A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture

Ming-Bo Lin, *Member, IEEE*, Jang-Feng Lee, and Gene Eu Jan

***Abstract*—In this paper, we propose a new two-stage hardware architecture that combines the features of both parallel dictionary LZW (PDLZW) and an approximated adaptive Huffman (AH) algorithms. In this architecture, an ordered list instead of the tree-based structure is used in the AH algorithm for speeding up the compression data rate. The resulting architecture shows that it not only outperforms the AH algorithm at the cost of only one-fourth the hardware resource but it is also competitive to the performance of LZW algorithm (compress). In addition, both compression and decompression rates of the proposed architecture are greater than those of the AH algorithm even in the case realized by software.**

***Index Terms*—Adaptive Huffman (AH) algorithm, approximated adaptive Huffman algorithm, canonical Huffman coding, lossless data compression, lossy data compression, parallel dictionary LZW (PDLZW) algorithm.**

I. 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. Currently, two basic classes of data compression are applied in different areas [1], [3], [16]. One of these is lossy data compression, which 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 required to keep their information intact at any time.

Lossless data compression algorithms mainly include Lempel and Ziv (LZ) codes [23], [24], Huffman codes [2], [14], and others such as [13] and [16]. Most of the implementations of the LZ algorithm are based on systolic arrays [4], or its variant such as [25], which combines a systolic array with trees for data broadcast and reduction. An area-efficient VLSI architecture for the Huffman code can be found in [14]. In this paper, we do not consider the Huffman code due to its inherent feature of being required to know *a priori* probability of the input symbols. Instead, we consider a variant of the Huffman code called the adaptive Huffman code or the dynamic Huffman code [9], which does not need to know the probability of the input symbols in advance.

Manuscript received June 29, 2000; revised October 18, 2003, June 13, 2005, and December 9, 2005.

M.-B. Lin and J.-F. Lee are with the Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan, R.O.C. (e-mail: mblin@mail.ntust.edu.tw).

G. E. Jan is with the Department of Computer Science, National Taipei University, Taipei 237, Taiwan, R.O.C. (e-mail: gejan@mail.ntpu.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2006.884045

Another most popular version of the LZ algorithm is called the word-based LZ (LZW) algorithm [20], which is proposed by T. Welch and is a dictionary-based method. In this method, the second element of the pair $\langle i, c\rangle$ is removed, i.e., the encoder would only send the index to the dictionary. However, it requires quite a lot of time to adjust the dictionary. To improve this, two alternative versions of LZW were proposed. These are dynamic LZW (DLZW) and word-based DLZW (WDLZW) [7] 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 in which the word widths are successively increased. Third, each entry in their dictionaries associates a frequency counter. That is, the least recently used (LRU) policy is used. It was shown that both algorithms outperform LZW [7]. However, they also complicate the hardware control logic.

In order to reduce the hardware cost, a simplified DLZW architecture called parallel dictionary LZW (PDLZW) [10] is proposed. In this architecture, it also uses the hierarchical parallel dictionary set with successively increasing word widths as that of DLZW architecture but improves and modifies the other 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, where Σ represents the set of input symbol and $|\Sigma|$ is the number of elements of the set Σ. This dictionary only takes up a part of address space but actually has no cost in hardware. Second, the simplest dictionary update policy called first-in first-out (FIFO) is used to simplify the hardware implementation. The resulting architecture shows that it outperforms the Huffman algorithm in all cases, is only about 7% inferior to UNIX *compress* on the average cases, and outperforms the *compress* utility in some cases. The compress utility is an implementation of LZW algorithm with dictionary size varying from 512 to 64 kB dynamically.

However, as demonstrated in [10], the size of the dictionary set used in the PDLZW still requires 3072 B, which may be inhibited in some area-constrained applications since the dictionary set costs too much in the silicon area. Therefore, in this paper, we will propose a new two-stage data compression architure that combines features from both PDLZW and Adaptive Huffman (AH) algorithms. The resulting architecture shows that it outperforms the AH algorithm in most cases and requires only one-fourth of the hardware cost of the AH algorithm. In addition, its performance is competitive to the *compress* utility in the case of the executable files. Furthermore, both compression

and decompression rates are greater than those of the AH algorithm even in the case realized by software. Please note that the performance of the proposed algorithm and architecture in this paper is quite dependent on the dictionary size used in the PDLZW stage, which in turn determines the hardware cost of both PDLZW and the modified AH stages. Hence, the major design consideration is a tradeoff process between performance and hardware cost.

The rest of the paper is organized as follows. Section II describes features of both AH and PDLZW algorithms. In this section, many approximated AH algorithms and their relative performance are also proposed and analyzed. Section III discusses the tradeoff between the dictionary size used in the PDLZW algorithm and the resulting performance. Section IV describes the proposed two-stage architecture for lossless data compression in detail. In this section, we also present the performance of the new architecture. Section V concludes this paper.

II. Related Work

Before presenting our new hardware architecture for data compression, we first discuss both the features and limitations of the hardware implementations of the PDLZW and AH algorithms.

A. PDLZW Algorithms

Like the LZW algorithm proposed in [20], the PDLZW algorithm proposed in [12] also encounters the special case in the decompression end. In this paper, we remove the special case by deferring the update operation of the matched dictionary one step in the compression end so that the dictionaries in both compression and decompression ends can operate synchronously. The detailed operations of the PDLZW algorithm can be referred to in [12]. In the following, we consider only the new version of the PDLZW algorithm.

1) PDLZW Compression Algorithm: As described in [10] and [12], the PDLZW compression algorithm is based on a parallel dictionary set that consists of m small variable-word-width dictionaries, numbered from 0 to $m-1$, 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 the 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 for exploring information correlation property of these applications so that an optimal dictionary set can be determined. For a more detailed discussion about how to determine the dictionary set of the algorithm, please refer to [12].

The detailed operation of the proposed PDLZW compression algorithm is described as follows. In the algorithm, two variables and one constant are used. The constant max_dict_no denotes the maximum number of dictionaries, excluding the first single-character dictionary (i.e., dictionary 0), in the dictionary set. The variable $max\_matched\_dict\_no$ is the largest dictionary number of all matched dictionaries and the variable $matched\_addr$ identifies the matched address within the $max\_matched\_dict\_no$ dictionary. Each compressed codeword is a concatenation of $max\_matched\_dict\_no$ and $matched\_addr$.

Algorithm: PDLZW Compression

Input: The string to be compressed.
Output: The compressed codewords with each having $\log_2 k$ bits. Each codeword consists of two components: $max\_matched\_dict\_no$ and $matched\_addr$, where k is the total number of entries of the dictionary set.
Begin
1: Initialization.
  1.1: $string$-1 $\longleftarrow$ **null**.
  1.2: $max\_matched\_dict\_no \longleftarrow max\_dict\_no$.
  1.3: $update\_dict\_no \longleftarrow max\_matched\_dict\_no$;.
    $update\_string \longleftarrow \phi\{\text{empty}\}$.
2: while (the input buffer is **not empty**) **do**
  2.1: Prepare next $max\_dict\_no + 1$ characters for searching.
    2.1.1: $string$-2 $\longleftarrow$ read next. $(max\_matched\_dict\_no+1)$ characters from the input buffer.
    2.1.2: $string \longleftarrow string$-1 $\|$ $string$-2. {Where $\|$ is the concatenation operator.}
  2.2: Search $string$ in all dictionaries in parallel and set the $max\_matched\_dict\_no$ and $matched\_addr$.
  2.3: Output the compressed codeword containing $max\_matched\_dict\_no \,\|\, matched\_addr$.
  2.4: if ($max\_matched\_dict\_no < max\_dict\_no$ **and** $update\_string \neq \phi$) **then** add the $update\_string$ to the entry pointed by $UP[update\_dict\_no]$ of dictionary$[update_dict_no]$. {$UP[update_dict_no]$is the update pointer associated with the dictionary.}
  2.5: Update the update pointer of the dictionary$[max_matched_dict_no + 1]$.
    2.5.1: $UP[max\_matched\_dict\_no + 1] = UP[max\_matched\_dict\_no + 1] + 1$
    2.5.2: if $UP[max\_matched\_dict\_no + 1]$ reaches its upper bound **then** reset it to 0. {FIFO update rule.}
  2.6: $update\_string \longleftarrow$ extract out the first $(max\_matched\_dict\_no + 2)$ bytes from $string$; $update\_dict\_no \longleftarrow max\_matched\_dict\_no + 1$.
  2.7: $string$-1 $\longleftarrow$ shift $string$ out the first $(max\_matched\_dict\_no + 1)$ bytes.
End {End of PDLZW Compression Algorithm.}

An example to illustrate the operation of the PDLZW compression algorithm is shown in Fig. 1. Here, we assume that the alphabet set Σ is $\{a, b, c, d\}$ and the input string is $ababbcabbabbabc$. The address space of the dictionary set is 16. The dictionary set initially contains only all single characters: a, b, c, and d. Fig. 1(a) illustrates the operation of PDLZW compression algorithm. The input string is grouped together by characters. These groups are denoted by a number

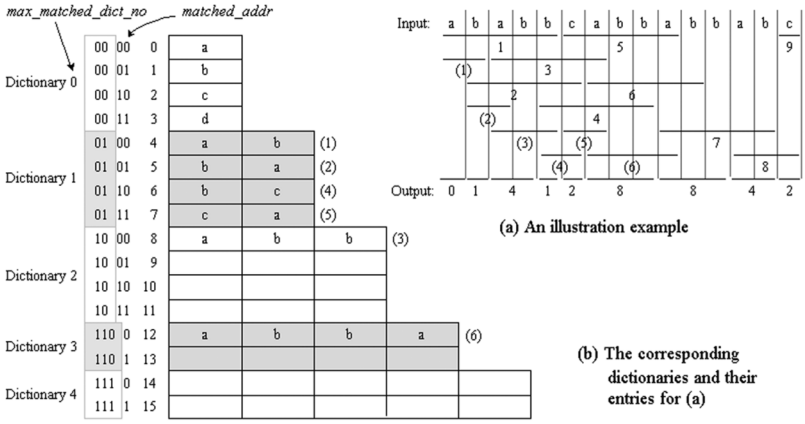


Fig. 1. Example to illustrate the operation of PDLZW compression algorithm.

with or without parenthesis "()." The number without parenthesis "()" denotes the order to be searched of the group in the dictionary set and the number with parenthesis "()" denotes the order to be updated of the group in the dictionary set. After the algorithm exhausts the input string, the contents of the dictionary set and the compressed output codewords will be: $\{a, b, c, d, ab, ba, bc, ca, abb, , , , abba, , , \}$, and $\{0, 1, 4, 1, 2, 8, 8, 4, 2\}$, respectively.

2) PDLZW Decompression Algorithm: To recover the original string from the compressed one, we reverse the operation of the PDLZW compression algorithm. This operation is called the PDLZW decompression algorithm. By decompressing the original substrings from the input compressed codewords, each input compressed codeword is used to read out the original substring from the dictionary set. To do this without loss of any information, it is necessary to keep the dictionary sets used in both algorithms, the same contents. Hence, the substring concatenated of the last output substring with its first character is used as the current output substring and is the next entry to be inserted into the dictionary set.

The detailed operation of the PDLZW decompression algorithm is described as follows. In the algorithm, three variables and one constant are used. As in the PDLZW compression algorithm, the constant max_dict_no denotes the maximum number of dictionaries in the dictionary set. The variable $last\_dict\_no$ memorizes the dictionary address part of the previous codeword. The variable $last\_output$ keeps the decompressed substring of the previous codeword, while the variable $current\_output$ records the current decompressed substring. The output substring always takes from the $last\_output$ that is updated by $current\_output$ in turn.

Algorithm: PDLZW Decompression

Input: The compressed codewords with each containing $\log_2 k$ bits, where k is the total number of entries of the dictionary set.
Output: The original string.
Begin

1: Initialization.
  1.1: if (input buffer is **not empty**) **then**
    $current\_output \longleftarrow$ **empty;** $last\_output \longleftarrow$ **empty**;
    $addr \longleftarrow$ read next $\log_2 k$-bit codeword from input buffer. {Where codeword $= dict\_no \,||\, dict\_addr$ and $||$ is the concatenation operator.}
  1.2: if (dictionary$[addr]$ is **defined**) **then**
    $current\_output \longleftarrow$ dictionary$[addr]$;
    $last\_output \longleftarrow current\_output$;
    $output \longleftarrow last\_output$;
    $update\_dict\_no \longleftarrow dict\_no[addr] + 1$.
2: while (the input buffer is **not empty**) **do**
  2.1: $addr \longleftarrow$ read next $\log_2 k$-bit codeword from input buffer.
  2.2: {output decompressed string and update the associated dictionary.}
    2.2.1: $current\_output \longleftarrow$ dictionary$[addr]$.
    2.2.2: if $(max\_dict\_no \geq update\_dict\_no)$ **then** add $(last\_output \,||$ the first character of $current\_output)$ to the entry pointed by $UP[update\_dict\_no]$ of dictionary$[update_dict_no]$.
    2.2.3: $UP[update\_dict\_no] \longleftarrow UP[update\_dict\_no] + 1$.
    2.2.4: if $UP[update\_dict\_no]$ reaches its upper bound **then** reset it to 0.
    2.2.5: $last\_output \longleftarrow current\_output$;
      $output \longleftarrow last\_output$;
      $update\_dict\_no \longleftarrow dict\_no(addr) + 1$.
End {End of PDLZW Decompression Algorithm.}

The operation of the PDLZW decompression algorithm can be illustrated by the following example. Assume that the alphabet set Σ is $\{a, b, c, d\}$ and input compressed codewords are $\{0, 1, 4, 1, 2, 8, 8, 4, 2\}$. Initially, the dictionaries numbered from 1 to 3 shown in Fig. 1(b) are empty. By applying the entire input compressed codewords to the algorithm, it will generate the same content as is shown in Fig. 1(b) and output the decompressed substring $\{a, b, ab, b, c, abb, abb, ab, c\}$.

B. AH Algorithm

The output codewords from the PDLZW algorithm are not uniformly distributed but each codeword has its own occur-

rence frequency, depending on the input data statistics. Hence, it is reasonable to use another algorithm to encode statistically the fixed-length code output from the PDLZW algorithm into a variable-length one. Up to now, one of the most commonly used algorithms for converting a fixed-length code into its corresponding variable-length one is the AH algorithm. However, it is not easily realized in VLSI technology since the frequency count associated with each symbol requires a lot of hardware and needs much time to maintain. Consequently, in what follows, we will explore some approximated schemes and detail their features.

1) Approximated AH Algorithm: The Huffman algorithm requires both the encoder and the decoder to know the frequency table of symbols related to the data being encoding. To avoid building the frequency table in advance, an alternative method called the AH algorithm [9], allows the encoder and the decoder to build the frequency table dynamically according to the data statistics up to the point of encoding and decoding.

The essence of implementing the AH algorithm in the hardware is centered around how to build the frequency table dynamically. Several approaches have been proposed [19], [22]. These approaches are usually based on tree structures on which the LRU policy is applied. However, the hardware cost and the time required to maintain the frequency table dynamically are not easy to be realized in VLSI technology. To alleviate this, the following schemes are used to approximate the operation of the frequency table. In these schemes, an ordered list instead of the tree structure is used to maintain the frequency table required in the AH algorithm. An index corresponding to an input symbol stored in the list, say n, of the ordered list is searched and output by the AH algorithm when the input symbol is received. The item associated with the index n is then swapped with some other item in the ordered list according to the following various schemes based on the concept that the higher occurrence frequency symbol will "bubble up" in the ordered list. Hence, we can code the indices of these symbols with a variable-length code so as to take their occurrence frequency into account and reduce the information redundancy in the following.

- Adaptive Huffman algorithm with transposition (AHAT): it is proposed in [19] and used in [11]. In this scheme, the swapping operation is carried out only between two adjacent items with indices i and $i+1$, where i is the index of the matched input symbol.
- Adaptive Huffman algorithm with fixed-block exchange (AHFB): In this scheme, the ordered list is partitioned into k fixed-size blocks $b_{k-1}, \ldots, b_1, b_0$ with that the size of each block is determined in advance and a pointer is associated with each block. Each pointer associated with its block is followed in the FIFO discipline. The swapping operation is carried out only between two adjacent blocks except the first block. More precisely, the matched item in block b_i is swapped with an item in block b_{i+1} pointed by the pointer associated with the block b_{i+1}, for all $k-2 \geq i \geq 0$. The matched item in block b_{k-1} is swapped with the item pointed by the pointer in the same block.
- Adaptive Huffman algorithm with dynamic-block exchange (AHDB): This scheme is similar to AHFB except that the ordered list is dynamically partitioned into several variable-size blocks, that is, each block can be shrunk or expanded in accordance with the occurrence frequency of the input characters. Of course, the ordered list has a predefined size in practical realization. Also a pointer is associated with each block. Initially, all pointers are pointed to the first item of the ordered list. Along with the progress of the algorithm, each pointer p_i maintains the following invariant: the symbols with indices between above p_i and pointer p_{i+1}, i.e., in the interval $(p_i, p_{i+1}]$, for all $k \geq i \geq 1$, just appeared i times in the input sequence, where p_0 and p_{k+1} are virtual pointers, indicate the lowest and highest bounds, respectively. Based on this, the swapping operation is carried out between two adjacent blocks except the first block, which is carried out in itself. An example illustrating the operations of the AHDB algorithm is depicted in Fig. 2.

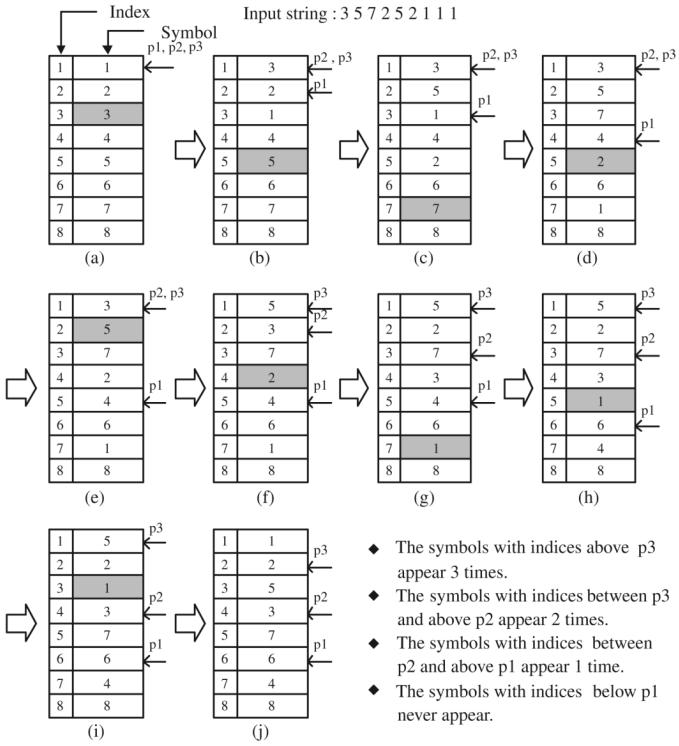


Fig. 2. Illustrating example of the AHDB algorithm.

- Adaptive Huffman algorithm with dynamic-block exchange without initialization (AHDBwi): In all the above schemes, the ordered list is initialized with its initial values from input alphabet set, such as the one shown in Fig. 2(a), in which the ordered list is initialized with symbols: $1, 2, \ldots, 8$ in sequence. This scheme is similar to AHDB except that the ordered list is not initialized.

In summary, by using a simple ordered list to memorize the occurrence frequency of symbols, both of the search and update times are significantly reduced from $O(\log_2 n)$, which is required in the general tree structures except the one using the parallel search, to $O(1)$, where n is the total number of input symbols. Please note that a parallel search tree is generally not easy to be realized in VLSI technology. Tables I and II compare the performance of various approximated AH algorithms with that of the AH algorithm.

TABLE I
PERFORMANCE COMPARISON BETWEEN THE AH ALGORITHM AND ITS VARIOUS APPROXIMATED VERSIONS IN THE CASE OF TEXT FILES

Testfile	AH	Approximated AH				Comment
		AHAT	AHFB	AHDB	AHDBwi	
1. alice29.txt (152,089)	42.33	38.01	33.24	39.31	39.13	Canterbury corpus
2. Asyoulik.txt (125,179)	39.38	34.63	29.90	36.19	36.82	Canterbury corpus
3. book1 (768,771)	43.00	39.10	34.10	39.32	39.26	Calgary corpus
4. book2 (610,856)	40.02	36.80	33.11	37.38	37.84	Calgary corpus
5. cp.htm (24,603)	33.70	21.66	23.66	29.55	31.29	Canterbury corpus
6. fields.c (11,150)	35.96	20.22	30.28	35.69	35.37	Canterbury corpus
7. lcet10.txt (426,754)	41.52	38.40	33.42	38.93	38.92	Canterbury corpus
8. paper1 (53,161)	37.29	31.28	30.71	35.56	36.15	Calgary corpus
9. paper2 (82,199)	42.03	35.80	33.49	38.37	38.54	Calgary corpus
10. paper3 (46,526)	41.14	32.94	32.72	37.44	37.86	Calgary corpus
11. paper4 (13,286)	40.07	23.18	32.64	37.24	37.45	Calgary corpus
12. paper5 (11,954)	36.86	19.48	30.96	35.40	35.51	Calgary corpus
13. paper6 (38,105)	36.97	30.39	32.34	36.56	36.84	Calgary corpus
14. plrabn12.txt (481,861)	42.79	38.88	33.31	39.18	39.18	Canterbury corpus
Average	39.50	31.48	31.71	36.86	37.15	

TABLE II
PERFORMANCE COMPARISON BETWEEN THE AH ALGORITHM AND ITS VARIOUS APPROXIMATED VERSIONS IN THE CASE OF EXECUTABLE FILES

Testfile	AH	Approximated AH				Comment
		AHAT	AHFB	AHDB	AHDBwi	
1. acrord32.exe (2,318,848)	21.77	17.99	16.53	20.12	21.44	acrobat reader
2. command.com (94,292)	32.44	25.32	25.32	30.77	28.32	DOS command.com
3. cutftp32.exe (813,568)	27.74	23.75	23.32	27.52	27.09	cuteftp
4. fdisk.exe (64,124)	11.10	2.80	9.37	11.45	13.94	DOS fdisk command
5. format.com (49,665)	32.69	25.08	26.94	31.77	29.04	DOS format command
6. netterm.exe (383,488)	28.47	25.56	25.01	29.24	29.01	netterm
7. tc.exe (292,249)	14.72	8.23	10.77	12.65	15.27	turbo C compiler
8. telnet.exe (81,920)	33.22	25.11	26.99	31.63	29.49	telnet
9. uedit32.exe (1,064,960)	28.58	23.61	21.29	25.77	25.51	
10. waterfal.exe (186,880)	28.58	22.78	23.71	28.12	27.20	
11. winamp.exe (892,928)	34.21	33.44	35.34	40.39	37.09	
12. winword.exe (8,441,907)	25.01	19.51	16.79	20.93	21.49	winword
13. winzip32.exe (983,040)	31.26	26.86	26.82	31.47	29.96	winzip32
14. xmplayer.exe (398,848)	26.70	21.99	21.53	25.39	25.14	XMplayer
Average	26.89	21.57	22.12	26.23	25.71	

Table I shows the simulation results in the case of the text files while Table II in the case of executable files. The overall performance and cost are compared in Table III. From the table, the AHDB algorithm is superior to both AHAT and AHFB algorithms and its performance is worse than the AH algorithm only by at most an amount of 2%, but at much less memory cost. The memory cost of the AH algorithm is estimated from the software implementation in [22], see Section V. Therefore, in the rest of the paper, we will use this scheme as the second stage of the proposed two-stage compression processor. The detailed AHDB algorithm is described as follows.

TABLE III
OVERALL PERFORMANCE COMPARISON BETWEEN THE AH ALGORITHM AND ITS VARIOUS APPROXIMATED VERSIONS

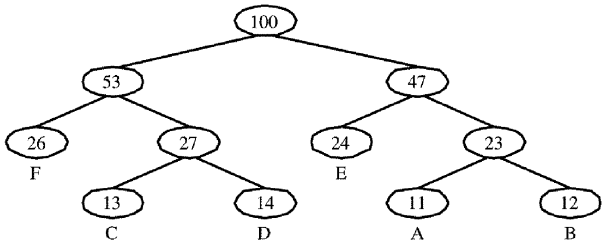
	AH	Approximated AH			
		AHAT	AHFB	AHDB	AHDBwi
Text file	39.50	31.48	31.71	36.86	37.15
Executable file	26.89	21.57	22.12	26.23	25.71
Average	33.20	26.53	26.92	31.55	31.43
Memory (bytes)	≈ 4.5 k	256	256	256	257

Algorithm: AHDB

Input: The compressed codewords from PDLZW algorithm.
Output: The compressed codewords.
Begin
1: Input $pdlzw\_output$;
2: while ($pdlzw\_output$! = null) **do**
  2.1: $matched\_index \longleftarrow$ search_ordered_list($pdlzw\_output$);
  2.2: $swapped\_block \longleftarrow$ determine_which_block_to_be_swapped ($matched\_index$);
  2.3: if ($swapped\_block! = k$) **then**
    2.3.1: swap(ordered_list[$matched\_index$], ordered_list[$pointer\_of\_swapped\_block$]);
    2.3.2: $pointer\_of\_swapped\_block = pointer\_of\_swapped\_block + 1$;
    2.3.3: reset_check($pointer\_of\_swapped\_block$); {Divide the $pointer\_of\_swapped\_block$ by two (or reset) when it reaches a threshold.}
  else
    2.3.4: if ($matched\_index! = 0$) **then** swap(list[$matched\_index$], list[$matched\_index - 1$]);
  2.4: Input $pdlzw\_output$;
End {End of AHDB Algorithm.}

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)



(b)

Fig. 3. Example of the Huffman tree and its three possible encodings. (a) Illustration example. (b) Huffman tree associated with (a).

TABLE IV
CANONICAL HUFFMAN CODE USED IN THE AHDB PROCESSOR

Codeword length	First_codeword	$Num\_codewords\_l[]$	Codeword_offset
4	15(1111)	1	0
5	22(10110)	8	1
6	35(100011)	9	9
7	57(0111001)	13	18
8	44(00101100)	70	31
9	35(000100011)	53	101
10	53(0000110101)	17	154
11	91(00001011011)	15	171
12	00(000000000000)	182	186

2) Canonical Huffman Code: The Huffman tree corresponding to the output from the PDLZW algorithm can be built by using the offline AH algorithm. An example of the Huffman tree for an input symbol set $\{A, B, C, D, E, F\}$ is shown in Fig. 3(a). Although the Huffman tree for a given symbol set is unique, such as Fig. 3(b), the code assigned to the symbol set is not unique. For example, three of all possible codes for the Huffman tree are shown in Fig. 3(a). In fact, there are 32 possible codes for the symbol set $\{A, B, C, D, E, F\}$ since we can arbitrarily assign 0 or 1 to each edge of the tree.

For the purpose of easy decoding, it is convenient to choose the encoding type three depicted in Fig. 3(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 carried out as follows [22]. First, the AH 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 array $num\_codewords\_l[]$. Finally, the starting value (or called $codeword\_offset$) for each codeword group of the same codeword length is calculated from array $num\_codewords\_l[]$. Based on this procedure, the codeword length, $first\_codeword$ (of each group with the same length), the number of codewords (in column $num\_codewords\_l[]$), and $codeword\_offset$ for the input symbols, consisting of the output codewords from the PDLZW algorithm with the dictionary set $\{256, 64, 32, 16\}$, are calculated and shown in Table IV. In general, different input data statistics to the PDLZW algorithm will produce different PDLZW output codes and canonical Huffman codes. Table IV is obtained from the output of the PDLZW algorithm with the best compression ratio when input data profiles are applied to the proposed two-stage compression architecture. For the rest of the paper, we will assume that the canonical Huffman code shown in Table IV is used for simplifying the VLSI realization.

III. TRADEOFF BETWEEN DICTIONARY SIZE AND PERFORMANCE

In this section, we will describe the partition approach of the dictionary set and show how to tradeoff the performance with the dictionary-set size, namely, the number of bytes of the dictionart set.

A. Dictionary Size Selection

In general, different address-space partitions of the dictionary set will present significantly distinct performance of the PDLZW compression algorithm. However, the optimal partition is strongly dependent on the actual input data files. Different data profiles have their own optimal address-space partitions. Therefore, in order to find a more general partition, several different kinds of data samples are run with various partitions of a given address space. As mentioned before, each partition corresponds to a dictionary set. To confine the dictionary-set size and simplify the address decoder of each dictionary, the partition rules are described as follows.

1) The first dictionary address space of all partitions is always 256, because we assume that each symbol of the source alphabet set is a byte, and it does not really need hardware.
2) Each dictionary in the dictionary set must have address space of 2^k words, where k is a positive integer.

For instance, one possible partition for the 368-address space is: $\{256, 64, 32, 8, 8\}$. As described in rule 1, the first dictionary (DIC-1) is not presented in the table. Note that for each given address space, there are many possible partitions and may have different dictionary-set sizes. Ten possible partitions of the 368-address dictionary set are shown in Table V, the sizes of the resulting dictionary sets are from 288 to 648 B.

B. Performance of PDLZW Only

The efficiency of a compression algorithm is often measured by the compression ratio, which is defined as the percentage of the amount of data reduction with respect to the original data. This definition of the compression ratio is often called the percentage of data reduction to avoid ambiguity. It is shown in [12] that the percentage of data reduction is improved as the address space of the dictionary set is increased. Thus, the algorithm with 4 k ($k = 1024$) address space has the best average compression

TABLE V
TEN POSSIBLE PARTITIONS OF THE 368-ADDRESS DICTIONARY SET

Partition	DIC-2	DIC-3	DIC-4	DIC-5	DIC-6	DIC-7	DIC-8	Memory (bytes)
368-1	64	32	16					288
368-2	64	32	8	8				296
368-3	64	16	16	16				320
368-4	32	32	32	16				368
368-5	32	32	16	16	16			400
368-6	32	32	16	16	8	8		408
368-7	32	16	16	16	16	16		464
368-8	32	16	16	16	16	8	8	504
368-9	16	16	16	16	16	16	16	560
368-a	8	8	16	16	16	16	32	648

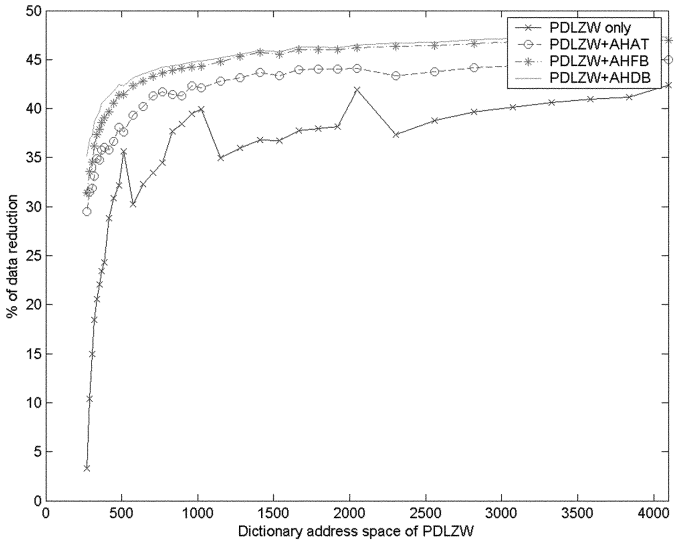


Fig. 4. Percentage of data reduction of various compression schemes.

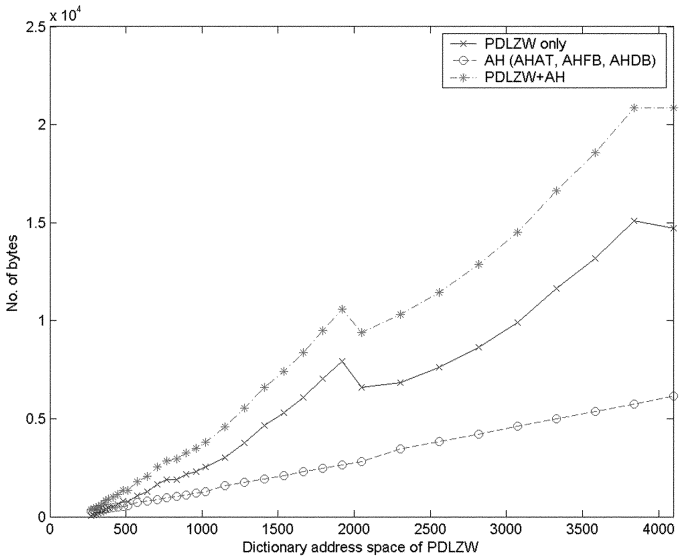


Fig. 5. Number of bytes required in various compression schemes.

ratio in all cases that we have simulated. The percentage of data reduction versus address space from 272 to 4096 of the dictionary used in the PDLZW algorithm is depicted in Fig. 4. From the figure, the percentage of data reduction increases asymptotically with the address space but some anomaly phenomenon arises, i.e., the percentage of data reduction decreases as the address space increases. For example, the percentage of data reduction is 35.63% at an address space of 512 but decreases to 30.28% at an address space of 576 because the latter needs more bits to code the address of the dictionary set. Some other examples also appear. As a consequence, the percentage of data reduction is not only determined by the correlation property of underlying data files being compressed but also depends on an appropriate partition as well as address space. Fig. 5 shows the dictionary size in bytes of the PDLZW algorithm at various address spaces from 272 to 4096.

An important consideration for hardware implementation is the required dictionary address space that dominates the chip cost for achieving an acceptable percentage of data reduction. From this view point and Fig. 4, one cost-effective choice is to use 1024 address space which only needs a 2,528-B memory and achieves 39.95% of data reduction on the average. The cost (in bytes) of memory versus address space from 272 to 4096 of the dictionary used in the PDLZW algorithm is depicted in Fig. 5.

C. *Performance of* PDLZW + AHDB

As described previously, the performance of the PDLZW algorithm can be enhanced by incorporating it with the AH algorithm, as verified from Fig. 4. The percentage of data reduction increases more than 5% in all address spaces from 272 to 4096. This implies that one can use a smaller dictionary size in the PDLZW algorithm if the memory size is limited and then use the AH algorithm as the second stage to compensate the loss of the percentage of data reduction.

From both Figs. 4 and 5, we can conclude that incorporating the AH algorithm as the second stage not only increases the performance of the PDLZW algorithm but also compensates the percentage of data reduction loss due to the anomaly phenomenon occurred in the PDLZW algorithm. In addition, the proposed scheme is actually a parameterized compression algorithm because its performance varies with different dictionary-set sizes but the architecture remains the same. Furthermore, our design has an attractive feature: although simple and, hence, fast but still very efficient, which makes this architecture very suitable for VLSI technology. The performance in percentage of data reduction of various partitions using the 368-address dictionary of the PDLZW algorithm followed by the AHDB algorithm is shown in Tables VI and VII. The percentage of data reduction and memory cost of various partitions using

TABLE VI
PERCENTAGE OF DATA REDUCTION OF VARIOUS PARTITIONS USING 368-ADDRESS DICTIONARY OF THE PDLZW ALGORITHM FOLLOWED BY THE AHDB ALGORITHM IN THE CASE OF TEXT FILES

	Partition type									
	368-1	*368-2	368-3	368-4	368-5	368-6	368-7	368-8	368-9	368-a
1. alice29.txt (152,089)	41.10	42.64	42.56	42.90	43.07	43.22	42.51	42.54	42.03	41.25
2. asyoulik.txt (125,179)	38.88	40.07	39.85	39.99	40.17	40.17	39.31	39.32	37.98	35.46
3. book1 (768,771)	39.16	40.98	40.80	40.98	40.74	40.64	40.02	40.04	39.26	38.05
4. book2 (610,856)	40.10	41.85	41.74	42.20	42.30	42.35	41.74	41.84	40.78	39.08
5. cp.htm (24,603)	36.95	39.42	40.48	39.68	40.68	40.56	41.59	42.38	41.14	39.38
6. fields.c (11,150)	45.79	46.27	46.44	48.05	48.38	48.49	48.93	48.93	47.53	44.28
7. lcet10.txt (426,754)	40.76	42.51	42.49	42.48	42.50	42.56	42.01	42.00	41.43	40.17
8. paper1 (53,161)	38.82	40.40	40.19	40.67	40.90	40.83	39.93	39.91	39.70	37.95
9. paper2 (82,199)	40.00	41.83	41.55	41.80	41.94	42.00	41.14	41.26	40.89	39.45
10. paper3 (46,526)	38.85	40.63	40.56	40.97	40.69	40.66	40.10	40.13	39.20	37.70
11. paper4 (13,286)	39.29	40.77	40.66	41.13	41.16	41.28	40.48	40.38	39.82	37.97
12. paper5 (11,954)	39.05	40.25	40.00	40.40	40.75	40.64	39.89	39.87	38.80	36.72
13. paper6 (38,105)	40.16	41.56	41.33	41.98	41.77	42.05	41.48	41.71	40.58	38.30
14. plrabn12.txt (481,861)	40.34	41.85	41.75	41.96	41.79	41.70	41.05	40.99	39.12	37.79
average	39.95	41.50	41.46	41.80	41.92	41.94	41.44	41.52	40.59	38.83

(∗ denotes the partition used in the paper.)

TABLE VII
PERCENTAGE OF DATA REDUCTION OF VARIOUS PARTITIONS USING 368-ADDRESS DICTIONARY OF THE PDLZW ALGORITHM FOLLOWED BY AN AHDB ALGORITHM IN THE CASE OF EXECUTABLE FILES

	Partition type									
	368-1	*368-2	368-3	368-4	368-5	368-6	368-7	368-8	368-9	368-a
1. acrord32.exe (2,318,848)	32.38	31.41	31.49	32.53	32.79	32.87	32.74	32.78	32.02	29.53
2. command.com (94,292)	43.22	43.30	43.38	43.59	44.17	44.59	44.56	44.85	44.33	43.30
3. cutftp32.exe (813,568)	39.13	38.41	38.32	39.22	39.64	39.81	39.68	39.79	39.54	37.88
4. fdisk.exe (64,124)	27.26	26.05	26.36	26.52	26.81	26.83	26.60	26.63	24.89	22.55
5. format.com (49,655)	45.34	45.32	45.26	45.43	46.15	46.55	46.48	46.68	46.50	44.92
6. netterm.exe (383,488)	41.50	40.81	40.99	42.32	42.97	43.05	43.06	43.16	42.69	40.47
7. tc.exe (292,249)	28.21	26.90	27.16	27.51	27.92	27.90	27.95	28.04	26.76	24.99
8. telnet.exe (81,920)	43.73	43.16	43.03	43.89	44.48	44.71	44.38	44.58	44.38	42.60
8. uedit32.exe (1,064,960)	36.16	35.26	35.12	36.04	36.33	36.44	36.24	36.27	36.02	34.52
10. waterfal.exe (186,880)	39.45	38.77	38.68	39.27	39.69	39.87	39.62	39.74	39.15	37.59
11. winamp.exe (892,928)	52.24	52.39	52.20	52.79	53.36	53.68	53.51	53.74	53.32	52.19
12. winword.exe (8,441,907)	32.44	31.50	31.40	32.02	32.30	32.38	32.25	32.33	31.84	30.47
13. winzip32.exe (983,040)	41.19	40.52	40.57	41.24	41.64	41.87	41.74	41.80	41.62	40.32
14. xmplayer.exe (398,848)	36.42	35.65	35.50	36.44	36.79	36.91	36.75	36.87	36.43	34.87
average	38.48	37.82	37.82	38.49	38.93	39.10	38.97	39.09	38.54	36.87

(∗ denotes the partition used in the paper.)

TABLE VIII
PERCENTAGE OF DATA REDUCTION AND MEMORY COST OF VARIOUS PARTITIONS USING 368-ADDRESS DICTIONARY PDLZW ALGORITHM FOLLOWED BY THE AHDB ALGORITHM

	Partition type									
	368-1	*368-2	368-3	368-4	368-5	368-6	368-7	368-8	368-9	368-a
Memory cost in bytes										
PDLZW cost	288	296	320	368	400	408	464	504	560	648
AHDB cost	414	414	414	414	414	414	414	414	414	414
total cost	702	*710	734	782	814	822	878	918	974	1062
Comparison of the average of performance in percentage of data reduction										
Text files	39.95	41.50	41.46	41.80	41.92	41.94	41.44	41.52	40.59	38.83
Executable files	38.48	37.82	37.82	38.49	38.93	39.10	38.97	39.09	38.54	36.87
Total Average	39.22	*39.66	39.64	40.15	40.43	40.52	40.21	40.31	39.57	37.85

(∗ denotes the partition used in the paper.)

a 368-address dictionary PDLZW algorithm followed by the AHDB algorithm is depicted in Table VIII.

To illustrate our design, in what follows, we will use the PDLZW compression algorithm with the 368-address dictionary set as the first stage and the AHDB as the second stage to constitute the two-stage compression processor. The decompression processor is conceptually the reverse of the compression counterpart and uses the same data path. As a consequence, we will not address its operation in detail in the rest of the paper.

IV. PROPOSED DATA COMPRESSION ARCHITECTURE

In this section, we will show an example to illustrate the hardware architecture of the proposed two-stage compression scheme. The proposed two-stage architecture consists

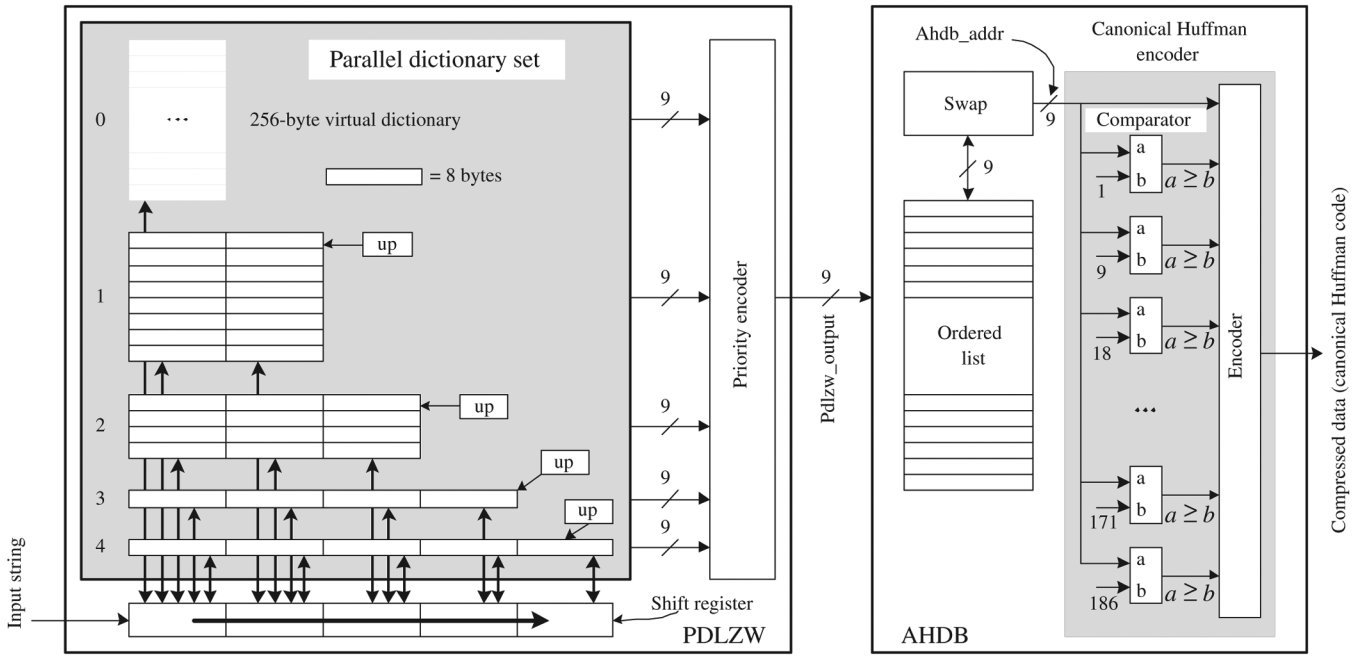


Fig. 6. Architecture of proposed two-stage compression processor.

of two major components: a PDLZW processor and an AHDB processor, as shown in Fig. 6. The former is composed of a dictionary set with partition $\{256, 64, 32, 8, 8\}$. Thus, the total memory required in the processor is 296 B $(= 64 \times 2 + 32 \times 3 + 8 \times 4 + 8 \times 5)$ only. The latter is centered around an ordered list and requires a content addressable memory (CAM) of 414 B $(= 368 \times 9$ B). Therefore, the total memory used is a 710-B CAM.

A. PDLZW Processor

The major components of the PDLZW processor are CAMs, a 5-B shift register, and a priority encoder. The word widths of CAMs increase gradually from 2 to 5 B with four different address spaces: 64, 32, 8, and 8 words, as portrayed in Fig. 6.

The input string is shifted into the 5-B 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 $pdlzw\_output$, which is the compressed codeword output from the PDLZW processor. This codeword is then encoded into canonical Huffman code by the AHDB processor. In general, it is not impossible that many (up to five) 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 picked out and the matched address within its dictionary along with the matched signal of the dictionary is encoded and output to the AHDB 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. 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.

B. AHDB Processor

The AHDB processor encodes the output codewords from the 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 the PDLZW processor and, thus, to remove the information redundancy contained in the codewords. The encoding process is carried out as follows. The $pdlzw\_output$, which is the output from the PDLZW processor and is the "symbol" for the AHDB algorithm, is input into $swap$ unit for searching and deciding the matched index, n, from the ordered list. Then the $swap$ unit exchanges the item located in n with the item pointed by the pointer of the swapped block. That is, the more frequently used symbol bubbles up to the top of the ordered list. The index $ahdb\_addr$ of the "symbol" $pdlzw\_output$ 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 $ahdb\_addr$ is compared with all $codeword\_offset$: 1, 9, 18, 31, 101, 154, 171, and 186 simultaneously, as shown in Table IV and Fig. 6, for deciding the length of the codeword to be encoded. Once the length is determined, the output codeword can be encoded as $ahdb\_addr - code\_offset + first\_codeword$. For example, if $ahdb\_addr = 38$, from Table IV, the length is 8 b since 38 is greater than 31 and smaller than 101. The output codeword is: $38 - 31 + 44 = 51 = 00110011_2$.

TABLE IX
PERFORMANCE COMPARISON IN PERCENTAGE OF DATA REDUCTION BETWEEN COMPRESS, PDLZW + AH, PDLZW + AHAT, PDLZW + AHFB, AND PDLZW + AHDB

File index	Text files					Executable files				
			PDLZW+					PDLZW+		
	Compress	AH	AHAT	AHFB	AHDB	Compress	AH	AHAT	AHFB	AHDB
1	59.52	42.33	35.70	39.59	42.64	34.10	21.77	29.93	30.15	31.41
2	56.07	39.38	32.53	37.12	40.07	41.27	32.44	38.44	42.53	43.40
3	56.81	43.00	39.93	37.78	40.98	40.72	27.74	34.74	37.47	38.41
4	58.95	40.02	40.18	38.88	41.85	23.75	11.10	19.23	25.01	26.05
5	54.00	33.70	30.67	36.14	39.42	43.17	32.69	39.27	44.86	45.32
6	55.48	35.96	31.94	43.71	46.27	45.26	28.47	34.94	39.98	40.81
7	61.03	41.52	39.85	39.58	42.51	25.45	14.72	23.09	25.79	26.90
8	52.83	37.29	30.65	37.47	40.40	40.72	33.22	36.84	42.63	43.16
9	56.01	42.03	32.26	38.64	41.83	37.62	28.58	33.54	34.24	35.26
10	52.36	41.14	30.04	37.45	40.63	35.28	28.58	32.85	37.96	38.77
11	47.64	40.07	26.73	37.60	40.77	52.25	34.21	48.14	51.80	52.39
12	44.96	36.86	25.93	37.13	40.25	31.76	25.01	31.14	30.24	31.49
13	50.94	36.97	30.75	38.73	41.56	41.35	21.26	36.47	39.95	40.52
14	58.32	42.79	39.77	38.72	41.85	36.28	26.70	31.00	34.87	35.65
Avg.	54.64	39.50	33.35	38.47	41.50	37.78	26.89	33.54	36.96	37.82

TABLE X
COMPARISON SUMMARY

Chip	Hi/fn9602 [?], [?]	X-MatchPROv4 [?]	[?]	[?]	[?]	This paper
Process	0.35 μm	0.25 μm	0.8 μm	2 μm	1.2 μm	0.35 μm
Complexity	100 kgates	9039 Tile's 70% of a A500K130-BG456	3 kgates	NA	NA	130 kgates (cell-based)
Clock speed	80 MHz	25 MHz	100 MHz	40 MHz	100 MHz	100 MHz
Throughput	80 MB/s	100 MB/s	10 Mbps	13.3 MB/s	12.5 MB/s	16.7 to 125 MB/s (compression) 25 to 83 MB/s (decompression)
Algorithm	LZS	X-MatchPRO	LZ	LZ	LZ	PDLZW+AHDB
I/O Width	16 and 32	32	NA	NA	NA	8

As described above, the compression rate is between 1–5 B per memory cycle.

C. Performance

Table IX shows the compression ratio of the LZW (compress), the AH algorithm, PDLZW + AHAT, PDLZW + AHFB, and PDLZW + AHDB. The dictionary set used in PDLZW is only 368 addresses (words) and partitioned as $\{256, 64, 32, 8, 8\}$. From the table, the compression ratio of PDLZW + AHDB is competitive to that of the LZW (i.e., compress) algorithm in the case of executable files but is superior to that of the AH algorithm in both cases of text and executable files.

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 basis for comparing the hardware cost of different architectures. According to the usual implementation of the AH algorithm, the memory requirement of an N-symbol alphabet set is $(N+1)+4(2N-1)$ integer variables [22], which is equivalent to $2\times\{(N+1)+4(2N-1)\}\approx 4.5$ kB, where $N=256$. The memory required in the AHDB algorithm is only a 256-B CAM, which corresponds to the 384-B static random-access memory (SRAM). Here, we assume the complexity of one CAM cell is 1.5 times that of a SRAM cell [21]. However, as seen from Tables I and II, the average performance of the AHDB algorithm is only $1.65\% = ((39.50-36.86)+(26.89-26.23)/2)\%$ worse than that of the AH algorithm.

After cascading with the PDLZW algorithm, the total memory cost is increased to 710-B CAM equivalently, which corresponds to 1065 B of RAM and is only one-fourth of that of the AH algorithm. However, the performance is improved by $8.11\% = (39.66-31.55)\%$, where numbers 39.66% and 31.55% are from Tables VIII and III, respectively.

D. Experimental Results

The proposed two-stage compression/decompression processor has been synthesized and simulated with a 0.35-μm 2P4M cell library. The resulting chip has a die area of 4.3×4.3 mm^2 and a core area of 3.3×3.3 mm^2. The simulated power dissipation is between 632 and 700 mW at the operating frequency of 100 MHz. The compression rate is between 16.7 and 125 MB/s; the decompression rate is between 25 and 83 MB/s.

Fig. 7 shows the chip layout of the proposed two-stage data compression/decompression processor. Since we use D-type flip-flops associated with needed gates as the basic memory cells of CAMs (the dictionary set in the PDLZW processor) and of ordered list (in the AHDB processor), these two parts occupy most of the chip area. The remainder only consumes about 20% of the chip area. To reduce the chip area and increase performance, the full-custom approach can be used. A flip-flop may take between 10 to 20 times the area of a six-transistor static RAM cell [18]; a basic CAM cell may take up to 1.5 times

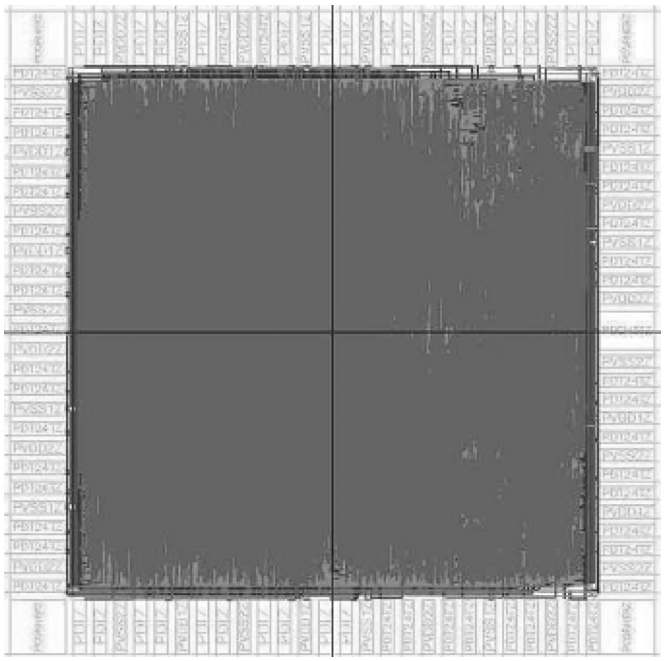


Fig. 7. Chip layout of the proposed two-stage data compression/decompression processor.

the area (nine transistors) of a static RAM cell [21]. Thus, the area of the chip will be reduced dramatically when full-custom technology is used. However, our HDL-based approach can be easily adapted to any technology, such as FPGA, CPLD, or cell library. The comparison with other recently published results are summarized in Table X.

V. Conclusion

In this paper, a new two-stage architecture for lossless data compression applications, which uses only a small-size dictionary, is proposed. This VLSI data compression architecture combines the PDLZW compression algorithm and the AH algorithm with dynamic-block exchange. The PDLZW processor is based on a hierarchical parallel dictionary set that has successively increasing word widths from 1 to 5 B with the capability of parallel search. The total memory used is only a 296-B CAM. The second processor is built around an ordered list constructed with a CAM of 414 B ($= 368 \times 9$ b) 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 and an ordered list such that the control logic is essentially trivial. In addition, in the case of executable files, the performance of the proposed architecture is competitive with that of the LZW algorithm (compress). The data rate for the compression processor is at least 1 and up to 5 B 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 B for the PDLZW processor and 414 B for the AHDB processor. Therefore, a very high data rate can be achieved.

Acknowledgment

The authors would like to thank the anonymous reviewers and the communicating editor for their useful comments which improved the presentation of this paper. In addition, they would like to thank Mr. S.-Y. Lin for his assistance in implementing the design layout and doing the post-layout simulation of the chip.

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. New York: McGraw-Hill, 2001.
[3] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Reading, MA: Addison-Welsley, 1992.
[4] S. Henriques and N. Ranganathan, "A parallel architecture for data compression," in *Proc. 2nd IEEE Symp. Parall. Distrib. Process.*, 1990, pp. 260–266.
[5] *9600 Data Compression Processor*. Los Gatos, CA: Hi/fn Inc., 1999.
[6] S.-A. Hwang and C.-W. Wu, "Unified VLSI systolic array design for LZ data compression," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 4, pp. 489–499, Aug. 2001.
[7] J. Jiang and S. Jones, "Word-based dynamic algorithms for data compression," *Proc. Inst. Elect. Eng.-I*, vol. 139, no. 6, pp. 582–586, Dec. 1992.
[8] B. Jung and W. P. Burleson, "Efficient VLSI for Lempel-Ziv compression in wireless data communication networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 6, no. 3, pp. 475–483, Sep. 1998.
[9] D. E. Knuth, "Dynamic Huffman coding," *J. Algorithms*, vol. 6, pp. 163–180, 1985.
[10] M.-B. Lin, "A parallel VLSI architecture for the LZW data compression algorithm," in *Proc. Int. Symp. VLSI Technol., Syst., Appl.*, 1997, pp. 98–101.
[11] M.-B. Lin and J.-W. Chen, "A multilevel hardware architecture for lossless data compression applications," in *Proc. 1999 Nat. Comput. Symp.*, 1999, pp. 289–292.
[12] M.-B. Lin, "A parallel VLSI architecture for the LZW data compression algorithm," *J. VLSI Signal Process.*, vol. 26, no. 3, pp. 369–381, Nov. 2000.
[13] 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.
[14] H. Park and V. K. Prasanna, "Area efficient VLSI architectures for Huffman coding," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 40, no. 9, pp. 568–575, Sep. 1993.
[15] N. Ranganathan and S. Henriques, "High-speed vlsi designs for lempel-ziv-based data compression," *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.*, vol. 40, no. 2, pp. 96–106, Feb. 1993.
[16] S. Khalid, *Introduction to Data Compression*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 2000.
[17] A. Sieminski, "Fast decoding of the Huffman codes," *Inform. Process. Lett.*, vol. 26, no. 5, pp. 237–241, 1988.
[18] M. J. S. Smith, *Application-Specific Integrated Circuits*. Reading, MA: Addison-Wesley, 1997.
[19] B. W. Y. Wei, J. L. Chang, and V. D. Leongk, "Single-chip lossless data compressor," in *Proc. Int. Symp. VLSI Technol., Syst., Appl.*, 1995, pp. 211–213.
[20] T. A. Welch, "A technique for high-performance data compression," *IEEE Comput.*, vol. 17, no. 6, pp. 8–19, Jun. 1984.
[21] N. H. E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 3rd ed. Reading, MA: Addison-Wesley, 2005, pp. 747–748.
[22] I. H. Witten, Alistair, and T. C. Bell, *Managing Compressing and Indexing Documents and Images*, 2nd ed. New York: Academic, 1999, pp. 36–51.
[23] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, Mar. 1977.
[24] ——, "A compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 5, pp. 530–536, Sep. 1978.

[25] R. J. Zito-Wolf, "A broadcast/reduce architecture for high-speed data compression," in *Proc. 2nd IEEE Symp. Parall. Distrib. Process.*, 1990, pp. 174–181.

Ming-Bo Lin (S'90–M'93) received the B.S. degree in electronic engineering from the National Taiwan Institute of Technology, Taipei, Taiwan, R.O.C., the M.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C., and the Ph.D. degree in electrical and computer engineering from the University of Maryland, College Park.

Since February 2001, he has been a Professor with the Department of Electronic Engineering at the National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include VLSI system design, mixed-signal integrated circuit designs, parallel architectures and algorithms, and embedded computer systems.

Jang-Feng Lee received the B.S. and M.S. degrees in electronic engineering from the National Taiwan University of Science and Technology, Taipei, Taiwan, R.O.C.

His research interests include VLSI system design.

Gene Eu Jan received the B.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C., in 1982, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Maryland, College Park, in 1988 and 1992, respectively.

He has been a Professor with the Department of Computer Science at the National Taipei University, San Shia, Taiwan, R.O.C., since 2004. Prior to joining the National Taipei University, he was a Visiting Assistant Professor in the Department of Electrical and Computer Engineering at the California State University, Fresno, in 1991, and an Associate Professor with the Departments of Computer Science and Navigation at the National Taiwan Ocean University, Keelung, Taiwan, from 1993 to 2004. His research interests include parallel computer systems, interconnection networks, motion planning, and VLSI systems design.