Automatic Generation of High-Performance Trace Compressors

Martin Burtscher and Nana B. Sam
Computer Systems Laboratory, Cornell University
{burtscher, besema}@csl.cornell.edu

Abstract

Program execution traces are frequently used in industry and academia. Yet, most trace-compression algorithms have to be re-implemented every time the trace format is changed, which takes time, is error prone, and often results in inefficient solutions. This paper describes and evaluates TCgen, a tool that automatically generates portable, customized, high-performance trace compressors. All the user has to do is provide a description of the trace format and select one or more predictors to compress the fields in the trace records. TCgen translates this specification into C source code and optimizes it for the specified trace format and predictor algorithms. On average, the generated code is faster and compresses better than the six other compression algorithms we have tested. For example, a comparison with SBC, one of the best trace-compression algorithms in the current literature, shows that TCgen's synthesized code compresses SPECcpu2000 address traces 23% more, decompresses them 24% faster, and compresses them 1029% faster.

1. Introduction

Execution traces are widely used by researchers and educators to study program behavior and to drive simulators. They are easy to process and guarantee repeatability. The problem with traces from interesting applications is that they are often large and storing them can be a challenge, even on today's high-capacity disks. Thus, traces are usually compressed.

Many trace-compression algorithms have been proposed [1], [3], [7], [8], [18], [19], [21], [22], [25], [26], [32], [33], [39]. While most implementations work well for a predefined trace type, they are either domain specific, do not compress all that well, or cannot adapt to different trace formats. As a consequence, users may find themselves re-implementing their favorite compression algorithm every time they start a new project that requires different traces. Alas, writing new code is not only time consuming but also error prone and likely to result in suboptimal performance because algorithm details are left out and novel optimization opportunities are overlooked.

This paper presents TCgen, an application-specific compiler that solves the above-mentioned problems by automatically translating simple user-provided trace descriptions and predictor selections into high-performance trace compression utilities. The generated code is typically faster and compresses better than the other compression algorithms we have evaluated. TCgen emits portable C source code that is highly optimized for the given trace format and predictors. The generated code is human readable to the extent that it is correctly indented, does not utilize macros, includes only one statement per line, and contains meaningful variable and function names. Users can choose between a number of prediction algorithms and combinations to optimize the compression rate and speed. A typical trace description, including the predictor selection, requires a couple of hundred characters.

TCgen is quite fast, taking under three thousandths of a second on our reference machine to generate and optimize code even for sophisticated trace descriptions. Compiling the emitted C code with a high optimization level typically takes under one second. In other words, the synthesis and compilation overhead is negligible compared to the time it takes to compress or decompress a multi-gigabyte trace.

TCgen was inspired by VPC3 [3], a fast and well-performing trace-compression algorithm. Like VPC3, TCgen employs value-prediction algorithms to convert a trace into streams that are highly compressible and that can be compressed and decompressed very quickly with a general-purpose compression algorithm.

Value predictors identify patterns in sequences of numbers to forecast the likely next value. In recent years, hardware-based value predictors have been researched extensively to predict the content of CPU registers [6], [10], [23], [24], [34], [35], [38], making them good candidates for predicting the kind of values typically found in program traces.

The following simplified example illustrates how the value predictors are used to convert traces into streams and compress them. Let us assume we have a set of predictors and that we want to compress a trace containing records with a single field. During compression, the current field's value is compared with the predicted values. If at least one of the predictions is correct, the identification number of one of the correct predictors is written to the first stream. If none of the predictions is right, a reserved identification number is written to the first stream

and the unpredictable value is output to the second stream. Then the predictors are updated and the procedure repeats until all records have been processed.

Decompression works as follows. First, one entry is read from the first stream. If it contains the reserved identification number, the field's value is obtained from the second stream. If, on the other hand, the entry contains a predictor identification number, the value from the corresponding predictor is used. Then the predictors are updated to ensure that their state is consistent with the corresponding state during compression. This process is iterated until the entire trace has been reconstructed.

This approach already compresses the traces somewhat. However, the key is that it converts the traces into streams that a general-purpose compressor can compress well and quickly. We use BZIP2 for this purpose, but users are free to select any other algorithm.

A comparison with BZIP2 [13], MACHE [33], PDATS II [18], SBC [26], SEQUITUR [22], and VPC3 [3] shows that a TCgen-generated compressor outperforms all of them on average (harmonic mean) in compression rate, decompression speed, and compression speed on the three types of SPECcpu2000 traces we studied (SBC decompresses one type 2% faster).

TCgen is available on-line at http://www.csl.cornell.edu/~burtscher/research/TCgen/. We have successfully tested a large number of compressors generated by TCgen on a 64-bit UNIX system using *cc* as well as on a 32-bit Windows machine using *gcc* under cygwin [15].

The remainder of this paper is organized as follows. Section 2 summarizes related work. Section 3 introduces the value predictors available to TCgen. Section 4 presents our trace-specification language. Section 5 describes TCgen's code generation and optimization. Section 6 explains the evaluation methods. Section 7 discusses the results. Section 8 summarizes our findings.

2. Related Work

Automated code generation for the manipulation of files has been in use for a long time. For example, a paper by Norton [31], which dates back to 1978, proposes a tailored language for the automated generation of file modification tools. It describes the use of the Recursive Macro Actuated Generator (RMAG), a macro processor capable of generating source code in any language to create programs that manipulate sequential data files. These programs can create, update, and invert textual data as well as produce reports.

Haines et al. [11] propose an object-oriented approach to replace data files with "smart files". A SmartFile consists of a file descriptor, the data itself, and a set of associated library routines for interacting with the data at a relatively high level of abstraction. The authors introduce the DAFT (Data File Type) specification language, which includes three types of declarations: attributes, parameters, and fields. The attributes can be used to describe information related to the elements of a SmartFile, such as the fields, field types, and the files themselves. The parameters can be used to specify symbolic size and shape relationships for fields. The fields allow the user to define data abstractions. The DAFT compiler reads the file type declarations and produces an enhanced symbol table for the SmartFile access routines. An inheritance mechanism allows the user to derive new file types from existing ones. TCgen shares some of the file-description concepts and also compiles the specification into a tool.

Chilimbi et al. designed the Heap Allocation Trace Format (HATF) [7] to allow the collection and sharing of large allocation traces. HATF is a binary format that includes inline metadata that can dynamically modify the encoding of trace events. Data in a trace comes in two variants: metadata and data. The metadata indicates the size and interpretation of each field that follows. It can be dynamically defined and changed. For example, varying field sizes allows data gathering on 32-bit and 64-bit machines, empty fields can be omitted, regular patterns can be exploited, etc. The authors found that HATF was effective at compressing size fields but not address fields. Separating and compressing the address stream independently from the rest of the trace resulted in better compression. TCgen also separates the streams in the trace and compresses them individually. Additionally, it is more general in the sense that it can be used with traces created by arbitrary trace generators, i.e., it does not rely on the presence of metadata.

Meta-TF [7] builds on HATF. It is a meta specification language that enables a trace to contain metadata to vary the encoding of the trace on the fly. It can be used to specify HATF. The first step is for the user to define a Document Type Definition (DTD). The DTD documents the format of the trace in a human readable way and allows meaningful names for records, fields, and attributes. The MetaTFtool, which includes a compiler for Meta-TF, then takes the Meta-TF DTD and produces a set of Java classes representing a reader/writer for traces that conform to the DTD. Record formats in the DTD allow trace events to be encoded in a number of ways. For instance, an allocation address may be given as a delta from a previous address. This approach also simplifies the automatic generation of trace reader/writer implementations from the DTD. TCgen's specification language also allows the automatic generation of trace (de-)compressors (a form of readers and writers). Note that Meta-TF generates variable trace formats with embedded encoding hints while TCgen produces variable encoders for user-defined trace formats.

STEP [1] provides a standard method of encoding general trace data to reduce the need for developers to construct specialized systems and as such is probably the

closest system to TCgen. STEP uses a definition language designed specifically to reuse records and feed definition objects to its adaptive encoding process, which employs strategies to increase the compressibility of the traces. As with TCgen, the traces are compressed using a general-purpose compressor. STEP comprises a definition language, *STEP-DL*, a compiler, *stepc*, and the architectural framework.

STEP-DL is similar to (and was inspired by) Meta-TF, but provides a more generalized trace format with better encoding properties. It is targeted towards application and compiler developers and focuses on Java programs running on a Java Virtual Machine. One key difference is that while Meta-TF uses a dynamic encoding policy with explicit changes, STEP-DL uses an adaptive encoding process, i.e., it monitors various characteristics of the input data and, when appropriate, makes adjustments to the encoding policy automatically. In other words, encoding strategies are associated with individual record types as opposed to the system as a whole. TCgen also uses different compression algorithm for different parts of a trace. STEP-DL, Meta-TF, and TCgen all use human-readable ASCII input files.

Sucu and Krintz developed ACE [37], an adaptive compression environment for a Java Virtual Machine to improve Internet transfer rates. It decides whether to compress or not, based on an estimation of the cost of performing on-the-fly compression. ACE monitors the number of bytes per second that the local host sends through socket write calls. If this rate is lower than the bandwidth between the local and remote host, ACE computes the compressed and uncompressed transfer time for each 32kB block of data to be sent and selects compression if the uncompressed transfer time is higher. ACE also estimates (via sampling) the compression rate, the effective data transfer rate, and the decompression rate. If the smallest of these rates exceeds the available bandwidth, then the data is sent compressed. Compression decisions are only made for blocks of data that are 32kB or larger. Smaller blocks are transferred uncompressed.

2.1 Compression Algorithms

This subsection describes the compression schemes with which we compare TCgen's output in Section 7. BZIP2 is a lossless, general-purpose algorithm that can be used to compress any kind of file. The remaining algorithms are special-purpose trace compressors that we modified (where necessary) to include efficient block I/O operations, to understand our trace format, and to utilize a post-compression stage to improve the compression rate. They are all single-pass, lossless compression schemes.

BZIP2: BZIP2 [13] is a general-purpose compressor that operates at byte granularity. It implements a variant of the block-sorting algorithm described by Burrows and Wheeler [2]. BZIP2 applies a reversible transformation to a block of inputs, uses sorting to group bytes with similar contexts together, and then compresses them with a Huffman coder. The block size is adjustable. We use version 1.0.2 with the "--best" option. BZIP2 requires about 10MB of memory to compress and decompress our traces. We evaluate BZIP2 as a standalone compressor and as the post-compressor for the other algorithms.

MACHE: MACHE [33] was designed to compress address traces. It distinguishes between three types of addresses, namely instruction fetches, memory reads, and memory writes. A label precedes each address in the trace to indicate its type. MACHE works as follows. After reading in a label and address pair, the address is compared with the base for the current label type. If the difference between the address and the base can be expressed in a single byte, the difference is emitted directly. Otherwise, the full address is emitted and this address becomes the new base for the current label. The algorithm repeats until the entire trace has been processed.

Since PC and data entries alternate in our trace format (Section 6.3), no labels are necessary to identify the type. MACHE only updates the base when the full address needs to be emitted. We retain this policy for the PC entries in the traces. However, for the data entries, we found it better to always update the base due to the frequently encountered stride behavior. Our implementation uses 2.3MB of memory to run.

PDATS II: PDATS II [18] improves upon PDATS [19] by exploiting common patterns in program behavior. For example, jump-initiated sequences are often followed by strided sequences. PDATS encodes such patterns using one record to specify the jump and another record to describe the sequential references. PDATS II combines the two records into one. Moreover, when a program writes to a particular memory location, it is also likely to read from that location. PDATS separates read and write references, resulting in two large offsets whenever the location changes. PDATS II does not treat read and write references separately. Additionally, common data offsets are encoded in the header byte and instruction offsets are stored in units of the default instruction stride (e.g., four bytes per instruction on most RISC machines). Thus, PDATS II achieves about twice the compression rate of PDATS on average.

We modified PDATS II as follows. Since our traces do not include both read and write accesses, we do not distinguish between them in the header. This makes an extra bit available, which we use to encode data offsets of ±16, ±32, and ±64. We further extended PDATS II to also accommodate six- and eight-byte offsets. Our traces do not exhibit many jump-initiated sequences that are followed by strided sequences. Hence, we do not need the corresponding PDATS II feature. Our implementation needs 2.2MB of memory to run.

SEQUITUR: SEQUITUR [22] compresses traces by converting them into a context-free grammar [28], [29], [30]. The algorithm applies two constraints while constructing the grammar: each digram (pair of consecutive symbols) in the grammar must be unique and every rule must be used more than once. The biggest drawback of SEQUITUR is its memory usage, which depends on the data to be compressed (it is linear in the size of the grammar) and can exhaust the system's resources when compressing large traces.

The SEQUITUR algorithm we use is a modified version of Nevill-Manning and Witten's implementation [12], which we changed as follows. We converted the C++ code into C, inlined the access functions, increased the symbol table size to 33,554,393 entries, and added code to decompress the grammars. To accommodate 64-bit trace entries, we included a function that converts each trace entry into a unique number (in expected constant time). Moreover, we construct two grammars, one for the PC entries and one for the data entries in the traces. To cap the memory usage, we start new grammars when eight million unique symbols have been encountered or 384MB of storage have been allocated for rule and symbol descriptors. We found these cutoff points to work well on our traces and system. According to *ps*, our implementation's memory usage never exceeds 951MB. To prevent SEQUITUR from becoming very slow due to hash-table inefficiencies, we also start a new grammar whenever the last 65,536 searches for entries required an average of more than thirty trials before terminating.

SBC: The Stream-Based Compression (SBC) algorithm [26], [27] is one of the newest trace compressors in the literature. It splits the traces into segments called instruction streams. A stream is a dynamic sequence of instructions from the target of a taken branch to the first taken branch in the sequence. SBC creates a stream table that records relevant information such as the starting address, the number of instructions in the stream, and the instruction words and their types. During compression, groups of instructions that belong to the same stream are replaced by the corresponding stream table index. To compress addresses of memory references, SBC further records information about the strides and the number of stride repetitions. This information is attached to the instruction stream. Note that TCgen's streams are unrelated to SBC's streams.

We made the following changes to SBC [16]. Since our traces contain only dynamic instances of some but not all instructions, we redefined an instruction stream as a sequence in which each subsequent instruction has a higher PC than the previous instruction and the difference between subsequent PCs is less than a preset threshold. We found a threshold of four instructions to provide the best compression rate on our traces. SBC uses 10MB of memory to run.

VPC3: We use the Third Value-Prediction-Based Compression (VPC3) algorithm [3] as a starting point for TCgen. VPC3 employs value predictors to convert traces into streams. BZIP2 is then utilized to compress the streams. Internally, the value predictors divide the traces into ministreams, one for each PC (i.e., static instruction), which exhibit more locality than the original trace in which the data are interleaved in complicated ways.

Since we utilize the VPC trace format in this paper, we did not have to make changes to this algorithm and were able to use it directly [14]. VPC3 requires 27MB of memory to execute.

3. Value Predictors

This section describes the value predictors available to TCgen, i.e., the predictors the user can select and configure. All predictors predict the next trace entry based on previously processed entries.

Last-value predictor: The first type of predictor TCgen can emit is the last-value predictor (LV[*n*]) [6], [24], [38]. It predicts the *n* most recently seen values. This type of predictor can accurately predict sequences of repeating and alternating values as well as repeating sequences of no more than *n* arbitrary values. Figure 1 shows a diagram of an LV[*n*] predictor with *s* lines in its (first-level) table.

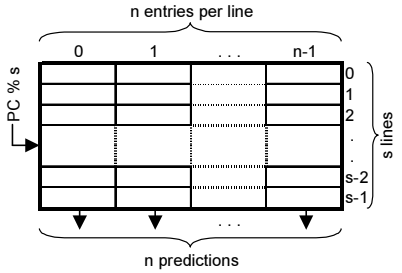

Figure 1: LV[*n*] predictor with *s* lines.

The LV[*n*] predictor always predicts the *n* values stored in the selected line. The PC (extracted from the current trace record) modulo *s* determines the line index. If no PC is available, *s* has to be one. After a prediction, the selected line is updated by discarding the oldest entry, moving the remaining entries to the right by one slot, and copying the update value into the first slot.

Finite-context-method predictor: The second type of predictor TCgen can produce is the finite-context-method predictor (FCM*x*[*n*]) [35]. It computes a hash out of the *x* most recently encountered values (*x* is the *order* of the predictor), which are stored in the predictor's first-level table, using the select-fold-shift-xor hash function [34]. The hash is then used to index the predictor's second-level table (i.e., the hash table), which works just like the LV[*n*] table. After updating the second-level table, the

entries in the selected line of the first-level table are moved to the right by one slot, thus dropping the oldest value and making room for the update value. Figure 2 shows an FCM*x*[*n*] predictor with *s* lines in the first-level table (L1) and *t* lines in the second-level table (L2).

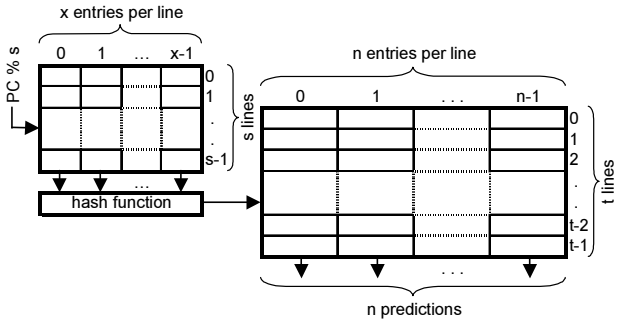

Figure 2: FCM*x*[*n*] predictor with L1 = *s* and L2 = *t*.

This predictor predicts the *n* values that followed the last *n* times the same *x* preceding values (that is, the same context) have been encountered [34], [35]. Thus, FCM predictors can memorize long arbitrary sequences of values and accurately predict them when they repeat.

Differential-finite-context-method predictor: The third type of predictor TCgen can create is the differential-finite-context-method predictor (DFCM*x*[*n*]) [10]. It works just like an FCM*x*[*n*] predictor except it predicts and is updated with differences (strides) between consecutive trace entries rather than with absolute values. To form the final prediction, the predicted stride is added to the most recently seen value (the last value). DFCM predictors are often superior to FCM predictors because they warm up faster, make better use of the hash table, and can predict values that have never been seen before. Figure 3 illustrates a DFCM*x*[*n*] predictor with *s* lines in the first-level table and *t* lines in the second-level table.

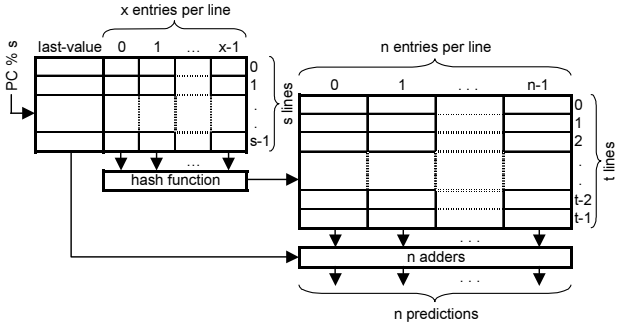

Figure 3: DFCM*x*[*n*] predictor with L1 = *s* and L2 = *t*.

In addition to predicting long arbitrary sequences of values that repeat, DFCMs can accurately predict long arbitrary sequences of offsets (between consecutive values) that repeat.

4. Trace Specification Language

The input to TCgen is a trace format description combined with a value-predictor configuration, expressed in the regular language whose grammar is shown in Figure 4. The start symbol is Description. TCgen also supports comments, which begin with a hash character ("#") and extend to the end of the line. The trace descriptions are case sensitive.

```
Description = 'TCgen' 'Trace' 'Specification' ';' Header Field {Field} PCDef.
Header = Number '-' 'Bit' 'Header' ';'.
Field = Number '-' 'Bit' 'Field' Number '=' '{' [LevelSizes] ':' Predictors '}' ';'.
LevelSizes = LevelSize [',' LevelSize].
LevelSize = ('L1' '=' Number) | ('L2' '=' Number).
Predictors = Predictor {',' Predictor}.
Predictor = ('DFCM' Number '[' Number ']') | ('FCM' Number '[' Number ']') |
    ('LV' '[' Number ']').
PCDef = 'PC' '=' 'Field' Number ';'.
Number = Digit {Digit}.
Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
```

Figure 4: EBNF grammar of the description language.

Figure 5 shows the specification needed to emulate VPC3 and its trace format. We will use it as an example to discuss the features of TCgen's input language. VPC3 traces start with a four-byte header that is followed by records with four-byte PC and eight-byte data fields.

```
TCgen Trace Specification;
32-Bit Header;
32-Bit Field 1 = {L1 = 1, L2 = 131072: FCM3[2], FCM1[2]};
64-Bit Field 2 = {L1 = 65536, L2 = 131072: DFCM3[2], DFCM1[2], FCM1[2],
    LV[4]};
PC = Field 1;
```

Figure 5: VPC3 trace format and predictor description.

All TCgen trace specifications have to start with the phrase "TCgen Trace Specification". A semicolon terminates each statement. The second line in Figure 5 informs TCgen that there is a header of four bytes. Header bytes are copied to a separate stream without passing them through value predictors. Lines three and four specify that each record comprises two fields. Field 1 is four bytes and Field 2 eight bytes wide. Line five tells TCgen that the first field of each trace record holds the PC.

The expressions in the curly brackets specify the types and sizes of the value predictors to be used for compressing the fields with which they are associated. Since the first field contains the PC, no index is available and the level-one (L1) predictor size has to be set to one. No matter where in a trace record the PC field is, it is always accessed first to make it available for computing the indices to predict the remaining fields.

The specified FCM3[2] and FCM1[2] predictors provide four predictions for Field 1. Note that each FCMx and DFCMx predictor is allocated a second-level table

with $L2*2^{(x-1)}$ lines (see Section 5.2). Hence, the FCM1's hash table has 131,072 lines and the FCM3's hash table has 524,288 lines.

The predictors for the second field all have 65,536 lines in their first-level tables. There are four predictors, DFCM3[2], DFCM1[2], FCM1[2], and LV[4], providing a total of ten predictions for Field 2. According to the above formula, the DFCM3's level-two size is four times the specified L2 value. The last-value predictor does not have a second-level table, so the L2 value is irrelevant for this predictor.

The L1 and L2 sizes have to be powers of two to make the modulo computations fast. At least one predictor has to be specified for each field. However, providing L1 and L2 sizes is optional. If L1 is left out, a default value of one is used. If L2 is omitted, it defaults to 65,536, which we chose because it yields good results for most traces while at the same time keeping the memory footprint reasonable. The PC specification, e.g., the last statement in Figure 5, is mandatory as the value-prediction-based approach is particularly effective on traces containing PCs. Nevertheless, it is always possible to compress traces without PC information by specifying an L1 size of one for all fields. In fact, if only a single eight-bit field with an L1 size of one is specified, the resulting code can be used to compress and decompress arbitrary files, including non-trace files. This mode of operation, however, is not recommended as it typically underperforms BZIP2.

Unless TCgen terminates with a parse error, it will write the synthesized C code to the standard output. The listing starts with a commented out copy of the trace specification to document the code. This text can directly be used as input to TCgen. Moreover, this trace specification is emitted in canonical form and includes a comment line after each field specification (not shown) stating how many predictions will be made for that field and what the total size of the predictor tables is. The sum of the table size of all fields plus about 2MB accurately reflects the dynamic memory requirement of the synthesized code.

When the generated code is compiled and run, it will read and compress a trace from the standard input that matches the given specification. At the end of the compression, predictor usage information is written to the standard output. This feedback is provided to help the user select the most effective predictors. If the code is run with the "-d" command-line flag, it will decompress the trace and write it to the standard output. No flag is needed for compression.

5. Code Generation and Optimization

This section describes the code generation and the application-specific optimizations that TCgen performs as well as the algorithmic enhancements over VPC3.

5.1 Code Generation

When emitting code, TCgen makes use of several techniques to aid the compiler in producing high-performance binary. For example, all code is written into one file (typically a few hundred lines of text), giving the compiler a global view of the program. All generated functions (except `main`) are declared static to allow the compiler to optimize the calling convention and to inline and eliminate any function it chooses. Similarly, all global variables are declared static. All local variables for which this is possible are declared as register variables to inform the compiler that no pointer analysis is necessary for them. Finally, all I/O is performed with efficient block I/O calls and the data is internally extracted from and inserted into buffers in a manner that avoids alignment problems.

5.2 Application-Specific Code Optimizations

TCgen performs several optimizations before emitting code. It includes a dead-code remover to eliminate statements that are only used for predictors that are not selected. For example, there is no need to compute a stride for any field that does not use a DFCM predictor. Similarly, if a trace format does not specify a header, no code to handle a header is emitted.

TCgen minimizes the memory footprint of the generated code by eliminating unnecessary predictor tables, coalescing all tables that hold the same information, and minimizing the table sizes. For instance, all DFCM and LV predictors need to retain the most recent values (in the last-value table). If only FCM predictors are specified for a given field, no last-value table is emitted. On the other hand, if multiple DFCM predictors or one or more DFCM and an LV predictor are specified, only one shared last-value table is generated. Furthermore, each FCM and DFCM predictor is assigned a second-level table with $L2*2^{(x-1)}$ lines, where x is the order of the predictor. This is done to accelerate the computation of the hash function and so that only one first-level table is needed for all FCM predictors and another one for all DFCM predictors of each field. In fact, only the first-level table for the highest order predictor is generated and the lower-order predictors utilize whatever fraction of that table they need. All table elements are declared to be of the smallest type that is sufficiently large. Eight-bit fields, for instance, result in tables with eight-bit entries while 64-bit fields result in 64-bit table entries. Likewise, elements of the smallest possible type are written to the streams that receive the unpredictable values.

No matter which predictors are included, they are always "re-named" so that the predictor identification codes range from 0 to n-1, where n is the number of predictors.

TCgen eliminates superfluous parameters from functions that do not use them. For example, the PC does not need to be passed to prediction functions for fields that use global predictors.

Finally, TCgen incrementally computes the hash functions, which results in substantial speedups for predictors with large orders. Moreover, the partial hash function values are chosen in such a way that they reflect the correct indices for the lower-order predictors (if they exist). As a result, only n operations have to be performed to compute the new index for an n^{th}-order FCM or DFCM predictor, and the intermediate results provide "free" indices for all lower-order predictors that are present.

5.3 Algorithmic Enhancements

When instructing TCgen to emulate VPC3 using the trace specification from Figure 5, the synthesized code actually compresses better and is faster than VPC3 (see Section 7). This is the result of the following enhancements. First, we eliminated some duplicated variables. Second, we modified the hash function to be faster for small fields by taking advantage of the size of fields. Third, we improved the hash function to increase the number of reachable second-level table entries for small fields by automatically adapting the shift amount in the select-fold-shift-xor hash function [34]. Fourth and most importantly, we enhanced the update policy. VPC3 always updates all predictor tables, which makes it very fast because the tables do not have to be searched for a matching entry. VPC2 [4], on the other hand, only updates tables if the update value is not already in the selected line. This is much slower but increases the prediction accuracy because only distinct values are retained. TCgen's update policy combines the benefits of both approaches essentially without their downsides. It only performs an update if the current value is different from the first entry in the selected line. This way, only one table entry needs to be checked (no code to perform this check is emitted if the table only holds one entry per line), which makes updates fast while at the same time guaranteeing that at least the first two entries in each line are distinct, which improves the prediction accuracy.

6. Evaluation Methods

This section provides information about the system, the timing measurements, the traces, and the compiler we used for our measurements. It includes a brief description of our performance metrics.

6.1 System

We performed all measurements for this study on a dedicated 64-bit CS20 system with two 833MHz 21264B Alpha CPUs [20]. Only one of the processors was used. Each CPU has separate, two-way set-associative, 64kB L1 caches and an off-chip, unified, direct-mapped 4MB L2 cache. The system is equipped with 1GB of main memory. The Seagate Cheetah 10K.6 Ultra320 SCSI hard drive has a capacity of 73GB, 8MB of built-in cache, and spins at 10,000rpm. For maximum disk performance, we used the advanced file system (AdvFS). The operating system is Tru64 UNIX V5.1B.

6.2 Timing Measurements

All timing measurements in this paper refer to the sum of the user and the system time as reported by the UNIX shell command *time*. In other words, we report the CPU time and ignore any idle time such as waiting for disk operations. We programmed all compression and decompression algorithms so that they read traces from the hard disk and write traces back to the hard disk. While these disk operations are subject to caching, any resulting effect should be minimal because of the large sizes of our traces. Note that we "diff" the decompressed traces with the original traces after each run to verify their integrity.

6.3 Traces

We used all integer and all but four floating-point programs from the SPECcpu2000 benchmark suite [17] to generate the traces for this study. We had to exclude the four Fortran 90 programs due to the lack of a compiler. The C programs were compiled with Compaq's C compiler V6.3-025 using "-O3 -arch host -non_shared" plus feedback optimization. The C++ and Fortran 77 programs were compiled with g++/g77 V3.3 using "-O3 -static". We used statically linked binaries to include the instructions from library functions in the traces. Only system-call code is not captured. We generated traces from complete runs with the SPEC-provided test inputs using the binary instrumentation tool-kit ATOM [9], [36]. Two programs, eon and vpr, require multiple runs and perlbmk executes itself recursively. For these programs, we concatenated the subtraces into a single trace each.

We generated three types of traces from the 22 programs to evaluate the different compression algorithms. The first type captures the PC and the effective address of each executed store instruction. The second type contains the PC and the effective address of all loads and stores that miss in a simulated 16kB, direct-mapped, 64-byte line, write-allocate data cache. The third type of trace records the PC and the loaded value of every executed load instruction (that is not a prefetch, a NOP, or a load immediate). All traces use the same format, that is, alternating 32-bit PC and 64-bit data values. We did not generate traces with different formats as doing so would have

required us to implement multiple versions of all compression algorithms with which we compare TCgen.

We selected the store-address traces because, historically, many trace-compression approaches have focused on address traces. Such traces are typically relatively easy to compress. We picked the cache-miss-address traces because the simulated cache acts as a filter and only lets some of the memory accesses through, which we expect to distort the access patterns, making the traces harder to compress. Finally, we chose the load-value traces because load values span large ranges and include floating-point numbers, addresses, integer numbers, bitmasks, etc., which makes them difficult to compress.

Table 1 shows the program name, the programming language (lang), the type (integer or floating point), and the uncompressed size (in megabytes) of the three traces for each SPECcpu2000 program as well as which traces were excluded. We had to exclude all traces with more than one billion entries, i.e., the traces that are larger than twelve gigabytes, because they would have exhausted the available disk space. The corresponding entries in Table 1 are crossed out.

Table 1: Information about the traces.

program	lang	type	store addresses	cache miss addresses	load values
eon	C++	integer	2,086.1 MB	94.6 MB	2,164.6 MB
bzip2	C	integer	~~16,769.9 MB~~	726.1 MB	~~23,947.4 MB~~
crafty	C	integer	3,368.0 MB	1,967.8 MB	~~14,227.6 MB~~
gap	C	integer	1,269.2 MB	255.5 MB	3,141.6 MB
gcc	C	integer	2,280.9 MB	366.6 MB	4,523.2 MB
gzip	C	integer	2,836.0 MB	731.3 MB	8,070.1 MB
mcf	C	integer	400.4 MB	150.1 MB	455.7 MB
parser	C	integer	4,224.2 MB	821.2 MB	9,805.9 MB
perlbmk	C	integer	570.3 MB	86.1 MB	1,089.4 MB
twolf	C	integer	239.8 MB	73.4 MB	827.6 MB
vortex	C	integer	~~16,770.6 MB~~	2,185.2 MB	~~26,571.4 MB~~
vpr	C	integer	1,984.9 MB	644.0 MB	7,167.0 MB
ammp	C	floating point	5,159.2 MB	3,442.0 MB	~~16,406.9 MB~~
art	C	floating point	1,781.8 MB	2,381.8 MB	11,249.9 MB
equake	C	floating point	1,229.8 MB	418.8 MB	4,323.0 MB
mesa	C	floating point	3,671.1 MB	266.8 MB	6,055.2 MB
applu	F77	floating point	522.7 MB	77.1 MB	1,002.7 MB
apsi	F77	floating point	8,058.2 MB	3,018.9 MB	~~15,911.5 MB~~
mgrid	F77	floating point	5,110.0 MB	6,377.0 MB	~~102,794.6 MB~~
sixtrack	F77	floating point	~~18,735.9 MB~~	2,224.5 MB	~~38,889.2 MB~~
swim	F77	floating point	452.0 MB	149.3 MB	1,985.5 MB
wupwise	F77	floating point	10,829.6 MB	889.9 MB	~~22,628.8 MB~~

6.4 Compiler

To make the running-time comparisons as fair as possible, we compiled all compressors with the same compiler (Compaq's C compiler V6.3-025) and the same optimization flags (-O3 -arch host).

6.5 Performance Metrics

We use the following three performance metrics to evaluate the quality of the compression algorithms. They are, in decreasing order of importance, the *compression rate*, the *decompression speed*, and the *compression speed*. Other factors are also important. However, they are either the same for all algorithms we evaluated (such as the use of a linear-time, single-pass, lossless algorithm) or only one algorithm differs from the others (such as SEQUITUR not having a fixed memory requirement).

Our three metrics are defined as follows. They are all higher-is-better metrics.

$$compression\ rate = \frac{uncompressed\ size}{compressed\ size}$$

$$decompression\ speed = \frac{uncompressed\ size}{decompression\ time}$$

$$compression\ speed = \frac{uncompressed\ size}{compression\ time}$$

Note that the compression rate has no unit while the decompression and compression speeds are throughputs measured in bytes per second. The three metrics are inversely normalized to the uncompressed trace size and are therefore independent of the trace length. Due to the inverse normalization, the natural way of averaging each of these metrics is the harmonic mean.

7. Results

The following sections compare the six compression algorithms described in Section 2.1 with TCgen's output. Unless otherwise specified, TCgen is used with the trace description from Figure 5. Section 7.1 discusses the compression rate, Section 7.2 studies the decompression speed, Section 7.3 investigates the compression speed, Section 7.4 evaluates the effectiveness of TCgen's optimizations, and Section 7.5 takes a look at the sensitivity of the predictor selection. Note that each algorithm includes a BZIP2 post-compression stage (except BZIP2).

7.1 Compression Rate

Figure 6 depicts the harmonic-mean compression rates of the seven compression algorithms on our three types of traces. For each trace type, the algorithms are sorted from left to right by increasing compression rate. For improved readability, we show the compression rates relative to TCgen. Absolute numbers are provided in Table 2 and in a technical report [5].

TCgen delivers the best compression rate for each type of trace and outperforms VPC3 because of the improved update policy. VPC3 is the second best performer on the store-address and the load-value traces and SBC on the

cache-miss-address traces. Interestingly, SEQUITUR exceeds SBC's compression rate only on the load-value traces. This is probably because SEQUITUR does not handle strided sequences well, which occur more frequently in the address traces than in the load-value traces. This also explains why even MACHE and PDATS II outperform SEQUITUR on the store-address traces. Both of these algorithms were specifically designed for address traces. While PDATS II is more sophisticated than MACHE, the latter is superior to the former on the load-value traces. In fact, on these traces PDATS II with a BZIP2 post-compression stage results in a lower compression rate than BZIP2 alone.

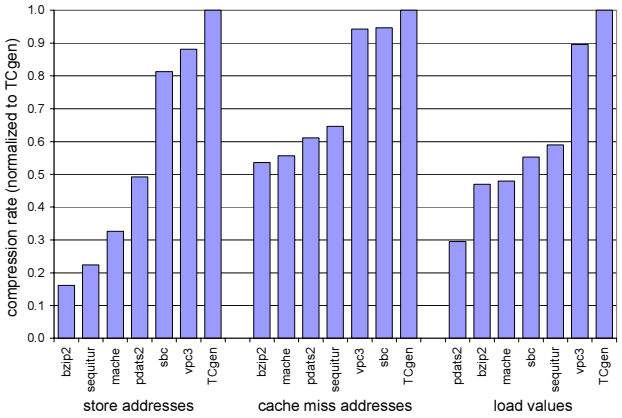

Figure 6: Harmonic-mean compression rates.

According to Figure 6, TCgen outperforms VPC3 by 6% to 13%, showing that the algorithmic improvements described in Section 5.3 are effective. SBC's compression rate is 6% to 81% lower than that of TCgen. SEQUITUR underperforms TCgen by more than 100% on the store-address traces, as do the remaining algorithms on both the store-address and the load-value traces.

Looking at the compression rates achieved by the seven algorithms on each individual trace [5], we find that no algorithm is the best for all traces. Nevertheless, TCgen outperforms the other six algorithms on every load-value trace. We suspect this to be the case because TCgen is based on value predictors, many of which were developed to predict load values, that is, the content of these traces.

On the 22 cache-miss-address traces, TCgen is outperformed by VPC3 on one trace (by 13%), by SEQUITUR on five traces (by up to 452%), by MACHE on three traces (by up to 46%), by SBC on nine traces (by up to 195%), by PDATS II on one trace (by 11%), and by BZIP2 on four traces (by up to 60%). However, on average and for the majority of the traces TCgen yields the highest compression rates.

On the 19 store-address traces, TCgen is inferior to VPC3 on one trace (by 7%), to SEQUITUR on four traces (by up to 168%), to MACHE on two traces (by up to 11%), to SBC on four traces (by up to 23%), and to PDATS II on three traces (by up to 42%), but exceeds BZIP2's compression rate on each trace. Again, on average and in the majority of the cases, TCgen yields the best compression rates. It compresses the store-address trace art by a factor of 77161, which is the highest compression rate we observed.

In the best case, TCgen outperforms VPC3 by a factor of 89, SEQUITUR by a factor of 4823, MACHE by a factor of 3991, SBC by a factor of 9.7, PDATS II by a factor of 461, and BZIP2 by a factor of 4001. TCgen compresses 36 of the 55 traces better than all the other algorithms we investigated.

7.2 Decompression Speed

Figure 7 shows the harmonic-mean decompression speed (the throughput in bytes per second at which the original trace is recreated) of the seven algorithms on our three types of traces normalized to TCgen. For each trace type, the algorithms are sorted from left to right by increasing decompression speed.

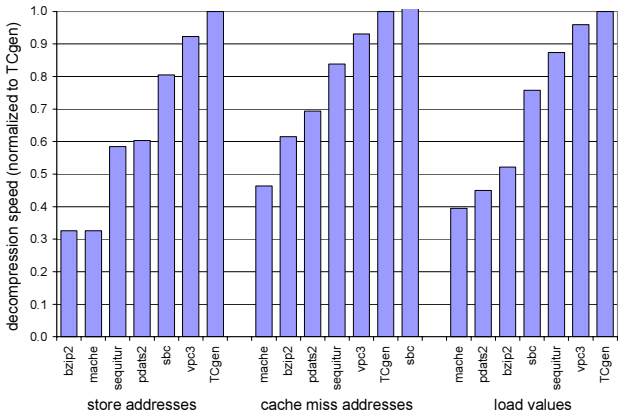

Figure 7: Harmonic-mean decompression speeds.

TCgen provides the fastest decompression speeds on the store-address and the load-value traces. On the cache-miss-address traces, SBC is 2% faster, but on the other two trace types, TCgen exceeds SBC's decompression speed by 24% and 32%. VPC3 is the next fastest decompressor, with TCgen being 4% to 8% faster. SEQUITUR is quite fast, too, except for the store-address traces, on which it is only a little over half as fast as TCgen. MACHE, PDATS II, and BZIP2 are in the bottom half for all three types of traces. They are at least 44% slower than TCgen.

Looking at the individual decompression speeds [5], we see that TCgen is faster than MACHE on all 55 traces. PDATS II beats TCgen on two traces (by up to 6%) and so does BZIP2 (by up to 19%). SBC is faster on one

load-value trace and on nine cache-miss-address traces (by up to 34%). VPC3 is faster on eight of the 55 traces, though never by more than 6%. Finally, SEQUITUR outperforms TCgen on eight of the 19 store-address traces (by up to 87%), on ten of the 22 cache-miss-address traces (by up to 125%), and on five of the load-value traces (by up to 57%). Nevertheless, TCgen is fastest on average and in the majority of the traces. It outperforms VPC3 by up to 28%, SEQUITUR by up to 599%, MACHE by up to 610%, SBC by up to 82%, PDATS II by up to 244%, and BZIP2 by up to 444%.

In absolute terms, TCgen regenerates the store-address traces at a harmonic-mean speed of 26MB/s (megabytes per second), the cache-miss-address traces at 11.8MB/s, and the load-value traces at 14.4MB/s. These rates exceed the throughput of a one-hundred megabit per second network connection and the transfer rates of many hard disks, suggesting that it may be faster to drive simulators and other trace-consumption tools by TCgen rather than from an uncompressed file on the hard drive.

7.3 Compression Speed

Figure 8 shows the harmonic-mean compression speed of the seven algorithms on the three types of traces relative to TCgen. Again, the algorithms are sorted from left to right by increasing speed.

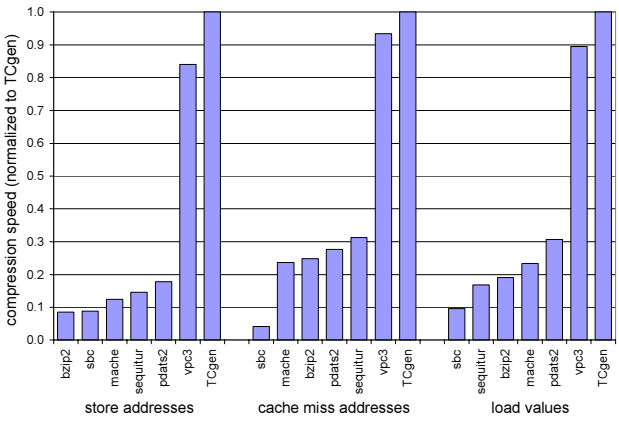

Figure 8: Harmonic-mean compression speeds.

TCgen and VPC3 (the algorithm TCgen is emulating) are dominant. In fact, MACHE, PDATS II, and BZIP2 compress exactly one of the 55 traces faster than TCgen (13%, 29%, and 17% faster, respectively). SBC is slower on every trace (up to 180 times slower) and so is SEQUITUR (up to 17 times slower). VPC3 is faster than TCgen on four traces, though never by more than 2% [5].

In absolute terms, TCgen yields harmonic-mean compression speeds of 7.5MB/s on the store-address traces, 4.2MB/s on the cache-miss-address traces, and 5.4MB/s on the load-value traces.

7.4 Optimizations

Table 2 evaluates the performance of TCgen's output when some of the application-specific optimizations and enhancements (Section 5) are turned off. The compression (c.spd) and decompression (d.spd) speeds are expressed in megabytes per second. The first line lists results when the new update policy is disabled and the predictors are always updated. The second line shows results when we do not minimize the types of array and stream elements. The third line presents results when all predictors get their own tables, i.e., there is no sharing. The fourth line displays results when the fast hash function is replaced by an equivalent function that always computes the hashes from scratch. The fifth line gives results when combining the first four "de-optimizations". Finally, the last line represents the performance when all optimizations are turned on. This is the configuration used everywhere else in this paper.

Table 2: Performance impact of TCgen's optimizations.

	store addresses			cache miss addrs			load values		
	rate	d.spd	c.spd	rate	d.spd	c.spd	rate	d.spd	c.spd
no smart update	132.6	25.8	7.0	17.9	11.6	4.0	21.9	13.8	5.1
no type minimization	142.5	25.4	6.7	18.5	10.9	3.5	23.0	14.0	5.1
no shared tables	142.9	24.5	7.3	18.6	11.3	4.1	23.0	13.8	5.2
no fast hash function	142.9	19.8	7.0	18.6	10.1	4.0	23.0	11.7	5.1
all of the above	131.9	17.6	5.6	17.7	8.8	3.0	21.9	10.9	4.4
full optimizations	142.9	26.0	7.5	18.6	11.8	4.2	23.0	14.4	5.4

The four investigated optimizations are useful in all cases. Disabling table sharing and using the unoptimized hash function do not change the compression rate but do slow down compression and decompression. The other two optimizations affect all of our performance metrics. Disabling the four optimizations simultaneously reduces the harmonic-mean compression rates by 4.8% to 7.7%, the decompression speeds by 24.4% to 32.3%, and the compression speeds by 17.8% to 28.7%.

7.5 Predictor Sensitivity

This section investigates how the predictor selection affects TCgen's performance. To do so, we generated a second compression utility that includes all the predictors we found to be useful for at least two of the 55 traces. The corresponding trace specification is shown in Figure 9. We call this configuration TCgen(B). The configuration used elsewhere in this paper (specified in Figure 5) is TCgen(A). Note that TCgen(B) is a true superset of TCgen(A). It uses 22 predictors and requires a total of 35MB of table space. TCgen(A) employs 14 predictors with a total table size of 20MB.

Table 3 compares the harmonic-mean performance of the two configurations on the three types of traces. The compression and decompression speeds are listed in megabytes per second.

```
TCgen Trace Specification;
32-Bit Header;
32-Bit Field 1 = {L1 = 1, L2 = 131072: FCM3[4], FCM1[4]};
64-Bit Field 2 = {L1 = 65536, L2 = 131072: DFCM3[4], DFCM1[2], FCM1[4],
   LV[4]};
PC = Field 1;
```

Figure 9: TCgen(B) specification.

Since TCgen(B) includes more predictors, one might expect it to compress better but be slower. However, this is only partially true because of the complex interaction with the post-compression stage. For instance, using more predictors will result in better compression in the first stage, but will also emit more distinct predictor codes, possibly making it harder for the post-compressor to be effective.

Table 3: Harmonic-mean performance of TCgen(A) and TCgen(B).

trace	compr. rate		decompr. speed		compr. speed	
	A	B	A	B	A	B
store addresses	142.9	132.2	26.0	23.6	7.5	5.4
cache miss addresses	18.6	19.1	11.8	11.7	4.2	4.4
load values	23.0	23.7	14.4	13.5	5.4	3.4

As Table 3 shows, TCgen(B) yields a 2% and 3% higher compression rate on the cache-miss-address and the load-value traces, respectively, but TCgen(A) is 8% more effective on the store-address traces. Similarly, TCgen(B)'s compression speed is 6% faster on the cache-miss-address traces, but TCgen(A) is 37% faster on the store-address and 57% faster on the load-value traces. Only the decompression speed is uniformly faster with TCgen(A) (by 1% to 10%).

These results show that TCgen's performance is relatively insensitive to the exact predictor choice. In fact, TCgen(B), a rather generic configuration, performs only slightly worse than TCgen(A), a configuration that is the result of manual fine tuning [3]. We take this as indication that TCgen will produce a high-performance compressor with any reasonable predictor specification. Note that, on average, TCgen(B)'s compression rate exceeds that of the other compression algorithms we tested, its decompression speed is higher (except for VPC3's on the store-address and load-value traces and SBC's on the cache-miss-address traces), and its compression speed is faster (except for VPC3's on the store-address and load-value traces). Given these results, we recommend that TCgen users start with a trace specification that covers a wide range of predictors and then eliminate the useless predictors as determined by the predictor usage information output after each compression.

In fact, the above approach could be used to optimize the predictor selection for each trace individually. Doing so would require the inclusion of the predictor configuration in the compressed trace so that a suitable decompressor can be generated when a trace needs to be read. This would incur an overhead of a few tens of bytes and about a second of CPU time to synthesize and compile the decompressor, for which the resulting higher compression rate and decompression speed should easily compensate.

8. Conclusions

This paper describes the code-generation and optimization process and evaluates the performance of TCgen, a tool that automatically synthesizes trace compressors from user-provided trace descriptions. Based on a variety of traces from the SPECcpu2000 benchmark suite, we have shown the generated code to be faster and to compress better on average than BZIP2, MACHE, PDATS II, SBC, SEQUITUR, and VPC3. In other words, the automatically generated code typically outperforms hand-crafted and optimized code. Note that all of the algorithms we compared our approach with, which include the best trace compressors from the current literature, have to be re-implemented every time the trace format changes. TCgen users, on the other hand, merely have to provide a new format description, expressed in a simple specification language, and a highly optimized compressor will be generated in about a second. Based on its ease of use, flexibility, performance, and portability, we believe TCgen to be a useful tool for trace-based research and teaching environments. TCgen is freely available at http://www.csl.cornell.edu/~burtscher/research/TCgen/.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant No. 0312966. We would like to thank Ilya Ganusov, Sandra J. Jackson, Jian Ke, and Paruj Ratanaworabhan for porting and adapting MACHE, PDATS II, and SBC to our platform and trace format.

References

[1] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. "STEP: a Framework for the Efficient Encoding of General Trace Data." *Workshop on Program Analysis for Software Tools and Engineering*, pp. 27-34. November 2002.

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

[3] M. Burtscher. "VPC3: A Fast and Effective Trace-Compression Algorithm." *Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 167-176. June 2004.

[4] M. Burtscher and M. Jeeradit. "Compressing Extended Program Traces Using Value Predictors." *International Conference on Parallel Architectures and Compilation Techniques*, pp. 159-169. September 2003.

[5] M. Burtscher and N. B. Sam. "Automatic Generation of High-Performance Trace Compressors." *Cornell University, Computer Systems Laboratory, Technical Report CSL-TR-2004-1042*. November 2004.

[6] M. Burtscher and B. G. Zorn. "Exploring Last *n* Value Prediction." *International Conference on Parallel Architectures and Compilation Techniques*, pp. 66-76. October 1999.

[7] T. Chilimbi, R. Jones, and B. Zorn. "Designing a Trace Format for Heap Allocation Events." *Second International Symposium on Memory Management*, pp. 35-49. October 2000.

[8] E. N. Elnozahy. "Address Trace Compression Through Loop Detection and Reduction." *International Conference on Measurement and Modeling of Computer Systems*, pp. 214-215. May 1999.

[9] A. Eustace and A. Srivastava. "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools." *WRL Technical Note TN-44, Digital Western Research Lab, Palo Alto*. July 1994.

[10] B. Goeman, H. Vandierendonck, and K. Bosschere. "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency." *Seventh International Symposium on High Performance Computer Architecture*, pp. 207-216. January 2001.

[11] M. Haines, P. Mehrotra, and J. V. Rosendale. "SmartFiles: An OO Approach to Data File Interoperability." *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 453-466. October 1995.

[12] http://sequence.rutgers.edu/sequitur/sequitur.cc

[13] http://sources.redhat.com/bzip2/

[14] http://www.csl.cornell.edu/~burtscher/research/trace compression/

[15] http://www.cygwin.com/

[16] http://www.ece.uah.edu/~lacasa/sbc/sbc.html

[17] http://www.spec.org/osg/cpu2000/

[18] E. E. Johnson. "PDATS II: Improved Compression of Address Traces." *International Performance, Computing and Communications Conference*, pp. 72-78. February 1999.

[19] E. E. Johnson and J. Ha. "PDATS: Lossless Address Trace Compression for Reducing File Size and Access Time." *IEEE International Phoenix Conference on Computers and Communication*, pp. 213-219. April 1994.

[20] R. E. Kessler, E. J. McLellan, and D. A. Webb. "The Alpha 21264 Microprocessor Architecture." *International Conference on Computer Design*, pp. 90-95. October 1998.

[21] J. R. Larus. "Abstract Execution: A Technique for Efficiently Tracing Programs." *Software–Practice and Experience*, Vol. 20, No. 12, pp. 1241-1258. December 1990.

[22] J. R. Larus. "Whole Program Paths." *Conference on Programming Language Design and Implementation*, pp. 259-269. May 1999.

[23] M. H. Lipasti and J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction." *29th International Symposium on Microarchitecture*, pp. 226-237. December 1996.

[24] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. "Value Locality and Load Value Prediction." *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147. October 1996.

[25] Y. Luo and L. K. John. "Locality-based Online Trace Compression." *IEEE Transactions on Computers*, Vol. 53, No. 6, pp. 723-731. June 2004.

[26] A. Milenkovic and M. Milenkovic. "Stream-Based Trace Compression." *Computer Architecture Letters*, Vol. 2, pp. 14-17. September 2003.

[27] A. Milenkovic and M. Milenkovic. "Exploiting Streams in Instruction and Data Address Trace Compression." *6th Annual Workshop on Workload Characterization*, pp. 99-107. October 2003.

[28] C. G. Nevill-Manning and I. H. Witten. "Linear-Time, Incremental Hierarchy Interference for Compression." *The Data Compression Conference*, pp. 3-11. March 1997.

[29] C. G. Neville-Manning and I. H. Witten. "Identifying Hierarchical Structure in Sequences: A linear-time algorithm." *Journal of Artificial Intelligence Research*, Vol. 7, pp. 67-82. September 1997.

[30] C. G. Nevill-Manning and I. H. Witten. "Compression and Explanation Using Hierarchical Grammars." *The Computer Journal*, Vol. 40, pp. 103-116. 1997.

[31] L. M. Norton. "A Program Generator Package for Management of Data Files - The Input Language." *ACM 1978 Annual Conference*, pp. 217-222. December 1978.

[32] A. R. Pleszkun. "Techniques for Compressing Program Address Traces." *27th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 32-40. November 1994.

[33] A. D. Samples. "Mache: No-Loss Trace Compaction." *International Conference on Measurement and Modeling of Computer Systems*, Vol. 17, No. 1, pp. 89- 97. April 1989.

[34] Y. Sazeides and J. E. Smith. "Implementations of Context Based Value Predictors." *Technical Report ECE-97-8, University of Wisconsin-Madison*. December 1997.

[35] Y. Sazeides and J. E. Smith. "The Predictability of Data Values." *30th International Symposium on Microarchitecture*, pp. 248-258. December 1997.

[36] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools." *Conference on Programming Language Design and Implementation*, pp. 196-205. June 1994.

[37] S. Sucu and C. Krintz. "ACE: A Resource-Aware Adaptive Compression Environment." *International Conference on Information Technology: Coding and Computing*, pp. 183-188. April 2003.

[38] K. Wang and M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors." *30th International Symposium on Microarchitecture*, pp. 281-290. December 1997.

[39] Y. Zhang and R. Gupta. "Timestamped Whole Program Path Representation and its Applications." *Conference on Programming Language Design and Implementation*, pp. 180-190. June 2001.