International Journal of Instrumentation Control and Automation

Volume 1 | Issue 3 Article 8

October 2011

A VLSI Approach for Cache Compression in Microprocessor

Sharada Guptha M N
Dept. of E&C, SSIT, Tumkur, Karnataka, India, gupthanps@gmail.com

H. S. Pradeep
Dept. of E&C, SSIT, Tumkur, Karnataka, India, Pradep.hs@gmail.com

M Z Kurian
Dept. of E&C, SSIT, Tumkur, Karnataka, India, mzkurianvc@yahoo.com

Recommended Citation

M N, Sharada Guptha; Pradeep, H. S.; and Kurian, M Z (2011) "A VLSI Approach for Cache Compression in Microprocessor," *International Journal of Instrumentation Control and Automation*: Vol. 1 : Iss. 3 , Article 8.
DOI: 10.47893/IJICA.2011.1034
Available at: https://www.interscience.in/ijica/vol1/iss3/8

A VLSI Approach for Cache Compression in Microprocessor

Sharada Guptha M N, H. S. Pradeep & M Z Kurian

Dept. of E&C, SSIT, Tumkur, Karnataka, India
E-mail : gupthanps@gmail.com, Pradep.hs@gmail.com, mzkurianvc@yahoo.com

Abstract - Speed is one of the important issues that generally customers consider for selecting any electronic component in the market. Speed of a microprocessor based system mainly depends on the speed of the microprocessor which in turn depends on the memory access time. Accessing on chip memory takes more time than accessing off-chip memory. Because of these, designers of memory system may find cache compression as an advantageous method to increase speed of a microprocessor based system, as it increases cache capacity and off-chip bandwidth. The 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. Proposed hardware compression algorithms fall into the dictionary-based category, which depend on building a dictionary and using its entries to encode repeated data values. Proposed algorithm has number of novel features like 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.

Keywords - *off-chip memory, memory latency, cache compression, memory hierarchy, parallel compression, compression ratio.*

I. INTRODUCTION

The widening gap between processor and memory speeds, results because of tight constraints on the amount of on-chip cache memory and the high latency of off-chip memory, such as dynamic random access memory. More time is essential to access off-chip memory time required to access generally takes an accessing on-chip cache. Hence to improve memory-system efficiency cache hierarchies is been incorporated on chip, but it is constrained by die area and cost. Cache compression is one such technique; data in last-level on-chip caches, e.g., L2 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]. However past work did not demonstrate whether the proposed compression and decompression hardware is appropriate for cache compression, considering the performance, area and power consumption requirements.

Cache compression has to overcome several constraints. 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 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.

II. RELATED WORK AND CONTRIBUTIONS

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 (com- Pressed size divided by uncompressed size) ranging from 16% to 50%.

X-Match is a dictionary-based compression algorithm. 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 com- pressing main memory, such hardware usually has a very large block 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.

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 my knowledge, there is no register-transfer-level hardware implementation or FPGA implementation of FPC power consumption, and area overheads are unknown.

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 paper a lossless compression algorithm is been proposed and developed. The algorithm is named C-Pack, for on-chip cache compression. The main contributions of this 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.
2) When cache compression algorithm is implemented using FPGA, performance and power requirements can be easily analyzed.
3) C-pack makes a pair of compressed lines to fit into a single uncompressed cache line.
4) The proposed hardware can be easily adapted to other high-performance lossless compression applications.

III. CAHE COMPRESSION ARCHITECTURE

For this work, consider private on-chip L2 cache is considered, because in contrast to a shared L2 cache, the design styles of private L2 caches remain consistent when the number of processor cores increases.

Fig. 1 gives an overview of a system architecture where compression is used. Processor has private L1 and L2 caches. L1 cache is subdivided into two parts to show separate code and data memory. L2 cache is unified in nature. Hence L2 cache is considered for this work.

The main point that can be considered here is that no architectural changes are needed to be done in processor to implement the proposed techniques for a L2 cache.

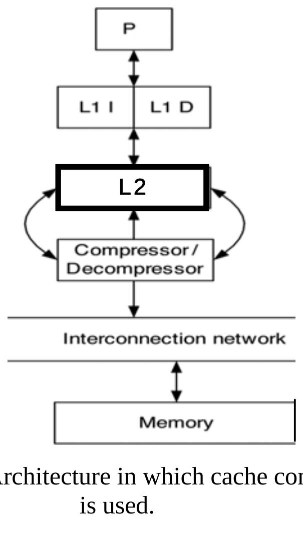


Fig. 1: System Architecture in which cache compression is used.

IV. COMPRESSION ALGORITHM

The algorithm used for compression and decompression here is C-pack which has several advantages as mentioned above. C-pack algorithm 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.

Cache compression algorithm is lossless to maintain correct microprocessor operation. The complexity of managing the locations of cache lines after compression influences feasibility. 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 [16]. However, this prior work was designed for software-based main memory compression and did not consider hardware implementation.

C-Pack achieves c ompression 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,. The 'Pattern' column describes frequently ap- pearing 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.

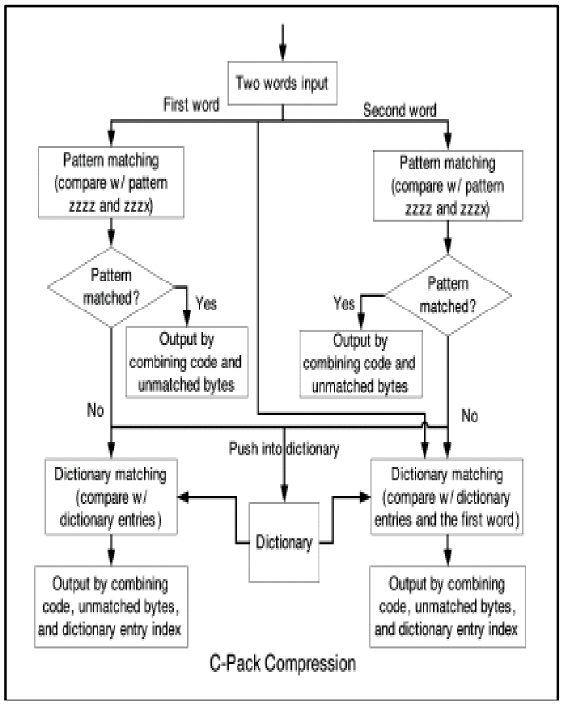

Two words input
First word
Second word
Pattern matching (compare w/ pattern zzzz and zzzx)
Pattern matching (compare w/ pattern zzzz and zzzx)
Pattern matched?
Yes
Output by combining code and unmatched bytes
Pattern matched?
Yes
Output by combining code and unmatched bytes
No
No
Push into dictionary
Dictionary matching (compare w/ dictionary entries)
Dictionary
Dictionary matching (compare w/ dictionary entries and the first word)
Output by combining code, unmatched bytes, and dictionary entry index
Output by combining code, unmatched bytes, and dictionary entry index
C-Pack Compression

Fig. 2 : Compression and decompression flow chart

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 en-tries is large. C-Pack's inherently parallel design parallel design allows an effi-cient hardware implementation, in which pattern matching, dictionary matching, and processing multiple words are all done simultaneously.

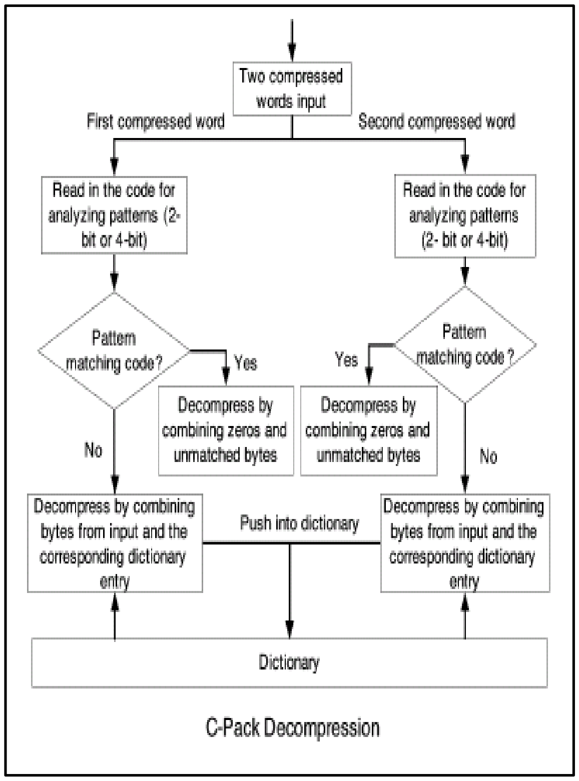

Two compressed words input
First compressed word
Second compressed word
Read in the code for analyzing patterns (2-bit or 4-bit)
Read in the code for analyzing patterns (2- bit or 4-bit)
Pattern matching code?
Yes
Yes
Pattern matching code ?
Decompress by combining zeros and unmatched bytes
Decompress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

Fig. 3: Compression examples for different input words.

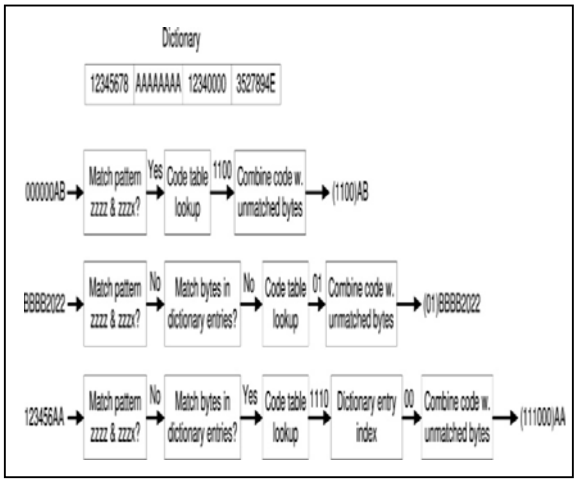
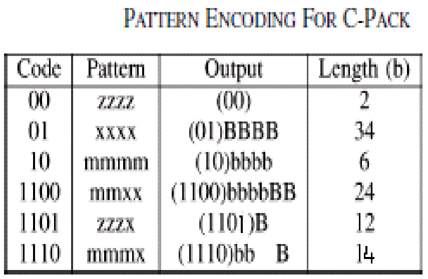

Dictionary
12345678
AAAAAAAA
12340000
3527894E
000000AB
Match pattern zzzz & zzzx?
Yes
Code table lookup
1100
Combine code w. unmatched bytes
(1100)AB
BBBB2022
Match pattern zzzz & zzzx?
No
Match bytes in dictionary entries?
No
Code table lookup
01
Combine code w. unmatched bytes
(01)BBBB2022
123456AA
Match pattern zzzz & zzzx?
No
Match bytes in dictionary entries?
Yes
Code table lookup
1110
Dictionary entry index
00
Combine code w. unmatched bytes
(111000)AA

Fig. 4 : Compresiion examples for different input wods.



PATTERN ENCODING FOR C-PACK

Code	Pattern	Output	Length (b)
00	zzzz	(00)	2
01	xxxx	(01)BBBB	34
10	mmmm	(10)bbbb	6
1100	mmxx	(1100)bbbbBB	24
1101	zzzx	(1101)B	12
1110	mmmx	(1110)bb B	14

Table 1: Pattern Encoding for C-pack

In the proposed implementation of C-pack , two words are processed parallely per cycle. Achieving this, while still permitting an accurate dictionary match for second word, is challenging task here. If two similar words are considered that have not been encountered by the compression algorithm recently, assuming the dictionary that 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.

The above shown algorithm steps is coded in verilog language as it is easier than any other HDL language and because of some of its salient features like it allows the descriptions of each module to done mathematically in terms its terminals and external parameters applied to the module.etc. The same has been simulated using the able simulation tool modelsim 6.2.The results in terms of numbers and waveforms are analyzed to get accurate results. Then the code can be synthesized using Xilinx software.

V. C-PACK HARDWARE IMPLEMENTATION

The proposed hardware implementation of compressor and decompressor of C-pack targets mainly on on-chip cache compression.. Even the same hardware can be used in other data compression applications, such as memory compression and network data compression, with few or no modifications.

VI. CONCLUSION

By the implementation of the proposed algorithm, it is possible to compress and decompress the data into the cache in an efficient way without altering the processor performance. This method maintains good compression ratio and area overhead and thus decreases memory latency and speeds up the processor and by making the system to work with high speed and thus helpful for mankind.

REFERENCES

[1] A. R. Alameldeen and D. A. Wood, “Adaptive cache compression for high performance processors,” in Proc. Int. Symp. Computer Architecture, pp. 212–223, Jun. 2004

[2] E. G. Hallnor and S. K. Reinhardt, “A compressed memory hierarchy using an indirect index cache,” in Proc. Workshop Memory Performance Issues, pp. 9–15, 2004,

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

[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.

[6] B. Tremaine et al., “IBM memory expansion technology,” IBM J. Res. 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. Very Large Scale Integr. (VLSI) Syst., vol. 11, no. 3, pp. 499–510, Jun. 2003.

[8] A. Alameldeen and D. A. Wood, “Frequent pattern compression: A significance- based compression scheme for 12 caches,” Dept. Comp. Scie. , Univ. Wisconsin- Madison, Tech. Rep. 1500, Apr. 2004.

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

[10] J.-S. Lee et al., “Design and evaluation of a selective compressed memory system,” in Proc. Int. Conf. Computer Design, pp. 184–191, Oct. 1999.