International Journal of Power System Operation and Energy Management

Volume 1 | Issue 2 Article 5

April 2012

C-Pack: Cache Compression for Microprocessor Performance

T. Narasimhulu
JNTU Anantapur SIETK, Narayana Vanam Road, Puttur, Chittoor Dt, Andhra Pradesh, INDIA, narasimhulu.thoti@gmail.com

C-Pack: Cache Compression for Microprocessor Performance

T.Narasimhulu [1]

M.TECH VLSI DESIGN, JNTU Anantapur
SIETK, Narayana Vanam Road, Puttur, Chittoor Dt, Andhra Pradesh, INDIA

Email ID: [1]narasimhulu.thoti@gmail.com

***Abstract*—Computer systems and micro architecture researchers have proposed using hardware data compression units within the memory hierarchies of microprocessors in order to improve performance, energy efficiency, and functionality. 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. In this work, I 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.**

***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 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 pre fetching 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. 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 *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.

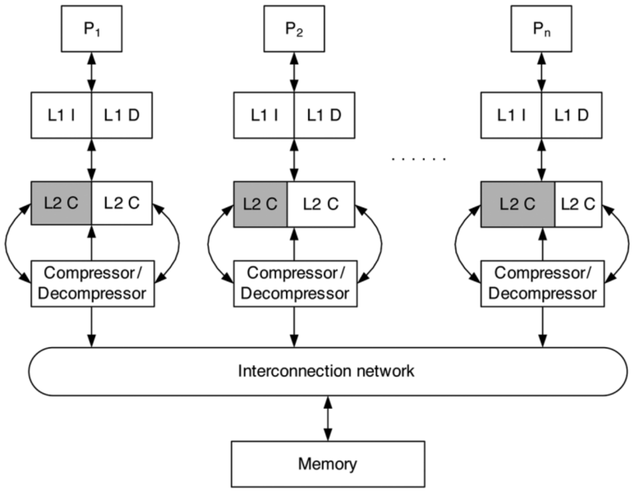


Fig. 1. System architecture in which cache compression is used.

III. 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 compressed-cache architecture.

A. Design Constraints and Challenges

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

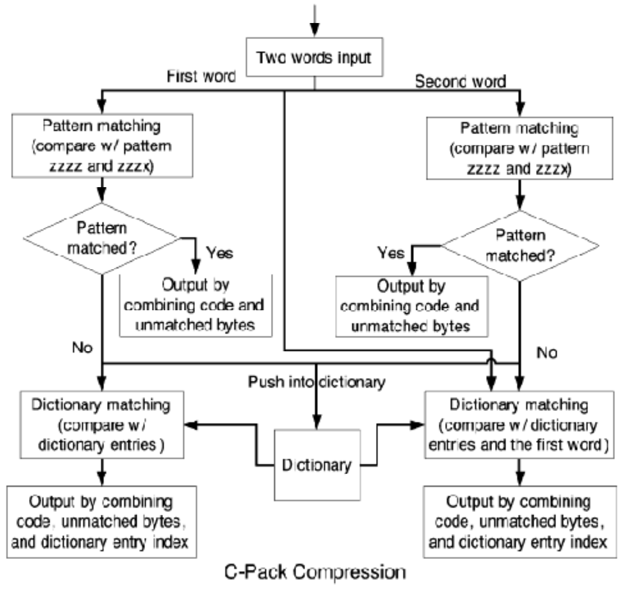
1) Cache compression requires hardware that can de/compress a word in only a few CPU clock cycles. This rules out software implementations and has great influence on compression algorithm design.
2) Cache compression algorithms must be lossless to maintain correct microprocessor operation.
3) The block 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. 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. 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. 2 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. 2 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. 2.


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

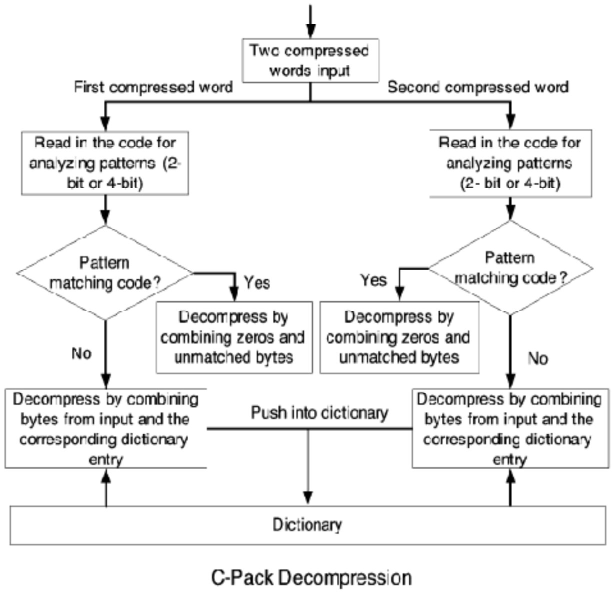

Two compressed words input
First compressed word
Second compressed word
Read in the code for analyzing patterns (2-bit or 4-bit)
Pattern matching code?
Yes
No
Decompress by combining zeros and unmatched bytes
Decompress by combining bytes from input and the corresponding dictionary entry
Push into dictionary
Dictionary
C-Pack Decompression

Fig. 2. 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

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.

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.

IV. 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 or 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], X-match, and FPC. Finally, we describe the implications of our findings on the feasibility of using C-Pack based cache compression within a microprocessor.

A. C-Pack Synthesis Results

We synthesized our design using 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 x 1.25 GHz) for compression and 76.8 Gb/s (64 B x 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 .

B. Comparison of Compression Ratio

2)
3)

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 lack hardware designs 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 Media bench and SPEC CPU2000 benchmark suites. The block size and the dictionary size are both set to 64 B in all test cases. Since we are unable to determine the exact compression algorithm used in MXT, we used the LZSS Lempel-Ziv compression algorithm to approximate its compression ratio. 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 is 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 m CMOS ASIC technology. The decompression rate is 8 B/cycle with 4 decompression engines. We scale the frequency up 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.

Comparing C-Pack with X-Match: X-Match has been implemented using 0.25 m 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 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.

4) *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. 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 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 compressor and 0.043 mm 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. Some publications assume that existing Lempel-Ziv compression algorithm based hardware would be sufficient to meet these requirements [2]; 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. 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 micro-architectural 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.

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.

Acknowledgment

The authors would like to acknowledge A. Rajni at E-soft academy, Bangalore 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*, 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 Performance Issues*, 2004, pp. 9–15.

[3] R. Alameldeen and D. A.Wood, "Interactions between compression and prefetching in chip multiprocessors," in *Proc. Int. Symp. High-Performance Computer Architecture*, Feb. 2007, pp. 228–239.

[4] Moffat, "Implementing the PPM data compression scheme," *IEEE Trans. Commun.* , vol. 38, no. 11, pp. 1917–1921, Nov. 1990.

[5] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.

[6] Tremaine *et al.*, "IBM memory expansion technology," *IBM J Res.Development*, vol. 45, no. 2, pp. 271–285, Mar. 2001.