Computing Number of Bits to be Processed using Shift and Log in Arithmetic Coding

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

Savita Gandhi
Department of Computer Science
Gujarat University
Ahmedabad-380009, INDIA

ABSTRACT

Arithmetic coding is a very efficient and most popular entropy coding technique. 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). In this method, when encoding a symbol, it first computes new interval [low, high) based on cumulative probability segment of the symbol. Thereafter it iterates in a loop to output code bits till the interval becomes 2^{b-2} wide, where b is number of bits used to store range of an interval. In conventional implementation of arithmetic coding algorithm, in single loop iteration, only one bit is processed at a time. When most significant bit (msb) of low and high of a subinterval matches, it writes this msb in coded message and doubles the interval by extracting msb. When underflow occurs, it extracts second msb to expand an interval. Processing such single bit and expanding an interval is also called renormalization in a loop. In this paper, an upgradation of this conventional arithmetic coding algorithm is proposed, wherein more than one bit is processed at a time instead of just single bit in single iteration. This proposed multi-bit processing arithmetic coding algorithm is implemented here to reduce the iterations needed in renormalizing an interval. It is observed that processing multiple output bits at a time leads to big improvement in execution time. To determine the number of maximum possible matching most significant bits to output, two alternatives are used here; (i) Using shift operation in a loop (ii) Using log function. It is found that first technique is far better than second one with respect to execution time. As compared to conventional implementations processing single bit at a time, about 52% overall saving in execution time is observed when processing multi-bits using shift operation in a loop; whereas about 31% overall loss in performance is observed with the technique of using log function. We have also tried these two alternative ways to determine the number of consecutive occurrences of underflow and process them all in single iteration; but it has not shown any significant gain in speed. As expected, in using any of the above methods, there is no compromise in compression ratio.

General Terms

Data Compression Algorithm

Keywords

Arithmetic coding, computing number of output bits, computing number of consecutive occurrences of underflow, faster execution, lossless data compression, multi-bit processing, renormalizing interval

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 execution speed.

Existing conventional implementations [20]-[27] output one bit at a time. Arithmetic encoding algorithm is explained in detail in section 3. It processes single output bit or single

underflow occurrence in single iteration as shown in section V.

Authors in the earlier paper [28] had proposed enhancement of traditional arithmetic coding by processing more than one bit in single iteration. This implementation [28] was done using four nested-if statements and thus processing maximum up to four bits at a time in single renormalizing iteration. It showed an overall gain of 17% in the execution speed of encoding, which led us to think further trying of maximum possible bits to process in a single iteration.

Thus, in this paper, we have proposed and implemented a **multi-bit processing arithmetic coding algorithm** which computes maximum number of possible bits that can be processed at a time in single loop iteration. This number of desirable bits is determined using two ways: (i) using shift operation in a loop (ii) using log function. Corresponding algorithm and programming code is given in section 6 and 7.

Here we have implemented our modified multi-bit processing arithmetic coding algorithm using 16 bit wide interval, so we can extract and process maximum up to 16 bits in single iteration. However, given logic and code can be applied for any number of bits depending upon the width (number of bits used) of an interval.

As compared to conventional method of outputting single bit at a time during compression, outputting multiple bits at a time using shift operations in a loop shows an overall gain of about 52% in performance in execution time, whereas use of log function shows an overall loss of nearly 31% in execution time.

We also tried to compute maximum possible number of consecutive occurrences of underflow and renormalize the interval accordingly in single iteration, but it has not shown considerable significant difference in the performance as compared to processing single underflow at a time.

2. STATISTICAL MODEL

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 ($0 \le i \le n$) is f-bit integer ($f \le b - 2$) representing the lower and upper bounds of cumulative frequency segments. For symbol Si, Fi-1 is lower bound and Fi is upper bound.

Here arithmetic coding algorithm is implemented with order-0 model for 256 symbol alphabet. Statistical model is built using source data before starting compression. The model goes in compressed file for use by decompression algorithm.

3. CONVENTIONAL ALGORITHM

Existing conventional algorithm processes only one bit at a time and renormalize interval in single renormalizing iteration during encoding and decoding. Here a traditional algorithm is given to show how it is enhanced later.

3.1 Encoding Algorithm

- Interval=[0,1)
- Qtr1=range/4, Qtr2=2*Qtr1, Qtr3=3*Qtr1
- cnt=0, a count for occurrences of underflow
- Repeat till not EOF
  - Read symbol
  - Compute corresponding new interval [low, high)
  - Repeat till interval becomes more than half (renormalization loop)
    - Case 1: low and high falls in upper half [0.5,1), i.e. low >= Qtr2. Here matching most significant bit (msb) is 1.
      - output bit 1
      - o/p bit 0 cnt times, cnt=0
      - expand an interval by doubling low and high; i.e. left shift low and high by 1 position. (padding on right: low with 0 and right with 1)
    - Case2: low and high falls in lower half [0, 0.5), i.e. high < Qtr2. Here matching msb 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) and (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) and (high ≥ Qtr3), i.e. interval is more than half; more than 2^{b-2}
      - Break loop
- At EOF
  - cnt++
  - if low < Qtr1; i.e. if its most significant bit is 0, then output bit 0 and cnt times 1. Otherwise output bit 1 and cnt times 0

3.2 Decoding Algorithm

During decoding, it reads and processes b bit code from coded message till end of compressed file. Computation of new interval, extracting bits and renormalizing interval using coded message is performed in exactly the same way as that done during encoding.

4. RENORMALIZING INTERVAL

As explained in section 3, in arithmetic coding, while encoding and decoding each symbol, it processes a single 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 (i.e. 2^{b-2}) of the 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, only one code bit is generated and the current interval is doubled. Case 1 and 2 of algorithm explained in section 3 are usually combined to output most significant matching bit (msb) whether it is 0 or 1.

5. CONVENTIONAL "C" LANGUAGE IMPLEMENTATIONS

Traditional algorithms implemented using 16 bit wide range is considered here. There are some implementations using 32bit wide range, which require 64 bit (long long) integer in some integer multiplications to avoid overflow. This long long data type is not available with many C compilers. Arithmetic coding being lossless compression technique, its traditional implementations use integer arithmetic for accuracy purpose.

Many implementations like E. Bodden [20] compare low and high bound of the interval with quarter (1/4th), half and 3/4th of an interval as explained in algorithm in section III. This may execute slower as compared to implementations [24, 26] using bitwise operations for better performance.

While encoding a symbol, it requires computing new value of low and high bound of an interval. After that, following loop (using bitwise operations [24, 26]) is executed till interval becomes more than half wide. In single iteration of this loop, either matching single bit is processed or one underflow occurrence is processed at a time.

Renormalizing loop [24, 26] to process single bit after computing new value of low and high bound of new interval is given below. Variables low, high are unsigned 16 bit integers; cnt is used to store the value of underflow counts.

```
for ( ; ; )  // renormalizing loop
{
   /* case 1 and 2, msb matching */
   if ( ( high & 0x8000 ) == ( low & 0x8000 ) )
   {
      output_bit( stream, (unsigned short) (high & 0x8000) );
      while ( cnt > 0 ) // cnt: underflow count
      {
        output_bit(stream,(unsigned short) (~high & 0x8000));
        cnt--;
      }
   }
 /* case 3: underflow, msb not matching, 2nd bit differs by 1*/
   else if ( ( low & 0x4000 ) && !( high & 0x4000 ))
   {
      cnt++;
      low &= 0x3fff;
      high |= 0x4000;
   }
  else   // neither case 1,2 nor case 3, return
      break;  // interval becomes more than half wide

/* rescale low, high: rightpad low with 0s and high with 1s */
   low <<= 1;
   high <<= 1;
   high |= 1;    // have 1 as lsb
}  // end for, renormalizing loop
```

6. PROPOSED "MULTI-BIT PROCESSING ARITHMETIC CODING ALGORITHM"

As mentioned earlier, in conventional implementations, only one bit is processed at a time in single iteration. Here it is proposed to extract and output more than one bit and expand the interval accordingly in a single iteration. This reduces the number of iterations used in renormalization. The best part of our proposed implementation is that it does not compromise on compression ratio at all. We have used two alternatives to compute number of matching most significant bits in low and high: (i) using shift operation in a loop and (ii) using log function.

An additional attempt is made to compute number of consecutive occurrences of underflow and expand an interval accordingly. Here interval is expanded by extracting multiple bits from 2nd position onwards in low and high in single loop iteration.

Proposed **multi-bit processing arithmetic coding algorithm** is given here.

6.1 Using Statistical Model

Same as in conventional implementation (as in section 2)

6.2 Renormalizing Interval

Here is the difference between implementation of conventional and our proposed multi-bit arithmetic coding algorithm. Here, renormalization is done by processing multiple bits at a time in single iteration.

Renormalization loop in proposed implementation is as given below: Variable nBits is used to store the value of number of matching most significant bits of low and high; k is used to store the value of number of consecutive occurrences of underflow at a time.

- Repeat till case 1 or 2 or 3 described in section 3 (renormalization loop)
  - Case 1, 2: most significant bits are matching
    - Compute number of most significant matching bits, say nBits
    - output first msb
    - o/p cnt times the complement of msb, cnt=0
    - o/p remaining (nBits-1) msb
    - expand interval by shifting low and high to left by nBits position (padding on right: low with 0 and high with 1)
  - Case 3: an occurrence of underflow; msb is not matching and 2nd bit differ by 1; i.e. low falls in [Qtr1, Qtr2) and high falls in [Qtr2, Qtr3)
    - Compute number of consecutive occurences of underflow, say k; add it to cnt.
    - Extract k bits from 2nd bit onwards from low and high. While doing so, right pad low bound with zeros and high bound with ones.

6.3 Computing number of matching msb

As we know, 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 result in 0 wherever it has matching bits. So to compute how many msb are matching, the only task is to determine occurrences of leading consecutive zeros or finding the position of first occurrence of bit 1 from left. This can be done as shown below.

- tmp=low XOR high

- Determine the number of matching most significant bits in high and low using either log function or shift operation as mentioned here
  - Using shift in a loop: left shift tmp, increment a counter, terminate loop when first bit of tmp is 1. Resulting counter is number of matching most significant bits in high and low.
  - Using log function: Determine 1st occurrence of bit 1 from left in tmp using expression int(log2(tmp)). Assuming low and high are represented using b bits, nBits=b-int(log2(tmp))+1 will be the 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 log2(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. Using constant 0.693147 for log(2) will reduce one function call.

6.4 Computing number of consecutive occurrences of underflow

Underflow occurs when low and high comes closer but it is not detected with most significant bit, i.e. most significant bit is not matching but next bit differs by 1. Thus most significant bit is 0 in low and 1 in high; and next bit is 1 in low and 0 in high. Our interest is to determine number of consecutive 1s in low and 0s in high after most significant bit. Then number of consecutive occurrences of underflow is lowest of these two numbers. Thus number of consecutive occurrences of underflow can be computed as minimum of leading 1s in (low<<1) and leading 0s in (high<<1). Required operations:

- Left shift low and high by one position
- Using left shift in a loop, compute consecutive occurrences of 1s in low. Let it be m.
- Using left shift in a loop, compute consecutive occurrences of 0s in high. Let it be n.
- Number of consecutive occurrences of underflow = k = minimum of m and n.

7. PRACTICAL IMPLEMENTATION OF PROPOSED ALGORITHM USING "C"

As said before, proposed **multi-bit processing arithmetic coding algorithm** is implemented using b=16 bit wide range. So possible maximum number of output bits at a time is 16. C code for computing number of matching most significant bits, consecutive occurrences of underflow and rescaling the interval is as follows: Variables low, high, tmp are all unsigned short (16 bit) integers. Array element mask[i] is assigned a value having all rightmost i bits set to1 and other bits set to 0.

7.1 Computing number of matching most significant bits

```
tmp = low ^ high; // high and low are in [0,0xffff]
if (tmp == 0)
   nBits=16;  // maximum 16 matching bits
else
  {  //using shift loop
        nBits=0;
        while (tmp < 0x8000)  // msb not 0
        {
          nBits++;
         tmp = tmp<<1;  // remove leading zero
        }
   }
```

To compute nBits using log function, replace code in else part with the following code.

```
// using log tmp to base 2
{
   nBits=log(tmp)/0.693147; // natural log 2 = 0.693147
   nBits = 15 - nBits;  // b=16
}
```

7.2 Rescaling interval in single iteration

```
low <<= nBits;  // double nBit times
high <<= nBits; // double nBits times
high |= mask[nBits];   // pad rightmost nBits with 1
```

7.3 Computing consecutive occurrences of underflow

```
//compute underflow count = k
//= min(leading 1s in (low<<1), leading 0s in (high<<1))
m=0; tmp=low<<1;
while (tmp >= 0x8000)
  {    m++;  tmp = tmp<<1;  }
n = 0; tmp = high<<1;
if (tmp==0)
        n=15;
else
  {
        while (tmp < 0x8000)
        {  n++; tmp = tmp<<1;  }
  }
k=m;
if ( n < m) k=n;
```

7.4 Rescaling interval by extracting k bits from 2nd position onwards

```
//starting from 2nd bit, extract k bits from low and high,
//rightpad high with 1s
low = low << k;
low = low & 0x7FFF;  // to have msb 0
high = high << k;
high = high | 0x8000;         // to have msb 1
high = high | mask[k];   // pad rightmost k bits with 1
```

8. Experimental Results

Both the conventional and our proposed multi-bit processing arithmetic coding 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 file types. Some of the test files are selected from act, Calgary and Canterbury corpus, a widely used benchmark. These files are downloaded from website http://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: act2may2002.xls (name shortened to act2may2.xls), calbook2.txt, ca-obj2, cal-pic, every.wav, frymire.tif, kennedy.xls, lena3.tif, monarch.tif, pine.bin, ptt5, world95.txt.

Here term ACEN is used for existing conventional implementation of arithmetic coding for encoding data. In ACEN, single bit is output at a time and single occurrence of underflow is considered at a time. Variations of multi-bit processing are denoted here as ACB1C1, ACB2C1, ACB1C2 where B1, B2, C1, C2 are used to denote the following alternatives used: B1- number of matching most significant bits is computed using shift operation, B2 – number of matching most significant bits is computed using log function,

C1 - single underflow is considered at a time, C2 - more than one consecutive occurrences of underflow is considered at a time.

Table 1 lists files used for testing of both existing and various proposed multi-bit implementations. It is to be noted that compressed file size remains same with all implementations.

Execution time given in table 2 and table 3 is taken as an average of five trials of executing a program for each test file.

TABLE 1. Test Files used

No.	File	File Size (Bytes)	Compressed File Size (Bytes)
1	act2may2.xls	1348036	789951
2	calbook2.txt	610856	367017
3	cal-obj2	246814	194255
4	cal-pic	513216	108508
5	cycle.doc	1483264	891974
6	every.wav	6994092	6716811
7	family1.jpg	198372	197239
8	frymire.tif	3706306	2200585
9	kennedy.xls	1029744	478038
10	lena3.tif	786568	762416
11	linuxfil.ppt	246272	175407
12	monarch.tif	1179784	1105900
13	pine.bin	1566200	1265047
14	ptt5	513216	108508
15	sadvchar.pps	1797632	1771055
16	shriji.jpg	4493896	4481092
17	world95.txt	3005020	1925940

Table 2. Compression (Encoding) Time in Seconds: ACB1C1 and ACB1C2

		ACB1C1	ACB1C2
No.	File name	Seconds	Seconds
1	act2may2.xls	1.2088	1.2088
2	calbook2.txt	0.5495	0.6044
3	cal-obj2	0.2747	0.2747
4	cal-pic	0.3297	0.3297
5	cycle.doc	1.3187	1.3187
6	every.wav	6.9231	7.0879
7	family1.jpg	0.2198	0.2198
8	frymire.tif	3.1868	3.1868
9	kennedy.xls	0.8791	0.8242
10	lena3.tif	0.7692	0.8242
11	linuxfil.ppt	0.2198	0.2198
12	monarch.tif	1.1538	1.1538
13	pine.bin	1.4835	1.4835
14	ptt5	0.3297	0.3297
15	sadvchar.pps	1.7582	1.8132
16	shriji.jpg	4.3956	4.5055
17	world95.txt	2.8571	2.9670
	Total	**27.8571**	**28.3516**

Table 2 presents the compression time (seconds) of multi-bit variations namely ACB1C1 and ACB1C2. As there is no significant difference observed in the execution performance, C2 is not considered in combination with B2 here. Figure 1 shows comparison of execution time of encoding using multi-bit processing variations ACB1C1 and ACB1C2.

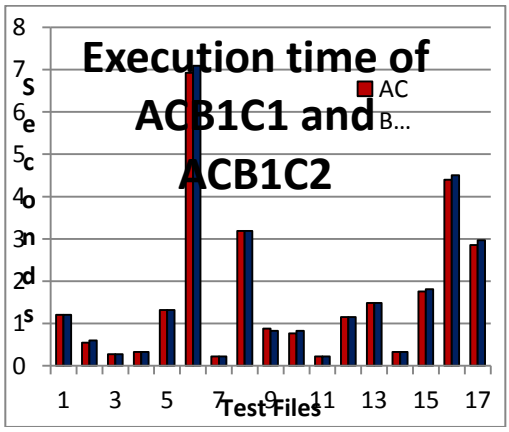

Fig. 1.Encoding time of ACB1C1 and ACB1C2

Table 3 shows the compression time (seconds) of conventional ACEN (single bit processing in single iteration) and multi-bit processing variations ACB1C1 and ACB2C1. Figure 2 represents these data in a graph for better visualization of differences in execution time.

Table 3. Compression (Encoding) Time in Seconds: ACEN, ACB1C1 and ACB2C1

		ACEN	ACB1C1	ACB2C1
No.	File name	Seconds	Seconds	Seconds
1	act2may2.xls	2.307	1.209	3.297
2	calbook2.txt	1.099	0.549	1.538
3	cal-obj2	0.495	0.275	0.604
4	cal-pic	0.659	0.330	0.879
5	cycle.doc	2.637	1.319	3.516
6	every.wav	15.000	6.923	18.791
7	family1.jpg	0.439	0.220	0.549
8	frymire.tif	6.648	3.187	8.791
9	kennedy.xls	1.640	0.879	2.418
10	lena3.tif	1.703	0.769	2.143
11	linuxfil.ppt	0.439	0.220	0.604
12	monarch.tif	2.528	1.154	3.187
13	pine.bin	3.077	1.484	4.066
14	ptt5	0.604	0.330	0.879
15	sadvchar.pps	3.791	1.758	4.835
16	shriji.jpg	9.505	4.396	12.088
17	world95.txt	5.604	2.857	7.802
	Total	**58.176**	**27.857**	**75.989**

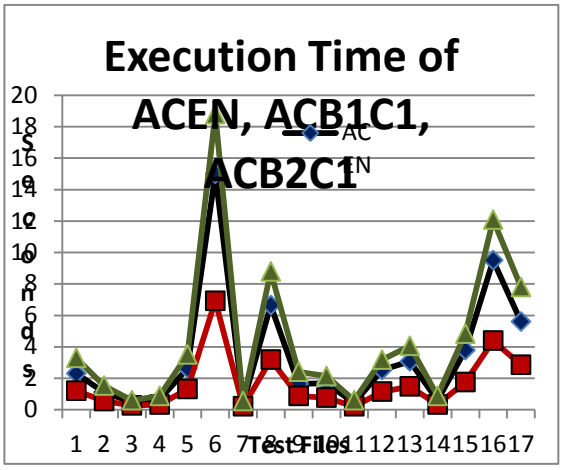

Fig. 2. Encoding time of ACEN, ACB1C1 and ACB2C1

9. RESULT ANALYSIS

From Table 2 and Figure 1, it is observed that there is no significant difference in the performance of algorithms implemented with processing of single underflow at a time and multiple underflow occurrences at a time. This may be due to the calculations required in computing number of consecutive occurrences of underflow. Thus C2 (i.e. computing multiple underflow count and renormalizing interval accordingly at once) has no significant benefit over C1 (i.e. processing single underflow at a time).

It is seen from Table 3 and Figure 2 that B2 (i.e. computing matching number of most significant bits using log function and renormalizing interval accordingly at once) has worst performance. It is not even better as compared to traditional single bit processing at a time (ACEN). Using B1 (computing matching number of most significant bits using shift operation in a loop and then renormalizing an interval accordingly at once) shows very large improvement of about 52% in execution speed.

Execution time saved with the use of shift operations (ACB1C1) as compared to ACEN and ACB2C1 is analyzed in Table 4. Expressions used in computations are shown below:

Percentage of gain of ACB1C1 over ACEN = 100 x (exec. time of ACEN – exec. time of ACB1C1)/exec. time of ACEN

Overall % gain of ACB1C1 over ACEN = 100 x (total exec. time of ACEN – total exec. time of ACB1C1) / total exec. time of ACEN

Overall ratio = total exec. time of ACB1C1 / total exec. time of ACEN

Interpretations**:**

Percentage in gain measures the % of time saved using ACB1C1 over ACEN. Overall ratio gives the fraction of ACEN execution time taken by ACB1C1.

Standard Deviation (SD) and Coefficient of variance (CV) are statistical measures. SD is a square root of mean of squared deviations taken from mean. It measures how dispersed the observations are from their mean. CV = 100*SD/mean. It measures consistency in data set.

Following observations are based on the analysis in table 4:

- Overall execution time taken by ACB1C1 is 47.88% of execution time of conventional ACEN. Thus using ACB1C1, it saves 52.12% execution time as compared to ACEN. It can also be seen that performance gain using ACB1C1 is very consistent in all the test files. It varies in the range of 44.44 to 54.83, has standard deviation 3 and just 6% coefficient of variance.
- Overall execution time taken by ACB1C1 is 36.67% of execution time of ACB2C1. Computing number of matching significant bits using shift in a loop gives overall performance benefit of 63.33% over computations using log function. This performance gain is found to be consistent in a range (54.54, 64.28) with standard deviation 2.25 and a small 3.59% coefficient of variance.
- Overall execution time taken by ACB2C1 is 130.60% of execution time of ACEN. Computing number of matching significant bits using log function shows overall 30.60% loss in execution speed over conventional ACEN. The variation in performance loss is found to be large in various test files. It ranges in (22.222, 46.667) with standard deviation 7.482 and a 22.64% coefficient of variance.

Table 4. Result Analysis

		% gain of ACB1C1 over		% loss with ACB2C1 over ACEN
No.	**File name**	**ACEN**	**ACB2C1**	
1	act2may2.xls	47.619	63.333	42.857
2	calbook2.txt	50.000	64.286	40.000
3	cal-obj2	44.444	54.545	22.222
4	cal-pic	50.000	62.500	33.333
5	cycle.doc	50.000	62.500	33.333
6	every.wav	53.846	63.158	25.275
7	family1.jpg	50.000	60.000	25.000
8	frymire.tif	52.066	63.750	32.231
9	kennedy.xls	46.667	63.636	46.667
10	lena3.tif	54.839	64.103	25.806
11	linuxfil.ppt	50.000	63.636	37.500
12	monarch.tif	54.348	63.793	26.087
13	pine.bin	51.786	63.514	32.143
14	ptt5	45.455	62.500	45.455
15	sadvchar.pps	53.623	63.636	27.536
16	shriji.jpg	53.757	63.636	27.168
17	world95.txt	49.020	63.380	39.216
	overall gain%	**52.125**	**63.341**	**(loss)30.595**
	overall ratio%	**47.875**	**36.659**	**130.595**
	S.D.	**3.047**	**2.248**	**7.482**
	CV	**6.040**	**3.586**	**22.640**
	minimum	**44.444**	**54.545**	**22.222**
	maximum	**54.839**	**64.286**	**46.667**

10. CONCLUSION

Our proposed **multi-bit processing arithmetic coding algorithm** executes faster when number of matching bits is computed using shift operation instead of using log function. As compared to existing conventional implementations of arithmetic coding, it has resulted into a tremendous gain of about 52% in execution speed while encoding without any compromise in compression ratio. Multi-bit processing at

once using log function is found to be very poor as compared to even conventional single-bit at a time processing.

11. 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*

[28] Jyotika Doshi, Savita Gandhi, "Improved Performance Of Arithmetic Coding By Extracting Multiple Bits At A Time", International Journal of Engineering Research & Technology (IJERT) ISSN: 2278-0181, Vol. 1 Issue 8, October – 2012

AUTHOR'S PROFILE

Jyotika Doshi, M.Sc. (Statistics) from M.S.University at Vadodara, Gujarat, India; PGDCA (Computer Science) from Allagappa Unibversity, Tamilnadu, India; MCA (Computer Science) from IGNOU, India; pursuing Ph.D. (Computer Science) from Gujarat University, Ahmedabad, Gujarat, India. She has 3 years of industry experience in software development and about 25 years experience in academics. At present she is working as ASSOCIATE PROFESSOR at GLS Institute of Computer Technology at Ahmebabad, Gujarat, India. Her four research papers are published in international journals.
Ms. Doshi is life member of CSI.

Savita Gandhi (MIEEE' 2003 SMIEEE' 2005) is Professor & Head at the Department of Computer Science, Gujarat University and Joint Director, K.S. School of Business Management, Gujarat University. She is with Gujarat University for about 24 years. Before that she has worked with M.S. University of Baroda, Department of Mathematics for about 10 years. She has been actively associated with IEEE activities at Gujarat Section. She is M.Sc. (Mathematics), Ph.D (Mathematics) and A.A.S.I.(Associate Member of Actuarial Society of India by the virtue of having completed the "A" group examinations comprising six subjects conducted by Institute of Actuaries , London). She was awarded Gold Medal for standing first class first securing 93% marks in M.Sc. and several prizes at M.Sc. as well as B.Sc. Examinations for obtaining highest marks.