International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 1 Issue 8, October - 2012

Improved Performance Of Arithmetic Coding By Extracting Multiple Bits At A Time

Jyotika Doshi
GLS Inst.of Computer Technology
Opp. Law Garden, Ellisbridge
Ahmedabad-380006, INDIA

Savita Gandhi
Dept. of Computer Science; Gujarat University
Navrangpura Ahmedabad-380009, INDIA

Abstract

Arithmetic coding is a very efficient and most popular entropy coding technique. Compression ratio cannot be improved as it is dependent on statistical probability model. An improvement that is possible is only with respect to time and space complexity. In arithmetic coding, for each symbol, it renormalize the interval and output code bits till an interval becomes 2b-2 wide, where b is number of bits used to store range. In conventional implementation of this algorithm, at each iteration, single bit is output in the coded message when most significant bits of low and high of a subinterval match. Thereafter it outputs its complement as many times as an underflow has occurred before. In this paper, an algorithm is implemented by extracting more than one bit at a time instead of just one. Here it outputs one bit, then its complement for underflow number of times and then remaining number of bits. Doing so, execution speed is increased. As compared to conventional implementations, there is a performance 17% gain in encoding and 10% in decoding, without affecting compression ratio.

Index Terms: arithmetic coding, data compression, improved performance, lossless data compression, multi-bits output at a time

1. Introduction

Arithmetic coding was introduced by Rissanen [1] in 1976. Arithmetic coding [2]-[5] is a very efficient entropy coding technique. It is optimal in theory and nearly optimal in practice, in that it encodes arbitrary data with minimal average code length. It works with any sample space so it can be used for the coding of text in arbitrary character sets as well as binary files. It encodes data using a variable number of bits. The number of bits used to encode each symbol varies according to the probability assigned to that symbol. The idea is to assign short codeword to more probable events and long codeword to less probable events [5].

Arithmetic coding has been developed extensively since its introduction several decades ago, and is notable for offering extremely high coding efficiency. That is why it is most popular for entropy coding and widely used in practice. There are many data compression methods that first transform input data by some algorithm, and then compress resulting data using arithmetic coding [18]. For instance, the run length code, many implementations of Lempel-Ziv codes, the context-tree weighting method [6], Grammar—based codes [7]-[8] and many methods of image compression, audio and video compression. While many earlier-generation image and video coding standards such as JPEG, H.263, and MPEG-2, MPEG-4 relied heavily on Huffman coding for the entropy coding steps in compression, recent generation standards including JPEG2000 [9] and H.264 [10]-[13] utilize arithmetic coding. It is also considered as a suitable candidate for a possible encryption-compression [14]-[16] combine providing security [17] and reduced size for internet applications.

Arithmetic coding has a major advantage over other entropy coding methods, such as Huffman coding. Huffman coding uses an integer number of bits for each code, and therefore only has a chance of reaching entropy performance when probability of a symbol is a power of 2 for all the symbols. Arithmetic code encodes arbitrary data with minimal average code

length, so its coding efficiency is generally higher. The main disadvantage of arithmetic coding is its relatively high computational complexity. It is usually slower than Huffman coding and other fixed-length to variable-length coding schemes [19]. Compression ratio cannot be further improved as compression ratio that can be reached by any encoder under a given statistical model is actually bounded by the quality of that model. However one can optimize one's algorithms in at least two dimensions: memory usage and speed [20]. Here we have worked to increase an execution speed.

Existing conventional implementations [20]-[27] output one bit at a time whereas our implementation output multiple bits at a time. It has increased the performance drastically without any loss in compression ratio of conventional implementations.

2. Statistical model in Arithmetic Coding

Arithmetic coding method is based on the fact that the cumulative probability of a symbol sequence corresponds to a unique subinterval of the initial interval [0, 1). Before starting encoding process, symbols are assigned segments on interval [0, 1) according to their cumulative probabilities. It doesn't matter which symbols are assigned which segment of the interval as long as it is done in the same manner by both the encoder and the decoder [24]. If S = (S1, S2, . . ., Sn) is the alphabet of a source having n symbols with an associated cumulative probability distribution P = (P1, P2, … , Pn), an initial interval [0, 1) is divided into n subintervals as [0, P1), [P1, P2), [P2, P3), …,[Pn-1, Pn) where Pi is the cumulative probability of symbol Si. Each subinterval length is proportional to the probability of the symbols [22].

When arithmetic coding is implemented using integer arithmetic, a coding interval is usually represented by [L,H), where L and H are two b-bit integers denoting the interval's lower end and higher end, respectively. An initial interval is [0,1). Cumulative probability is a ratio of cumulative frequency and total frequency. So instead of using cumulative probability, cumulative frequencies are used in computation. Thus the probability model is described by an array, [F0, F1, F2, . . . ,Fn], where Fi (0 ≤ i ≤ n) is f-bit integer (f ≤ b − 2) representing the lower and upper bounds of cumulative frequency segments. For symbol Si, Fi-1 is lower bound and Fi is upper bound.

3. Encoding and decoding algorithm

Encoding Algorithm

- Interval=[0,1)
- Qtr1=range/4, Qtr2=2*Qtr1, Qtr3=3*Qtr1
- cnt=0, cnt is count for occurrences of underflow
- Repeat till not EOF
  - Read symbol
  - Compute corresponding new interval [low, high)
  - Repeat (renormalization loop)
    - Case 1: low and high falls in upper half, i.e. in [0.5,1). So low >= Qtr2. Here matching most significant bit (msb) is 1.
      - output bit 1
      - o/p bit 0 cnt times, cnt=0
      - left shift low and high by 1 position, i.e. double low and high (padding on right: low with 0 and right with 1)
    - Case2: Both low and high falls in lower half, i.e. in [0, 0.5). So high < Qtr2. Here matching most significant bit is 0.
      - output bit 0
      - o/p bit 1 cnt times, cnt=0
      - left shift low and high by 1 position, i.e. double low and high (padding on right: low with 0 and right with 1)
    - Case3: low falls in [Qtr1, Qtr2) and high falls in [Qtr2, Qtr3), i.e. (high < Qtr3) && (low ≥ Qtr1). Here msb is not matching and 2nd bit differ by 1, thus underflow occurs.
      - cnt++ (underflow)
      - extract 2nd bit from low and high and then double, i.e subtract Qtr1 from low and high, double low and high
    - Other cases: (low<Qtr2) && (high ≥ Qtr3), i.e. interval is more than half
      - Break loop
- At EOF
  - cnt++
  - if low < Qtr1, i.e. its most significant bit is 0, then output bit 0 and cnt times 1else output bit 1 and cnt times 0

During decoding, new interval is computed and bits are extracted from the coded message exactly the same way as done during encoding.

4. Renormalizing Interval

As explained in section 3, in arithmetic coding, while encoding and decoding each symbol, it output a bit and expands the current interval. This is considered as renormalization of an interval. An algorithm renormalizes an interval in a loop till interval length becomes more than half of the interval.

5. Conventional Implementations

All integer implementations of arithmetic coding uses cumulative frequencies a statistical model as explained in section 2. As explained in section 3, while encoding and decoding each symbol, it renormalizes an interval in a loop till interval length becomes more than half of the interval. During renormalization (section 4), it performs two tasks: outputting code bits and expanding the current interval.

In conventional implementations [20]-[27], renormalization is performed through the renormalization loop in a bitwise manner, i.e., during each execution of the renormalization loop, one code bit is generated and the current interval is doubled. Case 1 and 2 of algorithm explained in section 3 are usually combined that output most significant matching bit (whether 0 or 1), output underflow cnt times complement of msb and then double the interval.

6. Proposed Implementation

As seen in conventional implementations, algorithm outputs only one bit at a time in single iteration. Here it is proposed to extract and output more than one bit in a single iteration and expand the interval accordingly. This reduces the number of iterations used in renormalization. It has resulted in tremendous improvement in the execution speed without compromising on compression ratio.

6.1. Using Statistical Model

Same as in conventional implementation (as in section 2)

6.2. Renormalizing Interval

Here is the difference between conventional and proposed implementations.

Renormalization loop in proposed implementation:

- Repeat till case 1 or 2 or 3 (renormalization loop)
  - Case 1, 2: nBits most significant bits are matching (Either 0 or 1)
    - Compute number of most significant bits matching, say nBits
    - output first most significant matching bit
    - o/p cnt times the complement of msb, cnt=0
    - o/p remaining nBits-1 most significant matching bits
    - expand interval shifting low and high to left by nBits position (padding on right: low with 0 and high with 1)
  - Case 3: low falls in [Qtr1, Qtr2) and high falls in [Qtr2, Qtr3), i.e. most significant bit is not matching and 2nd bit differ by 1
    - cnt++ (increment underflow counter)
    - extract 2nd bit from low and high and then double

6.3. Computing number of matching most significant bits

When bitwise xor operation is performed on bits, resulting bit is 0 when both operand bits are matching and 1 otherwise. Thus low xor high will have resulting bit 0 wherever bits of low and high are matching. Now the only task is to determine occurrences of leading consecutive zeros, i.e. the position of first occurrence of bit 1 from left. This can be done as shown below.

- tmp=low XOR high
- Determine 1st occurrence of bit 1 from left
  - Either using expression int(log2(tmp))
  - Or using shift in a loop
- Assuming low and high are represented using b bits, nBits=b-int(log2(tmp))+1 will be number of consecutive zeros on left in tmp, i.e. number of matching most significant bits in high and low

There might be a problem in using $\log_2(x)$ function, as it is not be available in all C (ex. TurboC 3.0). In such cases, use log(tmp)/log(2) where log is natural logarithm. An alternative is to shift left in a loop and terminate it when first bit is 1.

7. Experimental Results

Both the conventional and proposed algorithms are implemented using 16 bit Turbo C compiler on Intel(R) Pentium (R) D, CPU 3.00 GHz, 1 GB RAM. Execution time is measured in seconds for 17 files with varying sizes and different file types. Some of the test files are selected from Calgary and Canterbury corpus, a widely used benchmark and also from web site compression.ca/act/act_files.html. Selected test files are of various types like text files, image files, audio files, excel files, power point files, word documents, executable files etc. Used benchmark files are: act2may2.xls, calbook2.txt, ca-obj2, cal-pic, frymire.tif, kennedy.xls, lena3.tif, monarch.tif, pine.bin, ptt5, world95.txt.

Here terms ACEN and ACDE are used for existing conventional implementations of arithmetic coding for encoding and decoding respectively. Similarly terms ACEC_JS and ACDE_JS are used for proposed implementations.

Table 1 lists files used for testing of both existing and proposed implementations. Table 2 and Table 3 presents compression and decompression time (seconds) respectively and gain (in percentage) in speed using proposed implementation. Figure 1 and 2 shows comparison of execution time of encoding and decoding respectively.

TABLE I
TEST FILES USED

No	File name	Size (Bytes)
1	act2may2.xls	1348036
2	calbook2.txt	610856
3	cal-obj2	246814
4	cal-pic	513216
5	cycle.doc	1483264
6	every.wav	6994092
7	family1.jpg	198372
8	frymire.tif	3706306
9	kennedy.xls	1029744
10	lena3.tif	786568
11	linuxfil.ppt	246272
12	monarch.tif	1179784
13	pine.bin	1566200
14	ptt5	513216
15	sadvchar.pps	1797632
16	shriji.jpg	4493896
17	world95.txt	3005020

TABLE II
COMPRESSION (ENCODING) TIME

		ACEN	ACEN-JS	%Gain
No	File name	Sec	Sec	
1	act2may2.xls	2.307	2.0329	11.8812
2	calbook2.txt	1.099	0.989	10.0091
3	cal-obj2	0.495	0.3846	22.3030
4	cal-pic	0.6593	0.6044	8.3270
5	cycle.doc	2.637	2.3077	12.4877
6	every.wav	15.000	12.0329	19.7807
7	family1.jpg	0.439	0.3297	24.8975
8	frymire.tif	6.648	5.4945	17.3511
9	kennedy.xls	1.640	1.4835	9.5427
10	lena3.tif	1.703	1.3187	22.5661
11	linuxfil.ppt	0.439	0.3846	12.3918
12	monarch.tif	2.528	1.978	21.7563
13	pine.bin	3.077	2.5824	16.0741
14	ptt5	0.604	0.6044	0.0000
15	sadvchar.pps	3.791	3.0769	18.8367
16	shriji.jpg	9.505	7.6923	19.0710
17	world95.txt	5.604	4.8351	13.7206
			Overall Performance	**17.2805**

TABLE III
DECOMPRESSION (DECODING) TIME

		ACEN	ACEN-JS	%Gain
No	File name	Sec	Sec	
1	act2may2.xls	6.868	6.0989	11.1983
2	calbook2.txt	3.351	3.1319	6.5383
3	cal-obj2	1.374	1.2637	8.0277
4	cal-pic	2.362	1.5384	34.8688
5	cycle.doc	7.912	7.1429	9.7207
6	every.wav	40.769	36.7582	9.8379
7	family1.jpg	1.154	1.0989	4.7747
8	frymire.tif	19.615	18.3516	6.4410
9	kennedy.xls	5.270	3.8462	27.0171
10	lena3.tif	4.560	4.1209	9.6373
11	linuxfil.ppt	1.31	1.1542	11.8931
12	monarch.tif	6.868	6.1538	10.3990
13	pine.bin	8.791	8.1319	7.4974
14	ptt5	2.362	1.5385	34.8645
15	sadvchar.pps	10.384	9.4505	8.9898
16	shriji.jpg	26.099	23.6264	9.4739
17	world95.txt	16.648	13.6813	17.8202
			Overall Performance	**11.2401**

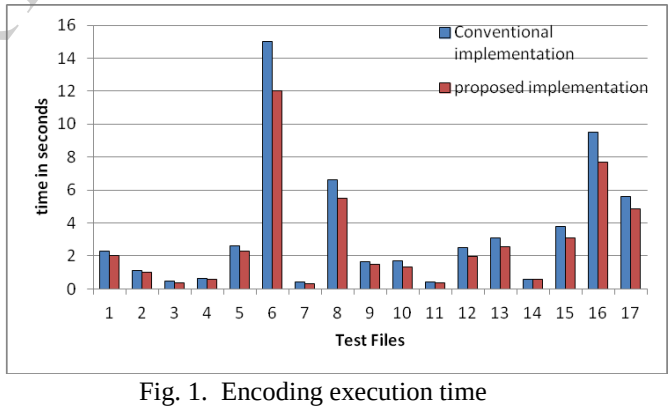

Fig. 1. Encoding execution time

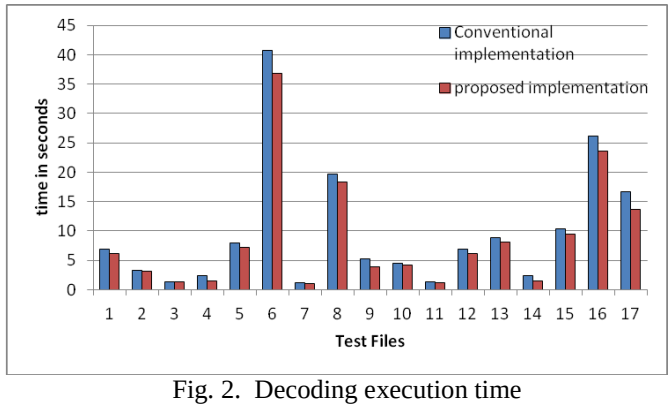

Fig. 2. Decoding execution time

8. Conclusion

As compared to existing conventional implementations of arithmetic coding, proposed implementation has resulted into a tremendous gain in execution speed of about 17% while encoding and 11% while decoding, without any compromise in compression ratio.

9. Further enhancement

Both encoding and decoding can be further improved in execution speed by determining how many times consecutive underflow will occur. When underflow occurs, most significant bit (msb) is not matching and next bit differs by 1. So msb is 1 and next bit is 0 in high bound of interval, whereas msb is 0 and next bit is 1 in low bound. Our interest is to find number of leading 0s (say n0) after msb in high and how many leading 1s (say n1) after msb in low. Using this, consecutive occurrences of underflow can be computed minimum of these two numbers n0 and n1.

10. References

[1] J. Rissanen, "Generalized kraft inequality and arithmetic coding", *IBM J. Res. Develop.*, vol. 20, pp. 198–203, May 1976.

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

[3] C. B. Jones, "An efficient coding system for long source sequences", *IEEE Trans. Inform. Theory*, vol. IT–27, pp. 280–291, 1981.

[4] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression" *Commun. ACM*, vol. 30, pp. 520–540, 1987.

[5] P. G. Howard and J. S. Vitter, "Arithmetic coding for data compression", *Proc. IEEE*, vol. 82, pp. 857–865, 1994.

[6] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "The context-tree weighting method: Basic properties", *IEEE Trans. Inform. Theory*, vol.41, pp. 653–664, May 1995.

[7] J. C. Kieffer and E. H. Yang, "Grammar-based codes: A new class of universal lossless source codes", *IEEE Trans. Inform. Theory*, vol. 46, pp. 737–754, 2000.

[8] J. C. Kieffer, E. H. Yang, G. J. Nelson, and P. Cosman, "Universal lossless compression via multilevel pattern matching", *IEEE Trans. Inform.Theory*, vol. 46, pp. 1227–1245, July 2000.

[9] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice*. Norwell, MA: Kluwer Academic, 2002.

[10] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst.Video Technol.*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[11] Detlev Marpe, Heiko Schwarz, and Thomas Wiegand, "Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard", *IEEE Trans. On Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 620-636, July 2003

[12] M. Dyer,D. Taubman, and S. Nooshabadi, "Improved throughput arithmetic coder for JPEG2000", *Proc. Int. Conf. Image Process.*, Singapore, Oct. 2004, pp. 2817–2820.

[13] R. R. Osorio and J. D. Bruguera, "A newarchitecture for fast arithmetic coding in H.264 advanced video coder", *Proc. 8th Euromicro Conf. Digital System Design*, Porto, Portugal, Aug. 2005, pp. 298–305.

[14] Ranjan Bose,Saumitr Pathak, "A Novel Compression and Encryption Scheme Using Variable Model Arithmetic Coding and Coupled Chaotic System", *IEEE Trans. Circuits and Systems*, vol. 53, no. 4, pp. 848-857, April 2006

[15] Kwok-Wo Wong, Qiuzhen Lin, Jianyong Chen, "Simultaneous Arithmetic Coding and Encryption Using Chaotic Maps", *IEEE Trans. On Circuits and Systems, vol. 57, no. 2*, pp. 146-150, February 2010

[16] M. Grangetto, E. Magli, and G. Olmo, "Multimedia selective encryption by means of randomized arithmetic coding," *IEEE Trans. Multimedia*, vol. 8, no. 5, pp. 905–917, Oct. 2006.

[17] Hyungjin Kim, Jiangtao Wen, John D. Villasenor, "Secure Arithmetic Coding", *IEEE Trans. On Signal Processing*, vol. 55, no. 5, pp. 2263-2272, May 2007

[18] Boris Ryabko and Jorma Rissanen, "Fast Adaptive Arithmetic Code for Large Alphabet Sources With Asymmetrical Distributions" , IEEE COMMUNICATIONS LETTERS, VOL. 7, NO. 1, JANUARY 2003 pp. 33-35

[19] A. Moffat, N. Sharman, I. H. Witten, and T. C. Bell, "An empirical evaluation of coding methods for multi-symbol alphabets," *Inf. Process.Manage.*, vol. 30, pp. 791–804, 1994.

[20] E.Bodden, MalteClasen, Joachim Kneis, "Arithmetic Coding revealed-A guided tour from theory to praxis", Sable Technical Report No. 2007-5, May 2007, available at http://www.bodden.de/legacy/arithmetic-coding/

[21] I.MengyiPu, *Fundamental Data Compression*, Butterworth-Heinemann, 2006

[22] D. Salomon, *Data Compression-The Complete Reference*, 3rd Edition, Springer, 2004

[23] A.Drozdek, *Elements of data compression*, Brooks/Cole, 2002

[24] M. Nelson and Jean-loupGailly, *The Data Compression Book*,2nd edition, M&T Books, New York, NY 1995
[25] *Compression and Coding Algorithms*: Kluwer Academic Publishers, 2002.
[26] A. Moffat, R. Neal, and I. Witten, "Arithmetic coding revisited," *ACM Trans. Inform. Syst.*, vol. 16, no. 3, pp. 256–294, July 1998.
[27] A. Said, "Introduction to Arithmetic Coding - Theory and Practice", available at http://www.hpl.hp.com/techreports/2004/HPL-2004-76.pdf