An Enhanced CAM Architecture to Accelerate LZW Compression Algorithm

Rupak Samanta and Rabi. N. Mahapatra
Department of Computer Science, Texas A&M University
rupak@cs.tamu.edu, rabi@cs.tamu.edu

Abstract

This paper presents efficient hardware architecture for Lempel-Ziv-Welch (LZW) data compression algorithm that can perform both encoding and decoding operations simultaneously using a CAM array. An enhanced CAM cell design has been proposed to achieve search and twofold store operations in single access during regular match operations. The proposed architecture utilizes these enhanced CAM cells to accelerate the implementation of the LZW algorithm. The performance of the proposed design is evaluated using the Corpus benchmarks, where on an average a performance improvement of **53x** *is achieved when compared to the software approach.*

1. Introduction

Lossless data compression methods are widely used to reduce data overhead in communication and storage. Redundant information present in the source data are removed in a way such that original data can be reconstructed without any loss of information. Data compression has been one of the major areas of research during last decade as a result of which many lossless data compression algorithms such as Lempel-Ziv (LZ) [8], Huffman coding [3] and arithmetic coding are in use today.

Huffman coding is one of the most popular compression algorithms that translate fixed-size input data to variable length symbols in the dictionary. However, the major drawback of the Huffman coding lies in its dependency on input data pattern to construct its dictionary. In LZW-based data compression the dictionary is constructed dynamically without any prior knowledge of source data. The LZW data compression algorithm is used extensively in many applications such as GIF, TIFF, ZIP and postscript level 2.

The LZW algorithm requires sequential construction of a dictionary and involves extensive comparison of input data with the dictionary content during compression. Most of the existing implementation sequentially compares input data with the dictionary contents. This sequential comparisons turns out to be a bottleneck of the compression throughput. To improve the throughput of LZW algorithm the hashing and tree based search techniques are being used to fetch data from dictionary for comparison. Software hashing is slow and may not yield optimal result. Several hardware implementations of LZW algorithm have been reported to speedup the data compression performance [8, 7]. While the author in [8] uses SRAM to store the dictionary, a CAM based dictionary has been employed by authors in [7]. The SRAM based implementation uses a hardware hashing technique in the encoder to fetch data from the dictionary. This hashing scheme incurs delay of few SRAM cycle in accessing the dictionary, thereby increasing the compression latency. The decoder in SRAM based implementation store their dictionary in SRAM to reproduce the characters from the input code. The decoding operation has to iterate through several SRAM accesses before the last character of a decoded string is found. Thus the decoders can take several SRAM cycles to output the original character string. Though the authors in [7] have proposed a CAM based dictionary implementation, they have focused on adaptive update of dictionary table to improve the compression ratio of the LZW algorithm. They implement the encoder dictionary table on a CAM. The decoder in their scheme also uses the same CAM array for its dictionary, but the access to the dictionary data is through an address decoding scheme similar to the address decoding scheme used in a SRAM. Thus decoding takes more time to reproduce data thereby increases the total time of compression. Since they use same dictionary table for encoding and decoding, the encoder and decoder operations has to be sequential.

In this paper, we propose a novel CAM based architecture to enhance processing speed of LZW algorithm. Unlike the approach in [7], we implement both the encoder and decoder on CAM to achieve the compression operation. In our approach we integrate both encoding and decoding operations that can operate in parallel on the same hardware block thereby eliminating the requirement for separate encoder and decoder units. In order to facilitate this operation, we propose an enhanced CAM cell that performs search and double store operation in addition to the original match operation. The performance of the above integrated approach has been validated using Corpus benchmark [1] used for standard compression algorithm. The results show about 53 times performance gain over the software counterpart. Also a performance gain of 6 times was achieved as compared to the SRAM-based and 2 times compared to the CAM-based hardware approach. Also the scheme proposed in [7] can always be integrated with our approach to improve the compression ratio.

The rest of the paper is organized as follows. Section 2 describes background and related work. Section 3 introduces the enhanced CAM cell. The details of S^3CAM architecture is described in section 4. Section 5 presents the experimental results. Section 6 concludes the paper and mention the future work.

2. Background and Related Work

The LZW coding takes a greedy approach and divides the texts into substrings. The LZW algorithm has two subroutines i.e. compression and decompression. These are restated below to understand the design of the architecture.

2.1 LZW Compression

The original algorithm for LZW encoding is discussed in [8] and is repeated in Figure 1.

```
1. Initialize table to contain single character strings.
2. Read first input character $\rightarrow$ prefix string $\omega$
3. Read the next input character K
   3.1. If no such K (input exhausted):
            code($\omega$) $\rightarrow$ output; EXIT
     3.1.2. If $\omega$K exists in string table:
               $\omega$K $\rightarrow$ $\omega$; repeat Step
            Else $\omega$K not in string table;
               code ($\omega$) $\rightarrow$ output;
     3.1.3. $\omega$K $\rightarrow$ string table
            K $\rightarrow$ $\omega$; repeat Step
```

Figure 1: **LZW Compression Algorithm**

The dictionary table is initialized to contain all single character strings. If a match is found the code (i.e. index) is stored as prefix string (ω). The next character is read and the substring consisting of prefix string (ω) and next character (K) is searched to find a match. If there is a mismatch the prefix string is sent to the output and the input character (K) becomes the first character of the next string. On a match the string (ωK) is assigned to the prefix string. The characters are read until all the inputs are exhausted.

2.2 LZW Decompression

The algorithm for LZW decoding is described in [8] and is repeated in Figure 2 for quick reference.

```
1. Decompression:   Read first input code
     1.1. CODE $\rightarrow$ OLDCODE;
     1.2. With CODE=code(K); K $\rightarrow$ output;
2. Next Code:   Read next input code
     2.1. CODE $\rightarrow$ INCODE;
     2.2. If no new code: EXIT; else
3. Next Symbol:
     3.1. If CODE=code ($\omega$K); K $\rightarrow$ output
          Code($\omega$) $\rightarrow$ CODE;
          Repeat Next Symbol
     3.2. Else if CODE=code(K); K $\rightarrow$ output;
          OLDCODE, K $\rightarrow$ String Table;
          INCODE $\rightarrow$ OLDCODE;
          Repeat Next Code;
```

Figure 2: LZW Decompression Algorithm

The decompression algorithm uses two registers OLDCODE and INCODE. The first input is read and stored in the OLDCODE. If a match is found for the code, the character is sent to the output. Next input is read and searched again for a matching string. If a match is found and the string consists of single character the OLDCODE and the string is stored in the string table and the INCODE register is copied to OLDCODE. On the other hand for a match consisting of a code and a single character, the character is then sent to the output and the code is searched again for another match. The process is repeated till a single character match is found. The next code is read and the process continues.

We have profiled the LZW algorithm to get an idea about the percentage of time each of the subroutine runs. The profiling information for E.coli application obtained from Corpus [1] is shown in Table 1. The detailed description of each subroutine can be found in Section 5. From Table 1 we can observe that the *find_match* routine takes 58% of the total execution time. *Find_match* is the dictionary matching operation and can be implemented on CAM hardware to accelerate the search.

With this motivation in mind, we devise a new CAM cell called Enhanced CAM (S^3CAM) that provides additional features for searching and dual storage in addition to its regular matching operation. The CAM array built upon this new cell can perform both the encoding and decoding operations on a single CAM array thereby eliminating the need to build separate blocks for the encoders and decoders. Also, the parallel search in the CAM cell provides an O (1) time search per symbol.

3. Enhanced CAM Cell (S^{3}CAM)

The proposed LZW hardware architecture employs an enhanced CAM cell that does not require maintaining separate dictionary tables for encoder and decoder. Thus a single table made up of S^{3}CAM cells can support both encoding and decoding operations. The transistor level details of S^{3}CAM are shown in Figure 3. The modifications to the standard CAM cell are highlighted in three different colors i.e. blue, red and green. The part of the circuit highlighted in blue performs the store operation for the decoder; the red portion of the circuit is used to store the index in an encoder and the transistors shown in green perform the match operation for the decoder. In the following sections we describe the design details of such enhancement and its operations.

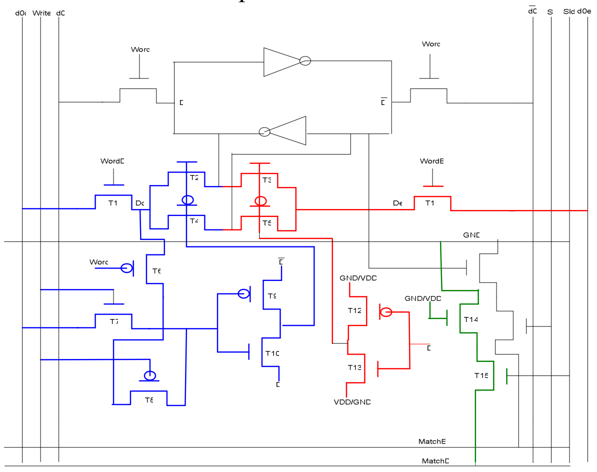

Figure 3: A S^{3}CAM Cell

The S^{3}CAM cell shown in Figure 3 provides three simultaneous operations i.e. a Search and a double store in addition to the traditional match operation. The first store operation is for the decoder to store characters in the dictionary table. The second store operation is needed to store the index of each row and output the index value during the encoding process. S^{3}CAM cell consists of basic SRAM and a comparison circuit as shown in black, and the enhanced features are imposed upon the circuit through the transistors on various colors. The 'd0d' signal in Figure 3 is the data wire for the decoder that is used to store and retrieve data during decoding operation while 'write' signal enables the decoder data to be written to the SRAM in the CAM cell. The signal 'WordD' activates the word line of the decoder while signal 'wordE' activates the word line for the encoder to output the index value. 'Sld' is the search line for the decoder search operation. The content of the latch D ($\overline{D}$) in the CAM cell is used conveniently as per the requirements in the three basic operations. The details of all the three operations are described as follows.

3.1 Store operation in the Decoder

As mentioned before, both the encoder and decoder use the same S^{3}CAM array for building their dictionary table during encoding and decoding operation. The S^{3}CAM cell shown Figure 3, has one SRAM which is used to store the data bits for encoder dictionary table. Since the encoding and decoding operations are done in parallel, we need to devise a scheme to store the data bit for the decoder dictionary table without affecting the content of the encoder dictionary table. This is achieved by using the blue components of the circuit shown in Figure 3. Since a '0' or a '1' is stored either at D or $\overline{D}$ of SRAM, we can connect signal D_d (i.e. decoder data line) dynamically either to D or $\overline{D}$ to retrieve the correct data for the decoder. Thus SRAM acts as a virtual storage for the decoder dictionary table.

There are two possible cases here. In the first case, i.e. when a decoder wants to store data, it will do so by placing a data bit on 'd0d'. The 'write' signal is asserted, to enable the content of 'd0d' to be compared with the bit stored in SRAM i.e. at D. The 'write' signal turns on the transistor T_7 thereby allowing the bit on 'd0d' to be compared with D through the transistors T_9 and T_{10}. The comparison result enables one of the two transistors T_2 or T_4 thereby connecting D_d to either D or $\overline{D}$, which is the input value needed to be stored at D_d. When the 'wordD' signal goes high the decoder data can be retrieved correctly. The 'wordD' signal is connected to the match line of the decoder. Once there is a match for the corresponding row, the 'wordD' signal goes high and data appears on the data line. The transistors T_6 and T_8 act as feedback circuit and provide correct enable signal for T_2 and T_4 even after the 'write' signal is disabled. In the second case, the encoder may update its content when a new substring needs to be stored in the dictionary table. To store the data in the SRAM of the CAM cell the 'word' line is asserted which turns off the transistor T_6 to ensure that the input at the comparator is the previous value stored at the input of transistor T_1 i.e. original decoder value stored at D_d. As soon as D changes the new comparison result will enable one of transistors T_2 or T_4 to connect the data line of the decoder i.e. D_d, correctly to D or $\overline{D}$. After the 'word' line is disabled T_6 and T_8 provide feedback circuit as described above.

3.2 Store operation in the Encoder

The encoder outputs a code when a substring is stored in any row of its dictionary. Generally the code (i.e. index of each row) is stored in the registers which get activated when the word line of the corresponding row is asserted to store a substring. Since the index of each row is fixed, we can permanently fix the index for each row by connecting required transistors to either ‘0’ or a ‘1’. The part of S^3CAM circuit shown in the red color is the component responsible for the store operation. Transistors T_{12} and T_{13} are connected permanently to either ‘0’ or ‘1’ depending upon whether the corresponding index bit is ‘1’ or ‘0’. For encoder a substring is stored by asserting the word line high which stores a bit in SRAM of each S^3CAM in that row. As soon as the bit is stored, the comparator circuit (transistor T_{12} and T_{13}) compares the contents of D with the index bit for each cell which is permanently written to the transistors T_{12} and T_{13}. The result of the comparison enables one of the transistors T_3 or T_5 so that the proper output appears at D_e. The index can be available at the output once ‘wordE’ is asserted high

3.3 Search operation in the Decoder

The code to be searched for the decoder is the index of a row i.e. the output of an encoder. The decoder search is achieved by the green components shown in the S^3CAM cell of Figure 3. Since the index of the decoder string table remains constant for a row, we can set it permanently. Thus the transistor T_{14} is permanently connected to ‘0’ or ‘1’ depending upon the index value for this cell which is either a ‘1’ or ‘0’. To search for a code, the code bit is asserted on ‘Sld’ line, and the ‘MatchD’ line will be held high if a match occurs.

4. Proposed Architecture

A high level architecture for LZW algorithm implementation is shown in Figure 4. We will focus on the discussion of implementing LZW encoder and decoder using S^3CAM architecture.

4.1 LZW Encoder

The S^3CAM array is initialized with single character string. The first character in the string is read and the corresponding code is output after searching for a match in the dictionary table i.e. S^3CAM array. Since S^3CAM array search can be done in parallel, the search for a match in the dictionary takes O (1) time. On arrival of next input character i.e. K the new substring ωK is stored in the encoding register, where ω is the code for previous matching string. The substring ωK is searched in the S^3CAM array. If a match occurs as indicated on ‘matchE’ line, the index of matching position is stored in the 12 MSB bits of the encoding register. On the other hand if none of the element stored in the dictionary matches, the index of the previous matching string i.e. index stored in encoding register is sent as code to the output and the index of the previous input character is stored in the encoding register. The substring for which no match was found is stored in the next vacant position in the S^3CAM. The next input is read and the process is repeated till all the input characters are exhausted. We have implemented an incremental counter that points to the next vacant position in S^3CAM. The counter is incremented each time a substring is stored in the S^3CAM array.

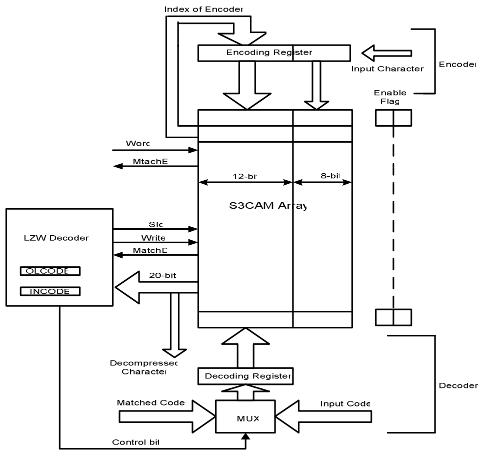

Figure 4: S^3CAM based LZW Architecture

4.2 LZW Decoder

For LZW decoding operation we employ a LZW decoder as shown in Figure 4. The LZW decoder consists of two registers OLDCODE, INCODE and a 12-bit comparator. The first input code to be decoded is stored in the decoding register and the OLDCODE register. This operation is similar to the first line of the decoding algorithm shown in Section 2.2. The code stored in the decoding register is searched in the S^3CAM array and the matched character is sent to the output. The search operation for decoder is similar to the search described in Section 3.3. The next input code is read and stored both in the decoding register and in the INCODE register. The code value stored in the decoding register of the S^3CAM is searched to find a match. If the matched output is a single character (i.e. 12 MSB bits are all 0’s) then the code stored in OLDCODE and the last character matched (i.e. K) is stored in the next vacant position in the S^3CAM. This is similar to the store operation for decoder as discussed in Section 3.1. The INCODE register is then copied to OLDCODE register and the next code is read. On the other hand if the matched output consists of a character

Name	% Time	Cumulative Seconds	Self seconds	Calls	Self ms/call	Total ms/call
find_match	57.27	0.36	0.36	4638689	0.00	0.00
Expand	17.42	0.48	0.12	1	210.37	210.37
Compress	13.07	0.57	0.09	1	480.84	480.84
decode_string	9.44	0.64	0.07	836654	0.00	0.00
output_code	4.36	0.67	0.03	836657	0.00	0.00
input_code	3.63	0.69	0.03	836656	0.00	0.00

Table 1: Profile Information of E.Coli

and a code then the character is sent to the output and the code is stored in the decoding register for next search. The search continues till a single character match is found. The multiplexer (MUX) shown in Figure 4 allows either the input code or the matched code to be stored in the decoding register for search in the S^3CAM. The multiplexer is controlled by the comparator output of the LZW decoder. The comparator consists of 12 XOR gates where one input to each gate is '0' while the other input is one of the 12 MSB bits of the matched output.

There are two enable flags for each row, one for the encoder and the other is for the decoder. The enable flag of a row is set when a string is written to the corresponding row. It acts as a mask for the match line for proper operation of the circuit. In the S^3CAM array a single character string is appended with 12 0's before it is stored in the array.

5. Experimental Results

To validate the S^3CAM based architecture, we evaluate its performance on standard compression files obtained from Corpus [1]. We compare the performance of the proposed architecture with the performance of software LZW implementation, the SRAM based hardware approach [8] and the CAM based approach [7].

The total execution time of the LZW algorithm in the software approach was obtained by running the LZW code for each benchmark on a Linux platform running on a 1GHz Pentium processor. The source code for LZW compression and decompression was obtained from [6]. We also obtained the profile information for each application in order to determine the number of accesses to individual routine while performing LZW compression and decompression. The profile information for E. coli application (one of the Corpus benchmark) is shown in Table 1. We exclude the profile information of rest of the applications in the benchmark due to space constraint. As seen in Table 1, the LZW implementation consists of six routines i.e. find_match, compress, expand, output code, input_code and decode_string. For each routine the table provides information on the percentage of total running time used by the routine, cumulative time in seconds, self second i.e. the number of seconds accounted for by this function alone, total number of times the routine was evoked, self milli second spent by the routine and total milli second spent by itself and its descendant routines. Out of six routines *compress* and *expand* are the main routines used for compression and decompression respectively. *Find_match* is referenced inside the compress routine and is used to find a match for the substrings with the entries stored in the dictionary. The proposed method uses S^3CAM array to implement the *find_match* routine in hardware. *Output_code* routine is called inside the *compress routine* and is used to output code for the longest matching substring. In the proposed architecture the encoding register is used to store the code i.e. index for the longest matching substring. The *input_code* subroutine is called inside the *expand* routine to input the code to be decompressed. This function is implemented by the decoding register in the proposed architecture. *Decode_string* is referenced inside the *expand* routine and decompresses the input code to string of characters. This operation is achieved by searching in the S^3CAM array for the decoder.

The results of the proposed S^3CAM architecture were obtained for a string length of 20-bits i.e. 12-bit code and 8-bit character. Thus S^3CAM array consists of 4096 entries in the string table. We performed CACTI simulation [9] on a fully associative CAM array having 4096 entries. Since the critical path for search and store operation remains the same as that of the traditional CAM cell, the access time of S^3CAM array can be assumed to be the same as that of CAM array having equivalent number of tag entries. The tag access time for a CAM array of 4096 entries was found to be 3.5ns for a 0.18 μ technology from the CACTI simulation. Thus, we can safely assume the access time of S^3CAM to be 4ns. To calculate the time taken to complete the LZW algorithm for the proposed architecture, we multiply S^3CAM access time with the number of calls for a find_match operations obtained from the profile data.

Name	Software	SRAM	CAM	S^{3}CAM
PFF5	0.09	0.011	0.0024	0.002
Kenedy	0.25	0.023	0.0078	0.0041
E.Coli	0.69	0.1045	0.028	0.018
Bible	0.76	0.091	0.032	0.0161
World192	0.55	0.055	0.024	0.009
Pi	0.22	0.022	0.0081	0.003
Book1	0.17	0.0170	0.0065	0.003
Book2	0.12	0.013	0.0056	0.0024
Pic	0.06	0.011	0.0024	0.002
News	0.09	0.008	0.0036	0.0015

Table 2: Comparison of S^{3}CAM Performance against software and hardware approaches (sec)

The operations like storing in a register and sending the compressed code to the output can be done in parallel to the S^{3}CAM search, thus the delays incurred by these operations are not included in the calculation of total time. The comparator needed for the LZW decoder is sequential with the S^{3}CAM search. Since comparator incurs one XOR gate delay which is very small as compared to the S^{3}CAM search time, we can neglect the comparator delay from the total time. Also, encoding and decoding are done in parallel in S^{3}CAM architecture, thus the effective time for an application requiring multiple encoding and decoding operations simultaneously is the maximum of the time taken either by the encoding or decoding operations. Generally, encoding of LZW algorithm takes more time than the decoding, thus the effective time for algorithm is the total time taken for the encoding operation. On the other hand for software approach the encoding and decoding are sequential. However, for SRAM based approach [8], the encoders and decoders use separate blocks of hardware, thus the total time is the time required for encoding. The total time for the algorithm is calculated by multiplying the number of *find_match* calls with the time taken to encode each symbol (i.e. 1.5 SRAM cycle), 22.5ns considering SRAM access time to be 15ns. For CAM based approach [7], since encoder and decoder share the same dictionary table, both encoding and decoding has to operate serially. Thus we add the encoding time i.e. CAM access time found from CACTI as 3.5ns and the decoding time i.e. SRAM access time of 15ns.

Table 2 shows the total time required to run the LZW algorithm for 10 benchmarks. The S^{3}CAM architecture achieves on an average a performance gain of 53 times as compared to the software approach. As evident from the table, the proposed architecture achieves a performance gain of about 6 times as compared to the SRAM approach and 2 times as compared to the CAM based approach.

6. Conclusion

In this paper we proposed a modified CAM cell that can handle both the encoding and decoding operations required for LZW Data compression algorithm simultaneously on a single unit thereby eliminating the need to implement encoder and decoder on separate blocks. A S^{3}CAM based architecture has been proposed to accelerate the LZW data compression algorithm. The performance of the architecture has been evaluated using Corpus benchmarks. The results show about 53 times performance gain over the software counterpart. Also a performance gain of 6 times was achieved as compared to the SRAM-based and 2 times compared to the CAM-based hardware approach.

References

[1] The Canterbury Corpus. http://corpus.canterbury.ac.nz/index.html

[2] W.J. Huang, N.R. Saxena and E. J. McCluskey, " A Realizable Data Compressor on Reconfigurable Coprocessor," *FCCM'00 Symposium on Field-Programmable Custom Computing Machines,* pp. 249-258, Napa Valley, CA, April 16-19, 2000.

[3] D. A. Huffman, "A method for construction of Minimum Redundancy Code," *Proceedings of IRE*, vol. 40, pp. 1098-1102, 1952.

[4] K. Pagiamtzis and A. Sheikholeslami, "A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 9, , pp. 1512–1519, September 2004

[5] A. J. McAuley and C. J. Cotton, "A reconfigurable content addressable memory," *Custom Integrated Circuits Conference, 1990., Proceedings of the IEEE* , pp. 24.1/1 – 24.1/4,13-16 May 1990

[6] M. Nelson, "LZW Data Compression," *Dr Dobbe's Journal*, October 1989. http://www.dogma.net/markn/articles/lzw/lzw.htm

[7] Chauchin Su, Chenq-Fan Yen and Jang-Chuang Yo, "Hardware Efficient Updating Technique for LZW CODEC Design," *IEEE International Symposium on Circuits and Systems*, pp. 2797-2800, volume. 4, 9-12 June1997.

[8] T. A. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, pp. 8-19, June, 1984.

[9] S. J. Wilton and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, volume.31, issue.5, pp. 677-688, May 1996