A DATA COMPRESSION METHOD USING ADAPTIVE BINARY ARITHMETIC CODING AND FUZZY LOGIC

Jer Min JOU, Pei-Yin CHEN

Department of Electrical Engineering, National Cheng Kung University
1 University Road, Tainan, Taiwan 70101, Republic of China
Tel: +886-6-2387092 Fax: +886-6-2345482 E-mail:jou@cad2.ee.ncku.edu.tw

Abstract

This paper describes an on-line lossless data compression method using adaptive binary arithmetic coding. To achieve better compression efficiency, we employ an adaptive fuzzy-tuning modeler, which uses fuzzy inference to deal with the problem of conditional probability estimation. The design is simple, fast and suitable for VLSI implementation because we adopt the table-look-up approach. As compared with the outcomes of other lossless coding schemes, our results are good and satisfactory for various types of source data.

Keywords: Lossless data compression, Adaptive arithmetic coding, Fuzzy inference

1. Introduction

Since arithmetic coding [1-8] can approach the entropy limit as long as the statistics are accurate, it is superior to Huffman coding, and has been used vastly for data compression. However, arithmetic coding tends to be slow, because in standard form it requires at least one multiplication operation per input symbol. Moreover, an extra division operation, if used adaptively, may be needed at every coding step. 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 [2-7]. Some VLSI architectures of arithmetic coding also have been presented [1-4] to improve the coding speed. The drawbacks of the methods are that the probability estimations are not accurate enough for various types of source data or the arithmetic computations are still complex as far as hardware implementation is considered. 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 always provides accurate probability estimation for different types of data. Here, we use fuzzy inference [9][10] to solve the problem of probability estimation.

This paper presents a novel division-free binary arithmetic coding method, suited for implementation with VLSI technology. In this design we adopt a binary *order-o fixed-context* model, which uses the *o* previous symbols as the state (or context). To reduce the complexity of hardware implementation and to prompt the coding throughput, we use the table-look-up method to construct an *adaptive fuzzy-tuning modeler*. With the aid of fuzzy inference process which dynamically selects the probability-tuning step, the *modeler* can determine the estimated probabilities more efficiently and precisely. Therefore, the compression efficiency of our method can be improved very much. According to the experimental results, the proposed method works better than other lossless compression methods, such as Huffman, approximate arithmetic [3, 4, 6, 8], and Lempel-Ziv, for different types of source data: text files, image files, and binary files. Besides, in the design some on-line processing issues of arithmetic coding such as source termination and carry-over are solved efficiently. A VLSI architecture for the method has been developed and implemented by using $0.8\mu m$ CMOS technology, and it yields an encoding and decoding rate of 12 Mbits/sec with a clock rate of 50 MHz.

2. The 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 sub-intervals in proportion to the conditional probabilities of the symbol being encoded. Let A_n denote the width of the selected sub-interval at stage n and C_n represent the position of the lower boundary of the selected sub-interval. While encoding the nth symbol, x, in the input stream, the 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 $\hat{p}_n(x \mid s)$ is the estimated, conditional probability of symbol x at stage n given the previous string s.

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

2.1 The adaptive fuzzy-tuning modeler

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

$$\hat{p}_n(0 \mid s) = \frac{count0_n(s)}{countN_n(s)}, \tag{3}$$

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

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

$$\hat{p}_{n+1}(0 \mid s) = \frac{count0_{n+1}(s)}{countN_{n+1}(s)}$$
$$= \begin{cases} \dfrac{count0_n(s)+1}{countN_n(s)+1} & \text{if the } n\text{th bit is a 0's,} \\[2ex] \dfrac{count0_n(s)}{countN_n(s)+1} & \text{if the } n\text{th bit is a 1's.} \end{cases} \tag{4}$$

Apparently, the conditional probabilities will change faster/slower when few/many input data has 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:

$$\hat{p}_{n+1}(0 \mid s) = \begin{cases} \dfrac{\dfrac{count0_n(s)}{\sigma_n(s)}+1}{\dfrac{countN_n(s)}{\sigma_n(s)}+1} & \text{if the } n\text{th bit is a ' s,} \\[4ex] \dfrac{\dfrac{count0_n(s)}{\sigma_n(s)}}{\dfrac{countN_n(s)}{\sigma_n(s)}+1} & \text{if the } n\text{th bit is a 1's,} \end{cases} \tag{5}$$

where $\sigma_n(s)$ is the *probability-tuning step* used to reflect the degree of variation of conditional probabilities. To avoid the division operation, two offset tables are applied to approximate the $\hat{p}_{n+1}(0 \mid s)$. If the current input bit is a 0's and the tuning step σ is given, we can prefind by using (5) the value of $\hat{p}_{n+1}(0 \mid s)$ for each of the 128 possible values of $\hat{p}_n(0 \mid s)$ stored in the Prb table, calculate the 128 offsets (or distances) between the corresponding indexes in Prb table for the values of $\hat{p}_{n+1}(0 \mid s)$ and $\hat{p}_n(0 \mid s)$, and then store the 128 offsets in an offset table, named as $\mathrm{Ost0}_\sigma$. Similarly, the $\mathrm{Ost1}_\sigma$ table stores the 128 offsets for the current input "1." Figure 2 shows the block diagram of *AFTM*. By changing the pointer's value stored in the Adr table with the distance provided by either the table $\mathrm{Ost0}_\sigma$ or $\mathrm{Ost1}_\sigma$, we can obtain the corresponding $\hat{p}_{n+1}(0 \mid s)$ as follows:

$$\hat{p}_{n+1}(0 \mid s) = \hat{p}_n(0 \mid s) + \Delta\hat{p}_n(0 \mid s)$$
$$= \begin{cases} \mathrm{Prb}[\mathrm{Adr}[s] + \mathrm{Ost0}_\sigma[\mathrm{Adr}[s]]] & \text{if the } n\text{th bit is a 0 s,} \\ \mathrm{Prb}[\mathrm{Adr}[s] + \mathrm{Ost1}_\sigma[\mathrm{Adr}[s]]] & \text{if the } n\text{th bit is a 1 s.} \end{cases}$$

In fact, there are a large number of tuning steps can be selected. It is certainly impossible and unpractical to use too many steps, since too many tables (memories) are required. Therefore, after many experiments we chose only five different steps: 8, 24, 32, 40, and 64 in the design. As shown in Fig. 2, the five offset tables are indexed by the value of Adr[*s*] simultaneously, and one of the five offset values, selected with a proper tuning step σ generated by the fuzzy inference process, is used to determine the $\hat{p}_{n+1}(0 \mid s)$.

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 [10]. The main function of fuzzy inference is to select a proper probability-tuning step to tune the $\hat{p}_n(0 \mid s)$ effectively. For each possible state, we use a 10-bit queue to record the 10 previously coded bits under the state. Two evaluation parameters are used to observe the queue: the switching activity and the repeating activity. The switching activity (*sa*) means the number of binary symbol transitions (0 changes to 1 or 1 changes to 0) in the state queue, and the repeating activity (*ra*) means the number of identical bits counted from the last bit of the queue. Obviously, small *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 the 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. 3. For the purpose of reducing the design complexity and achieving higher inference speed, the fuzzy inference process is implemented by table-lookups.

2.2 The Coder

While coding the *n*th input bit, the *coder* accepts the $\hat{p}(0 \mid s_n)$, calculates the A_{n+1} and C_{n+1}, normalizes the A_{n+1} and C_{n+1} if necessary, and produces the compressed result at the encoding mode and the uncompressed data at the decoding mode. Besides, in the design the *coder* uses a new bit-stuffing technique to solve the problems of source termination and carry-over 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 $C_n + NewA$ at *Step* 2 in Fig. 1 can propagate into *R*. However, if *R* contains a consecutive sequence of 1-bits, the carry propagating into it would propagate through and out into the coded outputs which have been transmitted. Two extra stuffed bits are added and used to resolve this problem. If all of 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 carry-over propagating. The second stuffed bit (or bit 0 of *R* now) may be changed to 1 if a carry-over occurs during encoding. Because no carry-over can propagate to the same bit position of *R* twice, as demonstrated in [7], 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 "1*x*" (*x*: don't care), the decoder will end the decoding process since the termination mark (consecutive (*k*+1) 1's), which consists of *k* 1's in *R* and a 1's in the first stuffed bit, is detected.

3. Experimental results and implementation

For comparison purposes, we considered some different schemes for various kinds of source data. Table 1 shows the compression efficiency of text, image, and binary files for various schemes. In it, we chose randomly 100 text files, 50 image files and 40 binary files, and let the total size of each type of files be about 30 Mbytes. HUF (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 [4]. AR_B, AR_MF, and AR_MDF are the arithmetic coding methods presented in [8], [3] and [6], respectively. AFT is the proposed coder. For easier comparisons, AR_B, AR_MF, AR_MDF, and AFT are all implemented with the binary order-10 and order-16 fixed-context modeler, respectively. Consequently, our method achieves better compression efficiency for the three types of data.

Based on the hardware optimization and tradeoff concepts taken from high level synthesis, we have developed a feasible VLSI architecture for the proposed method using Cadence's Verilog simulator run on a SUN SPARC10 station. 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. Under Verilog simulation, it yields a compression and decompression ratio of 12 Mbits/sec with a clock rate of 50 MHz.

4. Conclusions

For on-line lossless data compression, we proposed a novel adaptive arithmetic coding method, which can be easily realized with VLSI technology. With the help of fuzzy inference, better compression efficiency can be achieved for various types of source data. The drawback of the method is that we require 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 [11], which requires approximately half of the area of a standard parallel multiplier without sacrificing any performance.

5. Acknowledgement

This research was supported in part by the National Science Council, Republic of China, under the Grant NSC-86-2221-E-006-022.

References

[1] B. Fu and K. K. Parhi, "Two VLSI design advances in arithmetic coding," in *IEEE Int. Symposium of Circuits and Systems*, 1995, pp. 1440-1443.

[2] 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. and Develop.*, vol. 32, no. 6, pp. 717-725, 1988.

[3] 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 Confernce*, 1993, pp. 118-127.

[4] J. Jiang, "Novel design of arithmetic coding for data compression," *IEE Proc.-Comput. Digit. Tech.*, vol. 142, no. 6, pp. 419-424, 1995.

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

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

[7] G. G. Langdon and J. Rissanen, "Compression of black-white image with arithmetic coding," *IEEE Trans. Communications*, vol. 29, no. 6, pp. 858-867, 1981.

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

[9] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 6, pp. 338-353, 1965.

[10] C. C. Lee, "Fuzzy logic in control systems: Fuzzy logic controller-Part I & Part II," *IEEE Trans. Syst., Man, Cybern.*, vol. 20, no. 2, pp. 404-435, 1990.

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

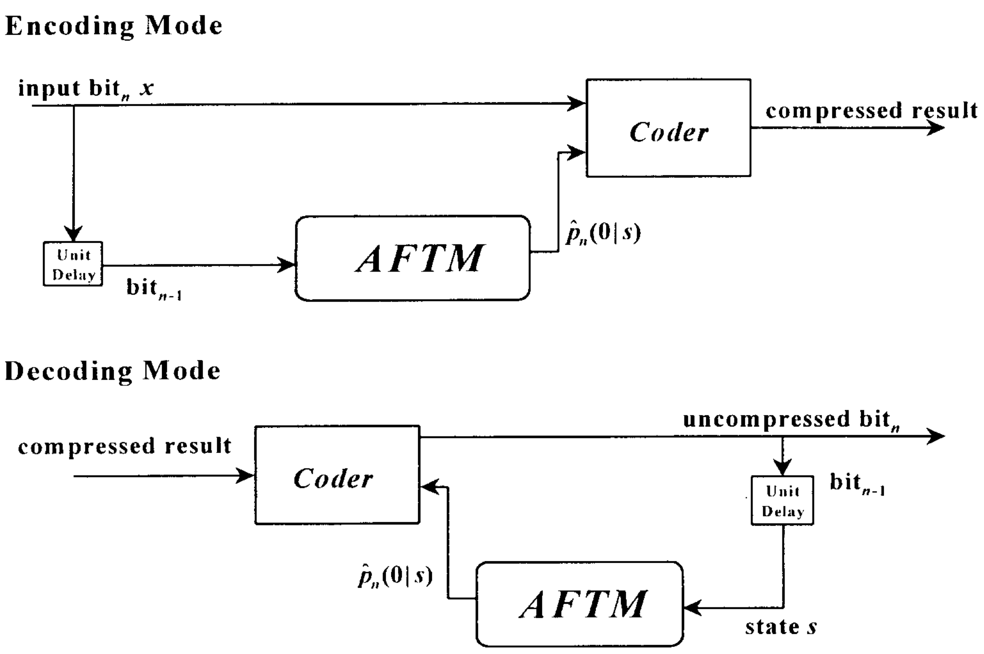


Fig. 1. The encoding procedure of the proposed method.

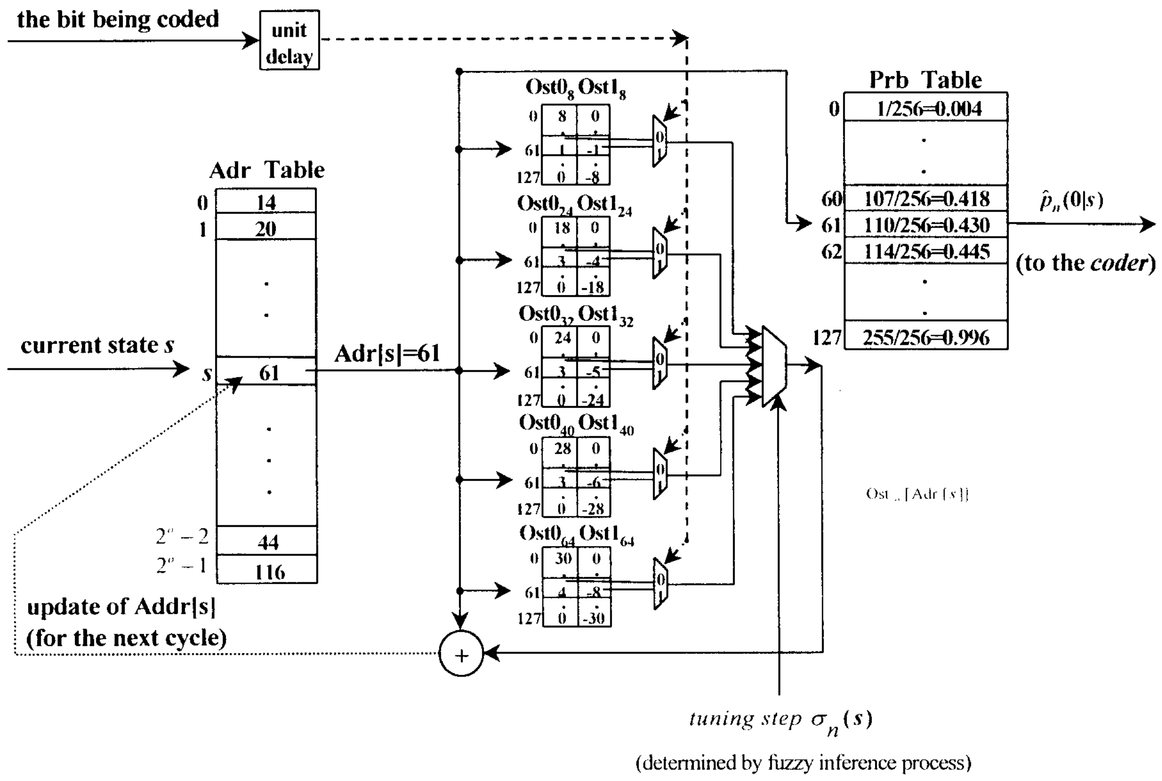
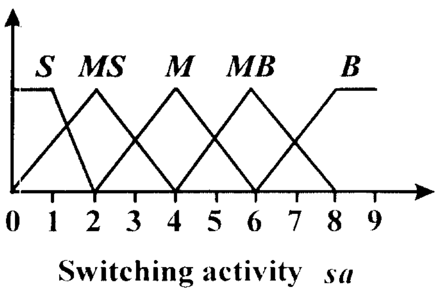
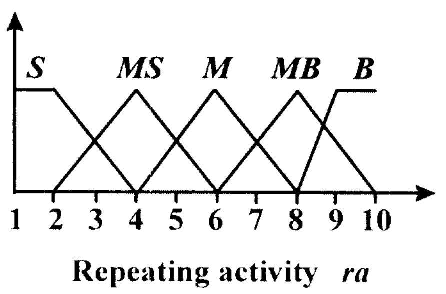
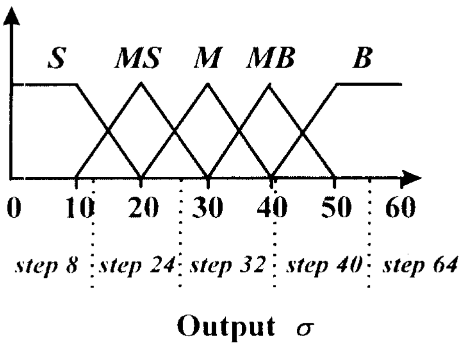


Fig. 2. The block diagram of the *AFTM*.







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

S: Small *MS*: Medium Small *M*: Medium
MB: Medium Big *B*: Big

Fig. 3. The corresponding membership functions and fuzzy rules used for the step selection problem.

TABLE 1 The average compression results of three types of files for various coding methods

File type	100 Text Files	50 Image Files	40 Binary Files	Average
Total Size (bytes)	30074354	30114290	30012386	*ce*
HUF	44.0%	10.9%	24.0%	26.3%
LZW	59.2%	20.4%	31.0%	36.9%
ARTH	49.0%	19.8%	34.4%	34.4%
AR_B(10)	50.3%	27.8%	29.6%	35.9%
AR_MF(10)	49.5%	25.4%	28.1%	34.3%
AR_MDF(10)	50.2%	27.6%	29.4%	35.7%
AFT(10)	51.1%	31.1%	39.1%	40.4%
AR_B(16)	60.4%	34.7%	40.2%	45.1%
AR_MF(16)	59.1%	34.5%	39.0%	44.2%
AR_MDF(16)	60.3%	35.4%	40.3%	45.3%
AFT(16)	63.0%	35.9%	48.7%	49.2%

$$ce\text{: compression efficiency} = \frac{\text{the number of bits of original input} - \text{the number of bits of compressed output}}{\text{the number of bits of original input}} \times 100\%.$$