VPC3: A Fast and Effective Trace-Compression Algorithm

Martin Burtscher
Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University, Ithaca, NY 14853
burtscher@csl.cornell.edu

ABSTRACT

Trace files are widely used in research and academia to study the behavior of programs. They are simple to process and guarantee repeatability. Unfortunately, they tend to be very large. This paper describes *vpc3*, a fundamentally new approach to compressing program traces. *Vpc3* employs value predictors to bring out and amplify patterns in the traces so that conventional compressors can compress them more effectively. In fact, our approach not only results in much higher compression rates but also provides faster compression and decompression. For example, compared to *bzip2*, *vpc3*'s geometric mean compression rate on SPECcpu2000 store address traces is 18.4 times higher, compression is ten times faster, and decompression is three times faster.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: Data Compaction and Compression. B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Trace compression, predictor-based compression, trace files.

1. INTRODUCTION

Program execution traces are widely used to study program and processor behavior. Unfortunately, traces from interesting programs tend to be very large and storing them can be a challenge, even on today's large hard disks. One obvious solution is data compression.

Many trace-compression algorithms have been proposed [2, 5, 6, 16, 17, 20, 21, 27, 30, 37, 38]. Most of them do an excellent job at compressing program traces that record the program-counter values (PCs) of the executed instructions. However, traces that contain additional information such as effective addresses or the contents of registers are much harder to compress because those values repeat less and span larger ranges than PCs. Yet, extended traces, as we call them, are gaining importance as more and more researchers investigate the dynamic activities in computer systems.

Our initial idea was to use value-prediction techniques to directly compress extended traces. Value predictors identify patterns in a sequence of values to forecast the likely next value (Section 2.2). In recent years, a number of value predictors have been developed to predict the content of CPU registers [4, 8, 9, 22, 23, 31, 32, 34, 36]. Hence, they are good candidates for predicting the kind of values we are concerned with, i.e., values that span large ranges and that do not necessarily repeat often.

The following greatly simplified example illustrates how value predictors can be used to compress traces. Let us assume we have a set of predictors and that the data to be compressed consist of eight-byte values. When compressing a trace, the current entry is compared with the predicted values. If at least one of the predictions is correct, we write only the identification number of one of the correct predictors, encoded in a one-byte value, to indicate that the predictor is correct. If none of the predictions are right, we write a special code followed by the unpredictable eight-byte value. Then the predictors are updated with the true value and the procedure repeats for the remaining trace entries.

Decompression proceeds analogously. First, one byte is read from the compressed trace. If it contains the special code, the next eight bytes are read to obtain the actual value. If, on the other hand, the byte contains a predictor identification number, the value from the corresponding predictor is used. Then the predictors are updated. This process iterates until the entire trace has been reconstructed.

In this example, nine bytes are required to encode an unpredictable value but only one byte for a predictable value. Hence, if the predictors correctly predict more than one out of eight entries, the trace will be compressed.

We found a more sophisticated version of the above algorithm (*vpc1*) to perform quite well, especially once we added a *gzip* postcompression stage (*vpc2*) [2]. Unfortunately, *vpc2*'s decompression speed is about four times slower than that of other algorithms, meaning that it is only likely to be used for archiving traces.

Clearly, we needed a faster algorithm. Moreover, we decided to capitalize on the fact that traces seemed to be more compressible after processing them with value predictors. Thus, we designed the completely new algorithm *vpc3*, in which the objective of the value predictors is not to compress the traces but rather to convert them into a format that is as amenable to the second-stage compressor as possible.

Indeed, using value predictors to preprocess traces is much more fruitful than using them for the actual compression. *Vpc3* not only compresses traces better than *vpc2*, *sequitur*, and *bzip2* but also compresses and decompresses them faster. Moreover, *vpc3* meets all of the following criteria.

- ♦ lossless compression
- ♦ single-pass algorithm
- ♦ excellent compression rate
- ♦ fixed memory requirement
- ♦ fast decompression speed
- ♦ fast compression speed

We want lossless compression in order to recreate the original trace exactly, which is a requirement for many experiments. A single-pass algorithm ensures that the uncompressed trace never has to exist as a whole because the trace can be compressed while it is generated and stored directly in the compressed format. Similarly, a single-pass decompression scheme can directly drive trace-consuming tools such as simulators, obviating the need to first decompress the entire trace. A good compression rate is obviously desirable to save as much disk space as possible and to keep transfer times and costs small when sending traces over a network. To be useful, a new compression algorithm has to exceed the compression rate of preexisting algorithms such as *lz77* [39], *lzw* [37], *bzip2* [1, 11], and *sequitur* [21, 24, 25, 26]. We opted for an algorithm with a fixed memory footprint that is independent of the trace content and length so that all computers with a reasonable amount of memory can compress and decompress *vpc3* traces. (*Sequitur*'s memory requirement, on the other hand, depends on the data to be compressed, which causes problems when compressing extended traces.) Fast decompression is a necessity for any trace-compression utility to become widespread. Finally, fast compression is also desirable, particularly in real-time and academic environments.

Vpc3 runs in a single pass in linear time over the data. Its compression rate and speed are very good, outperforming *gzip* and *bzip2* (both with the "--best" option) as well as *sequitur*. For example, our *vpc3* algorithm compresses a 2.28-gigabyte SPEC-cpu2000 gcc trace of store-instruction PCs and effective addresses by a factor of 67.3 in 3.7 minutes on our reference machine. Decompression takes 1.8 minutes. *Sequitur* compresses the same trace by a factor of 42.2 in 31.2 minutes and decompresses it in 1.6 minutes (slightly faster than *vpc3*). *Bzip2* achieves a compression rate of 22.6 and takes 57.3 minutes to compress the trace and 4.3 minutes to decompress it. Section 5 presents more results.

The C source code of *vpc3* is available on-line at http://www.csl.cornell.edu/~burtscher/research/tracecompression/. A sample test trace and a brief description on how to use and modify the code are also included. The code has been successfully tested on 32- and 64-bit UNIX/Linux systems using *cc* and *gcc* as well as on Windows under *cygwin* [12].

The remainder of this paper is organized as follows. Section 2 summarizes related work and introduces the value predictors we use. Section 3 describes the *vpc3* algorithm in detail. Section 4 explains the evaluation methods. Section 5 presents the results. Section 6 points out directions for future work and Section 7 concludes the paper.

2. RELATED WORK

Most early trace compression techniques are lossy since they employ filtering or sampling methods. The lossless approaches concentrate mostly on address traces. Larus proposed Abstract Execution [20], where a small amount of runtime data drives the re-execution of the program slices that generate the program's addresses. Pleszkun designed a two-pass trace compression algorithm that encodes the dynamic basic block successors using a dense representation [27]. Other lossless trace-compression algorithms include Mache [30], PDATS [16], PDI [17], and LD&R [6]. Mache, PDATS, and PDI work by exploiting spatiality (address differences) and sequentiality (repeat counts) of the trace. The three algorithms include a postcompression phase with an LZ77 [39] or LZW [37] algorithm to boost the compression rate. LD&R (Loop-Detection and Reduction) detects loops in address traces and extracts the references that are constant or change by a constant stride between loop iterations before encoding the remainder of the references. While our approach shares some of the same ideas (it also exploits sequentiality and spatiality and employs a second compression stage), the above-mentioned algorithms do not reach our algorithm's compression rate because *vpc3* exploits a much wider range of patterns and uses the first stage exclusively to enhance the performance of the second-stage compressor.

2.1 Compression Algorithms

We now describe the compression schemes with which we compare our approach in Section 5. The first two are general-purpose algorithms that can be used to compress any kind of file. The last one is a special-purpose algorithm tailored to our trace format.

Gzip: *Gzip* is a general-purpose compression utility found on most UNIX systems [13]. It operates at byte granularity and implements a variant of the LZ77 algorithm [39]. It looks for duplicated sequences of bytes (strings) within a 32kB sliding window. The length of the string is limited to 256 bytes, which corresponds to the lookahead-buffer size. *Gzip* uses two Huffman trees, one to compress the distances in the sliding window and another to compress the lengths of the strings as well as the individual bytes that were not part of any matched sequence. The algorithm finds duplicated strings using a chained hash table where each entry records three consecutive bytes. In case of a collision, the hash chain is searched beginning with the most recently inserted string. A command-line argument determines the maximum length of the hash chains and whether lazy evaluation is to be used (we use the "--best" option). With lazy evaluation, the algorithm does not immediately utilize the matched sequence for the byte currently being processed but first compares it to the matched sequence of the next input byte before selecting the longer of the two matches. According to *ps*, *gzip* requires approximately 2.3MB of memory when compressing our traces.

Bzip2: *Bzip2* [11] is quickly gaining popularity in the UNIX world. It is a general-purpose compressor that operates at byte granularity. It implements a variant of the block-sorting algorithm described by Burrows and Wheeler [1]. The algorithm 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 the "--best" option. *Bzip2* generally compresses better than *gzip* but is slower. It requires about 10MB of memory to compress our traces.

Sequitur: *Sequitur* is one of the best trace compression algorithms in the current literature. It has the unique feature that interesting information about the trace can be derived from the compressed format without the need for decompression. *Sequitur* converts a trace into a context-free grammar and thereby identifies hierarchical structures [24, 25, 26]. 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 is linear in the size of the grammar.

The *sequitur* algorithm we use is a modified version of Nevill-Manning and Witten's implementation [10], which we changed as follows. We manually converted the C++ code into C, removed the access functions (i.e., we inlined them), 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 employed a split-stream approach, that is, we construct two separate grammars, one for the PC entries and one for the extended data entries in our traces. To limit the memory usage, we start new grammars when eight million unique symbols have been encountered or 384 megabytes of storage have been allocated for rule and symbol descriptors. We found these cutoff points to work well on our traces. According to *ps*, our implementation of *sequitur* never exceeds an overall memory usage of 951MB, which is crucial for good performance on a system with 1GB of main memory. To prevent *sequitur* from becoming very slow due to hash-table inefficiencies, we also start a new grammar whenever the last 65,536 searches required an average of more than thirty trials before an entry was found. Finally, our version of *sequitur* includes a *bzip2* postcompression stage to improve the compression rate. Because of the postcompression and the slowdown-prevention mechanisms, the *sequitur* algorithm we use in this paper is both faster and compresses better than the version we used previously [2].

2.2 Value Predictors

Our *vpc3* algorithm employs the following predictors, which have been experimentally determined to result in good performance on the gcc load-value trace with *bzip2* as the second-stage compressor. See also Section 4.4.

Last-four-value predictor: The first type of predictor we use is the last-four-value predictor (L4V) [3, 22, 36]. It stores the four most recently seen values in a first-in first-out manner. All four values are provided when a prediction is requested, i.e., the predictor can be thought of as comprising four components that make four independent predictions. The L4V can accurately predict sequences of repeating and alternating values as well as repeating sequences of no more than four arbitrary values. Since PCs infrequently exhibit such behavior, we only use the last-four-value predictor for the extended data.

Finite-context-method predictor: The finite-context-method predictor (FCM) [31, 32] computes a hash out of the *n* most recently encountered values. *n* is referred to as the order of the predictor. Whenever a new value is seen, the predictor puts it into a hash table using the current hash as an index. To predict the next value, a hash-table lookup is performed in the hope that the next value will be equal to the value that followed last time the same sequence of *n* previous values (i.e., the same hash) was encountered [28, 29, 31]. Thus, the FCM can memorize long arbitrary sequences of values and accurately predict them when they repeat. This trait makes FCMs ideal for predicting PCs. We use two FCM predictors with orders one and three (called FCM1 and FCM3 henceforth) for predicting PCs. Both FCMs retain and predict the most recent (a) and the second most recent (b) value that map to each line in the hash table. Thus, the FCM1a, FCM1b, FCM3a, and FCM3b predictors provide a total of four predictions. For the extended data, we use an FCM1a and an FCM1b predictor, which provide two predictions.

Differential-finite-context-method predictor: The differential-finite-context-method predictor (DFCM) [9] works just like the FCM with the exception that it predicts and is updated with differences (strides) between consecutive values rather than absolute values. To form the final prediction, the predicted stride has to be added to the most recently seen value.

Again, we use different orders of DFCMs to capture a broader variety of patterns. We use a DFCM1a, DFCM1b, DFCM3a, and DFCM3b for predicting the extended data. The "a" and "b" again refer to the two most recent values in each line of the hash table. DFCMs are often superior to FCMs because they warm up faster, can predict never before seen values, and make better use of the hash table [9]. Unfortunately, these benefits rarely weigh in with PC sequences, which is why we do not use any DFCM predictors to predict PCs.

3. ALGORITHM

3.1 Trace Format

Our traces consist of pairs of numbers. Each pair comprises a PC and an extended data (ED) field. The PC field is 32 bits wide and the extended data field is 64 bits wide. Thus, our traces have the following format, where the subscripts indicate bit widths.

$$\mathbf{PC0_{32}, ED0_{64}, PC1_{32}, ED1_{64}, PC2_{32}, ED2_{64}, \ldots}$$

This trivial format was deliberately chosen to simplify the following presentation of our algorithm. We chose 64 bits for the ED fields because this is the native word size of the Alpha system on which we performed our measurements. 32 bits suffice to represent PCs, especially since we do not need to store the two least significant bits, which are always zero because Alphas only support aligned instructions.

3.2 History

Our first attempt at a value-predictor-based compression algorithm compresses only the extended data and works as follows. The PC of the current PC/ED pair is fed to a set of value predictors that produces *n* (not necessarily distinct) predictions. Each prediction is compared to the ED from the trace. If a match is found, the corresponding predictor identification code is written to the compressed file using a fixed *m*-bit encoding. If no predictor is correct, a special *m*-bit code is written followed by the unpredictable 64-bit extended-data value. Then the predictors are updated. The algorithm repeats for the remaining PC/ED pairs in the trace. Decompression is achieved by running the compression steps in reverse.

Unfortunately, this algorithm does not work well because the PCs are not compressed and because *m* is too large. Since we use 27 predictors plus the special code, five bits are needed to represent a predictor identification code (i.e., *m*=5). Furthermore, unpredictable entries are not compressed. Overall, the algorithm cannot exceed a compression rate of 2.6 because even assuming that every ED entry is predictable, a 96-bit PC/ED pair (32-bit PC plus 64-bit extended data) is merely compressed down to 37 bits (32-bit PC plus five-bit predictor code).

Our *vpc1* algorithm [2] corrects these shortcomings. It compresses the PCs like the ED using a separate set of ten predictors. Moreover, it employs a dynamic Huffman encoder [19, 35] to minimize the number of bits required to express the predictor identification codes. If more than one predictor is correct, we select the one that has the shortest Huffman code associated with it, i.e., the one with the highest usage frequency. Finally, *vpc1* compresses unpredictable values in the following manner. In case of PCs, only $\lceil \log_2(range) \rceil$ bits are written, where the provided *range* has to be greater or equal to the largest PC in the trace. In case of extended data, the special predictor code is followed by the identification code of the predictor whose prediction is closest to the actual value in terms of absolute difference. *Vpc1* then emits the difference between the predicted value and the actual value in encoded sign-magnitude format to save bits.

We added several enhancements to *vpc1* to boost the compression rate. First, we included saturating up/down counters in the hash table of the FCMs to provide an update hysteresis. Second, we retain only distinct values in all multi-value predictors to maximize the number of different predictions and therefore the chances of at least one of them being correct. Third, we keep the values in all multi-value predictors in least recently used order to skew the usage frequency of the predictor components. Skewing the usage frequencies increases the compression rate because it allows the dynamic Huffman encoder to assign shorter identification codes to the frequently used components and to use them more often. Fourth, we initialize the dynamic Huffman encoder with biased, nonzero frequencies for all predictors. This allows the more sophisticated predictors, which perform poorly in the beginning because they take longer to warm up, to stay ahead of the simpler predictors. Thus, biasing the frequencies ensures that the most powerful predictors are used whenever they are correct and the remaining predictors are only utilized occasionally, resulting in shorter Huffman codes and better compression rates.

Vpc1 outperforms *gzip* and *sequitur* on hard-to-compress extended-data traces but is not competitive on easily compressible traces [2]. The reason for this deficiency is that *vpc1* cannot compress traces by more than a factor of 48. This is because at least one bit is needed to encode a PC and one bit to encode an extended-data entry. Since an uncompressed PC/ED pair requires 32+64=96 bits, the maximum compression rate is 96/2=48. Interestingly, *vpc1* almost reaches this theoretical maximum in several cases [2], showing that the dynamic Huffman encoder works well and almost always requires only one bit to encode a predictor identification number. This implies that the same PC and ED predictors are used most of the time because only one PC and one ED predictor can have a one-bit identification code at a time. Since PC and ED predictor codes alternate in the compressed traces, highly-compressible *vpc1* traces contain long bit strings of all zeros, all ones, or alternating zeros and ones, depending on the two predictors' identification codes. This, of course, means that the compressed trace is itself highly compressible.

To exploit this fact, we created *vpc2*, which is *vpc1* with a *gzip* post-compression stage. *Vpc2* works very well and improves upon *vpc1* in all studied cases [2]. More importantly, *vpc2*'s geometric-mean compression rate is almost twice that of *sequitur* and more than twice that of other schemes we tested (the arithmetic mean is even higher). On the downside, *vpc2* is over 3.5 times slower at decompressing traces, which may render it uninteresting for potential users.

3.3 The *VPC3* Algorithm

Based on our experiences with *vpc1* and *vpc2*, we created a brand new algorithm called *vpc3* that is not only much faster but also compresses traces better. *Vpc3* is still based on value predictors, but the innovation is that it does not compress the traces per se. Instead, the purpose of the value predictors is to expose and enhance the patterns in the trace to make the second (compression) stage as effective as possible.

To make *vpc3* faster than *vpc2*, we decided to remove the predictors that did not provide many useful predictions. Thus, we reduced the total number of predictors from 37 to 14. Moreover, we abolished the saturating up/down counters and eliminated the frequency bias and the dynamic Huffman coder, assuming that the compressor in the second stage could do a better job at compressing the predictor identification codes. To further speed up the algorithm, we reverted back to updating the predictors with all values, not just distinct ones. Also, we no longer attempt to compress the unpredictable values in the first stage, which has the pleasant side effect of eliminating the need for the PC range information that had to be provided to *vpc2*. Finally, we converted from bit to byte granularity, which simplified and sped up the code substantially because no more bit shifting is necessary.

Note that we investigated many more enhancements to accelerate *vpc3* but ended up implementing only the ones listed above because they not only make *vpc3* faster but also boost the overall compression rate. For instance, removing infrequently used predictors decreases the number of distinct predictor codes and thus increases the regularity of the emitted codes, resulting in a better compression rate despite the concomitant increase in the number of emitted unpredictable values. Similarly, removing the saturating counters and not updating with distinct values only slightly decreases the prediction accuracy but greatly increases the uniformity of the resulting patterns. Finally, emitting values at byte granularity and thus possibly including unnecessary bits increases the amount of data transferred from the first stage to the second stage but at the same time exposes more patterns and makes them easier to detect for the second stage, which also operates at byte granularity.

The final *vpc3* algorithm converts a trace into four data streams and then compresses the four streams individually using *bzip2* (except where noted otherwise). The four streams are generated as follows. The PC of the current trace entry is read and compared to the four PC predictions (Sections 2.2 and 4.4 describe the predictors). If none of the predictions are correct, a "4" (one byte) is written to the first stream and the unpredictable four-byte PC is written to the second stream. If at least one of the PC predictions is correct, a one-byte predictor identification number ("0", "1", "2", or "3") is written to the first stream and nothing is written to the second stream. If more than one predictor is correct, *vpc3* picks the predictor with the highest use count. The predictors are prioritized to break ties. The ED of the current trace entry is handled analogously. It is read and compared to the ten ED predictions (see Sections 2.2 and 4.4). A one-byte predictor identification number ("0", "1", …, "9") is written to the third stream if at least one of the ED predictions is correct. If none are correct, a "10" is written to the third stream and the unpredictable eight-byte ED is written to the fourth stream. Then all predictors are updated with the true PC or ED and the algorithm repeats until all trace entries have been consumed. Figure 1 illustrates *vpc3*'s operation. The dark arrows in the figure mark the prediction paths while the light arrows mark the update paths. To decompress a trace, *vpc3* essentially runs the compression steps in reverse.

The following is a summary of some ideas we experimented with that turned out to be disadvantageous. They are not included in *vpc3*. (1) The order in which the predictors are accessed and prioritized appears to have no noticeable effect on the performance. (2) Interestingly, biasing the initial use counts of the predictors seems to always hurt the compression rate. (3) Writing differences rather than absolute values to the four streams decreases the compression rate. (4) Larger predictor tables than the ones listed in Section 4.4 do not significantly improve the compression rate (on our traces) and have a negative effect on the memory footprint. (5) Dynamically renaming the predictors such that the predictor with the highest use count always has an identification code of “0”, the second highest a code of “1”, etc., lowers the compression rate. (6) Similarly, decaying the use counts with age, that is, giving more weight to recent behavior, decreases the compression rate.

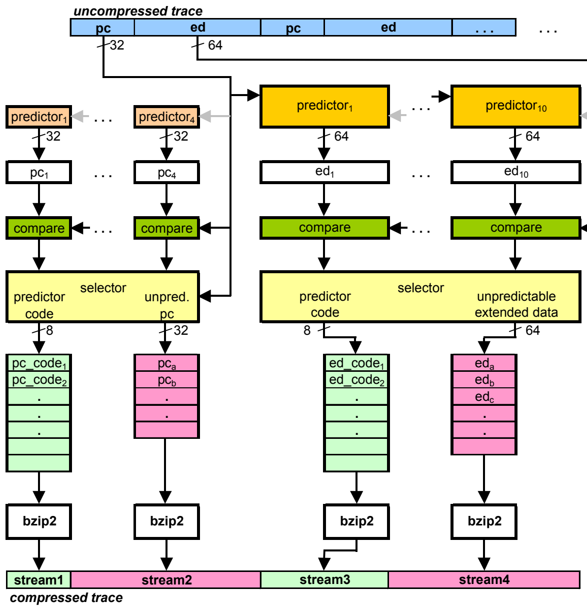

Figure 1: The *vpc3* compression algorithm.

4. EVALUATION METHODS

4.1 System

We performed all measurements for this paper on a dedicated 64-bit CS20 system with two 833MHz 21264B Alpha CPUs [18]. 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 Ultra2/LDV SCSI hard drive has a capacity of 73GB and spins at 10,000rpm. For maximum disk performance, we used the advanced file system (AdvFS). The operating system is Tru64 UNIX V5.1A.

4.2 Traces

We used all integer programs and all but four of the floating-point programs from the SPECcpu2000 benchmark suite [15] 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 instructions from library calls in the traces. Only system-call code is not captured. To generate the traces, we used the binary instrumentation tool-kit ATOM [7, 33] and ran the programs to completion with the SPEC-provided test inputs. Two programs, eon and vpr, require multiple runs and perlbmk executes itself recursively. For each of these programs, we concatenated the subtraces into a single trace.

We generated three types of real-world traces from the 22 programs to evaluate the compression algorithms. The first type of trace captures the PC and the effective address of each executed store instruction. The second type of trace 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).

We selected the store-effective-address traces because, historically, many trace-compression approaches have focused on address traces. 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, and integer numbers, which may make them difficult to compress. After all, a good new compression scheme is particularly important for hard-to-compress traces.

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 we excluded. We excluded all traces with more than one billion entries, i.e., the traces that are larger than twelve gigabytes, because they would have slowed down our experiments needlessly. The corresponding entries in Table 1 are crossed out.

Table 1: Sizes of the studied 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		~~16,769.9 MB~~	726.1 MB	~~23,947.4 MB~~
crafty	C		3,368.0 MB	1,967.8 MB	~~14,227.6 MB~~
gap	C		1,269.2 MB	255.5 MB	3,141.6 MB
gcc	C		2,280.9 MB	366.6 MB	4,523.2 MB
gzip	C		2,836.0 MB	731.3 MB	8,070.1 MB
mcf	C		400.4 MB	150.1 MB	455.7 MB
parser	C		4,224.2 MB	821.2 MB	9,805.9 MB
perlbmk	C		570.3 MB	86.1 MB	1,089.4 MB
twolf	C		239.8 MB	73.4 MB	827.6 MB
vortex	C		~~16,770.6 MB~~	2,185.2 MB	~~26,571.4 MB~~
vpr	C		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		1,781.8 MB	2,381.8 MB	11,249.9 MB
equake	C		1,229.8 MB	418.8 MB	4,323.0 MB
mesa	C		3,671.1 MB	266.8 MB	6,055.2 MB
applu	F77		522.7 MB	77.1 MB	1,002.7 MB
apsi	F77		8,058.2 MB	3,018.9 MB	~~15,911.5 MB~~
mgrid	F77		5,110.0 MB	6,377.0 MB	~~102,794.6 MB~~
sixtrack	F77		~~18,735.9 MB~~	2,224.5 MB	~~38,889.2 MB~~
swim	F77		452.0 MB	149.3 MB	1,985.5 MB
wupwise	F77		10,829.6 MB	889.9 MB	~~22,628.8 MB~~

4.3 Compressors

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). Doing so made *bzip2* 4% faster relative to using the default compiler (*gcc*) and optimization flags, and *gzip* ended up 8.5%

faster than the preinstalled version. We use *bzip2* version 1.0.2 and *gzip* version 1.3.3. Both programs are exclusively used with the "--best" option.

4.4 Predictor Configurations

This section lists the parameters and table sizes of the predictors used by *vpc3*. These configurations have been experimentally determined to yield a good compression rate and speed on the gcc load-value trace. We found the following parameters to work well and use them throughout this paper.

The four PC predictors are global predictors and therefore have no index. The FCM3ab predictor requires three four-byte entries to store three hash values in the first level. These entries are shared with the FCM1ab predictor. The second-level of the FCM1ab (the hash table) has 131,072 lines, each of which holds two four-byte PCs (1MB). The FCM3ab's second-level table is four times larger (4MB). Overall, five megabytes are allocated to the PC predictors.

The predictors for the extended data use the PC modulo the table size as an index, which allows them to store information on a per instruction basis. To enable maximum sharing, all first-level tables have the same number of lines (65,536). The last-four-value predictor retains four eight-byte values per line (2MB). The FCM1ab predictor stores one four-byte hash value per line in the first level (256kB) and two eight-byte values in each of the 524,288 lines in its second-level table (8MB). The DFCM1ab and DFCM3ab predictors share a first-level table, which stores three four-byte hash values per line (768kB). The last value needed to compute the final prediction is obtained from the last-four-value predictor. The second-level table of the DFCM1ab has 131,072 lines, each holding two eight-byte values (2MB). The DFCM3ab's second-level table is four times as large (8MB). The extended data predictors use a total of 21 megabytes of table space.

Overall, 26 megabytes are allocated to the predictor tables in our compression algorithm. Including the code, stack, etc., our compression utility requires 27MB of memory to run as reported by the UNIX command *ps*.

5. RESULTS

The following sections describe the results. Section 5.1 discusses the compression rate, Section 5.2 studies the decompression speed, Section 5.3 investigates the compression speed, and Section 5.4 takes a look at the value-predictor performance.

5.1 Compression Rate

Table 2 shows the compression rates of *gzip*, *bzip2*, *sequitur* (seq), *vpc2*, and *vpc3* on the three sets of traces we generated (higher numbers are better). For each type of trace, we highlighted the best compression rate in bold print. Three stars mark excluded traces.

On the store-effective-address traces, *vpc3* exceeds the compression rate of the other algorithms for the majority of the traces. It compresses art almost seven hundred times more than *sequitur* does and reaches a compression rate of over 36,000, the highest we observed. *Sequitur* outperforms *vpc3* on eon, crafty, twolf, and vpr, though never by more than a factor of three. *Vpc3*'s geometric mean compression rate is over eight times that of *sequitur*, and the general-purpose algorithms *gzip* and *bzip2* fare even worse. *Vpc3* is about twice as effective on average as its predecessor *vpc2*. The latter is superior to the former in two cases, notably on mgrid, which it compresses over six times more than any of the other schemes. Evidently, this trace greatly benefits from a predictor in *vpc2* that is not included in *vpc3*. There is no case where *gzip* or *bzip2* exceeds *vpc3*'s compression rate. However, *bzip2* outperforms *sequitur* on seven store-effective-address traces. Note how much more compressible the floating-point traces are than the integer traces, especially with the two *vpc* algorithms.

Looking at the cache-miss-address traces, we find that the compression rates are generally much lower, which is what we expected. While there is still no case where *gzip* outperforms *vpc3*, there are now five instances (all integer program traces) where *bzip2* yields a better compression rate than *vpc3*. However, the difference between *bzip2* and *vpc3* is less than a factor of two in each case. Surprisingly, *bzip2* exceeds *sequitur*'s compression rate on almost half of the cache-miss-address traces. *Vpc3* is inferior to *sequitur* on six traces, three of which are most compressible by *sequitur*. In fact, *sequitur* compresses sixtrack almost six times better than any of the other algorithms. *Vpc3* underperforms *vpc2* in three instances by less than 7% (including eon) and in two more instances by less than a factor of two. Overall, *vpc3* exceeds *sequitur*, *bzip2*, and *gzip*'s compression rates on two thirds of the cache-miss-address traces and by a factor of 1.8 on average (geometric mean). Again, the floating-point traces are more compressible by all schemes than the integer traces, although the discrepancy is not as high as with the store-effective-address traces.

On the load-value traces, *vpc3* outperforms all the other algorithms we evaluated on every trace with the exception of twolf (where *sequitur* is 7% better) and swim (where *vpc2* is 30% better). We surmise that *vpc3* works so well on these traces because we used one of them, gcc, to tune our algorithm. Maybe this indicates that with finer tuning, *vpc3* could perform even better on the other types of traces. *Vpc3*'s geometric mean compression rate is 1.7 times that of *sequitur*, the best non-*vpc* algorithm, on the load-value traces. One interesting point to note is that, with the exception of mesa (and to a much smaller degree art), the load-value floating-point traces are less compressible than their integer counterparts.

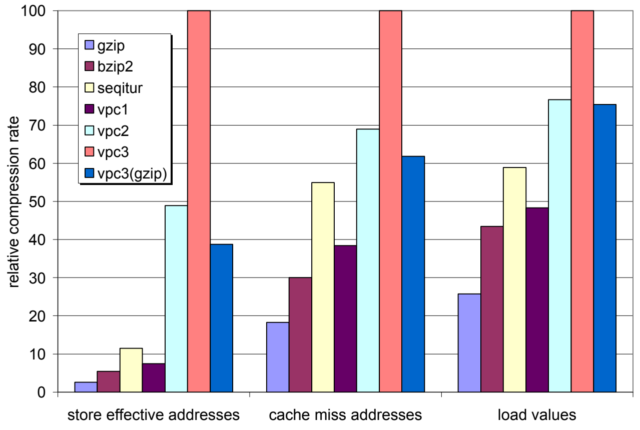

Figure 2: Gmean compression rates relative to *vpc3*.

Figure 2 depicts the geometric mean compression rates over the three types of traces relative to *vpc3*, whose compression rate is normalized to one hundred. The figure includes results from two additional algorithms, *vpc1* and *vpc3(gzip)*, the latter of which uses *gzip* instead of *bzip2* as the second-stage compressor.

Table 2: Compression rates.

	store effective addresses					cache miss addresses					load values				
	gzip	bzip2	seq	vpc2	vpc3	gzip	bzip2	seq	vpc2	vpc3	gzip	bzip2	seq	vpc2	vpc3
eon	6.8	24.3	**793.4**	92.9	366.9	13.1	**30.0**	22.3	17.4	17.4	7.3	10.8	14.5	16.0	**23.8**
bzip2	***	***	***	***	***	3.9	5.1	4.0	4.6	**5.8**	***	***	***	***	***
crafty	15.3	37.1	**102.7**	58.8	82.9	9.1	**19.2**	14.2	12.9	14.8	***	***	***	***	***
gap	9.2	17.8	15.5	70.9	**76.3**	7.1	10.6	6.5	8.2	**11.0**	11.1	21.7	26.9	35.3	**51.0**
gcc	10.9	22.6	42.2	53.1	**67.3**	5.4	7.2	6.3	5.2	**9.0**	8.3	15.7	22.6	24.6	**28.9**
gzip	11.3	20.5	19.6	60.8	**64.0**	5.4	7.3	7.9	7.0	**8.4**	8.8	13.8	12.1	12.9	**19.6**
mcf	6.4	22.7	14.2	74.1	**131.1**	5.1	9.8	12.5	12.0	**17.1**	6.8	13.7	18.4	13.7	**20.3**
parser	9.1	14.2	18.4	**57.9**	56.6	5.6	6.3	7.3	8.0	**11.5**	11.9	18.8	21.2	33.1	**34.5**
perlbmk	13.6	25.5	47.9	73.0	**117.2**	10.1	**17.3**	13.2	12.5	17.0	11.7	24.7	42.1	40.9	**62.4**
twolf	11.8	34.5	**126.9**	25.3	42.7	6.6	**15.9**	11.7	6.7	10.5	6.9	14.8	**21.9**	16.2	20.5
vortex	***	***	***	***	***	10.3	20.1	**22.4**	22.3	21.2	***	***	***	***	***
vpr	9.2	21.1	**43.9**	20.4	34.9	4.2	**7.6**	6.9	3.7	6.7	9.6	18.0	19.2	17.5	**27.6**
ammp	20.0	33.1	38.8	473.4	**980.4**	5.7	18.1	42.0	64.9	**65.0**	***	***	***	***	***
art	9.0	22.9	52.0	4017.8	**36248.6**	6.3	9.1	218.6	413.5	**6088.7**	21.4	30.7	55.2	58.9	**85.9**
equake	27.6	50.5	48.7	331.2	**383.4**	7.8	9.4	24.3	**30.0**	28.2	8.3	11.0	13.2	20.7	**25.6**
mesa	17.8	53.5	134.0	1540.6	**4341.1**	27.1	67.5	78.3	189.3	**304.8**	41.2	117.9	293.0	1996.4	**2750.5**
applu	9.9	14.0	500.8	856.5	**1787.4**	5.1	5.9	**176.5**	37.9	66.2	3.2	3.4	4.3	5.5	**6.6**
apsi	25.7	39.0	226.0	398.0	**13123.5**	6.4	9.9	285.8	163.4	**789.3**	***	***	***	***	***
mgrid	8.1	14.2	13.3	**3407.0**	557.8	6.0	7.8	6.9	**208.3**	108.5	***	***	***	***	***
sixtrack	***	***	***	***	***	7.9	11.4	**175.0**	26.4	29.8	***	***	***	***	***
swim	7.5	11.6	9.6	7544.2	**18638.4**	5.9	8.4	21.0	206.9	**359.3**	2.9	3.3	4.0	**8.6**	6.7
wupwise	44.8	118.8	101.0	779.1	**5645.3**	6.5	10.2	6.2	**80.2**	50.8	***	***	***	***	***
a mean	*14.4*	*31.5*	*123.6*	*1049.2*	***4355.0***	*7.8*	*14.3*	*53.2*	*70.1*	***365.5***	*11.4*	*22.7*	*40.6*	*164.3*	***226.0***
g mean	*12.4*	*26.3*	*55.5*	*237.2*	***484.7***	*7.0*	*11.5*	*21.0*	*26.3*	***38.2***	*9.1*	*15.3*	*20.7*	*27.0*	***35.2***
h mean	*11.1*	*23.1*	*31.9*	*85.7*	***126.0***	*6.5*	*10.0*	*12.0*	*12.9*	***17.6***	*7.5*	*10.8*	*13.6*	*17.1*	***20.6***

Table 3: Decompression time in minutes.

	store effective addresses					cache miss addresses					load values				
	gzip	bzip2	seq	vpc2	vpc3	gzip	bzip2	seq	vpc2	vpc3	gzip	bzip2	seq	vpc2	vpc3
eon	0.8	3.9	0.7	7.6	1.4	0.03	0.19	0.12	0.57	0.15	0.8	4.7	2.7	10.4	2.6
bzip2	***	***	***	***	***	0.36	2.02	2.51	5.69	2.71	***	***	***	***	***
crafty	1.0	6.1	2.1	12.7	2.5	0.68	4.21	3.63	12.77	3.14	***	***	***	***	***
gap	0.4	2.7	2.1	4.8	1.0	0.10	0.59	0.65	1.67	0.47	1.0	6.0	3.1	14.1	3.1
gcc	0.7	4.3	1.6	9.0	1.8	0.16	0.91	0.96	2.77	0.79	1.6	8.9	5.1	23.1	5.2
gzip	0.9	5.9	3.1	9.7	2.2	0.31	1.85	1.57	4.61	1.68	2.8	17.8	12.4	40.2	10.8
mcf	0.2	0.9	1.3	1.4	0.3	0.07	0.33	0.32	0.92	0.24	0.2	0.9	0.6	2.7	0.6
parser	1.5	8.5	4.9	15.5	3.3	0.37	1.97	1.90	5.35	1.46	3.2	18.1	10.6	41.3	9.4
perlbmk	0.2	1.1	0.5	2.3	0.5	0.03	0.19	0.14	0.53	0.14	0.4	2.1	0.9	4.8	1.1
twolf	0.1	0.4	0.1	1.1	0.3	0.03	0.16	0.14	0.59	0.16	0.3	1.7	1.0	4.8	1.1
vortex	***	***	***	***	***	0.72	4.69	2.88	12.50	3.27	***	***	***	***	***
vpr	0.7	4.0	1.6	9.4	2.1	0.30	1.67	1.72	6.09	1.93	2.5	14.1	8.8	34.1	8.1
ammp	1.4	9.5	3.8	13.3	2.9	1.50	7.08	3.12	13.76	2.93	***	***	***	***	***
art	0.7	2.9	1.4	4.7	0.9	1.08	4.96	1.15	7.99	1.55	3.1	21.1	6.6	36.4	7.9
equake	0.3	2.3	0.7	3.7	0.7	0.15	0.98	0.41	1.78	0.45	1.5	9.8	5.5	17.1	4.4
mesa	1.1	7.1	1.6	10.6	1.9	0.07	0.41	0.15	0.69	0.13	1.5	10.7	2.2	19.5	3.5
applu	0.2	1.1	0.2	1.7	0.3	0.03	0.19	0.04	0.38	0.08	0.5	3.4	2.7	5.3	2.1
apsi	2.2	13.4	3.6	22.2	3.8	1.22	5.97	1.54	9.00	1.91	***	***	***	***	***
mgrid	2.3	12.3	14.5	15.5	3.1	2.90	16.04	16.02	24.69	5.85	***	***	***	***	***
sixtrack	***	***	***	***	***	0.81	5.18	1.25	13.05	2.93	***	***	***	***	***
swim	0.2	1.1	1.4	1.4	0.2	0.07	0.40	0.17	0.57	0.09	1.1	7.2	6.2	8.9	4.0
wupwise	2.8	18.6	6.5	29.0	6.0	0.39	2.23	2.89	3.52	0.84	***	***	***	***	***
a mean	*0.93*	*5.59*	*2.72*	*9.23*	*1.86*	*0.52*	*2.83*	*1.97*	*5.89*	*1.50*	*1.46*	*9.03*	*4.89*	*18.76*	*4.56*
g mean	*0.60*	*3.61*	*1.56*	*6.20*	*1.25*	*0.23*	*1.30*	*0.80*	*3.00*	*0.76*	*1.06*	*6.42*	*3.44*	*13.53*	*3.39*
h mean	*0.36*	*2.13*	*0.77*	*3.80*	*0.76*	*0.10*	*0.59*	*0.28*	*1.43*	*0.33*	*0.69*	*4.00*	*2.18*	*9.27*	*2.32*

* The stars mark excluded traces.

Overall, *vpc3* is clearly dominant and delivers the best compression rates by a large margin. *Sequitur*, the best non-*vpc* algorithm we studied, reaches roughly half the compression rate of *vpc3* on the cache-miss-address and the load-value traces but only reaches 11.5% of vpc3's compression rate on the store-effective-address traces. Because we tuned *vpc3* to perform well in combination with *bzip2* and because of *bzip2*'s general advantage over *gzip* (there is no trace on which *gzip* outperforms *bzip2*), we expect *vpc3(gzip)* to be inferior to *vpc3*. Nevertheless, *vpc3(gzip)* has its merits, as we will see in Section 5.2. Also, it is interesting to compare *vpc3(gzip)* with *vpc2* because both algorithms use *gzip* in their second stage. In fact, *vpc2* compresses all three types of address traces better than *vpc3(gzip)*. In a sense, this loss in compression rate relative to *vpc2* is the price we have to pay for the much faster speed (see next section).

5.2 Decompression Speed

Table 3 shows the time it takes *gzip*, *bzip2*, *sequitur* (seq), *vpc2*, and *vpc3* to decompress the traces and write the resulting (uncompressed) traces back to disk (lower numbers are better). The times listed are in minutes and represent the sum of the user and the system time as reported by the UNIX shell command *time*.

Note that the results listed in this and the next section include the effects of caching disk data. Hence, part of an input file may be sourced from the disk cache, and it is likely that not all of the generated data will actually have been written back to the disk at the time the algorithms terminate. However, this effect should be limited due to the large sizes of our traces.

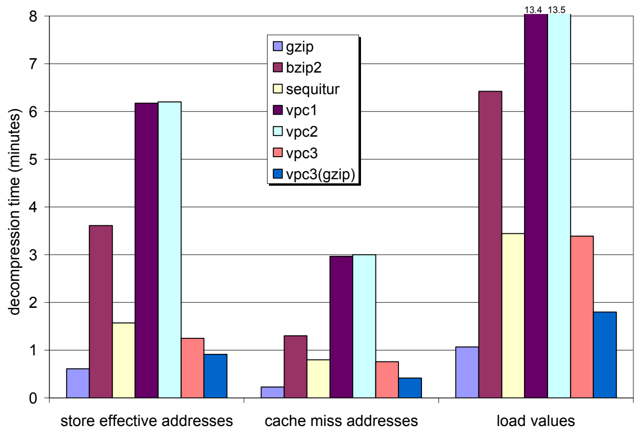

Figure 3: Geometric mean decompression time.

Vpc3 is four to five times faster than *vpc2* on average and over twice as fast on every trace. Moreover, it is about 1.7 to three times as fast as *bzip2*. This is somewhat surprising since *vpc3* uses *bzip2* in its second stage. Obviously, our idea of using value predictors to bring out and enhance the patterns in the traces not only drastically improves the compressibility but also makes it possible to decompress the traces faster. *Bzip2* exceeds *vpc3*'s decompression speed somewhat on three of our 55 traces (on the bzip2, twolf and vpr cache-miss-address traces). *Sequitur*'s decompression speed is largely comparable to that of *vpc3* but is slightly worse on the store-effective-address traces. *Gzip* is generally two to three times as fast at decompression as *vpc3* but does not deliver competitive compression rates.

If decompression speed is more important than an excellent compression rate, *vpc3(gzip)* should be considered. As Figure 3 shows, it is significantly faster than *vpc3* (almost twice as fast on the load-value and the cache-miss-address traces), but still compresses the traces substantially more than *gzip*, *bzip2*, and *sequitur* (see Figure 2).

Note that *vpc3* regenerates the store-effective-address traces at an average speed of 25 MB/s (megabytes per second), the cache-miss-address traces at 14 MB/s, and the load-value traces at 15 MB/s. These rates exceed the throughput of a 100 megabit per second network connection and the transfer rates of many hard disks. Moreover, note that these rates are probably limited by the speed at which the decompressed trace can be written back to the disk. This suggests that it may well be faster to read and decompress a compressed trace than to read the corresponding uncompressed trace when driving simulators or other trace-consumption tools.

5.3 Compression Speed

Figure 4 shows the geometric mean time it takes the various algorithms to compress the traces and write the compressed traces back to disk (lower numbers are better).

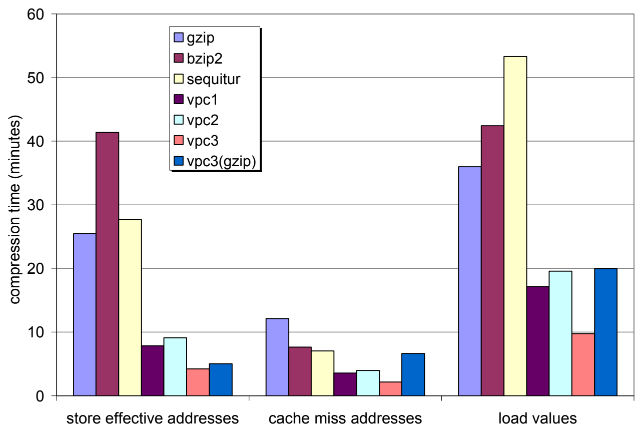

Figure 4: Geometric mean compression time.

Vpc3 is the fastest compressor for all three types of traces. This is surprising because it usually takes longer to achieve higher compression rates. While fast compression is generally not as critical as a good compression rate and decompression speed, it is certainly an added bonus. Once again, our value-prediction-based technique appears to be the right approach. On average, *vpc3* is 3.5 to ten times as fast at compressing the traces as *bzip2* and 3.3 to 6.6 times as fast as *sequitur*. It compresses traces at an unsurpassed rate of five to 7.5 megabytes per second.

5.4 Predictor Usage

Table 4 lists the usage distribution of the various predictor components in *vpc3*. The top portion of the table shows results for the PC predictors and the bottom portion for the ED predictors. The numbers are arithmetic means over the traces in each set.

In all cases, the PC trace entries are more predictable than the extended data, despite the smaller number of predictors. This indirectly shows that extended traces are harder to compress than PC-only traces. The PC predictability is very high for the store-effective-address and the load-value traces, where we trace every executed store or load instruction, respectively. The PC predictions for the cache-miss-address traces, on the other hand, are more spread across the four predictors and a six times larger fraction of the PCs is not predicted in these traces than with the other two types of traces. This verifies our assumption that the simulated cache filters out many of the patterns in the store-effective-address traces.

The cache-miss-address traces are also the least predictable on the extended-data side. However, there the load-value traces cause the widest spread of predictor usage. This is probably due to the large variety of data that load instructions fetch, which includes addresses as well as floating-point and integer values.

Overall, *vpc3* correctly predicts between about 82% and 96% of the ED trace entries and all predictors contribute predictions.

Table 4: Predictor usage (in percent).

	predictor	strore addrs	cache addrs	load values
PCs	FCM1a	93.0	63.0	94.2
	FCM1b	3.1	6.0	3.0
	FCM3a	0.3	9.9	0.1
	FCM3b	1.0	5.6	0.5
	not predicted	2.7	15.5	2.3
extended data	DFCM1a	88.3	60.0	20.1
	DFCM1b	0.5	1.0	0.2
	DFCM3a	1.9	2.4	4.2
	DFCM3b	0.3	2.6	2.1
	L4Va	0.5	0.3	40.2
	L4Vb	0.2	2.6	1.7
	L4Vc	0.3	1.2	1.0
	L4Vd	0.4	0.6	1.2
	FCM1a	3.2	8.1	9.9
	FCM1b	0.7	3.2	2.7
	not predicted	3.9	18.1	16.6

6. FUTURE WORK

In the future, we intend to study traces from other programs, traces containing different information, and traces from non-Alpha-based platforms to further evaluate and improve our compression algorithm. We also plan to investigate additional compression schemes, both generic ones and ones that we adapt to take advantage of our trace format. For example, lzop [14] is of interest because of its high compression and decompression speed. Moreover, we would like to investigate whether our approach is useful in contexts outside of instruction traces.

Zhang and Gupta improved the compression rate of *sequitur* by splitting traces up by functions, i.e., they generate a subtrace for each function in the program (called a path trace) and then compress the subtraces individually [38]. We believe the same approach can be used to further improve the compression rate of *vpc3*.

Another interesting idea, whose applicability to our algorithm we would like to investigate, is Chilimbi's hot data streams [5]. He uses *sequitur* to produce a series of traces with increasing compactness but lower precision. We will study the usefulness of our traces when certain trace entries, e.g., all the last-value predictable ones, are omitted.

Another possible extension of this work is to study the usefulness of special instructions to support compression and decompression in hardware, which could make our algorithm even faster.

Finally, we believe a hybrid scheme that uses one algorithm to compress the PCs and a different algorithm to compress the extended data would likely result in the best overall compression rates. In particular, it seems like sequitur should be used to compress the PCs and our algorithm for the extended data. We will investigate such an approach.

7. CONCLUSIONS

This paper presents a novel approach to compressing program traces, in particular traces that contain extended data such as register values or effective addresses. Our approach, called *vpc3*, uses a set of value predictors to convert the trace into four data streams that are much more compressible than the original trace. Moreover, the streams can be compressed and decompressed faster. For example, our scheme compresses SPECcpu2000 traces of store-instruction PCs and effective addresses up to 1600 times (8.7 times geometric mean) as much as *gzip*, *bzip2*, and *sequitur*, even though we modified *sequitur* to take advantage of our trace format. Additionally, our algorithm compresses the traces faster than the other three algorithms and decompresses them faster than *bzip2* and *sequitur*. Based on these results, we feel our approach is ideal for trace databases as well as any research and teaching environment where traces are used.

In addition to the above-mentioned qualities, *vpc3* features a single-pass linear-time algorithm and a fixed memory requirement. It is modular and extensible, making it easy to add and remove predictor components, allowing users to adapt the scheme to exploit additional patterns. The source code of our compression algorithm and a brief tutorial are available at http://www.csl.cornell.edu/~burtscher/research/tracecompression/.

8. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under Grants No. 0208567 and 0312966. We would like to thank Metha Jeeradit for his contributions to *vpc1* and Chris Fraser and the reviewers for their valuable comments.

9. REFERENCES

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

[2] 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.

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

[4] M. Burtscher and B. G. Zorn. "Hybrid Load-Value Predictors." *IEEE Transactions on Computers*, Vol. 51, No. 7, pp. 759-774. July 2002.

[5] T. M. Chilimbi. "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality." *Conference on Programming Language Design and Implementation*, pp. 191-202. June 2001.

[6] E. N. Elnozahy. "Address Trace Compression Through Loop Detection and Reduction." *International Conference on Measurement and Modeling of Computer Systems*, Vol. 27, No. 1, pp. 214-215. May 1999.

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

[8] F. Gabbay. *Speculative Execution Based on Value Prediction*. EE Department Technical Report #1080, Technion - Israel Institute of Technology. November 1996.

[9] 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.

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

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

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

[13] http://www.gzip.org/

[14] http://www.oberhumer.com/opensource/lzo/

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

[16] 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.

[17] E. E. Johnson, J. Ha and M. B. Zaidi. "Lossless Trace Compression." *IEEE Transaction on Computers*, Vol. 50, No. 2, pp. 158-173. February 2001.

[18] 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.

[19] D. E. Knuth. "Dynamic Huffman Coding." *Journal of Algorithms*, Vol. 6, pp. 163-180. 1985.

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

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

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

[23] 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.

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

[25] 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.

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

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

[28] G. Reinman and B. Calder. "Predictive Techniques for Aggressive Load Speculation." *31st International Symposium on Microarchitecture*, pp. 127-137. December 1998.

[29] B. Rychlik, J. W. Faistl, B. P. Krug and J. P. Shen. "Efficacy and Performance Impact of Value Prediction." *International Conference on Parallel Architectures and Compilation Techniques*, pp. 148-154. October 1998.

[30] 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.

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

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

[33] 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.

[34] D. Tullsen and J. Seng. "Storageless Value Prediction Using Prior Register Values." *26th International Symposium on Computer Architecture*, pp. 270-279. May 1999.

[35] J. S. Vitter. "Design and Analysis of Dynamic Huffman Codes." *Journal of the ACM*, Vol. 34, No. 4, pp. 825-845. October 1987.

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

[37] T. A. Welch. "A Technique for High-Performance Data Compression." *IEEE Computer*, pp. 8-19. June 1984.

[38] 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.

[39] J. Ziv and A. Lempel. "A Universal Algorithm for Data Compression." *IEEE Transaction on Information Theory*, Vol. 23, No. 3, pp. 337-343. May 1977.