Comparative Analysis of Arithmetic Coding Computational Complexity

Amir Said
Imaging Systems Laboratory
HP Laboratories Palo Alto
HPL-2004-75
April 21, 2004*

E-mail: said@hpl.hp.com, said@ieee.org

entropy coding, compression, complexity

Some long-held assumptions about the most demanding computations for arithmetic coding are now obsolete due to new hardware. For instance, it is not advantageous to replace multiplication--which now can be done with high precision in a single CPU clock cycle--with comparisons and table-based approximations. A good understanding of the cost of the arithmetic coding computations is needed to design efficient implementations for the current and future processors. In this work we profile these computations by comparing the running times of many implementations, trying to change at most one part at a time, and avoiding small effects being masked by much larger ones. For instance, we test arithmetic operations ranging from 16-bit integers to 48-bit floating point; and renormalization outputs from single bit to 16 bits. To evaluate the complexity of adaptive coding we compare static models and different adaptation strategies. We observe that significant speed gains are possible if we do not insist on updating the code immediately after each coded symbol. The results show that the fastest techniques are those that effectively use the CPU's hardware: full-precision arithmetic, byte output, table look-up decoding, and periodic updating. The comparison with other well known methods shows that the version that incorporates all these accelerations is substantially faster, and that its speed is approaching Huffman coding.

* Internal Accession Date Only
Results of this paper were presented as a poster at IEEE Data Compression Conference, 23-25 March 2004, Snowbird, Utah, USA

Approved for External Publication

Comparative Analysis of Arithmetic Coding Computational Complexity

Amir Said*
Hewlett Packard Laboratories
1501 Page Mill Rd., Palo Alto, CA

Abstract

Some long-held assumptions about the most demanding computations for arithmetic coding are now obsolete due to new hardware. For instance, it is not advantageous to replace multiplication—which now can be done with high precision in a single CPU clock cycle—with comparisons and table-based approximations. A good understanding of the cost of the arithmetic coding computations is needed to design efficient implementations for the current and future processors. In this work we profile these computations by comparing the running times of many implementations, trying to change at most one part at a time, and avoiding small effects being masked by much larger ones. For instance, we test arithmetic operations ranging from 16-bit integers to 48-bit floating point; and renormalization outputs from single bit to 16 bits. To evaluate the complexity of adaptive coding we compare static models and different adaptation strategies. We observe that significant speed gains are possible if we do not insist on updating the code immediately after each coded symbol. The results show that the fastest techniques are those that effectively use the CPU's hardware: full-precision arithmetic, byte output, table look-up decoding, and periodic updating. The comparison with other well known methods shows that the version that incorporates all these accelerations is substantially faster, and that its speed is approaching Huffman coding.

1 Introduction

Arithmetic coding is well known for its optimality, and the fact that it can be a very versatile and powerful tool for coding complex data sources [1, 2, 4, 6, 10]. At the same time, practitioners also know that it had not been more commonly used because of its high computational complexity. If we consider the many years of research on techniques for reducing its complexity, it may seem that there is little hope for new breakthroughs and for its widespread

*E-mail: said@hpl.hp.com, said@ieee.org.

adoption. However, new results show that, in fact, the evolution of arithmetic coding is following an unusual path, and the most promising alternative is to move back to the simplest implementations.

This happens because most of the cost-reduction techniques for arithmetic coding were developed for the hardware that was available 10 or more years ago, when multiplications and divisions were too slow for coding purposes. Currently even inexpensive processors can perform precise arithmetic very fast. Since arithmetic is such a fundamental task of any processor, we can expect even greater advantages in the future. Thus, there is a need to identify what are the arithmetic coding tasks that will remain truly important in determining the computational complexity. Using this information we should be able to find out how to better exploit the processor's arithmetic capabilities for faster coding.

The sources of arithmetic coding computational complexity include [1, 2]:

- Interval update and arithmetic operations;
- Symbol decoding (interval search);
- Interval renormalization;
- Carry propagation and bit moves;
- Probability estimation (source modeling);
- Support for non-binary symbol alphabets (binary coders only).

Because all these actions can be tightly integrated in a single implementation, it has been difficult to clearly identify the performance bottlenecks. In this work we tackle this problem by doing an extensive comparative evaluation. Our strategy is to measure the performance of several implementations, changing one parameter at time, or at least as few as possible. This way we can evaluate the importance of each of the tasks mentioned above, and also select the best techniques.

While the main objective of this paper is to evaluate the difference in complexity of a variety of tasks and techniques, in some graphs we add results from some well-known implementations, because they can provide an absolute reference (we explain when the comparisons are not fair). To avoid redundancy, in this document we do not present all the details of our implementation and analysis, since most of it can be found in references [1, 2] (which also provides a much more complete presentation, and set of references on arithmetic coding theory and practice). However, the reader should be aware that there is a considerable amount of programming for each experiment: we had to write specific programs to test all the important tasks, in more than 10 different full implementations of arithmetic coding.

Even though some of the techniques for complexity reduction we present here are not new, we believe this is the first time that their use for arithmetic coding is reported in this form of complexity comparisons. In addition, we believe that the sequential separation of the analysis of the different tasks, with the identification of the most suited implementation to be used in other tests enabled a much better understanding of the complexity issues.

For instance, we verify that in current processors the bit-based renormalization is much slower than byte-based renormalization [5]. In practice, this means that no meaningful analysis of other less important sources of complexity can be done using bit-based renormalization, since its effects are so large as to mask everything else. After replacing it with byte-based renormalization we found that we could immediately identify the next most important source of complexity, and clearly demonstrate the advantages of using specific techniques. In fact, this masking effect is another important motivation for this work. There are some previous analyses of arithmetic coding complexity that may be obsolete not only due to new processors. For instance, because the tests done by Moffat *et al.* [6, 8] used bit-based renormalization, we consider that they needed to be revisited and the conclusions updated.

Our experimental results confirm our expectation that the techniques that yield the best results are those that can fully exploit the processor's hardware capabilities. In fact, the idea of replacing one complex task with several simple ones is shown to be counterproductive in several cases. This clearly demonstrates, for instance, the strong limitations on throughput of arithmetic coding techniques that work only on binary symbols.

In addition, we show that we can have substantial speed gains for the renormalization, decoder symbol search, and code adaptation tasks. It was found that our fastest adaptive version has speeds on larger alphabets that are approaching Huffman coding, with the advantage of being easier to implement, and being much more versatile.

This document is organized in the following way. In Section 2 we explain how the experiments were performed, i.e., type of computer, compiler, speed measurements, etc. In Sections 3 to 6 we analyze the importance of renormalization, multiplications, and divisions, and also how the efficiency of coding can be measured by the information throughput. In Sections 7 and 8 we compare static and adaptive coding, and different adaptation strategies. Section 9 contains a comparison between our fastest version, Huffman coding, and other well-known implementations. Section 10 contains a summary of all the conclusions reached in each section, and discusses our evaluation of the best techniques and the current competitiveness of arithmetic coding.

2 Experimental Settings

All our experiments were performed on a computer with a 2 GHz Intel Pentium 4 processor, and Windows XP Professional operating system. The C++ code was compiled with Microsoft's VC++ 6.0 settings for maximum execution speed. We do not use macros, i.e., the encoding and decoding is done by function calls, but we allow the compiler to use inline functions for internal arithmetic coding tasks (e.g., renormalization, carry propagation, etc.).

The original and compressed data was moved to and from memory buffers (i.e., no disk IO). We report the average encoding and decoding times of 250 runs of the following four stage simulation sequence.

1. A first memory buffer is filled with 10^6 pseudo-random data symbols.

2. The time to encode all data symbols to a second memory buffer is measured.

3. The time to decode all data symbols to a third memory buffer is measured.

4. All decoded symbols are compared with the original to guarantee that the code is correct.

We used a simulated random data source with M symbols and truncated geometric probability distribution

$$\text{Prob(s)} = \frac{(1-\rho)\,\rho^s}{1-\rho^M}, \quad s = 0, 1, \ldots, M-1. \tag{1}$$

Values of ρ were computed to obtain the desired source entropy. The pseudo-random data symbols were generated using a pseudo-random number generator with period 2^{88} suited to this type of long simulations [7]. All the tests started with the same generator seed, which means that comparisons are done for exactly the same set of pseudo-random symbols. In addition, the order of the data symbols was randomly shuffled in each simulation run to avoid any bias that could occur due to the fact that the probability distribution is monotonic.

We do not report actual compression ratios for our simulations because we use predefined source entropy values, and the difference between them and the code rates is in all cases below the statistical uncertainty of the simulations. Every reported result corresponds to a simulation in which we had correct decoding for all symbols.

3 Renormalization and Carry Propagation

The first implementations of arithmetic coding had to deal with strong limitations in both arithmetic precision and memory. To save bits of precision, the arithmetic coding interval needs to be rescaled (renormalized) as soon as single bits are ready to be saved to a file or bit buffer [10]. For example, in a class of encoders (Q, QM, and MQ [3]) multiplications are avoided only because the interval length is always kept within a very limited range. This approach does not work well with the current processors, which are much more efficient when handling data streams organized in groups of 4, 8 or 16 bytes. The changes required in the arithmetic coding process to put out whole bytes in each renormalization are straightforward [1, 2], but it is necessary to have enough precision in the arithmetic operations to accommodate the much wider range of interval sizes.

Figure 1 shows how the encoder times change for different renormalizations, done when one or two bits are ready, or one or two bytes. Note that the difference is not large as we double the number of bits or bytes, but very significant when we change from bits to bytes. Figure 1 also shows why we prefer to use graphs instead of tables to report results. The slope in the curves shows that the bit-based renormalization is quite costly since the encoding times increases significantly with the entropy, i.e., with the number of times renormalization is performed. We conclude that the byte-based renormalizations, on the other hand, are so

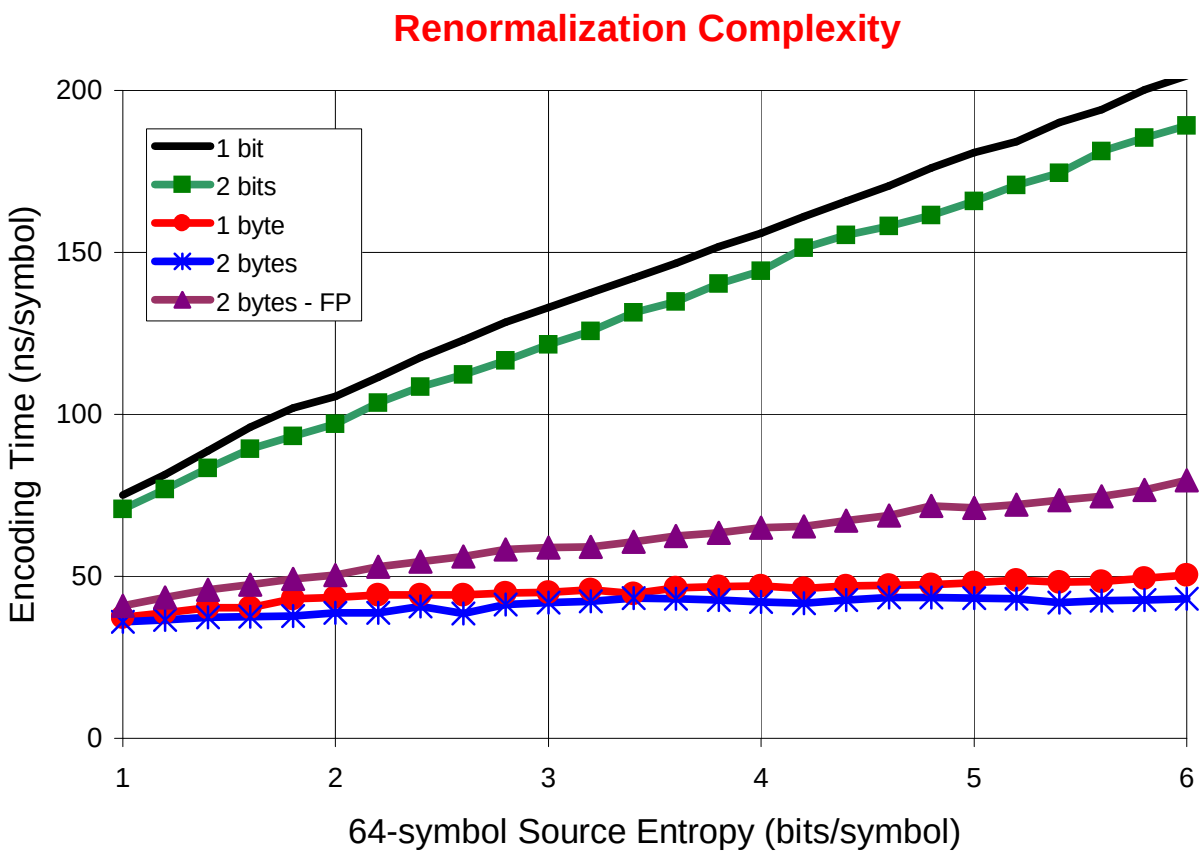

Figure 1: Encoding speed improvement by changing the number of bits extracted at each interval rescaling (renormalization).

much faster that the slope is very small, and the encoding times become dominated by other tasks.

We should also observe that in all our implementations we have carry propagation being done directly in the memory buffers where the compressed data is stored. In the past there were problems with this approach due to high memory costs, but now it is difficult to find applications in which this approach is not feasible. Dealing with bit carry propagation is intrinsically more complex than with bytes, but more importantly, bit carries occur much more frequently than byte carries, which is another reason the difference in the speed of the two approaches is so large.

The last graph in Figure 1 ("2 bytes – FP") contains the encoding times of a version that uses only double-precision floating-point arithmetic (in the next sections we will provide more information about it), which shows that even when slower operations are used, they still would be significantly smaller (and thus potentially masked) by the bit-based renormalization. In the rest of this document, all the experimental results corresponding to our implemention of arithmetic coding use byte-based renormalization and carry propagation.

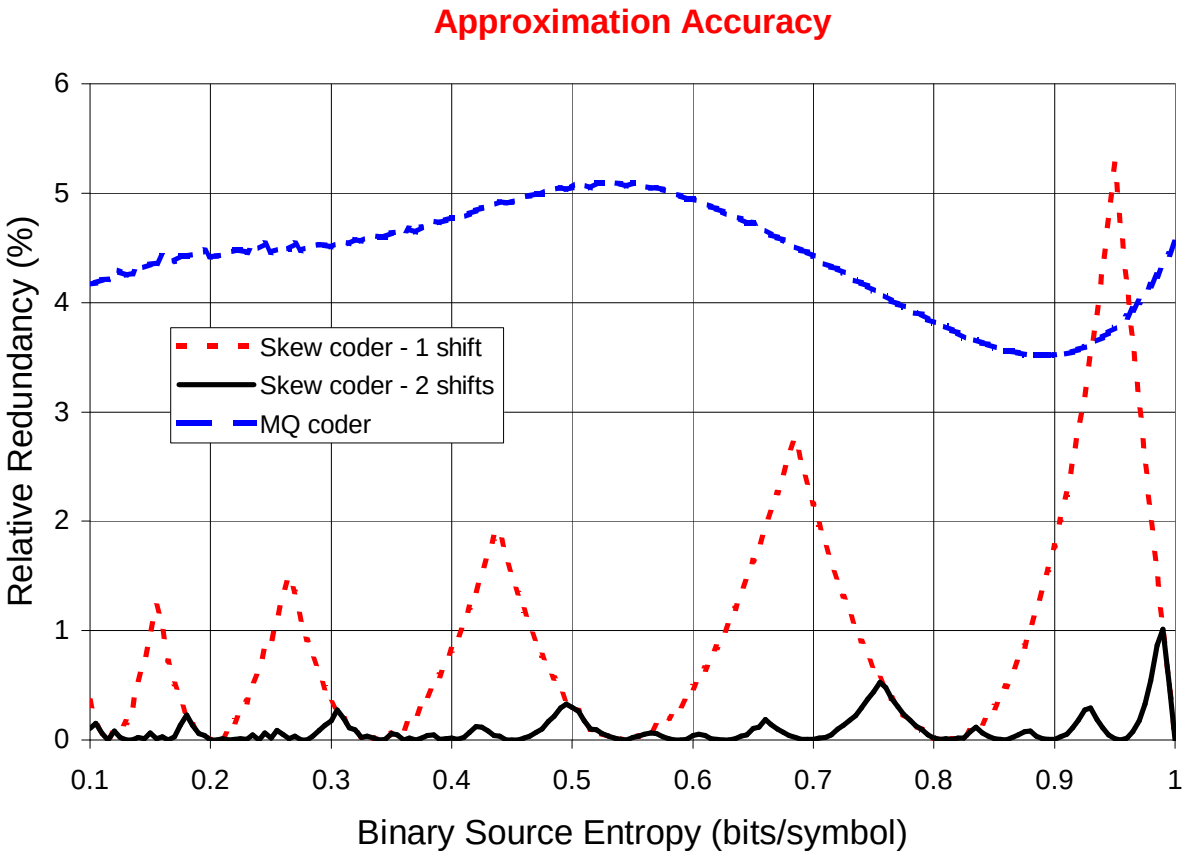

Figure 2: Relative redundancy of the Skew and MQ coders due to their approximation of multiplications.

4 Impact of Multiplications

Since integer multiplications are currently quite fast, we can only expect to see some of its impact in the simplest form of arithmetic coding, which corresponds to using static models with two data symbols. This type of coding is symmetrical since both the encoder and decoder basically need one multiplication, one addition, and two comparisons per coded symbol. There is also a variable number of operations for renormalization and carry propagation.

As a reference for comparisons we used three other implementations of binary arithmetic coding based on approximations. The first is "MQ Coder" that is used in the JPEG 2000 standard, and part of the Kakadu code by D. Taubman [3]. The other two are our adaptation of the "Skew Coder" proposed by Langdon and Rissanen [12], in which the multiplication is replaced with a bit shift. In our first implementation of the Skew coder, the probability of the least probable symbol (bit) is approximated by a value 2^{-b}, where b is a positive integer. Thus, the product probability times the interval length is computed by shifting the scaled interval length b bits. The second version approximates the probability of the least probable

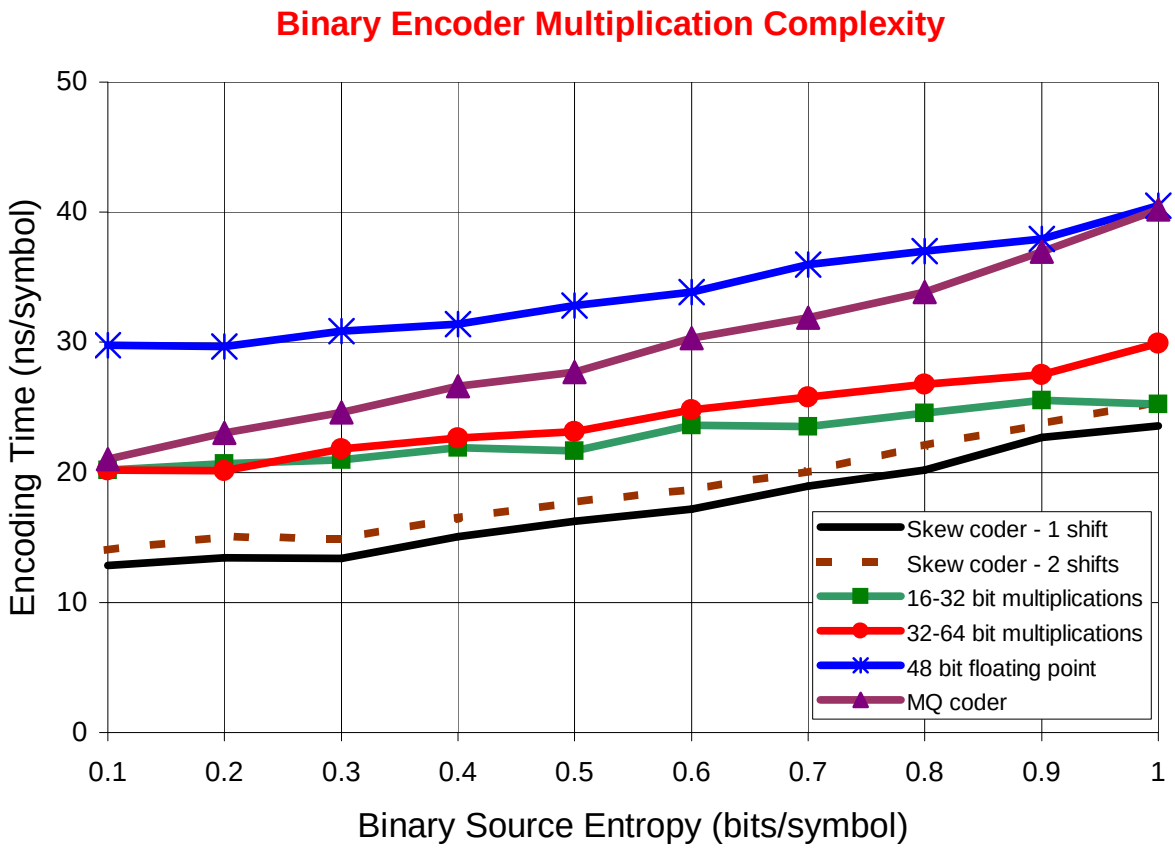

Figure 3: Comparison of encoding times of different implementations of binary arithmetic coding.

symbol as $2^{-b} + 2^{-c}$, which means that the product is computed by adding two bit shifts of the scaled interval length. Both these versions of the Skew coder use 32-bit registers. Note that, because of the approximations, the difference between the compression rates obtained with these encoders and the entropy may not be insignificant. Figure 2 shows the relative redundancy, which is the relative difference between the code rate and the entropy. Note that the MQ coder's bit rate is commonly about 4–5% above entropy.

Figures 3 and 4 show, respectively, the encoding and decoding time of our implementations of the Skew coders, and binary coders with 16, 32, and 48 bit arithmetic. The 16-bit version uses standard 32-bit arithmetic, but discards the 16 least significant bits before multiplication. The 32-bit version is more precise because it discards bits only after the multiplication, using assembler code to extract the most significant bits from the 64-bit product. These two versions are nearly identical, and their performance is quite similar. The version with 48-bit precision, on the other hand, is implemented using double-precision floating-point numbers, and for that reason is quite different from the others. It is fair to say that it is slower not only because of slower multiplication, but also because some bit

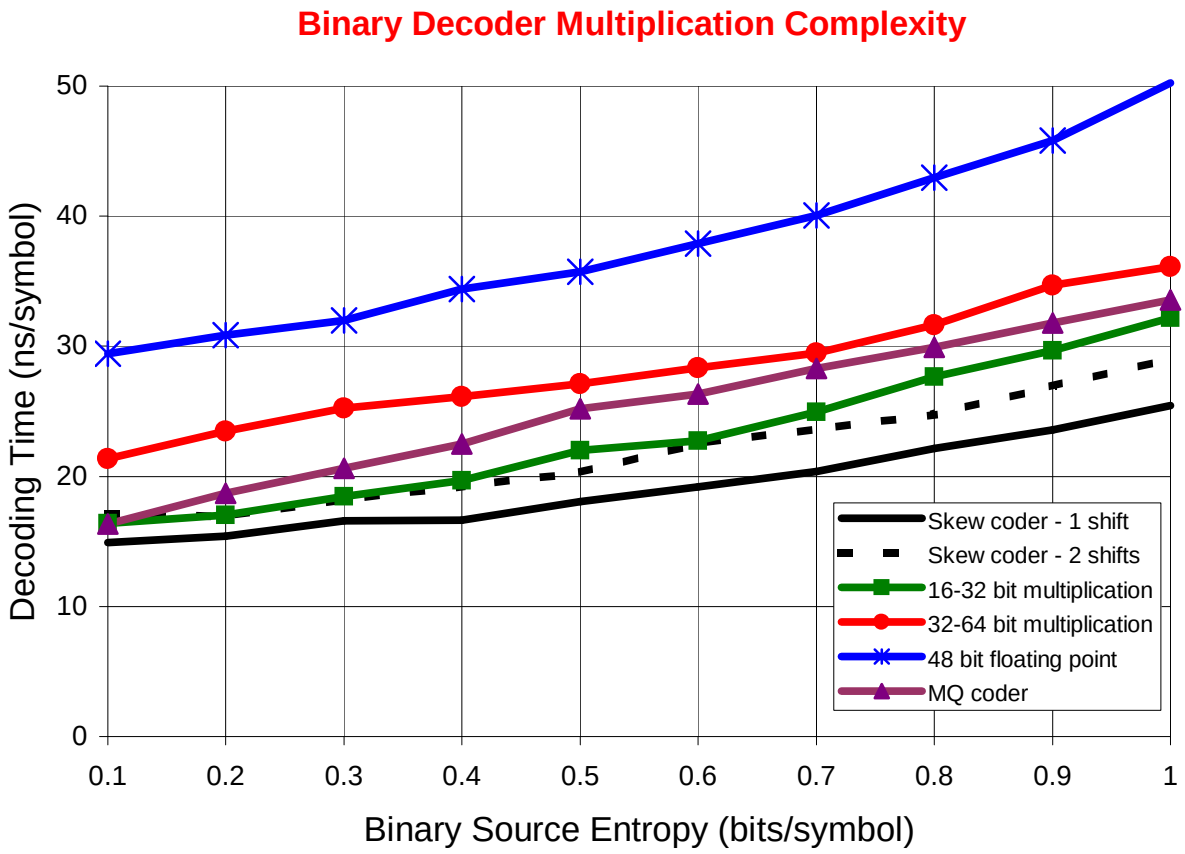

Figure 4: Comparison of decoding times of different implementations of binary arithmetic coding.

manipulations have to be replaced with extra multiplications and additions.

We can see from Figures 3 and 4 that the variation between all the different implementations is not large. Even the floating-point arithmetic, which would be unpractical a few years ago, now has comparable performance. It is important to note that the each of these versions is implemented in a different manner, and we have to consider that some of the differences in speed can be explained by other factors, like the optimization decisions taken by the compiler. However, we can reach the conclusion that even for this special case (binary coders), the use of multiplications certainly does not affect speed by a substantial factor, while it enables compression rates nearly identical to the entropy.

5 Information Throughput

One important measure for evaluating speed efficiency is the coder's *information throughput*, which is the number of bits per second coming out of the encoder, or getting into the decoder.

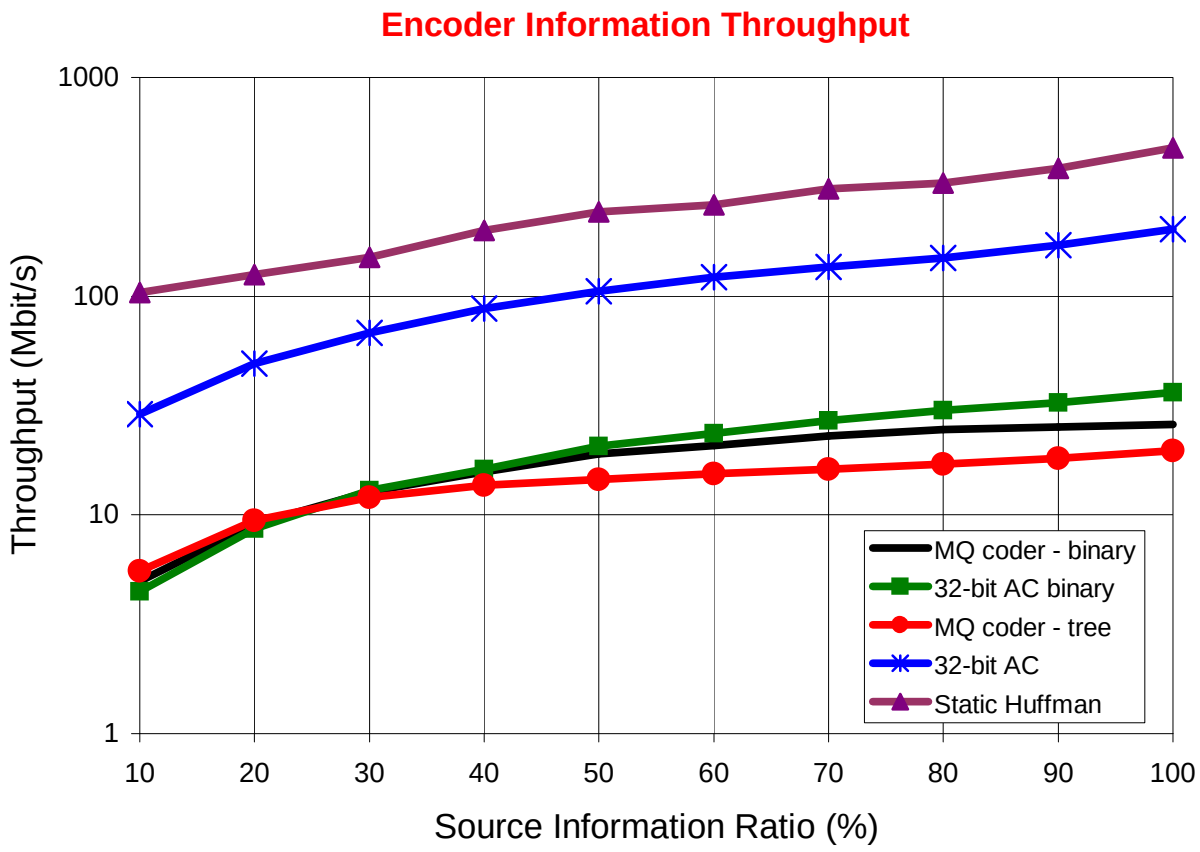

Figure 5: Information throughput of binary and multi-symbol encoders.

The value of this measure is that at the output of the encoder we have true bits of information, as defined by information theory, while at the encoder's input we have data symbols with different amounts of information content.

Figure 5 shows the information throughput of some encoders. The horizontal axis indicates the relative amount of information of the data source, per coded symbol. For example, the value 10% means 10:1 compression, 50% means 2:1 compression, and so on. The first two lines correspond to the MQ and our 32-bit precision implementation of arithmetic binary encoders, i.e., they are the same simulation results presented in Figure 3, but here they show that for low entropy sources, in which the binary coders may seem to be most efficient (smaller coding times), the information throughput is actually smaller.

Binary coders had been proposed as the only solution needed for any type of coding because of their relative simplicity, and because they can be used for an M-symbol data alphabet if we use $M-1$ binary coders in a binary tree structure [1, 2]. The next line of Figure 5 shows the information throughput of the MQ coder in such a tree structure. The data source used for these simulations has 256 symbols, which means that each symbol is coded with 8 calls to the binary encoder function. (The corresponding coding times are

shown in Figure 10.) Note that this sequential coding of the binary symbols in the tree does not affect the total throughput of the binary (MQ) coder by much. However, we can also see that the information throughput is almost the same because binary coders can put out at most one bit of information in a few CPU clock cycles.

The next two lines in Figure 5 shows that, in contrast, Huffman coding and the arithmetic coding methods that supports larger alphabets are significantly more efficient. (Note that the scale in the vertical axis is logarithmic). This happens because they can put out many bits of information in the same number of clock cycles—a more efficient parallelization of the coding process.

Since we are interested in the fastest coding techniques, in the rest of this document we consider only arithmetic coding that works with larger data alphabets. Despite the value of using information throughput for measuring performance, we use the encoding and decoding times because it allows more intuitive interpretations (as done in Section 3).

6 Use of Divisions for Decoding

While multiplications are relatively fast, divisions are still more demanding than other operations, and there is more variance of performance in different processors. There are two situations in which divisions may be needed for arithmetic coding. The first is during decoding, when we need to find the decoded symbol value s which satisfies

$$c_s \leq \frac{v-b}{l} < c_{s+1}, \tag{2}$$

where $\mathbf{c}$ is the array with cumulative distribution, v is the code value, and b and l are, respectively, the interval base and length. For decoding we can compute the fraction in (2) and then use a search method like bisection [1, 2].

An alternative way of decoding is to replace the division with a set of multiplications, in the form

$$l\, c_s \leq v-b < l\, c_{s+1}, \tag{3}$$

Figure 6 presents the results used for our analysis of how divisions affect the decoding complexity. The first line represents the encoding times, shown here as a reference, since the decoder cannot be faster than the encoder. The next line shows how the decoding times of the version that uses (3) is almost as efficient as the encoder for low-entropy sources, and grows linearly with the source entropy. This is expected because this version performs an optimal sequence of tests, and the number of multiplications is roughly proportional to the source entropy. The third line shows that if we use (2), then there is an initial difference in time that is due to the division, but a slower increase in time because we replace the multiplications and comparisons with only comparisons.

The last graph shows a version in which we use (2) in a different manner: first table look-up is used to reduce the initial search interval, followed by bisection search (if needed) [1]. We can see that this technique requires a larger initial time corresponding to the division

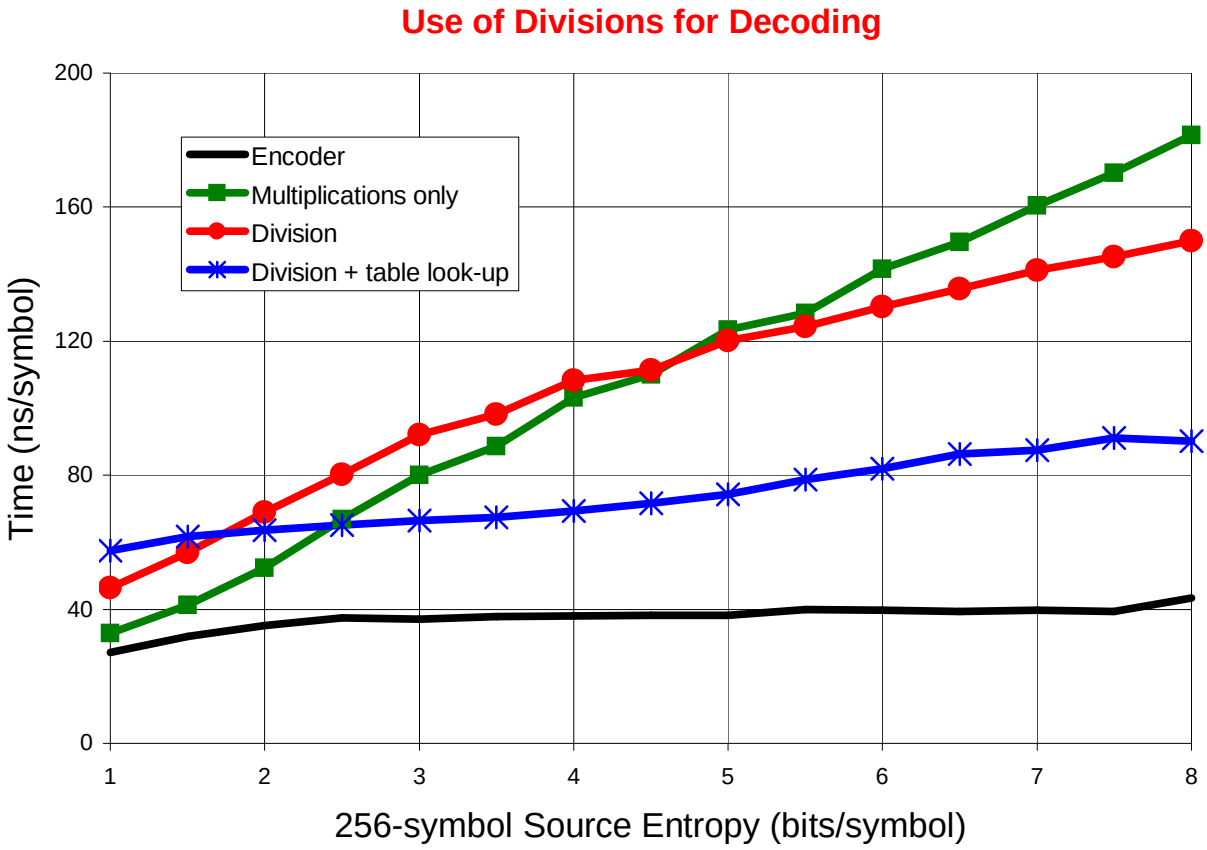

Figure 6: Effect of divisions and table look-up in the decoding times.

Table size	**256-symbol Source Entropy (bits/symbol)**							
	1	2	3	4	5	6	7	8
16	1.08	1.66	2.28	2.88	3.64	3.92	4.08	4.12
32	0.63	0.90	1.34	1.72	2.33	2.81	3.07	3.24
64	0.34	0.50	0.75	1.01	1.36	1.76	2.14	2.49
128	0.23	0.30	0.40	0.56	0.78	1.03	1.32	1.99
256	0.17	0.19	0.23	0.31	0.43	0.58	0.75	1.00

Table 1: Average number of tests in the bisection search after table look-up initialization.

and table look-up, but in consequence the growth in decoding time with source entropy is much slower. It is worth mentioning that for this technique the growth in decoding time is not linear. Table 1 shows how the average number of tests in the bisection search varies with table size and source entropy. The results in Figure 6 were obtained with a 64-entry table.

The results above lead to two main conclusions about the use of divisions. Since division can be significantly slower in some processors, for sources with small entropy (e.g., less than 3 bits/symbol, depending on the alphabet size and processor) it should be advantageous to use only multiplications. However, for larger entropies (and thus larger information throughputs) it is clearly better to use division and table look-up for decoding. Our results also show that, even for adaptive coders (Section 7), if divisions are used, then it is always better to use table look up. Table 1 shows that the tables do not have to be large for good results, and thus they can be computed quickly when doing periodic updates.

In this section we analyzed only the impact of the divisions required for decoding. In the next section we analyze the need for divisions for adaptive coding.

7 Probability Estimation Complexity

One of the main advantages of arithmetic coding is that it is relatively easy to make it adaptive. In comparison, adaptive versions of Huffman codes can be much more complex [11]. The main technical problem is that adaptive arithmetic coders need more than estimates of the symbol probabilities, which are easy to compute and update. Besides that, they also need an estimate of the cumulative distribution, which has the problem that changing the probability estimate of one symbol requires changing the cumulative distribution values of other symbols.

One solution to this problem is to sort the data source symbols according to probability. Assumed a skewed probability distribution, the symbols that have their probability estimate updated more frequently are those with higher probability. Thus, if the most probable symbols are last in the cumulative distribution, then on average only a few values are changed after each update. Unfortunately, if the distribution is flat (worst case) then we need, *for each coded symbol,* to change on average *half* the estimates of the cumulative distribution!

The first graph in Figure 7 shows the encoding times for the arithmetic coder of Witten, Neal, and Cleary [10] (called "WNC coder" in the next graphs), which uses this adaptation technique. This implementation is quite slow because it uses bit renormalization and some other obsolete techniques, but we use it as a reference since it is very well known. The encoder times increase with the source entropy due to both the more frequent renormalizations and the longer cumulative distribution updates.

Since the cumulative distribution of binary sources is basically defined by a single probability, binary coders can adapt very efficiently. However, the next graph in Figure 7 shows that this efficiency does not scale when we use binary coders for larger alphabets. Note that here MQ coder is used in a tree structure with 6 levels, i.e., each symbol is encoded with 6 calls to fast stages of encoding and adaptation.

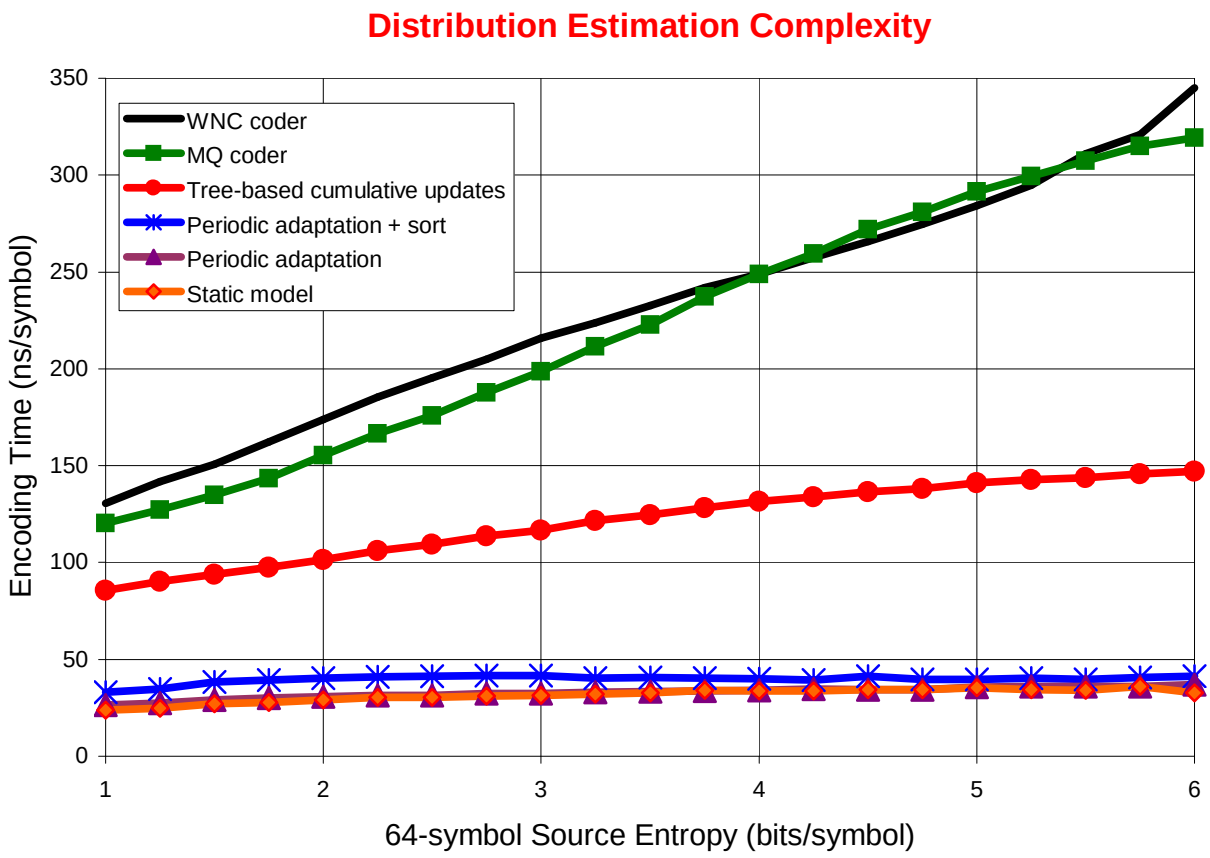

Figure 7: Encoding speed of adaptive coders with different techniques for updating the estimated cumulative distribution.

More recent work [9, 8] shows that we can use tree structures to efficiently update the cumulative distributions. The third line in Figure 7 shows its encoding speeds. We can see that it is significantly faster than using the MQ binary coders because instead of 6 stages of encoding and adaptation, it has 6 fast updates of partial cumulative sums, followed by a single coding stage, and possibly the byte-based renormalization.

An additional source of complexity for adaptive coders is that the estimates of probabilities and cumulative distribution values are normally in the form of a fraction, i.e.,

$$p_s = \frac{N_s}{C_M}, \quad c_s = \frac{C_s}{C_M}. \tag{4}$$

The consequence is that the two values that are always required for encoding and decoding each symbol—the new interval base and length—need to be computed using two multiplications and two divisions.

The last line of Figure 7, obtained with our static arithmetic coder (i.e., no adaptation), shows that the adaptation techniques presented above add a very substantial amount of

complexity to arithmetic coding. This extraordinary complexity may seem an unavoidable price to pay for the convenience of adaptation. However, an important observation is that the costliest data need to be recomputed for each coded symbol only if we insist on the requirement of always updating the estimates immediately after each coded symbol.

Some reflection on the properties of the estimation process shows that this requirement does not always makes sense. Except for trivial cases, the estimation of parameters of random processes is intrinsically slow and imprecise. While there is significant gains in the estimation precision after the first observations, later the precision improvement becomes quite slow. For example, we can get substantial information about a binary data source after observing the first 10 symbols, but, after 1000 symbols had already been observed, a much smaller amount of information is obtained by observing 10 more symbols.

We call *periodic adaptation* the process of updating the source estimates only after a certain number of symbols has been coded. Even though it is called periodic, we assume that the updates occur frequently when coding starts, and later become less frequent. This guarantees that there is little loss in compression.

The periodic adaptation strategy completely removes the need to use two divisions per coded symbol, which are replaced by one division in each update. In addition, since the updates in the cumulative distribution are not done for each coded symbol, during the periodic update we can do tasks that would be too expensive otherwise, like sorting the symbols according to probability to allow faster decoding [1, 2].

The fourth and fifth graphs of Figure 7 show that, because periodic adaptation is so effective, the resulting complexity reduction is indeed dramatic, allowing the adaptive coders to be, even with symbol sorting, nearly as fast as static coders. In this case, after updates of shorter period, the updates occur after every $8M$ coded symbols, where M is the number of data symbols. For instance, we eventually have one update every 512 symbols for the source used for the simulations of Figure 7. The worst-case complexity of just updating the cumulative distribution is $\mathrm{O}(M)$, while sorting and updating is $\mathrm{O}(M \log M)$.

8 Periodic Updates of Huffman Codes

Huffman coding is the most important competitor to arithmetic coding. It is well-known that static Huffman coding can be significantly faster than arithmetic coding (as seen in Figure 5). On the other hand, versions of Huffman coding that adapt for each coded symbol [11] are known to be much slower. Since we can use the same periodic adaptation strategy for Huffman codes, it is interesting to study its coding performance before comparing it to arithmetic coding.

Figure 8 shows our Huffman coding experimental results. The first graph shows static Huffman encoding, which is very fast because it consists primarily of a single table look-up, plus a few instructions for aligning bits and extracting the code bytes. The next two graphs show two different methods for Huffman decoding: using table look-up for instant decoding, or reading one bit at a time ("tree decoder") until a codeword is complete. Note how the

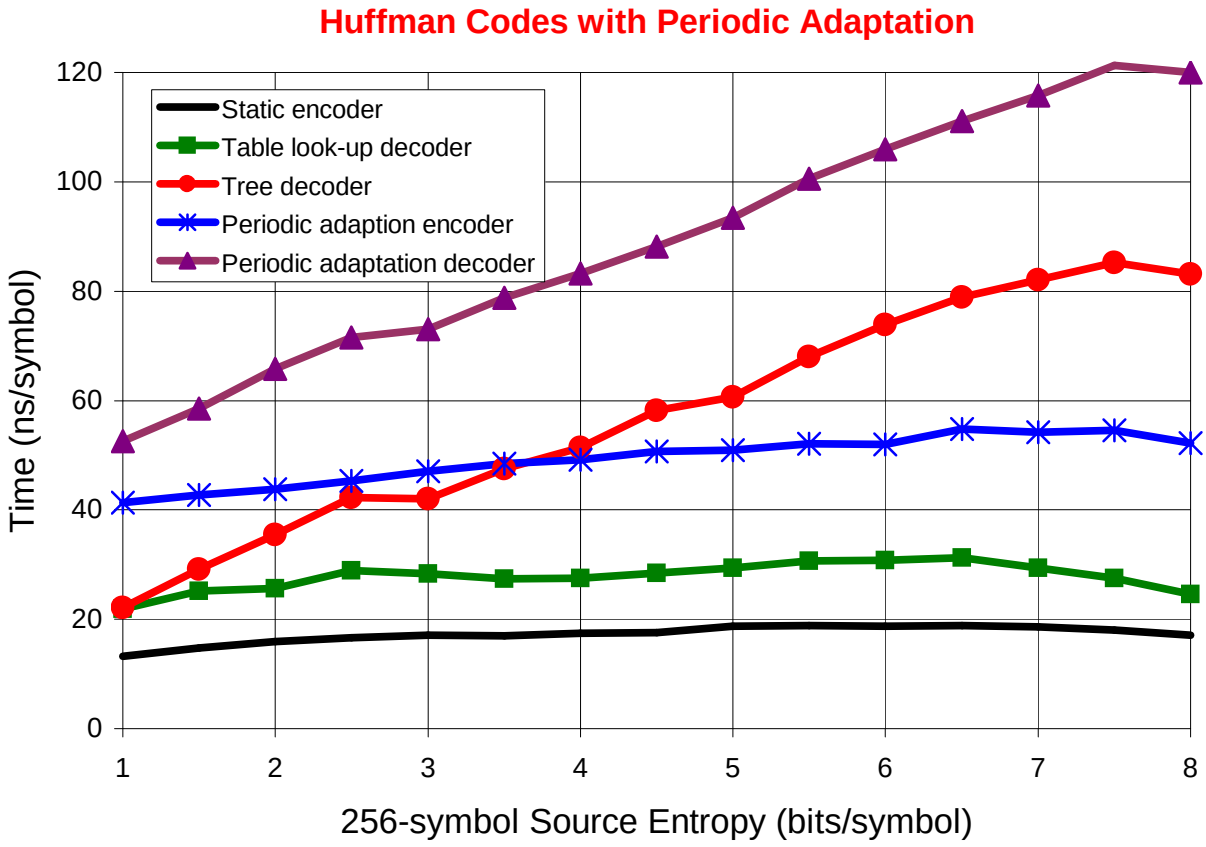

Figure 8: Speed of Huffman encoders and decoders using periodic adaptation.

decoding time of the latter grows linearly with entropy, providing one more example of the inefficiency of working with individual bits, and its heavy penalty on throughput.

The next graphs show that for Huffman coding even the periodic adaptation can increase complexity considerably. This happens because, even though the worst-case complexity of computing new Huffman codes is $O(M \log M)$, their computation is more complex than just sorting symbols and updating cumulative distributions. Similarly, the tables required for instant decoding can be very large, and it would be impractical to update them even in a periodic fashion. Smaller tables can be used, but their computation is not as intuitive as for arithmetic coding. For that reason our adaptive Huffman decoder does not use table look-up. Its performance is shown in the last graph in Figure 8.

A fundamental limit in the application of Huffman coding is that cannot be used efficiently at rates near or below 1 bit per symbol. For reference, in Figure 9 we show its relative redundancy when our data sources have low entropy. As expected, the redundancy of the Huffman code at 1 bit per symbol is quite large, but it drops to smaller values reasonably fast. The truncated geometric distributions of (1) do not correspond to the worst-case redundancy of Huffman codes, but we believe that the results in Figure 9 are not far from

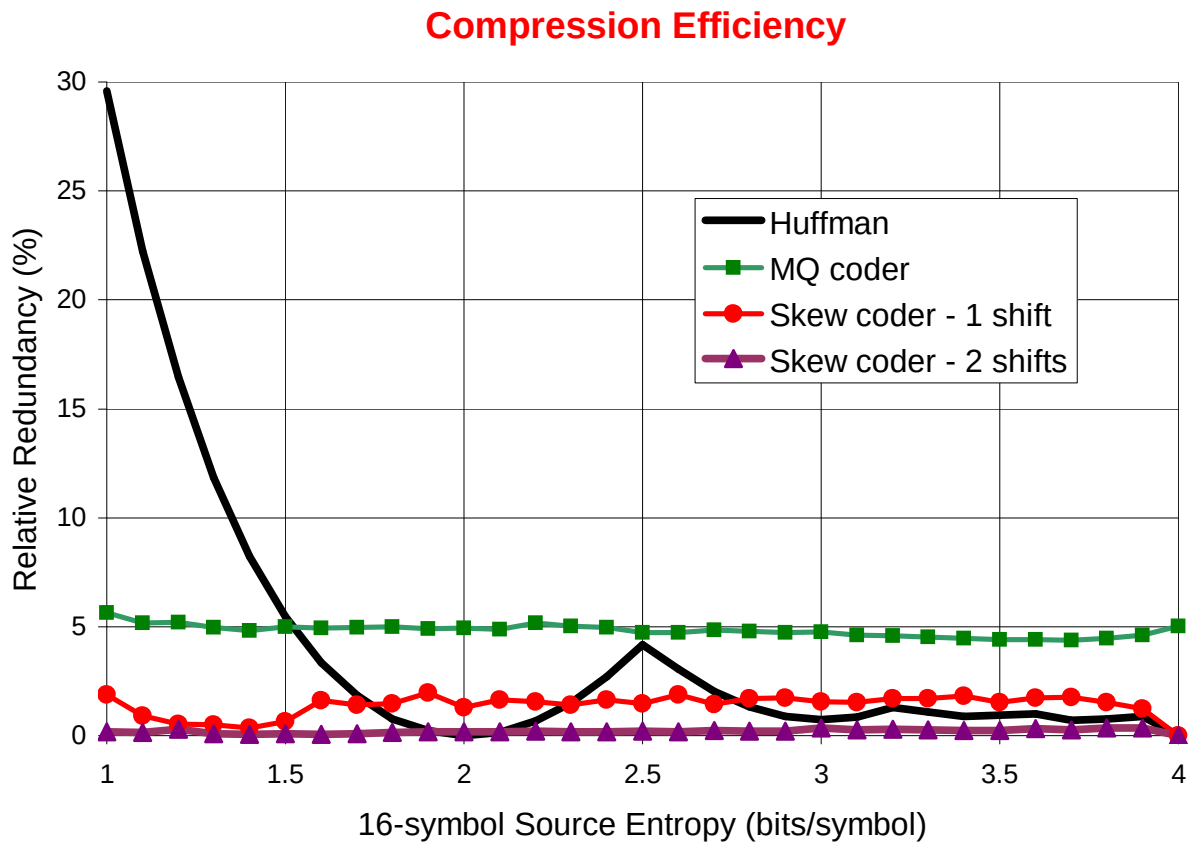

Figure 9: Relative redundancy of Huffman coding, compared to the redundancy of the binary Skew and MQ coders on decision trees.

typical. It shows that Huffman coding can produce near-optimal compression, but we may need to aggregate data symbols to increase their combined entropy. The practical problem is that it is not very easy to achieve this goal in the situations in which the source distribution is unknown, and it is exactly in those cases that arithmetic coding becomes a much more convenient solution.

We also display in Figure 9 the redundancy of arithmetic coders that use coarse approximations. Note that, when used in a tree structure to code larger alphabets, the redundancy of the MQ coder is nearly constant, which is consistent with results shown in Figure 2. However, we should stress that this is not the typical coding efficiency that can be achieved with the current processors. For instance, Figure 9 shows that even the Skew coder with one bit shift has significantly lower redundancy, while the Skew coder with two bit shifts has redundancy below 0.5%. The redundancy of all our implementations that use multiplications is much smaller than statistical uncertainty of the measurements.

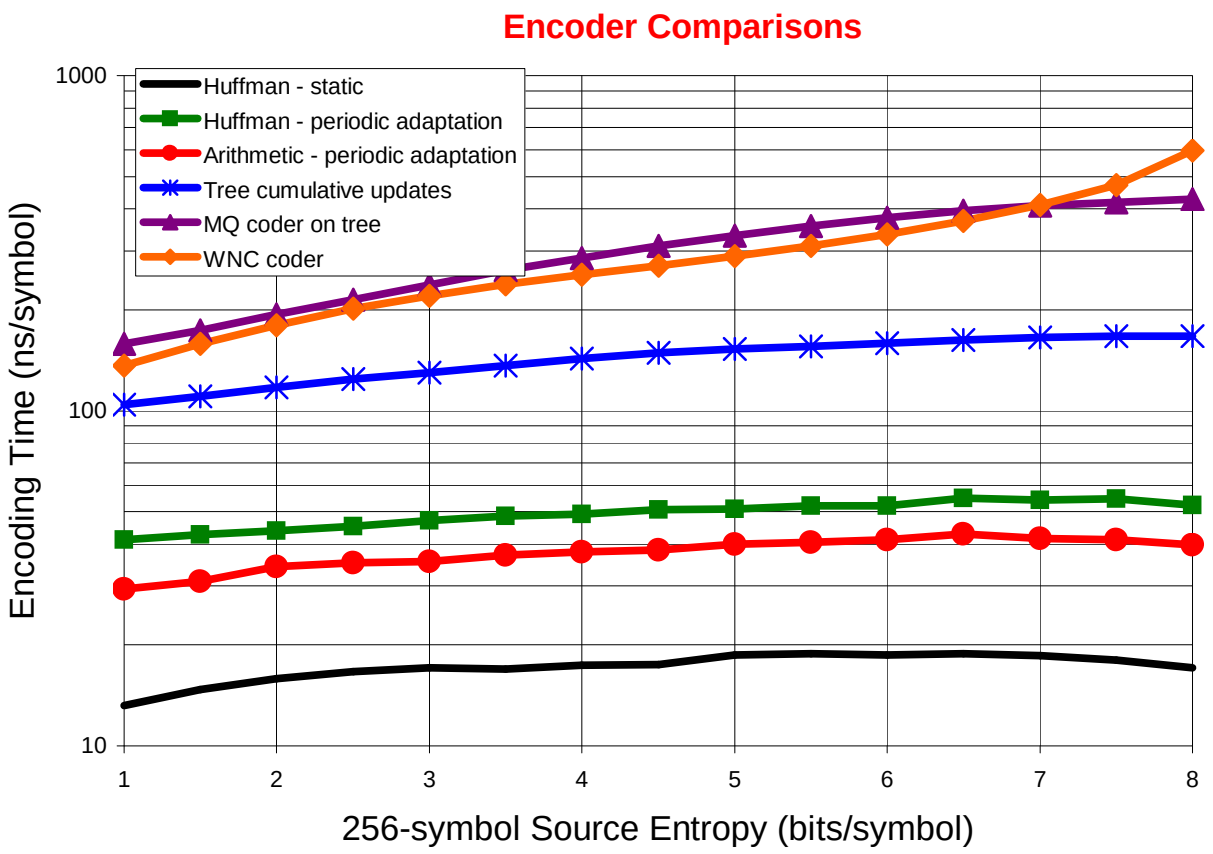

Figure 10: Encoding time of different algorithms compared to our fastest implementation of adaptive arithmetic coding.

9 Speed Comparisons

In this section we combine all the results to evaluate the current competitiveness of arithmetic coding implementations. In the previous sections we identified the choices in the implementation of arithmetic coding that yielded very significant reductions in complexity:

1. Full use of processor's supported arithmetic operations and precision.
2. Byte-based renormalization and carry propagation.
3. Periodic updates of cumulative distributions for adaptive coding.
4. Table look-up for faster symbol searches while decoding.

Our fastest arithmetic coding implementation uses all the techniques above, 32-bit registers and 16-bit precision for multiplications, 64-entry decoding tables, and updates with a frequency increasing from updates every 32 symbols to updates every 2048 symbols.

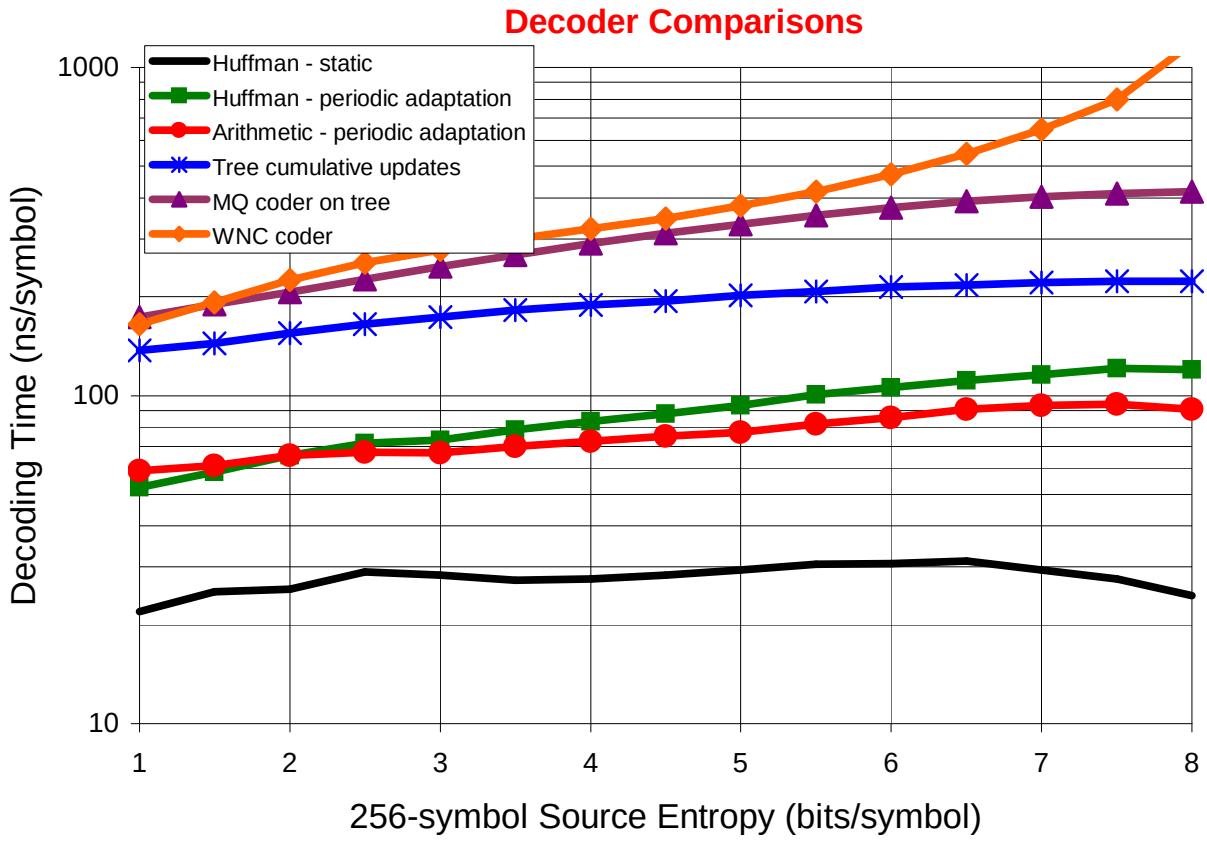

Figure 11: Decoding time of different algorithms compared to our fastest implementation of adaptive arithmetic coding.

Figure 10 shows the comparison of the encoder results. Note that the vertical axis here has a logarithmic scale. The first two graphs show that static Huffman coding is still clearly the fastest entropy-coding algorithm, but, at the same time, even with periodic adaptation, the adaptive version of Huffman coding can be significantly slower. The encoding times of our fastest implementation are shown next. We can see that it is definitely slower than static Huffman, but comparable to Huffman with periodic updates. The current speed may be acceptable considering the optimal compression and greater convenience for modeling unknown sources (see Figure 9).

The other graphs show that, without properly using the processor's capabilities, the other implementations are much slower. In fact, since they can be more than two orders of magnitude slower than static Huffman, it is indeed much harder to justify their use in practical applications. It may be surprising that the widely-used MQ binary coder turns out to be so slow, but we have shown that this is unavoidable because of its throughput limitations (Section 5). The binary Skew coder is slightly faster, but it has the same limitations, and produces similar results.

Figure 11 shows that very similar conclusions apply to arithmetic decoders. The WNC (Witten-Neal-Cleary) decoder is particularly slow at higher source entropies because it uses sequential search for decoding, which is also very inefficient in the worst case. It is easy to replace it with the more efficient bisection search, but we wanted to include these results only for providing references to well-known coders.

10 Conclusions

Our experimental results clearly show that the right implementations of arithmetic coding can fully exploit the power of current processors for fast and efficient coding. Furthermore, they confirm our expectation that the most efficient implementations can be the simplest, i.e., they are based almost entirely on the arithmetic operations for coding, without the need for any type of approximation or complicated acceleration techniques.

Using our comparative analysis we demonstrate that only the techniques that can fully exploit the CPUs hardware capabilities yield the best results in terms of encoding and decoding speed. We can also use these results to predict that a possible path to more efficient coding is the use of the increasing speed and resolution of general-purpose processors for parallelizing the coding process (e.g., use extended capabilities in SIMD and VLIW processor architectures).

Conversely, the results also show that the techniques that try to decompose the coding process in many simple steps are irrevocably obsolete. In fact, we were surprised to see that bit-based techniques are already one or two orders of magnitude slower, and we expect that the difference in speed will keep growing rapidly.

Below we list the main conclusions resulting from our experiments.

- There is a substantial speed gain as we move from renormalizations that save one bit a time, to those that save bits together in groups of one or more bytes.
- Byte-based renormalizations need enough precision from the arithmetic operations (e.g., at least 16 or 32 bits) to support a wider range of interval lengths. Normally these can be best supported by the native CPU operations, instead of approximations.
- Multiplications are now sufficiently fast, and their impact on the coding speed is small even for static binary coders.
- While binary coders perform all the coding operations in the shortest time, their information throughput is limited to at most one bit per coded symbol. For fastest coding we should use methods that code symbols from larger alphabets because they can yield much higher throughputs.
- Arithmetic decoding can be significantly slower that encoding, because of the search for the interval to which the coded symbol belongs. The best solution depends on the processor and data source.

  - If the source entropy is small (e.g., below 3 bits/symbol), then it is probably best to use a search method that uses only multiplications, and try to optimize the search sequence [1, 2].
  - Even though division is slower than the other operations, we can avoid long searches by using one division per decoded symbol and a table look-up for initializing the search. Small tables can significantly speed up the search.

- There are important advantages for not using probabilities estimates that are updated after every coded symbol, and instead use the new information only after a certain number of symbols had been coded (periodic updates).
  - There is a significant reduction in the complexity of updating the code, even compared to optimal tree-based updates of cumulative distributions.
  - It eliminates the need for two divisions per coded symbol, and makes possible high-precision adaptive implementations that use only multiplications.
  - Since probability estimation is by nature slow (as all statistical estimates), there is little loss in compression by waiting for a few symbols before updates.

References

[1] A. Said, *Introduction to Arithmetic Coding Theory and Practice,* Hewlett-Packard Laboratories Report, HPL–2004–76, April 2004.

[2] A. Said, "Arithmetic Coding," in *Lossless Compression Handbook,* K. Sayood, ed., Academic Press, San Diego, CA, 2003.

[3] D.S. Taubman and M.W. Marcellin, *JPEG 200 Image Compression Fundamentals, Standards and Practice,* Kluwer Academic Publishers, Boston, 2002.

[4] K. Sayood, *Introduction to Data Compression*, 2nd ed., Morgan Kaufmann Publishers, 2000.

[5] M. Schindler, "A fast renormalisation for arithmetic coding," *Proc. IEEE Data Compression Conf.*, 1998.

[6] A. Moffat, R.M. Neal, and I.H. Witten, "Arithmetic coding revisited," *ACM Trans. Information Systems*, vol. 6(3), pp. 256–294, 1998.

[7] P. L'Ecuyer, "Maximally equidistributed combined Tausworthe generators" *Mathematics of Computation,* vol. 65(213), pp. 203–213, Jan. 1996.

[8] A. Moffat, N. Sharman, I.H. Witten, and T.C. Bell, "An empirical evaluation of coding methods for multi-symbol alphabets," *Information Processing and Management,* 1994.

[9] P.M. Fenwick, "A new data structure for cumulative frequency tables," *Softw. Pract. Exper.*, vol. 24, pp. 327–336, March 1994.

[10] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression," *Commun. ACM,* vol. 30(6), pp. 520–540, June 1987.

[11] D.E. Knuth, "Dynamic Huffman coding," *Journal of Algorithms,* vol. 6, pp. 163–180, June 1985.

[12] G.G. Langdon Jr. and J.J. Rissanen, "Compression of black-white images with arithmetic coding," *IEEE Trans. Communications,* vol. 29(6), pp. 858–867, June 1981.