The VPC Trace-Compression Algorithms

Martin Burtscher, Ilya Ganusov, Sandra J. Jackson, Jian Ke, Paruj Ratanaworabhan, and Nana B. Sam
Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University, Ithaca, NY 14853
{burtscher, ilya, sandra, jke, paruj, besema}@csl.cornell.edu

ABSTRACT

Execution traces, which are used to study and analyze program behavior, are often so large that they need to be stored in compressed form. This paper describes the design and implementation of four value prediction based compression (VPC) algorithms for traces that record the PC as well as other information about executed instructions. VPC1 directly compresses traces using value predictors, VPC2 adds a second compression stage, and VPC3 utilizes value predictors to convert traces into streams that can be compressed better and more quickly than the original traces. VPC4 introduces further algorithmic enhancements and is automatically synthesized. Of the 55 SPECcpu2000 traces we evaluate, VPC4 compresses 36 better, decompresses 26 faster and compresses 53 faster than BZIP2, MACHE, PDATS II, SBC and SEQUITUR. It delivers the highest geometric-mean compression rate, decompression speed and compression speed because of the predictors' simplicity and their ability to exploit local value locality. Most other compression algorithms can only exploit global value locality.

Keywords

E.4.a Data Compaction and Compression, B.8.2 Performance Analysis and Design Aids

1. INTRODUCTION

Execution traces are widely used in industry and academia to study the behavior of programs and processors. The problem is that traces from interesting applications tend to be very large. For example, collecting just one byte of information per executed instruction generates on the order of a gigabyte of data per second of CPU time on a high-end microprocessor. Moreover, traces from

many different programs are typically collected to capture a wide variety of workloads. Storing the resulting multi-gigabyte traces can be a challenge, even on today's large hard disks.

One solution is to regenerate the traces every time they are needed instead of saving them. For example, tools like ATOM [11], [44] can create instrumented binaries that produce a trace every time they are executed. However, execution-driven approaches are ISA specific and thus hard to port, produce non-repeatable results for nondeterministic programs, and are undesirable in situations where trace regeneration is impossible, too expensive, or too slow.

An alternative solution is to generate the traces once and to store them (in compressed form). The disadvantage of this approach is that it requires disk space and that the traces must be decompressed before they can be used. The VPC trace compression algorithms presented in this paper aim at minimizing these downsides.

To be useful, a trace compressor has to provide several benefits in addition to a good compression rate and a fast decompression speed. For example, a fast compression speed may also be desirable. Moreover, lossless compression is almost always required so that the original trace can be reconstructed precisely. A constant memory footprint guarantees that a system can handle even the largest and most complicated traces. Finally, a single-pass algorithm is necessary to ensure that the large uncompressed trace never has to exist in its entirety because it can be compressed as it is generated and decompressed as simulators or other tools consume it. VPC3 [4] and VPC4 [6], our most advanced algorithms, possess all these qualities.

Many trace-compression algorithms have been proposed [2], [4], [10], [15], [16], [23], [24], [25], [28], [29], [32], [33], [38], [41], [48]. Most of them do an excellent job at compressing program traces that record the PCs of executed instructions (*PC traces*) or the addresses of memory accesses (*address traces*). However, *extended traces* that contain PCs plus additional information such as the content of registers, values on a bus, or even filtered address sequences are harder to compress because such data repeat less, exhibit fewer patterns, or span larger ranges than the entries in PC and address traces. Yet, such traces are gaining importance as more and more researchers investigate the dynamic activities in computer systems. We designed the VPC algorithms [4], [5], [6] especially for extended traces.

A comparison with SEQUITUR [29] and SBC [33], two of the best trace-compression algorithms in the current literature, on the three types of traces we generated from the SPECcpu2000 programs yielded the following results. On average (geometric mean), VPC4 outperforms SBC

by 79% in compression rate, 1133% in compression time, and 25% in decompression time on PC plus store-address traces and by 88% in compression rate, 850% in compression time, and 27% in decompression time on PC plus load-value traces. It outperforms SEQUITUR on average by 1356% in compression rate, 668% in compression time, and 36% in decompression time on the PC plus store-address traces and by 105% in compression rate, 535% in compression time, and 6% in decompression time on the PC plus load-value traces. Section 6 presents more results.

The VPC algorithms all have a set of value predictors at their core. Value predictors identify patterns in value sequences and extrapolate those patterns to forecast the likely next value. In recent years, hardware-based value predictors have been researched extensively to predict the content of CPU registers [7], [8], [13], [14], [31], [40], [43], [45], [47]. Since these predictors are designed to make billions of predictions per second, they use simple and fast algorithms. Moreover, they are good at predicting the kind of values typically found in program traces such as jump targets, effective addresses, etc., because such values are stored in registers. These features make value predictors great candidates for our purposes. In fact, since we are implementing the predictors in software, we can use more predictors and larger tables than are feasible in a hardware implementation, resulting in even more accurate predictions.

The following simple example illustrates how value predictors can be used to compress traces. Let us assume that we have a set of predictors and that our trace contains eight-byte entries. During compression, the current trace entry is compared with the predicted values. If at least one of the predictions is correct, we write only the identification code of one of the correct predictors to the output, encoded in a one-byte value. If none of the predictions are right, we emit a one-byte flag followed by the unpredictable eight-byte trace entry. Then the predictors are updated and the procedure repeats until all trace entries have been processed. Note that the output of this algorithm should be further compressed, e.g., with a general-purpose compressor.

Decompression proceeds analogously. First, one byte is read from the compressed input. If it contains the flag, the next eight bytes are read to obtain the unpredictable trace entry. If, on the other hand, the byte contains a predictor identification code, the value from that predictor is used. Then the predictors are updated to ensure that their state is consistent with the corresponding state during compression. This process iterates until the entire trace has been reconstructed.

VPC1 essentially embodies the above algorithm. It uses 37 value predictors, compresses the predictor identification codes with a dynamic Huffman encoder, and employs a form of differen-

tial encoding to compress the unpredictable trace entries [5]. VPC1 delivers good compression rates on hard-to-compress extended traces but is not competitive on other kinds of traces.

As it turns out, VPC1's output itself is quite compressible. Thus, VPC2 improves upon VPC1 by adding an additional compression stage [5]. It delivers good compression rates on hard-to-compress as well as simple extended traces. Unfortunately, VPC2's decompression speed is almost four times slower than that of other algorithms.

VPC3 capitalizes on the fact that extended traces are more compressible after processing them with value predictors. Hence, we changed the purpose of the value predictors from compressing the traces to converting them into streams that a general-purpose compressor can quickly compress well [4]. Using the value predictors in this manner is more fruitful than using them for the actual compression, as the compression rates and speeds in Section 6 show.

VPC4 adds several algorithmic enhancements to make it faster and to improve the compression rate. Moreover, it is not handcrafted like its predecessors. Instead, it is synthesized out of a simple description of the trace format and the predictors to be used [6].

Most data compressors comprise two general components, one to recognize patterns and the other to encode a representation of these patterns into a small number of bits. The VPC algorithms fit this standard compression paradigm, where the value predictors represent the first component and the general-purpose compressor represents the second. The innovation is the use of value predictors as pattern recognizers, which results in a powerful trace compressor for the following reason. Extended traces consist of records with a PC field and at least one extended data (ED) field. The value predictors separate the data component of extended traces into mini-streams, one for each distinct PC (modulo the predictor size). In other words, the prediction of the next ED entry is not based on the immediately preceding *n* trace entries but on the preceding *n* entries with the same PC. This allows VPC to correlate data from one instruction with data that that same instruction produced rather than with data from 'nearby' but otherwise unrelated instructions. As a result, the mini-streams exhibit more value locality than the original trace in which the data are interleaved in complicated ways. The VPC algorithms are thus able to detect and exploit patterns that other compressors cannot, which explains VPC's performance advantage on extended traces. Note that the value predictors are essential because they can easily handle tens of thousands of parallel mini-streams. While other algorithms could also split the traces into mini-streams, it would be impractical, for example, to simultaneously run ten thousand in-

stances of BZIP2 or to concurrently generate ten thousand grammars in SEQUITUR. Nevertheless, Zhang and Gupta have been able to improve the compression rate of SEQUITUR by taking a step in this direction. They generate a subtrace for each function and then compress the subtraces individually [49].

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 of the algorithm are also included. TCgen, the tool that generated VPC4, is available at http://www.csl.cornell.edu/~burtscher/research/TCgen/. The code has been successfully tested on 32- and 64-bit UNIX and Linux systems using *cc* and *gcc* as well as on Windows under *cygwin* [19].

The remainder of this paper is organized as follows. Section 2 summarizes related work. Section 3 introduces our trace format and the value predictors VPC uses. Section 4 describes the four VPC algorithms in detail. Section 5 presents the evaluation methods. Section 6 explains the results, and Section 7 concludes the paper.

2. RELATED WORK

A large number of compression algorithms exists. Due to space limitations, we mainly discuss lossless algorithms that have been specifically designed to compress traces. The algorithms we evaluate in the result section are described in Section 2.2.

VPC shares many ideas with the related work described in this section. For example, it exploits sequentiality and spatiality, predicts values, separates streams, matches previously seen sequences, converts absolute values into offsets, and includes a second compression stage.

2.1 Compression Approaches

Larus proposed Abstract Execution (AE) [28], a system designed mainly for collecting traces of memory addresses and data. AE starts by instrumenting the program to be traced. However, instead of instrumenting it to collect a full trace, it only collects significant events (SE). AE then generates and compiles a program that creates the complete trace out of the SE data. Normally, the SE trace size is much smaller than the size of the full trace, and the cost of running the program to collect the SEs is much lower than the cost of running the program to collect the complete trace. The AE system is language and platform specific.

PDATS [24] compresses address traces in which a reference type (data read, data write, or in-

struction fetch) distinguishes each entry. It converts the absolute addresses into offsets, which are the differences between successive references of the same type, and uses the minimum number of bytes necessary to express each offset. Each entry in the resulting PDATS file consists of a header byte followed by a variable-length offset. The header byte includes the reference type, the number of bytes in the offset, and a repeat count. The repeat count is used to encode multiple successive references of the same type and offset. The result is fed into a second compressor.

Pleszkun designed a two-pass algorithm to compress traces of instruction and data addresses [38]. The first pass partially compresses the trace, generates a collection of data structures, and records the dynamic control flow of the original program. During the second pass, this information is used to encode the dynamic basic block successors and the data offsets using a compact representation. At the end, the output is passed through a general-purpose compressor.

To handle access pattern irregularities in address traces due to, for example, operating system calls or context switches, Hammami [15] proposes to first group the addresses in a trace into clusters using machine-learning techniques. Then the 'mean' of each cluster is computed. This mean forms the base address for that cluster. Finally, the addresses within each cluster are encoded as offsets to the corresponding base address. This approach can be used, for instance, on top of PDATS but increases the number of passes over the trace.

Address Trace Compression through Loop Detection and Reduction [10] is a multi-pass algorithm that is geared towards address traces of load/store instructions. This compression scheme is divided into three phases. In the first phase, control-flow analysis techniques are used to identify loops in the trace. In the second phase, the address references within the identified loops are grouped into three types: (1) constant references, which do not change from one iteration to the next, (2) loop varying references, which change by a constant offset between consecutive iterations, and (3) chaotic references, which do not follow any discernible patterns. In the last phase, the first two types of address references are encoded while the third type is left unchanged.

Johnson et al. introduce the PDI technique [25], which is an extension of PDATS for compressing traces that contain instruction words in addition to addresses. In PDI, addresses are compressed using the PDATS algorithm, while the instruction words are compressed using a dictionary-based approach. The PDI file contains a dictionary of the 256 most common instruction words found in the trace, which is followed by variable length records that encode the trace entries. Identifying the most frequent instruction words requires a separate pass over the trace. PDI

employs a general-purpose compressor to boost the compression rate.

Ahola proposes RECET [1], a hardware and software platform to capture and process address traces and to perform cache simulations in real-time. RECET first sends the traces through a small cache to filter out hits, stores the remaining data in the MACHE format and further compresses them with the Lempel-Ziv method [50]. The resulting traces can be used to simulate caches with line sizes and associativities that are equal to or larger than that of the filter cache.

Hamou-Lhadj and Lethbridge [16] use common subexpression elimination to compress procedure-call traces. The traces are represented as trees, and subtrees that occur repeatedly are eliminated by noting the number of times each subtree appears in the place of the first occurrence. A directed acyclic graph is used to represent the order in which the calls occur. This method requires a preprocessing pass to remove simple loops and calls generated by recursive functions. One benefit of this type of compression is that it can highlight important information contained in the trace, thus making it easier to analyze.

STEP [2] provides a standard method of encoding general trace data to reduce the need for developers to construct specialized systems. STEP uses a definition language designed specifically to reuse records and feed definition objects to its adaptive encoding process, which employs several strategies to increase the compressibility of the traces. The traces are then compressed using a general-purpose compressor. STEP is targeted towards application and compiler developers and focuses on Java programs running on JVMs.

The Locality-Based Trace Compression algorithm [32] was designed to exploit the spatial and temporal locality in (physical) address traces. It is similar to PDATS except it stores all attributes that accompany each memory reference (e.g., the instruction word or the virtual address) in a cache that is indexed by the address. When the attributes in the selected cache entry match, a hit bit is recorded in the compressed trace. In case of a cache miss, the attributes are written to the compressed trace. This caching captures the temporal locality of the memory references.

Several papers point out or exploit the close relationship between prediction and compression. For example, Chen et al. [9] argue that a two-level branch predictor is an approximation of the optimal PPM predictor and hypothesize that data compression provides an upper limit on the performance of correlated branch prediction. While we use prediction techniques to improve compression, Federovsky et al. [12] use compression methods to improve (branch) prediction.

2.2 Compression Algorithms

This subsection describes the compression schemes with which we compare our approach in Section 6. 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 to include efficient block I/O operations and to include a second compression stage to improve the compression rate. They are all single-pass, lossless compression schemes that 'know' about the PC and extended data fields in our traces. However, none of these algorithms separate the extended data into mini-streams like VPC does.

BZIP2 [18] 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 [3]. 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 the '--best' option throughout in this paper. According to *ps*, BZIP2 requires about 10MB of memory to compress and decompress our traces. We evaluate BZIP2 version 1.0.2 as a standalone compressor and as the second stage compressor for all the other algorithms.

MACHE [41] was primarily 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. After reading in a label and address pair, MACHE compares the address with the current base. There is one base for each possible label. If the difference between the address and the base can be expressed in a single byte, the difference is emitted directly. If the difference is too large, the full address is emitted, and this address becomes the new base for the current label. This algorithm repeats until the entire trace has been processed.

Our trace format is different from the standard MACHE format. It consists of pairs of 32-bit PC and 64-bit data entries. Since PC and data entries alternate, no labels are necessary to identify their type. MACHE only updates the base when the full address needs to be emitted. We retain this policy for the PC entries. 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.

SEQUITUR [29], [35], [36], [37] is one of the most sophisticated trace compression algo-

rithms in the literature. It converts a trace into a context-free grammar and 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. SEQUITUR has the interesting feature that information about the trace can be derived from the compressed format without having to decompress it first. 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 extended traces.

The SEQUITUR algorithm we use is a modified version of Nevill-Manning and Witten's implementation [17], which we changed as follows. We manually 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 employ a split-stream approach, that is, we construct two separate grammars, one for the PC entries and one for the data entries in our traces. To cap 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 and system. Our implementation's memory usage never exceeds 951MB, thus fitting into the 1GB of main memory in our system. 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.

PDATS II [23] improves upon PDATS by exploiting common patterns in program behavior. For example, jump-initiated sequences are often followed by sequential 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. To reduce the number of large offsets, 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 ac-

cesses, we do not need to distinguish between them in the header. This makes an extra bit available, which we use to expand the encoded data offsets to include ±16, ±32, and ±64. Because our traces contain offsets greater than 32 bits, we extended PDATS II to also accommodate six- and eight-byte offsets. Our traces do not exhibit many jump-initiated sequences that are followed by sequential sequences. Hence, we do not need the corresponding PDATS II feature. Our implementation uses 2.2MB of memory.

SBC (Stream-Based Compression) [33], [34] is one of the best trace compressors in the literature. It splits traces into segments called instruction streams. An instruction 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 in the original trace 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 VPC streams are unrelated to SBC streams as the former span the entire length of the trace.

We made the following changes to Milenkovics' SBC code [20]. Since our traces contain some but not all instructions (e.g., only instructions that access the memory), we redefined the notion of 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 experimented with different thresholds and found a threshold of four instructions to provide the best compression rate on our traces. SBC uses 10MB of memory to run.

3. BACKGROUND

3.1 Value Predictors

This subsection describes the operation of the value predictors that the various VPC algorithms use to predict trace entries. The exact predictor configurations are detailed in Sections 5.4 and 5.5. All of the presented value predictors extrapolate their forecasts based on previously seen values, that is, they predict the next trace entry based on already processed entries. Note that PC predictions are made based on global information while ED predictions are made based on local information, i.e., based on mini-streams whose entries all have the same PC.

Last *n* value predictor: The first type of predictor VPC uses is the last *n* value predictor (LnV) [7], [30], [47]. It predicts the most likely value among the *n* most recently seen values. To improve the compression rate, all *n* values are used (and not just the most likely value), i.e., the predictor can be thought of as comprising *n* components that make *n* independent predictions. The LnV predictor accurately predicts sequences of repeating and alternating values as well as repeating sequences of no more than *n* arbitrary values. Since PCs infrequently exhibit such behavior, the LnV predictor is only used for predicting data entries.

Stride 2-delta predictor: Stride predictors retain the most recently seen value along with the difference (stride) between the most recent and the second most recent values [13]. Adding this difference to the most recent value yields the prediction. Stride predictors can therefore predict sequences of the following pattern.

$$A, A+B, A+2B, A+3B, A+4B, \ldots$$

Every time a new value is seen, the difference and the most recent value in the predictor are updated. To improve the prediction accuracy, the 2-delta method (ST2D) has been proposed [43]. It introduces a second stride that is only updated if the same stride is encountered at least twice in a row, thus providing some hysteresis before the predictor switches to a new stride.

Finite context method predictor: The finite context method predictor (FCMn) [43] computes a hash out of the *n* most recently encountered values, where *n* is referred to as the order of the predictor. VPC utilizes the select-fold-shift-xor hash function [39], [40], [42]. During updates, the predictor puts the new value into its hash table using the current hash as an index. During predictions, 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 [42], [43]. Thus, FCMn predictors can memorize long arbitrary sequences of values and accurately predict them when they repeat. This trait makes FCMn predictors ideal for predicting PCs as well as EDs. Note that the hash table is shared among the mini-streams, making it possible to communicate information from one mini-stream to another.

Differential finite context method predictor: The differential finite context method predictor (DFCMn) [14] works just like the FCMn predictor except it predicts and is updated with differences (strides) between consecutive values rather than with absolute values. To form the final prediction, the predicted stride is added to the most recently seen value. DFCMn predictors are often superior to FCMn predictors because they warm up faster, can predict never before seen

values, and make better use of the hash table.

Global/local last value predictor: The global/local last value predictor (GLLV) [5] works like the last-value predictor (i.e., like an LnV predictor with $n = 1$), with the exception that each line contains two additional fields: an index and a counter. The index designates which entry of the last-value table holds the prediction. The counter assigns a confidence to this index. Every time a correct prediction is made, the counter is set to its maximum and every misprediction decrements it. If the counter is zero and the indexed value incorrect, the counter is reset to the maximum and the index is incremented (modulo the table size) so that a different entry will be checked the next time. This way, the predictor is able to correlate any mini-stream with any other mini-stream without the need for multiple comparisons per prediction or update.

3.2 Trace Format

The traces we generated for evaluating the different compression algorithms 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 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 format was chosen for its simplicity. We use 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.

4. THE VPC ALGORITHMS

4.1 Prototype

The development of VPC started with a prototype of a value-predictor-based compression algorithm that only compresses the extended data and works as follows. The PC of the current PC/ED pair is copied to the output and is used to index a set of value predictors to produce 27 (not necessarily distinct) predictions. The predicted values are then compared to the ED. If a match is found, the corresponding predictor identification code is written to the output using a fixed m-bit encoding. If no prediction is correct, an m-bit flag is written followed by the unpredictable 64-bit extended-data value. Then the predictors are updated with the true ED value. The

algorithm repeats until all PC/ED pairs in the trace have been processed. Decompression is essentially achieved by running the compression steps in reverse.

This prototype algorithm is ineffective because it does not compress the PCs and because *m* is too large. Since it uses 27 predictors plus the flag, five bits are needed, i.e., *m* = 5. Moreover, unpredictable ED entries are not compressed. The prototype's compression rate cannot exceed a factor of 2.6 because even assuming that every ED entry is predictable, a 96-bit PC/ED pair (32-bit PC plus 64-bit ED) is merely compressed down to 37 bits (a 32-bit PC plus a 5-bit code).

4.2 VPC1

VPC1 [5] corrects the shortcomings of the prototype. It includes a separate bank of ten predictors to compress the PCs, bringing the total number of predictors to 37. It uses a dynamic Huffman encoder [27], [46] to compress the identification codes. If more than one predictor is correct simultaneously, VPC1 selects the one with the shortest Huffman code. Moreover, it compresses the unpredictable trace entries in the following manner. In case of PCs, only *p* bits are written, where *p* is provided by the user and has to be large enough to express the largest PC in the trace. In case of unpredictable ED entries, the flag is followed by the identification code of the predictor whose prediction is closest to the ED value in terms of absolute difference. VPC1 then emits the difference between the predicted value and the actual value in compressed sign-magnitude format.

VPC1 includes several enhancements to boost the compression rate. First, it incorporates saturating up/down counters in the hash table of the FCMn predictors so that only entries that have proven useless at least twice in a row can be replaced. Second, it retains only distinct values in all of its predictors to maximize the number of different predictions and therefore the chance of at least one of them being correct. Third, it keeps the values in the predictors in least recently used order. Due to the principle of value locality, sorting the values from most recent to least recent ensures that components holding more recent values have a higher probability of providing a correct prediction. This increases the compression rate because it allows the dynamic Huffman encoder to assign shorter identification codes to the components holding more recent values and to use them more often. Fourth, VPC1 initializes the dynamic Huffman encoder with biased, nonzero frequencies for all predictors to allow the more sophisticated predictors, which perform poorly in the beginning because they take longer to warm up, to stay ahead of the simpler predic-

tors. Biasing the frequencies guarantees 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 only performs well on hard-to-compress traces [5]. The problem is that its compression rate is limited to a factor of 48. Since at least one bit is needed to encode a PC and one bit to encode an extended-data entry, and an uncompressed PC/ED pair requires 32+64=96 bits, the maximum compression rate is 96/2=48. We found VPC1 to almost reach this theoretical maximum in several cases, showing that the dynamic Huffman encoder indeed often uses only one bit to encode a predictor identification code. Because the PC and ED predictor codes alternate in the compressed traces and because at most one of the PC predictors and one of the ED predictors can have a one-bit identification code at any time, highly compressed VPC1 traces contain long bit strings of all zeros, all ones, or alternating zeros and ones, depending on the two predictors' identification codes. As a result, compressed VPC1 traces can be compressed further.

4.3 VPC2

VPC2 exploits the compressibility in VPC1's output by adding a second compression stage. In other words, VPC2 is VPC1 plus GZIP [21]. VPC2 outperforms VPC1 in all studied cases [5]. More importantly, VPC2 also outperforms other algorithms, including SEQUITUR, on easy-to-compress traces. Unfortunately, it is over 3.5 times slower at decompressing traces.

4.4 VPC3

Analyzing the performance of VPC2, we noticed that compressing traces as much as possible with the value predictors often obfuscates patterns and makes the resulting traces less amenable to the second stage compressor. In fact, we found that compressing less in the first stage can speed up the algorithm and boost the overall compression rate at the same time. Hence, we designed VPC3 to optimize this interaction between the two stages, i.e., instead of maximizing the performance of each stage individually we maximized the overall performance. Thus, VPC3's value predictors ended up only compressing the traces by a factor of between 1.4 to close to six (6.0 being the maximum possible) before they are sent to the second stage.

VPC3 differs from VPC2 in the following ways. We found that removing infrequently used predictors decreases the number of distinct predictor codes and increases the regularity of the

emitted identification codes, which results in better compression rates despite the concomitant increase in the number of emitted unpredictable values. As a result, VPC3 incorporates only 4 instead of 10 PC predictors and only 10 instead of 27 data predictors. Similarly, we discovered that abolishing the saturating counters and updating with non-distinct values decreases the prediction accuracy somewhat but greatly accelerates the algorithm. Moreover, emitting values at byte granularity and thus possibly including unnecessary bits increases the amount of data transferred from the first to the second stage but exposes more patterns and makes them easier to detect for the second stage, which also operates at byte granularity. Thus, VPC3 functions exclusively at byte granularity (and integer multiples thereof), which further simplified and accelerated the code. Finally, VPC3 does not use a frequency bias or a dynamic Huffman coder because the second stage is more effective at compressing the predictor identification codes. For the same reason, unpredictable values are no longer compressed in the first stage, which has the pleasant side effect of eliminating the need for the number of bits (p) required to express the PCs in the trace.

Note that we investigated many more ideas but ended up implementing only the ones listed above because they increase the overall compression rate and accelerate the algorithm. The following is a summary of some of our experiments 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 decreases the compression rate. (4) Larger predictor tables than the ones listed in Section 5.5 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, i.e., weighing recent behavior more, decreases the compression rate.

VPC3 converts PC/ED traces into four data streams in the first stage and then compresses each stream individually using BZIP2. Note that VPC3's first stage is so much faster than VPC2's that we were able to switch from the faster GZIP to the better compressing BZIP2 in the second stage. The four streams are generated as follows. The PC of the current trace entry is read and compared to the four PC predictions (Sections 3.1 and 5.5 describe the predictors). If none of

the predictions are correct, the special code of '4' (one byte) representing the flag 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 code ('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 selects 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 3.1 and 5.5). A one-byte predictor identification code ('0', '1', …, '9') is written to the third stream if at least one of the ED predictions is correct. If none are correct, the special code of '10' representing the flag 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 processed. Decompression reverses these steps. Note that VPC3 does not interleave predictor codes and unpredictable values in the same stream as VPC2 does. Rather, the codes and the unpredictable values are kept separate to improve the effectiveness of the second stage.

4.5 VPC4

VPC4 is the algorithm that is synthesized by TCgen [6] when TCgen is instructed to 'regenerate' VPC3. The two algorithms are quite similar, but VPC4 is faster and compresses better primarily due to the following two modifications. First, we improved the hash function of the FCMn and DFCMn predictors and accelerated its computation. Second, we enhanced the predictor's update policy. VPC3 always updates all predictor tables, which makes it very fast because the tables do not first have to be searched for a matching entry. VPC2, on the other hand, only updates the predictors with distinct values, which is much slower but increases the prediction accuracy. VPC4'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, 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.

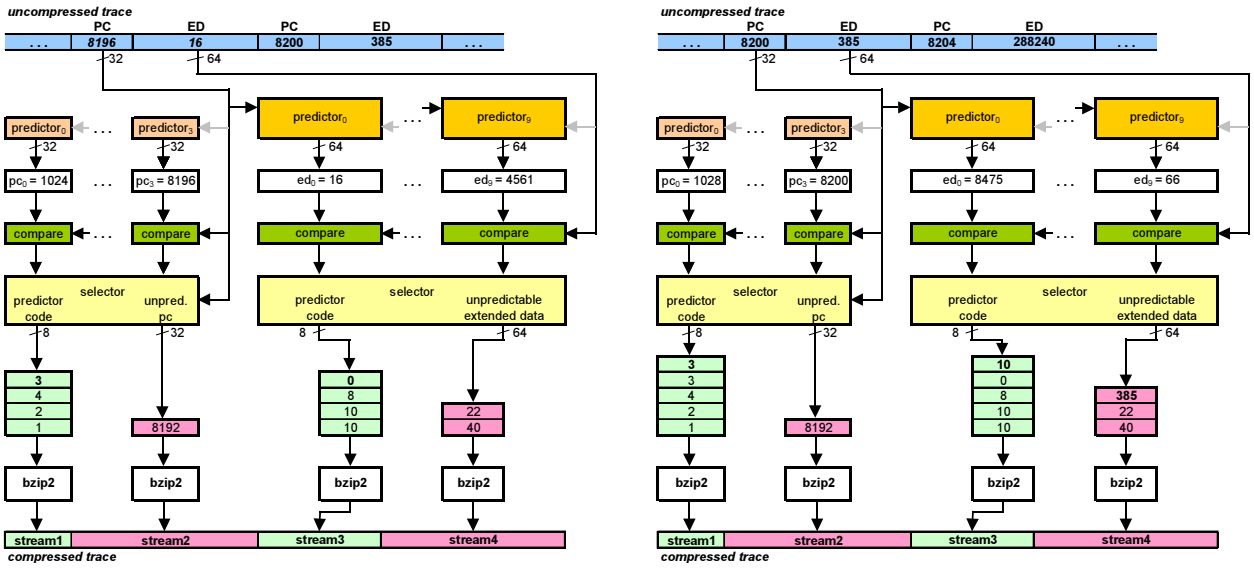

Figure 1: Example of the operation of the VPC4 compression algorithm.

Figure 1 illustrates VPC4's operation. The dark arrows mark the prediction paths while the light arrows mark the update paths. The left panel shows how the PC/ED pair 8196/16 is handled assuming that PC predictor 3 and ED predictor 0 make correct predictions. The right panel shows the compression of the next PC/ED pair in the trace. This time, none of the ED predictions are correct and the unpredictable ED value of 385 is written to the fourth stream.

5. EVALUATION METHODS

This section gives information about the system and compiler (Section 5.1), the timing measurements (Section 5.2), and the traces (Section 5.3) we used for our measurements and details the predictor configurations in VPC1/2 (Section 5.4) and VPC3/4 (Section 5.5).

5.1 System and Compiler

We performed all measurements for this study on a dedicated 64-bit CS20 system with two 833MHz 21264B Alpha CPUs [26]. 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 build-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. To make the running-time comparisons as fair as possible, we compiled all

compression algorithms with Compaq's C compiler V6.3-025 and the same optimization flags (-O3 -arch host).

5.2 Timing Measurements

All timing measurements in this paper refer to the sum of the user and the system time 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 coded 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 given the large sizes of our traces.

5.3 Traces

We used all integer and all but four of the floating-point programs from the SPECcpu2000 benchmark suite [22] 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 our traces. Only system-call code is not captured. We used the binary instrumentation tool-kit ATOM [11], [44] to generate traces from complete runs 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 to form a single trace.

We generated three types of traces from the 22 programs to evaluate the various 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 holds 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. 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: Uncompressed 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~~

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. We had to exclude all traces that are larger than twelve gigabytes because they would have exceeded the available disk space. The corresponding entries in Table 1 are crossed out. Note that we found no correlation between the length and the compressibility of our traces. Hence, we do not believe that omitting the longest traces distorts our results.

5.4 VPC1 and VPC2 Predictor Configurations

This subsection lists the sizes and parameters of the predictors used in VPC1/2. They are the result of experimentation with the load-value traces to balance the speed and the compression rate as well as the fact that we wanted to maximize the sharing of predictor tables.

VPC1/2 makes ten PC predictions using an ST2D, FCM6, DFCM4ab, DFCM5abcd, and a

DFCM6ab predictor, where ‘a’, ‘b’, ‘c’, and ‘d’ refer to the up-to-four most recent values in each line of the hash table. VPC1/2 further makes 27 ED predictions using an L6V, ST2D, GLLV, FCM1, FCM2, FCM3, FCM6, DFCM1abcd, DFCM2abcd, DFCM3ab, DFCM4, and a DFCM6abcd predictor.

Since no index is available for the PC predictors, all PC predictors are global predictors. Accordingly, their first levels are very small. The ST2D predictor requires only 24 bytes of storage: eight bytes for the last value plus sixteen bytes for the two stride fields. The FCM6 predictor requires six two-byte fields to store six hash values in the first level. The second level requires 4.5MB to hold 524,288 lines of nine bytes each (eight bytes for the value and one byte for the saturating counter). All DFCMn predictors use a shared first-level table, which requires six two-byte fields for retaining hash values. The most recent value is obtained from the ST2D predictor. The DFCM4ab has two second-level tables of one megabyte each (131,072 lines), the DFCM5abcd has four tables of two megabytes each (262,144 lines), and the DFCM6ab has two tables of four megabytes each (524,288 lines). Since we do not use saturating counters in the DFCMn predictors, each second-level entry requires eight bytes to hold a 64-bit value. Overall, 22.5 megabytes are allocated for the ten PC predictors.

The predictors for the extended data are local and use the PC modulo the table size as an index into the first-level table, which allows them to store information on a mini-stream basis. The L6V predictor uses six tables of 128kB (16,384 lines). The ST2D predictor requires 256kB of table space for the strides (16,384 lines). The last-value table is shared with the L6V predictor. The FCMn predictors share six first-level, 32kB tables, each of which holds 16,384 two-byte hash values. Similarly, the DFCMn predictors share a set of six 32kB first-level tables. The second-level table sizes are as follows. The FCM1 uses a 144kB table (16,384 lines holding an eight-byte value plus a one-byte counter), the FCM2 uses a 288kB table (32,768 lines), the FCM3 has a 576kB table (65,536 lines), and the FCM6 requires 4.5MB (524,288 lines). The DFCM1abcd has four tables of 128kB (16,384 lines holding an eight-byte value), the DFCM2abcd uses four 256kB tables (32,768 lines), the DFCM3ab requires two 512kB tables (65,536 lines), the DFCM4 has one 1MB table (131,072 lines), and the DFCM6abcd needs four 4MB tables (524,288 lines). Finally, the GLLV predictor uses 128kB of storage (16,384 lines containing a four-byte index and a four-byte counter). Together, the 27 value predictors use 26.5MB of table space.

Overall, 49 megabytes are allocated for predictor tables. Including padding, code, libraries, etc., VPC1/2 requires 88MB of memory to run as reported by the UNIX command *ps*.

5.5 VPC3 and VPC4 Predictor Configurations

This subsection lists the parameters and table sizes of the predictors used in VPC3/4. These configurations have been experimentally determined to yield a good compression rate and speed on the gcc load-value trace.

VPC3 uses an FCM1ab and an FCM3ab for predicting PCs, which provide a total of four predictions. For the extended data, it uses an FCM1ab, DFCM1ab, DFCM3ab, and a L4V predictor, providing a total of ten predictions.

The four PC predictors are global and do not need an 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. To enable maximum sharing, all first-level tables have the same number of lines (65,536). The L4V 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 L4V 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 for predictor tables in VPC3. Including the code, stack, etc., it requires 27MB of memory to run according to *ps*. VPC4 uses the same predictors and the same tables sizes as VPC3 with one exception. Due to a restriction of TCgen, the second-level tables of the FCM1ab and DFCM1ab have to have the same number of lines. Hence, VPC4's FCM1ab predictor for the extended data has a hash table with only 131,072 lines, resulting in a size of 2MB. Overall, VPC4 allocates 20MB of table space and has a 21MB memory footprint.

6. RESULTS

Section 6.1 discusses the performance of the four VPC algorithms. Section 6.2 compares the compression rate, Section 6.3 the decompression speed, and Section 6.4 the compression speed of BZIP2, MACHE, PDATS II, SBC, SEQUITUR, and VPC4. Note that we added a second compression stage using BZIP2 to each of these algorithms except BZIP2 itself. Section 6.5 investigates GZIP as an alternative second-stage compressor.

6.1 VPC Comparison

Table 2 lists the geometric-mean compression rate (c.rate), decompression time (d.time), and compression time (c.time) of the four VPC algorithms on our three types of traces. The results for VPC2, VPC3, and VPC4 include the second stage. The compression and decompression times are in seconds. Higher numbers are better for the compression rates while lower numbers are better for the compression and decompression times.

Table 2: Geometric-mean performance of the VPC algorithms.

	store addresses			cache misses			load values		
	c.rate	d.time	c.time	c.rate	d.time	c.time	c.rate	d.time	c.time
VPC1	36.0	370.4	469.9	14.7	177.9	213.3	17.0	803.1	1,029.5
VPC2	237.2	372.0	544.4	26.3	180.0	239.1	27.0	811.9	1,172.6
VPC3	484.7	74.8	252.9	38.2	45.4	128.6	35.2	203.1	584.9
VPC4	808.7	69.1	216.3	43.5	41.9	118.3	42.6	194.6	503.6

For the reasons explained in Section 4.2, VPC1 compresses relatively poorly and is slow. Adding a GZIP compression stage (VPC2) significantly increases the compression rate, especially on the easy-to-compress store address traces. At the same time, the extra stage does not increase the compression and decompression times much. Nevertheless, VPC2 is the slowest of the VPC algorithms.

The changes we made to VPC3 (see Section 4.4) eliminate this weakness. It decompresses four to five times faster than VPC2 and compresses about twice as fast while at the same time boosting the compression rate by 30% to 100%. Clearly, it is more fruitful to use the value predictors to transform the traces rather than to compress them. Finally, the enhanced hash function and update policy of VPC4 further increase the compression rate and decrease the compression and decompression times, making VPC4 the fastest and most powerful of the VPC algorithms.

6.2 Compression Rate

Figure 2 depicts the geometric-mean compression rates of six algorithms from the literature on our three types of traces normalized to VPC4 (higher numbers are better). For each trace type, the algorithms are sorted from left to right by increasing compression rate. The darker section at the bottom of each bar indicates the compression rate after the first stage. The total bar height represents the overall compression rate including the second stage.

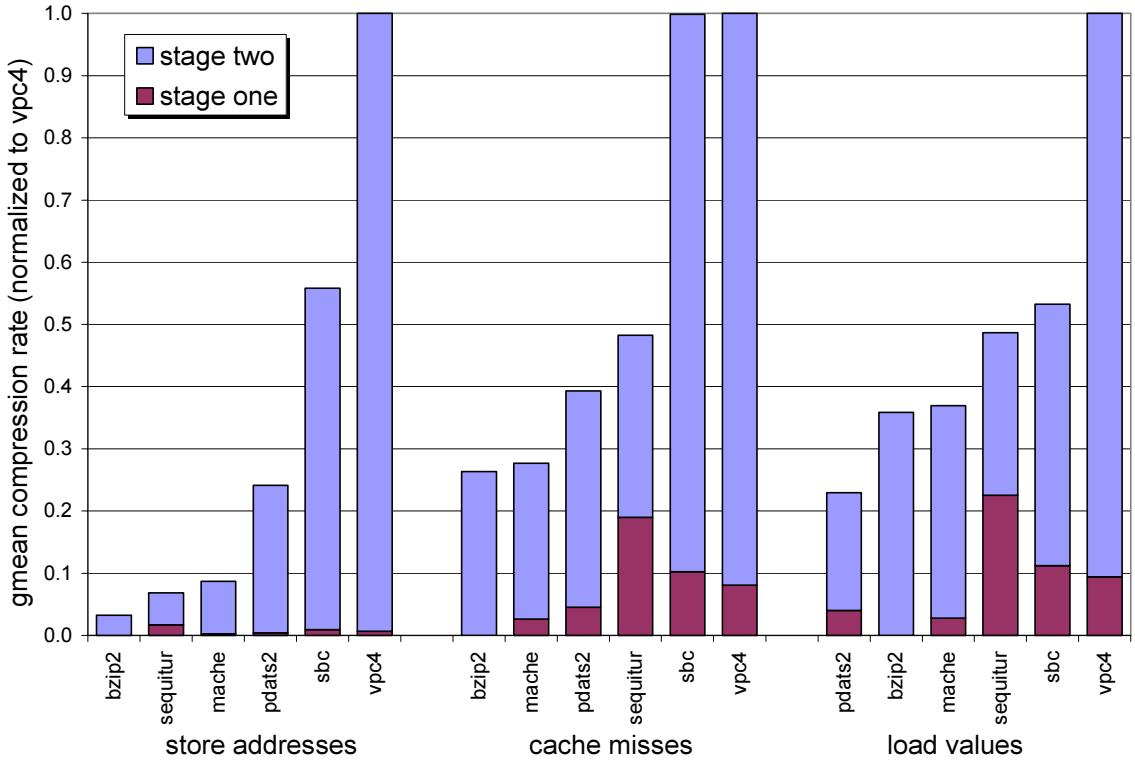

Figure 2: Geometric-mean compression rates normalized to VPC4.

VPC4 delivers the best geometric-mean compression rate on each of the three trace types. SBC, the second best performer, comes within 0.2% of VPC4 on the cache-miss traces but only reaches slightly over half of VPC4's compression rate on the other two types of traces. It appears that the cache-miss traces contain patterns that are equally exploitable with SBC and VPC4 while the other two types of traces include important patterns that only VPC4 can take advantage of. This makes sense as store addresses often correlate with the previous addresses produced by the same store. Similarly, load values correlate with the previously fetched values of the same load instruction. Since the mini-streams expose these correlations, VPC4 can exploit them. Filtering

out some of the values, as is done in the cache-miss traces, breaks many of these correlations and reduces VPC4's advantage.

SEQUITUR is the next best performer except on the store-address traces, where PDATS II and MACHE outperform it. This is probably because these two algorithms were designed for address traces. SEQUITUR, on the other hand, does not work well on long strided sequences, which are common in such traces. MACHE and BZIP2's compression rates are in the lower half but they both outperform PDATS II on the load-value traces.

Looking at the geometric-mean compression rate of the first stage only, we see that SEQUITUR is dominant, followed by SBC. Nevertheless, high initial compression rates do not guarantee good overall performance. In fact, with the exception of MACHE on the load-value traces (and BZIP2, which has no first stage), all algorithms have a higher ratio of first-stage to overall compression rate than VPC4. VPC4's first stage provides less than ten percent of the compression rate on all three types of traces (only 0.7% on the store-address traces). Note that the second stage (BZIP2) boosts the compression rate by more than a factor of two in all cases.

Table 3 shows the absolute compression rate achieved by the six algorithms on each individual trace. Only the total compression rates over both stages are listed. We highlighted the highest compression rate for each trace in bold print. The table further includes the harmonic, geometric, and the arithmetic mean of the compression rates for each trace type.

We see that no algorithm is the best for all traces. However, VPC4 yields the highest compression rate on all load-value traces. This is most likely because many of the value predictors VPC4 uses were specifically designed to predict load values. Moreover, VPC4's predictor configuration was tuned using one of the load-value traces (gcc).

On the cache-miss traces, SBC outperforms all other algorithms on six traces, SEQUITUR on two traces, and VPC4 on ten traces. BZIP2 provides the best compression rate on four traces even though it is the only compressor that has no knowledge of our trace format and all the other algorithms use BZIP2 in their second stage. On sixtrack, SEQUITUR outperforms VPC4 by a factor of 5.5, which is the largest such factor we found. The cache-miss trace bzip2 is the least compressible trace in our suite. No algorithm compresses it by more than a factor of 5.8.

Table 3: Absolute compression rates.

		bzip2	sequitur	vpc4	mache	sbc	pdats2
store effective address traces	ammp	33.1	38.8	**1,115.5**	41.1	476.8	60.4
	applu	14.0	500.8	**3,249.3**	18.2	2,270.8	652.0
	apsi	39.0	226.0	**21,721.6**	66.2	4,886.2	603.0
	art	22.9	52.0	**77,160.9**	342.1	7,936.4	167.5
	crafty	37.1	102.7	82.9	86.4	89.3	**118.1**
	eon	24.3	**793.4**	343.0	151.3	275.4	405.3
	equake	50.5	48.7	**679.5**	118.4	291.3	167.1
	gap	17.8	15.5	88.1	46.8	**103.0**	38.1
	gcc	22.6	42.2	71.6	63.3	**74.6**	61.8
	gzip	20.5	19.6	**95.9**	26.6	60.2	16.7
	mcf	22.7	14.2	**132.2**	111.6	84.3	129.4
	mesa	53.5	134.0	**5,997.8**	121.0	3,832.3	304.0
	mgrid	14.2	13.3	**49,750.9**	155.6	19,684.6	6,494.5
	parser	14.2	18.4	66.2	37.4	**81.3**	25.6
	perlbmk	25.5	47.9	**134.3**	62.6	93.1	52.4
	swim	11.6	9.6	**46,518.3**	11.7	28,536.5	7,544.6
	twolf	34.5	**126.9**	47.3	52.4	41.3	60.5
	vpr	21.1	**43.9**	35.8	28.2	23.5	25.6
	wupwise	118.8	101.0	**9,015.7**	697.0	1,472.1	6,613.1
	har_mean	23.1	31.9	**142.9**	46.7	116.2	70.3
	geo_mean	26.3	55.5	**808.7**	70.2	451.4	195.2
	arith_mean	31.5	123.6	**11,384.6**	117.8	3,700.7	1,238.9
cache miss address traces	ammp	18.1	42.0	**71.0**	24.2	46.5	37.8
	applu	5.9	**176.5**	76.5	6.6	56.9	10.9
	apsi	9.9	285.8	**931.5**	9.4	660.0	73.4
	art	9.1	218.6	**6,936.4**	14.1	2,079.5	35.0
	bzip2	5.1	4.0	**5.8**	5.0	5.2	4.2
	crafty	**19.2**	14.2	16.0	16.3	16.9	12.0
	eon	**30.0**	22.3	18.7	27.4	24.7	20.7
	equake	9.4	24.3	**30.0**	9.6	24.8	11.5
	gap	10.6	6.5	11.7	10.7	**12.8**	7.6
	gcc	7.2	6.3	**9.3**	7.4	8.0	5.4
	gzip	7.3	7.9	**8.9**	7.0	8.1	6.5
	mcf	9.8	12.5	**18.7**	10.6	14.6	10.2
	mesa	67.5	78.3	323.1	88.3	**432.1**	133.4
	mgrid	7.8	6.9	273.1	7.8	**495.0**	83.0
	parser	6.3	7.3	**12.1**	9.3	11.1	8.2
	perlbmk	17.3	13.2	18.0	16.5	**19.7**	11.9
	sixtrack	11.4	**175.0**	31.7	9.5	29.8	9.1
	swim	8.4	21.0	319.1	8.4	**715.3**	144.5
	twolf	**15.9**	11.7	10.9	13.0	11.8	10.5
	vortex	20.1	22.4	**22.7**	19.5	20.5	13.8
	vpr	**7.6**	6.9	6.8	6.0	6.2	4.7
	wupwise	10.2	6.2	102.6	16.8	**302.1**	39.0
	har_mean	10.0	12.0	**18.6**	10.4	17.6	11.4
	geo_mean	11.5	21.0	**43.5**	12.0	43.4	17.1
	arith_mean	14.3	53.2	**420.7**	15.6	227.4	31.5
load value traces	applu	3.4	4.3	**7.2**	3.6	3.8	2.2
	art	30.7	55.2	**110.4**	30.9	38.2	15.5
	eon	10.8	14.5	**27.1**	11.6	10.5	5.0
	equake	11.0	13.2	**27.9**	11.3	14.2	6.7
	gap	21.7	26.9	**56.3**	23.4	34.4	14.5
	gcc	15.7	22.6	**32.9**	16.8	20.3	10.7
	gzip	13.8	12.1	**20.6**	13.0	14.1	8.9
	mcf	13.7	18.4	**22.8**	13.0	14.1	9.3
	mesa	117.9	293.0	**9,082.8**	131.5	2,428.4	92.3
	parser	18.8	21.2	**40.7**	20.1	28.7	12.6
	perlbmk	24.7	42.1	**66.9**	28.0	39.9	19.6
	swim	3.3	4.0	**7.8**	3.3	4.1	2.2
	twolf	14.8	21.9	**22.3**	16.0	16.3	10.1
	vpr	18.0	19.2	**29.2**	15.7	17.3	10.5
	har_mean	10.8	13.6	**23.0**	11.0	12.7	6.8
	geo_mean	15.3	20.7	**42.6**	15.8	22.7	9.8
	arith_mean	22.7	40.6	**682.5**	24.1	191.7	15.7

On the store-address traces, which are far more compressible than the other two types of traces, VPC4 yields the highest compression rate on twelve traces, SEQUITUR and SBC on three traces each, and PDATS II on one trace. VPC4 surpasses the other algorithms by a factor of 9.7 on art, which it compresses by a factor 77,161, the highest compression rate we observed.

Overall, VPC4 provides the best compression rate on two thirds of our traces and delivers the highest arithmetic-, geometric-, and harmonic-mean compression rates on the three types of traces, though SBC comes close on the cache-miss traces.

6.3 Decompression Time

Figure 3 shows the geometric-mean decompression time of the six algorithms on our three types of traces normalized to VPC4 (lower numbers are better). For each trace type, the algorithms are sorted from left to right by decreasing decompression time. The darker section at the bottom of each bar represents the time spent in the first stage. The lighter top portion of each bar depicts the time spent in the second stage. The total bar height corresponds to the overall decompression time. Note that during decompression, the second stage is invoked first.

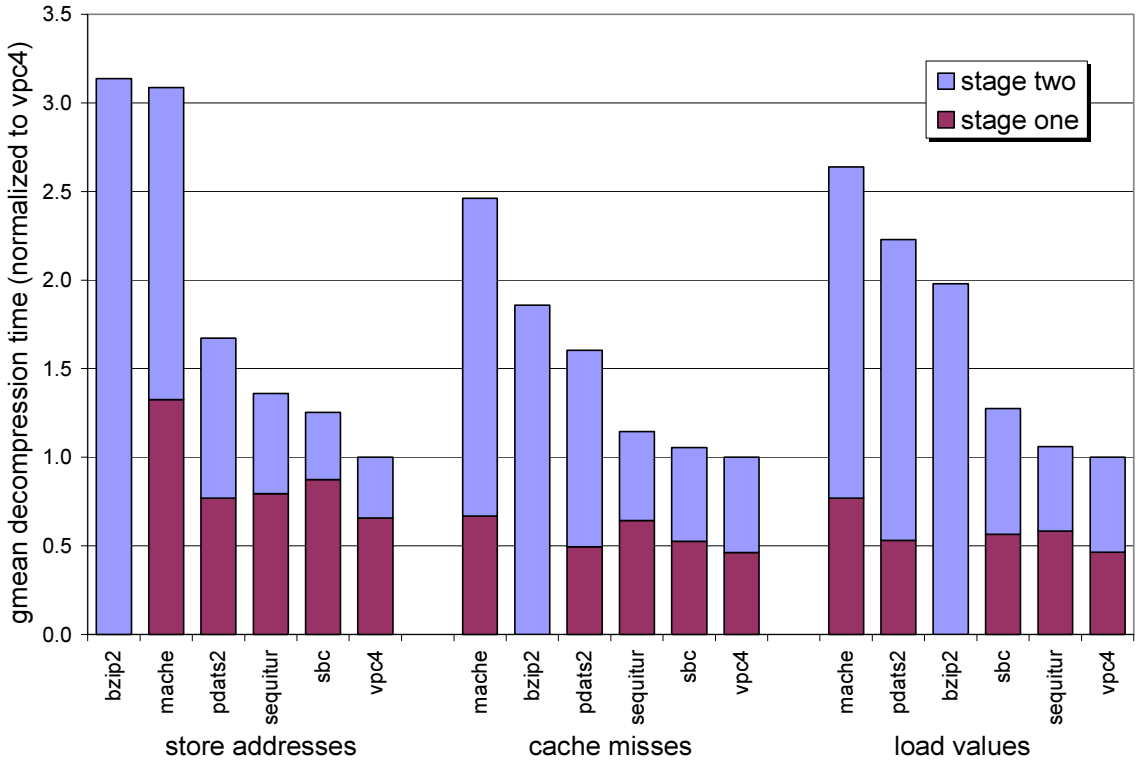

Figure 3: Geometric-mean decompression time normalized to VPC4.

Table 4: Absolute decompression time in seconds.

		bzip2	sequitur	vpc4	mache	sbc	pdats2
store effective address traces	ammp	569.7	230.8	**153.5**	620.2	214.5	368.3
	applu	64.6	**10.8**	15.8	76.9	22.8	32.9
	apsi	803.3	**216.6**	235.9	934.9	303.1	475.4
	art	176.1	84.1	**60.4**	134.0	73.0	84.3
	crafty	368.1	**126.4**	133.5	384.1	168.5	182.9
	eon	232.5	**42.9**	75.5	208.5	114.5	108.7
	equake	138.7	44.7	**38.9**	137.8	55.7	73.5
	gap	160.9	123.7	**56.7**	151.9	66.4	89.0
	gcc	257.1	**95.6**	99.8	237.0	121.1	123.6
	gzip	353.4	184.0	**118.2**	433.8	130.9	255.6
	mcf	55.3	75.2	**13.9**	32.5	15.0	19.5
	mesa	425.3	**98.3**	104.0	457.4	135.3	227.5
	mgrid	740.0	872.4	**183.7**	473.8	235.4	288.9
	parser	512.7	296.0	**182.1**	463.3	211.6	268.1
	perlbmk	67.4	27.3	**24.4**	72.8	29.1	41.0
	swim	63.3	86.6	**12.4**	80.9	16.9	37.9
	twolf	26.4	**7.5**	14.1	31.6	15.6	17.9
	vpr	237.1	**95.5**	125.5	275.2	146.5	162.8
	wupwise	1,116.2	387.2	**379.8**	1,022.1	469.8	490.5
	har_mean	127.7	46.0	**40.5**	126.2	50.2	68.8
	geo_mean	216.6	93.9	**69.1**	213.1	86.6	115.4
	arith_mean	335.2	163.4	**106.7**	327.8	134.0	176.2
cache miss address traces	ammp	424.5	187.0	**144.5**	487.8	189.9	276.1
	applu	11.5	**2.2**	4.5	14.1	5.4	8.7
	apsi	358.4	**92.4**	105.5	515.8	124.3	245.2
	art	297.3	**69.3**	93.4	325.6	108.5	184.5
	bzip2	121.0	150.6	126.6	157.3	**104.3**	125.7
	crafty	252.4	217.9	181.5	378.2	**173.0**	257.7
	eon	11.4	**7.1**	7.9	16.3	7.2	10.4
	equake	58.7	**24.7**	25.0	79.1	28.7	47.2
	gap	35.4	38.8	27.1	47.0	**23.9**	36.4
	gcc	54.8	57.5	**46.2**	73.6	49.2	59.5
	gzip	110.8	94.4	101.0	150.4	**81.9**	112.3
	mcf	19.9	19.3	**12.7**	23.2	15.2	16.6
	mesa	24.6	8.7	**6.7**	37.7	12.2	23.0
	mgrid	962.6	961.3	**297.3**	1,290.9	354.2	672.9
	parser	118.2	113.7	84.4	130.9	**81.7**	98.5
	perlbmk	11.6	8.3	8.0	16.3	**7.3**	11.6
	sixtrack	310.7	**74.7**	168.5	448.3	177.4	325.9
	swim	24.2	10.2	**4.4**	31.6	6.6	13.0
	twolf	9.7	8.1	9.5	13.7	**7.6**	9.9
	vortex	281.2	**173.0**	185.4	423.7	197.6	368.4
	vpr	100.2	102.9	119.3	141.5	**89.2**	112.8
	wupwise	133.7	173.5	**48.7**	153.1	49.6	72.8
	har_mean	35.3	**17.1**	17.8	47.3	20.1	30.6
	geo_mean	77.9	47.9	**41.9**	103.1	44.1	67.2
	arith_mean	169.7	118.0	**82.2**	225.3	86.1	140.4
load value traces	applu	202.4	164.8	**121.6**	243.9	196.4	290.6
	art	1,266.4	**394.5**	439.9	1,764.4	619.1	1,404.2
	eon	284.5	161.2	**143.9**	385.7	230.8	398.2
	equake	585.2	331.7	**251.0**	785.3	358.7	711.3
	gap	360.8	184.1	**172.9**	512.3	215.8	383.4
	gcc	534.7	307.8	**297.0**	715.7	357.8	569.8
	gzip	1,066.7	745.9	665.6	1,368.7	**633.4**	1,128.9
	mcf	55.5	**33.9**	36.4	73.0	37.7	58.9
	mesa	641.6	**131.6**	206.3	926.0	283.6	644.3
	parser	1,083.7	638.8	**548.9**	1,434.1	608.1	1,139.8
	perlbmk	129.0	**53.2**	57.3	176.3	70.8	126.9
	swim	434.5	371.8	**222.0**	529.7	391.9	567.7
	twolf	99.6	**57.9**	64.4	131.7	67.9	104.8
	vpr	843.4	528.8	**500.2**	1,133.7	552.4	888.2
	har_mean	240.1	**130.6**	132.5	317.4	159.6	264.0
	geo_mean	385.4	206.3	**194.6**	513.4	248.0	433.5
	arith_mean	542.0	293.3	**266.3**	727.2	330.3	601.2

VPC4 provides the fastest mean decompression time on our traces but SBC and SEQUITUR are close seconds. They are only 5% to 36% slower. The remaining three algorithms are at least 60% slower. SEQUITUR is faster on the load-value traces than SBC. The situation is reversed on the other two trace types. The majority of the algorithms that use BZIP2 in their second stage are faster than BZIP2 alone.

Looking at the two stages separately, we find that the three fastest decompressors, VPC4, SBC, and SEQUITUR, spend almost equal amounts of time in the two stages, except on the store-address traces, where the second stage (BZIP2) accounts for about a third of the total decompression time. MACHE and PDATS II, on the other hand, typically spend more time in the second stage than in the first.

Table 4 gives the absolute decompression time in seconds of the six algorithms on each individual trace. Only the total decompression time over both stages is listed. We highlighted the shortest decompression time for each trace in bold print. The table further includes the harmonic, geometric, and the arithmetic mean of the decompression times for each trace type.

SEQUITUR, VPC4, and SBC deliver the fastest decompression times on all traces. On the store-address traces, SEQUITUR is fastest on eight and VPC4 on eleven traces. On the cache-miss traces, SBC is fastest on eight and VPC4 and SEQUITUR on seven traces each. On average (any of the three means), VPC4 outperforms SBC, but SEQUITUR has the best overall harmonic-mean decompression time. On the load-value traces, SBC is fastest on one, SEQUITUR on five, and VPC4 on eight traces. Again, SEQUITUR outperforms VPC4 in the harmonic mean.

VPC4 is maximally 68% faster than the other algorithms on the load-value trace swim. SEQUITUR is 125% faster than VPC4 on the cache-miss trace sixtrack. In other words, the range from one extreme to the other is relatively narrow, meaning that VPC4's decompression time and that of the fastest other algorithm are fairly close on all traces.

6.4 Compression Time

Figure 4 shows the geometric-mean compression time of the six algorithms on our three types of traces normalized to VPC4 (lower numbers are better). Again, the algorithms are sorted from left to right by decreasing time. The darker section at the bottom of each bar represents the time spent in the first stage and the lighter top portion depicts the time spent in the second stage. The total bar height corresponds to the overall compression time. On the cache-miss traces, SBC's

first stage takes 13.9 and the second stage 2.3 times as long as VPC4's total compression time.

VPC4's geometric-mean compression time is over three times shorter than that of the other algorithms we evaluated. This is particularly surprising as VPC4 also achieves the highest mean compression rates (Section 6.2). Typically, one would expect longer computation times to result in higher compression rates but not the other way around. SBC, the otherwise best non-VPC algorithm, takes 9.5 to 16.2 times as long as VPC4 to compress the traces. The SBC authors verified that the current version of their algorithm has not been optimized to provide fast compression. SEQUITUR is 3.6 to 7.7 times slower than VPC4.

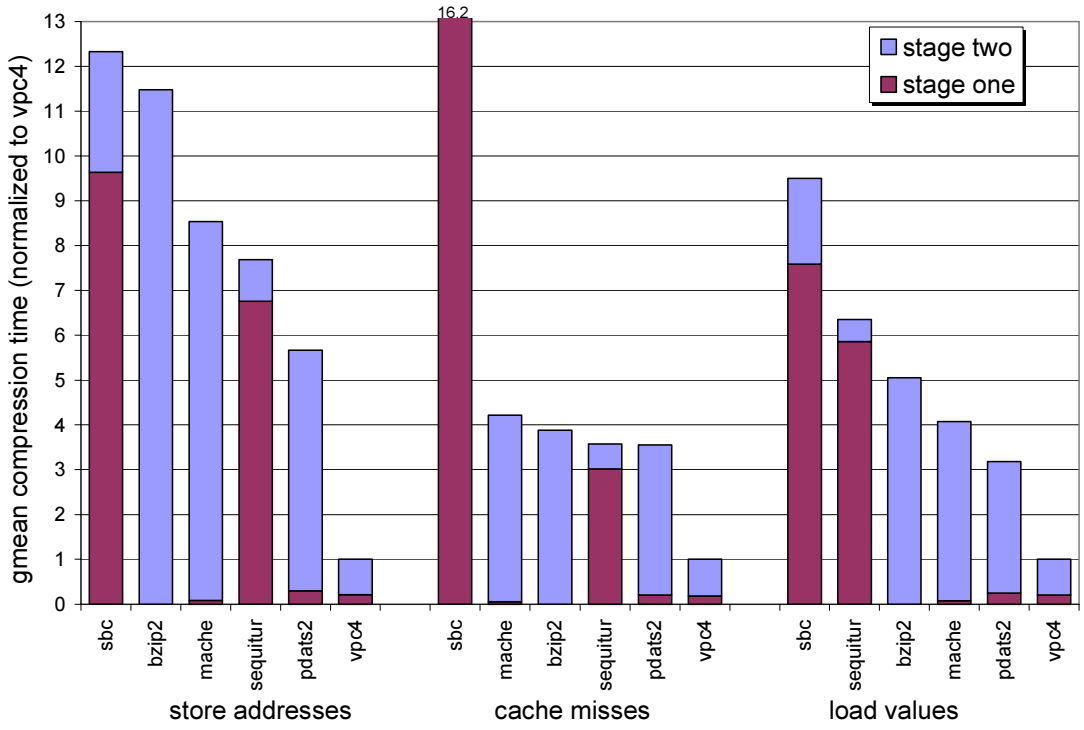

Figure 4: Geometric-mean compression time normalized to VPC4.

SEQUITUR's algorithm is quite asymmetric, i.e., constructing the grammar (compression) is a much slower process than recursively traversing it (decompression). The average (geometric mean for the three trace types) compression over decompression time ratio for SEQUITUR is 8.8 to 17.7. For BZIP2 it is 5.9 to 11.5, for MACHE 4.0 to 8.7, for PDATS II 3.7 to 10.6, and for SBC 19.3 to 43.4. VPC4 is the most symmetric of the six algorithms, which explains its fast compression speed. It basically performs the same operations during compression and decompression. The only asymmetry in VPC4 is that during compression it needs to check each predictor for a match, which makes compression 2.6 to 3.1 times slower than decompression.

Table 5: Absolute compression time in seconds.

		bzip2	sequitur	vpc4	mache	sbc	pdats2
store effective address traces	ammp	13,912.0	5,081.6	**331.5**	5,938.0	6,216.3	5,090.7
	applu	653.2	619.9	**91.9**	399.1	699.5	610.1
	apsi	18,574.5	8,793.8	**1,055.2**	16,902.8	9,911.1	7,177.4
	art	4,772.9	1,310.3	**78.5**	1,929.7	2,689.8	1,327.6
	crafty	3,814.1	2,683.3	**232.4**	2,289.2	5,096.5	973.8
	eon	2,167.3	2,643.4	**222.4**	1,549.2	4,330.6	1,073.0
	equake	3,317.4	1,202.3	**254.0**	1,947.3	1,237.4	944.9
	gap	1,353.6	1,101.3	**155.9**	1,036.5	2,003.4	558.9
	gcc	3,440.7	1,873.5	**196.2**	1,846.4	6,090.1	988.9
	gzip	3,842.1	2,694.6	**552.6**	3,628.9	3,319.3	2,039.8
	mcf	256.4	433.1	**33.2**	508.6	450.3	350.8
	mesa	8,271.8	4,051.8	**685.2**	7,572.2	4,010.2	3,223.7
	mgrid	2,269.5	3,948.7	**371.9**	5,427.5	7,548.9	4,520.6
	parser	3,851.8	3,416.1	**449.5**	4,201.0	7,278.8	2,204.1
	perlbmk	649.6	405.4	**98.7**	500.5	1,515.6	266.1
	swim	277.2	466.9	**27.7**	198.8	398.3	980.7
	twolf	293.7	129.4	**35.8**	199.4	386.8	101.3
	vpr	2,056.0	1,667.4	**286.3**	1,317.2	2,677.1	525.1
	wupwise	39,910.5	9,361.3	**3,903.6**	13,544.0	15,558.2	9,365.3
	har_mean	1,057.6	854.9	**113.1**	862.6	1,439.8	643.4
	geo_mean	2,482.4	1,662.1	**216.3**	1,846.7	2,666.4	1,226.3
	arith_mean	5,983.4	2,730.7	**477.0**	3,733.5	4,285.2	2,227.5
cache miss address traces	ammp	11,007.6	2,772.2	**276.8**	4,396.2	6,396.8	3,162.7
	applu	42.0	48.4	**17.9**	33.0	212.2	27.0
	apsi	1,564.4	2,473.2	**485.7**	1,454.0	5,701.6	4,074.5
	art	1,007.4	1,555.6	**438.4**	3,519.5	4,093.7	3,324.4
	bzip2	403.4	595.5	**274.6**	420.1	1,858.2	293.9
	crafty	1,709.2	1,940.8	**318.2**	1,529.4	13,039.8	779.3
	eon	100.3	50.2	**14.9**	90.1	214.6	41.7
	equake	573.4	306.1	**83.4**	614.7	735.8	261.8
	gap	208.8	190.0	**56.7**	235.0	1,597.0	133.6
	gcc	239.5	331.7	**87.9**	256.3	15,800.6	187.4
	gzip	883.1	610.9	**211.8**	934.1	1,587.4	818.3
	mcf	85.2	133.4	**36.3**	237.6	255.5	134.1
	mesa	757.4	85.0	**13.5**	832.2	415.9	461.0
	mgrid	3,110.6	5,939.6	3,322.0	**2,929.0**	15,929.3	11,078.1
	parser	442.9	767.9	**163.1**	1,290.9	10,231.1	782.6
	perlbmk	92.6	50.1	**17.2**	96.0	886.8	64.8
	sixtrack	2,407.1	2,140.6	**1,041.0**	1,899.0	5,175.6	1,169.1
	swim	65.6	90.2	**11.5**	61.8	242.2	276.9
	twolf	52.8	44.5	**22.1**	47.9	193.7	23.8
	vortex	2,504.4	1,996.4	**364.4**	2,573.4	30,486.9	1,291.3
	vpr	343.4	586.3	277.3	335.6	1,487.0	**214.5**
	wupwise	475.8	648.9	**270.9**	442.0	2,013.0	1,182.1
	har_mean	188.3	161.9	**44.6**	188.7	697.6	134.5
	geo_mean	458.7	422.8	**118.3**	499.1	1,917.1	420.2
	arith_mean	1,276.2	1,061.7	**354.8**	1,101.3	5,388.9	1,353.8
load value traces	applu	556.1	1,281.2	**303.5**	458.6	1,527.7	508.0
	art	20,394.5	10,568.4	**777.9**	16,672.9	7,608.3	12,069.2
	eon	1,441.2	3,113.0	**307.2**	1,187.4	3,737.6	942.4
	equake	4,295.4	6,262.6	**912.9**	3,659.7	4,951.1	3,173.4
	gap	2,668.1	4,037.5	**422.1**	2,227.9	6,673.7	1,560.9
	gcc	3,264.2	4,260.3	**559.2**	2,344.2	28,355.6	1,986.2
	gzip	5,727.6	6,578.8	**2,148.2**	4,174.4	10,264.8	3,172.2
	mcf	260.7	352.7	**98.7**	210.3	634.8	171.2
	mesa	17,768.9	7,338.4	**1,861.4**	15,179.0	8,561.7	10,144.3
	parser	9,974.2	8,260.2	**1,034.6**	6,685.5	22,074.1	5,616.9
	perlbmk	885.7	1,405.1	**177.1**	785.9	4,057.7	541.2
	swim	1,444.2	1,821.2	**558.4**	1,306.7	1,469.2	1,140.0
	twolf	464.4	952.9	**119.0**	389.5	1,174.3	287.4
	vpr	5,002.5	7,570.1	**899.8**	3,812.0	9,378.3	2,386.0
	har_mean	1,186.7	1,860.3	**332.2**	978.5	2,712.8	794.2
	geo_mean	2,546.0	3,199.6	**503.6**	2,052.7	4,784.2	1,603.7
	arith_mean	5,296.3	4,557.3	**727.1**	4,221.0	7,890.6	3,121.4

Looking at the time spent in the two stages, we notice a striking difference between SBC and SEQUITUR on the one hand and VPC4 on the other hand. Both SBC and SEQUITUR spend most of the total compression time in the first stage. In VPC4, the first stage only contributes 18% to 22% of the compression time, meaning that the separation of the traces into the four streams happens very quickly. Only SEQUITUR spends less time in the second stage than VPC4 (on the cache-miss and the load-value traces). All other algorithms (and SEQUITUR on the store-address traces) spend more time in BZIP2 than VPC4 does. This is the result of our efforts to tune VPC4's first stage to convert the traces into a very amenable form for the second stage (see Sections 4.4 and 4.5).

Table 5 lists the absolute compression time in seconds for the six algorithms on each individual trace. Only the total compression time over both stages is included. We highlighted the shortest compression time for each trace in bold print. The table also shows the harmonic, geometric, and the arithmetic mean of the compression times for each trace type.

VPC4 compresses 53 of the 55 traces faster than the other five algorithms and is also the fastest compressor on average. MACHE is 13% faster on the mgrid cache-miss trace and PDATS II is 29% faster on the vpr cache-miss trace. VPC4 compresses the store-address trace art 1569% faster than the other algorithms do.

6.5 Second Stage Compressor

Table 6 shows the geometric-mean compression rate, decompression time, and compression time of SBC and VPC4, the two best-performing trace compressors in the previous sections, over our three trace types when GZIP is used as the second stage instead of BZIP2. We use GZIP version 1.3.3 with the '--best' option.

Table 6: Geometric-mean performance of SBC and VPC4 with GZIP.

	store addresses		cache misses		load values	
	SBC	VPC4	SBC	VPC4	SBC	VPC4
compression rate	**235.5**	232.0	26.2	**26.6**	18.9	**31.1**
decompression time	65.4	**51.4**	26.1	**23.8**	132.1	**111.6**
compression time	2,130.3	**277.8**	1,752.2	**373.3**	4,177.2	**1,184.7**

As the results show, VPC4 roughly retains its performance advantage with the GZIP stage. It is faster and delivers a higher mean compression rate than SBC except on the store-address

traces, which SBC with GZIP compresses 1.5% more than VPC4 with GZIP. Comparing the VPC4 results from Table 6 with those from Table 2, we find that VPC4 compresses better and faster with BZIP2 but decompresses faster with GZIP.

7. SUMMARY AND CONCLUSIONS

This paper presents the four VPC trace-compression algorithms, all of which employ value predictors to compress program traces that comprise PCs and other information such as register values or memory addresses. VPC4, our most sophisticated approach, converts traces into streams that are more compressible than the original trace and that can be compressed and decompressed faster. For example, it compresses SPECcpu2000 traces of store-instruction PCs and effective addresses, traces of the PCs and the addresses of loads and stores that miss in a simulated cache, and traces of load instruction PCs and load values better and compresses and decompresses them more quickly than SEQUITUR, BZIP2, MACHE, SBC, and PDATS II. Based on these results, we believe VPC4 to be a good choice for trace databases, where good compression rates are paramount, as well as trace-based research and teaching environments, where a fast decompression speed is as essential as a good compression rate.

VPC4 features a single-pass linear-time algorithm with a fixed memory requirement. It is automatically synthesized out of a simple user-provided description, making it easy to adapt VPC4 to other trace formats and to tune it.

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 the anonymous reviewers for their helpful feedback.

9. REFERENCES

[1] J. Ahola. “Compressing Address Traces with RECET.” *2001 IEEE International Workshop on Workload Characterization*, pp. 120-126. December 2001.

[2] 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.

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

[4] 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.

[5] 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.

[6] M. Burtscher and N. B. Sam. “Automatic Generation of High-Performance Trace Compressors.” *2005 International Symposium on Code Generation and Optimization*, pp. 229-240. March 2005.

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

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

[9] I. K. Chen, J. T. Coffey and T. N. Mudge. “Analysis of Branch Prediction via Data Compression.” *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 128-137. October 1996.

[10] 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.

[11] 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.

[12] E. Federovsky, M. Feder and S. Weiss. “Branch Prediction based on Universal Data Compression Algorithms.” *25th International Symposium on Computer Architecture*, pp. 62-72. June 1998.

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

[14] 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.

[15] O. Hammami. “Taking into Account Access Pattern Irregularity when Compressing Address Traces.” *Southeastcon*, pp. 74-77. March 1995.

[16] A. Hamou-Lhadj and T. C. Lethbridge. “Compression Techniques to Simplify the Analysis of Large Execution Traces.” *10th International Workshop on Program Comprehension*, pp. 159-168. June 2002.

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

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

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

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

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

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

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

[24] 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.

[25] 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.

[26] 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.

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

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

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

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

[31] 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.

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

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

[34] 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.

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

[36] 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.

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

[38] A. R. Pleszkun. "Techniques for Compressing Program Address Traces." *27th Annual*

IEEE/ACM International Symposium on Microarchitecture, pp. 32-40. November 1994.

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

[40] 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.

[41] 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.

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

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

[44] 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.

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

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

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

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

[49] 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.

[50] 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.