Compressing Extended Program Traces Using Value Predictors

Martin Burtscher
Computer Systems Laboratory
Cornell University, Ithaca, NY 14853
burtscher@csl.cornell.edu

Metha Jeeradit
Computer Systems Laboratory
Cornell University, Ithaca, NY 14853
mj48@cornell.edu

ABSTRACT

Trace files record the execution behavior of programs for future analysis. Unfortunately, nontrivial program traces tend to be very large and have to be compressed. While good compression schemes exist for traces that capture only the PCs of the executed instructions, these schemes can be ineffective on extended traces that include important additional information such as register values or effective addresses. Our novel, value-prediction-based approach compresses extended traces up to 22.8 times better and about two and a half times as well on average. In addition to the higher compression rate, our lossless single-pass algorithm has a fixed memory requirement and compresses traces faster than other algorithms. It achieves compression rates of up to 6170. This paper describes the design of our compression method and illustrates how value predictors can be used to effectively compress extended program traces.

1. Introduction

Program execution traces are widely used to study program and processor behavior. Unfortunately, even capturing only a byte of information per executed instruction generates on the order of a gigabyte of data per second on a modern, high-end microprocessor. Hence, storing traces of nontrivial programs poses a serious problem, even with today's cheap and large hard disks. Of course, the solution is to compress the traces.

Compressing program traces that record only the program-counter values (PCs) of the executed instructions is relatively easy since PCs are range limited and tend to repeat frequently. Powerful algorithms to compress such traces already exist [19, 22, 23, 24, 38]. The goal of the work presented in this paper is to successfully compress *extended* traces, i.e., traces that contain additional information such as effective addresses, values on a bus, or the content of registers. Such values usually repeat less and span much larger ranges than PCs, making it harder to compress them well. We believe extended traces are of particular interest nowadays as more and more researchers investigate dynamic activities in computer systems.

We propose to employ techniques from the value-prediction literature to compress the 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 hardware-based value predictors have been developed to accurately predict the content of registers [3, 7, 8, 20, 21, 29, 30, 32, 34]. Hence, they are good candidates for predicting the kind of values we are concerned with, that is, values that span large ranges and that do not necessarily repeat often. In fact, since we are implementing the predictors in software, we can use more predictors with larger tables than are practical in hardware implementations, thus boosting the prediction accuracy. Indeed, our algorithm compresses hard-to-compress extended traces up to 22.8 times better and 2.6 times as well on average as preexisting schemes.

To illustrate the basic idea behind our algorithm, let us assume we have one predictor and that the extended data consist of 64-bit values. Instead of writing each trace entry directly to a file, we first compare it with the predicted value. If the two values are identical, we write only one bit, say a 1, to indicate that the predictor is correct. If the two values differ, we write a 0 followed by the unpredictable 64-bit value. This is a simple form of differential encoding. In either case, the predictor is updated with the true value and the procedure repeats for the remaining trace entries.

Decompression proceeds analogously. First, one bit is read from the compressed trace. If it is a 0, the next 64 bits are read to obtain the actual value. If the bit is a 1, the value from the predictor is used. The predictor is then updated to keep its state consistent with the state it was in during compression. The process is iterated until the entire trace has been reconstructed.

Our actual compression algorithm is more sophisticated, but the general principle is the same. We use multiple predictors, employ schemes to compress the unpredictable values, utilize dynamic Huffman coders, etc. See Section 3 for more detail.

The above example requires 65 bits to encode an unpredictable value but only one bit for a predictable value. Hence, the predictor needs to correctly predict more than one out of every 64 entries to make the compressed trace smaller than the uncompressed trace. In other words, the prediction accuracy needs to be at least 1.6% for this algorithm to be useful.

Of course, our goal is not only to compress traces but also to compress them well. In particular, our algorithm should outperform preexisting algorithms such as LZ77 [38], LZW [35], and Sequitur [19, 22, 23, 24]. Moreover, we were opting to design a compression utility for extended traces that meets the following criteria.

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

We need a lossless compression algorithm so that the original trace can be reconstructed exactly, which is necessary 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 whole 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. We want a fixed memory requirement to guarantee that if a computer can compress and decompress one trace, it can compress and decompress all our traces, regardless of the trace content and length. (Sequitur's memory requirement, on the other hand, depends on the data to be compressed, which caused problems when we tried to compress extended traces.) Naturally, fast decompression speeds are desirable. Finally, fast compression is important in real-time and academic environments.

Our algorithm meets the above requirements with the decompression speed being the only weak point. The algorithm runs in a single pass in linear time over the data both during compression and decompression and does not allocate any memory while processing a trace. The compression rate and speed are good, outperforming gzip (with the --best option), sequitur, and lz77. For example, our algorithm compresses a 4.36-gigabyte SPECcpu2000 *gcc* trace of load instruction PCs and values by a factor of 23.0 in 33 minutes. Decompression takes 25 minutes. Sequitur compresses the same trace by a factor of 13.4 in 214 minutes, but decompresses it in seven minutes. Section 6 presents results for more algorithms and traces.

The C source code for our value-prediction-based compression utility is available to the research and teaching community. The code can be found at http://www.csl.cornell.edu/~burtscher/research/tracefilecompression/. A sample test file and a short tutorial on how to adapt the code to other trace formats are also included. The code has been tested and compiled on UNIX systems using cc and gcc as well as on Windows under cygwin [10].

The remainder of this paper is organized as follows. Section 2 introduces the trace format and the value predictors we use. Section 3 describes our compression and decompression algorithm. Section 4 summarizes related work. Section 5 explains the evaluation methods. Section 6 presents the results. Section 7 points out directions for future work and Section 8 concludes the paper.

2. Background

2.1 Trace Format

We use a generic trace format throughout this paper to keep the discussion simple. Our traces consist of pairs of numbers. The first number in a pair records the PC of an executed instruction and the second number records the corresponding extended data (ED). The PCs are 32 bits wide and the extended data are 64 bits wide. The uncompressed trace begins with a 32-bit header that encodes an approximation of the range of the PCs in the trace (see below). Thus, our traces have the following format (the subscripts indicate bit widths).

$$\mathbf{Range_{32}, PC\_0_{32}, ED\_0_{64}, PC\_1_{32}, ED\_1_{64}, \ldots}$$

It would be straightforward to add a file magic, a length field, etc. Other improvements such as storing only one PC per basic block are also possible but would unnecessarily complicate the presentation of our algorithm. We chose 64 bits for the ED entries because this is the native word size of the Alpha machine on which we performed our measurements.

$0 \le PC < range$ has to hold for all PCs in the trace, but not all PCs in the range have to occur. In our traces, we use the number of (static) instructions in the program as the range and assign each instruction a virtual PC between zero and *range-1*. With a trace-generation tool such as ATOM [6, 31], generating these virtual PCs is as easy as providing actual PCs. Note that no pass over the data is required to obtain this information, as it is a statically known property of the binary. All our binaries are non-shared, i.e., they do not load dynamically linked libraries.

2.2 Value Predictors

We investigated a wide variety of value predictors and decided to use the following predictors, which have been experimentally determined to result in a good balance between the speed and the compression rate of our algorithm on the load-value traces. See also Section 5.3.

Last *n* value predictor: The first type of predictor we use is the last *n* value predictor [2, 20, 34]. It predicts the most likely value among the *n* most recently seen values. To improve the compression rate, we use all *n* values (and not only the most likely value) and only update the predictor if the update value is not already among the *n* values in the selected predictor line. If it is, a second copy of the value is not added but instead the value is moved to the front, which essentially makes the predictor a last *n* distinct value predictor with a least recently used (LRU) replacement policy. We found *n* = 6 to work well. We only use the last-six-value predictor for predicting the extended data but not for the PCs.

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 [7]. Adding this difference to the most recent value yields the prediction. Stride predictors can predict sequences that look as follows.

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

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 has been proposed [30], which uses a second stride. The second stride is updated only if the same stride is encountered at least twice in a row, thus providing some hysteresis before the predictor switches to a new stride. We use stride 2-delta predictors for predicting the PCs and the extended data.

Finite context method predictor: The finite context method predictor (FCM) [29, 30] stores the *n* most recently seen values in a sliding window (FIFO). *n* is referred to as the order of the predictor. The predictor records the value that follows every seen sequence of *n* values in a hash table. When making a prediction, a lookup is performed to find out if the current sequence of the *n* most recent values has already been encountered before. If so, the value that followed this sequence last time becomes the predicted value. Most FCMs, including ours, do not actually check whether the sequence has already been seen before but simply accept the hash-table entry to speed up the predictor operation and to reduce the storage requirement [26, 27, 29]. We use several FCMs with different orders and the hash function proposed by Burtscher [1]. For the extended data, we use orders one, two, three, and six. For the PCs, we only use a sixth-order FCM. The FCM predictors include saturating counters in the hash table to provide an update hysteresis. Thus, entries that have provided correct predictions are only replaced after having been wrong at least twice in a row.

Differential finite context method predictor: The differential finite context method predictor (DFCM) works exactly like the FCM. The only dissimilarity is that it predicts and is updated with differences (strides) between consecutive values rather than absolute values [8]. The predicted stride has to be added to the most recently seen value to form the final prediction.

Again, we use several different orders of DFCMs to maximize the chance of at least one predictor being correct. Moreover, we retain up to four values in each DFCM hash-table entry. As with the last six value predictor, we make sure that the four values are distinct and use an LRU replacement policy. Section 3.1 explains the reasons for these choices. Since our DFCM predictors can hold multiple values, we found that no update hysteresis is needed.

The DFCM predictors are the workhorses of our algorithm. We use orders four and six with two values per hash-table entry and order five with four values per entry for predicting the PCs. For the extended data, we use orders one, two, and six with four values, order three with two values, and order four with one value per table entry.

Global/local last value predictor: This is a new type of value predictor that we have developed for a different project. We use it for compressing the extended data only. It works like a last-value predictor, with the exception that each table entry contains two additional fields: an index and a counter. The index designates which entry of the last-value table contains the prediction. The counter assigns a confidence to the index. If the confidence is low and the indexed value incorrect, the index is incremented (modulo the table size) so that a different entry will be checked next time. This way, the predictor is able to correlate any instruction with any other instruction but without the need for multiple comparisons per prediction or update (i.e., without associativity).

3. Algorithm

3.1 Compression

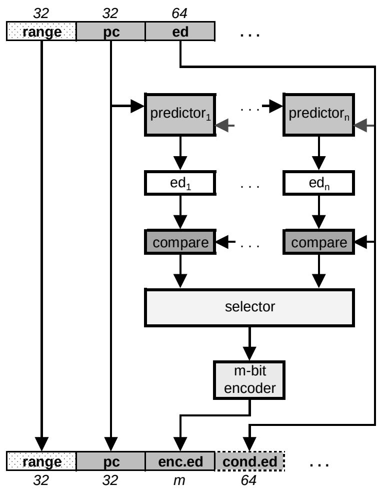

Figure 1: Initial compression algorithm.

Our first attempt at a value-predictor-based compression algorithm compressed only the extended data and worked 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 number is written to the compressed file using a fixed *m*-bit encoding. If no predictor is correct, an *m*-bit dummy predictor number is written followed by the unpredictable 64-bit value. Then the predictors are updated (light arrows). The algorithm repeats for the remaining PC/ED pairs in the trace. Figure 1 illustrates the process for the first PC/ED pair in the trace.

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 a dummy predictor, five bits are needed to encode a predictor number (i.e., *m*=5). Furthermore, unpredictable entries are not compressed. Overall, the algorithm cannot exceed a compression rate of 2.6 because even in the best case, a 96-bit PC/ED pair (32-bit PC plus 64-bit extended data) is compressed to 37 bits (32-bit PC plus 5-bit predictor number).

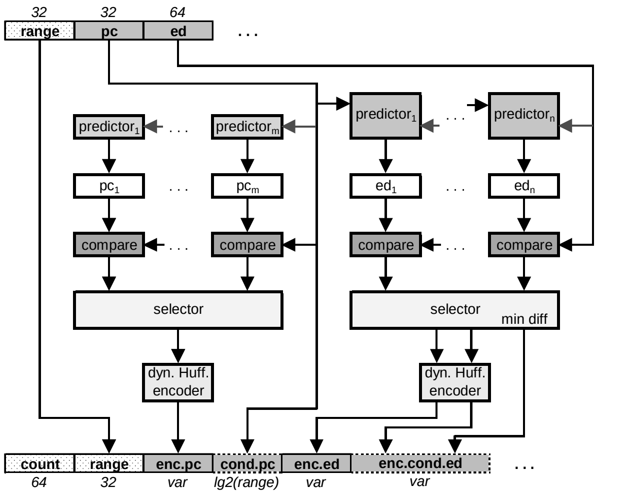

Figure 2: Vpc compression algorithm.

Our vpc algorithm corrects the above-mentioned shortcomings. First, it compresses the PCs like the ED using a (separate) set of predictors. Moreover, it encodes the predictor numbers using a dynamic Huffman encoder [17, 33] to minimize the number of bits required to express a predictor number. If more than one predictor is correct, we pick the one that has the shortest code associated with it, i.e., the one with the highest usage frequency. Finally, our algorithm compresses the unpredictable values in the following manner. In case of PCs, only $\lceil \log_2(\text{range}) \rceil$ bits are written after the Huffman code for the dummy predictor. All dropped bits are zero and can therefore be trivially reconstructed during decompression. In case of extended data, the dummy-predictor code is followed by the encoded number for the predictor whose prediction is closest to the actual value in terms of absolute difference. We then write the difference between the predicted value and the actual value in encoded sign-magnitude format to save bits. Figure 2 illustrates the operation of vpc's compression algorithm on the first PC/ED pair.

We implemented several additional enhancements to increase the compression rate. First, we use an improved hash function for the FCM and DFCM predictors, which has been shown to utilize the hash table more effectively [1] and thus increases the number of correct predictions. Second, we added saturating up/down counters to the hash-table entries in the FCMs to provide an update hysteresis (see Section 2.2). Third, 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. Fourth, we keep the values in all multi-value predictors in least recently used order to skew the usage frequency of the predictor components. This works because of value locality [7, 21], i.e., because the most recently seen value has a higher chance of resulting in a correct prediction in the near future than an older value. Skewing the usage frequencies increases the compression rate because it allows the dynamic Huffman encoder to assign shorter codes to the frequently used components and to use them more often. Fifth, we initialize the dynamic Huffman encoder with biased, non-zero frequencies for all predictors. Doing so assigns valid codes to all predictors from the start. Hence, we never have to dynamically extend the symbol set, which would complicate and slow down the algorithm. Moreover, we bias the more sophisticated predictors more heavily than the simpler predictors. If we did not do this, the more sophisticated predictors would initially be assigned long codes because they take longer to warm up. As a consequence, the simpler predictors would be chosen whenever possible and the more sophisticated predictors would only get to predict the values that the simple predictors cannot. Thus, the more sophisticated predictors could not catch up to the simpler predictors, resulting in suboptimal codes because all predictors would end up being used relatively often. Biasing the frequencies in the beginning 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.

3.2 Decompression

To decompress the trace, the compression steps are simply reversed. Since regular files have a granularity of bytes but our algorithm requires a bit granularity, it is not always possible to determine the end of the compressed trace from the file length, which is why an additional header field is needed. We use a 64-bit count to identify the end of the trace, which also allows us to check if the file has been corrupted. Figure 3 illustrates the decompression process.

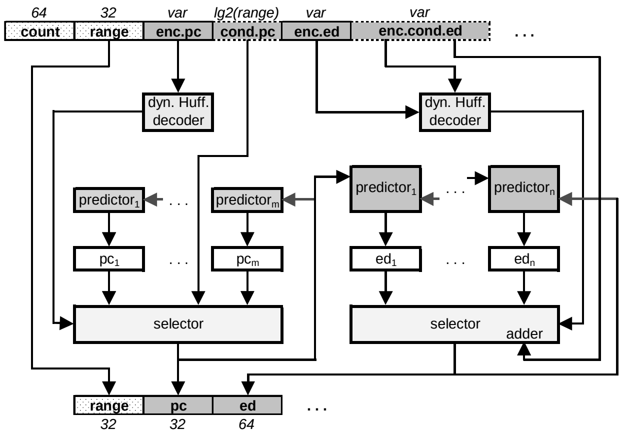

Figure 3: Vpc decompression algorithm.

4. Related Work

Most early trace compression techniques are lossy because they employ filtering or sampling methods. The few lossless approaches concentrate mostly on address traces. Larus proposed Abstract Execution [18], 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 compact representation [25]. Other lossless trace compression algorithms include Mache [28], PDATS [14], PDI [15], and LD&R [5]. Mache, PDATS, and PDI work by exploiting spatiality (address differences) and sequentiality (repeat counts) of the trace. These algorithms use a second compression with an LZ77 [38] or LZW [35] algorithm to boost the overall compression rate. LD&R (Loop-Detection and Reduction) detects loops in the address traces and extracts those references that are constant or change by a constant value between loop iterations before encoding the remainder of the references. While our approach incorporates some of the same ideas (it also exploits sequentiality and spatiality and benefits from a second compression step), the above-mentioned algorithms do not reach our algorithm's compression rate because our algorithm can exploit a number of different patterns in addition to strided sequences and repetitions.

4.1 Compression Algorithms

This section describes the compression schemes with which we compare our approach in Section 6. The first two are general-purpose algorithms that can be used to compress any kind of file. The last two are special-purpose algorithms that are tailored to specifically exploit our trace format.

compress: The UNIX command compress implements the dictionary-based LZW algorithm [35]. The dictionary starts out with 512 entries and doubles in size whenever it gets full. Once the maximum of 65,536 entries is reached, the algorithm continues to use the existing dictionary but does not update it anymore. If the compression rate falls below a predefined threshold, the old dictionary is abandoned and a new one is started, allowing the algorithm to adapt to the next "block" in the file. Compress operates at a byte granularity. Its memory usage is insignificant compared to the other algorithms.

gzip: Gzip is another general-purpose compression utility found on most UNIX systems [11]. It also operates at a byte granularity and implements a variant of the LZ77 algorithm [38]. 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 --best). With lazy evaluation, the algorithm does not immediately use the matched sequence for the currently processed byte but first compares it to the matched sequence of the next input byte before selecting the longer of the two matches. According to *ps*, the algorithm requires approximately 2MB of memory when compressing our traces.

lz77: Since gzip yields substantially better compression rates on our traces than compress (see Section 6.1.2), we modified gzip's underlying LZ77 algorithm to handle the PC and ED streams separately (split-stream approach) and to exploit our trace format. The resulting special-purpose algorithm works on the granularity of trace entries. It uses a 32,768-entry sliding window and a 256-entry lookahead buffer for each stream. String duplications are found using a 65,536-entry hash table. The hash-table entries are truncated if more than 32,768 strings with same start symbol are encountered. The memory usage of this algorithm is approximately 12MB. Note that our implementation of this algorithm is not optimized for speed and could be made significantly faster. We include it only to see how well a gzip-like algorithm can compress on our traces when it is made aware of the trace format.

sequitur: The sequitur algorithm identifies hierarchical structures in the input sequence and converts them into a context-free grammar [22, 23, 24]. The algorithm applies two constraints while constructing the grammar: each digram 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. We modified Larus' implementation of sequitur [19] in two ways. First, we construct two separate grammars in parallel, one for the PCs and the other for the extended data (split streams). Second, we start new grammars once the combined size of the two grammars reaches 800MB to limit the algorithm's memory usage. The actual memory usage ranges from 20MB to around 1GB depending on the trace. Sequitur is one of the best trace compression algorithms in the current literature.

5. Evaluation Methods

5.1 System

Unless otherwise noted, all our measurements were performed on a 64-bit Alpha system with two 750MHz 21264A CPUs [16]. Only one of the processors was used at a time, allowing the other CPU to handle daemons and other tasks to improve the timing accuracy. Both processors have separate, on-chip, 2-way set-associative, 64kB L1 caches, a unified, direct-mapped 8MB L2 cache, and share 1.5GB of main memory. The SCSI Ultra2/LVD hard drive has a capacity of 18GB (with about 10GB free) and spins at 10,000rpm. The operating system is Tru64 UNIX V5.1.

5.2 Traces

We used the eleven integer and four floating-point C programs from the SPECcpu2000 benchmark suite [13] to generate the traces for this study. All programs were compiled with a high optimization level using the bundled C compiler and are run to completion with the SPEC-provided test inputs.

We generated three types of traces from these programs to evaluate our compression utility. We picked traces for which we are aware of ongoing research that could benefit from the traced information.

The first type captures the PC and load value of every executed load instruction (that is not a prefetch, a NOP, or a load immediate). The second type of trace contains the PC and target of all indirect branch instructions. The third type of trace stores the PC and effective address of each executed store instruction. Note that our algorithm has only been optimized for the PC/load-value traces but not for the other two types of traces.

Table 1 shows the uncompressed size (in megabytes) of the three traces for each program as well as which traces we excluded. We excluded all traces with a size above ten gigabytes or below ten megabytes. The former are excluded because they exceed the capacity of our hard drive and the latter because we consider them too short to be of interest since they can be compressed and decompressed in a matter of seconds.

Table 1: Size of the studied traces.

		trace size in megabytes		
		load values	indirect branch targets	store effective addresses
integer	gzip	**7,881.4**	too small	**3,102.5**
	vpr	**6,384.6**	too small	**1,857.6**
	gcc	**4,361.7**	**80.0**	**2,151.6**
	mcf	**456.6**	too small	**405.5**
	crafty	too large	**105.4**	**3,119.0**
	parser	**9,002.0**	too small	**4,023.4**
	perlbmk	**1,026.0**	**32.9**	**527.6**
	gap	**2,988.6**	**135.7**	**1,238.9**
	vortex	too large	**26.8**	too large
	bzip2	too large	too small	too large
	twolf	**684.8**	too small	**190.0**
floating pnt	mesa	**5,972.7**	**259.6**	**3,655.4**
	art	**4,466.0**	too small	**1,730.4**
	equake	**3,701.0**	**32.2**	**1,264.4**
	ammp	too large	**21.1**	**3,813.0**

5.3 Predictor Configurations

This section lists the sizes and parameters of the predictors our algorithm uses. They are the result of experimentation with the load-value traces to balance the speed and compression rate as well as the fact that we wanted to share tables between the predictors. We used these configurations to obtain the results presented in Section 6.

Since no index is available for the PC predictors, all PC predictors are global predictors. Accordingly, their first levels are very small. The stride 2-delta predictor requires only 24 bytes of storage: eight bytes for the last value plus sixteen bytes for the two stride fields. The sixth-order FCM 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). The DFCM predictors share a first-level table, which requires six two-byte fields for retaining the hash values. The most recent value is obtained from the stride predictor. The fourth-order DFCM has two second-level tables of one megabyte each (131,072 lines), the fifth-order DFCM has four tables of two megabytes each (262,144 lines), and the sixth-order DFCM has two tables of four megabytes each (524,288 lines). Since we do not use saturating counters in the DFCMs, 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 use the PC as an index, which allows them to store information on a per instruction basis. Their first-level tables are correspondingly larger. The last six value predictor uses six tables of 128kB (16,384 lines). The stride 2-delta predictor requires 256kB of table space for the strides (16,384 lines). The last-value table is shared with the last six value predictor. The FCM predictors share six first-level, 32kB tables, each holding 16,384 two-byte hash values. Similarly, the DFCMs share a set of six 32kB first-level tables. The second-level table sizes are as follows. The first-order FCM uses a 144kB table (16,384 lines holding an eight-byte value plus a one-byte counter), the second-order FCM uses a 288kB table (32,768 lines), the third-order FCM has a 576kB table (65,536 lines), and the six-order FCM requires 4.5MB (524,288 lines). The first-order DFCM has four tables of 128kB (16,384 lines holding an eight-byte value), the second-order DFCM uses four 256kB tables (32,768 lines), the third-order DFCM requires two 512kB tables (65,536 lines), the fourth-order DFCM has one 1MB table (131,072 lines), and the sixth-order DFCM needs four 4MB tables (524,288 lines). Finally, the global/local last value 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 to the predictor tables in our compression algorithm. Including the code, libraries, padding, etc., our compression utility requires 88MB of memory to run as reported by the UNIX command *ps*.

6. Results

The following sections describe the results. Section 6.1 focuses on the load-value traces, Section 6.2 on the indirect-branch-target traces, and Section 6.3 on the store-effective-address traces.

6.1 Load-Value Traces

6.1.1 Predictability

The effectiveness of our compression algorithm hinges on the predictability of the trace entries. To reach good compression rates, the prediction accuracy has to be in the nineties. This is much higher than what value predictors normally deliver. The predictors in the literature are geared towards hardware implementations and therefore tend to be small and less accurate. More importantly, they are allowed to predict only one value at a time. Furthermore, if multiple predictors are used (hybrid), a selector has to choose one of them, which introduces additional inaccuracies [3]. Our algorithm does not have these restrictions. First, it is easy to make the predictor tables large in software (Section 5.3). Second, our selection process is perfect since we know ahead of time what value we are looking for and therefore which predictor to choose. Consequently, we can make effective use of a large number of predictors, which is why we are able to achieve the high prediction accuracies required for good compression. Figure 4 shows the predictability of the PCs and load values in the eleven load-value traces.

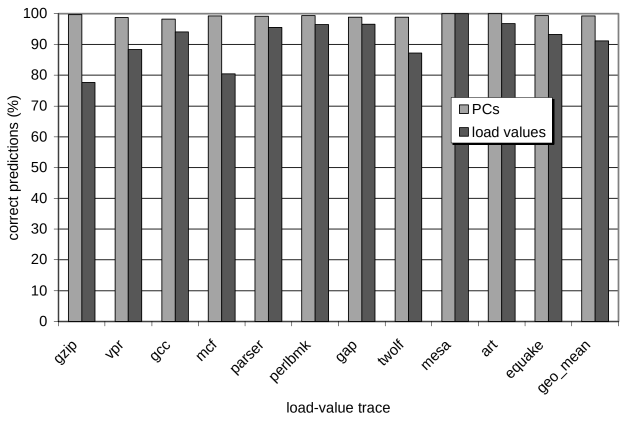

Figure 4: Predictability of the load-value trace entries.

As the figure illustrates, the set of predictors we use can indeed predict most of the trace entries correctly, especially the PCs, which we expected to be more predictable (and compressible) than the load values. Even in the worst case (*gcc*), 98.2% of the PCs are predicted correctly by at least one of the ten PC predictors, with an average of 99.3%. On average, 91.2% of the load values are predictable. In the worst case (*gzip*), 77.6% of the load values can be predicted by at least one of the 27 extended-data predictors.

While these numbers are encouraging, they are at best an approximation of the resulting compression rate, which is our true metric. The compression rate depends not only on how many of the trace entries are predictable but also on which predictor can predict them and when a prediction is made since the length of the Huffman codes is different for different predictors and changes over time.

6.1.2 Compression Rate

Figure 5 shows the compression rates of the four compression algorithms discussed in Section 4.1 as well as vpc, our value-prediction-based approach, on the load-value traces. Vpc2 is discussed below.

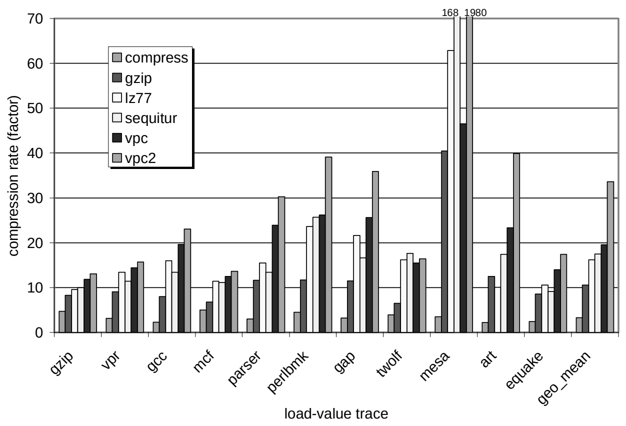

Figure 5: Compression rates on the load-value traces.

For nine of the eleven traces, vpc reaches higher compression rates that the four non-vpc schemes. Sequitur and lz77 outperform vpc slightly on *twolf* and significantly on *mesa*.

The reason for vpc's relatively poor performance on *mesa* is that it 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. The fact that vpc almost reaches this compression rate on *mesa* (factor 46.5) is reassuring. It shows that the dynamic Huffman encoder works well and almost always requires only one bit to encode a predictor number. Note that 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 code at a time. Since PC and ED predictor codes alternate in our compressed traces (Section 3.1), the vpc-compressed *mesa* trace should therefore contain long bit strings of zeros, ones, or alternating zeros and ones, depending on whether both predictors are assigned a code of 0, both predictors are assigned a code of 1, or one predictor's code is 0 and the other one's code is 1. This, of course, means that the compressed trace is itself highly compressible. To exploit this fact, we further compressed the vpc-compressed traces with gzip. We call this scheme vpc2.

As Figure 5 shows, vpc2 works very well and improves upon vpc in all cases. It compresses *mesa* by a factor of 1980.6, which is 11.8 times as much as sequitur. There is only one trace (*twolf*) on which sequitur slightly outperforms vpc2. Vpc2's geometric mean compression rate is almost twice that of sequitur and more than twice that of the other three non-vpc schemes. Note that all schemes (including gzip itself) benefit from an additional gzip compression step albeit not nearly as much as vpc.

6.1.3 Compression/Decompression Speed

A good compression rate is highly desirable but at the same time useless if it can only be attained at an extraordinary cost in compression and particularly in decompression time.

Figure 6 shows the average (geometric mean) time in minutes it takes to compress, transfer, and decompress the load-value traces when they are sent over an Internet connection with a throughput of 330 kilobytes per second, as we measured between Cornell University and the University of Colorado at Boulder.

Our algorithm is the second fastest at compressing the traces in almost all cases and also on average. Only compress is faster, but its compression rate is by far the lowest. Vpc takes on average 18.8 minutes and vpc2 21.7 minutes to compress one of the traces. The same task takes compress 6.8, gzip 51, sequitur 158, and lz77 259 minutes. This is surprising because the slower algorithms do not reach vpc's and vpc2's compression rates.

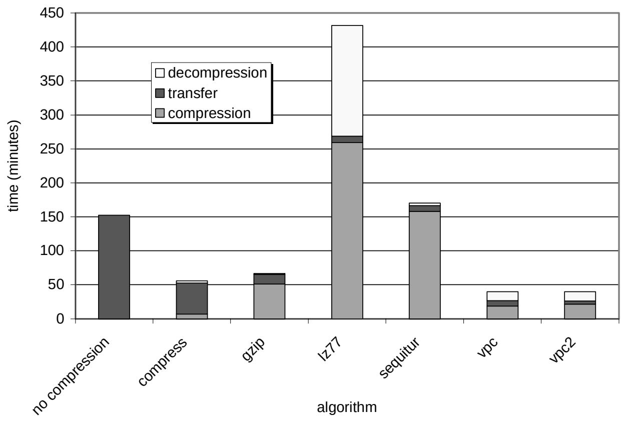

Figure 6: Average time to compress, send, and decompress a load-value trace.

Unfortunately, vpc2 is the second slowest algorithm at decompressing the traces. It takes vpc 13.3, vpc2 13.5, compress 3.6, gzip 1.3, sequitur 3.7, and lz77 162.8 minutes on average to decompress the load-value traces. However, our algorithms easily make up for the longer decompression time by the fast compression and transfer speed. Since vpc2 has the highest compression rate, it also incurs the shortest transfer time. On average, it delivers (i.e., compresses, transfers, and decompresses) the traces 41% faster than compress and 3.8 times as fast as no compression.

Interestingly, compress is the second best choice due to its fast compression and decompression speed and despite its low compression rate. Using sequitur or lz77 is slower than using no compression at all.

Note how little time the gzip step adds to vpc2's compression and in particular decompression time. The latter is not surprising as gzip has the fastest decompression speed. The extra compression time is not very high, either, because gzip starts out with an already compressed trace, i.e., a much shorter trace.

Clearly, transferring traces over a network benefits from compression. However, a more likely scenario is to drive a trace consumption tool (e.g., a simulator) from a locally stored, compressed trace. In such a case, only the decompression speed matters. Figure 7 shows the decompression times of the six algorithms on the load-value traces. Since the times vary greatly, the y-axis is logarithmic.

Except for lz77, which is not optimized for speed, vpc and vpc2 are slower at decompressing the traces than the other schemes (see Section 7 for possible remedies). Due to the symmetric nature of our algorithm, compression and decompression take roughly the same amount of time. On average, sequitur is 3.6 times faster at decompressing the traces. Note, however, that in spite of its relatively slow decompression speed, our algorithm still recreates the original traces at 3.5 megabytes per second.

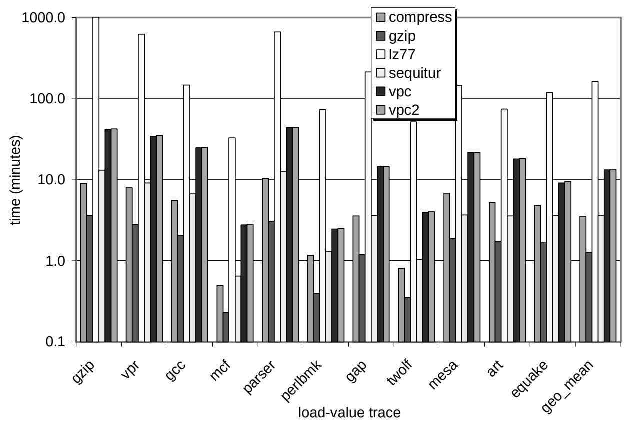

Figure 7: Decompression time on the load-value traces.

6.2 Indirect-Branch-Target Traces

Since we optimized several aspects of our algorithm (in particular the type and configuration of the predictors) to work well with the load-value traces, we also wanted to test our algorithm on traces for which it has not been optimized.

One type of trace we chose for this experiment consists of the PCs and targets of indirect branches. The compression rates on these traces are shown in Figure 8.

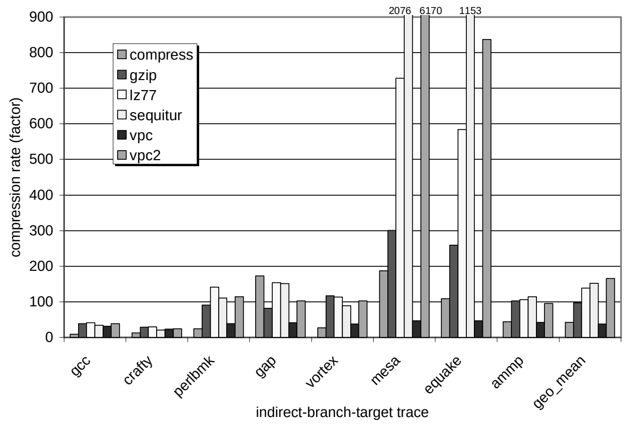

Figure 8: Compression rate on the indirect-branch-target traces.

Since branch targets are themselves PCs, the indirect-branch-target traces effectively comprise only PCs. This is why they are so much more compressible than the load-value traces, as is evident from the high compression rates shown in Figure 8. While no algorithm we investigated was able to compress the load-value traces by more than a factor of 35, sequitur and vpc2 compress the indirect-branch-target traces by more than a factor of 150 on average. Vpc2 compresses *mesa* by factor of 6170.

Vpc achieves an average compression rate of 38 on these traces. The reason for its poor performance is the aforementioned maximum compression rate of 48, which it almost reaches on *equake* (47.3). On half of the traces, vpc exceeds a compression rate of forty.

On the majority of the indirect-branch-target traces, at least one of the other schemes outperforms vpc2, although never by more than 69%.

The indirect-branch-target traces are the only traces we studied in which the extended data were more predictable than the PCs. Even in the worst case, 99.6% of all the branch targets are predictable (*vortex*) compared to 98.4% for the PCs (*gcc*).

6.3 Store-Effective-Address Traces

The other kind of trace our compression scheme has not been optimized for tracks the PCs and the effective addresses of store instructions. The corresponding compression rates are depicted in Figure 9.

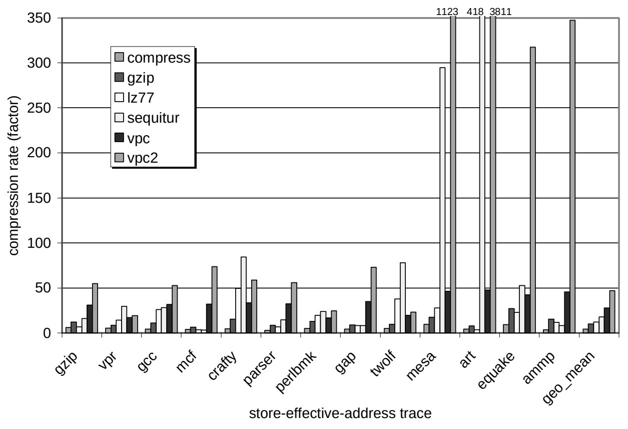

Figure 9: Compression rate on the store-effective-address traces.

Like the load-value traces and unlike the indirect-branch-target traces, the average attainable compression rates for the store-effective-address traces are below a factor of fifty for all approaches.

Sequitur outperforms vpc2 on *vpr*, *crafty*, and *twolf*, and lz77 outperforms vpc on *twolf*. In all other cases, vpc2 delivers the best compression rate, often by a large margin. It compresses *ammp* 22.8 times as much as the second best approach (gzip).

Vpc2's geometric mean compression rate is 2.6 times higher than that of the second best algorithm (sequitur). Note that vpc2 compresses all store-effective-address traces by at least a factor of 19.1 while the remaining schemes only reach minimum factors of 2.9 (compress) to 6.5 (gzip).

7. Future Work

While our compression algorithm delivers the highest compression rates and very fast compression, it is rather slow at decompression. To improve the relatively slow decompression speed, we intend to fuse the prediction and update code. In the current implementation, all the predictors are accessed once to make a prediction and then again to perform the updates. Since most of the time is spent computing the indices into the various tables, consolidating the prediction and update code should significantly speed up the compression and in particular the decompression as only half as many indices will have to be computed. Moreover, we are planning to integrate the gzip step into vpc2, which should further speed up the algorithm because no intermediate file will have to be created.

We intend to study traces from other programs as well as traces containing different information to further evaluate and improve our compression utility. We will also investigate additional compression schemes, both generic ones and ones that we adapt to take advantage of our trace format. For example, bzip2 [9] is of interest because of its high compression rates and lzop [12] because of its speed.

Our algorithm is modular and extensible, making it easy to add or remove predictor components. This way, we should be able to trade off compression speed for compression rate and vice versa. Also, when new predictors become available, we will incorporate them to see if they improve the compression rate further.

It is unclear whether our compressed traces reveal any interesting information about the original trace. This is a key strength of sequitur, which exposes hot program paths in the compressed format [19]. We will investigate whether, for example, the type of predictor used to compress a given PC/ED trace entry provides useful information.

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 [37]. We believe the same approach can be used to further improve the compression rate of our scheme.

Our current trace format requires decompressions to always start at the beginning of a trace, even if only a section from the middle of the trace is needed. We will investigate whether an approach like Zhang and Gupta's path traces would allow us to more quickly access information in the trace.

Another interesting idea whose applicability to our algorithm we would like to investigate is Chilimbi's hot data streams [4]. He uses sequitur to produce 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.

We are planning to replace the dynamic Huffman coder with an arithmetic coder [36], which should yield better compression rates because it can represent predictor codes with less than one bit on average.

Another possible extension of this work is to study the usefulness of special instructions to support compression and decompression in hardware.

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 a hybrid approach.

8. Conclusions

This paper presents a novel compression algorithm for program traces that contain extended data such as register values or effective addresses. Our approach uses a set of value predictors to compress the trace entries and delivers substantially improved compression rates, especially on traces where it matters the most, i.e., on traces that other algorithms cannot compress well. For example, our scheme compresses SPECcpu2000 traces of store-instruction PCs and effective addresses up to 22.8 times (2.6 times on average) as much as gzip, compress, lz77, and sequitur, even though we modified lz77 and sequitur to take advantage of our trace format. Moreover, the lowest compression rate of our algorithm on these traces is 19.1 while the other schemes reach rates of only 2.9 to 6.5. These results make our scheme ideal for trace databases and on-line trace collections.

In addition to the good compression rates, our approach features a single-pass linear-time algorithm, a fixed memory requirement, and fast compression. 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 utility and a brief tutorial are available at http://www.csl.cornell.edu/~burtscher/research/tracefile-compression/.

9. Acknowledgments

This work was supported in part by National Science Foundation grants CCR-0208567 and CCR-0312966.

10. References

[1] M. Burtscher. "An Improved Index Function for (D)FCM Predictors." *Computer Architecture News*, Vol. 30, No. 3, pp. 19-24. June 2002.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

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

[23] 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.

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

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

[27] 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.

[28] 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.

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

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

[31] 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.

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

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

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

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

[36] I. H. Witten, R. Neal, and J. G. Cleary. "Arithmetic Coding for Data Compression." *Communications of the Association of Computing Machinery*, Vol. 30, pp. 520-540. June 1987.

[37] 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.

[38] 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.