**A Seminar Report**

**On**

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

***Submitted by***

***Pawar Manojkumar***

***11IT18F***

***II Sem M.Tech(IT)***

***In partial fulfilment for the award of the degree***

***Of***

**Master of Technology**

In

**Information Technology**

At



**Department Of Information Technology**

**National Institute Of Technology Karnataka, Surathkal.**

**2012-2013**

**DECLARATION BY STUDENT**

I hereby declare that an implementation of the IEEE/ACM/Journal paper entitled “**C-Pack: A High-Performance Microprocessor Cache Compression Algorithm**”, by **Xi Chen, Lei Yang & Robert P. Dick** published in the year **2010** was carried out by me during the even semester of the academic year 2011 – 2012, for the course Seminar/Professional Practice as per the M.Tech (I.T) degree curriculum. I declare that this is my original work and has been completed successfully according to NITK specifications.

------------------------------------------------------

(Signature of the Student)

Place:

Date:

# ABSTRACT

Modern processors use two or more levels of cache memories to bridge the rising disparity between processor and memory speeds. Compression can improve cache performance by increasing effective cache capacity and eliminating misses. However,

decompressing cache lines also increases cache access latency, potentially degrading performance. We propose a two-level cache hierarchy where the L1 cache holds uncompressed data and the L2 cache dynamically selects between compressed and uncompressed storage. The L2 cache is 8-way set-associative with LRU replacement, where each set can store up to eight compressed lines but has space for only four uncompressed lines.

Computer systems and micro architecture researchers have pro-posed 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 design, permitting performance, power consumption, and area estimation.

# Table Of Contents

1. Introduction 1

2. Literature Survey 2

2.1 Cache Compression Architecture 2

2.2 C-Pack Compression Algorithm 3

2.3 C-Pack Algorithm Overview 4

3. Results And Analysis 9

4. Future Work 12

5. Conclusion 13

6. References 14

# 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 .Conventional compression algorithm quality metrics, such as block compression ratio, are not appropriate for judging quality in this domain. Instead, one must consider the effective system-wide compression ratio .This paper will point out a number of other relevant quality metrics for cache compression algorithms, some of which are new. Finally, cache compression should not increase power consumption substantially.

# Literature Survey

## 2.1 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 ac-cess 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 .

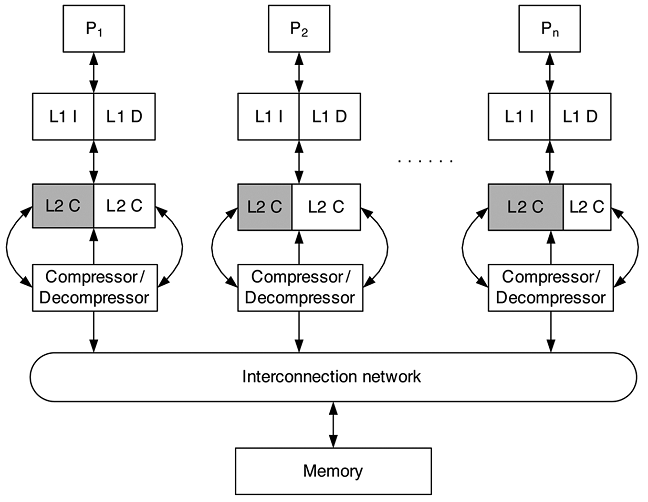


Figure 1: System Architecture of Cache

## 2.2 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 soft-ware 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/com-press 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.

## 2.3 C-Pack Algorithm Overview

C-Pack (for Cache Packer) is a lossless compression algorithm designed specifically for high-performance hard-ware-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 pat-tern-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 file. 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.

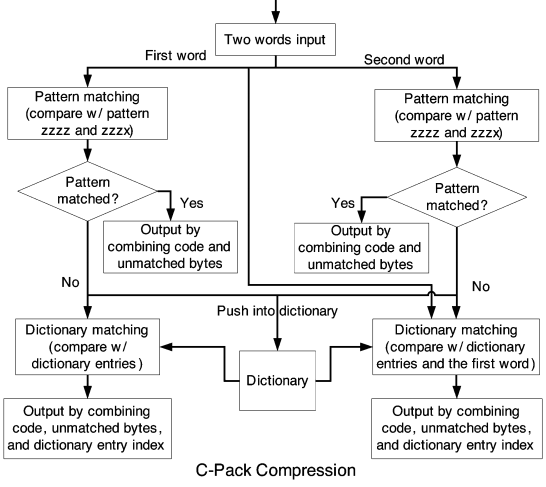


Figure 2: C-Pack Compression

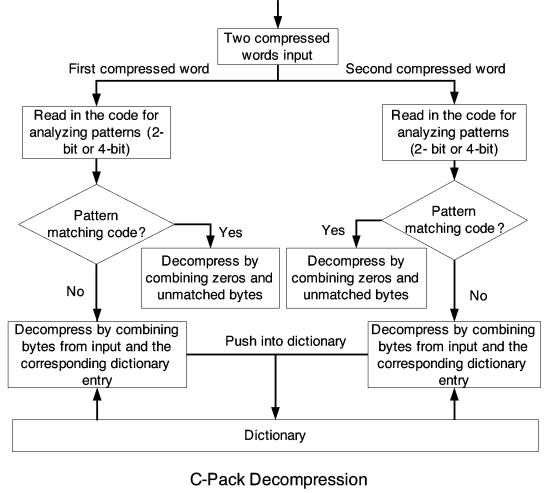


Figure 3: C-Pack Decompression

**Table 1**

**Pattern Encoding for C-Pack**

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

The C-Pack compression and decompression algorithms are illustrated in Fig. 2 & Fig 3. 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. During decompression, the decompressor first reads com-pressed 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 suit-able 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 en-tries 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 de-grade 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.

# Results And Analysis

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 . 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.

**Table 2**

**Effective System-wide Compression Ratios for C-Pack**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 FIFO | 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 | Huffman | 57.20 | 56.68 | 56.63 | 56.73 | 56.75 | 56.87 |
| Two-level | 57.66 | 57.31 | 57.08 | 57.11 | 57.35 | 57.44 |

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 hard-ware 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 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 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.

**Table 3**

**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 |

# Future Work

For future work, it is possible to improve performance by implementing advanced hardware. If it is possible then by looking at application we may choose algorithm to be applied.

# 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,” inProc. 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,” inProc. Workshop Memory Performance Issues, 2004, pp. 9–15.

[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, pp. 228–239.

[4] A. 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.