Instruction Trace Compression for Rapid Instruction Cache Simulation

Andhi Janapsatya
Computer Science and Engineering,
The University of New South Wales
Sydney, NSW 2052, Australia
andhij @ cse.unsw.edu.au

Aleksandar Ignjatovic and Sri Parameswaran
School of CSE and NICTA,
University of New South Wales
Sydney, NSW 2052, Australia
ignjat, sridevan @ cse.unsw.edu.au

Joerg Henkel
Dept. of Computer Science
Karlsruhe University
Zirkel 2, D-76131 Karlsruhe, Germany
henkel @ informatik.uni-karlsruhe.de

ABSTRACT

Modern Application Specific Instruction Set Processors (ASIPs) have customizable caches, where the size, associativity and line size can all be customized to suit a particular application. To find the best cache size suited for a particular embedded system, the application(s) is/are executed, traces obtained, and caches simulated. Typically, program trace files can range from a few megabytes to several gigabytes. Simulation of cache performance using large program trace files is a time consuming process. In this paper, a novel instruction cache simulation methodology that can operate directly on a compressed program trace file without the need for decompression is presented. This feature allowed our simulation methodology to have an average speed up of 9.67 times compared to the existing state of the art tool (Dinero IV cache simulator), for a range of applications from the Mediabench suite.

1. Introduction

In the last decade, the emergence of ASIPs (such as Xtensa processor from Tensilica [1]) has provided a superior alternative compared to simply designing an off-the-shelf processor-based embedded systems. ASIP allows the customization of functional units for improved performance and lower energy consumption when compared to off-the-shelf general purpose processor based systems.

The use of ASIPs allows the micro-architecture feature within the Processor (such as the functional units, cache size, data width size, etc.), to be customized during the design process. The choice of cache size or functional units are part of the ASIP design process and can affect the performance, energy consumption, and the overall footprint of the system.

The availability of customizable cache memory in a system provides a designer the option of specifying an optimal cache memory configuration to match the application requirements in order to

- minimize cache energy cost per access,
- minimize access time per cache access,
- minimize the number of cache misses and reduce the number of off-chip memory accesses,
- or any combination of the above.

In a typical design process, cache simulation is performed for various cache configurations to determine the optimal cache parameters for the given program trace. However, a typical program trace can be several tens of Gigabytes long. For a 1GHz processor, a one second snapshot stored as a program trace might translate to a ten gigabyte file. This becomes even more critical, as tens of processors are included in a single embedded system. Although disk space may no longer be an issue in the near future with the emergence of large hard drives (up to 750GB [2]), getting large amounts of data to and from the disk is still a time consuming process.

One method to alleviate the large cost of processing a program trace file is to compress the trace file. Compression methods have been proposed in the past to reduce the size of these program trace files. Although compression allows the reduction of the program trace file size, the required intermediate memory during decompression and program trace analysis is still large, requiring enormous number of reads and writes to disks.

In this paper, we present a novel method to allow cache simulation to be performed from a compressed program trace file such that only 'partial decompression' is necessary. Our experimental results showed that our cache simulation methodology is on average 9.67 times faster compared to the existing state-of-the-art cache simulation tool. To compress the program trace file, we developed a compression methodology that allows for random access decompression [3]. Random access decompression is a term used to describe a compression methodology that allows the decompression to start at any point in the compressed file. The random access decompression feature allows our cache simulation methodology to operate on the compressed program trace file and provide the opportunity to parallelize the cache simulation methodology.

Compared to existing data compression tools (such as gzip[4]), our compression rate is anywhere from 2 to 10 times worse when compared to gzip. This is because the compression methodology is not designed to achieve maximal compression rate, instead it is designed for minimal processing cost. If a high compression rate is required, a post-compression step with LZ compression can be used to further improve the compression rate.

Besides cache analysis, there are many reasons for wanting to analyze a program trace; such as estimating the energy of a system, evaluating system performance, etc. Depending on the type of analysis to be performed, different compression algorithms can be developed to overcome the bottleneck posed by the large amount of time needed to read and write from disks.

The rest of this paper is structured as follows. Section 2 presents existing trace compression algorithms and cache simulation tools; Section 3 outlines our methodology; Section 4 describes the trace compression algorithm; Section 5 provides the cache simulation algorithm using the compressed program trace as its input; Section 6 describes the experimental setup and presents the results; Section 7 concludes this paper.

2. Related Work

There are two types of compression algorithms; lossless and lossy. Lossless compression allows the exact original source to be reproduced from the compressed format. In comparison, lossy compression generally results in higher compression ratio by removing some information from the data, thus, it is not possible to recreate the original file. The work presented in this paper is a lossless compression algorithm.

Existing data compression method such as the LZ77 [4] introduced by Liv and Zempel in 1977, is a well-known compression method used in gzip. The algorithm reads a file as a stream and uses previous

978-3-9810801-2-4/DATE07 © 2007 EDAA

texts in the stream to encode the incoming texts in the stream. In 1978, Liv and Zempel presented another version of their algorithm, known as LZ78 [5]. In 1983, Welch [6] introduces an improvement to the LZ78 to limit the growth of the dictionary file, known as the LZW algorithm.

In 1994, Pleszkun [7] presented a two pass algorithm for compressing a program trace file. The first pass is a pre-processing step to identify the dynamic basic blocks, procedure calls, etc. The trace is encoded by specifying the basic block and its successor.

Johnson et al. in [8] and [9] presented an offset-encoding compression scheme known as PDATS. Each address reference in the trace are to be encoded as its offset from the previous address reference. In addition, a run-length coding is used to encode sequence of addresses with the same offset. Their experimental results show an average compression ratio of seven for large traces and they achieved a speedup factor of 10 when executing Dinero with PDATS traces.

In 1997, Nevill-Manning and Witten [10] introduced the hierarchical compression method known as 'SEQUITUR'. It constructs grammar based on the two rules: no pair of adjacent symbols appears more than once in the grammar, and every rule is used more than once. SEQUITUR can be applied to any type of information streams. The processing time of SEQUITUR to process one symbol is $O(\sqrt{n})$, where n is the number of input symbols encountered. Our work differs from SEQUITUR as we allow symbols to appear more than once in a single grammar; this is because the goal of our compression method is to allow minimal processing cost of the compressed program trace.

In 2003, Kaplan [11] presented two algorithms for trace compression; 'Safely Allowed Drop' (SAD) and 'Optimal LRU Reduction' (OLR). These algorithms are lossy in their compression methodology and are designed for simulating a virtual memory. The methodology removes references from a trace that do no affect the order of fetches for a virtual memory.

Milenkovic and Milenkovic [12] presented a compression methodology called 'Stream-Based Compression' (SBC). SBC performs compression by replacing each address stream by its index in the stream table. Address streams are identified according to basic blocks in the program.

In 2004, Burtscher in [13] introduced VPC3 tool. VPC3 runs in a single pass in linear time over the trace data. The trace data in VPC3 contains many attributes in addition to the memory address. It encodes the trace data using value predictors.

Luo and John [14] presented a 'Locality Based Trace Compression' (LBTC). The methodology employs two techniques; the first technique is offset encoding of the memory references, and the second technique is to statically encode the attributes associated with a memory location through the assumption that most attributes are static and do not change frequently from one dynamic access to another. Their experimental results showed that they improve the compression rate by 2x over the PDATS method.

Zhang and Gupta in [15] and [16] presented a unified representation of profiles called 'Whole Execution Trace' (WET). WET is constructed by labeling a static program representation with profile information. Their experimental results showed that their method achieved an average compression ratio of 41.

2.1 Existing Work on Cache Analysis

An existing tool called Dinero IV [18] allows for exact cache simulation. Dinero IV simulates a single processor architecture to estimate the number of cache misses given a cache configuration.

Ghosh et al. [19] introduced the cache miss equation to estimate cache misses to guide memory optimization methods. Hill and Smith [20] presented two methods for simultaneous simulations of multiple cache configurations, namely the forest simulation and the all-associative simulation. Janapsatya et al. [21] designed a novel data structure to reduce the time needed to estimate the cache misses exactly of multiple cache configurations simultaneously. Ghosh et al. [22] presented a heuristic method to estimate cache configurations that can predict the worst case cache miss rates.

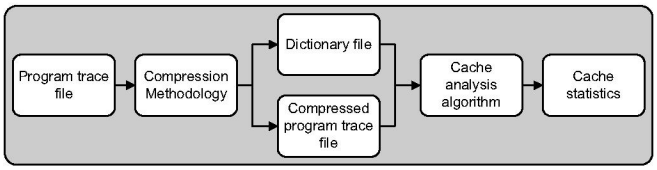

Figure 1: Compression and cache analysis methodology

2.2 Our Contribution

The motivation for a trace compression methodology is to reduce the time required to analyze large program trace files. Existing work on trace compression exclusively aims for maximum compression ratio which does not alleviate the problem of performing analysis on large trace files. To use Dinero or other tools for cache analysis, an initial decompression of the compressed trace file is still needed before analysis can be performed.

In this paper, a lossless trace compression methodology to compress a program trace is implemented. To the best of our knowledge, this is the first time a compression methodology that allows the processing of the compressed program trace without the need for decompression is presented.

The compression methodology allows random-access to the compressed trace file to allow direct access at any point in the compressed file either for decompression or to allow analysis of the compressed trace file without the need of decompression. In summary, this paper presents the following contributions:

- Novel stream compression methodology that is designed for minimal processing cost of cache simulation analysis on the compressed stream.
- A novel cache analysis method that allows the analysis of a compressed trace file without the need for decompression.

Limitation

This work is limited to instruction trace only. Data trace compressions have been shown previously in [7], [8], [12], [13], [14], [15] (It should be noted that none of the existing data compression methods support random access decompression).

3. Methodology

The methodology flow of the compression and cache analysis process are as shown in Figure 1. Inputs to the process is a program trace file. The compression process outputs a dictionary file and a compressed program trace file. Cache analysis uses the dictionary file and the compressed program trace file as its input; and it outputs the cache statistics. As mentioned, the cache analysis process is performed directly on the compressed trace, without decompression.

3.1 Format of the Trace

We view the program trace as a bit-stream with no knowledge of the program functionality. We do not know the number of instructions in the program trace, nor how instructions interact with each other, and we have no knowledge of which instructions belong to which process or function. Nevertheless, our method can achieve high compression rates.

The program trace file contains a trace of memory references. In comparison to other existing trace format, our trace is identical to the Dinero input format. An example of the program trace are shown in Figure 3(a), 3(b), and 3(c). The numbers on the left column are the sequence number and the numbers on the right column are the

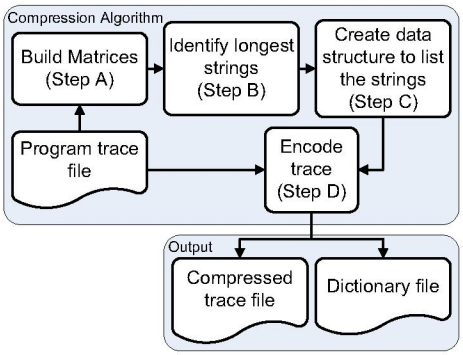

Figure 2: Compression algorithm

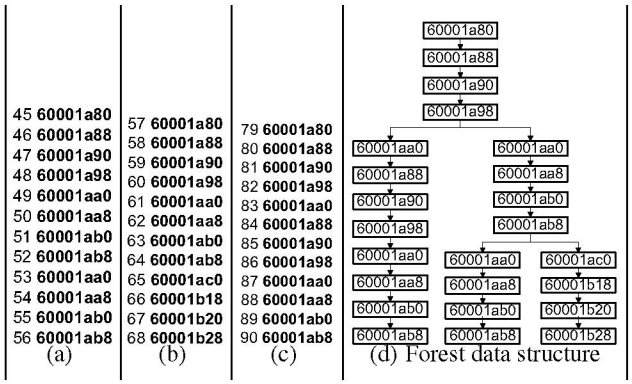

Figure 3: Program trace and data structure examples

memory references. We assume all memory addresses are cacheable. Each memory reference does not contain attributes, such as data dependencies, etc. If required, a data dependencies graph can be built using the information available in the trace. At the current moment, our tool only looks at trace of instruction memory references.

4. Compression Algorithm

The compression algorithm is designed to compress the trace file but also allows the compressed trace to be analyzed without the need for decompression. The compression algorithm is designed as a random-access compression algorithm to allow analysis and decompression to start from anywhere in the compressed trace file.

Figure 2 shows the compression algorithm. For ease of explanation, the boxes shown in Figure 2 are labeled Step A, Step B, Step C, and Step D. In step A and Step B, the trace is analyzed and the longest repetitive strings are identified. These repetitive strings are then stored within a forest data structure, this is step C. Step D will use the forest data structure to encode strings of instructions as read from the program trace file.

Step A

The purpose of step A and Step B is to identify patterns that exist in a program trace file. In our algorithm, we identify two occurring patterns. First pattern is the consecutive repetition of the same instructions string, for example 'abcabcabcabcabc'. The second pattern is the reoccurrence of a string repetitively but not consecutively, for example the string 'abc' in the text '**abc**qwe**abc**rtyu**abc**io **abc**k**abc**fg' occurs repetitively but not consecutively and has a non-uniform stride between occurrences.

To identify the pattern within the instruction trace file, a matrix is build to assist the process. Figure 3(a), 3(b), 3(c) show examples of a program trace. Figure 5 shows the matrix built from the program trace shown in Figure 3.

The numbers shown on the bottom of Figure 5 represent the instruction sequence, and the numbers on the left (1st column) show

```
\\ Creating the matrix
\\ n = total number instructions in the window
\\ h = the height of the matrix, equal to the maximal distance to look for
\\      repeated instructions
for i = 1 to n
    for j = 1 to h
        if (W[i] == W[i+j])
            M[i,j] = 1;
        else
            M[i,j] = 0;
```

Figure 4: Building matrix M

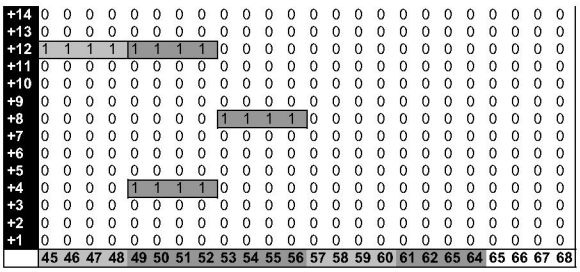

Figure 5: Matrix M

the distance. Since a program trace can be very long, the matrix of width n covers only a substring of the trace consisting of n consecutive instructions. Even though this is a small substring of the whole trace, if n is reasonably large (in our implementation of the algorithm $n = 4000$), each loop in the program should be identifiable within a single window. To cover the whole program trace, a sliding window is used. To prevent some loops being lost in between windows boundary, the sliding window mechanism will advance the window by $n/2$ instructions at a time.

The matrix M of size $n \times h$ is built as follows. We take n consecutive instructions as one window $W(i)$, $1 \leq i \leq n$. The entry in i^{th} column and j^{th} row of our matrix M is equal to one just in case the $W(i) = W(i+j)$, $1 \leq i \leq n$, $1 \leq j \leq h$; otherwise it is equal to zero.

In the matrix shown in Figure 5, the grey shading highlighting the string of '1s' found in the matrix indicates the repeated occurrence of the same instruction. These strings are identified in Step A and analyzed in Step B (shown below) to ensure no repeated sequences are identified.

Figure 4 shows the algorithm for identifying the patterns in the program trace file by utilizing the matrix shown in Figure 5. We now construct another matrix S of the size $(n-h) \times h$, as shown in Figure 6. The entry $S[i,j]$ is equal to the number of consecutive '1s', in row j and columns k of Matrix M such that $k \geq i$ (see Figure 8).

Step B

We now identify strings that are potentially useful for compression as follows.

The heuristics behind the value of Gain$(W[i..i+j])$ of the string $W[i..i+j]$ is to assess how useful the string is for the purpose of compression. We want to use the strings that are both long and frequent, i.e., those that have high gain. In the algorithm shown in Figure 7, the first "if" clause ($S[i,j] \geq j$) detects strings that repeat consecutively,

```
for j = 1 to h
    for i = 1 to n-h
S[i,j] = 0; \\ Creating the counting matrix S of size n-h x h
for j = 1 to h
    for i = n-h downto 1
        if (M[i,j] == 1)
            S[i,j] = S[i+1,j]+1;
        else
            S[i,j] = 0;
```

Figure 6: Building matrix S

```
\\ Evaluating the gain of strings that are candidates for the dictionary
for i = 1 to n − h
    for j = 1 to h
        if (S[i, j] ≥ j)
            Gain(W[i..i + j]) = IntegerPart(S[i, j]/ j) ∗ ( j − 1);
            j = h;
            i = i + IntegerPart(S[i, j]/ j);
        else if (S[i, j] > 0)
            for k = 1 to h
                if S[i,k] ≥ S[i, j]
                    p = p + 1;
            if (p ∗ (S[i, j] − 1) > Gain(W[i..i + j]))
                Gain(W[i..i + j]) = p ∗ (S[i, j] − 1);
            i = i + 1;
```

Figure 7: Evaluating the gain of strings

+14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+12	8	7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+8	0	0	0	0	0	0	0	0	4	3	2	1	0	0	0	0	0	0	0	0	0	0	0
+7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+4	0	0	0	0	4	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	65	64	65	66	67	68

Figure 8: Matrix S

such as loops in programs; the second "if" ($S[i, j] > 0$) finds remaining strings that are executed frequently within the same window, but in a scattered manner. For each i the algorithm picks a string with the highest gain. The algorithm can be easily extended so that if there are more than one string with the highest gain, the longest string is picked.

Step C

We now create a forest data structure to store the strings identified in Step B; we store the starting memory address of the sequences identified in Step B. Each tree within the forest is used to store the subsequent memory addresses in the strings that were identified. An example of the forest data structure is shown in Figure 3(d). This tree corresponds to the program trace shown in Figure 3; the left branch corresponds to the trace shown in Figure 3(c) and the left branch of the right branch corresponds to the trace shown in Figure 3(a), and the right part of the right branch corresponds to the trace shown in Figure 3(b). Each node where a string terminates has a counter associated with it; this counter is incremented each time the corresponding string is found. Thus, the final value of the counter will reflect how many times the corresponding string has been found in step B.

Step D

In step D, we encode the trace. The encoded trace file is a sequence of either addresses that are not initial element of a string from the forest (i.e., not a root of any of the trees in the forest) or as a sequence of indices that point to the dictionary entries. Figure 9 shows the trace encoding procedure. We read the trace file matching read addresses with strings in the tree, until we cannot traverse down the tree. We look at all substrings of the traversed branch of the tree and pick the string with largest gain (if several strings have such gain, the longest string is picked).

4.1 Computational Complexity

Building the matrix M for a window of size n is quadratic in n; building the counting matrix S is cubic in n. Note that this is an extremely pessimistic estimate and in practice the time complexity is essentially quadratic in n, because the matrix M is very sparse, and the largest sequence of consecutive ones to be counted is very small compared to the window size n. If the trace is of size t, then the number of windows to process is $2t/n$. Thus the total work prior

```
trace_index = 0;
dict_index = 1;
while trace_file != EOF
    max_string = null;
    addr = read(trace_file, trace_index);
    trace_index++;
    string = null;
    until found_in_the_tree(string + addr) == null
        string = string + addr;
        if gain(string) ≥ gain(max_string)
            max_string = string;
        addr=read(trace_file, trace_index);
        trace_index++;

    if max_string == NULL
        compress = compress + addr;
    else
        if dict_index(max_string) == 0
            dict_index(max_string)=dict_index;
            dict_index++;
        compress = compress + index(max_string);
        dictionary = dictionary + (dict_index(max_string), max_string);
        trace_index = trace_index(max_string) + 1;
```

Figure 9: Trace encoding procedure

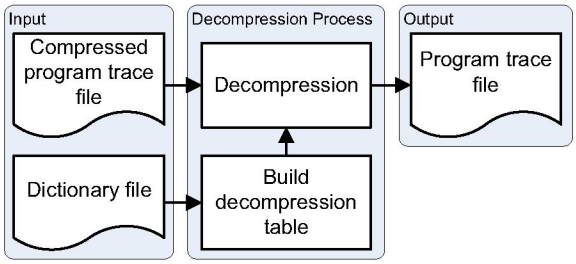

Figure 10: Decompression process

to encoding is $O(t \cdot n)$ (expected), $O(t \cdot n^2)$ (unrealistic worst case). Finally, encoding is linear in t. Thus, as verified in practice, the expected run time is proportional to $t \cdot n$. Notice that the compression efficiency, for as long as reasonably feasible, is not very critical. This is because the compression is just a preprocessing step that is done only once for each program trace. Subsequently, a set of stored pre-processed (compressed) traces is reused many times for different statistical analyses of various applications with various inputs.

4.2 Decompression for Verification Purpose

To verify and guarantee the correctness of the compression algorithm, a decompression algorithm was built to reproduce the trace file given the dictionary file and the compressed trace file. Note that in actual use, decompression is **not** needed; our analysis algorithm executes on the compressed trace.

The decompression algorithm is shown in Figure 10. The decompression algorithm takes the compressed trace file and the dictionary file as inputs, and outputs the trace file.

As shown in Figure 10, the algorithm reads the dictionary file to build a decompression table. The decompression step reads the list of indexes stored in the compressed trace file and uses the information in the decompression table to produce the trace file.

5. Cache Simulation Algorithm

Cache simulation algorithm aims to calculate the number of cache misses that can occur given a cache configuration and a compressed program trace. Inputs to the cache simulation algorithm are the cache parameters, the dictionary file, and the compressed program trace file. Figure 11 shows the procedures of the cache simulation algorithm.

Each entry in the dictionary is an instruction string that represent

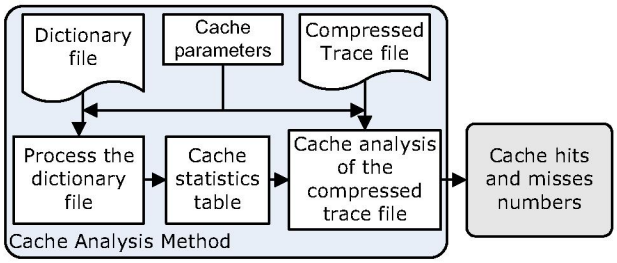

Figure 11: Cache analysis methodology

```
\\ Initialize the cache address array
\\ M = total number of cache location
\\ M is equal to number of cache set for a direct-mapped cache
for a = 0 to M
    cache = 0;

\\ process the dictionary file
\\ N = total number on entries in the dictionary based on the dictionary index
for b = 1 to N
    sim_cache(b);

\\ start of cache analysis
\\ S = total number of entries in the compressed trace file
for c = 0 to S
    if (c == dictionary_index)
      update_cache(dict_data(c));
    else
      update_cache(c);
```

Figure 12: Cache simulation algorithm

the sequence of memory references. Figure 12 displays the cache simulation algorithm. The algorithm reads the compressed program trace file and determines the number of cache misses that occur.

The cache simulation algorithm starts by initializing an array to keep track of the content of each cache location. Size of the array for keeping track of the content of the cache is dependent on the cache parameters (i.e. cache associativity, cache line size, and cache size).

The dictionary file is then read and cache simulation (function *sim_cache*) is performed for each instruction string entry in the dictionary. Cache simulation on the instruction strings will calculate the cache compulsory miss for loading the particular string into the cache, the cache conflict miss for executing the string, and finally the resulting cache footprint after the execution of the instruction string.

sim_cache function simulates the instruction string as if the instructions were to be executed through the cache memory. After processing the entries in the dictionary, a cache statistics table is obtained. The table is indexed using the instruction string index and each table entry store the following information.

- *cache_compulsory_content*, a list of instructions that occupy the cache pre-execution of the instruction string
- *cache_conflict*, cache conflict miss number due to the execution of the string
- *cache_footprint*, a list of instructions that occupy the cache post-execution of the instruction string.

The cache simulation algorithm then continues by reading the compressed program trace file. If a memory reference is read, it is to be compared with the existing entry or entries (depending on cache associativity parameters) in the same cache location and a cache hit or miss can then be determined and the cache content for that location is to be updated with the read memory reference. Otherwise, if a dictionary index is read from the compressed trace file, the index

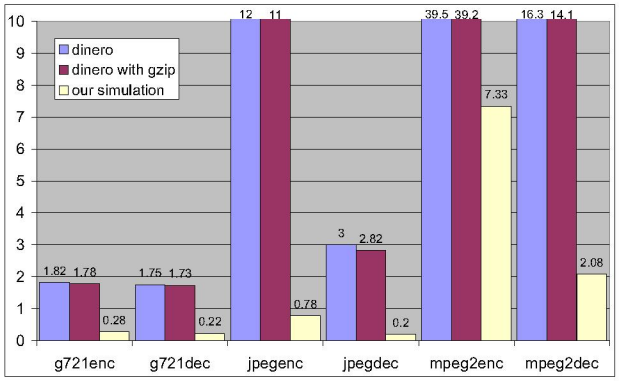

Figure 13: Cache simulation total runtime

is used to access the entry in the cache statistics table using the function *update_cache*. The following three steps are performed by the *update_cache* function.

1. The existing cache entry is compared with the entries in the *cache_compulsory_content* list from the table to determine the number of cache hit or miss occurred.
2. The *cache_conflict* value is added to the total cache miss number.
3. The content of the cache is updated with the resulting cache footprint, *cache_footprint* list from the table.

Performance improvement over existing cache analysis tools such as DineroIV [18] is expected due to the smaller compressed trace file to process, compared to the existing trace file size, and due to pre-simulation of subsets of the trace when analyzing the dictionary entries. Results from the cache simulation is verified by comparing against the output from DineroIV [18]. We found our cache simulation algorithm to be 100% accurate compared to DineroIV outputs (i.e. there are no errors in the output).

6. Experimental Setup and Results

Experiments were performed to gauge the compression ratio of the compression algorithm and to verify the cache simulation algorithm with an existing tool (Dinero IV[18]).

Benchmarks were taken from Mediabench [17]. Column 1 in Table1 shows the six benchmarks used in experiments. The rest of Table 1 displays the following information; column 2 shows the number of memory references in the trace, column 3 shows the size of the trace file in bytes, column 4 shows the size of the gzip trace file in bytes, column 5 shows the resulting compressed file size in bytes, column 6 shows the ratio of the compressed trace file compared to the original trace file (column 5 divided by column 3), column 7 shows the size of the compressed trace file with 'gzip', column 8 shows the ratio of Dinero IV total runtime compared to the total runtime of our cache simulation methodology, and column 9 shows the ratio of Dinero IV with inputs piped from a gzip trace file using zcat compared to the total runtime of our simulation.

Program traces were generated using Instruction Set Simulator (ISS) for Tensilica Xtensa proccesor [1]. The compression algorithm was implemented in C and compiled with gcc version 4.0.2; the program was executed in a dual processor dual Opteron 2.2GHz machine.

Cache analysis is performed for various cache configurations. The cache parameters used were: cache size of 128 bytes to 16384 bytes, cache line size of 8 bytes to 256 bytes, and cache associativity from 1 to 32 (each parameter is incremented in powers of two; total cache configuration simulated is 206). Cache simulations were executed on the compressed file and the cache miss numbers output from the cache simulation was verified against Dinero IV to ensure correctness of the cache simulation output.

Application	Trace size	File size (bytes)	gzip file size (bytes)	Compressed file size (bytes)	compression ratio	gzip compressed file size (bytes)	runtime ratio	runtime ratio (gzip)
g721enc	108,635,308	977,717,772	12,878,673	38,049,493	25.7	2,743,007	6.41	6.35
g721dec	105,800,842	952,207,578	12,161,013	25,646,458	37.1	1,473,615	8.08	8.0
jpegenc	11,508,580	103,577,220	1,051,695	9,895,273	10.5	195,292	15.32	14.10
jpegdec	2,842,673	25,584,057	422,638	2,681,652	9.5	42,771	15	14.08
mpeg2enc	2,359,305,107	21,233,745,963	297,736,877	1,139,867,708	18.6	46,256,026	5.39	5.35
mpeg2dec	937,319,532	8,435,875,788	111,485,710	255,749,085	33.0	11,621,180	7.8	6.78

Table 1: Benchmark list

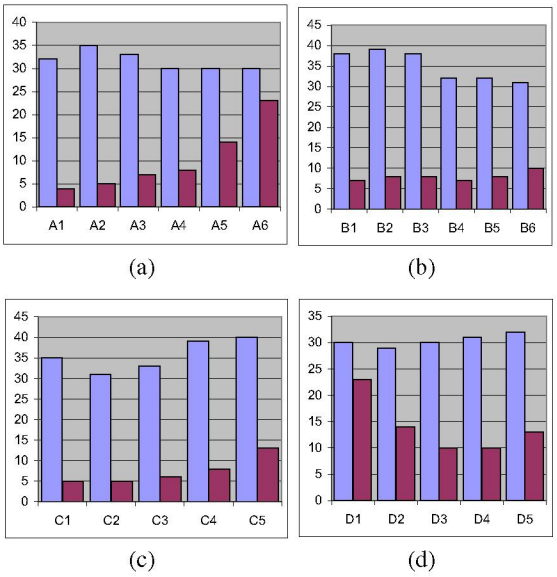

Figure 14: Cache simulation runtime comparison

Cache config.	Cache parameters size, line size, assoc.	Cache config.	Cache parameters size, line size, assoc.
A1	512,8,1	B1	512,8,8
A2	1024,8,1	B2	1024,8,8
A3	2048,8,1	B3	2048,8,8
A4	4096,8,1	B4	4096,8,8
A5	8192,8,1	B5	8192,8,8
A6	16384,8,1	B6	16384,8,8
C1	1024,8,1	D1	16384,8,1
C2	1024,8,2	D2	16384,8,2
C3	1024,8,4	D3	16384,8,4
C4	1024,8,8	D4	16384,8,8
C5	1024,8,16	D5	16384,8,16

Table 2: Cache configuration

Figure 13 shows the comparison of the total execution time for simulating multiple cache configurations with Dinero IV, Dinero IV with gzip, and our simulation tool. Due to large differences between the total runtime of the individual benchmarks, the bar graphs in Figure 13 are not on the same scale. Benchmarks 'jpegenc' and 'jpegdec' total runtime bar is measured in minutes, while 'g721enc', 'g721dec', 'mpeg2enc', and 'mpeg2dec' runtime are measured in hours. The ratio between the time taken by Dinero IV over our simulation methodology is shown in column 8 on Table 1.

Figure 14 shows a comparison of the runtime of Dinero IV compared to our simulation methodology for simulating individual cache configuration of 'g721enc' program trace file. The left bars shown in Figure 14 indicate the Dinero IV runtime and the right bars indicate the runtime of our simulation methodology, the y-axis shows the runtime in seconds. Table 2 shows the cache configuration for each of the result shown in Figure 14. Each bar comparison in Figure 13 and Figure 14 show that our simulation methodology is always faster when compared to Dinero IV runtime.

Column 6 in Table 1 shows that the compression algorithm can compress program traces by an average factor of 22.4. Results shown in column 8 of Table 1 show that by using our simulation methodology, it is possible to accurately perform faster cache simulation by an average factor of 9.67 when compared to Dinero IV, and average speedup factor of 9.10 when compared to Dinero IV with gzip.

7. Conclusions

This paper presents a novel method for cache analysis by analyzing a compressed program trace file without the need for prior decompression. An average compression ratio of 22.4 is seen in the experimental results for large program trace files with size up to 21Gbytes. Cache simulation method speedup average of 9.67 is observed when compared to Dinero IV, and average speedup factor of 9.10 is observed when compared to Dinero IV with gzip.

8. References

[1] Xtensa Processor, *http://www.tensilica.com.*

[2] Seagate Technology, *http://www.seagate.com/docs/pdf/marketing/po_db35.pdf*

[3] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design," *DAC,* pp. 294-299, 2000.

[4] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. on Information Theory,* v. 23, pp. 337 - 343, May 1977.

[5] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. on Information Theory,* v. 24, n. 5, Sept. 1978.

[6] T. A. Welch, "A Technique for High-Performance Data Compression," *Computer,* v. 17, n. 6, June 1984, pp. 8 - 19.

[7] A. R. Pleszkun, "Techniques for Compressing Program Address Traces," *IEEE/ACM Micro,* pp. 32-40, Nov. 1994.

[8] E. E. Johnson and J. Ha, "PDATS: Lossless address space compression for reducing file size and access time," *IEEE International Phoenix Conference on Computers and Communication,* 1994.

[9] E. E. Johnson, J. Ha, and M. B. Zaidi, "Lossless Trace Compression," *IEEE Transactions on Computers,* v. 50, n. 2, February 2001.

[10] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A linear-time algorithm," *JAIR,"* pp.67-82, 1997.

[11] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson, "Flexible Reference Trace Reduction for VM Simulations," *ACM TOMACS,* v.13, n.1, pp. 1-38, 2003.

[12] A. Milenkovic and M. Milenkovic, "Exploiting Streams in Instruction and Data Address Trace Compression," *IEEE WWC,* pp. 99-107, 2003.

[13] M. Burtscher., "VPC3: A Fast and Effective Trace-Compression Algorithm," *SIGMETRICS'04,* June 2004.

[14] Y. Luo and L. K. John, "Locality-Based Online Trace Compression," *IEEE Transactions on Computers,* v. 53, n. 6, June 2004.

[15] X. Zhang and R. Gupta, "Whole Execution Traces," *IEEE Micro,* pages 105 - 116, 2004.

[16] X. Zhang and R. Gupta, "Whole Execution Traces and Their Applications," *ACM TACO,* v. 2, n. 3, pp. 301 - 334, September 2005.

[17] C. Lee et.al., "MediaBench: A Tool for Evaluating Multimedia and Communications Systems," *IEEE MICRO 30,* 1997.

[18] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," *http://www.cs.wisc.edu/~markhill/DineroIV/.*

[19] S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: An Analytical Representation of Cache Misses," *ICS,* Vienna, Austria, 1997.

[20] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers,* v. 38, n. 12, pp. 1612-1630, 1989.

[21] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "Finding Optimal L1 Cache Configuration for Embedded Systems," *ASP-DAC,* 2006.

[22] A. Ghosh and T. Givargis, "Analytical Design Space Exploration of Caches for Embedded Systems," *DATE,* pp. 650 - 655, 2003.