The Anatomy of a Quasi-Static Arithmetic Encoder

Sabin Belu, Daniela Coltuc
University Politehnica of Bucharest, Romania
Contact author e-mail: daniela.coltuc@upb.ro

***Abstract*— The paper proposes a new architecture for arithmetic encoders called Quasi-Static. Unlike the classical implementations, the Quasi-Static Encoder buffers the input stream and uses a static model for encoding the data in the buffer. The big advantage of such approach is the higher encoding speed obtained however with the price of slight degraded compression rates. The Quasi-Static Encoder is tested on three types of data - English text, log files and binary files - and compared with two dynamic arithmetic encoders. For 128kB buffers, the Quasi-Static Encoder is about 3.6 times faster than the dynamic counterparts. The loss in compression rate depends on data type and the considered dynamic implementation. The Quasi-Static Encoder outperforms Dippenstein's implementation on English texts and exe files and is below Sachin Garg implementation that gains between 0.08 and 0.20 bpb in compression rate.**

Keywords - data compression; arithmetic encoding; coder architecture.

I. Introduction

Arithmetic data compression, as a de facto entropy encoder, has gone thru an ongoing improvement process, ever since its first appearance in the famous article "Arithmetic Coding for Data Compression" written by Ian H. Witten, Radford M. Neal and John C. Cleary. As an entropy coder, the algorithm collects symbols statistics about to be encode, before or on the fly, and represents them into a more compact form. Following an entropy coder principle the most often characters will be encoded into fewer bits and some not so often symbols will be encoded into more bits then their original encoding (e.g. ASCII American Standard Code for Information Interchange, 8 bpb). One of the utmost difference of an Arithmetic Encoder with respect to most of the entropy coders, such as Huffman (dynamic, also known as Gallager Entropy Coder or static variants), Splay encoding, Shannon–Fano etc. is that it encodes the entire message (buffer or file) into a single big number, that is carefully chosen to be a fraction q within the initial interval [0.0, 1.0).

In this article, we will further discuss and analyse some algorithm limitations, strengths and weaknesses and an interesting practical approach is proposed. We are proposing a different approach to Arithmetic Encoding, a Quasi-Static Approach that goes beyond many of the drawbacks in the Arithmetic Encoding, one of the main important aspect being the speed of execution. The Quasi Static approach is superior in terms of speed to a factor of 3 to 5 times while stating very few percentages below the compression ratio of a full Arithmetic Encoder. We give results obtained with an experimental model of Quasi-Static Arithmetic Encoder and note the results against other full Arithmetic Encoders.

The rest of the paper is organised as follows: Section II presents the Arithmetic Encoding and Decoding processes along with some practical considerations concerning the implementation, Section III details the architecture of the Quasi-Static Encoder and exposes a series of theoretical considerations regarding the compression rate and the execution time, Section IV gives compression results on three types of files and, finally, in Section V some conclusions are drawn.

II. Arithmetic Encoding

A message to be encoded is composed of symbols from a given alphabet. As stated above, arithmetic encoder takes the message and converts it into a fractional number q with the condition that q belongs to [0.0, 1.0).The fractional number cannot be output in the end all at once simply because of the practical impossibility of storing such a big fractional number in a software program. Using a 32 or 64 bit Single/Double Precision Floating Point Numbers (IEEE Standard 754 for Floating Numbers) will not be enough to hold the bits required for encoding a certain message, let alone big files. Variable data sizes used in any programming language are bound to data types, and tipically do not go beyond 128 bits. Since Arithmetic Encoding process eventually outputs one single number that theoretically could have the size of the message that is encoded, keeping the output stream of bits in a single or more data variables is not a feasible solution for practical implementation. We will resume later on the way the output is realised and means to make it more feasible for practical implementation.

A. The statistical model of symbols

In this section we explain what is a statistical model and which is its role in the Arithmetic Encoding. The main function of the Arithmetic Encoding model is to provide probabilities for symbols to be encoded, and also, in case of an adaptive algorithm, to further update these probabilities.

The actors of the encoding process are:

1) The symbols to encode e.g., A, B, C.
2) The alphabet defined as the body of symbols to encode e.g., ASCII symbols.
3) The message to encode, which is a collection of symbols over the alphabet e.g., $AAAAABBBCC$.
4) The message length N.
5) Stastistical probabilities of symbols.

TABLE I: Statistical model of sequence $AAAAABBBCC$.

Symbol	Frequency	Probability	Range in [0, 1]
A	5	0.5	[0.0, 0.5)
B	3	0.3	[0.5, 0.8)
C	2	0.2	[0.8, 1.0)

6) Cumulated probabilities of symbols.
7) Current interval with respect to initial [0.0, 1.0) interval.

In a static or quasi-static model, the encoding process is performed in a two-step stage: a first pass when all probabilities are updated/calculated and the second pass when the symbols are encoded one by one without changing adaptively the probabilities.

Our approach is a quasi–static encoding meaning that the probabilities are estimated in an earlier stage, prior to encoding the symbols. Consider the following example of message: $AAAAABBBCC$. It is a message of length 10 consisting in three different symbols from ASCII alphabet. We derive the statistical model i.e., the symbol probabilities by simply counting the symbols in the message. We note the probability of A with $p(A)$ and define it as being number of occurrences of A in a message divided by message length $L = 10$. The symbols are shown in Table I alongside their frequency, probability and occupied range in the interval $[0, 1)$.

B. Arithmetic Encoding Process

Given the interval [0.0, 1.0) the distribution of probabilities can be vizualized such as each symbol "occupies" a range directly proportional to its own probability. Thus, A occupies $[0, 0.5)$, B occupies $[0.5, 0.8)$ and C occupies the rest of $[0.8, 1)$.

At each step the encoder takes care of:

1) The next symbol to encode;
2) The symbol probability as defined by the statistical model;
3) The current interval, which initially is $[0.0, 1.0)$.

In our example, the first symbol to be encoded is A, which has the probability $p = 0.5$. The current interval is the initial interval $[0.0, 1.0)$. Afterwards, the current interval becomes [0.0, 0.5) with the endpoints calculated as follows:

$$LOW_1 = 0.0$$
$$HIGH_1 = 0.0 + 0.5 \times (1.0 - 0.0) = 0.5$$

The second symbol to be encoded is also A with the same probabilty $p = 0.5$. The current interval now becomes $[0.0, 0.25)$ with the endpoints:

$$LOW_2 = 0.0$$
$$HIGH_2 = 0.0 + 0.5 \times (0.5 - 0.0) = 0.25$$

When the n-th succesive A is processed, the LOW_{n} and $HIGH_{\mathrm{n}}$ endpoints of the current interval are:

$$\begin{aligned} LOW_n &= LOW_{n-1} \\ HIGH_n &= LOW_{n-1} + p \times (HIGH_{n-1} - LOW_{n-1}) \end{aligned} \tag{1}$$

where p is the symbol probability as it was estimated in the beginning of encoding.

The current interval develops and changes with each new symbol. Figure 1 depicts the current interval development during all A symbols in the beginning of the message. As seen, with each new symbol, only the right endpoint changes. It evolves from 1 to 0.0625.

When the algorithm starts encoding B series, both LOW and $HIGH$ endpoints change, since B is not anymore the first symbol in the alphabet. The current interval where B is to be encoded is $[0, 0.03125)$. After B is encoded, the current interval becomes:

$$LOW_6 = 0.01562250$$
$$HIGH_6 = 0.01562250 + 0.3 \times (0.03125 - 0.0) = 0.025$$

The interval development during the encoding of $BBBB$ series is depicted in Fig. 2. The current interval endpoints are calculated as follows:

$$LOW_n = HIGH_n[S_{i-1}] \quad i > 1 \tag{2}$$
$$HIGH_n = HIGH_n[S_{i-1}] + p \times (HIGH_{n-1} - LOW_{n-1})$$

where $HIGH_n[S_{i-1}]$ is the left endpoint of the preceding symbol in the alphabet. Here, the index i signifies the position of the symbol in the alphabet. The process proceeds similarly for C symbols (Fig. 3). With each new symbol, the endpoints of the assigned interval are calculed with eq. 2. After the last C in the message, the current interval becomes $[0.0225288, 0.0225625)$. The width of interval is the mere probability of the message, estimated by supposing that the symbols are statistically independent and their probabilities do not change along the message. The beauty of the algorithm consists in the fact that we could choose a number $\hat{c}$ from this interval to encode the sequence. This number converted into binary is the message codeword. As for any entropic method, the length of the codeword depends on the message probability i.e., $HIGH_n - LOW_n$:

$$L_n = \lceil log_2 \frac{1}{HIGH_n - LOW_n} \rceil \tag{3}$$

The representation on a limited number of bit shifts the chosen number to the left. A choice that guarantees that $\hat{c}$ remains inside the interval is:

$$\hat{c} = 2^{-L_n} \lfloor 2^{L_n} LOW_n + 1 \rfloor > LOW_n \tag{4}$$

In our example, $L_n = 15$, $\hat{c} = 0.02255249023$ and the codeword is 000001011100010. Due to the positioning of $\hat{c}$ the decoder will be able to perform the inverse steps correctly in order to find out the message.

C. Output streaming

In practical implementations the codeword is built progressively. For each new encoded symbol, the encoder can release none, one or more bits of the codeword. This process is based on the following observation: within the interval [0.0, 1.0) any number that lies below 0.5 has 0 as MSB and any number that lies above 0.5, has the MSB 1. When LOW and $HIGH$

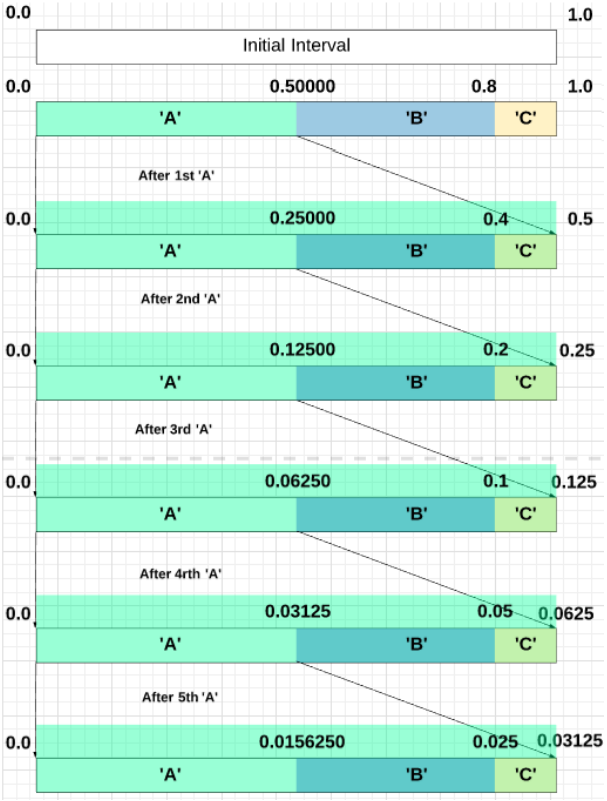

Fig. 1: Current interval development for the sequence $AAAAA$.

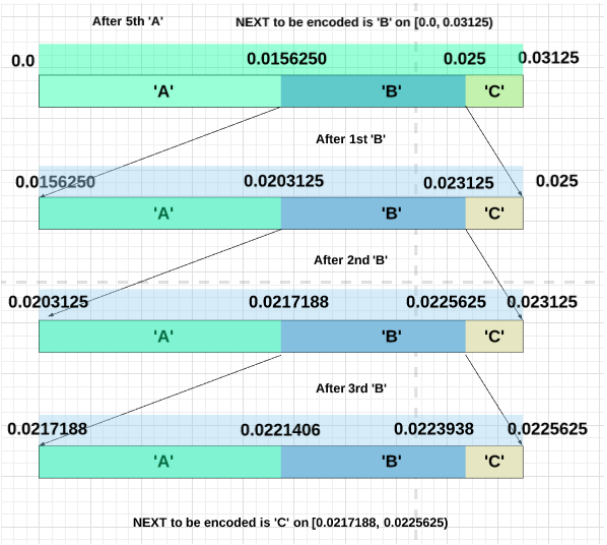

Fig. 2: Current interval development for the sequence BBB.

endpoints lie on the same side of 0.5, $\hat{c}$ will certainly have the MSB of that side. Consequently, a bit is released and in order to reproduce this condition, the binary representations of LOW and $HIGH$ are left shifted. This is equivalent with a scaling of the interval by 2. The encoding process is described

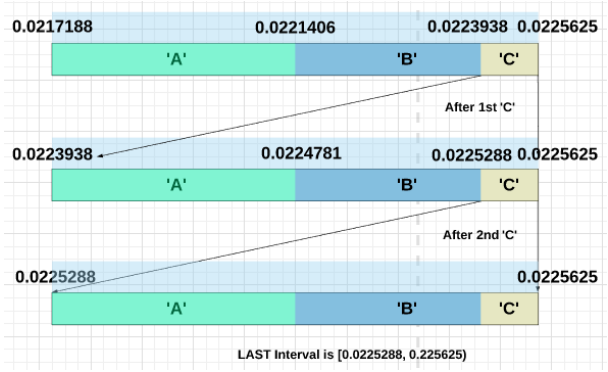

Fig. 3: Current interval development for the sequence CC.

by the following pseudo code:

```
WHILE INPUT symbol c
CALCULATE NEW Low and HIGH for c
REPEAT
IF (high < Half) SEND BIT0 TO OUTPUT
ELSEIF (low >= Half) SEND BIT1 TO OUTPUT
UPDATE LOW AND HIGH
low = low « 1;
high = (high « 1) + 1;
END REPEAT
END WHILE
```

At each step the encoder checks the position of LOW and $HIGH$ by respect to 0.5. Whenever $HIGH$ goes below 0.5, it means that LOW and $HIGH$ have the same leading bit 0, thus the encoder outputs a 0. Whenever LOW goes above 0.5, LOW and $HIGH$ have the same leading bit 1, thus the encoder outputs an 1.

D. The Decoding Process

At the decoder, the bitstream is cut into codewords and for each codeword is identified the interval that originated it. The decoder must be aware of the statistical model otherwise the decoding is not possible. After the decimal conversion of the codeword, the decoder deals with a fractional number. Suppose that it is 0.02255249023. At the first step, the number is compared with the partition of the initial interval $[0, 1)$. (see Fig. 4). Since the number is lower that 0.5, the decision is that the fist symbol of the message is A. At the next step, the interval is rescaled at $[0, 0.5)]$ obtained as follows:

$$LOW_1 = 0.0$$
$$HIGH_1 = 0.0 + 0.5 \times (1.0 - 0.0) = 0.5$$

and the number is compared with its partition. Since the number is lower than 0.25, the second decoded symbol will be also A. At the n-th step, the interval is scaled as in eq. 1 or 2 and the number is compared with the corresponding partition. The algorithm stops when the rescaled interval becomes narrower than 2^{-L_n}, a sign that the message is completely decoded. The interval rescaling during the decoding of the first A symbols is depicted in Fig. 4.

The interval shrinks after each decoded symbols. It is noticeable how the number belong to each and everyone of

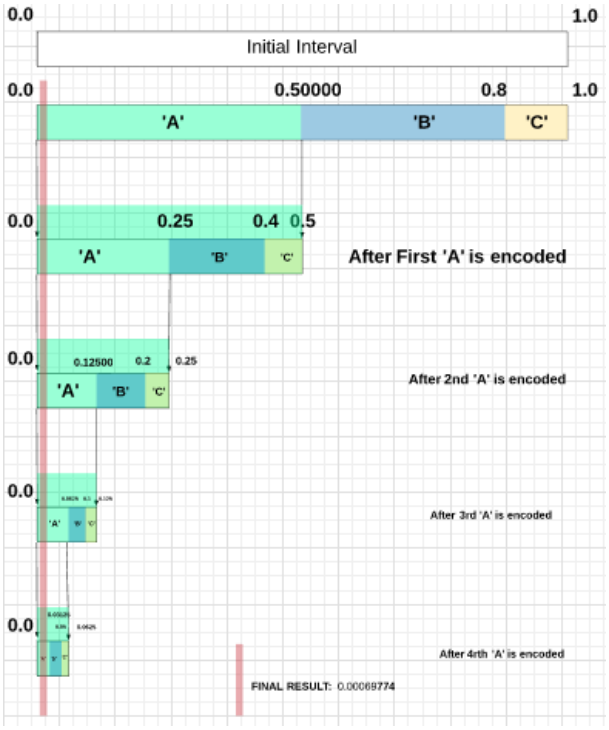

Fig. 4: Decoding of the first 4 symbols of the message.

the succeeding intervals: $[0.0, 0.5)$, $[0.0, 0.25)$, $[0.0, 0.125)$, $[0.0, 0.0625)$, etc.

E. Practical Consideration

Considering the interval [0.0, 1.0) and the fast way it shrinks, it looks completely impractical to use this interval for an implementation of arithmetic coding. Using [0.0, 1.0) gives out two major defects when implementing arithmetic coding. The first is the speed of floating point operations, which is much slower than working with integer values. The second is the precision of the algorithm which is limited to the standard used for representing such numbers in single or double precision floating point number. The precision specified in the IEEE Standards 754 ensures a single precision on 32 and 64bit. For instance, with a x86-64 Intel Ivy Bridge, floating point Arithmetic speed on 80 and 128 bit precision floating point is usually below 100 milion ops, as compared to 220 milion ops at 64bit. As we need more floating point precision, the speed drastically decreases; therefore, a fully-fledged floating point implementation is not taken into consideration when implementing arithmetic coding.

One idea generally used for implementation the arithmetic coding is to scale the interval towards a chosen precision, for instance 32 or 64 bit integers. What we need is:

- Range of scaled interval defined as $HIGH_s - LOW_s$,
- Frequencies of symbols,
- Cumulated frequencies of symbols,
- Value per Unit (VpU).

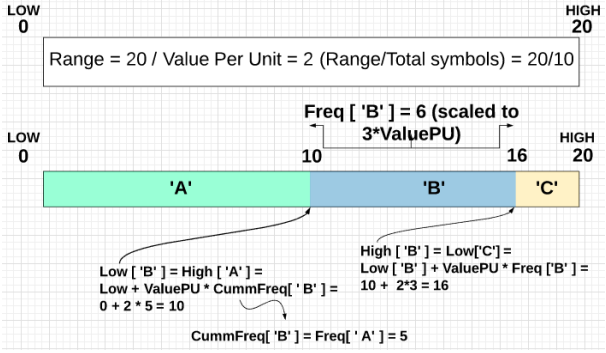

Fig. 5: The partition of the interval scaled at $[0, 20)$.

where LOW_s and $HIGH_s$ are the interval endpoints, Cumulated frequencies of symbols are calculated as the sum of previous symbols frequencies and Value per Unit defines the ratio between the range and the sum of frequencies.

Suppose that for our example, we want to rescale the interval $[0, 1)$ to $[0, 20)$. The symbols frequencies and cumulated frequencies are given in the following table:

TABLE II: Frequencies and cumulated frequencies of A, B and C.

Symbol	Freq.	Cumulated freq.	Range in [0, 10)
A	5	0	[0.0, 0.5)
B	3	5	[0.5, 0.8)
C	2	8	[0.8, 1.0)
Sum of freq.		10	

The cumulated frequencies are helping us to define the interval assigned to each symbol. The endpoints of the interval assigned to symbol S_i are calculated as follows:

$$\begin{aligned} LOW[S_i] &= LOW_0 + VpU \times CumFreq[S_{i-1}] \\ HIGH[S_i] &= LOW_i + VpU \times CumFreq[S_i] \end{aligned} \tag{5}$$

where VpU is in our case $20/10$ and LOW_0 is the left endpoint of the initial scaled interval. Here the index i specifies the symbols order in the interval. Without a good mapping, meaning without the use of VpR, we would have a change of probabilities from the initial estimation.

The partition after the interval scaling is shown in Fig. 5. Following the eq. 5, the fractionary endpoints were mapped to integers:

- for symbol A, the interval $[0, 0.5)$ is mapped to $[0, 10)$,
- for symbol B, the interval $[0.5, 0.8)$ is mapped to $[10, 16)$,
- for symbol C, the interval $[0.8, 1)$ is mapped to $[16, 20)$.

When dealing with full 32 bit or 64 bit the upper limit of the rescaled interval becomes 4,294,967,295 or 0xFFFFFFFFU in base 16. The result is much larger intervals to work with:

- $[0\text{x}00000000, 0\text{x}80000000)$ for A
- $[0\text{x}80000000, 0\text{x}\texttt{CCCCCCCC})$ for B
- $[0\text{x}\texttt{CCCCCCCC}, 0\text{x}\texttt{FFFFFFFF})$ for C

III. Quasistatic arithmetic encoder

It is a known fact that arithmetic coding tends to optimality as long as the source model probabilities are equal to the probabilities of the symbols to be encoded. Any difference reduces the compression ratio. In classic implementations the arithmetic encoder uses a dynamic model that adapts its statistics with each input symbol. All statistics are computed based on the previously encountered symbols. Since all previous symbols are already known at the n-th encoding step, these statistics will be updated into the same way by the decoder as well; no overhead for statistics in needed. The drawback is its complexity. At each iteration, the following steps need to be performed:

- frequency of symbol S_i is updated to match the encountered symbol,
- all frequencies are rescaled if frequency of $S_i \geq MAX\_FREQUENCY$
- all cummulated frequencies from S_{i+1} need to be recalculated.

At opposite pole, the Static Model gathers statistics once for the entire input stream. It is fast but the compression might be very poor because of the non-stationary nature of most input streams.

We propose a quasistatic model, which is a halfway solution between the dynamic and static model. The idea is to buffer the input stream and for encoding the buffer to use a static model. The statistics are gathered prior to encoding and during encoding no symbol frequency is updated, since all symbols have already been counted throughout the entire buffer. These statistics are not known to the decoder, therefore, the frequency table from which all probabilities are recalculated during decoding process will be extracted from the output buffer. Therefore, the only parameter worth mentioning to the decoder as well, is the buffer size. The frequency table is stored in the output buffer, or file, as needed. It has a variable size, and the stored arrangement is:

- 1st byte: Symbol
- 2nd byte and 3rd byte: Symbols' 16bit frequency

The above arrangement has a major advantage over writing all 256 possible frequencies. It accommodates according to the encoded alphabet, which could consist in fewer symbols, thus minimising the size of the stored data; second, it is large enough for 1-symbol.

The QuasiStatic model has the advantage of being a very flexible solution: by adjusting the buffer size the tradeoff between the encoding speed and the compression performance can be tuned in the favour of one or another depending on application requirements.

The block scheme of the QuasiStatic encoder is depicted in Fig. 6: the Buffering Module ensures the input stream is processed in chunks of size N, the Statistical Module performs the frequency count and updates the frequency and cumulative frequency tables and finally, the Statistical Modelling Module may perform "changes" on probabilities gathered by the second module. More precisely, if necessary, it adjusts the frequencies below MAX_FREQUENCY.

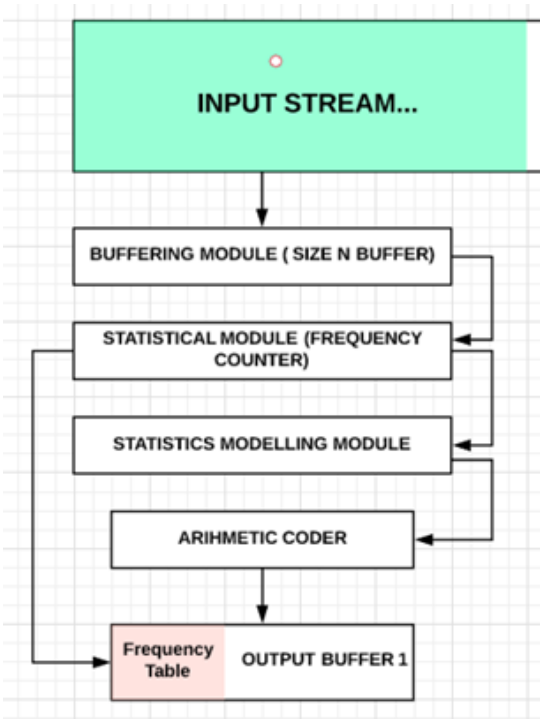

Fig. 6: The Quasi-Static Arithmetic Encoder.

IV. Experimental Results

The QuasiStatic Encoder was tested on three types of data. The first test series is from Calgary Corpus, a well known collection of English text and binary data files, commonly used in data compression tests since the 90s [8]. The size of the first test series is 6,285,846 bytes. The second series is a formatted text file that represent a Perforce log for a syncronization operation. The file is still in English-language, but formatted with insertion of file names, operation date and time, file sizes, file paths etc. The file size is 11,924,676 bytes. The third test series is binary code and consists of a collection of 32bit/64bit windows PE/PE32+ executable files. The size of this series is 21,742,646 bytes.

In a first step, we analysed the impact of the buffer size on the compression rate (Fig. 7). The larger the buffer becomes the lower the 'stored' frequency table size is as compared to the final output size. Thus, compression savings tend to overcome to output. This explains the best compression results for 64 and 128kB buffer sizes.

The downside of using an unique frequency table for the entire buffer is a possible decay of the compression rate because of static model mismatch with the instant symbol probabilities. An input that is highly non-stationary will be in places poorly compressed. To evaluate this effect, we compared our Quasi-Static Coder with two dynamically adapting encoders, the former written by Diperstein [7] and the latter by Sachin Garg [6]. The decoder files have been downloaded

TABLE III: Comparison with two Dynamic Encoders

	Calgary Corpus compr.(bpb)	Exec. time (ms)	xTime	Perforce log compr.(bpb)	Exec. time (ms)	xTime	Exe files compr.(bpb)	Exec. time (ms)	xTime
Original	8.00			8.00			8.00		
Dippenstein	5.37	1,155	3.52	5.45	2,839	4.33	6.23	5,023	3.50
Quasi-Static	4.53	328		5.47	655		5.63	1,435	
Sachin Garg	4.41	1,342	4.09	5.39	2,262	3.45	5.43	4,430	3.08

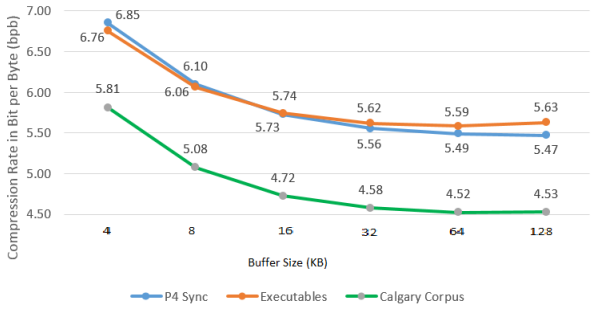

Fig. 7: Compression rates obtained by Quasi-Static Encoder and various buffer sizes. Tests on 3 types of data: Calgary Corp., Perforce log, exe files.

and recompiled in 64b Windows executable format. This is the same executable format as our Quasistatic Encoder binary. The compiler used was Microsoft (R) C/C++ Optimized Compiler Version 19.12 Build 25835 for x64. Table III shows the compression rates obtained for the three test series with the QuasiStatic encoder (128kB buffer) and the two dynamic encoders. Sachin Garg Encoder compresses better in al the cases but the implementation of Dippenstein is acctually worse then our Quasi-Static Encoder. For Calgary Corpus and exe files it loses 0.84 bpb and 0.60 bpb, respectively, and gains only 0.02 bpb in compressing Perforce log file.

We analysed also the execution time (Fig. 8). The values represent average execution times obtained by running 50 times the encoder. The execution time varies with the buffer

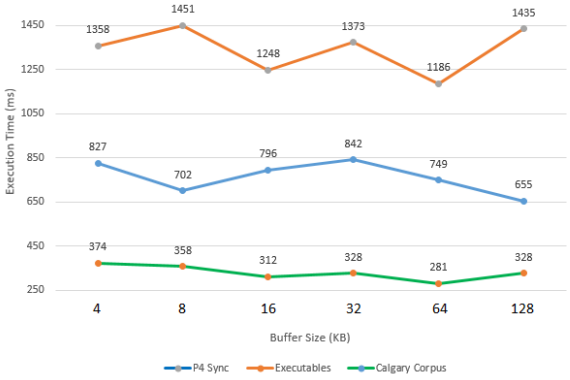

Fig. 8: Encoding speed of Quasi-Static encoder and 128kB buffer size. Tests on 3 types of data: Calgary Corp., Perforce log, exe files.

size, we could not identify a common trend for the three types of test files. To adjust the test sizes according to the three series, the Calgary Corpus has been doubled in size.

The encoding speed is the main advantage of the Quasi-Static Encoder over the dynamic counterparts (Table III). Comparing with Sachin Garg and Dippenstein, for 128kB buffers, the Quasi-Static Encoder is about 3.6 times faster. The loss in compression rate depends on data type and the considered dynamic implementation. The Quasi-Static Encoder outperforms Dippenstein's implementation on English texts and exe files and is below Sachin Garg implementation that gains between 0.08 and 0.20 bpb in compression rate. It is worth to note the execution time versus compression rate performances in the case of English texts where the Quasi-Static Encoder is 3.52 times faster and gains 0.84 bpb comparing with Dippenstein implementation and is 4.09 times faster and loses only 0.08 bpb comparing with Sachin Garg implementation.

V. Conclusions

The paper presents details for construction of a Quasi-Static Arithmetic Encoder, along with results of compression rate and speed over three test series.

We believe that this approach to data compression is a viable solution when an entropy coder is preferred or speed versus compression rate is a big factor. A drastic improvement on speed with a low impact on compression rate compared to two dynamic Arithmetic Encoders has been observed during tests, thus, making this approach a viable solution to any dynamic entropy coder, especially when arithmetic encoding is preferred.

References

[1] J. H. Witten, R. M. Neal and J. G. Cleary, "ARITHMETIC CODING FOR DATA COMPRESSION", *Communication of the ACM (1987)*, vol. 30, no. 6, pp. 520-540, 1987.

[2] M. Nelson, "Data Compression With Arithmetic Coding", *Dr. Dobbs Journal*, November 04, 2014.

[3] P. G. Howard and J. S. Vitter, "Practical Implementation of Arithmetic Coding',' *Kluwer Academic Publishers*, Norwell, MA, pp. 85-112, 1992.

[4] A. Mofat and R. N. Neal, "Aritmetic Coding Revisited',' *ACM Transactions on Information Theory*, vol. 16, no. 3, 1998.

[5] C. E. Shannon, "A Mathematical Theory of Communication", *Dell Syst. Tech J.* 27 July 1948, pp. 398-403.

[6] Sachin Garg, "32/64-BIT RANGE CODING AND RITHMETIC CODING", $https://sachingarg.com/compression/entropy_coding/$.

[7] M. Dipperstein, "Arithmetic Code Discussion and Implementation", http://michael.dipperstein.com/arithmetic/index.html, 23rd November, 2014.

[8] I. Witten, T. Bell and John Cleary, "MODELING FOR TEXT COMPRESSION ", *ACM Computing Surveys*, University of Calgary, CA, vol. 21, no. 4, pp. 557-591, December 1989.