A Fast and Efficient Lossless Data-Compression Method

Jer Min Jou and Pei-Yin Chen, *Member, IEEE*

***Abstract*— This paper describes an online lossless data-compression method using adaptive arithmetic coding. To achieve good compression efficiency, we employ an adaptive fuzzy-tuning modeler that applies fuzzy inference to deal efficiently with the problem of conditional probability estimation. In comparison with other lossless coding schemes, the compression results of the proposed method are good and satisfactory for various types of source data. Since we adopt the table-lookup approach for the fuzzy-tuning modeler, the design is simple, fast, and suitable for VLSI implementation.**

***Index Terms*— Adaptive arithmetic coding, fuzzy inference, lossless data compression.**

I. Introduction

SINCE arithmetic coding [1]–[12] can approach the entropy limit as long as the statistics are accurate, it is superior to Huffman coding and has been used widely for data compression. However, the speed of arithmetic coding tends to be slow, because in standard form it requires at least one multiplication operation per input symbol. Moreover, it needs an extra division operation at every coding step when used adaptively. Many approximate methods, which may replace the multiplication or division operations by less expensive operations such as shifts and additions, have been proposed to reduce the computational complexity [4]–[8]. Some VLSI architectures of arithmetic coding also have been presented [9]–[12] to improve the coding speed. Because the implementation of multialphabet arithmetic coding is very complicated, few VLSI designs that use multialphabet arithmetic coding algorithm have been presented. To make the implementation of arithmetic coding easier and more practicable, the size of alphabet must be reduced to binary, so the coding process can be simplified correspondingly. The Q-coder, an adaptive binary arithmetic coding chip for the bilevel image compression, has been presented [4], [9]. Furthermore, the QM-coder, a linear descendent of the Q-coder, has been adopted by both JPEG and JBIG still-image compression algorithms [13]. Fu and Parhi [12] also proposed an algorithm that uses redundant arithmetic to obtain further speedup for the QM-coder. However, all these Q-coder-based arithmetic coding hardwares described above are designed to process mainly bilevel image data and may be poor for other types of data. It would be desirable to have a compressor universal enough to quickly compress any type of data and still achieve a good compression ratio.

The characteristics of various source data bear a lot of uncertainty and are hard to be extracted, so it is not easy to construct a good probability estimator that can provide accurate probability estimation for different types of data. To solve the problem, we employ fuzzy inference [14], [15] and propose a novel division-free arithmetic coding algorithm that can be readily implemented in hardware. In the design, we adopt a binary *order-o fixed-context* model that uses the o previous symbols as the state (or context). To reduce the complexity of hardware implementation and to increase the coding throughput, we use the table-lookup approach to construct *an adaptive fuzzy-tuning modeler* (AFTM). With the help of fuzzy inference process that dynamically selects the probability-tuning step, the *modeler* can determine the estimated probabilities more efficiently and precisely. Therefore, the compression efficiency of the proposed method is improved. Experimental results demonstrate that the proposed method performs better than other lossless compression methods, such as Huffman, approximate arithmetic [5]–[7], and Lempel–Ziv for different types of source data: text files, image files, and binary files. Besides, some online processing problems of arithmetic coding, such as source termination and carryover, are solved efficiently in the design.

II. Proposed Adaptive Arithmetic Coding Method

The process of general arithmetic coding can be split into two tasks: coding and modeling. A coder actually produces the compressed bit-stream, and a modeler feeds probability estimation information to it. The encoding process starts by initializing a semi-open interval [0, 1), which is recursively divided to subintervals in proportion to the conditional probabilities of the symbol being encoded. Let A_n denote the width of the selected subinterval at stage n and C_n represent the position of the lower boundary of the selected subinterval. The encoding process requires the following iterative computations:

$$A_{n+1} = A_n \times \hat{p}_n(x|s) \tag{1}$$

$$C_{n+1} = C_n + A_n \times \sum_{i=1}^{x-1} \hat{p}_n(i|s) \tag{2}$$

where $\hat{p}_n(x|s)$ is the estimated conditional probability of input symbol x at stage n, given the previous string

If a general arithmetic coding is applied to a binary alphabet (0 or 1), it permits a simple and fast coding process and is

Paper approved by C. S. Ravishankar, the Editor for Speech Processing of the IEEE Communications Society. Manuscript received November 5, 1997; revised June 15, 1998 and December 23, 1998. This research was supported in part by the National Science Council, R.O.C., under Grant NSC-86-2221-E-006-022.

J. M. Jou is with the Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan 701, R. O. C. (e-mail: jou@cad.ee.ncku.edu.tw).

P.-Y. Chen was with the Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan 701, R.O.C. He is now with the Department of Electronics Engineering, Southern Taiwan University of Technology, Tainan, Taiwan 710, R. O. C.

Publisher Item Identifier S 0090-6778(99)07455-3.

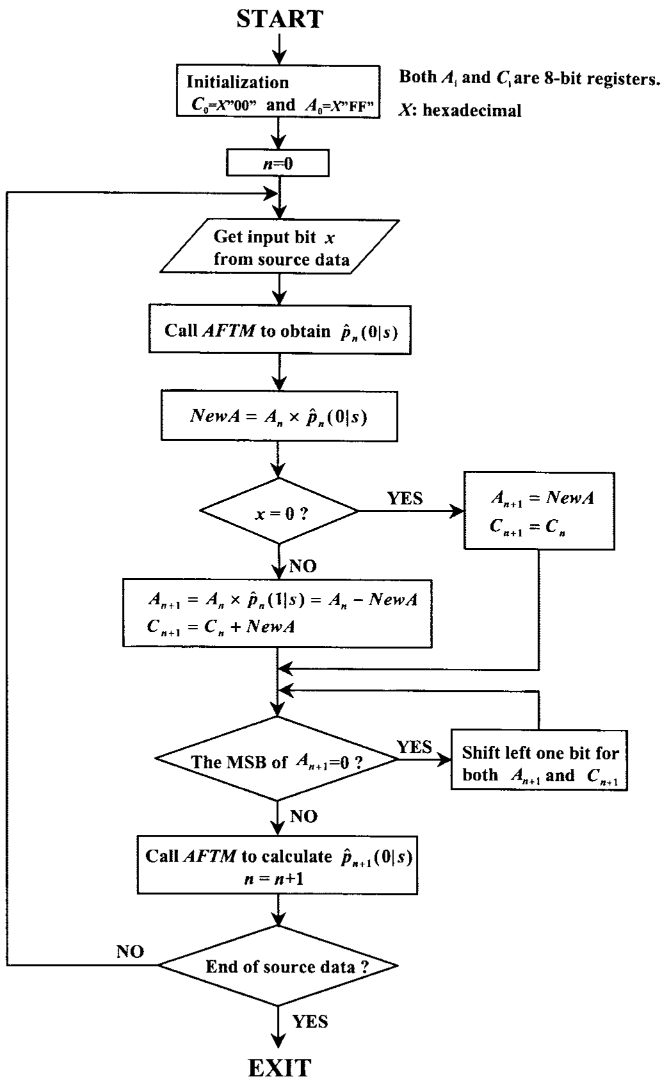

START
Initialization
C0=X"00" and A0=X"FF"
Both Ai and Ci are 8-bit registers.
X: hexadecimal
n=0
Get input bit x from source data
Call AFTM to obtain p̂n(0|s)
NewA = An × p̂n(0|s)
x = 0 ?
YES
An+1 = NewA
Cn+1 = Cn
NO
An+1 = An × p̂n(1|s) = An − NewA
Cn+1 = Cn + NewA
The MSB of An+1=0 ?
YES
Shift left one bit for both An+1 and Cn+1
NO
Call AFTM to calculate p̂n+1(0|s)
n = n+1
NO
End of source data ?
YES
EXIT

Fig. 1. Flowchart for encoding procedure of the proposed method.

more suitable for hardware implementation. Thus, we adopt the binary arithmetic coding in our method. The coding system consists of two main components: the *AFTM* and the *coder*. While encoding the nth input binary symbol x (symbol "0" or symbol "1"), the AFTM determines the $\hat{p}_n(x|s)$ and feeds it to the *coder*. The *coder* uses the $\hat{p}_n(x|s)$ and x to produce the compressed bit-stream. Finally, the AFTM updates the conditional probability [or determines the new conditional probability $\hat{p}_n(x|s)$] for the next coding cycle. Here, we adopt the *order-o fixed-context* model (or o-memory Markov model), which means s is composed of o previously coded bits before x. Fig. 1 shows the flowchart for the encoding procedure of our method. Obviously, $\hat{p}_n(0|s)$ is the only probability concerned during the whole coding process, since $\hat{p}_n(1|s)$ can be calculated by $1-\hat{p}_n(0|s)$. In the following discussion, the AFTM and the coder are stated respectively.

A. The AFTM

The first task of AFTM is to determine $\hat{p}_n(0|s)$ for each input bit. Here, the $\hat{p}_n(0|s)$ is calculated based on the relative occurring frequency of symbol "0" at stage n under current state s. It can be described as

$$\hat{p}_n(0|s) = \frac{\text{count}\,0_n(s)}{\text{count}\,N_n(s)} \tag{3}$$

where $count0_n(s)$ is the number of 0's that have occurred under state s until stage n, and $countN_n(s)$ stands for the total number of input bits that have occurred under state s until stage n. To deal with the necessary expensive division operation and to make hardware implementation easier, we apply two tables to approximate the $\hat{p}_n(0|s)$. We simulated the coding process, found the 128 possible probability values with higher occurring frequency for $\hat{p}_n(0|s)$, and then saved them in a probability table called Prob, which is applied to approximate the possible values of $\hat{p}_n(0|s)$. Since each of the 2^o possible states has its own $\hat{p}_n(0|s)$, an Addr table is constructed to store the 2^o pointers, each of them points to one of the entries of Prob so as to get the corresponding state's $\hat{p}_n(0|s)$.

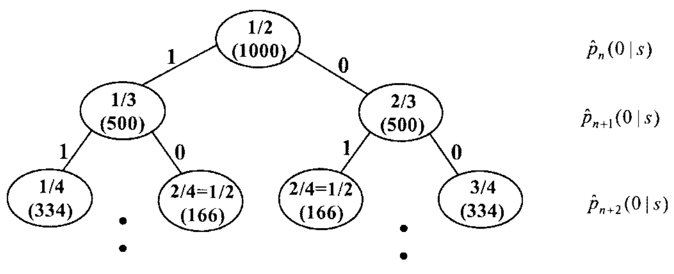


Fig. 2. Part of the probability estimation tree.

TABLE I
RESULT OF THE PARTIAL TREE OF FIG. 2

$\hat{p}(0\|s)$	1/4	1/3	1/2	2/3	3/4
Occurring times	334	500	1332	500	334
Percentage	11.1%	16.7%	44.4%	16.7%	11.1%

The probability table Prob contains the first 128 probabilities with the highest occurring frequencies for symbol "0." To find these probability values, we use a complete binary tree and some counters to simulate the process of calculating conditional probabilities in coding (see Fig. 2). Every node in the tree represents a state of the probability calculating process and contains two kinds of data: the value of $\hat{p}_n(0|s)$ at state n and its weight. The weight of a node shown in parentheses stands for the relative occurring frequency of the state compared to all other states. The left child of a node is the new state when an input symbol "1" is input, and the right child is "0." Initially, the value of the root node is "1/2" and its weight is set as 1000. The value of the right child of the root is "2/3," which means that the $\hat{p}_{n+1}(0|s)$ is "2/3" after an input symbol "0" is inputted, and the weight of the node is 500 because its parent node's weight is 1000 and the probability of getting the input symbol "0" is assumed to be 1/2. In this way, we build a complete binary tree of 255 levels. The partial tree and its corresponding results are shown in Fig. 2 and in Table I, respectively. After the complete binary tree of 255 levels is constructed, the first 128 probability values according to their corresponding weights (occurring frequencies) are selected and put into the Prob table.

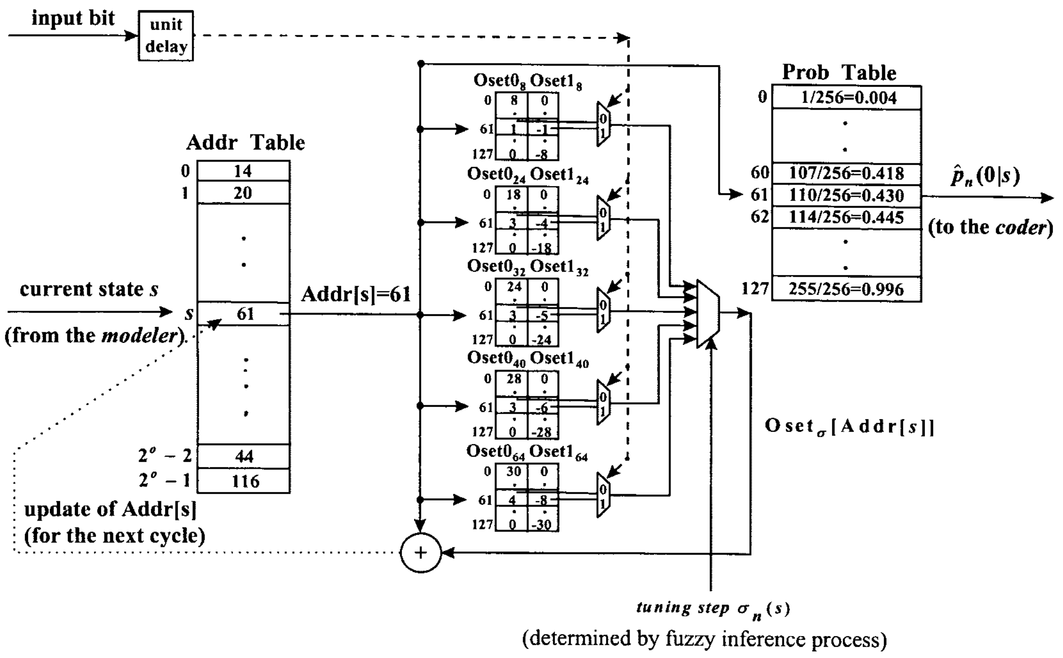


Fig. 3. Block diagram of the AFTM.

After the nth bit is coded, the AFTM will update $\hat{p}_n(0|s)$ [or determine $\hat{p}_{n+1}(0|s)$] for the following coding cycles. From (3), we know $\hat{p}_{n+1}(0|s)$ can be achieved by

$$\hat{p}_{n+1}(0|s) = \frac{\mathrm{count}\,0_{n+1}(s)}{\mathrm{count}\,N_{n+1}(s)} = \begin{cases} \dfrac{\mathrm{count}\,0_n(s)+1}{\mathrm{count}\,N_n(s)+1}, & \text{if the } n\text{th input bit is a 0} \\ \dfrac{\mathrm{count}\,0_n(s)}{\mathrm{count}\,N_n(s)+1}, & \text{if the } n\text{th input bit is a 1.} \end{cases} \quad (4)$$

Apparently, the conditional probabilities will change faster/slower when few/many input data have been compressed. After many experiments, we found that some source data files require faster probability changes, and some require slower probability changes to achieve better compression efficiency. Hence, based on (4), a new way is applied to approximate the new conditional probability as follows:

$$\hat{p}_{n+1}(0|s) = \begin{cases} \dfrac{\mathrm{count}\,0_n(s)/\sigma_n(s)+1}{\mathrm{count}\,N_n(s)/\sigma_n(s)+1}, & \text{if the } n\text{th input bit is a 0} \\ \dfrac{\mathrm{count}\,0_n(s)/\sigma_n(s)}{\mathrm{count}\,N_n(s)/\sigma_n(s)+1}, & \text{if the } n\text{th input bit is a 1} \end{cases} \quad (5)$$

where $\sigma_n(s)$ is the *probability-tuning step* used to reflect the degree of variation of conditional probabilities. To avoid the division operation, we employ two offset tables to approximate the $\hat{p}_{n+1}(0|s)$. If the current input bit is a 0 and the tuning step σ is given, we can prefind by using (5) the value of $\hat{p}_{n+1}(0|s)$ for each of the 128 possible values of $\hat{p}_n(0|s)$ stored in the Prob table, then calculate the 128 offsets (or distances) between the corresponding indexes in the Prob table for the values of $\hat{p}_{n+1}(0|s)$ and $\hat{p}_n(0|s)$, and finally store the 128 offsets in an offset table named $\mathrm{Oset0}_\sigma$. Similarly, the $\mathrm{Oset1}_\sigma$ table stores the 128 offsets for the current input "1." Fig. 3 shows the block diagram of AFTM. By changing the pointer's value stored in the Addr table with the distance provided by either the table $\mathrm{Oset0}_\sigma$ or $\mathrm{Oset1}_\sigma$, we can obtain the corresponding $\hat{p}_{n+1}(0|s)$ as follows:

$$\hat{p}_{n+1}(0|s) = \hat{p}_n(0|s) + \Delta\hat{p}_n(0|s) = \begin{cases} \mathrm{Prob}[\mathrm{Addr}[s] + \mathrm{Oset0}_\sigma[\mathrm{Addr}[s]]], & \text{if the } n\text{th input bit is a 0} \\ \mathrm{Prob}[\mathrm{Addr}[s] + \mathrm{Oset1}_\sigma[\mathrm{Addr}[s]]], & \text{if the } n\text{th input bit is a 1.} \end{cases} \quad (6)$$

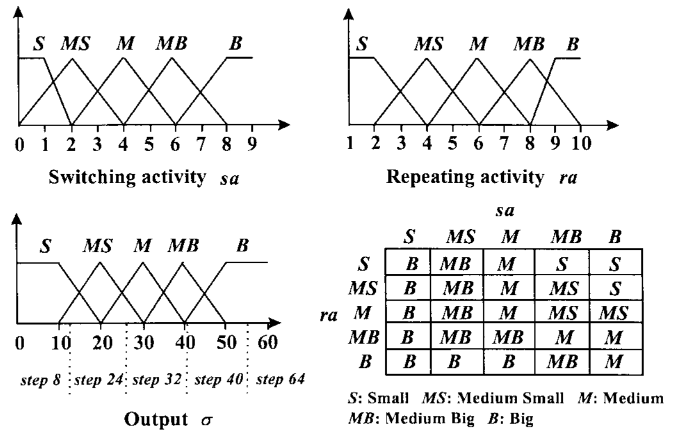
In fact, there are a large number of tuning steps that can be selected. It is certainly impossible and impractical to use too many steps, because that requires too many tables (memories). After many experiments, therefore, we chose only five different steps: 8, 24, 32, 40, and 64 in our design. As shown in Fig. 3, the five offset tables are indexed according to the value of Addr[s] simultaneously. One of the five offset values, selected with a proper tuning step σ generated by the fuzzy inference process, is used to determine the $\hat{p}_{n+1}(0|s)$.

The fuzzy inference process, performed in our design, is based on the concepts of fuzzy implication and the compositional rules of inference for approximate reasoning [14], [15]. The main function of fuzzy inference is to select a proper probability-tuning step to tune the $\hat{p}_n(0|s)$ effectively. For each possible state, we use a ten-bit queue to record the ten previously coded bits under the state. The corresponding queues of the 2^o states are all stored in the table Queue. Two evaluation parameters are used to observe the queue: the switching activity and the repeating activity. The switching activity (sa) reflects the number of binary symbol transitions (0 changes to 1 or 1 changes to 0) in the state queue, and the repeating activity (ra) reflects the number of identical bits counted from the last bit of the queue. Obviously, small

TABLE II
MEMORY SIZES OF PROPOSED CODING METHOD FOR DIFFERENT ORDERS

Unit: Bytes

Order	S_{Addr}	S_{Prob}	S_{Queue}	S_{Fuzzy}	S_{Oset0}	S_{Oset1}	S_{total}
Order 10	896	128	1280	384	400	400	3488≈3.4K
Order 12	3584	128	5120	384	400	400	10016≈9.8K
Order 14	14336	128	20480	384	400	400	36218≈35.3K
Order 16	57344	128	81920	384	400	400	140576≈137.3K
Order 18	229376	128	327680	384	400	400	558368≈545.3K



ra \ sa	S	MS	M	MB	B
S	B	MB	M	S	S
MS	B	MB	M	MS	S
M	B	MB	M	MS	MS
MB	B	MB	MB	M	M
B	B	B	B	MB	M

Fig. 4. Corresponding membership functions and fuzzy rules used for the step selection problem.

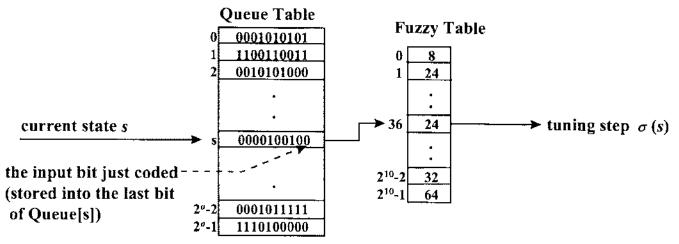


Fig. 5. Tables used for fuzzy inference.

sa means almost no transitions in the previously coded bits and suggests that higher tuning step is more suitable for good performance. Small ra means almost no repetition and suggests that lower tuning step is more suitable for good performance. Here, we use the sa and ra as inputs to infer the proper probability-tuning step σ. The corresponding membership functions and the fuzzy control rules used for the step selection problem are shown in Fig. 4. For the purpose of reducing the design complexity and achieving higher fuzzy inference speed, the inference process is implemented by table-lookups. As shown in Fig. 5, the current state s is used both to index the Queue table and to find the contents of that state's queue, which contains the ten previous bits under state s. Then, the ten bits are used to index the Fuzzy table to determine the corresponding $\sigma(s)$ for the updating of $\hat{p}_n(0|s)$.

B. The Coder

While coding the nth input bit, the *coder* accepts the $\hat{p}(0|s_n)$, calculates A_{n+1} and C_{n+1}, normalizes A_{n+1} and C_{n+1} if necessary, and produces the compressed result at the encoding mode or the uncompressed data at the decoding mode. In [2], a *bit-stuffing* technique is presented to solve the carryover problem of C_{n+1}. In our design, the *coder* applies a new bit-stuffing technique to solve the problems of source termination and carryover together with efficiency. A k-bit register called R is used as the output buffer during the encoding process. That is, the compressed bits shifted out from C are put temporarily into R instead of being sent out directly. Thus, carries generated from $C_n + NewA$, as shown in Fig. 1, can propagate into R. However, if R contains a consecutive sequence of 1-bit, the carry propagating into it would propagate through and out into the coded outputs that have been transmitted. Two extra stuffed bits are added and used to resolve this problem. If all the k bits of R are 1's, two stuffed bits "00" are added and shifted into the right side of R to block the carryover propagating. The second stuffed bit (or bit 0 of R now) may be changed to 1 if a carryover occurs during the process of encoding. Because no carryover can propagate to the same bit position of R twice as demonstrated in [2], the first stuffed bit (or bit 1 of R) will always be 0. This feature is used to indicate the source termination condition, that is, we send the consecutive $(k+1)$ 1's as the termination mark when all the input bits are coded. If the decoder receives k 1's while decoding, it will check the next two input bits (stuffed bits). If the stuffed bits are "00," the decoder just ignores the two stuffed bits. If the stuffed bits are "01," the decoder will add 1 to C and set R to 0. If the stuffed bits are "1x" (x: don't care), the decoder will end the decoding process since the termination mark [consecutive $(k+1)1$'s], which consists of $k1$'s in R and 1's for the first stuffed bit, is detected.

III. EXPERIMENTAL RESULTS AND IMPLEMENTATION

To implement our coding method, we have to use enough memory to store the related tables. S_{total}, the total size for the tables in the design, can be given as follows:

$$S_{\text{total}} = S_{\text{Addr}} + S_{\text{Prob}} + S_{\text{Queue}} + S_{\text{Fuzzy}} + S_{\text{Oset0}} + S_{\text{Oset1}}$$

where $S_{\text{Addr}}, S_{\text{Prob}}, S_{\text{Queue}}, S_{\text{Fuzzy}}, S_{\text{Oset0}}$, and S_{Oset1} represent the storage sizes used for tables Addr, Prob, Queue, Fuzzy, Oset0, and Oset1, respectively. Since the design uses an *order-o fixed-context* modeler, the real storage size is proportional to the value of o. Table II shows the memory sizes of our method for different orders.

TABLE III
AVERAGE COMPRESSION RESULTS OF THREE TYPES OF FILES FOR VARIOUS CODING METHODS

File Type	Text Files	Image Files			Binary Files
		Bi-level	Grayscale	Color	
HUFF	1.722	4.098	1.228	1.266	1.310
LZW	2.199	6.798	1.476	1.896	1.838
ARITH	1.767	5.370	1.370	1.383	1.522
JPEG_1	-	-	1.545	2.156	-
JPEG_2	-	-	1.554	2.044	-
JPEG_3	-	-	1.456	1.767	-
JPEG_4	-	-	1.522	2.314	-
JPEG_5	-	-	1.538	2.123	-
JPEG_6	-	-	1.521	2.036	-
JPEG_7	-	-	1.539	1.916	-
JBIG	-	11.967	-	-	-
AR_MF(12)	1.821	8.051	1.427	1.895	1.271
AR_MF(16)	2.096	8.128	1.538	2.011	1.502
AR_MDF(12)	1.986	8.447	1.471	2.020	1.263
AR_MDF(16)	2.413	8.513	1.596	2.172	1.561
AFT(10)	1.813	8.654	1.476	1.987	1.657
AFT(12)	2.066	8.767	1.534	2.115	1.765
AFT(14)	2.262	8.784	1.563	2.133	1.861
AFT(16)	2.424	8.900	1.605	2.214	1.952

-: Not available

For comparison purposes, we considered several different schemes for various source data. Table III shows the outcomes of compressing text, image, and binary files in various schemes. In the table, these figures represents the ratios of the total corpus size/the total compressed size for each kind of source data. In these schemes, HUFF (compact utility on UNIX) is the adaptive Huffman scheme. LZW (compress utility on UNIX) is the LZ78 coders. ARTH is an arithmetic coding scheme implemented by Jiang [7]. JPEG, obtained from the public domain facility at Stanford University,[1] is the lossless JPEG coder. It supports seven different prediction methods denoted as JPEG_1 to JPEG_7. JBIG is the lossless compression software.[2] AR_MF and AR_MDF are the arithmetic coding methods presented in [5] and [6], respectively. AFT is the proposed coder. For easier comparisons, AR_MF, AR_MDF, and AFT are all implemented with the fixed-context modelers of different orders, represented by the number in each parenthesis. To evaluate the performance of the schemes for text data, we adopt the text files used in [1] that had been gotten from the Calgary/Canterbury text compression corpus.[3] In addition, three types of image files are used for image evaluation. The bilevel images include the eight CCITT document images, four bilevel graph images, and two halftone images. The grayscale images and color images from the Waterloo BragZone[4] contain 12 8-bit grayscale test images and 8 24-bit full-color test images, respectively. The binary files include ten execution files: gcc, ghostview, xv, gzip, edit, tcsh, nroff, audioplay, fmli, and csh on UNIX. Consequently, our method achieves good compression efficiency for these different types of data.

[1] [Online]. Available FTP: ftp://havefun.stanford.edu

[2] Version 0.9 of the JBIG-Kit is obtained from ftp://ftp.informatik.uni-erlangen.de

[3] [Online]. Available FTP: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus

[4] [Online]. Available FTP: ftp://links.uwaterloo.ca/pub/BragZone

Based on the hardware optimization and tradeoff concepts taken from high-level synthesis, we have developed a simple version of VLSI architecture with order 10 for the proposed method using Cadence's Verilog simulator run on a SUN SPARC10 station. In it, an asynchronous interface circuit for I/O communication is designed; thus, the I/O operation and coding operation can be done in parallel. Besides, the concept of "design for testability" is used, and a full scan is implemented in the design. The tables Prob, Fuzzy, $\mathrm{Oset0}$, and $\mathrm{Oset1}$ are designed with ROM, while the tables Addr and Queue are designed with RAM. Under Verilog simulation, the design yields a compression and decompression ratio of 12 Mbits/s with a clock rate of 50 MHz [16]. A full version is under development, and we are investigating to reduce the sizes of the tables (memories) needed in the coding method.

IV. CONCLUSIONS

For online lossless data compression, we proposed a novel division-free adaptive arithmetic coding method that can be easily realized with VLSI technology. With the help of fuzzy inference, we achieve better compression results for various types of source data. The drawback of the method is that it requires one multiplication operation per input bit to get more accurate probability estimation. However, the method still can achieve high coding speed by using a simplified parallel multiplier proposed by us in [17], which requires approximately half of the area of a standard parallel multiplier without sacrificing any performance.

ACKNOWLEDGMENT

The authors would like to thank the editor C. S. Ravishankar and the anonymous reviewers for their valuable suggestions and careful review that helped to enhance the quality of the manuscript.

REFERENCES

[1] T. C. Bell, I. H. Written, and J. G. Cleary, "Modeling for text compression," *ACM Computing Survey*, vol. 21, no. 4, pp. 557–591, 1989.

[2] G. G. Langdon and J. Rissanen, "Compression of black-white image with arithmetic coding," *IEEE Trans. Commun.*, vol. COM-29, pp. 858–867, 1981.

[3] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[4] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, and R. B. Arps, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 717–725, 1988.

[5] D. Chevion, E. D. Karnin, and E. Walach, "High efficiency, multiplication free approximation of arithmetic coding," in *Proc. IEEE Data Compression Conf.*, 1991, pp. 43–52.

[6] L. Huynh, "Multiplication and division free adaptive arithmetic coding techniques for bi-level images," in *Proc. IEEE Data Compression Conf.*, 1994, pp. 264–273.

[7] J. Jiang, "Novel design of arithmetic coding for data compression," in *Proc. Inst. Electron. Eng. Comput. Digit. Technol.*, vol. 142, no. 6, pp. 419–424, 1995.

[8] D. L. Duttweiler and C. Chamzas, "Probability estimation in arithmetic and adaptive-huffman entropy coders," *IEEE Trans. Image Processing*, vol. 4, pp. 237–249, Mar. 1995.

[9] Arps, T. Truong, D. Lu, R. Pasco, and T. Friedman, "A multi-purpose VLSI chip for adaptive data compression of bilevel images," *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 775–794, 1988.

[10] G. Feygin, P. G. Gulak, and P. Chow, "Minimizing error and VLSI complexity in the multiplication free approximation of arithmetic coding," in *Proc. IEEE Data Compression Conf.*, 1993, pp. 118–127.

[11] ——, "Architectural advances in the VLSI implementation of arithmetic coding for binary image compression," in *Proc. IEEE Data Compression Conf.*, 1994, pp. 254–263.

[12] B. Fu and K. K. Parhi, "Two VLSI design advances in arithmetic coding," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1995, pp. 1440–1443.

[13] W. Pennebaker and J. Mitchell, *JPEG Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.

[14] L. A. Zadeh, "Fuzzy sets," *Inform. Control*, vol. 8, no. 6, pp. 338–353, 1965.

[15] C. C. Lee, "Fuzzy logic in control systems: Fuzzy logic controller—Part I & Part II," *IEEE Trans. Syst., Man, Cybern.*, vol. 20, pp. 404–435, Mar./Apr. 1990.

[16] J. M. Jou, S.-R. Kuang, and Y.-L. Chen, "The design of an adaptive on-line arithmetic code codec chip," in *Proc. IEEE Int. Symp. Circuits and Systems*, 1996, pp. 253–256.

[17] J. M. Jou and S.-R. Kuang, "Design of a low-error fixed-width multiplier for DSP applications," *Electron. Lett.*, vol. 33, no. 19, pp. 1597–1598, 1997.