A Multilevel Hardware Architecture for Lossless Data Compression Applications

Ming-Bo Lin, and Jiun-Woei Chen
Department of Electronic Engineering
National Taiwan University of Science and Technology
43, Keelung Road, Section 4, Taipei (106), Taiwan
email: mblin@et.ntust.edu.tw

ABSTRACT

In this paper, we propose a new multilevel hardware architecture that combines the features of both Huffman and PDLZW algorithms. In particular, the memory allocation and the data unit of compression is changed from byte, which is used by PDLZW algorithm, to byte stream. In addition, the tree-based dynamic update method used in adaptive Huffman algorithm is changed to an order list for speeding up the compression rate. The resulting architecture shows that it can reach the same compression ratio as adaptive Huffman algorithm but only at the cost of one-half hardware resource. Furthermore, the compressing data rate is one codeword per cycle rather than one bit per cycle.

Keywords: *Adaptive Huffman algorithm, adaptive Huffman algorithm using transposition, canonical Huffman coding, lossless data compression, lossy data compression, PDLZW algorithm.*

1 Introduction

Data compression is a method of encoding rules that allows substantial reduction in the total number of bits to store or transmit a file. Two basic classes of data compression are applied in different areas currently [1]. One of these is lossy data compression that is widely used to compress image data files for communication or archives purposes. The other is lossless data compression that is commonly used to transmit or archive text or binary files that required to keep their information intact at any time.

Lossless data compression algorithms include mainly LZ codes [9, 10]. A most popular version of LZ algorithm is called LZW algorithm [8], which is a dictionary-based method. However, it requires quite a lot of time to adjust the dictionary. To improve this, two alternative versions of LZW were proposed. These are DLZW (dynamic LZW) and WDLZW (word-based DLZW) [2] algorithms. Both improve LZW algorithm in the following ways. First, they initialize their dictionaries with different combinations of characters instead of single character of the underlying character set. Second, they use a dictionary hierarchy of which the word widths are successively increased. Third, each entry in their dictionaries associates a frequency counter. That is, the LRU policy is used. It was shown that both algorithms outperform LZW [2]. However, they also complicates the hardware control logic.

To reduce the hardware required for VLSI implementation, a simplified DLZW architecture called PDLZW (parallel dictionary LZW) [4] is proposed. This architecture improves and modifies the features of both LZW and DLZW algorithms in the following ways. First, instead of initializing the dictionary with single character or different combinations of characters a virtual dictionary with the initial $|\Sigma|$ address space is reserved. This dictionary only takes up a part of address space but costs no hardware actually. Second, a hierarchical parallel dictionary set with successively increasing word widths is used. Third, the simplest dictionary update policy called FIFO (first-in first-out) is used to simplify the hardware implementation. The resulting architecture shows that it outperforms Huffman algorithm in all cases and about only 5% below UNIX *compress* on the average case but in some cases outperforms the *compress* utility.

However, as demonstrated in [4], the size of the dictionary set used in the PDLZW still requires 3072 bytes which may be inhibited in some area-constrained applications since the dictionary set costs too much silicon area. Therefore, in this paper, we will propose a new multilevel data compression architecture that combines features from both adaptive Huffman and PDLZW algorithms. The resulting architecture shows that to achieve the same compression ratio as that of adaptive Huffman algorithm is only requires one-half the

hardware cost. In addition, both compression and decompression rate are greater than those of the adaptive Huffman algorithm.

The rest of the paper is organized as follows. Section 2 describes features of both adaptive Huffman and PDLZW algorithms. Section 3 describes in detail the proposed multilevel architecture. Section 4 presents the performance of the new architecture. Section 5 concludes the paper.

2 Related Work

Before presenting our new architecture for data compression VLSI chip, both the features and limitations of the hardware implementations of PDLZW and adaptive Huffman algorithms must be discussed in advance.

2.1 PDLZW algorithm

As described in [4], the essence of PDLZW compression algorithm is based on a parallel dictionary set that consists of m small variable-word-width dictionaries, numbered from 0 to $m-1$, with each of which increases its word width by one byte. More precisely, dictionary 0 has one byte word width, dictionary 1 two bytes, and so on. The actual size of dictionary set used in a given application can be determined by the information correlation property of the application. To facilitate a general PDLZW architecture for a variety of applications, it is necessary to do a lot of simulations based on the information correlation property of these applications for determining an optimal dictionary set. However, the PDLZW algorithm proposed in [4] actually suggests a class of data lossless compression algorithms. The more detailed discussion about how to determine the dictionary set of the algorithm and the algorithm, please refer to [4].

In general, different address space distributions of the dictionary set will present significantly distinct performance of the PDLZW compression algorithm. However, the optimal distribution is strongly dependent on the actual input data files. Different data profiles have their own optimal address space distributions. Therefore, in order to find a more general distribution, several different kinds of data samples are run with various partitions of a given address space. Each partition corresponds to a dictionary set. For instance, the $1K$ address space may be partitioned as: {256, 256, 128, 128, 64, 64, 64, 64}.

In general, the compression ratio, which is defined as the ratio of the bit number required for representing compressed data and that for original data, is improved as the address space of dictionary increases. Thus, the algorithm with 4-K address space has the best average compression ratio in all cases that we have simulated [4]. However, as we examine the partitions in different address spaces, each exhibits some optimal partitions. For instance, in the case of 1-K address space, the partitions with {256, 256, 128, 128, 128, 64, 64} and {256, 256, 128, 128, 64, 64, 64, 64} are optimal and have the same compression ratio of 54%. In comparison with three different address spaces, the best compression ratio is 54% and appears in both 1-K and 2-K address spaces, respectively. As a consequence, the compression ratio is not only determined by the correlation property of underlying data files to be compressed but also depends on an appropriate partition.

An important consideration for hardware implementation is the required dictionary address space that dominates the chip cost for achieving an acceptable compression ratio. From this point, the optimal solution of dictionary address space to be used in [4] is 1-K address space with partition: {256, 256, 128, 128, 64, 64, 64, 64}.

To further reduce the size of the dictionary set of PDLZW algorithm while keeping the compression ratio above an accepted level, in this paper, we explore the possibility of using a multilevel architecture. This is due to that the compression ratio is deteriorated significantly when the size of the dictionary set is too small. To compensate the loss of compression ratio, a second stage is used to encode statistically the fixed-length code output from PDLZW algorithm into a variable-length one. The rational behind this is based on the observation that the output codewords from PDLZW algorithm are not uniform distributed but each codeword has its own occurrence frequency that depends on the distribution of input data. Up to now, one of the most common used algorithms for converting fixed-length code into variable-length one is adaptive Huffman algorithm. However, it is not easy to realize it in VLSI technology since the frequency count associated with each symbol requires a lot of hardware to implement and time to maintain.

In the following subsection, we will describe a modified adaptive Huffman algorithm that is easy to realize in VLSI technology.

2.2 Adaptive Huffman algorithm using transposition

The Huffman algorithm requires both the encoder and the decoder to know the frequency table of symbols relating to the data being encoding. To avoid building the frequency table in advance, an alternative method called adaptive Huffman algorithm [3] allows the encoder and the decoder to build the frequency table dynamically according

to the data statistics up to the point being encoding and decoding.

The essence of implementing adaptive Huffman algorithm in hardware is centered around how to build the frequency table dynamically. Several approaches have been proposed [6, 7]. These approaches are usually based on tree structures on which LRU policy is applied. However, the hardware cost and the time required to maintain the frequency table dynamically are too complicated to realize in VLSI technology. To alleviate this, an approximate version of adaptive Huffman algorithm, called AHAT (adaptive Huffman algorithm using transposition), is proposed in [6]. In the algorithm, an ordered list instead of the tree structure is used to maintain the frequency table required in adaptive Huffman algorithm. More precisely, an index corresponding to an input symbol, say n, of the ordered list is searched and output when receiving it and then swap both items located in n and $n-1$, respectively. Thus, the higher occurrence frequency symbol will "bubble up" to the top of the ordered list and we can code the indices of these symbols by using a variable-length code to take their occurrence frequency into account and to reduce the information redundancy.

By using a simple ordered list to memorize the occurrence frequency of symbols, both of the search and update time are significantly reduced from $O(log_2 n)$, which is required in tree structures, to $O(1)$, where n is the total number of input symbols.

2.3 Canonical Huffman code

To facilitate a fast speed for both compression and decompression operations, it is necessary to take the code assigned to the symbol set into account since there exists many codes corresponding to the Huffman tree of a given input data. A fast decoding technique for Huffman code is proposed in [5]. The approach divides compressed codewords into a sequence of fixed-length bit strings, called groups. Each group consists of n bits. The decoding process is carried out much the same way as in the case of fixed-length codes except that several decoding tables are used instead of one. In this paper, we use the same idea to code the indices output from ordered list that contains the output codewords from PDLZW algorithm for speeding up both coding and decoding operations.

The Huffman tree corresponding to the output from PDLZW algorithm can be built by using offline adaptive Huffman algorithm. An example of the Huffman tree for input symbol set {A, B, C, D, E, F} is shown in Figure 1(a). Although the Huffman tree for a given symbol set is unique, such as Figure 1(b), the code assigned to the symbol set is not unique. For example, three codes of all possible codes for the Huffman tree is shown in Figure 1(a). In fact, there are 32 possible codes for the symbol set {A, B, C, D, E, F} since we can arbitrarily assign 0 and 1 to each edge of the tree.

For the purpose of easy decoding, it is convenient to choose the type three encoding scheme shown in Figure 1(a) as our resulting code in which symbols with consecutively increasing occurrence frequency are encoded as a consecutively increasing sequence of codewords. This encoding rule and its corresponding code will be called as canonical Huffman coding and canonical Huffman code, respectively, for the rest of the paper.

The general approach for encoding a Huffman tree into its canonical Huffman code is first to run the test data files using adaptive Huffman algorithm for generating Huffman code and then to use the following algorithm, called **Algorithm: Canonical Huffman Code Encoder,** to generate the corresponding canonical Huffman code.

Algorithm: Canonical Huffman Code Encoder

{Assume that each symbol s has $num\_bits[s]$ bits, no codeword is longer than $maxlength$, and n is the total number of symbols under consideration. }

Input: A set of symbols s and its length array $num\_bits[s]$.

Output: The canonical Huffman codeword for each symbol s.

Begin

1: **for** $i = 1$ **to** $maxlength$ **do**
$num\_codewords\_l[i] = 0$

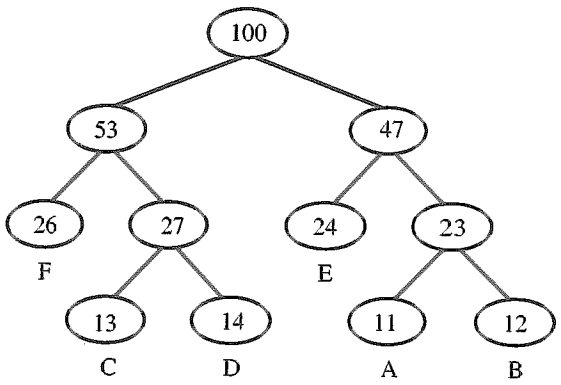
2: **for** $i = 1$ **to** n **do**
$num\_codewords\_l[num\_bits[i]] =$
$num\_codewords\_l[num\_bits[i]] + 1$
{The number of codewords of length l is stored in $num\_codewords\_l[l]$. }

3: {The codeword for the first codeword of length l is stored in $first\_codeword[l]$. }
3.1: **set** $first\_codeword[maxlength] = 0$;
$nextcode[maxlength] = 0$
3.2: **for** $i = maxlength - 1$ **downto** 1 **do**
$first\_codeword[i] = (first\_codeword[i+1]$
$+num\_codewords\_l[i+1])/2$;
$nextcode[i] = first\_codeword[i]$

4: **for** $i = 1$ **to** n **do** {The canonical code for symbol i is in $codeword[i]$; the rightmost $num\_bits[i]$ bits should be used. }
4.1: $codeword[i] = nextcode[num\_bits[i]]$
4.2: $nextcode[num\_bits[i]] =$
$nextcode[num\_bits[i]] + 1$

Symbol	Frequency	Encoding type		
		One	Two	Three
A	11	000	111	000
B	12	001	110	001
C	13	100	011	010
D	14	101	010	011
E	24	01	10	10
F	26	11	00	11

(a) Three possible encodings



(b) Huffman tree

Figure 1: An example of Huffman tree and its three possible encodings.

End {End of Canonical Huffman Code Encoder. }

As shown in Figure 1(a), in the beginning adaptive Huffman algorithm is used to compute the corresponding codeword length for each input symbol. Then it counts the number of codewords of the same length and saves the result into the array *num_codewords_l*[]. Finally, the starting values (or called *codeword_offset*) for each codeword group of the same codeword length are calculated from array *num_codewords_l*[]. Based on this procedure, the codeword length, *first_codeword*, number of codewords, and *codeword_offset* for the input symbols, consisting of the output codewords from PDLZW algorithm with dictionary set: {256, 64, 32, 16}, are calculated and shown in Table 1. Of course, different data distributions for the input to PDLZW algorithm will generate different PDLZW output code and hence different data set shown in Table 1. However, one of the main contributions of this paper is to propose a new multilevel architecture for lossless data compression applications which uses only a small-size dictionary.

3 Proposed Multilevel Architecture

The proposed multilevel architecture consists of two major components: a PDLZW processor and an AHAT processor, as shown in Figure 2. The former is composed of a dictionary with partition: {256, 64, 32, 16}. Thus, the memory size used in the processor is only 288 bytes. Please note that the dictionary 0 with address space of 256 bytes does not occupy a physical memory hardware. The latter is centered around an ordered list and requires 414 (= 368 × 9 bits) bytes CAM. Therefore, the total memory used is 702 bytes.

3.1 PDLZW processor

The major components of PDLZW processor are CAMs, a 4-byte shift register, and a priority encoder. The word widths of CAMs increase gradually from 2 bytes up to 4 bytes with three different address spaces: 64, 32, and 16 words, as shown in the figure.

The input string is shifted into the 4-byte shift register. Once in the shift register the search operation can be carried out in parallel on the dictionary set. The address along with a matched signal within a dictionary containing the prefix substring of the incoming string is output to the priority encoder for encoding the output codeword *pdlzw_addr*. This codeword is then encoded into canonical Huffman code by AHAT processor. In general, it is not impossible that many (up to four) dictionaries in the dictionary set containing prefix substrings of different lengths of the incoming string simultaneously. In this case, the prefix substring of maximum length is ruled out and the matched address within its dictionary along with the matched signal of the dictionary is encoded and output to the AHAT processor.

In order to realize the update operation of the dictionary set, each dictionary in the dictionary set except the dictionary 0 has its own update pointer (UP) that always points to the word to be inserted next. All UPs count from 0 up to its maximum value and then wrap back to 0. Hence, the FIFO update policy is realized. The update operation of the dictionary set is carried out as follows. The maximum length prefix substring matched in the dictionary set is written to the next entry pointed by the UP of the next dictionary along with the next character in the shift register. The update operation is inhibited if the next dictionary number is greater than or equal to the maximum dictionary number.

Table 1: The canonical Huffman code used in AHAT processor.

Codeword length	First_codeword	Number of codewords	Codeword_offset
6	29	35	35
7	45	13	48
9	20	160	208
12	0	160	368

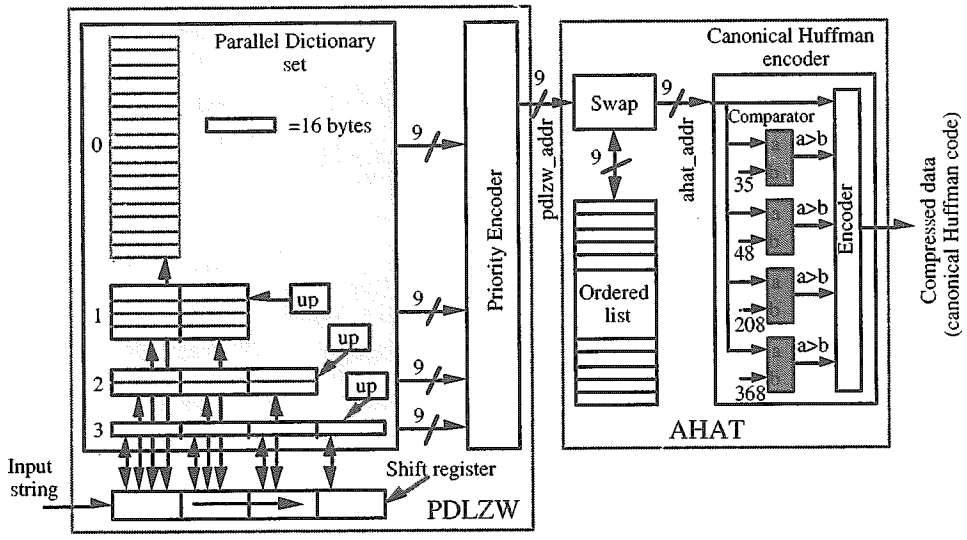


Figure 2: The architecture of proposed multilevel compression processor.

3.2 AHAT processor

The AHAT processor encodes the output codewords from PDLZW processor. As described previously, its purpose is to recode the fixed-length codewords into variable-length ones for taking the advantage of statistical property of the codewords from PDLZW processor and thus to remove the information redundancy contained in the codewords. The encoding process is carried out as follows. The *pdlzw_addr*, which is the output from PDLZW processor and is the "symbol" for AHAT algorithm, is input into *swap* unit for searching and deciding the matched index, n, from the ordered list. Then the *swap* unit exchanges both items located in n and $n-1$, respectively. That is, the more frequently used symbol bubbles up to the top of the ordered list. The index *ahat_addr* of the term *pdlzw_addr* of the ordered list is then encoded into a variable-length codeword (i.e., canonical Huffman codeword) and output as the compressed data for the entire processor.

The operation of canonical Huffman encoder is as follows. The *ahat_addr* is compared with all $codeword\_offset$, as shown in Table 1, 35, 48, 208, and 368 simultaneously, for deciding the length of the codeword to be encoded. Once the length is determined, the output codeword can be encoded as $ahat\_addr - code\_offset + first\_codeword$. For example, if $ahat\_addr = 38$, from Table 1, the length is 6 bits since 38 is greater than 35 and smaller than 48. The output codeword is: $38 - 35 + 29 = 32 = 100000_2$.

As described above, the compression rate is at least one and up to four bytes per memory cycle.

4 Performance

The proposed architecture is mainly to reduce the size of dictionary set used in PDLZW data compression processor described in [4]. However, the compression ratio will be reduced accordingly if the size is too small. In order to reduce the dictionary size while keeping the compression ratio above an accepted level, a multilevel architecture is used instead of single PDLZW processor.

Table 2 shows the compression ratio of adaptive Huffman algorithm (AHA), PDLZW + AHA, and PDLZW+AHAT. The dictionary set used in PDLZW is {256, 64, 32, 16}. From the table, the compression ratio of PDLZW+AHAT is competitive to that of AHA.

Because the cost of memory is a major part of any dictionary-based data compression processor discussed in the paper, we will use this as the base for comparing the hardware cost of different archi-

Table 2: Comparison of compression ratio of various architectures.

	exe. 1	exe. 2	exe. 3	text 1	text 2	doc.	graphics
PDLZW+AHAT	0.46	0.72	0.76	0.67	0.73	0.59	0.70
PDLZW+AHA	0.44	0.79	0.79	0.68	0.77	0.79	0.77
AHA	0.44	0.77	0.81	0.61	0.66	0.56	0.71

Table 3: Cost comparison of AHA and proposed architecture.

Architecture	Memory requirement	Total memory ($N = 256$)
AHA	$2(2N-2)$-B CAM + $(2N-2)$-B ROM + $(2N-2)$ ROM	1020-B CAM+514-B ROM
PDLZW+AHAT (proposed)	288-B CAM (PDLZW)+ 414-B CAM (AHAT)	702-B CAM

tectures. Table 3 shows the memory size required for AHA and PDLZW + AHAT.

5 Conclusion

In this paper, a multilevel VLSI architecture based on the combination of PDLZW compression algorithm and adaptive Huffman algorithm with transposition is proposed. The PDLZW processor is based on a hierarchical parallel dictionary set that has successively increasing word widths from 1 to 4 bytes with the capability of parallel search. The total memory used is only 288 bytes. The second processor is built around an ordered list consisting of 414(= 368 × 9 bits) bytes CAM and a canonical Huffman encoder. The resulting architecture shows that it is not only to reduce the hardware cost significantly but also easy to be realized in VLSI technology since the entire architecture is around the parallel dictionary set such that the control logic is essentially trivial. The simulation result shows that this architecture has the competitive performance with adaptive Huffman algorithm but is only at the cost of one-half that of adaptive Huffman algorithm in hardware. The data rate for the compression processor is at least one and up to four bytes per memory cycle. The memory cycle is mainly determined by the cycle time of CAMs but it is quite small since the maximum capacity of CAMs is only 64 × 2 bytes for PDLZW processor and 414 bytes for AHAT processor. Therefore, a very high data rate can be achieved.

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*, Englewood Cliffs:N. J. Prentice-Hall, 1990.

[2] J. Jiang and S. Jones, "Word-based dynamic algorithms for data compression," *IEE Proceedings-I,* Vol. 139, No. 6, pp. 582-586, December 1992.

[3] D. E. Knuth, "Dynamic Huffman coding," *Journal of Algorithms,* Vol. 6, pp. 163-180, 1985.

[4] Ming-Bo Lin, "A parallel VLSI architecture for the LZW data compression algorithm," *International Symposium on VLSI Technology, Systems, and Applications,* Taiwan, pp. 98-101, 1997.

[5] A. Sieminski, "Fast decoding of the Huffman codes," *Information Processing Letters,* Vol. 26, No. 5, pp. 237-241, 1988.

[6] B. W. Y. Wei, J. L. Chang, and V. D. Leongk, "Single-chip lossless data compressor," *International Symposium on VLSI Technology, Systems, and Applications,* Taiwan, pp. 211-213, 1995.

[7] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes Compressing and Indexing Documents and Imagess,* Chapter 9, 1994.

[8] Terry A. Welch, "A Technique for high-performance data compression," *IEEE Computer,* Vol. 17 No. 6, pp. 8-19, June 1984.

[9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Information Theory,* Vol. IT-23 No. 3, pp. 337-343, March 1977.

[10] J. Ziv and A. Lempel, "A compression of individual sequences via variable-rate coding," *IEEE Trans. Information Theory,* Vol. IT-24 No. 5, pp. 530-536, September 1978.