Design and Implementation of a High-Performance Microprocessor Cache Compression Algorithm

Xi Chen*, Lei Yang*, Haris Lekatsas†, Robert P. Dick*, and Li Shang‡

*EECS Dept.
Northwestern University
Evanston, IL
{xi-chen-0, l-yang, dickrp}
@northwestern.edu

†Vorras Corporation
Princeton, NJ
lekatsas@vorras.com

‡ECE Dept.
University of Colorado
Boulder, CO
li.shang@colorado.edu

Abstract

Researchers have proposed using hardware data compression units within the memory hierarchies of microprocessors in order to improve performance, energy efficiency, and functionality. However, most past work, and in particular work on cache compression, has made unsubstantiated assumptions about the performance, power consumption, and area overheads of the required compression hardware. We present a lossless compression algorithm that has been designed for on-line memory hierarchy compression, and cache compression in particular. We reduced our algorithm to a register transfer level hardware implementation, permitting performance, power consumption, and area estimation. The results of experiments comparing our work to previous work are presented.

I. Introduction

Microprocessor speeds have been increasing faster than off-chip memory latency, raising a "wall" between processor and memory. The ongoing move to chip-level multiprocessors (CMPs) is further fortifying this wall; 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 compression* is 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 uniprocessors 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 prefetching techniques can improve CMP throughput by 10–51% [3]. However, such benefits come at the cost of area and power consumption overheads of the compression/decompression hardware.

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. Third, the algorithm should losslessly compress small blocks, e.g., 64-byte cache lines, while maintaining a good compression ratio (throughout this paper we use the term *compression ratio* to denote the ratio of the compressed data size over the original data size). Conventional compression algorithm quality metrics, such as block compression ratio, are not appropriate for judging quality

This work was supported in part by AFOSR award FA9550-06-1-0152 and in part by NSF awards ITR-CCR-0325207 and CNS-0347941. We would like to thank Alok Choudhary for supporting this project.

in this domain. Instead, one must consider the effective system-wide compression ratio (defined precisely in Section III-C). This paper will point out a number of other relevant quality metrics for cache compression algorithms, some of which are new.

II. Related Work and Contributions

Assumptions about cache compression algorithms and hardware made in prior work can be divided into two main categories. A number of researchers have assumed the use of general-purpose main memory compression hardware, e.g., MXT [4], for on-chip cache compression. Although appropriate for compressing main memory, such hardware has performance, area, or power consumption costs that contradict its use in cache compression. For example, if the MXT hardware were scaled to a 65 nm fabrication process and integrated within a 1 GHz processor, the decompression latency would be 16 processor cycles, about twice the normal L2 cache hit latency. Other work proposes special-purpose cache compression hardware and evaluates only the compression ratio, disregarding other important criteria such as area and power consumption costs. For example, although the area cost for FPC [5] is not discussed, our analysis shows that FPC would have an area overhead of at least 290 K gates, almost eight times the area of the approach proposed in this paper, to achieve the claimed 5-cycle decompression latency. In short, assuming desirable cache compression hardware with adequate performance and low area and power overheads is common in cache compression research [2, 6–10]. However, without a cache compression algorithm and hardware implementation designed and evaluated under careful considerations of effective system-wide compression ratio, hardware overheads, and interaction with other portions of the cache compression system, it is difficult to determine whether the proposed architectural schemes are beneficial.

In this work, we propose and develop a lossless compression algorithm, named C-Pack, for on-chip cache compression. The main contributions of our work follow: 1) In contrast to other schemes such as X-match which contain complicated hardware to achieve an equivalent effective system-wide compression ratio, C-Pack has much lower performance, area, and power overheads for practical use; 2) C-Pack is twice as fast as the best existing hardware implementations potentially suitable for cache compression. It would require at least 8× the area of C-Pack for FPC to match this performance; 3) We are the first to fully design, optimize, and evaluate the performance and power consumption of a cache compression algorithm using a design flow appropriate for on-chip integration; and 4) We demonstrate when line compression ratio reaches 50%, further improving the line compression ratio has little impact on effective system-wide compression ratio.

III. C-Pack Compression Algorithm

In this section, we briefly describe the C-pack algorithm and several important features that permit its efficient hardware implementation, and validate the effectiveness of C-pack in a compressed cache architecture.

III.A. Design Constraints and Challenges

We first point out several design constraints and challenges peculiar to the cache compression problem: 1) Cache compression requires hardware that can de/compress a word in only a few CPU clock cycles, thus ruling out software implementations; 2) Cache compression algorithms must be lossless to maintain correctness; 3) The block size for cache compression is small, e.g., 64 bytes; and 4) The complexity of managing the

locations of cache lines after compression influences feasibility. Allowing arbitrary, i.e., bit-aligned, locations would increase complexity to the point of infeasibility. A scheme that permits pairs of compressed lines to fit within an uncompressed line is advantageous.

III.B. C-Pack Algorithm Overview

C-Pack (for Cache Packer) is a lossless compression algorithm designed specifically to permit a high performance hardware implementation. It achieves a good compression ratio when used to compress data commonly found in on-chip microprocessor caches, e.g., L2 caches. Its design was strongly influenced by prior work on pattern-based partial dictionary match compression [11]. However, this prior work was designed for main memory compression and did not consider hardware implementation.

C-Pack achieves compression by encoding frequently-appearing words through pattern matching and dictionary matching. The dynamically-updated dictionary supports full/partial word matching. The patterns and coding schemes used by C-Pack are summarized in Table I, which also reports the actual frequency of each pattern observed in the cache trace data described in Section III-D. In the 'Pattern' column, 'z' represents a zero byte, 'm' represents a byte matched against a dictionary entry, and 'x' represents an unmatched byte. In the 'Output' column, 'B' represents a byte and 'b' represents a bit.

TABLE I
PATTERN ENCODING FOR C-PACK

Code	Pattern	Output	Length	Freq. (%)
00	zzzz	(00)	2	39.7
01	xxxx	(01)BBBB	34	32.1
10	mmmm	(10)bbbb	6	7.6
1100	mmxx	(1100)bbbbBB	24	6.1
1101	zzzx	(1100)B	12	7.3
1110	mmmx	(1110)bbbbB	16	7.2

The C-Pack compressor and decompressor process two words per iteration. During compression, each word is first compared with pattern "zzzz" and "zzzx". If there is a match, the compression output is produced by combining the corresponding code and unmatched bytes as indicated in Table I. Otherwise, the compressor compares the word with all dictionary entries and finds the one with the most bytes matched. The compression result is then obtained by combining code, dictionary entry index, and unmatched bytes (if present). Words that fail pattern matching are pushed into the dictionary.

During decompression, the decompressor 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 present). 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.

For the above 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 a first-in first-out (FIFO) dictionary. The appropriate dictionary content when processing the second word depends on whether the first word matches 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.

III.C. Pair Matching Cache and Effective System-Wide Compression Ratio

Compressed cache organization is a difficult task because different compressed cache lines may have different lengths. Some researchers have proposed line segmentation

techniques [1, 2] to handle this problem. 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. However, the segmentation approach has significant overhead due to latency and complicated hardware to address all segments. As a result, the number of segments per line is tightly constrained, resulting in wasted space.

We propose a new scheme, called *pair-matching*, to organize compressed cache lines. In a pair-matching based cache, the compressed line locator first tries to locate a partner line with sufficient unused space 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.

In a pair-matching compressed cache, a newly-compressed line has an effective compression ratio of 100% when it takes up a whole cache line, and an effective compression ratio of 50% when it is placed with a partner in the same cache line. The *effective system-wide compression ratio* is defined as the average effective compression ratio of all cache lines in a compressed cache. It indicates how well a compression algorithm performs for pair-matching based cache compression. This effective compression ratio metric can also be adapted to a segmentation-based approach. For example, for a cache line with 4 fixed-length segments, a line has an effective compression ratio of 25% when it takes up one segment, 50% for two segments, etc. Varying raw compression ratio between 25% and 50% has little impact on the effective cache capacity. For real cache trace data, pair-matching generally achieves a better effective system-wide compression ratio (58%) than line segmentation with four segments per line (62%), and achieves the same compression ratio as line segmentation with eight segments, which would impose substantial hardware overhead. In the following sections, we use the pair-matching effective system-wide compression ratio as a metric for comparing compression algorithms.

III.D. Design Tradeoffs and Validation

In this section, we present several design tradeoffs encountered during the design and implementation of the C-Pack. We also validate C-Pack's effectiveness in pair-matching.

Dictionary design and pattern coding: To decide the optimal dictionary replacement policy, dictionary size, and pattern coding scheme, we evaluated the effective system-wide compression ratio achieved by several configurations. Our test data are real cache data traces collected from full microprocessor, operating system, and application simulation using the Simics simulator [12], running various workloads such as multimedia applications and SPEC CPU2000 benchmarks on a simulated 1 GHz processor. The on-chip L2 cache is set to 8-way associative with a 64 B line size. The candidates for different dictionary parameters and the final selected values are shown in Table II. Note that two/three level coding scheme in Table II refers to one in which the code length is fixed within the same level, but differs from level to level. For example, a two-level code only contains 2-bit and 4-bit codes. With the selected parameters, the effective system-wide compression ratio for a 64 byte cache line is 58.47% on our test data.

Validating C-Pack's Effectiveness In Pair Matching: In order to determine whether the mean and variance of the compression ratio achieved by C-Pack is sufficient for most

TABLE II
DESIGN CHOICES FOR DIFFERENT PARAMETERS

Parameters	Candidates	Selected Candidate
Dictionary replacement policy	(1) First-in first out (FIFO) (2) Least recently used (LRU) (3) Using two FIFO queues to simulate LRU (4) FIFO combined with run-length encoding (RLE)	FIFO - least HW complexity with only 1.32% higher CR than best case
Coding scheme	(1) Huffman coding (2) Two/Three-level coding	Two-level coding due to only up to 0.95% increase in CR with best HW complexity
Dictionary size	Ranging from 16 B to 512 B	64 B - optimal CR for FIFO and low HW cost

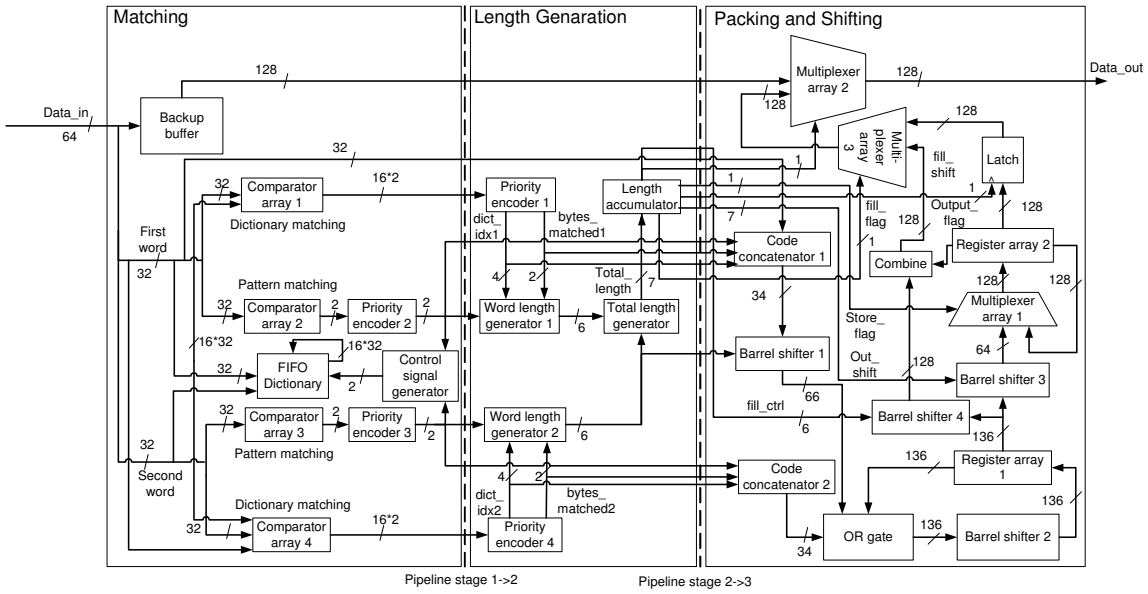

Figure 1. Compressor Architecture.

lines to find partners, we simulated a pair-matching based cache using the cache trace data described above to compute the probability of two cache lines fitting within one uncompressed cache line. The simulated cache size ranges from 64 KB to 2 MB and set associativities of 4 and 8 are considered. A "best fit + best fit" policy is used: for a given compressed cache line, we first try to find the cache line with minimal but sufficient unused space. If the attempt fails, the compressed line replaces one or two compressed lines. With this scheme, we are penalized only when two lines are evicted to store the new line. Experimental results indicate the worst-case probability of requiring the eviction of two lines is 0.55%, i.e., the probability of fitting a compressed line into the cache without additional penalty is at least 99.45%.

IV. C-PACK HARDWARE IMPLEMENTATION

In this section, we provide a detailed description of the proposed hardware implementation of C-Pack. We assume the bus between L1 cache and L2 cache is 128 bits wide [13] and use this as the input data width of both the compressor and decompressor.

IV.A. Compression Hardware

Figure 1 illustrates the hardware compression process. In our descriptions and in the figure, a bold font indicates devices and an italic font indicates signals. The compressor is decomposed into three pipeline stages. This design supports incremental transmission,

i.e., compressed data can be transmitted before the whole data block is compressed, and thereby reduces compression latency.

1) **Pipeline Stage 1**: The first pipeline stage performs pattern matching and dictionary matching on two uncompressed words in parallel. As illustrated in Figure 1, **comparator array 1** matches the first word against patterns "zzzz" and "zzzx" and **comparator array 2** matches it with all dictionary entries, both in parallel. The same is true for the second word. However, during dictionary matching, in addition to the dictionary entries, the second word is also compared with the first word. 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.

2) **Pipeline Stage 2**: This stage computes the total length of the two compressed words and generates control signals based on this length. Depending on the dictionary matching results from Stage 1, **priority encoder 1 and 4** find the dictionary entries with the most matched bytes and their corresponding indices, which are then sent to **word length generator 1 and 2** to 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_partial* and *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 1** should 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. The compressed pair of words are placed into the right location within **register array 1**, denoted by Reg_1[135:0], which is then shifted by *out_shift* using **barrel shifter 3** to align with **register array 2**, denoted by Reg_2[135:0]. **Multiplexer array 1** selects the shifting result as the input to Reg_2[135:0] when *store_flag* is 1, i.e., the number of accumulated compressed bits have exceeded 64 bits, and the original content of Reg_2[135:0] otherwise. Whether **latch** is enabled depends on the number of compressed bits accumulated in Reg_2[135:0] that have not been transmitted. When *output_flag* is 1, indicating that 128 compressed bits have been accumulated in Reg_2[135:0], Reg_2[135:0] is passed to **multiplexer array 1**. **Multiplexer array 3** selects between *fill_shift* and the output of **latch** using *fill_flag*. *Fill_shift* represents 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 2** then 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 3** is selected.

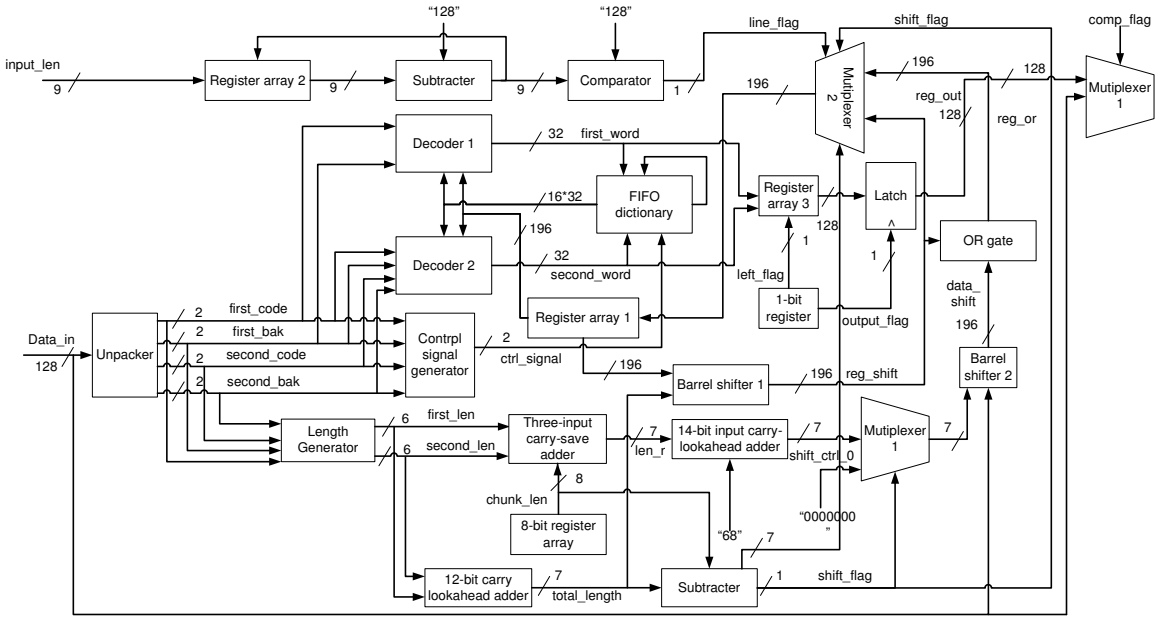

Figure 2. Decompressor Architecture.

IV.B. Decompression Hardware

This section describes the design and optimization of the decompression hardware. Figure 2 illustrates the decompressor architecture. When decompression starts, the codes for the first and the second words are first compared with the static codes described in Table I. If the code indicates a pattern match, the original word is recovered by combining zeroes and unmatched bytes. Otherwise, the original word is recovered by combining bytes from the compressed word and the corresponding dictionary entry (if there is a dictionary match), and inserted into the FIFO dictionary. Note that the dictionary index is determined from successive bits in the compressed word. The decompressed words are then pushed into the output buffer. Meanwhile, the total length of the two compressed words are subtracted from the input length. The decompression results are emitted as soon as 128 decompressed bits have accumulated.

Recall that compressed lines, which may be as long as 512 bits (original line length, i.e., 64 bytes), are 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 one 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 the input buffer (68 bits is the maximum length of two compressed words). The decompressor then shifts in more compressed data using a barrel shifter and concatenates them with the remaining compressed bits. In this way, the decompressor can always fetch two whole compressed words per cycle.

V. 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 [4], Xmatch [14], and FPC [5]. Finally, we describe the implications

of our findings on the feasibility of using C-Pack based cache compression within a microprocessor.

V.A. C-Pack Synthesis Results

We synthesized our design using Synopsys Design Compiler with 180 nm, 90 nm, and 65 nm libraries. Table III presents the resulting performance, area, and power consumption at maximum internal frequency. "Loc" refers to the compressed line locator/arbitrator in a pair-matching compressed cache and "worst case delay" refers to the number of cycles required to compress, decompress, or locate a 64 B line in the worst case. As indicated in Table III, the proposed hardware design achieved a throughput of 80 Gb/s (64 B $\times$ 1.25 GHz) for compression and 76.8 Gb/s (64 B $\times$ 1.20 GHz) for decompression in 65 nm technology. Its area and power consumption overheads are low enough for practical use. The total power consumption of the compressor, decompressor, and compressed line arbitrator at 1 GHz is 48.82 mW (32.63 mW/1.25 GHz + 24.14 mW/1.20 GHz + 5.20 mW/2.00 GHz) in 65 nm technology.

TABLE III
SYNOPSYS DESIGN COMPILER SYNTHESIS RESULTS

Parameters	180 nm			90 nm			65 nm		
	Comp.	Decomp.	Loc.	Comp.	Decomp.	Loc.	Comp.	Decomp.	Loc.
Worst case delay (cycles)	13	8	2	13	8	2	13	8	2
Max. frequency (GHz)	0.38	0.31	0.60	1.09	0.91	1.79	1.25	1.20	2.00
Area (mm^2)	0.34	0.25	0.063	0.076	0.076	0.013	0.043	0.043	0.007
Power consumption at max. internal freq. (mW)	111.78	75.18	110.03	73.88	51.50	15.96	32.63	24.14	5.20

V.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 tested the compression ratios of different algorithms on four distinct test benches: 1) **Cache data** gathered from full-system simulation (Section III-D); 2) **Memory data** by taking a snapshot of the memory contents of a Linux workstation during operation; 3) **Disk data** randomly gathered from the disk of a Linux workstation; and 4) **Swap data** gathered from the swap partition of a Linux workstation.

We tested X-Match, MXT, and FPC on the same set of test benches to determine their compression ratios. We used 64 B block size and dictionary sizes 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 compression results of MXT [15]. The raw compression ratios and effective system-wide compression ratios in a pair-matching scheme are summarized in Table IV. The last row refers to the effective system-wide compression ratios for different algorithms based on the cache trace data set. As indicated in Table IV, raw compression ratio varies from algorithm to algorithm, with X-Match being the best and MXT being the worst on average. The poor raw compression ratios of MXT are mainly due to the limited dictionary size. The same trend is seen in effective system-wide compression ratios, where X-Match has the lowest (best) effective system-wide compression ratio and MXT has the highest. Since the raw compression ratios of X-Match and C-Pack are close to 50%, they achieve almost the same effective system-wide compression ratio.

TABLE IV
COMPRESSION RATIO COMPARISON

Compression Algorithm	MXT	X-Match	FPC	C-Pack
Raw compression ratio (%)				
Cache data	70.88	49.50	63.39	52.10
Memory data	71.66	51.80	62.91	55.40
Disk data	93.36	80.0	90.17	81.96
Swap data	69.52	48.40	63.26	51.26
System-wide compression ratio (%)				
Cache data	75.55	57.97	64.28	58.47

V.C. Comparison of Hardware Performance

This subsection compares the decompression latency, peak frequency, and area of C-Pack hardware to that of to MXT, X-Match, and FPC. Power consumption comparisons are excluded because they are not reported for the alternative compression algorithms. Decompression latency is defined as the time to decompress a 64 B cache line.

V.C.1) Comparing C-Pack with MXT : MXT has been implemented in a memory controller chip operating at 133 MHz using 0.25 µm CMOS ASIC technology [16]. The decompression rate is 8 B/cycle with 4 decompression engines. We scale the frequency up to 511 MHz by a factor of (250/65), i.e., using constant electrical field scaling, to reflect the move to 65 nm technology. 511 MHz is far below a modern processor frequency. We assume an on-chip counter/divider is available to clock the MXT decompressor. However, decompressing a 64 B cache line will take 16 processor cycles in a 1 GHz processor, twice the time for C-Pack. The area cost of MXT is not reported.

V.C.2) Comparing C-Pack with X-Match: X-Match has been implemented using 0.25 µm field programmable gate array (FPGA) technology. The compression hardware achieved a maximum frequency of 50 MHz with a throughput of 200 MB/s. To the best of our knowledge, the design was not synthesized using a flow suitable for microprocessors. Therefore, it is difficult to directly compare the performance of C-Pack and X-Match.

V.C.3) Comparing C-Pack with FPC: FPC has not been implemented on a hardware platform; no area or peak frequency numbers are reported. To estimate the area cost of FPC, we observe that the FPC compressor and decompressor are decomposed into multiple pipeline stages (as described in its tentative hardware design [5]). Each of these stages imposes area overhead. For example, assuming each 2-to-1 multiplexer takes 5 gates, the fourth stage of the FPC decompression pipeline takes approximately 290 K gates or 0.31 mm^2 in 65 nm technology, more than the total area of our compressor and decompressor. Although this work claims that time-multiplexing two sets of barrel shifters could help reduce area cost, doing so increases the overall latency of decompressing a cache line to 12 cycles, instead of the claimed 5 cycles. In contrast, our hardware implementation achieves much better compression ratio and a comparable worst case delay in terms of cycles at a high clock frequency, at an area cost of 0.043 mm^2 and 0.043 mm^2 in 65 nm technology for the compressor and decompressor.

V.D. Implications on Claims in Prior Cache Compression Work

Many prior publications on cache compression assume the existence of lossless algorithms supporting a consistent good compression ratio on small (e.g., 64-byte) blocks and allowing decompression within a few microprocessor clock cycles (e.g., 8 ns) with low area and power consumption overheads [6, 8, 9]. Some publications assume that Ziv–Lempel compression algorithm based hardware would be sufficient to meet the requirements [2]. As shown in Section V-C1, these assumptions are incorrect. Past work also placed too much weight on cache line compression ratio instead of effective system-wide compression ratio. As a result, compression algorithms producing lower compressed line sizes were favored. However, the hardware overhead of permitting arbitrary locations of these compressed lines prevents arbitrary placement, resulting in system-wide compression ratios much poorer than predicted by line compression ratio. In fact, the compression ratio metric of merit for cache compression algorithms should be effective system-wide compression ratio, not average line compression ratio. C-Pack was designed

to optimize performance, area, and power consumption under a constraint on effective system-wide compression ratio.

C-Pack meets or exceeds the requirements assumed in former microarchitectural research on cache compression. It therefore provides a proof of concept supporting the system-level conclusions drawn in much previous microarchitectural work on cache compression. Many prior system-wide cache compression results hold, provided that they use a compression algorithm with characteristics similar to C-Pack.

VI. Conclusions

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 matching. 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 58%, and permits a hardware implementation that holds decompression latency to 6.67 ns in 65 nm process technology. 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," in *Proc. Int. Symp. Computer Architecture*, June 2004.

[2] E. G. Hallnor and S. K. Reinhardt, "A compressed memory hierarchy using an indirect index cache," in *Proc. 3rd workshop on 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] B. Tremaine, et al., "IBM memory expansion technology," *IBM J. of Research and Development*, vol. 45, no. 2, pp. 271–285, Mar. 2001.

[5] 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.

[6] J.-S. Lee, et al., "Design and evaluation of a selective compressed memory system," in *Proc. Int. Conf. Computer Design*, Oct. 1999.

[7] N. S. Kim, T. Austin, and T. Mudge, "Low-energy data cache using sign compression and cache line bisection," in *Proc. Wkshp. on Memory Performance Issues*, May 2002.

[8] K. S. Yim, J. Kim, and K. Koh, "Performance analysis of on-chip cache and main memory compression systems for high-end parallel computers," in *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications*, June 2004, pp. 469–475.

[9] N. R. Mahapatra, et al., "A limit study on the potential of compression for improving memory system performance, power consumption, and cost," *J. Instruction-Level Parallelism*, July 2005.

[10] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," *SIGARCH Comput. Archit. News*, pp. 74–85, May 2005.

[11] L. Yang, H. Lekatsas, and R. P. Dick, "High-Performance Operating System Controlled Memory Compression," in *Proc. Design Automation Conf.*, July 2006, pp. 701–704.

[12] "Simics," http://www.virtutech.com.

[13] T. Lyon, et al., "Data cache design considerations for the Itanium 2 processor," in *Proc. Int. Conf. Computer Design*, Sept. 2002.

[14] J. Nunez and S. Jones, "Gbit/s lossless data compression hardware," *IEEE Trans. VLSI Systems*, vol. 11, no. 3, pp. 499–510, June 2003.

[15] P. Franaszek, J. Robinson, and J. Thomas, "Parallel compression with cooperative dictionary construction," in *Proc. Data Compression Conf.*, Apr. 1996.

[16] R. B. Tremaine, et al., "Pinnacle: IBM MXT in a memory controller chip," in *Proc. Int. Symp. Microarchitecture*, Apr. 2001.