C-Pack: A High-Performance Microprocessor Cache Compression Algorithm

Xi Chen∗ Lei Yang§ Robert P. Dick∗ Li Shang† Haris Lekatsas‡

∗ EECS Dept.
University of Michigan
Ann Arbor, MI
{chexi, dickrp}
@umich.edu

§ Google Inc.
Mountain View, CA
leiyang@google.com

† ECEE Dept.
University of Colorado
Boulder, CO
li.shang@colorado.edu

‡ Vorras Corporation
Princeton, NJ
lekatsas@vorras.com

***Abstract*—Microprocessor designers have been torn between tight constraints on the amount of on-chip cache memory and the high latency of off-chip memory, such as dynamic random access memory. Accessing off-chip memory generally takes an order of magnitude more time than accessing on-chip cache, and two orders of magnitude more time than executing an instruction. Computer systems and microarchitecture 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 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 work, 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. We reduced the proposed algorithm to a register transfer level hardware implementation, permitting performance, power consumption, and area estimation. Experiments comparing our work to previous work are described.

***Index Terms*–Cache compression, effective system-wide compression ratio, 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. 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 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 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, past work did not demonstrate whether the proposed compression/decompression hardware is appropriate for cache compression, considering the performance, area, and power consumption requirements. This analysis is also essential to permit the performance impact of using cache compression to be estimated.

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

Researchers have commonly made assumptions about the implications of using existing compression algorithms for cache compression and the design of special-purpose cache compression hardware.

A number of researchers have assumed the use of general-purpose main memory compression hardware for cache compression. IBM's MXT (Memory Expansion Technology) [6] is a hardware memory compression/decompression technique that improves the performance of servers via increasing the usable size of off-chip main memory. Data are compressed in main memory and decompressed when moved from main memory to the off-chip shared L3 cache. Memory management hardware dynamically allocates storage in small sectors to accommodate storing variable-size compressed data block without the need for garbage collection. IBM reports compression ratios (compressed size divided by uncompressed size) ranging from 16% to 50%. X-Match is a dictionary-based compression algorithm that

This work was done when Xi Chen, Lei Yang, and Robert P. Dick were at Northwestern University.

has been implemented on an FPGA [7]. It matches 32-bit words using a content addressable memory that allows partial matching with dictionary entries and outputs variable-size encoded data that depends on the type of match. To improve coding efficiency, it also uses a move-to-front coding strategy and represents smaller indices with fewer bits. Although appropriate for compressing main memory, such hardware usually has a very large block size (1 KB for MXT and up to 32 KB for X-Match), which is inappropriate for compressing cache lines. It is shown that for X-Match and two variants of Lempel-Ziv algorithm, i.e., LZ1 and LZ2, the compression ratio for memory data deteriorates as the block size becomes smaller [7]. For example, when the block size decreases from 1 KB to 256 B, the compression ratio for LZ1 and X-Match increase by 11% and 3%. It can be inferred that the amount of increase in compression ratio could be even larger when the block size decreases from 256 B to 64 B. In addition, 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. Frequent pattern compression (FPC) [8] compresses cache lines at the L2 level by storing common word patterns in a compressed format. Patterns are differentiated by a 3-bit prefix. Cache lines are compressed to predetermined sizes that never exceed their original size to reduce decompression overhead. Based on logical effort analysis [9], for a 64-byte cache line, compression can be completed in three cycles and decompression in five cycles, assuming 12 fan-out-four (FO4) gate delays per cycle. To the best of our knowledge, there is no register-transfer-level hardware implementation or FPGA implementation of FPC, and therefore its exact performance, power consumption, and area overheads are unknown. Although the area cost for FPC [8] 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. This will be examined in detail in Section VI-C3.

In short, assuming desirable cache compression hardware with adequate performance and low area and power overheads is common in cache compression research [2], [10]–[15]. It is also understandable, as the microarchitecture community is more interested in microarchitectural applications than compression. However, without a cache compression algorithm and hardware implementation designed and evaluated for effective system-wide compression ratio, hardware overheads, and interaction with other portions of the cache compression system, one can not reliably 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) C-Pack targets on-chip cache compression. It permits a good compression ratio even when used on small cache lines. The performance, area, and power consumption overheads are low enough for practical use. This contrasts with other schemes such as X-match which require complicated hardware to achieve an equivalent effective system-wide compression ratio [7].
2) We are the first to fully design, optimize, and report performance and power consumption of a cache compression algorithm when implemented using a design flow appropriate for on-chip integration with a microprocessor. Prior work in cache compression does not adequately evaluate the overheads imposed by the assumed cache compression algorithms.
3) We demonstrate when line compression ratio reaches 50%, further improving it has little impact on effective system-wide compression ratio.
4) C-Pack is twice as fast as the best existing hardware implementations potentially suitable for cache compression. For FPC to match this performance, it would require at least 8× the area of C-Pack.
5) We address the challenges in design of high-performance cache compression hardware while maintaining some generality, i.e., our hardware can be easily adapted to other high-performance lossless compression applications.

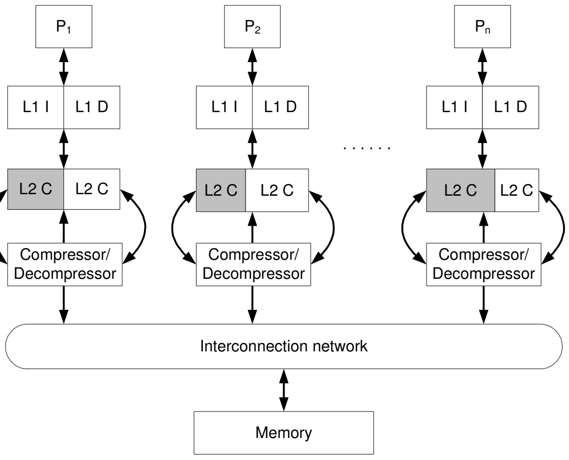

Fig. 1. System architecture in which cache compression is used.

III. 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.

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

IV. C-Pack Compression Algorithm

This section gives an overview of the proposed C-Pack compression algorithm. We first briefly describe the algorithm and several important features that permit an efficient hardware implementation, many of which would be contradicted for a software implementation. We also discuss the design trade-offs and validate the effectiveness of C-Pack in a compressed-cache architecture.

A. Design Constraints and Challenges

We first point out several design constraints and challenges particular to the cache compression problem:

TABLE I
PATTERN ENCODING FOR C-PACK

Code	Pattern	Output	Length (b)	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	(1101)B	12	7.3
1110	mmmx	(1110)bbbbB	16	7.2

1) Cache compression requires hardware that can de/compress a word in only a few CPU clock cycles. This rules out software implementations and has great influence on compression algorithm design.
2) Cache compression algorithms must be lossless to maintain correct microprocessor operation.
3) The block size for cache compression applications is smaller than for other compression applications such as file and main memory compression. Therefore, achieving a low compression ratio is challenging.
4) The complexity of managing the locations of cache lines after compression influences feasibility. Allowing arbitrary, i.e., bit-aligned, locations would complicate cache design to the point of infeasibility. A scheme that permits a pair of compressed lines to fit within an uncompressed line is advantageous.

B. *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-chip caches, e.g., L2 caches. Its design was strongly influenced by prior work on pattern-based partial dictionary match compression [16]. However, this prior work was designed for software-based main memory compression and did not consider hardware implementation.

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 dictionary supports partial word matching as well as full 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 file mentioned in Section IV-D. The 'Pattern' column describes frequently appearing patterns, where '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.

The C-Pack compression and decompression algorithms 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 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 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 compression results for several different input words. In each output, the code and the dictionary index, if any, are enclosed in parentheses. Although we used a 4-word dictionary in Figure 3 for illustration, the dictionary size is set to 64 B in our implementation. Note that the dictionary is updated after each word insertion, which is not shown in Figure 3.

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 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.

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. For example, matching against multiple patterns can be prohibitively expensive for software implementations when the number of patterns or dictionary entries is large. C-Pack's inherently parallel design allows an efficient hardware implementation, in which pattern matching, dictionary matching, and processing multiple words are all done simultaneously. In addition, we chose various design parameters such as dictionary replacement policy and coding scheme to reduce hardware complexity, even if our choices slightly degrades the effective system-wide compression ratio. Details are described in Section IV-D.

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. This improves compression ratio compared to the more naïve approach of not checking with the first word. Therefore, we can compress two words in parallel without compression ratio degradation.

C. *Effective System-Wide Compression Ratio and Pair-Matching Compressed Line Organization*

Compressed cache organization is a difficult task because different compressed cache lines may have different lengths. Researchers have proposed numerous variants of line segmentation techniques [1], [2], [10] 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. Hallnor et al. [2] proposed IIC-C, i.e., indirect index cache with compression. The proposed 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. However, the authors do not report worst-case latency. Lee et al. [10] proposed selective compressed caches using a similar idea. 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

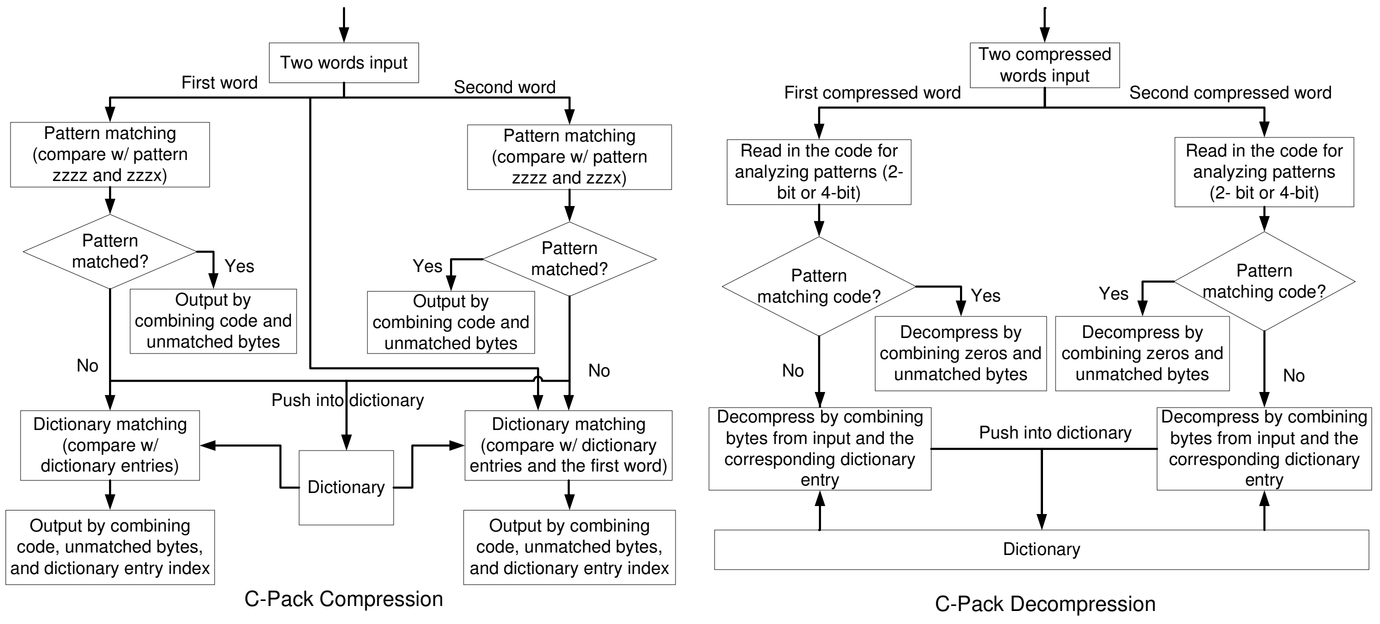
Two words input
First word
Second word
Pattern matching (compare w/ pattern zzzz and zzzx)
Pattern matched?
Yes
No
Output by combining code and unmatched bytes
Push into dictionary
Dictionary matching (compare w/ dictionary entries)
Dictionary matching (compare w/ dictionary entries and the first word)
Dictionary
Output by combining code, unmatched bytes, and dictionary entry index
C-Pack Compression
Two compressed words input
First compressed word
Second compressed word
Read in the code for analyzing patterns (2-bit or 4-bit)
Pattern matching code?
Decompress by combining zeros and unmatched bytes
Decompress by combining bytes from input and the corresponding dictionary entry
C-Pack Decompression

Fig. 2. C-Pack (a) compression and (b) decompression.

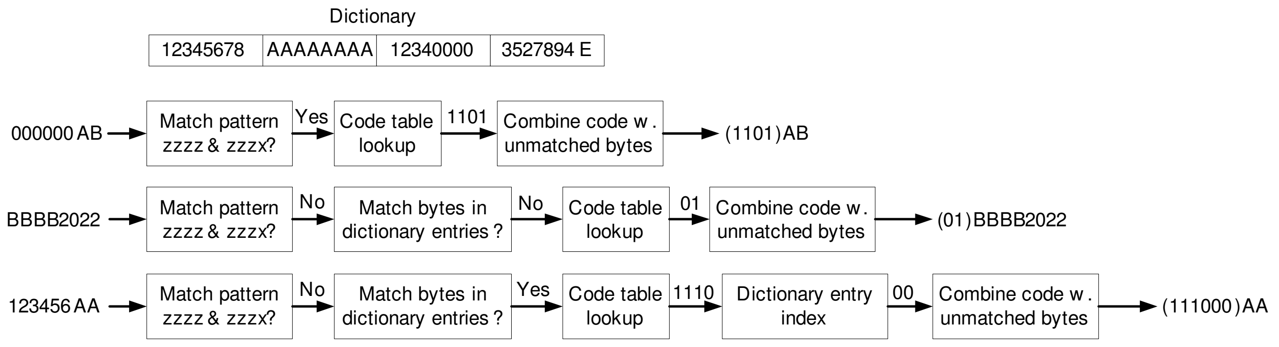

Fig. 3. Compression examples for different input words.

occur. The hardware overhead and worst-case access latency for addressing a compressed cache line is not discussed. Alameldeen et al. [1] proposed decoupled variable-segment cache, where the L2 cache dynamically allocates compressed or uncompressed lines depending on whether compression eliminates a miss or incurs an unnecessary decompression overhead. However, this approach has significant performance and area overhead, discussed later in this section.

We propose the idea of *pair-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 but also 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%. Note that successful placement of a line does not require that it have a compression ratio smaller than 50%. It is only necessary that the line, combined with a "partner line" be as small as an uncompressed line. 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. More specifically, line extraction in a pair-matching based cache only requires parallel address tag match and takes a single cycle to accomplish. For line insertion, neither LRU list search nor set compaction is involved.

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. A *size* 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. For a 64-byte line in a 32-bit architecture the tag is no longer than 32 bits, hence the worst-case overhead is less than 32 (tag) + 1 (valid) + 2 $\times$ 7 (size) bits, i.e., 6 bytes.

As we can see in Figure 4, the compressed line locator uses the bitlines for valid bits and compressed line sizes to locate a newly compressed line. Note that only one compressed line locator is required for the entire compressed cache. This is because for a given address, only the cache lines in the set which the specific address is mapped to are activated thanks to the set decoder. Each bitline is connected to a sense amplifier, which usually requires several gates [17], for signal amplification and delay reduction. The total area overhead is approximately 500 gates plus the area for the additional

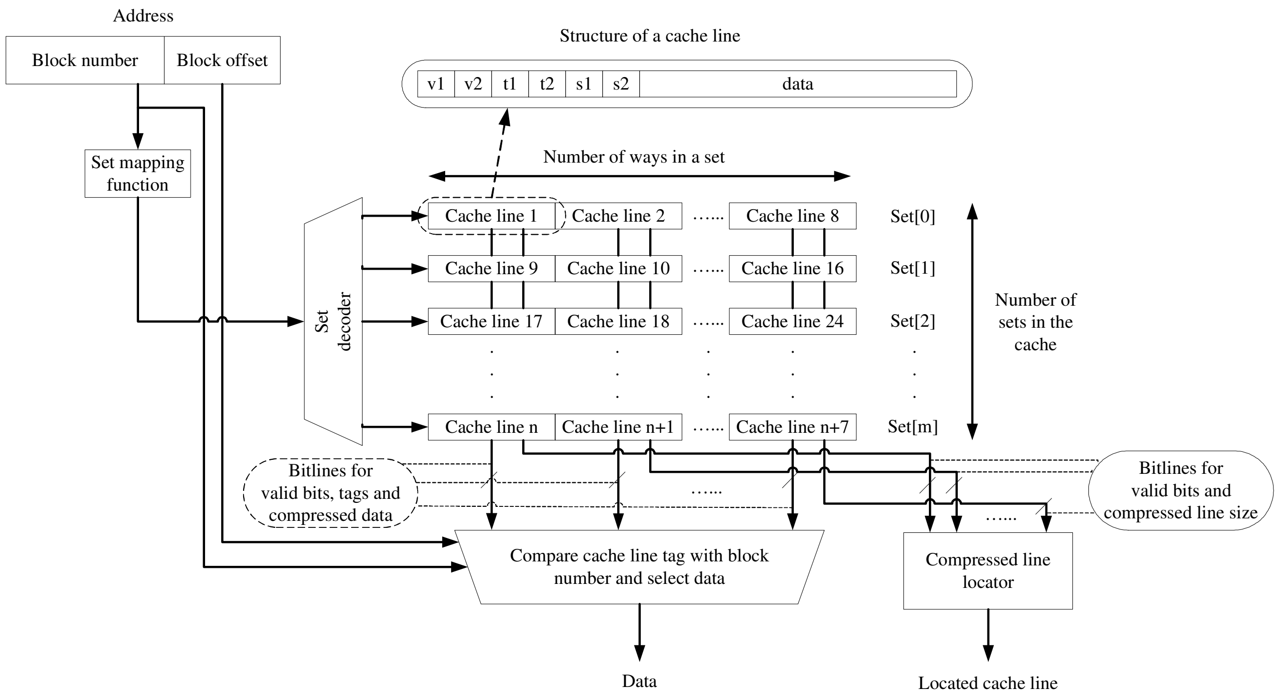

Fig. 4. Structure of a pair-matching based cache.

bitlines, compared to an uncompressed cache.

Based on the "pair-matching" concept, 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. Note that when a compressed line is placed together with its partner without evicting any compressed lines, its partner's effective compression ratio decreases to 50%. 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. For example, for a cache line with 4 fixed-length segments, a compressed line has an effective compression ratio of 25% when it takes up one segment, 50% for two segments, and so on. Varying raw compression ratio between 25% and 50% has little impact on the effective cache capacity of a four-part segmentation based technique. Figure 5 illustrates the distribution of raw compression ratios for different cache lines derived from real cache data. The x-axis shows different compression ratio intervals and y-axis indicates the percentage of all cache lines in each compression ratio interval. 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 the same compression ratio as line segmentation with eight segments, which would impose substantial hardware overhead.

We now compare the performance and hardware overhead of pair-matching based cache with decoupled variable-segment cache. The hardware overhead can be divided into two parts: tag storage overhead and compressed line locator overhead. For a 512 KB L2 cache with a line size of 64 bytes, the tag storage overhead is 7.81% of the total cache size for both decoupled variable-segment cache and pair-matching based cache. The area overhead of the compressed line locator is significant in a decoupled variable-segment cache. During line insertion, a newly inserted line may be larger than the LRU line plus the unused segments. In that case, prior work proposed replacing two lines by replacing the LRU line and searching the LRU list to find the least-recently used line that ensures enough space for the newly arrived line [1]. However, maintaining and updating the LRU list will result in great area overhead. Moreover, set compaction may be required after line insertion to maintain the contiguous storage invariant. This can be prohibitively expensive in terms of area cost because it may require reading and writing all the set's data segments. Cache compression techniques that assume it are essentially proposing to implement kernel memory allocation and compaction in hardware [18]. However, for pair-matching based cache, the area of compressed line locator is negligible (less than 0.01% of the total cache size).

The performance overhead comes from two primary sources: addressing a compressed line and compressed line insertion. The worst-case latency to address a compressed line in a pair-matching based cache is 1 cycle. For a 4-way associative decoupled variable-segment cache with 8 segments per line, each set contains 8 compression information tags and 8 address tags because each set is constrained to hold no more than eight compressed lines. The compression information tag indicates (1) whether the line is compressed and (2) the compressed size of the line. Data segments are stored contiguously in address tag order. In order to extract a compressed line from a set, eight segment offsets are computed in parallel with the address tag match. Therefore, deriving the segment offset for the last line in the set requires summing up all the previous 7 compressed sizes, which incurs a significant performance overhead. In addition, although the cache array may be split into two banks to reduce line extraction latency, addressing the whole compressed line may still take 4 cycles in the worst case. To insert a compressed line, the worst-case latency is 2 cycles for pair-matching based cache with a peak frequency of more than 1 GHz. The latency of a decoupled variable-segment cache is not reported [1]. However, as explained in the previous paragraph, LRU list searching and set compaction introduce great performance overhead. Therefore, we recommend pair-matching and use the pair-matching effective system-wide compression ratio as a metric for comparing different compression algorithms.

D. Design Tradeoffs and Details

In this section, we present several design tradeoffs encountered during the design and implementation of C-Pack. We also validate

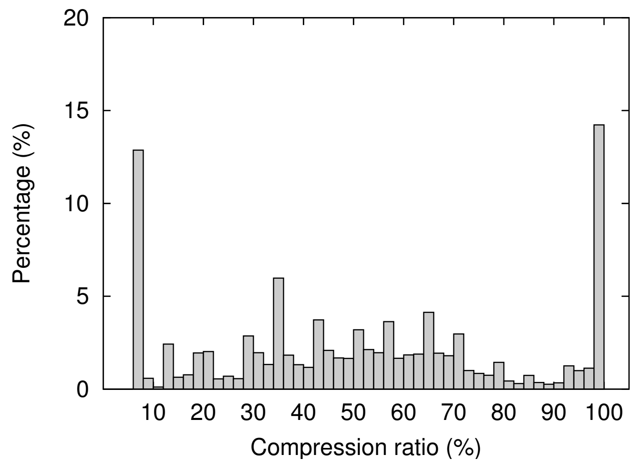

Fig. 5. Distribution of compression ratios.

TABLE II
EFFECTIVE SYSTEM-WIDE COMPRESSION RATIOS FOR C-PACK

Effective system-wide compression ratio (%)							
Dictionary size (B)		16	32	64	128	256	512
FIFO	Huffman	58.14	57.56	57.46	57.46	57.66	57.73
	Two-level	58.81	58.47	57.95	58.30	58.29	58.68
LRU	Huffman	58.13	57.70	57.61	57.91	58.07	58.17
	Two-level	58.97	58.54	58.38	58.73	58.72	58.92
Two FIFOs	Huffman	58.05	57.61	57.48	57.46	57.66	57.73
	Two-level	58.71	58.49	57.97	58.30	58.29	58.68
RLE w/ LRU	Huffman	57.20	56.68	56.63	56.73	56.75	56.87
	Three-level	57.66	57.31	57.08	57.11	57.35	57.44

C-Pack's effectiveness in pair-matching.

Dictionary design and pattern coding: We evaluated the impact of different parameters during algorithm and hardware design and optimization, including dictionary replacement policy, dictionary size, and pattern coding scheme. The effective system-wide compression ratio of C-Pack was evaluated based on cache trace data collected from a full microprocessor, operating system, and application simulation of various workloads, e.g., media applications and SPEC CPU2000 benchmarks on a simulated 1 GHz processor. The cache is set to 8-way associative with a 64 B line size. The evaluation results are shown in Table II. The candidates for different parameters and the final selected values are shown in Table III, in which the first column shows various parameters, the second column shows the corresponding candidates for each parameter, and the third column shows the selected values. Note that the two or three level coding scheme in Table III 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 can contain 2-bit and 4-bit codes only.

The criteria of choosing design parameters can be explained as follows. For design parameters that have only have small impact on design complexity, we choose values to optimize compression ratio. For design parameters that have great impact on design complexity, we choose the simplest design when the design complexity varies a lot as the current design parameter changes. For replacement policy, hardware LRU algorithm maintains a table of $n \times n$ bits given a dictionary size of n words. The kth row and kth column is updated on every access to word k, which complicates hardware design. Using 2 FIFO queues to simulate LRU essentially doubles the dictionary size. In addition, moving the words between two queues adds additional hardware complexity. Combining FIFO with RLE also complicates hardware design because special care is needed when the current word has been encountered several times before. For all of the replacement policies except the simplest FIFO, dictionary updates depend not only on the current word but also on recently processed words. This complicates algorithm design because decompression of the second word depends on the dictionary updates due to decompression of the first word. A replacement policy that records the exact sequence of recently processed words would incur a large area overhead during decompression hardware design. This is also true when selecting a coding scheme because for Huffman coding, there is a larger variance in the length of a compressed word, thus making it more difficult to determine the location of the second compressed word. Therefore, choosing Huffman coding also negatively affects the decompression hardware design. However, varying the dictionary size only affects the area by a negligible amount (in the order of several hundred gates). Moreover, it has no impact on the structure of compression or decompression hardware. Therefore, we choose the dictionary size that minimizes the compression ratio. With the selected parameters, the effective system-wide compression ratio for a 64 byte cache line is 58.47% for our test data.

Trade-Off Between Area and Decompression Latency: Decompression latency is a critically important metric for cache compression algorithms. During decompression, the processor may stall while waiting for important data. If the decompression latency is high, it can undermine potential performance improvements. It is possible to use increased parallelism to increase the throughput of a hardware implementation of C-Pack, at the cost of increased area. For example, decompressing the second word by combining bytes from the input and the dictionary is challenging because the locations of bytes that compose the decompression output depend on the codes of the first word and second word. Given that each code can have 6 possible values, as indicated in Table I, there are 36 possible combinations of the two codes, each of which corresponds to a unique combination of bytes from the input and dictionary. If we double the number of words processed in one cycle, i.e., 4 words per cycle, there can be 1,296 possible combinations for decompressing the fourth word, thereby dramatically increasing the area cost. To achieve a balance between area and throughput, we decided to compress or decompress two words per cycle.

Evaluating Compression Ratio for C-Pack Pair-Matching: In order to determine whether the mean and variance of the compression ratio achieved by C-Pack is sufficient for most 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 the set associativity ranges from 4 to 8. We adopt a "best fit + best fit" policy: 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. This scheme is penalized only when two lines are evicted to store the new line. Experimental results indicate that 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%. We implemented and synthesized this line replacement policy in hardware. The delay and area cost are reported in Table IV.

V. C-PACK HARDWARE IMPLEMENTATION

In this section, we provide a detailed description of the proposed hardware implementation of C-Pack. Note that although the proposed compressor and decompressor mainly target on-line cache compression, they can be used in other data compression applications, such as memory compression and network data compression, with few or no modifications.

A. Compression Hardware

This section describes the design and optimization of the proposed compression hardware. It first gives an overview of the proposed compressor architecture, and then discusses the data flow among different pipeline stages inside the compressor.

TABLE III
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 only 1.32% higher CR than best case
Coding scheme	(1) Huffman coding (2) Two or Three-level coding	Two-level coding due to best HW complexity with at most 0.95% increase in CR given the same dictionary size and replacement policy
Dictionary size	Ranging from 16 B to 512 B	64 B – optimal CR for FIFO and low HW cost

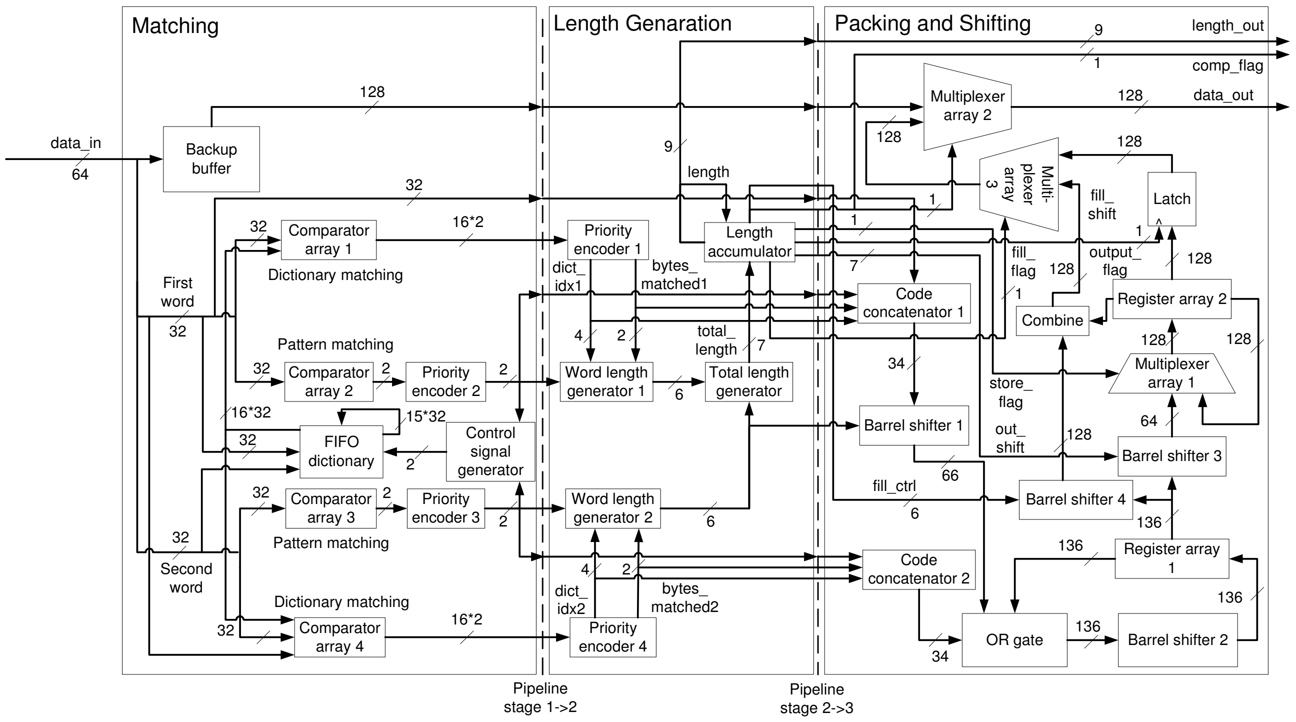

Fig. 6. Compressor architecture.

Compressor Architecture: Figure 6 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 6, **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.
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 4** find the dictionary entries with the most matched bytes and their corresponding indices, which are then sent to **word length generators 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.

Placing the compressed pair of words into the right location within **register array 1**, denoted by $\text{Reg}_1[135:0]$, 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 1** should be shifted to fit 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 concatenators 1 and 2** in the lower bits, with the higher unused bits set to zero. Reg_1[135:0] is shifted by *total_length* using **barrel shifter 2**, with the shifting result denoted by Reg_{1s}[135:0]. At the same time, A[33:0] is shifted using **barrel shifter 1** by 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 that Reg_1[135:68] only has one input source, i.e., Reg_{1s}[135:68], because the maximum total output length is 68. However, Reg_1[67:2] can have multiple input sources: B, S, and Reg_{1s}. For example, Reg_1[4] may come from B[4], S[2], or Reg_{1s}[0]. To obtain the input to Reg_1[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, Reg_1[135:0] is 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 has 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.

B. Decompression Hardware

This section describes the design and optimization of the proposed decompression hardware. We describe the data flow inside the decompressor and point out some challenges specific to the decompressor design. We also examine how to integrate data prefetching with cache compression.

1) Decompressor Architecture: Figure 7 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 tranmission, 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 **unpacker** 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.
2) **Word Decompressing: Decoders 1 and 2** compare the codes of the first and second word against the static codes in Table I to derive the patterns for the two words, which are then decompressed by combining zero bytes, bytes from **FIFO dictionary**, and bytes from **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 1** if *len_r* is less than 68. Meanwhile, **register array 1** is shifted by *total_length* (the sum of *first_len* and *second_len*) to make space for the new incoming compressed bits.

2) Decompressor & Data Prefetching: Data prefetching [19] has been proposed as a technique to hide data access latency. It anticipates future cache misses and fetches the associated data into the cache in advance of expected memory references. In order to integrate data prefetching with cache compression, resource conflicts must be taken into account: a processor may request a line in the compressed region of the L2 cache while the corresponding decompressor prefetches data from the compressed region into the uncompressed region. Although we can disable data prefetching from the compressed region of an L2 cache, i.e., only allowing prefetching data from off-chip memory into the uncompressed region of L2, this may result in higher average data prefetching latency and lower performance benefit compared to a scheme where prefetching from both off-chip memory and compressed region of L2 caches are enabled. One possible solution is to add an extra decompressor for each processor. This enables simultaneously serving processor requests and prefetching data from the compressed region into the uncompressed region.

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 [7], and FPC [20]. 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 Synopsys Design Compiler with 180 nm, 90 nm, and 65 nm libraries. Table IV 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 IV, the proposed hardware design achieves a throughput of 80 Gb/s (64 B $\times$ 1.25 GHz)

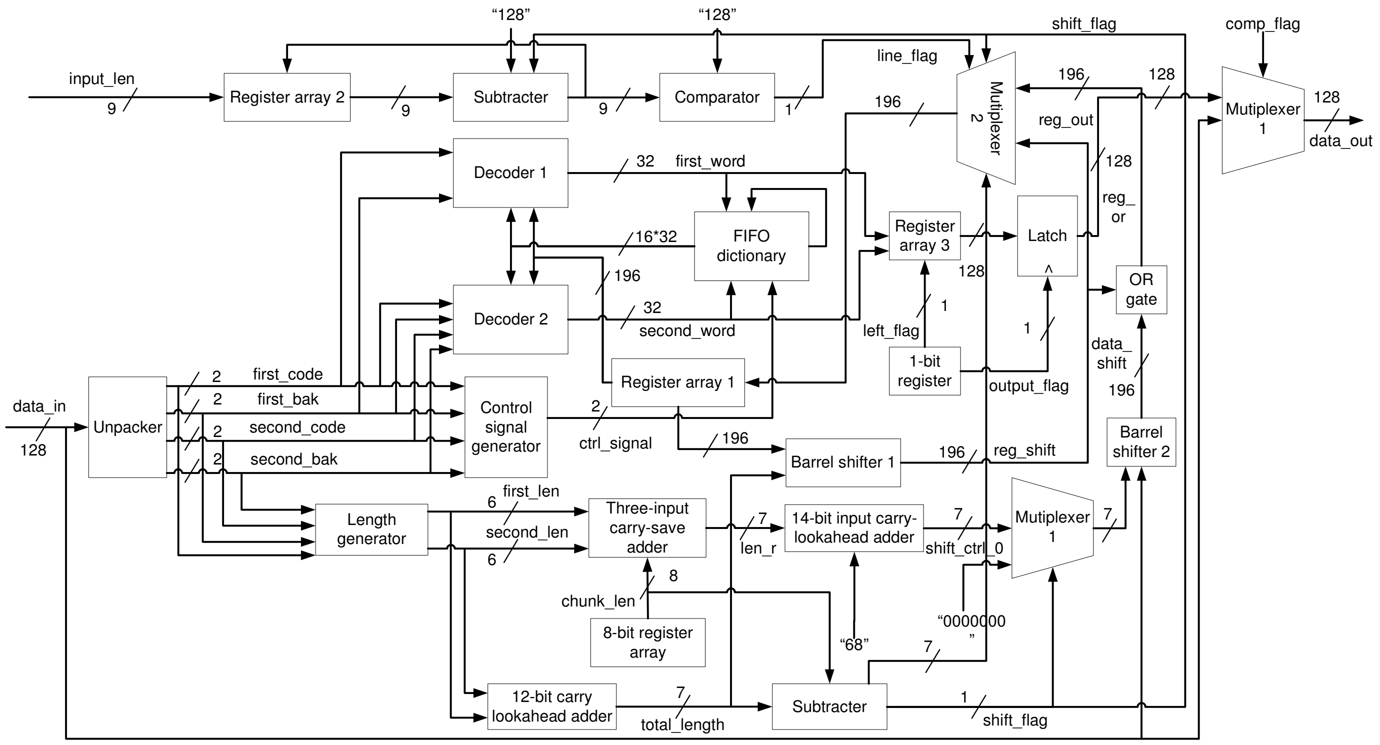

Fig. 7. Decompressor architecture.

TABLE IV
SYNOPSYS DESIGN COMPILER SYNTHESIS RESULTS

Parameters	180 nm			90 nm			65 nm		
	Compressor	Decompressor	Loc.	Compressor	Decompressor	Loc.	Compressor	Decompressor	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

for compression and 76.8 Gb/s (64 B $\times$ 1.20 GHz) for decompression in a 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 a 65 nm technology. This is only 7% of the total power consumption of a 512 KB cache with a 64 B block size at 1 GHz in 65 nm technology, derived using CACTI 5 [21].

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 algorithms 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 four cache data traces gathered from a full system simulation of various workloads from the Mediabench [22] 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 [23]. The raw compression ratios and effective system-wide compression ratios in a pair-matching scheme are summarized in Table V. Each row shows the raw compression ratios and effective system-wide compression ratios using different compression algorithms for an application. As indicated in Table V, 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 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. Since the raw compression ratios of X-Match and C-Pack are close to 50%, they achieve better effective system-wide compression ratios than MXT and FPC. On average, C-Pack's system-wide compression ratio is 2.76% worse than that of X-Match, 6.78% better than that of FPC, and 10.3% better than that of MXT.

C. Comparison of Hardware Performance

This subsection compares the decompression latency, peak frequency, and area of C-Pack hardware to that of 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.

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 [24]. The decompression rate is 8 B/cycle with 4 decompression engines. We scale the frequency up

TABLE V
COMPRESSION RATIO COMPARISON

	Raw compression ratio (%)				System-wide compression ratio (%)			
Benchmark	MXT	FPC	X-Match	C-Pack	MXT	FPC	X-Match	C-Pack
mpeg2	70.88	63.39	49.50	52.10	75.55	64.28	57.97	58.47
mesa	49.50	69.81	42.80	51.97	60.50	66.18	53.59	55.80
art	57.69	59.27	46.60	51.74	64.84	66.67	60.63	61.40
twolf	84.09	80.73	70.20	77.40	85.90	75.60	62.37	69.92
Average	65.54	68.30	52.28	58.30	71.70	68.18	58.64	61.40

to 511 MHz, i.e., its estimated frequency based on constant electrical field scaling if implemented in a 65 nm technology. 511 MHz is below a modern high-performance 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.

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, we ported our design for C-Pack for synthesis to the same FPGA used for X-Match [7] in order to compare the peak frequency and the throughput. Evaluation results indicate that our C-Pack implementation is able to achieve the same peak frequency as X-Match and a throughput of 400 MB/s, i.e., twice as high as X-Match's throughput. Note that in practical situations, C-Pack should be implemented using an ASIC flow due to performance requirement for cache compression.

3) Comparing C-Pack with FPC : FPC has not been implemented on a hardware platform. Therefore, 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 [20]. 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, our analysis suggest that doing so would increase 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 at a high clock frequency, at an area cost of 0.043 mm^2 compressor and 0.043 mm^2 decompressor in 65 nm technology.

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 [10], [12], [13]. Some publications assume that existing Lempel–Ziv compression algorithm based hardware would be sufficient to meet these requirements [2]. As shown in Section VI-C1, these assumptions are not supported by evidence or analysis. Past work also placed too much weight on cache line compression ratio instead of effective system-wide compression ratio (defined in Section IV-C). 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. Alameldeen et al. proposed *segmented compression ratio*, an idea similar to system-wide compression ratio. However, segmented compression ratio is only defined for a segmentation-based approach with fixed-size segments. Effective system-wide compression ratio generalizes this idea to handle both fixed size segments (segmentation-based schemes) and variable length segments (pair-matching based schemes). 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 of this research. Many prior system-wide cache compression results hold, provided that they use a compression algorithm with characteristics similar to C-Pack.

VII. 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 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 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.

VIII. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards CCF-0702761 and CNS-0347941. The authors would like to acknowledge Alaa Alameldeen at Intel Corporation for his help understanding his cache compression research results.

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. 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. Núñez 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.

[9] I. Sutherland, R. F. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, 1st ed. Morgan Kaufmann, 1999.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] P. Pujara and A. Aggarwal, "Restrictive compression techniques to increase level 1 cache capacity," in *Proc. Int. Conf. Computer Design*, Oct. 2005.

[16] 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.

[17] J. M. Rabaey, *Digital Integrated Circuits*. Prentice-Hall, NJ, 1998.

[18] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. O'Reilly & Associates, Inc., Dec. 2002.

[19] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174–199, June 2000.

[20] A. R. Alameldeen, "Using compression to improve chip multiprocessor performance," Computer Sciences Department, University of Wisconsin-Madison," Ph.D. dissertation, Mar. 2006.

[21] "CACTI: An integrated cache access time, cycle time, area, leakage, and dynamic power model," http://quid.hpl.hp.com:9082/cacti/.

[22] C. Lee, M. Potkonjak, and W. H. M. Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," http://cares.icsl.ucla.edu/MediaBench.

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

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

Xi Chen received his B.S. degree from Tsinghua University and his M.Sc. degree from Northwestern University. He is currently a Ph.D. student at University of Michigan's Department of Electrical Engineering and Computer Science. Chen has published in the areas of embedded system security, data compression, and user-driven power management.

Lei Yang received her Ph.D. degree in 2008 from Northwestern University, Dept. of Electrical Engineering and Computer Science. She also holds a Bachelor's degree from Peking University and a Master's degree from Northwestern University. In 2004 and 2005, Lei worked as a visiting researcher at NEC Laboratories America in Princeton, New Jersey. In 2007, she worked as a software engineering intern at Google, and took a full time position with Google in 2008. Lei was the recipient of the Presidential Fellowship award, the most prestigious fellowship for senior Northwestern Ph.D. and MFA students. Her work also won a Computerworld Horizon Award for high-impact commercially-used information technology. She serves on the technical program committees of IEEE International Conference on Embedded Software and Systems and IEEE International Conference on Embedded Computing. She has published in a wide range of embedded system design topics including data compression, memory hierarchy design, and user satisfaction driven power management.

Robert P. Dick (S'95-M'02) is an Associate Professor of Electrical Engineering and Computer Science at the University of Michigan. He received his Ph.D. degree from Princeton University and his B.S. degree from Clarkson University. He worked as a Visiting Professor at Tsinghua University's Department of Electronic Engineering, as a Visiting Researcher at NEC Labs America, and as an Associate Professor at Northwestern University. Robert received an NSF CAREER award and won his department's Best Teacher of the Year award in 2004. His technology won a Computerworld Horizon Award and his paper was selected by DATE as one of the 30 most influential in the past 10 years in 2007. He served as a technical program subcommittee chair for the International Conference on Hardware/Software Codesign and System Synthesis. He is an Associate Editor of IEEE Transactions on VLSI Systems and serves on the technical program committees of several embedded systems and CAD/VLSI conferences. Robert has published in the areas of embedded operating systems, data compression, embedded system synthesis, dynamic power management, low-power and temperature-aware integrated circuit design, wireless sensor networks, human perception aware computer design, reliability, embedded system security, and behavioral synthesis.

Li Shang (S'99-M'04) is an Assistant Professor at the Department of Electrical and Computer Engineering, University of Colorado at Boulder. Before that, he was with the Department of Electrical and Computer Engineering, Queen's University. He received his Ph.D. degree from Princeton University, and his B.E. degree with honors from Tsinghua University. He has published in the areas of embedded systems, design for nanotechnologies, design automation, distributed computing, and computer systems, particularly in thermal/reliability modeling, analysis, and optimization. His work on computer system thermal management was ranked the 10th most downloads in IEEE Trans. CAD, 2008. His work on multi-scale thermal analysis was nominated for the Best Paper Award at ICCAD 2008. His work on hybrid SET/CMOS reconfigurable architecture was nominated for the Best Paper Award at DAC 2007. His work on thermal-aware incremental design flow was nominated for the Best Paper Award at ASP-DAC 2006. His work on temperature-aware on-chip network has been selected for publication in MICRO Top Picks 2006. He also won the Best Paper Award at PDCS 2002. He is currently serving as an Associate Editor of IEEE Transactions on VLSI Systems and serves on the technical program committees of several embedded systems and design automation conferences. He won his department's Best Teaching Award in 2006. He is the Walter Light Scholar.

Haris Lekatsas is a consultant specializing in the embedded systems software area. His research interests include computer architecture, embedded system design and applications of information theory. He has a BS in electrical and computer engineering from the National Technical University of Athens, and an MA and PhD in electrical engineering from Princeton University.