9	11.8
FP									
168.wupwise	1389	185	1912	131	229	229	23.9	27.1	27.4
171.swim	1581	493	1839	707	707	707	93.6	146.3	130.8
172.mgrid	1456	871	1725	1944	1944	1944	240.1	94.1	420.8
173.applu	1463	502	1752	3162	3162	3162	411.5	449.1	462.4
177.mesa	1625	583	1938	550	266	550	14.8	18.5	18.15
178.galgel	1729	30	4153	264	111	264	18.4	21.8	21.8
179.art	435	75	976	168	561	561	10.3	8.7	9.0
183.equake	517	256	1355	44	623	623	8.6	28.3	27.7
188.ammp	818	71	1810	84	395	422	12.5	36.1	38.5
189.lucas	825	78	1414	101	427	427	27.1	128.7	113.3
191.fma3d	2069	840	5007	383	1158	1158	10.7	41.5	34.3
200.sixtrack	3507	81	6515	264	580	580	20.1	192.9	170.5
301.appsi	2388	303	2989	729	729	894	34.0	47.4	50.7
Average	1523.2	336.0	2568.1	656.2	837.8	886.2	71.2	95.4	117.3

4.2 SBC Compression Ratio

We compare compression ratio for SBC, PDATS, TCGEN, and LBTC compression techniques when these techniques are used alone, and when they are combined with *gzip* (SBC.gz, PDATS.gz, TCGEN.gz, and LBTC.gz) and *bzip2* (SBC.bz2, PDATS.bz2, TCGEN.bz2, and LBTC.bz2). We also evaluate the use of *gzip* (Dinero.gz) and *bzip2* (Dinero.bz2) alone.

For each compression technique and each selected benchmark, the compression ratio is calculated as the sum of the sizes of uncompressed Dinero traces in F1B and M1B segments, divided by the sum of the sizes of compressed traces. We also consider the total compression ratio, calculated as the sum of uncompressed traces of all benchmarks over the sum of all compressed traces.

To have a fair comparison, we used a Dinero trace format with separate instruction and data components. The size of a trace in this format is less than the size of a unified trace, since the type field is not required in the data component. In addition to that, split components can be better compressed by *gzip/bzip2*. PDATS and LBTC techniques are also slightly modified to have separate instruction and data components. TCGEN code is automatically generated by the TCgen tool [Burtscher and Sam 2005]; we modified the code to read the split Dinero files during compression, and not to write the decompressed files during decompression. TCGEN uses the combination of finite-context-method and differential-context-method predictors; its detailed specification is given in Fig. 8. The data address field exists only in load/store instructions. The FIFO buffer size for SBC is 8,192 entries.

Fig. 9 shows compression ratio results. SBC significantly outperforms all other techniques, with a total compression ratio of 35.9 compared to 7.2 for PDATS, 4.9 for TCGEN, and 7.6 for LBTC. SBC compression ratio varies across applications in a range from 18 to 308. It should be noted that SBC alone also outperforms *gzip* compression (Dinero.gz) for all benchmarks.

```
TCgen Trace Specification;
0-Bit Header;
# Instruction type
8-Bit Field 1 = {L1 = 65536, L2 = 4: FCM3[2]};
# Field 1 uses 2 predictors with a combined size of 0.8MB
# Instruction address
64-Bit Field 2 = {L1 = 1, L2 = 131072: FCM3[2]};
# Field 2 uses 2 predictors with a combined size of 8.0MB
# Data address
64-Bit Field 3 = {L1 = 65536, L2 = 131072: DFCM3[2], DFCM1[1],
FCM1[1]};
# Field 3 uses 4 predictors with a combined size 11.5MB
PC = Field 2;
```

Fig. 8. TCGEN specification.

Even better compression can be obtained by further compressing an SBC trace by *gzip*. The SBC.gz compression ratio is from 80 to 737 for INT and from 462 to 35,595 for FP. Again, the SBC.gz outperforms all other techniques: the sum of all traces is reduced 327 times, and the total compression ratio for other techniques is 271 for TCGEN, 85 for LBTC, and 62 for PDATS.

Using *bzip2* instead of *gzip* can even further improve the compression ratio, but at the price of a significant slowdown in compression/decompression time. SBC.bz2 compression ratio for the sum of all traces approaches 400. However, TCGEN.bz2 achieves an even better compression of 722.

It should be noted that Burtscher et al. find that TCGEN combined with bzip2 outperforms other compression techniques in both compression ratio and decompression time [Burtscher et al. 2005]. These seemingly contradictory results can be easily explained. The TCGEN authors do not use full address traces, but rather a subset with some additional information (e.g., traces of store instructions and store addresses, load instructions and load values, and addresses of instructions causing cache misses). Instruction streams cannot be easily recognized in such traces, so the authors assume that a “stream” in a trace is a sequence of trace records with increasing addresses, such that the difference between consecutive records’ addresses is less than a small threshold. These artificial streams are very short, so the full strength of SBC cannot be exploited. Slower SBC decompression time observed in [Burtscher et al. 2005] is a consequence of lower compression ratio and the lack of code optimization.

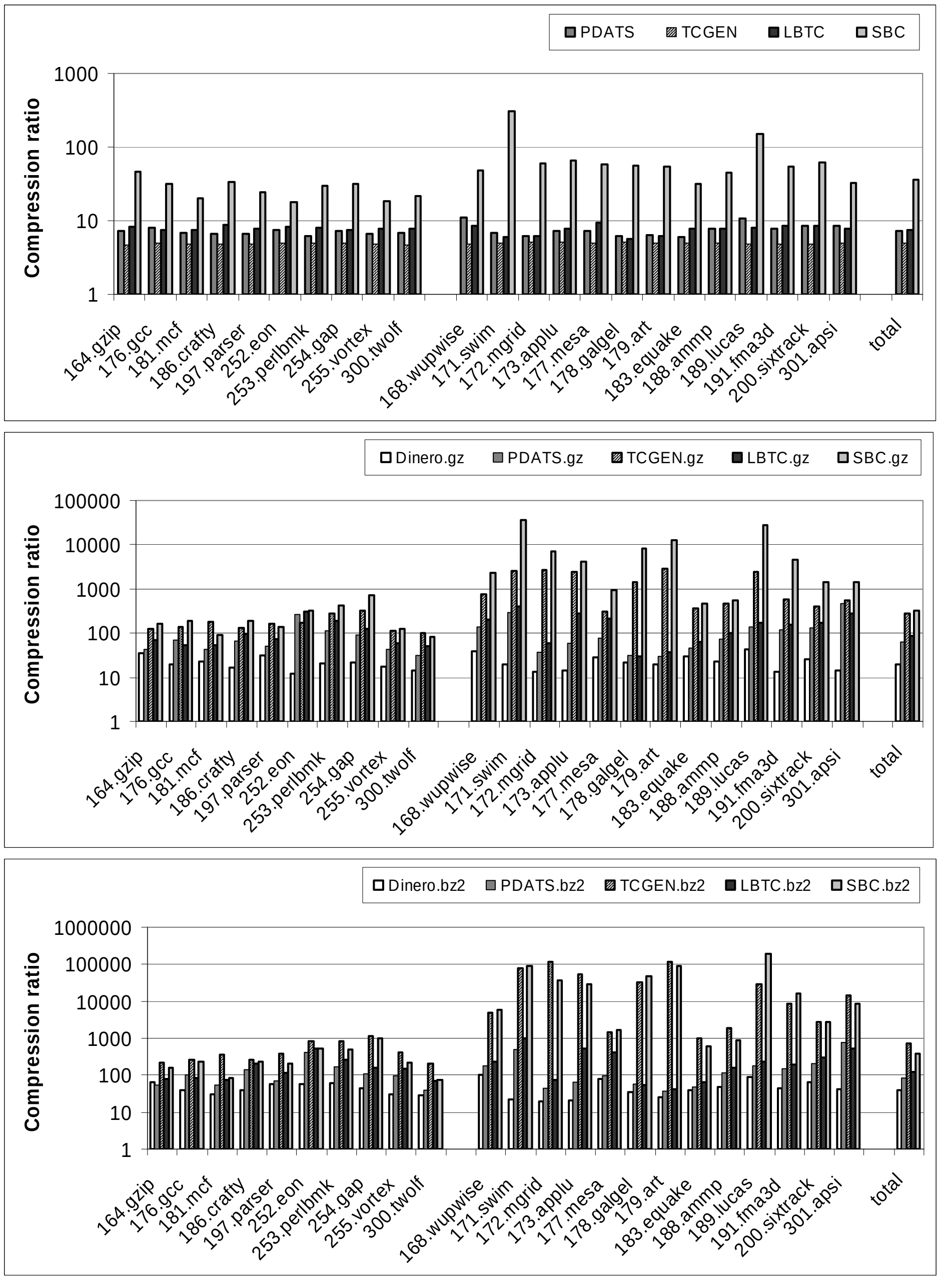


Fig. 9. Compression ratio.

4.3 SBC Decompression Time and Simulation Speedup

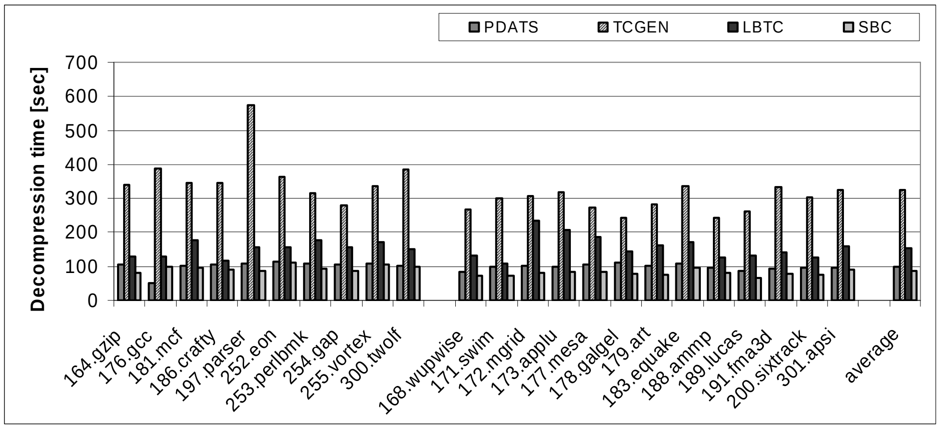
Decompression time is as important as compression ratio, especially when compressed traces are used for trace-driven simulation. We first measure real elapsed decompression time in a program that reconstructs an entire trace, with buffered reads and inlined decompression. The *gzip* and *bzip2* utilities run in a separate process and pipe their results to the main procedure. This program emulates a simulator adapted to the applied decompression technique. Total processor time needed for program execution is measured on a Pentium 4 with hyperthreading at 3 GHz, with the RedHat Linux operating system.

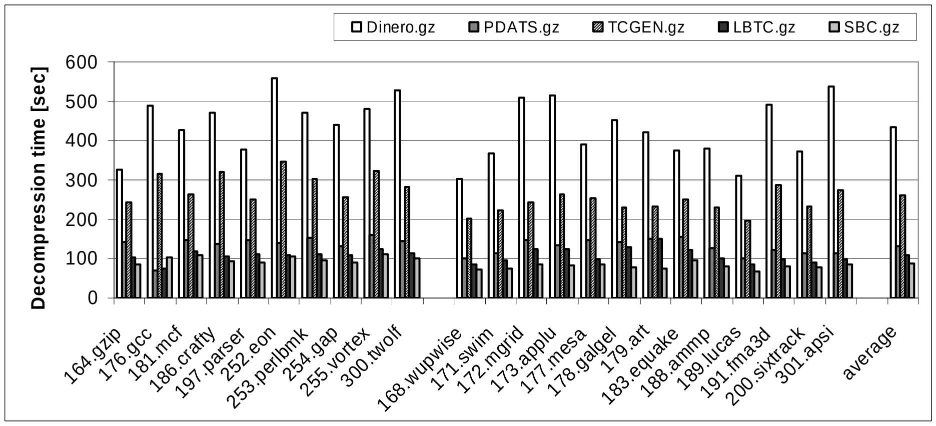
Fig. 10 shows decompression times for all considered techniques alone and when combined with *gzip* and *bzip2*. SBC performs better than other techniques, requiring on average 86 seconds per application; it is slightly better than PDATS, and it is almost two times faster than LBTC and four times faster than TCGEN. SBC also outperforms other techniques when it is combined with *gzip* and *bzip2*. For example, SBC.gz is on average 5 times faster than Dinero.gz, 1.5 times faster than PDATS.gz, 3 times faster than TCGEN.gz, and 1.25 times faster than LBTC.gz; SBC.bz2 decompression is almost 20 times faster than Dinero.bz2, 4.5 times faster than PDATS.bz2, 3 times faster than LBTC.bz2, and 2.5 times faster than TCGEN.bz2[1]. It should be noted that decompression time depends on both the compressed file size and time complexity of the decompression algorithm. Though *bzip2* always achieves better compression ratio than *gzip*, *bzip2* decompression time is longer.

[1] bz2 decompression times are shown with logarithmic scale.

One could argue that a TCGEN configuration with a smaller number of predictors should have faster decompression than the one we use, so we also evaluate a “small” TCGEN with only one predictor for each trace record component (the first predictors listed in Fig. 8). The smallTCGEN.gz indeed does decompress slightly faster than TCGEN.gz, on average 1770 vs. 1801 seconds. However, raw smallTCGEN and smallTCGEN.bz2 are slower than the corresponding TCGEN and TCGEN.bz2. A higher number of mispredictions, because of fewer predictors, significantly deteriorates the compression ratio: 4 vs. 5 for raw TCGEN, 209 vs. 271 for TCGEN.gz, and 495 vs. 722 for TCGEN.bz2.

SBC is also evaluated as a part of a trace input procedure for the DineroIV cache memory simulator [Edler and Hill 1998]. In DineroIV, a trace is read in a separate trace input procedure. The decompressed trace record is returned by value to the *next_in_trace* procedure, which than returns it to the main procedure. We implemented trace input procedures for all trace compression alternatives, with buffered read and pipes for *gzip* and *bzip2*. Fig. 11 shows the simulation speedup relative to the DineroIV simulator reading raw Dinero traces. SBC.gz outperforms all other techniques with an average simulation speedup of 2.6 (compared to 2.4 for SBC.bz2, 2.4 for LBTC.gz, 2.4 for PDATS.gz, 1.8 for TCGEN.gz, and 1.7 for TCGEN.bz2). Simulation speedup for DineroIV is lower than the decompression speedup shown in Fig. 10, because of simulation overhead and simulator implementation.





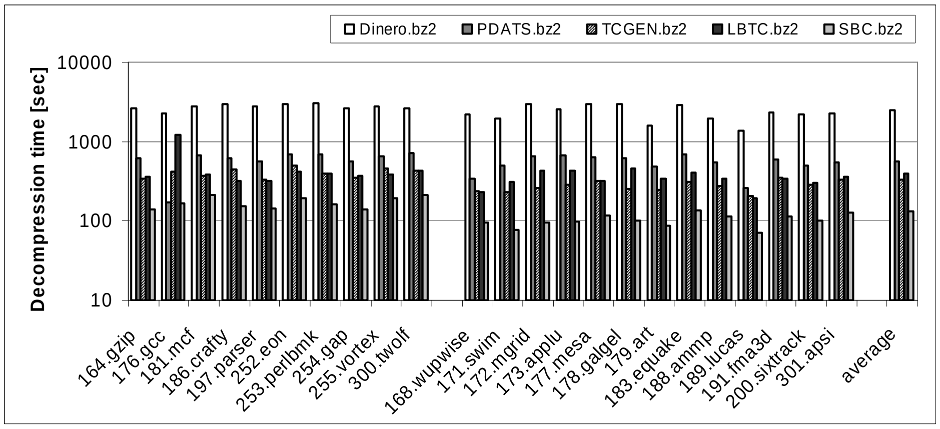


Fig. 10. Decompression time.

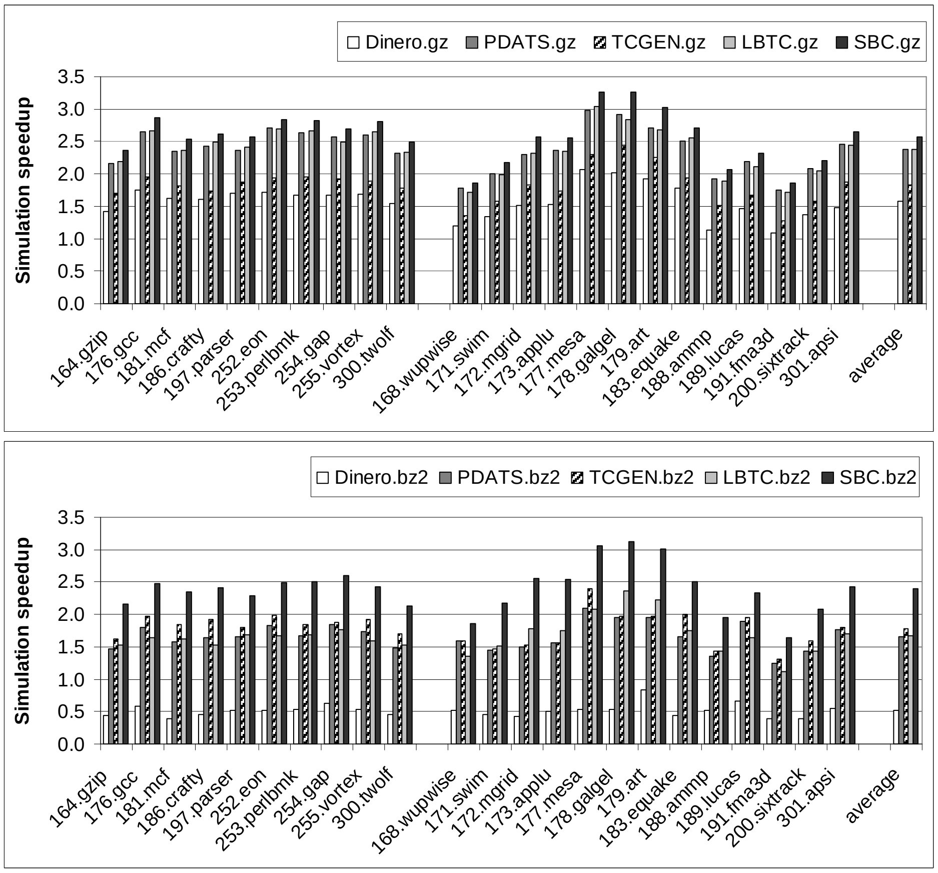


Fig. 11. Simulation speedup for the DineroIV simulator, relative to raw Dinero traces.

4.4 SBC Compression Time

Compression time is a less significant factor in the choice of a trace compression technique than is decompression time, since a trace is usually compressed only once but is decompressed many times. Without *gzip*/*bzip2* compression, compression time varies little across different techniques: the arithmetic mean for SBC is 28.9 minutes versus 29.1 minutes for PDATS, 33.3 for TCGEN, and 30.1 for LBTC (Fig. 12, top). The average total CPU time is 3 minutes for SBC, 3.77 for PDATS, 3.28 for LBTC, and

6.79 minutes for TCGEN. Similar real time can be explained as follows: a compression program spends its execution time on three components -- reading an uncompressed file, compressing trace records, and writing the compressed file. Hence, any compression program is I/O bound, spending most of its time waiting for I/O. All programs use the same input, so the time for the file read does not vary much. In this case, this component dominates others: compressed files are about an order of magnitude smaller than uncompressed ones, and total CPU time is also significantly less than I/O time. SBC compression has the longest duration for *gcc*, an application with many different streams.

Gzip compression alone lasts longer than any of the trace compression techniques combined with *gzip*, since the *gzip* output files are larger. SBC.gz, PDATS.gz, LBTC.gz, and TCGEN.gz all have an average compression time of about 29 minutes (Fig. 12, middle). The *bzip2* compression has a higher time complexity than linear, so it is very slow for large output files. Dinero.bz2 has an average compression time of about 708 minutes, SBC.bz2 and TCGEN.bz2 about 38 minutes, and PDATS.bz2 and LBTC.bz2 about 88 minutes (Fig. 12, bottom)[2].

[2] bz2 compression times are shown with logarithmic scale.

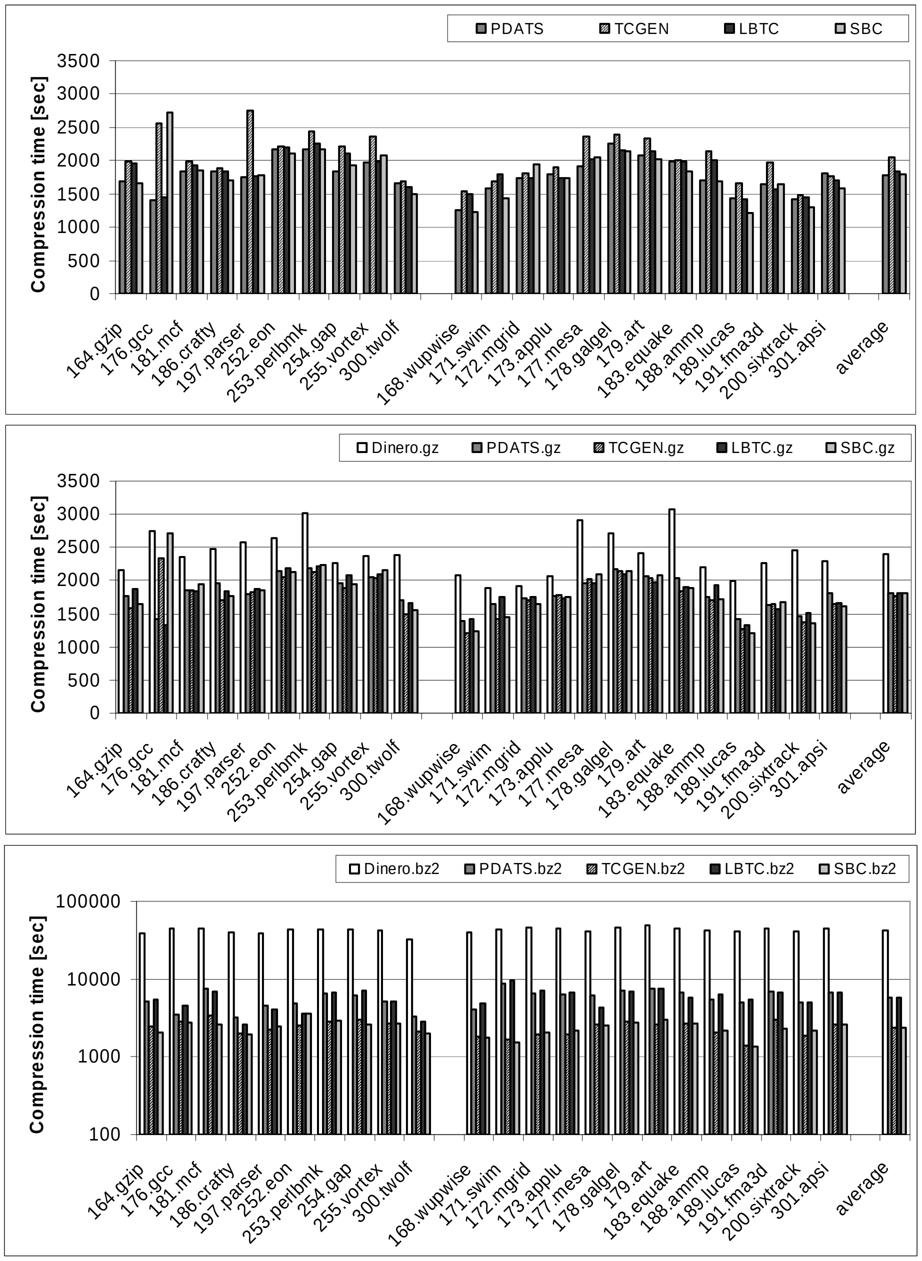


Fig. 12. Compression time.

4.5 Compression Ratio of SBC Hardware Implementation

To evaluate the influence of finite-size resources on the SBC algorithm, we compare the size of traces compressed with SBC and SBChw, without the additional general compression stage (Fig. 13). Two SBChw configurations are considered: an extremely small SBChw (64x4), with only 256 stream cache entries, and a relatively large SBChw (4Kx4), with 16K stream cache entries (Table II). For the majority of applications, SBChw (4Kx4) has almost the same compression ratio as SBC. The slightly higher total compression ratio of SBChw (4Kx4) is due to a smaller compressed data address trace component (SBDT) for most floating-point applications. This result can be explained as follows. With the SBC, the body of an inner loop appears as a part of three streams, so each load/store instruction in the loop has at least three data address FIFO entries and three trace records in the SBDT file. With the SBC hardware implementation, very long instruction streams are split into shorter ones, so some of the load/store instructions in a long loop body will belong to only one SBChw stream. Consequently, load/store instructions in relatively long loops in SPEC CPU2000 FP applications are compressed into fewer trace records with SBChw (4Kx4) than with SBC.

Table II. Parameters of SBC hardware implementation

SBChw	Data address FIFO entries	Stream cache sets	Stream cache ways	Max stream length	Max loads&stores per stream
(4Kx4)	8192	4096	4	256	128
(64x4)	4096	64	4	128	64

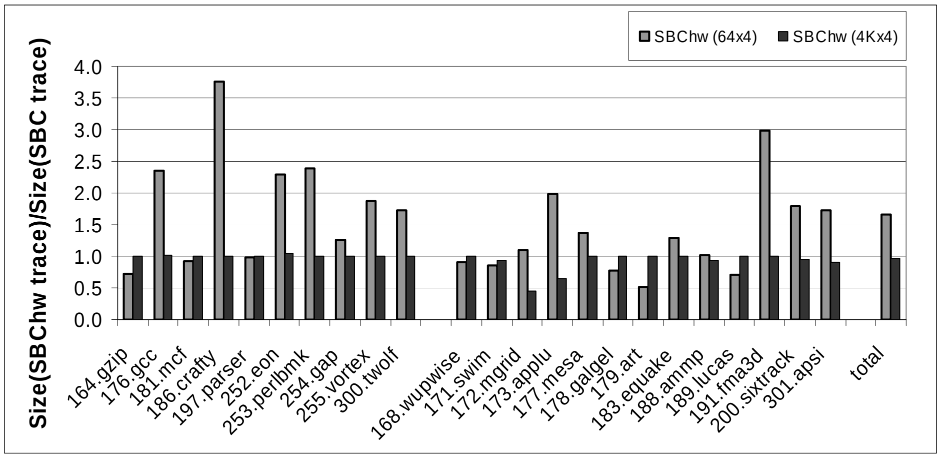


Fig. 13 SBChw versus SBC trace sizes.

For most applications, even SBChw (64x4) comes close to SBC, with 13 out of 23 traces being less than 1.4 times larger, and only 2 traces being more than 2.5 times larger. Although with a very small stream cache, SBChw (64x4) can even outperform SBC, because of the effect explained above and a shorter StreamID (one byte instead of two bytes for stream index). The total size of SBChw traces divided by the total size of SBC traces is 1.66 for SBChw (64x4) and 0.97 for SBChw (4Kx4).

4.6 Compressibility of Instruction and Data Trace Components

Instruction addresses are known to have more redundancy than data addresses have [Becker and Park 1993]. Fig. 14 and Fig. 15 illustrate how successful the evaluated trace compression techniques are in exploiting this redundancy. The SBC algorithm compresses instruction addresses much better than data addresses, for up to two orders of magnitude. Using again the total compression ratio as a metric, SBC instruction address compression is 5.3 times better than data address compression. A trace of stream identifiers has more redundancy than a trace of SBC data records, so SBC.gz and SBC.bz2 compress the instruction trace component 40.6 and 90.8 times better than they compress the data address component, respectively.

TCGEN replaces each trace component by a predictor identifier and possibly with misprediction information. Hence, the size of the instruction type component is actually expanded. As a consequence, TCGEN has a lower compression ratio for the instruction component than for the data component. TCGEN traces are very suitable for additional compression by *gzip*/*bzip2*, since traces of predictor identifiers have very few unique symbols. Second stage compression with *gzip*/*bzip2* benefits more from instruction address redundancy, so TCGEN combined with *gzip* compresses the instruction component 4.1 times better than it compresses the data component, and about 18.4 times better when combined with *bzip2*.

With LBTC, both instruction and data addresses are compressed using offsets, and only repeated data addresses benefit from the compression cache. Hence, for most applications there is not much difference between instruction and data component compression ratio. Instruction address offsets are better compressed with general compression algorithms than data address offsets are, so LBTC.gz and LBTC.bz2 compress the instruction component 16.3 and 50.7 times better than they compress the data component, respectively. PDATS is 2.9 times better for instruction than for data, due to encoding of repeated instruction offsets.

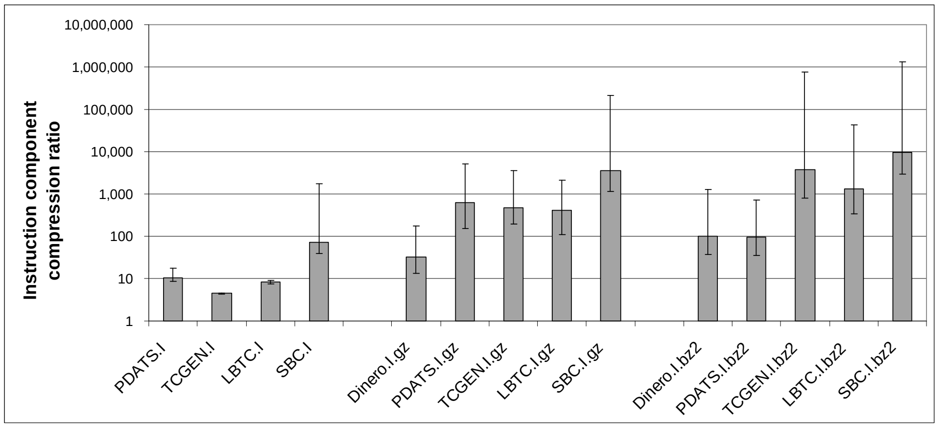


Fig. 14. Compression ratio of Dinero trace instruction component.

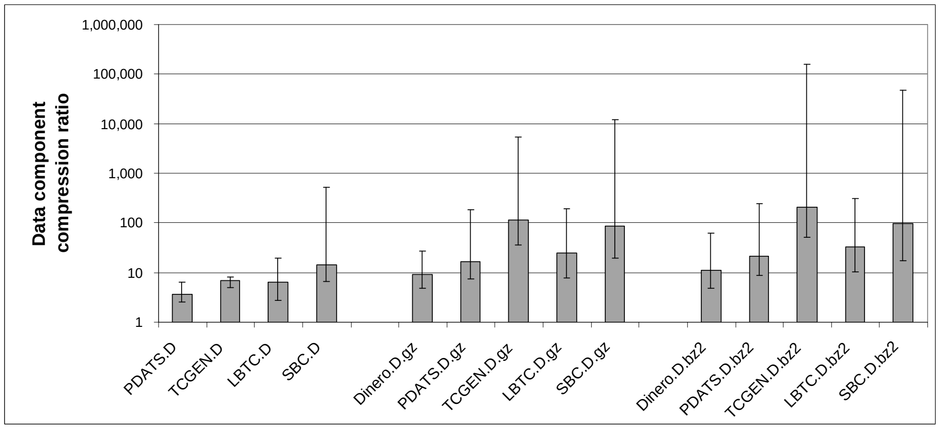


Fig. 15. Compression ratio of Dinero trace data component.

SBC compression of the instruction trace component depends only on the average stream length, which indicates how many instruction records in the original trace are replaced by one stream identifier in the SBC-compressed trace (Table I). Hence, traces with longer instruction streams are better compressed. The SBC compression ratio of the data component is more difficult to explain, since it depends on several factors: the regularity of data address strides; and the length of repetition, stride, and address offset fields in a particular SBDT file. The finite size of the Data FIFO Buffer can also affect SBC data component compression. Table III shows the ratio of the number of memory references in the trace, N_{MEM_D}, and the number of records in the SBDT trace, N_{SBDT}. The results indicate that a moderate FIFO size, such as 8,192 entries, yields a compression ratio close to that of an infinite FIFO for most benchmarks. Fig. 16 shows how the compression ratio of the data component depends on the FIFO size for *301.appsi*, the most sensitive application, and *189.lucas*, the least sensitive one. We can emulate an infinite FIFO, but that would require two passes through the trace. Instead, we conduct additional experiments with the 64K-entry data address FIFO buffer. Although the compression ratio significantly improves for some applications (e.g., 301.appsi, 191.fma3d, 173.applu), the total compression ratio does not considerably change: 36 for raw SBC with 8K FIFO vs. 37 with 64K FIFO, 326 vs. 346 for SBC.gz, and 390 vs. 406 for SBC.bz2. These gains in compression ratio are achieved without additional overhead in compression and decompression time.

Table III. Ratio of the number of data address trace references N_{MEM_D} and SBDT records N_{SBDT}

	N_{MEM_D}/N_{SBDT}	
INT	8192 entry FIFO	Infinite FIFO
164.gzip	12.21	12.78
176.gcc	20.88	24.63
181.mcf	10.62	11.66
186.crafty	5.43	7.78
197.parser	6.00	6.33
252.eon	2.94	4.05
253.perlbmk	6.93	8.36
254.gap	5.23	5.79
255.vortex	3.23	3.82
300.twolf	4.66	5.72

	N_{MEM_D}/N_{SBDT}	
FP	8192 entry FIFO	Infinite FIFO
168.wupwise	26.77	27.72
171.swim	95.81	102.72
172.mgrid	9.92	12.48
173.applu	6.23	15.18
177.mesa	13.90	19.27
178.galgel	27.40	29.15
179.art	373.22	502.32
183.equake	10.56	11.66
188.ammp	14.87	17.38
189.lucas	91.14	92.01
191.fma3d	12.62	21.38
200.sixtrack	8.55	11.70
301.appsi	3.26	6.86

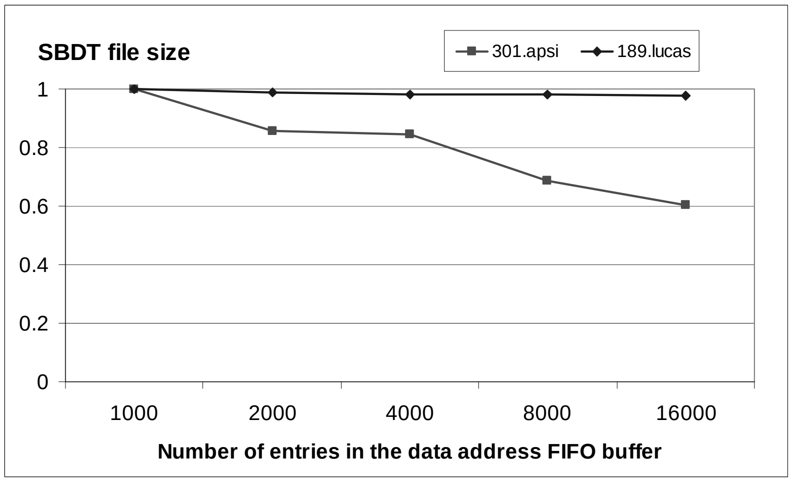


Fig. 16. SBDT file size as a function of the number of entries in the data address FIFO buffer, relative to the 1K-entry FIFO buffer.

5. CONCLUSION

The main contributions of this paper are the following:

- A single-pass trace compression technique, SBC. In the SBC algorithm an instruction stream is replaced by its index from the stream table. Data addresses are linked to a corresponding instruction stream and compressed using an efficient on-line algorithm that recognizes regular strides. This technique achieves an optimal balance between the most important compression requirements: high compression ratio and fast decompression time. The SBC web site, http://www.ece.uah.edu/~lacasa/sbc/sbc.html, includes SBC-compressed SPEC CPU2000 traces and utility programs.
- A modified SBC with fixed-size resources, SBChw, suitable for hardware implementation. The SBChw technique achieves a compression ratio comparable to that of SBC with very modest resources, by using a finite-size stream cache instead

of an unbounded stream table, and limiting the maximal stream length and the number of loads/stores in a stream.

- Evaluation of SBC versus the best existing single-pass lossless compression techniques, PDATS, LBTC, and TCGEN. SBC outperforms all other techniques when compression ratio, decompression time, and compression time are all taken into consideration. Without additional compression by *gzip* or *bzip2*, SBC compresses better than all other techniques, outperforming the next best technique, LBTC, 4.8 times. SBC also has the highest compression ratio when combined with *gzip*, for all but two considered applications. When combined with *bzip2*, it achieves a compression ratio comparable to that of TCGEN.bz2. Though TCGEN.bz2 has the best overall compression ratio, it has a significant decompression overhead. Evaluation of decompression time in two settings, with an optimized decompression program and a real simulator, shows that SBC traces decompress much faster than TCGEN traces. With the custom decompression program, SBC.gz and SBC.bz2 are 4.5 and 3 times faster than TCEN.bz2, respectively. Simulation with the DineroIV cache memory simulator finishes 1.56 times faster with SBC.gz than with TCGEN.bz2, and 1.35 times faster with SBC.bz2.

- Analysis of compressibility of instruction and data trace components for all evaluated techniques, as well as sensitivity of the SBC compression ratio to the data address FIFO buffer size.

- A novel classification of lossless trace compression techniques, based on the method used for compression of data and instruction components.

As our analysis indicates, future compression techniques should concentrate on the data trace component, since it makes up over 80% of the SBC compressed traces. Another avenue to be explored is the extension of the SBC trace format for other types of traces. An additional benefit of the SBC technique is the separation of the stream trace file from the rest of the compressed data. Hence, various statistics such as the number of executed instructions of a particular type, number of streams, and dynamic distribution of streams can be obtained by processing only the stream table file instead of the whole trace. The stream table file can also include information about basic blocks in a stream and a stream frequency field.

REFERENCES

AGARWAL, A., SITES, R. L. and HOROWITZ, M. 1986. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, June 1986, 119-127.

ARM. 2004. CoreSight On-chip Debug and Trace Technology. <http://www.arm.com/products/solutions/CoreSight.html> (July 2004).

BECKER, J. C. and PARK, A. 1993. An analysis of the information content of address and data reference streams. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, Santa Clara, CA, May 10 - 14, 1993, 262 - 263.

BURGER, D. and AUSTIN, T. 1997. The SimpleScalar Tool Set Version 2.0. CS-TR-97-1342, University of Wisconsin, WI, USA.

BURROWS, M. and WHEELER, D. J. 1994. A Block-sorting Lossless Data Compression Algorithm. Report 124, Digital SRC.

BURTSCHER, M. 2004. VPC3: A Fast and Effective Trace-Compression Algorithm. In *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04)*, New York, NY, USA, June 2004, 167-176.

BURTSCHER, M., et al. 2005. The VPC Trace-Compression Algorithms. *IEEE Transactions on Computers 54*, 11 (November 2005), 1329-1344.

BURTSCHER, M. and JEERADIT, M. 2003. Compressing Extended Program Traces Using Value Predictors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, LA, September 2003, 159-168.

BURTSCHER, M. and SAM, N. B. 2005. Automatic Generation of High-Performance Trace Compressors. In *2005 International Symposium on Code Generation and Optimization (CGO'05)*, San Jose, CA, USA, March 2005, pp. 229-240.

CANTIN, J. F. and HILL, M. D. 2003. Cache Performance for SPEC CPU2000 Benchmarks. <http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/> (April 2004).

DEROSE, L., et al. 2002. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, 1-13.

EDLER, J. and HILL, M. D. 1998. Dinero IV Trace-Driven Uniprocessor Cache Simulator. <http://www.cs.wisc.edu/~markhill/DineroIV/> (August 2003).

EGGERS, S. J., et al. 1990. 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, USA, 37 - 47.

ELNOZAHY, E. N. 1999. Address Trace Compression Through Loop Detection and Reduction. *ACM SIGMETRICS Performance Evaluation Review 27*, 1 (June), 214-215.

FISHER, J. A., FARABOSCHI, P. and YOUNG, C. 2005. *Embedded Computing: A VLIW Approach in Architecture, Compilers, and Tools*. Morgan Kaufmann, San Francisco, CA, USA.

HAMOU-LHADJ, A. and LETHBRIDGE, T. C. 2002. Compression techniques to simplify the analysis of large execution traces. In *Proceedings of the 10th International Workshop on Program Comprehension*, Paris, France, 27-29 June 2002, 159 -168.

JOHNSON, E. E., HA, J. and ZAIDI, M. B. 2001. Lossless Trace Compression. *IEEE Transactions on Computers 50*, 2 (February), 158-173.

LARUS, J. R. 1993. Efficient program tracing. *IEEE Computer 26*, 5 (May 1993), 52-61.

LARUS, J. R. 1999. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming language design and implementation*, Atlanta, GA, 259-269.

LUO, Y. and JOHN, L. K. 2004. Locality-Based Online Trace Compression. *IEEE Transaction on Computers 53*, 6 (June 2004), 723-731.

MCCULLOUGH, et al. 2003. Trace reporting method and system. United States Patent 6,615,371.

MILENKOVIC, A. and MILENKOVIC, M. 2003a. Exploiting Streams in Instruction and Data Address Trace Compression. In *Proceedings of IEEE 6th Annual Workshop on Workload Characterization*, Austin, TX, USA, October 27, 2003, 99-107.

MILENKOVIC, A. and MILENKOVIC, M. 2003b. Stream-Based Trace Compression. *Computer Architecture Letters 2*, (September 2003),

MILENKOVIC, A., MILENKOVIC, M. and KULICK, J. 2003. 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 (PDCS-2003)*, Reno, NV, USA, August 13-15, 2003, 49-55.

NEVILL-MANNING, C. G. and WITTEN, I. H. 1997. Linear-Time, Incremental Hierarchy Interference for Compression. In *Proceedings of the IEEE Data Compression Conference*, Snowbird, UT, 3-11.

PLESZKUN, A. R. 1994. Techniques for compressing program address traces. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, USA, November 30 - December 02, 1994, 32-39.

SAMPLES, A. D. 1989. Mache: no-loss trace compaction. In *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Oakland, CA, USA, May 23 - 26, 89 - 97.

SKADRON, K., et al. 2003. Challenges in Computer Architecture Evaluation. *IEEE Computer 36*, 8 (August 2003), 30 - 36.

SMITH, M. D. 1991. Tracing with Pixie. Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University.

SPEC. 2000. SPEC 2000 CPU Benchmark Suite. <http://www.spec.org> (February 2004).

THORNOCK, N. C. and FLANAGAN, J. K. 2001. A national trace collection and distribution resource. *ACM SIGARCH Computer Architecture News 29*, 3 (June 2001), 6-10.

UHLIG, R., et al. 1995. Instruction fetching: coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, S. Margherita Ligure, Italy, 345 - 356.

UHLIG, R. A. and MUDGE, T. N. 1997. Trace-driven memory simulation: a survey. *ACM Computing Survey 29*, 2 (June), 128-170.

ZHANG, Y. and GUPTA, R. 2001. Timestamped whole program path representation and its applications. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, USA, 180-190.

ZHOU, M. and SMITH, A. J. 2000. Tracing Windows95. *Microprocessors and Microsystems 24*, 333–347.

ZIV, J. and LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory 23*, 3 (May 1977), 337-343.