VOL. 10, NO. 7, APRIL 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences


www.arpnjournals.com

A NOVEL APPROACH FOR A HIGH PERFORMANCE LOSSLESS CACHE COMPRESSION ALGORITHM

K. Janaki[1], K. Indhumathi[2], P. Vijayakumar[3] and K. Ashok Kumar[4]
[1]Department of Electronics and Communication Engineering, Prathyusha Institute of Technology and Management, Thiruvallur, India
[2]Department of Electronics, Prathyusha Institute of Technology and Management, India
[3]Department of Electrical and Electronics Engineering, Karpagam College of Engineering, Coimbatore, India
[4]Department of Electronics, SNR Sons College, Coimbatore, India

ABSTRACT

Speed is one of the major issues for any electronic component. Speed based microprocessor system mainly depends on speed of the microprocessor and memory access time. The off-chip memory takes more time for accessing than on-chip memory. For these reasons, microprocessor system designers find cache compression is such a technique to increase the speed of a microprocessor based system, as it increases the cache capacity and off-chip bandwidth. Previous work on cache compression has made unsubstantiated assumptions about performance, power consumption and area overheads of the proposed compression algorithm and hardware. In this work we propose a lossless compression algorithm that has been designed for high performance, fast on-line data compression and particularly for cache compression. This 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.

Keywords: cache compression, on-line data compression, parallel compression.

1. INTRODUCTION

Nowadays microprocessor speeds have been increasing faster than off chip memory speed. Because today's microprocessors have on-chip cache hierarchies which has several megabytes of storage. When the system design utilizes the multiprocessor, it requires more access to memory [1]. Thus it rears a wall between the processor and memory, which results in fortifying the off chip communication. Microprocessor researchers found that a techniques that reduces the communication between the offchip which does not affecting the performance have capable to alleviate this problem. Cache compression is such a technique that reducing the off chip misses and improving the performance.

The challenges over cache compression are:

1. Compression and decompression should be very fast.
2. The hardware should occupy less area.
3. The algorithm should compress small block without losses. E.g. 64-byte cache lines when maintaining a good compression ratio (the ratio between the sizes of the compressed data over uncompressed data).
4. Cache compression should reduce the power consumption.

2. RELATED WORK AND CONTRIBUTIONS

The X-Match algorithm [3] is a dictionary based compression algorithm mainly depends on previous data and strives to match the current data element with the dictionary entries. The entries of each word in the dictionary are 4-bytes wide and many types of matches are possible. The bytes which do not match with the dictionary are sent separately. This partial match concept refers to the procedure 'X-Match'. The dictionary uses Move to front strategy (MTF), where new tuple is placed at the front of the dictionary while the remaining words or tuples are move down to one position. This MTF strategy generates a Least Replacement Policy (LRU). So the dictionary size tuples in the last position are used as a compression process. When the dictionary becomes full which is in the last position is worn-out and the leaving the place for a new one. A match coding function requires to code these three separate fields:

- Match location
- Match type
- Any extra characters

The X-Match algorithm was done in FPGA. Though it is appropriate for compressing main memory, hardware has very large block size which is difficult for compressing the cache lines.

Frequent pattern Compression (FPC) technique compresses the cache line by storing a frequently appearing word patterns [4]. Each cache line is compressed on a word by word basis and is splitted into a 32-bit word. Each 32-bit word is encoded as a 3-bit prefix and data shown in table 1. If the word matches with any of the patterns given in Table 1, then each word in the cache line is encoded into a compressed format. If theword does not match with any of these patterns, then it is stored in its original 32-bit format i.e. the whole word is stored with the prefix '111'.

Table-1.

Prefix	Pattern Encoded	Data Size
000	Zero run	3 bits
001	4-bit sign extended	4 bits
010	One byte sign extended	8 bits
011	Half word sign extended	16 bits
100	Half word padded with a zero half word	Non zero half word
101	Two half words, each a byte sign extended	2 bytes
110	Word consisting of repeating bytes	8 bits
111	Uncompressed word	Original word

Cache line compression takes place when data is written back from L1 cache to L2 cache. Using a simple circuit, a cache line can be compressed easily by checking each word for pattern matching. It can be done in a memory pipeline. Cache line decompression takes place when data is read from the L2 to the L1 cache. Compression is faster than the decompression process, since prefixes for all words are in series. Here prefix can be used to find the length of the encoded word. In this technique, no hardware implementation is possible and so its exact performance, power consumption and area are unknown.

Restrictive compression technique is used to reduce the cache access latency [5], which results in increase in the L1 data cache capacity. The basic technique used in this is All Words Narrow (AWN). This technique compresses a cache block only if all the words in the cache block are of narrow width. If the word can be represented using 16 bits, then it is considered as a narrow word. This AWN technique alone can be used to increase the cache capacity of L1 data by about 21%. The AWN technique can be extended by leaving some extra space for a few upper half words (AHS) in a cache block. Further the AHS technique can be widened to Adaptive AHS (AAHS), so that a cache block uses the number of upper half words. In the cache, the physical RAM space is provided for the cache block named as 'physical cache block' which holds a normal cache block (width bit="0") or up to two narrow cache blocks (width bit="1"). In the AWN technique, LRU policy acts as a replacement policy. In the cache block the byte offset of each word depends on the size of the words that present before it. So to read a word from the block, it will need to recalculate the byte offset. The drawback of this technique is to reduce the cache access latency, it cannot change the byte offset of the memory reference.In short cache compression hardware performance and low area and power overheads is common in cache compression research [2], [7]-[10].
In this work we present a c-pack algorithm which is a lossless compression algorithm mainly for on-chip cache compression. The main contributions of our work follow:

- C-Pack mainly aims on-chip cache compression. It allows a good compression ratio even when the compression is done on small cache lines. For practical use performance, area and power consumption are low enough [3].
- The performance and power consumption of a cache compression algorithm can be easily designed and optimized, when implemented using FPGA.
- C-Pack constitutes a pair of compressed lines to fit into a single uncompressed cache line.
- C-Pack is twice fast when compared to the existing hardware implementations that were potentially suitable for cache compression.
- The proposed hardware can be easily amended to other high performance lossless compression applications.

Cache compression architecture

In this work, private on-chip L2 caches can be examined, because in contrast to a shared L2 cache, the design styles of private L2 caches remain persistent when the number of processor core increases. A system architecture where compression used is shown in Figure-1. Each processor has private L1 and L2 caches. The L2 cache is divided into two regions: an uncompressed region (L2) and compressed region (L2C). For each processor, the sizes of the uncompressed region and compressed 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 consider a three level cache hierarchy consisting of consisting of L1 cache, uncompressed L2 region and compressed L2 region. The L1 cache can be used for communication purpose i.e. to communicate with the uncompressed region of the L2 cache, which in turn swaps data with the compressed region through compressor and decompressor, i.e. in the compressor the uncompressed line can be compressed and placed in the compressed region and vice versa. Compressed L2 is inherently a virtual layer in the memory hierarchy with larger size, but higher access latency than uncompressed L2. For the proposed technique i.e. for a shared L2 cache, no architectural changes are needed.

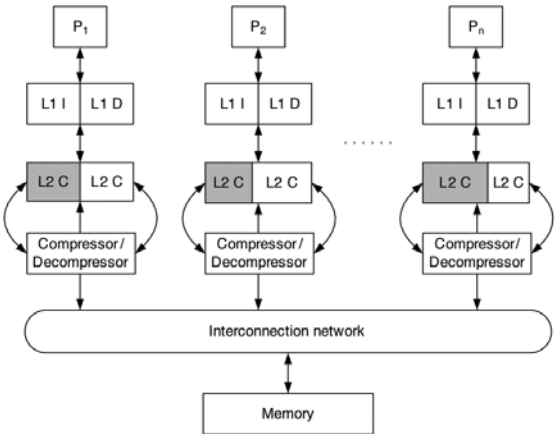

Figure-1. System architecture in which cache compression used.

3. C-PACK COMPRESSION ALGORITHM

This section briefly explains the proposed C-Pack compression algorithm and several important features that allow an efficient hardware implementation, many of which would be challenged for a software implementation.

Design constraints and challenges

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

- 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 improvement over compression algorithm design.
- To perpetuate the correctness of microprocessor operation, cache compression algorithms must be lossless.
- The block size for cache compression is small when compared to other compression applications such as file and main memory compression.

C-Pack algorithm overview

C-Pack is a lossless compression algorithm particularly 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 [6]. C-Pack achieves compression by two means 1) For frequently appearing word, it uses statically decided, compact encodings. 2) For other frequently appearing words, it encodes using dynamically updated dictionary. The dictionary supports partial word matching as well as full word matching. The patterns and coding schemes used by C-Pack are given in Table-2. The frequently appearing data is given in pattern column. In that pattern column '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.

Table-2.

Code	Pattern	Output	Length
00	zzzz	(00)	2
01	xxxx	(01)BBBB	34
10	mmmm	(10)bbbb	6
1100	mmxx	(1100)bbbbBB	24
1101	zzzx	(1100)B	12
1110	mmmx	(1110)bbbbB	16

The C-Pack compression and decompression algorithms are illustrated in Figure-2 and 4. Here two word input is used per cycle. This algorithm is pertinent for more than two words per cycle. During one iteration, each word is first compared with patterns "zzzz" and "zzzx". If there is a match against patterns, then the output is obtained by combining the corresponding code and unmatched bytes as indicated in Table-2. Otherwise the word can be compared with all dictionary entries and determines the one with the most matched bytes by compressor. The compression result is then generated by combining code, dictionary entry index and unmatched bytes if any. A word which does not match with the patterns is pushed into the dictionary. The compression result with different input words are shown in Figure-3. The code and the dictionary index are enclosed in parentheses in each output. In our implementation, though we used a 4-word dictionary the size of the dictionary is set to64B.

During decompression, the decompressor fetches the compressed words first and then extracts the codes for analyzing the patterns of each word, which are then compared against the codes indicated in Table-2. 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 obtained by combining bytes from the input word with bytes from dictionary entries. The C-Pack is particularly suitable for hardware implementation.

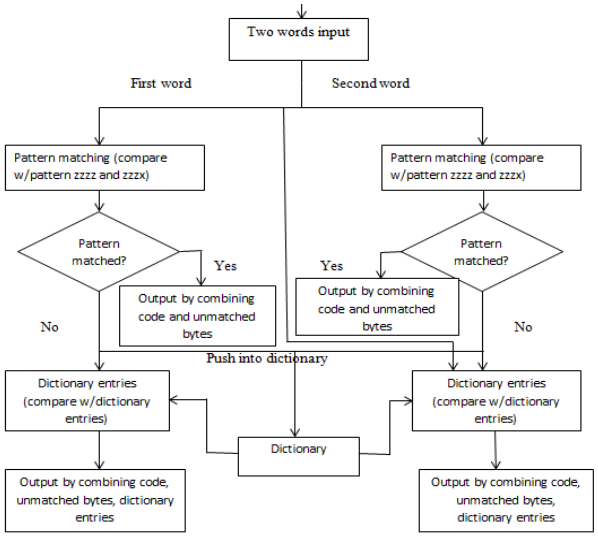

Figure-2. C-Pack compression.

The advantage of C-Pack is an input word is compared with multiple potential patterns and with dictionary entries simultaneously. This permits rapid execution with good compression ratio in a hardware implementation, but might not be suitable for software implementation. C-Pack's virtually parallel design allows an efficient hardware implementation, in which pattern matching, dictionary matching and processing multiple words are all executed simultaneously. To reduce hardware complexity, various design parameters such as dictionary replacement policy and coding scheme were chosen. In the proposed C-Pack implementation, two words are processed in parallel per cycle. Achieving this, while still permitting an

accurate dictionary match for the second word is challenging.

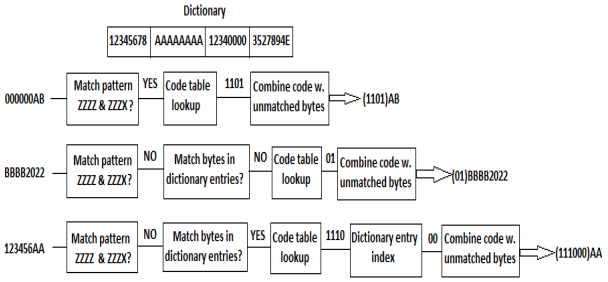

Figure-3. Compression examples with different inputs.

4. C-PACK HARDWARE IMPLEMENTATION

In this section, we briefly explain the description of proposed hardware implementation of C-Pack. Notice that though the proposed compressor and decompressor mainly target on-line cache compression, it can be used in other data compression applications such as memory compression and network data compression, with few or no modifications.

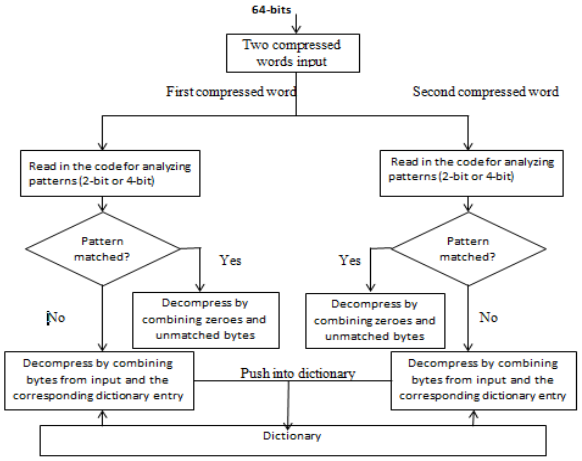

Figure-4. C-Pack decompression.

Compression hardware

This section describes the design and optimization of proposed compression hardware. The compressor is splitted into three pipeline stages shown in Figure-4. This design supports incremental transmission i.e. before the whole block has been compressed, the compressed data can be transmitted. So the compression latency can be reduced.

Pipeline stage 1

This stage can be used for matching purpose i.e. it can be used for matching the patterns as well as dictionary entries on two uncompressed words in parallel. The comparator array 1 is used to match the first word against patterns "zzzz" and "zzzx". Comparator array 2 matches it with all the dictionary entries (e.g. AAAAAAAA, 12340000), both in parallel. The same process is carried out for the second word also. During dictionary matching, the second word is compared with the first word as well as with the dictionary entries. The pattern matching results are then encoded using priority encoders 2 and 3. The first word and second word are processed simultaneously to increase the throughput. The result obtained from the priority encoder is used to determine whether these two words are used to push into the FIFO dictionary.

FIFO dictionary acts as a replacement policy. The dictionary size of FIFO here is 64 B. When the dictionary becomes full, it should remove the existing word and leave a place for a new word. The reading and writing operations can be performed in the FIFO memory. The dictionary supports partial word matching as well as full word matching. The appropriate dictionary content when processing the second word depends on whether the first word is matched with a pattern. If there is a match, the first word will not appear in the dictionary. Otherwise, it will be in the dictionary and the presence of the first word can be used to encode the second word.

Pipeline stage 2

This stagecomputes the total length of the two uncompressed and based on this length, it generates the control signal. Based on the dictionary matching from the stage 1, priority encoder 1 and 4 determines the dictionary entries with the most matched bytes. The obtained result is then sent to word length generator. Word length generators 1 and 2 are used to calculate the length of each compressed word. The total length calculator can be used to add the two lengths and it is represented by signal *total_length.* Then the value of total_length can be added to two internal signals, namely sum_partial and sum_length by using the length accumulator. Sum_partial represents the number of compressed bits stored in register array 1that have not been transmitted. If the updated sum_partial value is larger than 64 bits, then the sum_partial is decreased by 64 and the signal store_flag is generated. If the sum_total is larger than the original cache line size, then the compressor stops compressing and sends back the original cache line stored in the backup buffer. Backup buffer can be used to store the cache line.

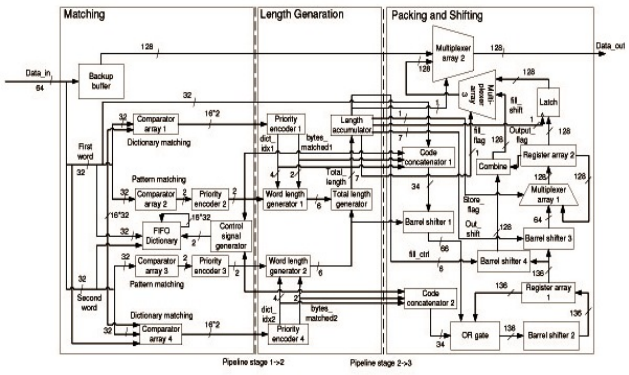

Figure-5. Compressor architecture.

Pipeline stage 3 is mainly for packing and shifting. This stage produces the compression output by combining codes, bytes from input word and bytes from dictionary entries depending on the pattern and dictionary matching results from the previous stages. Here the challenging is to place the compressed pair of words in the right location that is in register array 1, which is denoted by Reg1 [135:0]. It is impossible to pre select the output location, since the length of the compressed word differs from word to word. Without knowing the shift length, the register array 1 should be shifted to fit in the compressed output in a single. This problem can be overcome by analyzing the output length. Note that a single compressed word can have only seven possible output lengths, with maximum length of 34 bits. Therefore, we use two 34-bit buffers which can be used to store the first and second compressed outputs produced by code concatenators 1 and 2 in the lower bits, with the high unused bits set tozero. Thetwo 34-bit buffers can be denoted by A [33:0], B [33:0].

A barrel shifter can be used shift a data word by a specified number of bits in one clock cycle. It can be implemented as a sequence of multiplexers (mux). Reg1 [135:0] is shifted by total length using barrel shifter 2. The result obtained by shifting is denoted by Reg1s [135:0]. At the same time, A [33:0] is shifted using barrel shifter 1by the output length of the second compressed word and the result obtained by this shift is denoted by S [65:0], with all higher bits set to zero. Because of the maximum total output length is 68, Reg1 [135:68] has only one input source, i.e., Reg1s [135:68]. But the Reg1 [67:2] have multiple sources: B, S and Reg1s [0]. . The unused states in the input sources are all initialized to zero, which should not affect the OR function. The OR function is used to combine the inputs together. When the store flag is 1, then the multiplexer array 1 selects the input as Reg2 [135:0] which is obtained from the shifting result, otherwise it selects the original Reg2 [135:0].

Latch is enabled depending on the number of compressed bits accumulated in Reg2 [135:0] that have not been transmitted. Multiplexer array 3 selects fill_shift and the output of latch using fill_flag. Fill_shift represents 128-bit signal that stores the remaining compressed bits that have not been transmitted with zeroes. Fill_flag finds whether to select the padded signal. Multiplexer array 2 chooses the output data depends on the total number of compressed words. When the total compressed line has beyond the uncompressed line size, the contents in the backup buffer are selected as the output. Otherwise the multiplexer array 3 output is selected.

Decompression hardware

This section describes the design and improvement of the proposed decompression hardware. We describe the data flow inside the decompressor and point out the challenges specific to the decompressor design.

a) Word unpacking: When decompression starts, the two codes of the first and second word can be extracted by unpacker. Signals first_code and second_code represent the first two bits of the codes in the two compressed words. Signals first_bak and second_bak represent the next two bits following first_code and second_code respectively. It is mainly useful when the corresponding code is a 4-bit code.

b) Word decompressing: Decoders 1 and 2 can be used for comparing the codes of the first and second word against the static codes in Table 1 to derive the patterns for the two words, which are then decompressed by combining zero bytes, bytes from FIFO dictionary and bytes from register array 1. To produce the decompression results, the bytes are mainly depends on the values of the four code related signals. If there is no pattern match occurs, then the decompressed words are pushed into FIFO dictionary.

c) Length updating: Length generator can be used to derive the compressed lengths of the two words, i.e. first_len and second_len, based on the four code-related signals. The two lengths are then subtracted from chunk_length (denotes the number of the remaining bits to decompress in register array 1). The subtraction result is then compared with 68, and if the length is less than 68 then more data are shifted in and combined with the remaining compressed bits in register array 1.

5. EXPERIMENTAL RESULT

The Compression and decompression outputs according to C-Pack algorithm are shown below:

Compression results

The value for A is 1010 and the value for B is 1011. The input value given here is 000000AB. First the input is compared with the patterns “zzzz” and “zzzx”. If there is a match occurs, then it look up the code and the output is obtained by combining the zeroes (0000), code (1101), and Aand B shown in below Figure-6(a).

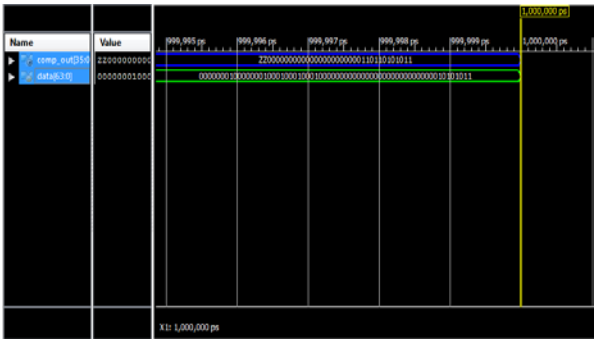

Figure-6(a). Compression output for 000000AB.

If there is no pattern match as well as no dictionary match, then the output is obtained by combining the unmatched bytes (zz),code word (01), and the inputs B(1101) and 2022(0010000000100010) shown in below Figure-6(b).

Figure-6(b). Compression output for BBBB2022

If there is no pattern match and dictionary match is possible then the output is obtained by combining the code (1110), dictionary entry index (00) and unmatched bytes is shown in below Figure-6(c)

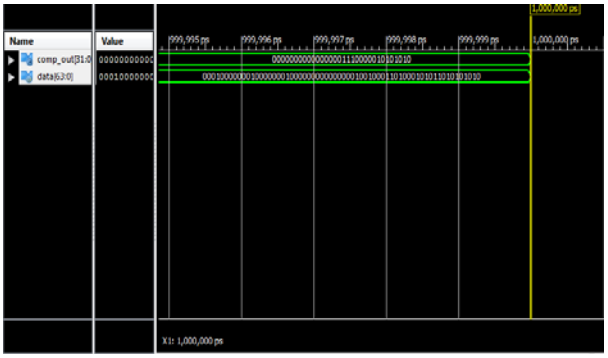

Figure-6(c). Compression output for 123456AA.

Decompression results

During decompression the original word is recovered. If the extracted code indicates a pattern match, then the original word is recovered by combining zeroes and it is given in Figure-7(a).

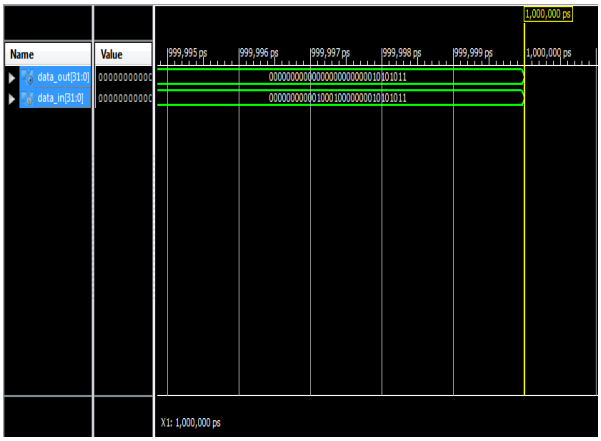

Figure-7(a). Decompression result for (1100)AB.

The decompression result is mainly depends on the values four code related signals. Figure-7(b) shows that it has the two code related input, so the output is as like input.

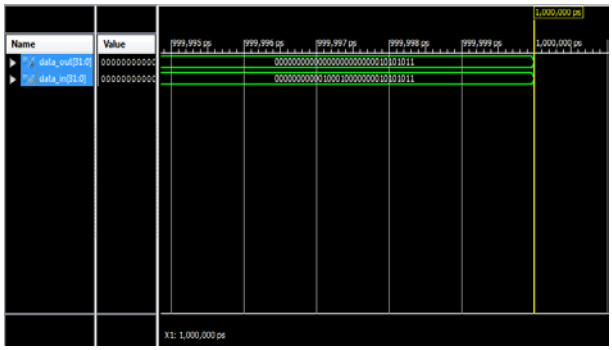

Figure-7(b). Decompression result for (01)BBBB2022.

If the code indicates that there is no match with the pattern but there is match with the dictionary entries then the original word is recovered by concatenating the zeroes and unmatched bytes, if any shown in Figure-7(c).

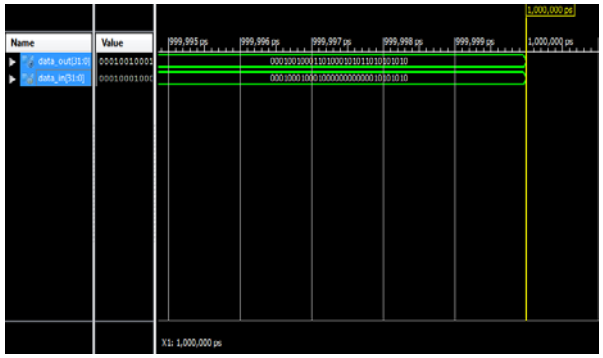

Figure-7(c). Decompression result for (111000)AA.

6. CONCLUSIONS

By the implementation of the proposed algorithm, it is possible to compress and decompress the data in to the cache in an efficient way without altering its 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. It can also be used for other high-performance lossless data compression applications with few or no modifications.

REFERENCES

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

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

[3] J. L. Núñez and S. Jones. 2003. "Gbit/s lossless data compression hardware," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., Vol. 11, no. 3, pp. 499–510, June.

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

[5] P. Pujara and A. Aggarwal. 2005. "Restrictive compression techniques to increase level 1 cache capacity," in Proc. Int. Conf. Computer Design, Oct. pp. 327–333.

[6] L. Yang, H. Lekatsas and R. P. Dick. 2006. "High-performance operating system controlled memory compression," in Proc. Design Automation Conf., Jul. pp. 701–704.

[7] J.-S. Lee *et al.* 2002. "Design and evaluation of a selective compressed memory system," in Proc. Int. Conf. Computer Design, October 1999, pp. 184–191.

[8] N. S. Kim, T. Austin, and T. Mudge, "Low-energy data cache using sign compression and cache line bisection," presented at the Workshop on Memory Performance Issues, May.

[9] K. S. Yim, J. Kim and K. Koh. 2004. "Performance analysis of on-chip cache and main memory compression systems for high-end parallel computers," in Proc. Int. Conf. Parallel Distributed Processing Techniques Appl., Jun. pp. 469–475.

[10] N. R. Mahapatra *et al.* 2005. "A limit study on the potential of compression for improving memory system performance, power consumption, and cost," J. Instruction-Level Parallelism, vol. 7, pp. 1–37, July.