Online compression of cache-filtered address traces

Pierre Michaud
INRIA Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu
35042 Rennes Cedex, France
pmichaud@irisa.fr

Abstract

Trace-driven simulation is potentially much faster than cycle-accurate simulation. However, one drawback is the large amount of storage that may be necessary to store traces. Trace compression techniques are useful for decreasing the storage space requirement. But the compression ratio of existing trace compressors is limited because they implement lossless compression. We propose two new methods for compressing cache-filtered address traces. The first method, bytesort, is a lossless compression method that achieves high compression ratios on cache-filtered address traces. The second method is a lossy one, based on the concept of phase. We have combined these two methods in a trace compressor called ATC. Our experimental results show that ATC gives high compression ratio while keeping the memory-locality characteristics of the original trace.

1 Introduction

Research in computer architecture is generally based on using performance simulators. Detailed cycle-accurate simulation is, by definition, the most accurate simulation method. However, cycle-accurate simulation is generally very slow. With the increase of the number of cores in multicore processors, approximate simulation methods become necessary for being able to explore the design space efficiently [6, 7, 9, 10, 12]. Trace-driven simulation is an important tool in the simulation toolbox. There are several reasons why one may want to use traces, like simplicity, speed, memory efficiency, and repeatability. However, traces can take a huge amount of storage. It is possible to generate traces on-the-fly to avoid storing them, but then we may lose some of the advantages of trace-driven simulation. Therefore, trace compression techniques are useful for decreasing the storage space consumed by traces or for allowing to store longer traces. We propose two new methods for compressing cache-filtered address traces. The first method, *bytesort*, is a reversible transformation that makes lossless compression more effective. We show that, on cache-filtered address traces, bytesort yields higher compression ratios than TCgen, one of the most effective trace compressor existing [5]. The second method is a lossy compression method that exploits execution phases [26]. It makes traces significantly more compact while keeping the memory-locality characteristics of the original trace. We have combined these two methods in a trace compressor called ATC (*Address Trace Compressor*). The paper is organized as follows. We explain our motivation in Section 2. Related work is mentioned in Section 3. We describe the lossless compression method in Section 4 and the lossy one in Section 5, providing some experimental evaluation results. The ATC compressor is described in Section 6. Finally, Section 7 concludes this study.

2 Motivation

Our goal with the ATC compressor is to be able to generate very compact address traces so that it is possible to capture the memory behavior of applications when executed to completion. Existing lossless compressors, like TCgen [3], give high compression ratios on structured traces. However, the compression ratio they achieve is limited by the constraint of being lossless. In practice, this means we can store traces only for some parts of the execution. This may be sufficient for understanding the "microscopic" interactions between the application and the microarchitecture (e.g., branch mispredictions, level-1 cache misses, etc.). However, understanding the "macroscopic" memory behavior of an application necessitates long traces that may take a huge storage space. The traces that ATC takes as input have the simplest format that an address trace can have : they are just sequences of 64-bit values. We target cache-filtered address traces, i.e., traces consisting of the cache block addresses that are generated after filtering by one or more cache levels. In this study we assume 64-byte cache blocks, hence we work with block addresses whose 6 most significant bits are null. These bits may be used to store some extra information, e.g., whether the address corresponds to a demand miss or a write-back. Combined with some other simulation tools (e.g., cycle accurate simulators, SimPoint [28], etc.), cache-filtered address traces can be used to simulate a multicore memory hierarchy, including main memory. Traces may

be collected by multiple methods, e.g., by external hardware probes, by simulation, by code annotation, etc. [32]. When the trace is obtained by software means, the information consisting of the program code and the input data may be viewed as a compressed form of the trace. For example, for a trace consisting of all addresses and data values generated by a program, the program code and the input data may be the most efficient representation of the trace, both in terms of compression ratio and decompression speed. The term *trace compression* generally refers to the case where we collect some information generated by a tracing tool and we want to store this information in a form that is as compact as possible. Trace compression is useful if decompressing the trace is faster than replaying the tracing tool. This is the case for a cache-filtered address trace because it is generally faster to decompress the trace, which represents infrequent events (cache misses), than to replay the trace collector.

3 Related work

Trace compression methods can be either lossless or lossy. Several lossless trace compression methods have been proposed. It is possible to use general-purpose compression utilities like *gzip* or *bzip2*, but they are not optimal. Higher compression ratios can be obtained by specializing the compressor. The *Mache* lossless compression method targets labeled address traces [23]. It consists in applying a reversible transformation to the original trace such that the transformed trace can be compressed more easily by a general-purpose compressor. Basically, the original trace is transformed by replacing the full address with the difference between the address and the previous address having the same label. This transformation, which exploits spatial locality, reveals some patterns that were hidden in the original trace. A similar idea was used by Johnson and Ha [8]. Luo and John used a method similar to *Mache*, combined with the idea of maintaining dynamically a cache of frequently occurring trace records and replacing these records with their index in the cache, which can be coded with fewer bits [14]. Pleszkun proposed a lossless compression method for traces consisting of instructions and data addresses. A high compression ratio was obtained by using an ad hoc trace format exploiting spatial locality and repetitions in the branch outcomes and data address strides associated with particular instructions [21]. The SBC compression method proposed by Milenković and Milenković is akin to that proposed by Pleszkun but is implemented differently [16]. In particular, SBC is a single-pass method. In 1951, Shannon described how a sequence of symbols can be compressed by having two predictors that behave identically, one for coding and one for decoding [24]. In the first method proposed by Shannon, the predictor gives a single prediction for the next symbol. If the prediction is correct, the symbol is replaced with the information that the prediction was correct. If the prediction is wrong, the symbol is transmitted normally. In the second method proposed by Shannon, the predictor is able to predict the next symbol as many times as necessary until the prediction is correct. The original symbol is replaced with the number of tries that were necessary to guess the symbol correctly. The VPC family of compressors is based on a similar idea [4]. These compressors work on traces of records whose fields are memory addresses or data values. The predictor consists of several sub-predictors, e.g., last-value predictor, stride predictor, etc. If one of the sub-predictors is correct, the input data is replaced with the identifier of one of the correct sub-predictors, otherwise a special code is output, followed by the original data. The *TCgen* tool generates automatically VPC-like compressors from a user-specified trace format [5, 31]. It was shown that the compressors generated by TCgen have a very high compression ratio [5, 16]. Another predictor-based compressor was proposed by Barr and Asanović but specialized for control-flow traces [2]. The compression method proposed by Marathe et al. targets trace records consisting of one instruction address and one data address [15]. The sequence of instruction addresses is compressed with the *Sequitur* compression algorithm [18]. The sequence of data addresses is compressed by trying to identify regular accesses that fit the *power regular section descriptor* model (typically, accesses generated in loop nests). This method was compared with VPC3 on the SPEC2000 FP benchmarks, that are highly regular. It achieves very high compression ratios on traces whose compression ratio with VPC3 is already high. But it is less effective than VPC3 on traces that are harder to compress. Overall, traces compressed with VPC3 occupy less storage space [15].

There exists very few published studies on lossy trace compression. [1] Lossy trace compression is analogous to image compression : the basic idea is to remove some details while keeping the important information. The definition of what constitutes important information depends on what we want to do with the trace. Hence lossy trace compression requires that the user of the compressed trace be aware of how the trace was compressed. Filtering with a cache is a simple and effective way to compress an address trace [22, 29]. The original address trace is transformed into a cache-filtered trace consisting only of the addresses that miss in the cache. This is the kind of trace that our ATC compressor takes as input. More complicated forms of filtering have been proposed, like *blocking* [1], which exploits spatial locality. Phase analysis [11, 25] has been proposed primarily as a way to extrapolate a global metric (e.g., the IPC) from a few short simulation samples.

4 Lossless compression

This section describes the *bytesort* reversible transformation that makes lossless compression more effective.

[1] Analytical modeling may be viewed as an extreme form of compression. However, the memory behavior of applications is notoriously difficult to model analytically but on simple cases.

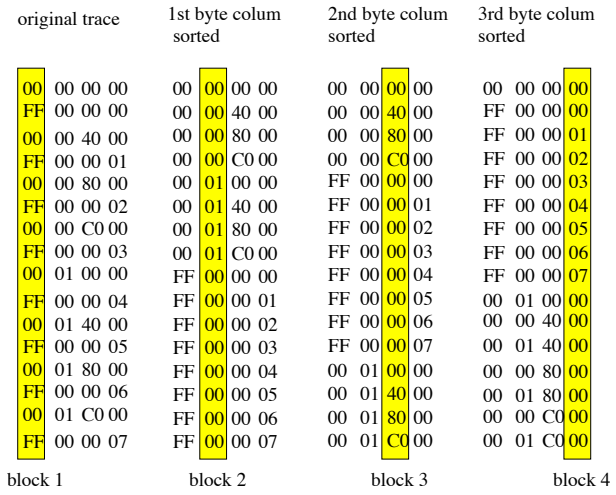

Figure 1: Example of bytesort transformation on a sequence of sixteen 32-bit addresses.

4.1 The bytesort transformation

Bytesort is a reversible transformation that takes as input a finite sequence of 64-bit addresses. It can be applied to any sequence of 64-bit values, but it is intended for compressing cache-filtered address traces, on which it achieves high compression ratios. Bytesort itself does not make the trace more compact, but it exposes existing regularities in a way that is easier to exploit by existing byte-level compressors like *gzip* of *bzip2*. Bytesort is based on the observation that an address sequence is easier to compress with a byte-level compressor if the bytes are **unshuffled**. That is, for a sequence of N 8-byte addresses, we output eight blocks of N bytes each : the first block consists of the first byte of each address in sequence order, the second block consists of the second byte of each address in sequence order, and so on. For example, consider the sequence of $N = 256$ addresses F200,F201,F202,...,F2FF (in hexadecimal). If these addresses are represented as 16-bit integers in big-endian format, the corresponding byte sequence is F2,00,F2,01,F2,02,...,F2,FF. There is some regularity coming from the fact that every other byte has value F2. But because of the interleaving, a byte-level compressor (e.g., *gzip*) may not be able to exploit the pattern. The byte sequence is easier to compress with a byte-level compressor if we unshuffle the bytes by outputting first the 256 high-order bytes and then the 256 low-order bytes (F2,F2,...,F2|00,01,...,FF). Byte-unshuffling requires to buffer the N addresses in memory before outputting the blocks. For long address traces, we use a finite size buffer of $B \times 8$ bytes, and we output the eight blocks every B addresses. Byte-unshuffling is very effective. It seems to be an obvious transformation for compressing address traces, yet we did not find any reference to it in the literature.

Bytesort uses byte-unshuffling, but it reorders the bytes in a way which is reversible and which exposes even more regularity than byte-unshuffling alone. For example, consider the sequence of 384 16-bit addresses :

```
typedef unsigned long long IntVal;
#define LL (sizeof(IntVal))
static int jb[256];

void unshuffle_bytes(struct atc *p, bblock *bb, IntVal
     a[])
{
  int i;
  unsigned char c;
  for (i=0; i<256; i++) {
    bb->hb[i] = 0;
  }
  for (i=0; i<p->nb; i++) {
    c = (a[i] >> ((LL-1)*8)) & 255;
    bb->b[i] = c;
    bb->hb[c]++; // compute the histogram
  }
}

void sort_bytes(struct atc *p, bblock *bb, IntVal a[],
     IntVal aa[])
{
  int i;
  unsigned char c;
  jb[0] = 0;
  for (i=1; i<256; i++) {
    jb[i] = jb[i-1] + bb->hb[i-1];
  }
  for (i=0; i<p->nb; i++) {
    c = (a[i] >> ((LL-1)*8)) & 255;
    aa[jb[c]] = a[i] << 8;
    jb[c]++;
  }
}

void output_bytesorted_blocks(struct atc *p)
{
  // from most significant to least significant bytes
  int i,j;
  unshuffle_bytes(p,&p->bb[0],p->a[0]);
  output_block(p,&p->bb[0]);
  p->x = 0;
  for (i=1; i<LL; i++) {
    sort_bytes(p,&p->bb[i-1],p->a[p->x],p->a[p->x^1]);
    p->x ^= 1;
    unshuffle_bytes(p,&p->bb[i],p->a[p->x]);
    output_block(p,&p->bb[i]);
  }
}
```

Figure 2: Excerpt from the ATC program (C language) implementing the bytesort transformation

F200,F201,A100,F202,F203,A101,F204,F205,A102,...,F2FE, F2FF,A17F. Byte unshuffling gives F2,F2,A1,F2,F2,A1,...,F2, F2,A1|00,01,00,02,03,01,...,FE,FF,7F. The first block of 192 bytes exhibits a periodic pattern that good byte-level compressors can exploit. However the second block looks more irregular. To expose more regularity in the second block, we sort the addresses according to their high-order byte before outputting the second block. After sorting according to the high-order byte, the address sequence becomes A100,A101,...,A17F,F200,F201,F202,...,F2FF and the unshuffled low-order byte block is 00,01,...,7F,00,01,02,...,FF. Overall, the transformed trace is F2,F2,A1,F2,F2,A1,...,F2,F2,A1|00,01,...,7F,00,01,02,...,FF. The second block is now more regular because the sequence 00,01,...,7F repeats twice. Moreover, the transformation is reversible because the sorting method is stable. By computing the histogram of byte values in the first block, we know that

trace	bz2	us	tcg	bs1	bs10
400.perlbench	3.95	4.41	3.09	3.06	2.61
401.bzip2	12.08	11.50	7.89	11.22	8.71
403.gcc	5.42	4.22	3.39	2.38	2.07
410.bwaves	13.01	1.57	4.56	0.20	0.17
429.mcf	15.56	10.68	3.17	7.81	5.07
433.milc	9.77	1.45	5.86	0.15	0.13
434.zeusmp	9.18	3.34	2.13	0.91	0.84
435.gromacs	7.61	7.94	5.06	8.23	5.94
444.namd	6.77	11.80	7.37	5.97	5.71
445.gobmk	7.01	8.57	5.35	5.20	4.44
447.dealII	3.88	2.20	1.57	1.29	1.18
450.soplex	10.08	4.81	3.14	2.33	1.87
453.povray	0.29	0.14	0.06	0.10	0.06
456.hmmer	7.30	5.10	1.68	1.30	1.19
458.sjeng	8.09	14.11	8.03	8.73	8.24
462.libquantum	4.72	0.45	0.64	0.06	0.05
464.h264ref	10.31	3.82	2.10	2.15	1.66
470.lbm	12.69	1.00	0.01	0.58	0.43
471.omnetpp	8.35	3.05	1.45	0.90	0.47
473.astar	10.82	8.53	7.54	4.22	4.11
482.sphinx3	16.02	5.01	2.33	2.48	1.69
483.xalancbmk	6.91	3.76	2.01	2.67	1.67
arith. mean	8.63	5.34	3.56	3.27	2.65

Table 1: Bits per address (smaller is better). Each trace represents 100 millions addresses. The second column is for bzip2 alone (*bz2*). The third column is for byte-unshuffling/bzip2 (*us*). The fourth column is for a TCgen compressor using 232 Mbytes of memory (*tcg*). The fifth column is for bytesort/bzip2 with a buffer of $B = 1$ million addresses (*bs1*). The sixth column is for bytesort/bzip2 with a buffer of $B = 10$ millions addresses (*bs10*).

the first 128 bytes in the second block are associated with the high-order byte value A1 and the last 256 bytes are associated with the high-order byte value F2. Figure 1 shows another example of applying the bytesort transformation on a sequence of 32-bit addresses. Because the sorting method is stable, the order between addresses having the same high-order byte is preserved after the high-order bytes have been sorted. Consequently, after successive sorts, addresses belonging to the same memory region are progressively grouped together. The access patterns inside a memory region are often very similar to those in some other regions. This is the kind of regularity that the bytesort transformation reveals.

Figure 2 shows an excerpt from the ATC program implementing the bytesort transformation. It should be noted that the time and space complexity of bytesort is linear with the buffer size B. The inverse transformation (not shown) is straightforward and, like the forward transformation, is linear in space and time.

4.2 Experimental evaluation

To evaluate the bytesort transformation, we generated some cache-filtered address traces using the Pin dynamic instrumentation tool [13, 20]. The instruction set architecture is x86-64. Our benchmarks are a subset of the SPEC CPU2006. We instrumented all basic blocks and all instructions accessing memory. The filtering is done with a level-1 instruction cache and a level-1 data cache. Both caches have a capacity of 32 Kbytes and are 4-way set-associative with a LRU (least-recently-used) replacement policy. Cache blocks are 64-byte long. The filtered address sequence contains missing instruction and data block addresses in sequential order. For the results in this section, we used only the first 100 millions filtered addresses from each benchmark. That is, the trace length is 100 millions and each uncompressed trace has a size of 800 millions bytes. For compressing traces, we use the *bzip2* compressor after having applied the bytesort transformation. We measure for each compressed trace the average number of bits per address (**BPA**). The smaller the BPA, the higher the compression ratio. We also provide the arithmetic mean of the BPA over all traces. The mean BPA, multiplied by the number of traces and by 100 millions, gives the total storage space occupied by the compressed traces. For bytesort, the BPA depends on the buffer size. A bigger buffer means that we work with bigger blocks, where long-term regularity can be exposed. Hence a bigger buffer yields a higher compression ratio. We measured the BPA with a buffer of $B = 1$ million addresses ("small bytesort") and with a buffer of $B = 10$ millions addresses ("big bytesort"). We compared bytesort with a VPC-like compressor/decompressor generated with TCgen [3, 31]. We have chosen a VPC-like compressor/decompressor that matches approximately the amount of memory used by the big bytesort. We used the following TCgen specification : *0-Bit Header; 64-Bit Field 1 = L1 = 1, L2 = 1048576: DFCM3[2], FCM3[3], FCM2[3], FCM1[3]; ID = Field 1; Compressor = 'bzip2 -c -z -9'; Decompressor = 'bzip2 -c -d';* . This compressor/decompressor uses 232 Mbytes of memory.

Table 1 gives the number of bits per address (BPA) for the compressed traces. The second column is for bzip2 alone, and the third column is for byte-shuffling combined with bzip2. In a majority of cases, byte-unshuffling improves compression significantly compared with using bzip2 alone. Overall, the traces on which we have applied byte-unshuffling occupy 38% less disk space than the traces compressed with bzip2 alone. The traces compressed with TCgen occupy 33% less disk space than the traces compressed with byte-unshuffling. The big bytesort gives the highest global compression ratio, saving 25% of disk space compared with TCgen, while using the same amount of memory. The small bytesort is less effective than the big bytesort. Still, it saves about 8% of disk space compared with TCgen, despite using 10 times less memory.

Decompression speed. We measured the total time to decompress the 22 traces of Table 1 (2.2 billions addresses). All measurement were done on a Dell Precision 360 workstation featuring 1 Gbyte of memory and a 3 Ghz Pentium 4 with a 1 Mbyte cache. The compressed traces were stored on the local

	TCgen	bytesort 1M	bytesort 10M
total time (sec)	1202	856	948
bzip2 contrib. (sec)	589	545	615
addr/second ($\times 10^6$)	1.83	2.57	2.32

Table 2: Decompression of the 22 traces on a Dell Precision 360 workstation.

disk [2] and the output of decompressors was redirected to the null device. We compiled all decompressors with "gcc -O3". Results are given in Table 2. Overall, the small bytesort and the big bytesort are respectively 40% and 26% faster than TCgen. We also measured the contribution of bzip2 to the decompression time and found that bzip2 contributes about 50% of the decompression time for TCgen and almost 65% for bytesort.

5 Lossy compression

Even with a very effective lossless compression method, some address traces are inherently difficult to compress, e.g., because the address sequence looks random. To make these traces significantly more compact, some form of lossy compression is necessary. For an image, lossy compression is acceptable provided the compressed image looks like the original image. This is achieved by removing the details that the eye cannot see or that the brain ignores. For an address sequence, the "eye" through which we look at the sequence is not clearly defined. It depends on what we want to do with the trace. For the purpose of architecture performance evaluation, it is important to preserve the sequence length (i.e., the number of addresses in the sequence) and the miss ratios for various cache configurations. For example, consider a loop accessing an array in a completely random fashion. The addresses appear to be drawn randomly from a set of N distinct addresses. Yet, the performance of this loop is quite stable (if the cache is able to hold $C \leq N$ tags, the hit ratio is approximately equal to C/N). If we partition the trace into intervals I_i consisting of L consecutive addresses each, and if L is much larger than N, all intervals look alike and a single interval can be used to characterize the whole trace. In other words, we replace the original trace $I_1, I_2, I_3, \cdots, I_k$ with a compressed trace $I_1, I_1, I_1, \cdots, I_1$. The new trace is more compact because we can represent the trace as a sequence of interval IDs. However, there are two questions to answer : how to detect that two intervals resemble each other, and how to choose the interval length L ? There are several possible ways to detect that two intervals look alike. For instance, we may use previously proposed phase analysis methods [11, 25]. However, these methods are generally implemented offline, and we are looking for a method that is simple enough to be done online so that the trace can be compressed with a single pass. Some online phase analysis methods have been proposed [19, 27]. We describe in Section 5.1 a method that is geared to our goals. The question of choosing a "good" value for L is actually more problematic, and we had to solve a particular problem, which is as follows. For online address trace compression, there is little choice but to choose an arbitrary value for L. However, on the previous example, a problem arises if we choose L too small. For example, let us assume we choose $L = N/2$. All intervals look similar to the first interval. But the compressed trace $I_1, I_1, I_1, \cdots, I_1$ is fundamentally different from the original trace : the original trace has N distinct addresses while the compressed trace has only about $0.4 \times N$. If we dimension the cache on the basis of what the compressed trace tells us, we will be misled in thinking that a cache with $N/2$ tags is sufficient to have a hit ratio close to 100%. In the remaining of this study, we refer to this problem as the *myopic interval* problem. To avoid the myopic interval problem, the interval length L should be taken as large as possible. But however large L, the problem may still occur. On the other hand, large intervals may yield poor compression ratios on traces that are unstable. We propose a method to lessen the myopic-interval problem. This method is based on *sorted byte-histograms*.

The basic idea to solve the myopic interval problem is that, if two intervals A and B have similar temporal structures but access different sets of memory addresses, we can use A to imitate B provided we transform the addresses of A so that they match those of B. Sorted byte-histograms, which we define in the next section, permit detecting when two intervals A and B are likely to have similar temporal structures and provide a way to transform addresses of A so as to imitate the spatiotemporal structure of B.

5.1 Sorted byte-histograms

Let us consider an interval of L consecutive 64-bit addresses $A(k)$ with $k \in [1, L]$. Each address $A(k)$ is coded with eight bytes $b[j](k) \in [0, 255]$:

$$A(k) = \sum_{j=0}^{7} b[j](k) \times 2^{8 \times j}$$

We define the *byte-histograms* as follows :

$$\text{for } j \in [0, 7] \text{ and } i \in [0, 255],\ h[j](i) = \sum_{k=1}^{L} \delta_i(b[j](k))$$

where

$$\delta_i(x) = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{if } x \neq i \end{cases}$$

In other words, the byte-histogram value $h[j](i)$ is the number of addresses in the interval whose byte of order j is equal to i (note that $\sum_{i=0}^{255} h[j](i) = L$). For a given j, the *sorted byte-histogram* $h'[j]$ is obtained from $h[j]$ by sorting the 256 values $h[j](i)$ in decreasing order. That is,

$$h'[j](i) = h[j](p[j](i)) \quad (1)$$

[2] Disk accesses contribute little to the total decompression time.

where $p[j]$ is the unique permutation of $[0, 255]$ such that $\forall i_1 < i_2$, $h'[j](i_1) \geq h'[j](i_2)$ and $\forall i_1 < i_2$, $h[j](i_1) = h[j](i_2) \Rightarrow p[j](i_1) < p[j](i_2)$. In practice, the permutation $p[j]$ is obtained by a stable sort algorithm (e.g., merge-sort). In particular, $p[j](0)$ is the most frequent byte value of order j. We define the distance between two intervals from the same trace as follows. Let A and B be two intervals of length L. Interval A is characterized by 8 sorted byte-histograms $h'_A[j]$ and interval B is characterized by 8 sorted byte-histograms $h'_B[j]$. We define the distance $D(A, B)$ between intervals A and B as

$$D(A, B) = \max_{j \in [0,7]} d(h'_A[j], h'_B[j]) \quad (2)$$

where the distance $d(h_A, h_B)$ between two histograms h_A and h_B is defined as

$$d(h_A, h_B) = \frac{1}{L} \sum_{i=0}^{255} |h_A(i) - h_B(i)|$$

Note that we have $0 \leq d(h_A, h_B) \leq 2$ and $0 \leq D(A, B) \leq 2$.

We say that intervals A and B *look like each other* is the distance $D(A, B)$ between them is less than a certain fixed threshold ϵ. To solve the myopic interval problem, when we find that interval B looks like a previous interval A, we record the information that A can be used to imitate B, but we also record a *byte translation* $t[j]$ which is defined as follows :

$$\forall i \in [0, 255],\ t[j](p_A[j](i)) = p_B[j](i)$$

where $p_A[j]$ and $p_B[j]$ are the permutations obtained by sorting the histograms (cf. equation (1)). Note that $t[j]$ itself is a permutation of $[0, 255]$. When using A to imitate B, we replace each byte $b[j](k)$ of A with the byte $t[j](b[j](k))$, but only if the distance $d(h_A[j], h_B[j])$ between the *non-sorted* histograms $h_A[j]$ and $h_B[j]$ is greater than the threshold ϵ. That is, we translate bytes only for values of j for which this is necessary. It should be noted that the most frequent byte of order j in interval A is replaced with the most frequent byte of order j in interval B. As an example, consider the interval A of length $L = 256$ consisting of addresses $F200, F201, F202, \cdots, F2FF$ and the interval B consisting of addresses $F300, F301, F302, \cdots, F3FF$. We have

$$\begin{aligned} p_A[1] &= (F2, 0, 1, 2, \cdots, F1, F3, F4, \cdots, FF) \\ p_B[1] &= (F3, 0, 1, 2, \cdots, F1, F2, F4, \cdots, FF) \\ t[1] &= (0, 1, 2, \cdots, F1, F3, F2, F4, \cdots, FF) \end{aligned}$$

As $h'_A[1] = h'_B[1] = (256, 0, 0, \cdots, 0)$, we have $d(h'_A[1], h'_B[1]) = 0$. As for the least-significant byte, we have $h_A[0](i) = h_B[0](i) = 1$ hence $d(h'_A[0], h'_B[0]) = 0$. Consequently, $D(A, B) = 0$ and we find that B looks like A. When using A to imitate B, we leave the bytes of order 0 unchanged because $d(h_A[0], h_B[0]) = 0$. As for bytes of order 1, we translate them because $d(h_A[1], h_B[1]) = 2$. That is, we replace byte $F2$ with $t[1](F2) = F3$. Notice that, on this example, the imitation is perfect. In general, this is not always the case.

trace	lossless	lossy	trace	lossless	lossy
400	5.08	0.70	401	11.37	0.81
403	1.39	1.09	410	0.19	0.04
429	5.57	1.02	433	0.16	0.06
434	0.98	0.34	435	8.27	1.41
444	6.14	2.26	445	5.18	2.17
447	1.51	1.30	450	4.20	0.97
453	0.22	0.02	456	1.52	0.08
458	9.45	1.08	462	0.03	0.004
464	2.17	0.26	470	0.64	0.01
471	1.08	0.37	473	3.70	0.86
482	2.54	0.08	483	3.07	0.97
			arith. mean	3.39	0.72

Table 3: Bits per address for lossless vs. lossy compression. Each trace represents 1 billion addresses. The lossless compressor is byte-sort with a buffer size of 1 million addresses. The interval length for lossy compression is $L = 10$ millions addresses, the threshold is $\epsilon = 0.1$.

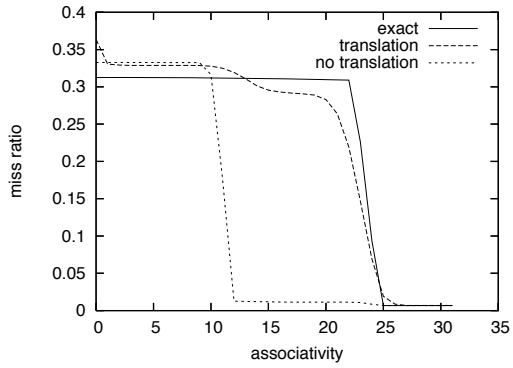

Figure 4: Impact of disabling byte translation on trace 470 for 256k cache sets.

Remark. The method relies on the assumption that if two intervals from the *same* trace have similar sorted byte-histograms, they have the same temporal structure. Of course, there is no guarantee that this is necessarily the case. Nevertheless, our experimental results in Section 5.3 show that, in practice, this is true with a high probability. As permutations $t[j]$ map each unique address of interval A to a unique address, the temporal structure of A is preserved by the byte translation. Moreover, the translated A has a spatiotemporal resemblance with B, as it has similar sorted byte-histograms. Nevertheless, the process of replacing B with a transformed A is approximate, and some distortion may be introduced. Even when there exists an exact one-to-one relation between addresses of A and those of B, byte translation does not necessarily transform A into an exact copy of B. Using a stable sorting algorithm and limiting the byte translations to values of j for which this is necessary is a way to minimize distortions.

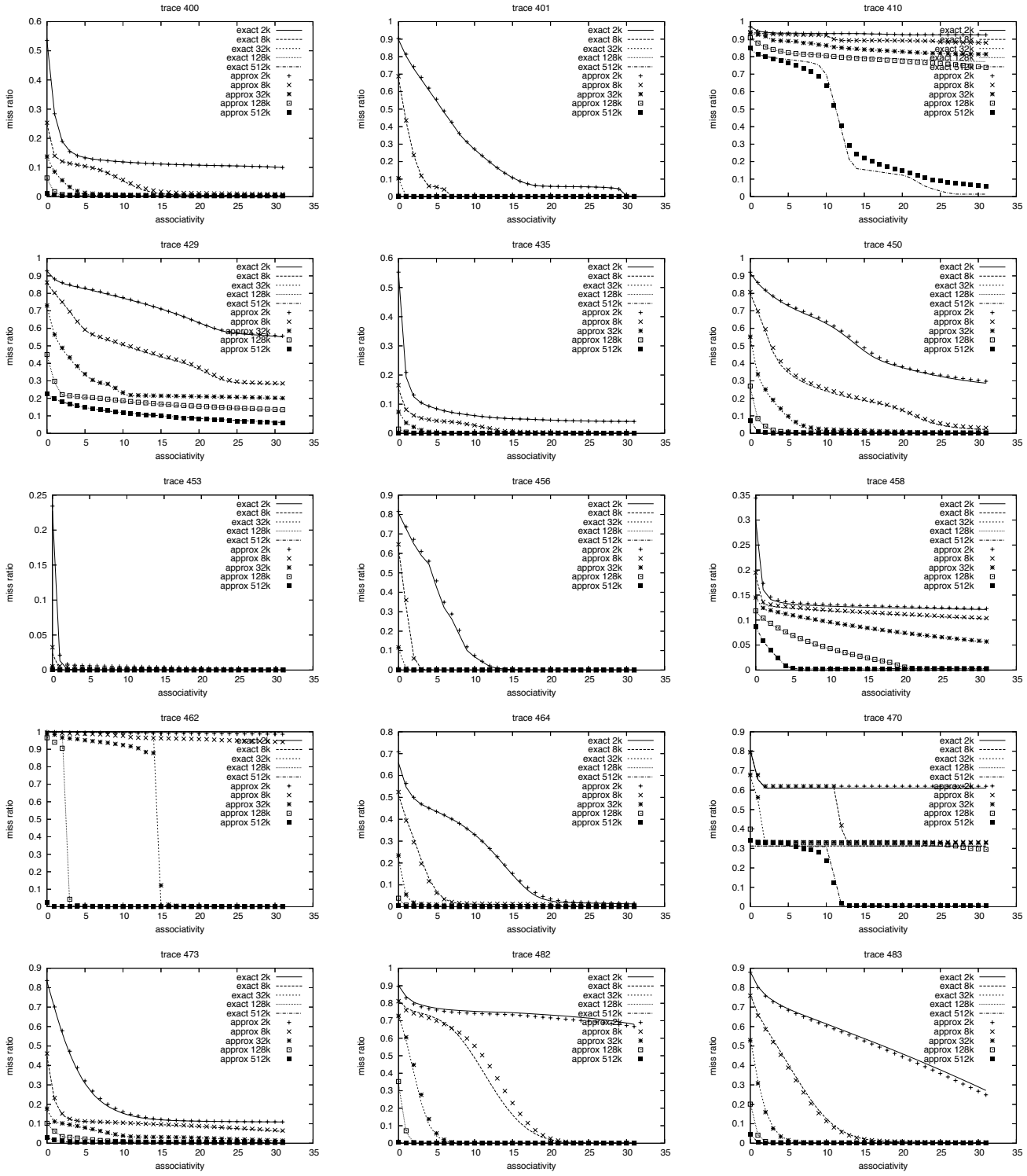

Figure 3: Cache miss ratio for exact and approximate traces as a function of the cache associativity and for a number of cache sets between 2k and 512k (LRU replacement policy).

5.2 Lossy compression scheme

Our lossy compression scheme uses online phase classification [19, 27]. The compressed trace consists of a set of *chunks* and an *interval trace*. A chunk is an interval of the original trace that we compress with a lossless compression scheme. In this study, all chunks are compressed with the bytesort method described in Section 4, using a buffer size of 1 million addresses. The interval trace is compressed with bzip2. Each time we create a chunk, we record an entry for it in a *histogram table* in memory, where we store the histograms for that chunk. When the table is full, we evict the entry belonging to the oldest chunk. We always create a chunk for the first interval in a trace. At the end of the first interval, we compute the histograms for the interval and we store them in the histogram table. Then we compute the histograms for the second interval and we compute the distance between intervals 1 and 2 using formula (2). If the distance is greater than the threshold, we create a chunk for the second interval and we store the histograms of the second interval in the histogram table. Otherwise, if the distance is less than the threshold, we do not create a chunk. We only record in the interval trace the fact that interval 1 can be used to imitate interval 2, along with the byte translations $t[j]$ (translations are completely described with 8×256 bytes). Then we compute the distance between the third interval and the previous chunks. And so on. When several chunks match the current interval, we imitate the interval using the chunk having the smallest distance with the interval. The fewer chunks are created, the more compact the trace. Most traces have a stable behavior, and a relatively small number of chunks is often sufficient to represent the whole trace. For these traces, the compression ratio increases with time : chunks are created for the first intervals, then chunks are reused when there are enough of them to imitate most subsequent intervals. The choice of the threshold ϵ has an impact on the compression ratio and the compression quality. If ϵ is too small, we obtain a low compression ratio. If ϵ it is too high, the compressed trace may not accurately reflect the original trace. We found experimentally that $\epsilon = 0.1$ provides high compression ratios while preserving the memory locality information contained in the original trace.

5.3 Accuracy of lossy vs. lossless compression

The experimental set-up is the same as described in Section 4.2. Each exact trace contains 1 billion addresses that were compressed with bytesort. Approximate traces are generated from the exact traces. Then we use exact and approximate traces to simulate a set of cache configurations, and we compare the cache miss ratios. We used the *Cheetah* cache simulator [30] to simulate a set-associative cache, varying the number of cache sets and the associativity. Table 3 gives the bits per address for lossy vs. lossless compression. The interval length for lossy compression is 10 millions addresses (a trace represents 100 intervals) and the threshold is $\epsilon = 0.1$. The compression ratio achieved with lossy compression depends on

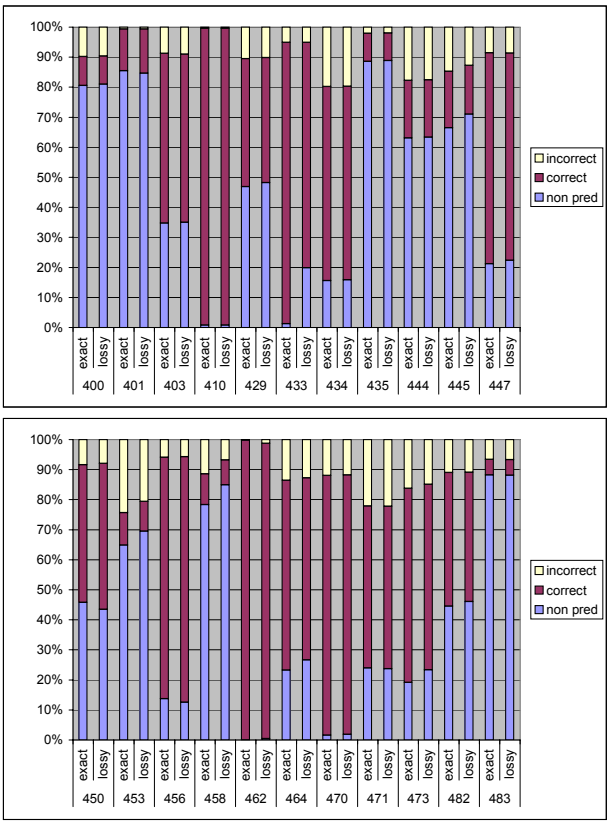

Figure 5: Simulation of a C/DC address predictor. Percentage of non-predicted, correctly predicted and mispredicted addresses for the exact traces and for the lossy-compressed ones.

the trace length and on the trace characteristics. For example, the compression ratio on traces 403.gcc and 447.dealII is small because these traces are unstable, so the distance between intervals is generally greater than the threshold. Figure 3 shows the cache miss ratio as a function of the cache associativity and for different number of cache sets. The replacement policy is LRU. In practice, the miss ratio of the approximate trace is generally very close to the exact value. Even when there is some distortion (e.g., traces 410 and 482), the shape of the miss ratio curves is preserved. To stress the importance of byte translation, Figure 4 shows the impact of disabling it on trace 470. As can be seen, disabling byte translation introduces a large distortion. The cache size that is necessary to remove capacity misses looks twice smaller with the approximate trace than it is in reality.

We also simulated an address predictor based on the C/DC prefetcher [17]. Our predictor assumes 64-Kbyte CZones, a 256-entry index table, a 256-entry global history buffer, and a 2-delta correlation key. For each address, the predictor tries to predict the next address in the same CZone. If there is no match for the correlation key, the next address in the CZone will not be predicted. Otherwise, the predicted address is stored in the index-table entry and will be compared with the next address in that CZone. Figure 5 shows the percentage of non-predicted,

```c
#include <stdio.h>
#include "atc.h"

struct atc cp;

int main(int argc, char **argv)
{
  unsigned long long x;
  char *dirname = argv[1];

  atc_open(&cp,'k',dirname,"bz2","bzip2 -c");

  while (fread(&x,sizeof(x),1,stdin)) {
    atc_code(&cp,x);
  }

  atc_close(&cp);
}
```

Figure 6: Example of C program that takes 64-bit values from the standard input and generates an ATC-compressed file whose name is provided as an argument.

```c
#include <stdio.h>
#include "atc.h"

struct atc cp;

int main(int argc, char **argv)
{
  unsigned long long x;
  char *dirname = argv[1];

  atc_open(&cp,'d',dirname,"bz2","bunzip2 -c");

  while (atc_decode(&cp,&x)) {
    fwrite(&x,sizeof(x),1,stdout);
  }

  atc_close(&cp);
}
```

Figure 7: Example of C program that takes an ATC-compressed file as an argument and writes the uncompressed trace on the standard output.

correctly predicted and mispredicted addresses for the exact traces and for the lossy-compressed ones. Lossy compression introduces a little distortion (e.g., trace 433), but overall the lossy-compressed traces "look" like the exact ones.

Finally, we ran each benchmark to completion. We obtained 22 compressed traces representing a total of about 500 billions addresses. Without compression, we would need 4 terabytes of disk space to store the traces. With lossy compression, the 22 traces take only 9 gigabytes of disk space. This means an average of 0.14 bits per address. [3]

6 The ATC compressor

The ATC compressor is written in C. It consists basically of 4 functions, atc_open, atc_close, atc_code and atc_decode. Their use is illustrated in the example programs of Figures 6 and 7. In the program of Figure 6, the atc_open function is called with argument *'k'* which means lossy compression. For lossless compression, the argument would be *'c'*. The atc_open function creates a directory, whose name is given by the argument $dirname$, in which the compressed trace will be stored. The argument $bz2$ means that we want compressed chunks to have the suffix *.bz2* and the last argument is the command that is used to compress bytesorted chunks. On this example, we use *bzip2*, but we could use another compressor, like *gzip*. After the compressed trace has been opened with atc_open, the compression is done by calling atc_code for each 64-bit input value. An example program for decompressing traces is shown in Figure 7. Here, the atc_open function is called with the argument *'d'* (decompression). The last argument is the command for decompressing bytesorted chunks. Figure 8 shows the result of using these two programs on a sequence of 100 millions random 64-bit values. We obtain a compression ratio of 10 on this example because ATC finds that all the intervals (length $L = 10$ millions) look like the first one. So a single chunk is created for the first interval, but the 9 subsequent intervals are regenerated from the first chunk and from the byte translation information.

```
% cat /dev/urandom | head -c 800000000 |
bin2atc foobar

% du -b foobar/*
80355941        foobar/1.bz2
853     foobar/INFO.bz2

% atc2bin foobar | wc -c
800000000
```

Figure 8: Example : *bin2atc* is the program of Figure 6 and *atc2bin* is the program of Figure 7. On this example, 100M random 64-bit values are compressed (this is lossy compression) and stored in the directory *foobar*. Only the first chunk is stored (*1.bz2*). This chunk represents the first 10 millions 64-bit values, "compressed" with bytesort and bzip2. The 9 subsequent intervals are regenerated from the first chunk and from the byte translation information stored in *INFO.bz2*.

The ATC compressor may be applied to any sequence of 64-bit values and might be used for compressing traces other than cache-filtered addresses. However, while the ATC lossless compression mode is completely safe, it does not necessarily yield the best compression ratios. For instance, applying ATC lossless compression to a control-flow trace does not yield compression ratios as high as those obtained with predictor-based compressors [2]. As for ATC lossy compression mode, it is intended for cache-filtered address traces only and should not be used without questioning its applicability to the situation.

7 Conclusion

We have proposed an effective compressor for traces consisting of cache-filtered addresses, that are useful for capturing the macroscopic behavior of applications. Our two main contributions are the introduction of *bytesort*, a reversible trans-

[3] We recall that the value of 0.72 bits per address given in Table 3 was for traces of 1 billion addresses only.

formation that makes lossless compression more effective, and a lossy compression method based on sorted byte-histograms. We have combined the two methods in a software called ATC. We have shown that, on cache-filtered address traces, byte-sort yields higher compression ratios than TCgen, one of the most effective trace compressors existing. We have shown that our lossy compression scheme permits obtaining traces that are up to several orders of magnitude more compact, while keeping the important memory locality information contained in the original trace. With the ATC compressor, it is possible to store compact traces representing hours of real execution.

Acknowledgments

The author would like to thank André Seznec, Erven Rohou, Yiannakis Sazeides and the anonymous reviewers for their helpful comments.

References

[1] A. Agarwal and M. Huffman. Blocking : exploiting spatial locality for trace compaction. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990.

[2] K.C. Barr and K. Asanović. Branch trace compression for snapshot-based simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2006.

[3] M. Burtscher. TCgen 2.0 : a tool to automatically generate lossless trace compressors. *ACM SIGARCH Computer Architecture News*, 34(3), June 2006.

[4] M. Burtscher, Ilya Ganusov, S.J. Jackson, J. Ke, P. Ratanaworabhan, and N.B. Sam. The VPC trace-compression algorithm. *IEEE Transactions on Computers*, 54(11):1329–1344, November 2005.

[5] M. Burtscher and N.B. Sam. Automatic generation of high-performance trace compressors. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[6] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng. TPTS : a novel framework for very fast manycore processor architecture simulation. In *Proceedings of the International Conference on Parallel Processing*, 2008.

[7] L. Eeckhout, S. Nussbaum, J.E. Smith, and K. De Bosschere. Statistical simulation : adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, September 2003.

[8] E.E. Johnson and J. Ha. PDATS : lossless address trace compression for reducing file size and access time. In *Proceedings of the IEEE International Phoenix Conference on Computers and Communications*, 1994.

[9] S. Kanaujia, I.E. Papazian, J. Chamberlain, and J. Baxter. FastMP : a multi-core simulation methodology. In *2nd Annual Workshop on Modeling, Benchmarking and Simulation*, 2006.

[10] T. Karkhanis and J.E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st International Symposium on Computer Architecture*, 2004.

[11] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations : a preliminary application to the data stream. In *IEEE 3rd Annual Workshop on Workload Characterization*, 2000.

[12] B.C. Lee, J. Collins, H. Wang, and D. Brooks. CPR : composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st International Symposium on Microarchitecture*, 2008.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, 2005.

[14] Y. Luo and L.K. John. Locality-based online trace compression. *IEEE Transactions on Computers*, 53(6):723–731, June 2004.

[15] J. Marathe, F. Mueller, T. Mohan, S.A. McKee, B.R. De Supinski, and A. Yoo. METRIC : memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems*, 29(2), April 2007.

[16] A. Milenković and M. Milenković. An efficient single-pass trace compression technique utilizing instruction streams. *ACM Transactions on Modeling and Computer Simulation*, 17(1), January 2007.

[17] K.J. Nesbit, A.S. Dhodapkar, and J.E. Smith. AC/DC : an adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[18] C.G. Nevill-Manning and I.H. Witten. Identifying hierarchical structure in sequences : a linear-time algorithm. *Journal of Artificial Intelligence Research*, 7, 1997.

[19] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2005.

[20] Pin. http://rogue.colorado.edu/Pin.

[21] A.R. Pleszkun. Techniques for compressing program address traces. In *Proceedings of the International Symposium on Microarchitecture*, 1994.

[22] T. Puzak. *Analysis of cache replacement algorithms*. PhD thesis, University of Massachusetts, 1985.

[23] A.D. Samples. Mache : no-loss trace compaction. In *Proceedings of the ACM SIGMETRICS Conference on Measurement, and Modeling of Computer Systems*, 1989.

[24] C.E. Shannon. Prediction and entropy of printed english. *Bell System Technical Journal*, 30:50–64, 1951.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[26] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, November 2003.

[27] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[28] SimPoint. http://www-cse.ucsd.edu/ calder/simpoint.

[29] A.J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, SE-3(1):94–101, January 1977.

[30] R.A. Sugumar and S.G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[31] TCgen. http://www.csl.cornell.edu/∼burtscher/research/TCgen/.

[32] R.A. Uhlig and T.N. Mudge. Trace-driven memory simulation : a survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.