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

Srikanth Pothula, Sk Nayab Rasool

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. 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 design, permitting performance, power consumption, and area estimation. 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, hardware implementation, pair matching, parallel compression.

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 effi- ciency. 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 uni-processors by up to 17% for memory-intensive commercial workloads [1] and up to 225% for memory-intensive scientific workloads [2]. Researchers have also found that cache compression and pre-fetchingtechniques can improve CMP throughput by 10%–51% [3].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.

Manuscript Received on 28 October, 2011

Srikant Pothula, Ece, Jntu Kakinada/Pydah Engineering College/Vishakapatnam,India,9573022727 (E-Mail: Srikanth_Pothula@Yahoo.Co.In).

Sk Nayab Rasool, Ece, Jntu Kakinada/ Pydah Engineering College/ Vishakapatnam, India, 9247972800, (E-Mail: Sknayab54@Gmail.Com).

IBM reports compression ratios (compressed size divided by uncompressed size) ranging from 16% to 50%.

X-Match is a dictionary-based compression algorithm that 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 indexes with fewer bits. Although appropriate for compressing main memory, such hardware usually has a very large block size (1 KB forMXT 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 1KBto 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.C.3

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 are as follows.

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 systemwide compression ratio.

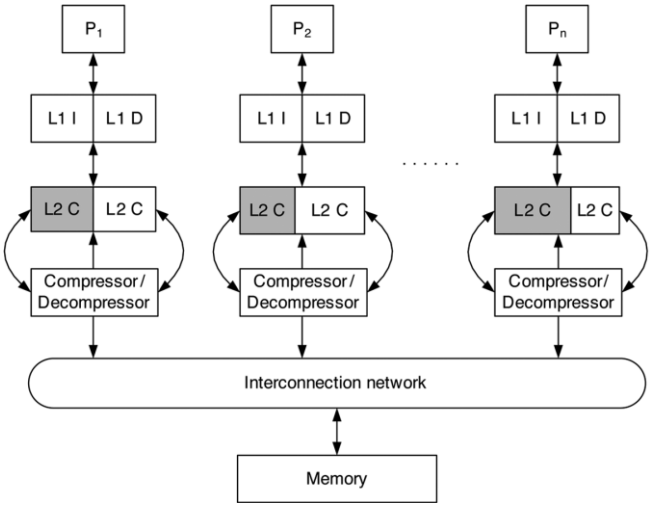
4) C-Pack is twice as fast as the best existing hardware implementations potentially suitable for cache compression. For FPC to match this performance, it would require at least the area of C-Pack.

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.

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

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.

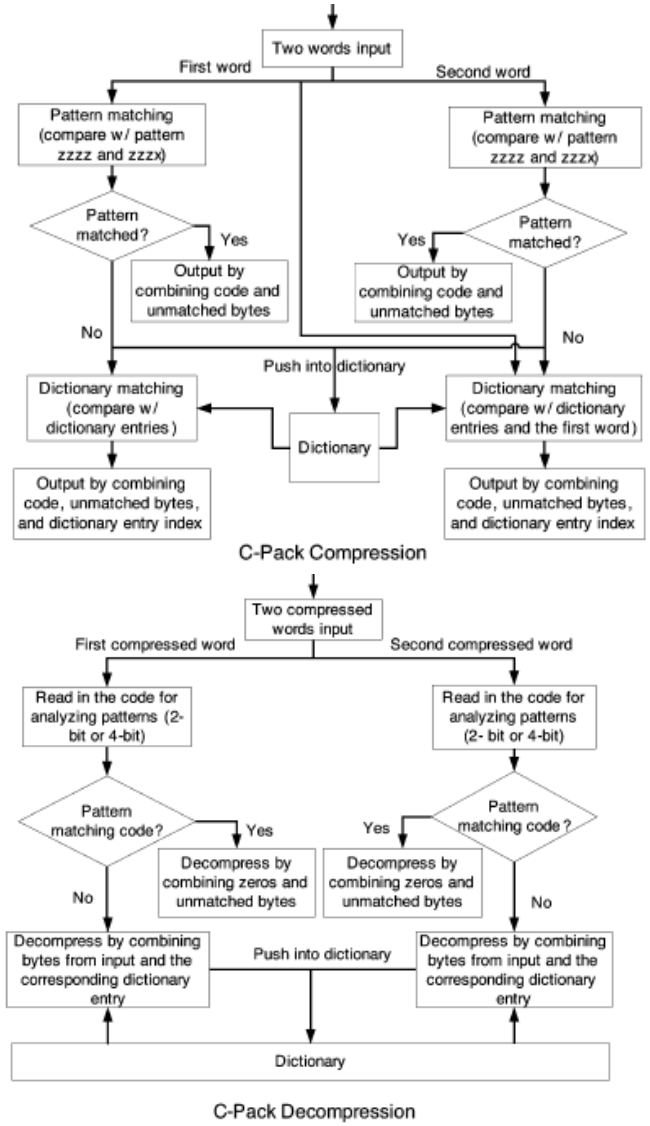
B. C-Pack Algorithm Overview

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 Fig. 2. We use an input of two words per cycle as an example in Fig. 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. Fig. 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 Fig. 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 Fig. 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.

Software implementations commonly serialize operations. 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 degrade 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 similarwords 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.

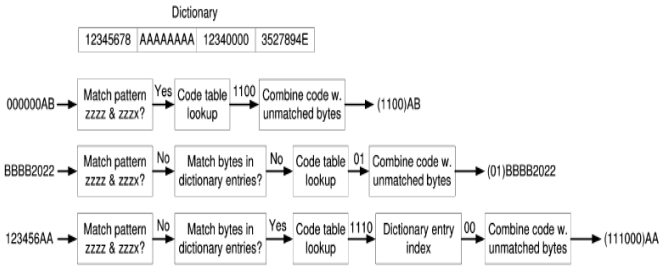


Fig. 3. Compression examples for different input words.

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	(1100)B	12	7.3
1110	mmmx	(1110)bbbbB	16	7.2

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 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, we 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 our proposed pair-matching based cache because each compression ratio, not the average, must be less than 0.5, for compression to 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

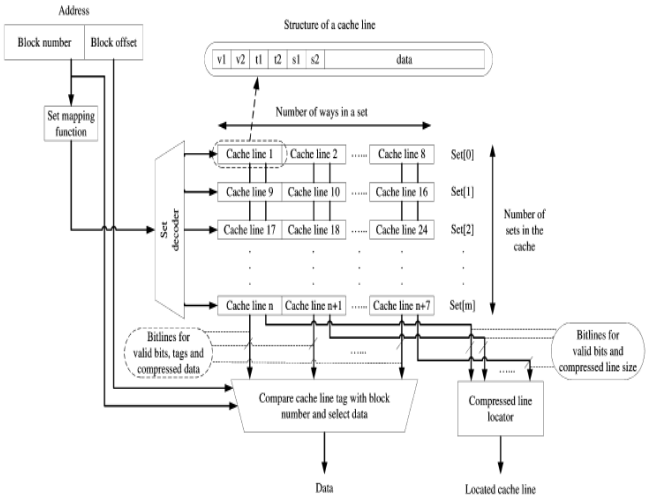
Cache, where the L2 cache dynamically allocates compressed or uncompressed lines depending on whether compression eliminates a miss or incurs an unnecessary decompression
overhead. However, this approach has significant performance and area overhead, discussed later in this section.

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.

Fig. 4 illustrates the structure of an 8-way associative pairmatching 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) (valid) (size) bits, i.e., 6 bytes. As we can see in Fig. 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 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 pairmatching 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. Fig. 5 illustrates the distribution of raw compression ratios for different cache lines derived from real cache data. The axis shows different compression ratio intervals and 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.



l. Structure of a pair-matching based cache.

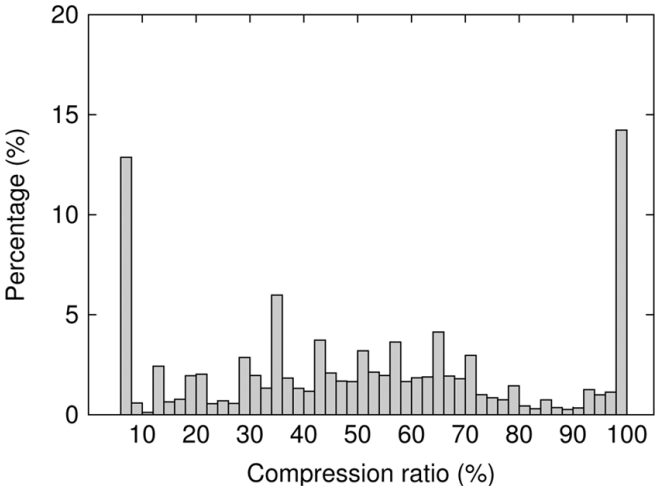


TABLE II
EFFECTIVE SYSTEM-WIDE COMPRESSION RATIOS FOR C-PACK

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

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 algorithm.

V. CONCLUSION

This paper has proposed and evaluated an algorithm for cache compression that honors the special constraints this application imposes. The algorithm is based on pattern matching and partial dictionary coding. Its hardware implementation permits parallel compression of multiple words without degradation of dictionary match probability. The proposed algorithm yields an effective system-wide compression ratio of 61%, and permits a hardware implementation with a maximum decompression latency of 6.67 ns 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*, Jun. 2004, pp. 212–223.
2. E. G. Hallnor and S. K. Reinhardt, "A compressed memory hierarchy using an indirect index cache," in *Proc. Workshop Memory PerformanceIssues*, 2004, pp. 9–15.
3. P. Pujara and A. Aggarwal, "Restrictive compression techniques to increase level 1 cache capacity," in *Proc. Int. Conf. Computer Design*, Oct. 2005, pp. 327–333.
4. L. Yang, H. Lekatsas, and R. P. Dick, "High-performance operating system controlled memory compression," in *Proc. Design Automation Conf.*, Jul. 2006, pp. 701–704.

AUTHORS PROFILE

Srikanth Pothula pursuing MTECH in Pydah college of engineering and did BE in Sir C R Reddy college of engineering…

Sk Nayab Rasool is an assistant professor in dept of ECE pydah college of engineering &pursuing mtech.