International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-1 Issue-4, September 2012

Single Dictionary based Cache Compression and Decompression Algorithm

Prasad Munasa, P.Jayanagalakshmi

Abstract: ***Computer systems and micro architecture researchers have proposed using hardware data compression units within the memory hierarchies of microprocessors in order to improve performance, energy efficiency, and functionality. All work on cache compression, has made unsubstantiated assumptions about the performance, power consumption, and area overheads of the proposed compression algorithms and hardware. It is not possible to determine whether compression at levels of the memory hierarchy closest to the processor is beneficial without understanding its costs. Furthermore, as we show in this paper, raw compression ratio is not always the most important metric. In this paper, we present a lossless compression algorithm that has been designed for fast on-line data compression, and cache compression in particular. The algorithm has a number of novel features tailored for this application, including combining pairs of compressed lines into one cache line and allowing parallel compression of multiple words while using a single dictionary and without degradation in compression ratio. Apart from that we reduced the proposed algorithm to a register transfer level hardware implementation on Xilinx xc3s500E fpga permitting performance, power consumption, and area estimation. The total power consumption of the device was estimated to be 0.081W.***

Index Terms: ***Cache compression, pair matching, parallel compression, hardware implementation***

I. INTRODUCTION

This paper addresses the increasingly important issue of controlling off-chip communication in computer systems in order to maintain good performance and energy efficiency.. The ongoing move to chip-level multiprocessors (CMPs) is further increasing the problem; more processors require more accesses to memory, but the performance of the processor–memory bus is not keeping pace. Techniques that reduce off-chip communication without degrading performance have the potential to solve this problem. Cache compressions one such technique; data in last-level on-chip caches, e.g., L2 caches, are compressed, resulting in larger usable caches. In the past, researchers have reported that cache compression can improve the performance of uni processors by up to 17% for memory-intensive commercial workloads [1] and up to 225% for memory-intensive scientific workloads [2]. Researchers have also found that cache compression and perfecting techniques can improve CMP throughput by 10%–51% [3]. Cache compression presents several challenges. First, decompression and compression must be extremely fast: a significant increase in cache hit latency will overwhelm the advantages of reduced cache miss rate. This requires an efficient on-chip decompression hardware implementation. Second, the hardware should occupy little area compared to the corresponding decrease in the physical size of the cache, and should not substantially increase the total chip power consumption.

Conventional compression algorithm quality metrics, such as block compression ratio, are not appropriate for judging quality in this domain. Instead, one must consider the effective system-wide compression ratio .This paper will point out a number of other relevant quality metrics for cache compression algorithms, some of which are new. Finally, cache com-press ion should not increase power consumption substantially. The above requirements prevent the use of high-overhead compression algorithms such as the PPM family of algorithms [4] or Burrows-Wheeler transforms [5]. A faster and lower-overhead technique is required.

II. CACHE COMPRESSION ARCHITECTURE

In this section, we describe the architecture of a CMP system in which the cache compression technique is used. We consider private on-chip L2 caches, because in contrast to a shared L2 cache, the design styles of private L2 caches remain consistent when the number of processor cores increases. We also examine how to integrate data prefetching techniques into the system.

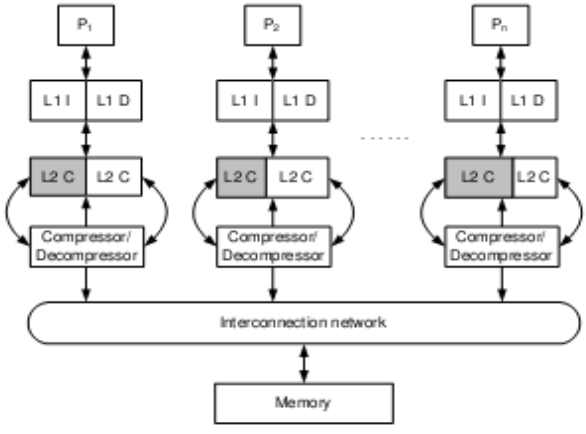

Fig. 1. System architecture in which cache compression is used.

Figure 1 gives an overview of a CMP system with processor cores. Each processor has private L1 and L2 caches. The L2 cache is divided into two regions:

Revised Manuscript Received on 30 September 2012
*Correspondence Author(s)
Prasad Munasa, M.Tech Department of ECE, JNTU Kakinada University, Kaushik College of Engineering, Visakhapatnam, India.
P. Jayanagalakshmi, Asst. Professor, Department of ECE, Kaushik College of Engineering, India.

an uncompressed region (L2 in the figure) and a compressed region (L2C in the figure). For each processor, the sizes of the uncompressed region and compression region can be determined statically or adjusted to the processor's needs dynamically. In extreme cases, the whole L2 cache is compressed due to capacity requirements, or uncompressed to minimize access latency.

We assume a three-level cache hierarchy consisting of L1 cache, uncompressed L2 region, and compressed L2 region. The L1 cache communicates with the uncompressed region of the L2 cache, which in turn exchanges data with the compressed region through the compressor and decompressor, i.e., an uncompressed line can be compressed in the compressor and placed in the compressed region, and vice versa. Compressed L2 is essentially a virtual layer in the memory hierarchy with larger size, but higher access latency, than uncompressed L2. Note that no architectural changes are needed to use the proposed techniques for a shared L2 cache. The only difference is that both regions contain cache lines from different processors instead of a single processor, as is the case in a private L2 cache.

III. C-PACK COMPRESSION ALGORITHM

A. C-Pack Algorithm Overview

C-Pack (for Cache Packer) is a lossless compression algorithm designed specifically for high-performance hardware-based on-chip cache compression. It achieves a good compression ratio when used to compress data commonly found in microprocessor low-level on-cchip caches.

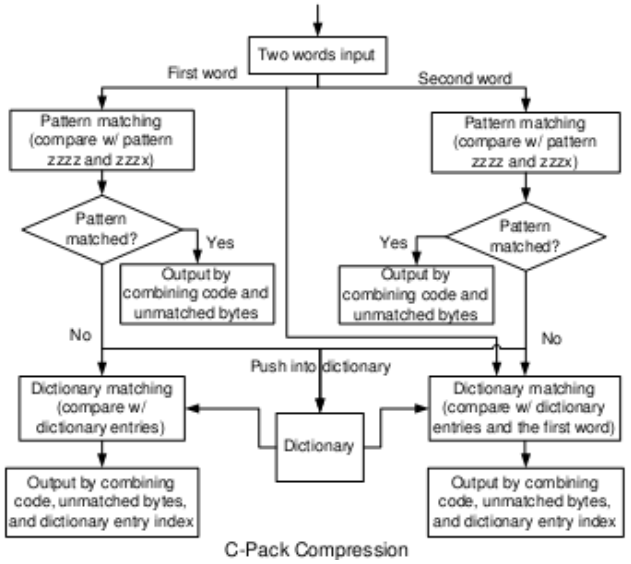

Figure 2: C-Pack Compression

C-Pack achieves compression by two means: (1) it uses statically decided, compact encodings for frequently appearing data words and 2) it encodes using a dynamically updated dictionary allowing adaptation to other frequently appearing words. The C-Pack compression algorithm are illustrated in Figure 2. We use an input of two words per cycle as an example in Figure 2. However, the algorithm can be easily extended to cases with one, or more than two, words per cycle. During one iteration, each word is first compared with patterns "zzzz" and "zzzx". If there is a match, the compression output is produced by combining the corresponding code. Otherwise, the compressor compares the word with all dictionary entries and finds the one with the most matched bytes. The compression result is then obtained by combining code, dictionary entry index, and unmatched bytes, if any. Words that fail pattern matching are pushed into the dictionary.

Figure 3 shows the decompression algorithm. During decompression, the decompressed first reads compressed words and extracts the codes for analyzing the patterns of each word, which are then compared against the codes defined in Table I. If the code indicates a pattern match, the original word is recovered by combining zeroes and unmatched bytes, if any. Otherwise, the decompression output is given by combining bytes from the input word with bytes from dictionary entries, if the code indicates a dictionary match.

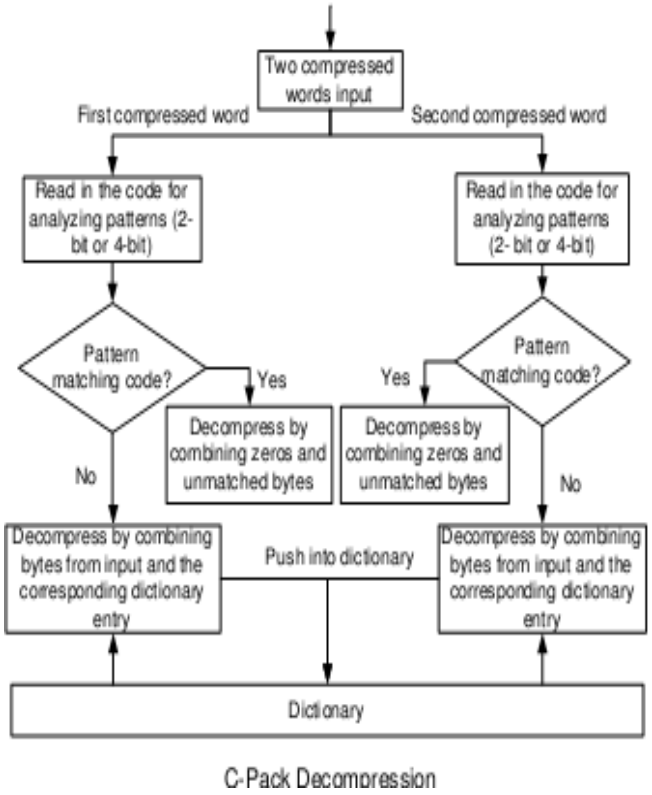

Figure 3: C-Pack Decompression

The C-Pack algorithm is designed specifically for hardware implementation. It takes advantage of simultaneous comparison of an input word with multiple potential patterns and dictionary entries. This allows rapid execution with good compression ratio in a hardware implementation, but may not be suitable for a software implementation. In general, software must process operations sequentially. In the proposed implementation of C-Pack, two words are processed in parallel per cycle. Achieving this, while still permitting an accurate dictionary match for the second word, is challenging. Let us consider compressing two similar words that have not been encountered by the compression algorithm recently, assuming the dictionary uses first-in first-out (FIFO) as its replacement policy. The appropriate dictionary content when processing the second word depends on whether the first word matched a static pattern. If so, the first word will not appear in the dictionary. Otherwise, it will be in the dictionary, and its presence can be used to encode the second word. Therefore, the second word should be compared with the first word and all but the first dictionary entry in parallel..

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

Therefore, we can compress two words in parallel without compression ratio degradation.

B. Effective System-Wide Compression Ratio

Compressed cache organization is a difficult task because different compressed cache lines may have different lengths. The main idea is to divide compressed cache lines into fixed-size segments and use indirect indexing to locate all the segments for a compressed line. The pro-posed cache design decouples accesses across the whole cache, thus allowing a fully-associative placement. Each tag contains multiple pointers to smaller fixed-size data blocks to represent a single cache block. However, the tag storage overhead of IIC-C is significant, e.g., 21% given a 64 B line size and 512 KB cache size, compared to less than 8% for our proposed pair-matching based cache organization. In addition, the hardware overhead for addressing a compressed line is not discussed in the paper. The access latency in IIC-C is attributed to three primary sources, namely additional hit latency due to sequential tag and data array access, tag lookup induced additional hit and miss latency, and additional miss latency due to the overhead of software management. Only the cache lines with a compression ratio of less than 0.5 are compressed so that two compressed cache lines can fit in the space required for one uncompressed cache line. However, this will inevitably result in a larger system-wide compression ratio compared to that of pair-matching based cache because each compression ratio, not the average, must be less than 0.5, for compression to occur.

C. Pair-Matching Compressed Line Organization

We propose the idea ofpair-matching to organize compressed cache lines. In a pair-matching based cache, the location of a newly compressed line depends on not only its own compression ratio butNalso the compression ratio of its "partner". More specifically, the compressed line locator first tries to locate the cache line (within the set) with sufficient unused space for the compressed line without replacing any existing compressed lines. If no such line exists, one or two compressed lines are evicted to store the new line. A compressed line can be placed in the same line with a partner only if the sum of their compression ratios is less than 100%.

To reduce hardware complexity, the candidate partner lines are only selected from the same set of the cache. Compared to segmentation techniques which allow arbitrary positions, pair-matching simplifies designing hardware to manage the locations of the compressed lines.

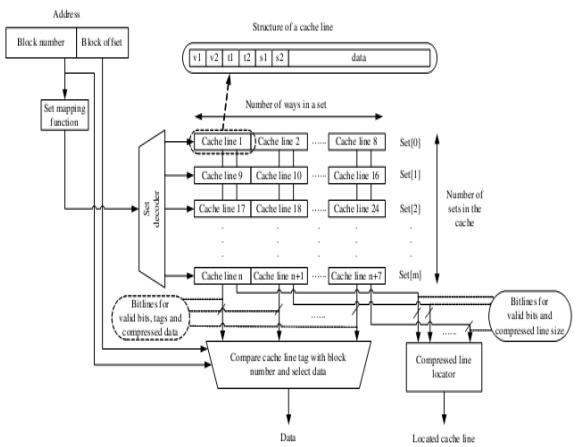

Figure 4 : Structure Of Pair-Matching Based Cache

Figure 4 illustrates the structure of an 8-way associative pair-matching based cache. Since any line may store two compressed lines, each line has two valid bits and tag fields to indicate status and indexing. When compressed, two lines share a common data field. There are two additional size fields to indicate the compressed sizes of the two lines. Whether a line is compressed or not is indicated by its size field. Assize of zero is used to indicate uncompressed lines. For compressed lines, size is set to the line size for an empty line, and the actual compressed size for a valid line. The effective system-wide compression ratio is defined as the average of the effective compression ratios of all cache lines in a compressed cache. It indicates how well a compression algorithm performs for pair-matching based cache compression. The concept of effective compression ratio can also be adapted to a segmentation based approach.

IV. COMPRESSOR ARCHITECTURE

This section gives an overview of the proposed compressor architecture, and then discusses the data flow among different pipeline stages inside the compressor.

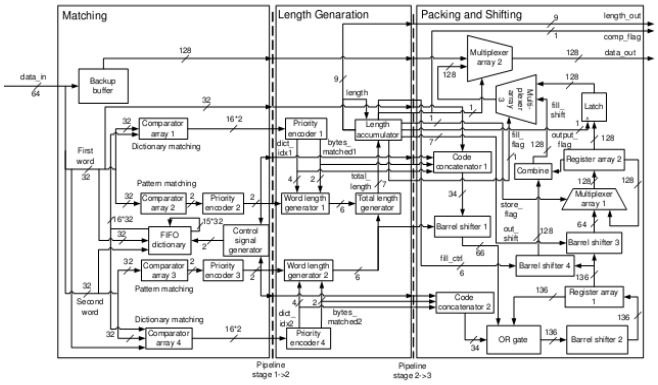

Figure 5: Compressor Architecture

Figure 5 illustrates the hardware compression process. The compressor is decomposed into three pipeline stages. This design supports incremental transmission, i.e., the compressed data can be transmitted before the whole data block has been compressed. This reduces compression latency. We use bold and italic fonts to represent the devices and signals appearing in figures

1) Pipeline Stage 1: The first pipeline stage performs pattern matching and dictionary matching on two uncompressed words in parallel. As illustrated in Figure 5, comparator array 1 matches the first word against pattern "zzzz" and "zzzx" and comparator array 2 matches it with all the dictionary entries, both in parallel. The same is true for the second word. However, during dictionary matching, the second word is compared with the first word in addition to the dictionary entries. The pattern matching results are then encoded using priority encoders 2 and 3, which are used to determine whether to push these two words into the FIFO dictionary. Note that the first word and the second word are processed simultaneously to increase throughput.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

2) Pipeline Stage 2: This stage computes the total length of the two compressed words and generates control signals based on this length. Based on the dictionary matching results from Stage 1,priority encoders 1 and 4find the dictionary entries with the most matched bytes and their corresponding indices, which are then sent toword length generators 1 and 2to calculate the length of each compressed word. The total length calculator adds up the two lengths, represented by signal total length. The length accumulator then adds the value of total length to two internal signals, namely sum partia land sum total. Sum partial records the number of compressed bits stored in register array 1 that have not been transmitted.

Whenever the updated sum partial value is larger than 64 bits, sum partial is decreased by 64 and signal store flag is generated indicating that the 64 compressed bits in register array 1 should be transferred to either the left half or the right half of the 128-bit register array 2, depending on the previous state of register array 2. It also generates signal out shift specifying the number of bits register array 1should shift to align with register array 2. Sum total represents the total number of compressed bits produced since the start of compression. Whenever sum total exceeds the original cache line size, the compressor stops compressing and sends back the original cache line stored in the backup buffer.

3) Pipeline Stage 3: This stage generates the compression output by combining codes, bytes from input word, and bytes from dictionary entries depending on the pattern and dictionary matching results from previous stages. Placing the compressed pair of words into the right location within register array 1, denoted by Reg1 [135:0], is challenging. Since the length of a compressed word varies from word to word, it is impossible to pre-select the output location statically. In addition, register array 1should be shifted to fi in the compressed output in a single cycle without knowing the shift length in advance. We address this problem by analyzing the output length. Notice that a single compressed word can only have 7 possible output lengths, with the maximum length being 34 bits. Therefore, we use two 34-bit buffers, denoted by A[33:0] and B[33:0], to store the first and second compressed outputs generated by code concatenates 1 and 2in the lower bits, with the higher unused bits set to zero. Reg1[135:0] is shifted by total length using barrel shifter 2, with the shifting result denoted by Reg1s[135:0]. At the same time, A[33:0] is shifted using barrel shifter 1by the output length of the second compressed word. The result of this shift is held by S[65:0], also with all higher (unused) bits set to zero. Note thatReg1[135:68] only has one input source, i.e., Reg1s[135:68],because the maximum total output length is 68. However,Reg1[67:2] can have multiple input sources: B, S, and Reg1s.For example, Reg1[4] may come from B[4], S[2], or Reg1s[0].To obtain the input to Reg1[135:0], we OR the possible inputs together because the unused bits in the input sources are all initialized to zero, which should not affect the result of an OR function.

Meanwhile, Reg1[135:0] is shifted by out shift using barrel shifter 3 to align with register array 2, denoted byReg2[135:0]. Multiplexer array 1selects the shifting result as the input to Reg2[135:0] when store flag is 1 (i.e., the number of accumulated compressed bits has exceeded 64 bits)and the original content of Reg2[135:0] otherwise. Whether Latch is enabled depends on the number of compressed bits accumulated in Reg2[135:0] that have not been transmitted. When output flag is 1, indicating that 128 compressed bits have been accumulated in Reg2[135:0], Reg2[135:0] is passed to Multiplexer array 1. Multiplexer array 3selects between fill shift and the output of latch using fill flag. Fill shift rep-resents the 128-bit signal that pads the remaining compressed bits that have not been transmitted with zeros and fill flag determines whether to select the padded signal. Multiplexer array 2then decides the output data based on the total number of compressed bits. When the total number of compressed bits has exceeded the uncompressed line size, the contents of backup buffer are selected as the output. Otherwise, the output from Multiplexer array 3is selected.

V. DECOMPRESSOR ARCHITECTURE

1) Decompressor Architecture: Figure 6 illustrates the decompressor architecture. Recall that the compressed line, which may be nearly 512 bits long, is processed in 128-bit blocks, the width of the bus used for L2 cache access. The use of a fixed-width bus and variable-width compressed words implies that a compressed word may sometimes span two 128-bit blocks. This complicates decompression. In our design, two words are decompressed per cycle until fewer than 68 bits remain in register array 1 (68 bits is the maximum length of two compressed words). The decompressor then shifts in more compressed data using barrel shifter 2 and concatenates them with the remaining compressed bits. In this way, the decompressor can always fetch two whole compressed words per cycle. The decompressor also supports incremental transmission, i.e., the decompression results can be transmitted before the whole cache line is decompressed provided that there are 128 decompressed bits in register array 3. The average decompression latency is 5 cycles.

1) Word Unpacking: When decompression starts, the un packer first extracts the two codes of the first and second word. Signals first code and second code represent the first two bits of the codes in the two compressed words. Signal first bak and second bak refer to the two bits following first code and second code, respectively. They are mainly useful when the corresponding code is a 4-bit code.

words, which are then decompressed by combining zero bytes, bytes from FIFO dictionary, and bytes from 2) Word Decompressing: To derive the patterns for the two register array 1(which stores the remaining compressed bits). The way the bytes are combined to produce the decompression results depends on the values of the four code-related signals. The decompressed words are then pushed into the FIFO dictionary, if they do not match pattern “zzzz” and “zzzx”, and register array 3. Note that the decompression results will be transmitted out as soon as register array 3 has accumulated four decompressed words, given the input line is a compressed line.

3) Length Updating: Length generator derives the compressed lengths of the two words, i.e., first len and second len, based on the four code-related signals. The two lengths are then subtracted from chunk len, which denotes the number of the remaining bits to decompress in register array 1. As we explained above, the subtraction result len r is then compared with 68, and more data are shifted in and concatenated with the remaining compressed bits in register array 1if len-r is less than 68. Meanwhile, register array 1is shifted by total length (the sum of first len and second len) to make space for the new incoming compressed bits.

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)

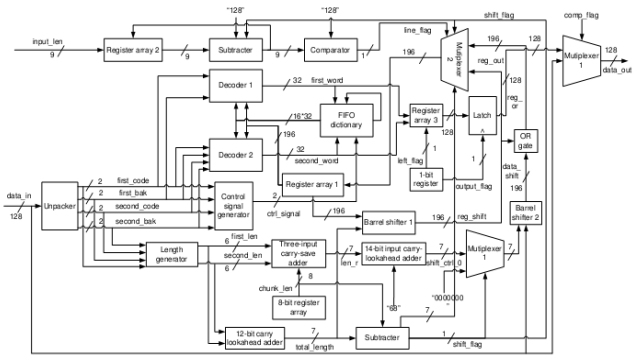

Fig. 6.Decompressor Architecture

VI. EVALUATION

In this section, we present the evaluation of the C-Pack hardware. We first present the performance, power consumption, and area overheads of the compression/decompression hardware when synthesized for integration within a microprocessor. Then, we compare the compression ratio and performance of C-Pack to other algorithms considered for cache compression: MXT [6], Xmatch,and FPC .Finally, we describe the implications of our findings on the feasibility of using C-Pack based cache compression within a microprocessor.

A. C-Pack Synthesis Results

We synthesized our design using Xilinx xc3s500E fpga and the device utilization summary is mentioned below.
Compressor architecture:

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3	4656	0%
Number of Slice Flip Flops	3	9312	0%
Number of 4 input LUTs	6	9312	0%
Number of bonded IOBs	12	232	5%
Number of GCLKs	1	24	4%

Decompressor architecture:

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	4	4656	0%
Number of Slice Flip Flops	5	9312	0%
Number of 4 input LUTs	7	9312	0%
Number of bonded IOBs	12	232	5%
Number of GCLKs	1	24	4%

Total Power Consumption:

Device	
Family	Spartan3e
Part	xc3s500e
Package	fg320
Grade	Commercial
Process	Typical
Speed Grade	-4

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.000	1	--	--
Logic	0.000	5	9312	0.1
Signals	0.000	11	--	--
IOs	0.000	12	232	5.2
Leakage	0.081			
Total	0.081			

B. Comparison of Compression Ratio

We compare C-Pack to several other hardware compression designs, namely X-Match, FPC, and MXT, that may be considered for cache compression. We exclude other compression algorithm because they either have not been implemented in hardware or are not suitable for cache compression. Although the proposed hardware implementation mainly targets online cache compression, it can also be used in other high-performance lossless data compression applications with few or no changes. We tested the compression ratios of different algorithms on cache data traces gathered from a full system simulation of various workloads from the Media bench and SPEC CPU2000 benchmark suites. The block size and the dictionary size are both set to 64 B in all test cases. Since we are unable to determine the exact compression algorithm used in MXT, we used the LZSS Lempel Ziv compression algorithm to approximate its compression ratio.. Each row shows the raw compression ratios and effective system-wide compression ratios using different compression algorithms for an application. The poor raw compression ratios of MXT are mainly due to its limited dictionary size. The same trend is seen for effective system-wide compression ratios: X-Match has the lowest (best) and MXT has the highest (worst) effective system-wide compression ratio.

VII. CONCLUSION

This paper has proposed and evaluated an algorithm for cache compression that honors the special constraints this application imposes. The algorithm is based on pattern matching and partial dictionary coding. Its hardware implementation permits parallel compression of multiple words without degradation of dictionary match probability. The proposed algorithm yields an effective system-wide compression ratio of 61%, and permits a hardware implementation with a maximum decompression latency of 6.67 ns. These results are superior to those yielded by compression algorithms considered for this application in the past. Although the proposed hardware implementation mainly targets online cache compression, it can also be used in other high-performance lossless data compression applications with few or no modifications.

REFERENCES

1. A. R. Alameldeen and D. A. Wood, “Adaptive cache compression for high-performance processors,” inProc. Int. Symp. Computer Architecture, June 2004.
2. E. G. Hallnor and S. K. Reinhardt, “A compressed memory hierarchy using an indirect index cache,” inProc. Wkshp. Memory Performance Issues, 2004.
3. A. R. Alameldeen and D. A. Wood, “Interactions between compression and prefetching in chip multiprocessors,” in Proc. Int. Symp. High-Performance Computer Architecture, Feb. 2007.
4. A. Moffat, “Implementing the PPM data compression scheme,” in IEEE Trans. on Communications, Nov. 1990.
5. M. Burrows and D. Wheeler, “A block sorting lossless data compression algorithm,” Digital Equipment Corporation, Tech. Rep. 124, 1994.
6. B. Tremaine, et al., “IBM memory expansion technology,” IBM J.Research and Development, vol. 45, no. 2, pp. 271–285, Mar. 2001.
7. J. L. N´u˜nez and S. Jones, “Gbit/s lossless data compression hardware,” IEEE Trans. VLSI Systems, vol. 11, no. 3, pp. 499–510, June 2003.
8. A. Alameldeen and D. A. Wood, “Frequent pattern compression: A significance-based compression scheme for l2 caches,” Dept. of Computer Sciences, University of Wisconsin-Madison, Tech. Rep., Apr. 2004.

Published By:
Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP)