Arithmetic Coding for Data Compression

PAUL G. HOWARD AND JEFFREY SCOTT VITTER, FELLOW, IEEE

Arithmetic coding provides an effective mechanism for removing redundancy in the encoding of data. We show how arithmetic coding works and describe an efficient implementation that uses table lookup as a fast alternative to arithmetic operations. The reduced-precision arithmetic has a provably negligible effect on the amount of compression achieved. We can speed up the implementation further by use of parallel processing. We discuss the role of probability models and how they provide probability information to the arithmetic coder. We conclude with perspectives on the comparative advantages and disadvantages of arithmetic coding.

Index terms— Data compression, arithmetic coding, lossless compression, text modeling, image compression, text compression, adaptive, semi-adaptive.

I. Arithmetic coding

The fundamental problem of lossless compression is to decompose a data set (for example, a text file or an image) into a sequence of events, then to encode the events using as few bits as possible. The idea is to assign short codewords to more probable events and longer codewords to less probable events. Data can be compressed whenever some events are more likely than others. Statistical coding techniques use estimates of the probabilities of the events to assign the codewords. Given a set of mutually distinct events e_1, e_2, e_3, $\ldots$, e_n, and an accurate assessment of the probability distribution P of the events, Shannon [1] proved that the the smallest possible expected number of bits needed to encode an event is the *entropy* of P, denoted by

$$H(P) = \sum_{k=1}^{n} -p\{e_k\} \log_2 p\{e_k\},$$

where $p\{e_k\}$ is the probability that event e_k occurs. An optimal code outputs $-\log_2 p$ bits to encode an event whose probability of occurrence is p. Pure arithmetic codes supplied with accurate probabilities provides optimal compression. The older and better-known Huffman codes [2] are optimal only among instantaneous codes, that is, those in which the encoding of one event can be decoded before encoding has begun for the next event.

In theory, arithmetic codes assign one "codeword" to each possible data set. The codewords consist of half-open subintervals of the half-open unit interval $[0, 1)$, and are expressed by specifying enough bits to distinguish the subinterval corresponding to the actual data set from all other possible subintervals. Shorter codes correspond to larger subintervals and thus more probable input data sets. In practice, the subinterval is refined incrementally using the probabilities of the individual events, with bits being output as soon as they are known. Arithmetic codes almost always give better compression than prefix codes, but they lack the direct correspondence between the events in the input data set and bits or groups of bits in the coded output file.

A statistical coder must work in conjunction with a modeler that estimates the probability of each possible event at each point in the coding. The probability model need not describe the process that generates the data; it merely has to provide a probability distribution for the data items. The probabilities do not even have to be particularly accurate, but the more accurate they are, the better the compression will be. If the probabilities are wildly inaccurate, the file may even be expanded rather than compressed, but the original data can still be recovered. To obtain maximum compression of a file, we need both a good probability model and an efficient way of representing (or learning) the probability model.

To ensure decodability, the encoder is limited to the use of model information that is available to the decoder. There are no other restrictions on the model; in particular, it can change as the file is being encoded. The models can be *adaptive* (dynamically estimating the probability of each event based on all events that precede it), *semi-adaptive* (using a preliminary pass of the input file to gather statistics), or *non-adaptive* (using fixed probabilities for all files). Non-adaptive models can perform arbitrarily poorly [3]. Adaptive codes allow one-pass coding but require a more complicated data structure. Semi-adaptive codes require two passes and transmission of model data as side information; if the model data is transmitted efficiently they can provide slightly better compression than adaptive codes, but in general the cost of transmitting the model is about the same as the "learning" cost in the adaptive case [4].

To get good compression we need models that go beyond global event counts and take into account the structure of the data. For images this usually means using the numeric intensity values of nearby pixels to predict the intensity of each new pixel and using a suitable probability distribution for the residual error to allow for noise and variation between regions within the image. For text, the previous letters form a context, in the manner of a Markov process.

In Section II, we provide a detailed description of pure arithmetic coding, along with an example to illustrate the process. We also show enhancements that allow incremental transmission and fixed-precision arithmetic. In Section III we extend the fixed-precision idea to low precision, and show how we can speed up arithmetic coding with little degradation of compression performance by doing all the arithmetic ahead of time and storing the results in lookup tables. We call the resulting procedure *quasi-arithmetic coding*. In Section IV we briefly explore the possibility of parallel coding using quasi-arithmetic coding. In Section V we discuss the modeling process, separating it into structural and probability estimation components. Each component can be adaptive, semi-adaptive, or static; there are two approaches to the probability estimation problem. Finally, Section VI pro-

Manuscript received Mmm DD, 1993. Some of this work was performed while both authors were at Brown University and while the first author was at Duke University. Support was provided in part by NASA Graduate Student Researchers Program grant NGT–50420, by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM, and by Air Force Office of Scientific Research grant number F49620–92–J–0515. Additional support was provided by Universities Space Research Association/CESDIS associate memberships.

P. G. Howard is with AT&T Bell Laboratories, Visual Communications Research, Room 4C–516, 101 Crawfords Corner Road, Holmdel, NJ 07733–3030.

J. S. Vitter is with the Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129.

IEEE Log Number 0000000.

vides a discussion of the advantages and disadvantages of arithmetic coding and suggestions of alternative methods.

II. How arithmetic coding works

In this section we explain how arithmetic coding works and give operational details; our treatment is based on that of Witten, Neal, and Cleary [5]. Our focus is on encoding, but the decoding process is similar.

A. Basic algorithm for arithmetic coding

The algorithm for encoding a file using arithmetic coding works conceptually as follows:

1. We begin with a "current interval" $[L, H)$ initialized to $[0, 1)$.
2. For each event in the file, we perform two steps.
   (a) We subdivide the current interval into subintervals, one for each possible event. The size of a event's subinterval is proportional to the estimated probability that the event will be the next event in the file, according to the model of the input.
   (b) We select the subinterval corresponding to the event that actually occurs next, and make it the new current interval.
3. We output enough bits to distinguish the final current interval from all other possible final intervals.

The length of the final subinterval is clearly equal to the product of the probabilities of the individual events, which is the probability p of the particular sequence of events in the file. The final step uses at most $\lfloor -\log_2 p \rfloor + 2$ bits to distinguish the file from all other possible files. We need some mechanism to indicate the end of the file, either a special end-of-file event coded just once, or some external indication of the file's length. Either method adds only a small amount to the code length.

In step 2, we need to compute only the subinterval corresponding to the event a_i that actually occurs. To do this it is convenient to use two "cumulative" probabilities: the cumulative probability $P_C = \sum_{k=1}^{i-1} p_k$ and the next-cumulative probability $P_N = P_C + p_i = \sum_{k=1}^{i} p_k$. The new subinterval is $[L + P_C(H - L),\ L + P_N(H - L))$. The need to maintain and supply cumulative probabilities requires the model to have a complicated data structure, especially when many more than two events are possible.

We now provide an example, repeated a number of times to illustrate different steps in the development of arithmetic coding. For simplicity we choose between just two events at each step, although the principles apply to the multi-event case as well. We assume that we know *a priori* that we have a file consisting of three events (or three letters in the case of text compression); the first event is either a_1 (with probability $p\{a_1\} = \frac{2}{3}$) or b_1 (with probability $p\{b_1\} = \frac{1}{3}$); the second event is a_2 (with probability $p\{a_2\} = \frac{1}{2}$) or b_2 (with probability $p\{b_2\} = \frac{1}{2}$); and the third event is a_3 (with probability $p\{a_3\} = \frac{3}{5}$) or b_3 (with probability $p\{b_3\} = \frac{2}{5}$). The actual file to be encoded is the sequence $b_1 a_2 b_3$.

The steps involved in pure arithmetic coding are illustrated in Table 1 and Fig. 1. In this example the final interval corresponding to the actual file $b_1 a_2 b_3$ is $[\frac{23}{30}, \frac{5}{6})$. The length of the interval is $\frac{1}{15}$, which is the probability of $b_1 a_2 b_3$, computed by multiplying the probabilities of the three events: $p\{b_1\}p\{a_2\}p\{b_3\} = \frac{1}{3}\frac{1}{2}\frac{2}{5} = \frac{1}{15}$. In binary, the final interval is $[0.110001\ldots, 0.110101\ldots)$. Since all binary numbers that begin with 0.11001 are entirely within this interval, outputting **11001** suffices to uniquely identify the interval.

Table 1 Example of pure arithmetic coding

Action	Subintervals
Start	$[0, 1)$
Subdivide with left prob. $p\{a_1\} = \frac{2}{3}$	$[0, \frac{2}{3}), [\frac{2}{3}, 1)$
Input b_1, select right subinterval	$[\frac{2}{3}, 1)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[\frac{2}{3}, \frac{5}{6}), [\frac{5}{6}, 1)$
Input a_2, select left subinterval	$[\frac{2}{3}, \frac{5}{6})$
Subdivide with left prob. $p\{a_3\} = \frac{3}{5}$	$[\frac{2}{3}, \frac{23}{30}), [\frac{23}{30}, \frac{5}{6})$
Input b_3, select right subinterval	$[\frac{23}{30}, \frac{5}{6})$ $= [0.110001\ldots_2, 0.110101\ldots_2)$
Output **11001**	0.11001_2 is the shortest binary fraction that lies within $[\frac{23}{30}, \frac{5}{6})$

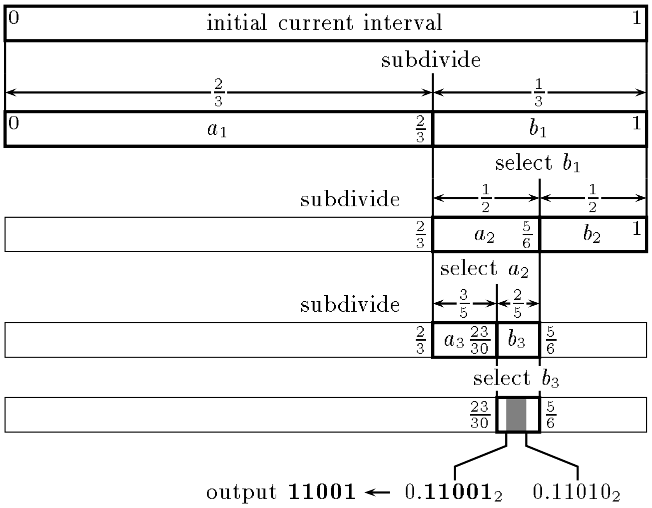

Fig. 1. Pure arithmetic coding graphically illustrated

B. Incremental output

The basic implementation of arithmetic coding described above has two major difficulties: the shrinking current interval requires the use of high precision arithmetic, and no output is produced until the entire file has been read. The most straightforward solution to both of these problems is to output each leading bit as soon as it is known, and then to double the length of the current interval so that it reflects only the unknown part of the final interval. Witten, Neal, and Cleary [5] add a clever mechanism for preventing the current interval from shrinking too much when the endpoints are close to $\frac{1}{2}$ but straddle $\frac{1}{2}$. In that case we do not yet know the next output bit, but we do know that whatever it is, the *following* bit will have the opposite value; we merely keep track of that fact, and expand the current interval symmetrically about $\frac{1}{2}$. This follow-on procedure may be repeated any number of times, so the current interval size is always strictly longer than $\frac{1}{4}$.

Mechanisms for incremental transmission and fixed precision arithmetic have been developed through the years by Pasco [6], Rissanen [7], Rubin [8], Rissanen and Langdon [9], Guazzo [10], and Witten, Neal, and Cleary [5]. The bit-stuffing idea of Langdon and others at IBM that limits the propagation of carries in the additions serves a function similar to that of the follow-on procedure described above.

Table 2 Example of pure arithmetic coding with incremental transmission and interval expansion

Action	Subintervals
Start	$[0,1)$
Subdivide with left prob. $p\{a_1\} = \frac{2}{3}$	$[0,\frac{2}{3}), [\frac{2}{3},1)$
Input b_1, select right subinterval	$[\frac{2}{3},1)$
Output **1**, expand	$[\frac{1}{3},1)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[\frac{1}{3},\frac{2}{3}), [\frac{2}{3},1)$
Input a_2, select left subinterval	$[\frac{1}{3},\frac{2}{3})$
Increment **follow** count, expand	$[\frac{1}{6},\frac{5}{6})$
Subdivide with left prob. $p\{a_3\} = \frac{3}{5}$	$[\frac{1}{6},\frac{17}{30}), [\frac{17}{30},\frac{5}{6})$
Input b_3, select right subinterval	$[\frac{17}{30},\frac{5}{6})$
Output **1**, output **0** (**follow** bit), expand	$[\frac{2}{15},\frac{2}{3})$
Output **01**	$[\frac{1}{4},\frac{1}{2})$ is entirely within $[\frac{2}{15},\frac{2}{3})$

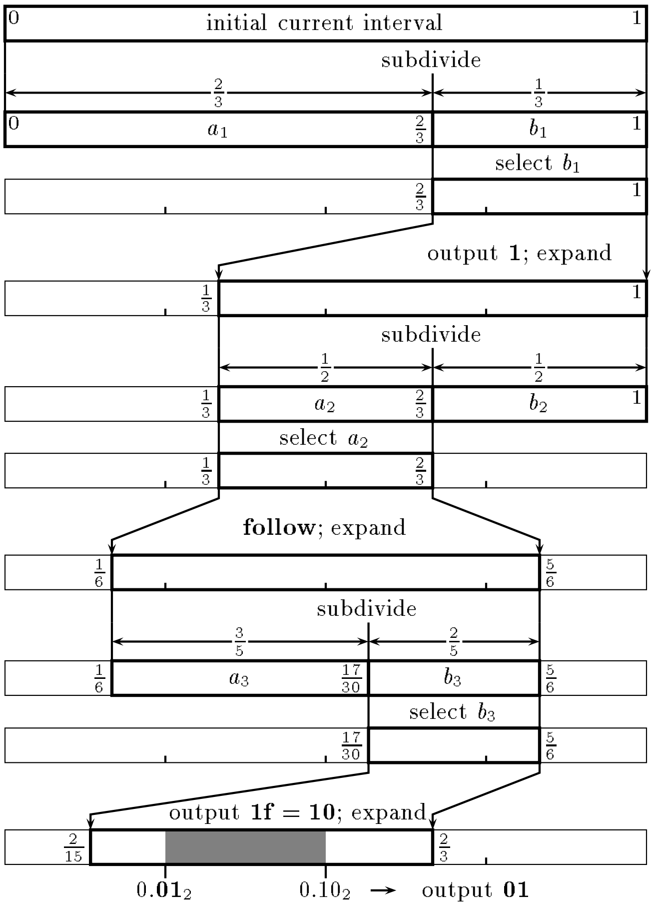

Fig. 2. Pure arithmetic coding with incremental transmission and interval expansion, graphically illustrated

We now describe in detail how the incremental output and interval expansion work. We add the following step immediately after the selection of the subinterval corresponding to an input event, step 2(b) in the basic algorithm above.

2. (c) We repeatedly execute the following steps in sequence until the loop is explicitly halted:
   1. If the new subinterval is not entirely within one of the intervals $[0,\frac{1}{2})$, $[\frac{1}{4},\frac{3}{4})$, or $[\frac{1}{2},1)$, we exit the loop and return.
   2. If the new subinterval lies entirely within $[0,\frac{1}{2})$, we output **0** and any following **1**s left over from previous events; then we double the size of the subinterval by linearly expanding $[0,\frac{1}{2})$ to $[0,1)$.
   3. If the new subinterval lies entirely within $[\frac{1}{2},1)$, we output **1** and any following **0**s left over from previous events; then we double the size of the subinterval by linearly expanding $[\frac{1}{2},1)$ to $[0,1)$.
   4. If the new subinterval lies entirely within $[\frac{1}{4},\frac{3}{4})$, we keep track of this fact for future output by incrementing the **follow** count; then we double the size of the subinterval by linearly expanding $[\frac{1}{4},\frac{3}{4})$ to $[0,1)$.

Table 2 and Fig. 2 illustrate this process. In the example, interval expansion occurs exactly once for each input event, but in other cases it may occur more than once or not at all. The follow-on procedure is applied when processing the second input event a_2. The **1** output after processing the third event b_3 is therefore followed by its complement **0**. The final interval is $[\frac{2}{15},\frac{2}{3})$. Since all binary numbers that start with 0.01 are within this range, outputting **01** suffices to uniquely identify the range. The encoded file is **11001**, as before. This is no coincidence: the computations are essentially the same. The final interval is eight times as long as in the previous example because of the three doublings of the current interval.

Clearly the current interval contains some information about the preceding inputs; this information has not yet been output, so we can think of it as the coder's state. If a is the length of the current interval, the state holds $-\log_2 a$ bits not yet output. In the basic method the state contains *all* the information about the output, since nothing is output until the end. In the incremental implementation, the state always contains fewer than two bits of output information, since the length of the current interval is always more than $\frac{1}{4}$. The final state in the incremental example is $[\frac{2}{15},\frac{2}{3})$, which contains $-\log_2 \frac{8}{15} \approx 0.907$ bits of information; the final two output bits are needed to unambiguously transmit this information.

C. Use of integer arithmetic

In practice, the arithmetic can be done by storing the endpoints of the current interval as sufficiently large integers rather than in floating point or exact rational numbers. We also use integers for the frequency counts used to estimate event probabilities. The subdivision process involves selecting non-overlapping intervals (of length at least 1) with lengths approximately proportional to the counts. Table 3 illustrates the use of integer arithmetic using a full interval of $[0, N) = [0, 1024)$. (The graphical version of Table 3 is essentially the same as Fig. 2 and is not included.) The length of the current interval is always at least $N/4 + 2$, 258 in this case, so we can always use probabilities precise to at least 1/258; often the precision will be near 1/1024. In practice we use even larger integers; the interval $[0, 65\,536)$ is a common choice, and gives a practically continuous choice of probabilities at each step. The subdivisions in this example are not quite the same as those in Table 2 because the resulting intervals are rounded to integers. The encoded file is **11001** as before, but for a longer input file the encodings would eventually diverge.

Table 3 Example of arithmetic coding with incremental transmission, interval expansion, and integer arithmetic. Full interval is $[0, 1024)$, so in effect subinterval endpoints are constrained to be multiples of $\frac{1}{1024}$.

Action	Subintervals
Start	$[0, 1024)$
$p\{a_1\} = \frac{2}{3}$; subdivide with left probability $= \frac{683}{1024} \approx 0.66699$	$[0, 683), [683, 1024)$
Input b_1, select right subinterval	$[683, 1024)$
Output **1**, expand	$[342, 1024)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[342, 683), [683, 1024)$
Input a_2, select left subinterval	$[342, 683)$
Increment **follow** count, expand	$[172, 854)$
$p\{a_1\} = \frac{3}{5}$; subdivide with left probability $= \frac{409}{682} \approx 0.59971$	$[172, 581), [581, 854)$
Input b_3, select right subinterval	$[581, 854)$
Output **1**, output **0** (**follow** bit), expand	$[138, 654)$
Output **01**	$[256, 512)$ is entirely within $[138, 654)$

Table 4 Example of arithmetic coding with incremental transmission, interval expansion, and small integer arithmetic. Full interval is $[0, 8)$, so in effect subinterval endpoints are constrained to be multiples of $\frac{1}{8}$.

Action	Subintervals
Start	$[0, 8)$
$p\{a_1\} = \frac{2}{3}$; subdivide with left prob. $= \frac{5}{8}$	$[0, 5), [5, 8)$
Input b_1, select right subinterval	$[5, 8)$
Output **1**, expand	$[2, 8)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[2, 5), [5, 8)$
Input a_2, select left subinterval	$[2, 5)$
Increment **follow** count, expand	$[0, 6)$
$p\{a_3\} = \frac{3}{5}$; subdivide with left prob. $= \frac{2}{3}$	$[0, 4), [4, 6)$
Input b_3, select right subinterval	$[4, 6)$
Output **1**, output **0** (**follow** bit), expand	$[0, 4)$
Output **0**, expand	$[0, 8)$

III. Limited-precision arithmetic coding

Arithmetic coding as it is usually implemented is slow because of the multiplications (and in some implementations, divisions) required in subdividing the current interval according to the probability information. Since small errors in probability estimates cause very small increases in code length, we expect that by introducing approximations into the arithmetic coding process in a controlled way we can improve coding speed without significantly degrading compression performance. In the Q-Coder work at IBM [11], the time-consuming multiplications are replaced by additions and shifts, and low-order bits are ignored. In [12] we describe a different approach to approximate arithmetic coding. Recalling that the fractional bits characteristic of arithmetic coding are stored as state information in the coder, we reduce the number of possible states, and replace arithmetic operations by table lookups; the lookup tables can be precomputed. Here we review this reduced precision binary arithmetic coder, which we call a *quasi-arithmetic coder*. It should be noted that the compression is still completely reversible; using reduced precision merely affects the average code length.

Table 5 Excerpts from quasi-arithmetic coding table, $N = 8$. Only the three states needed for the example are shown; there are nine more states. An "**f**" output indicates application of the follow-on procedure described in the text.

Start state	Probability of left input	Left (a) input: Output	Left (a) input: Next state	Right (b) input: Output	Right (b) input: Next state
$[0,8)$	$0.000 - 0.182$	**000**	$[0, 8)$	–	$[1, 8)$
	$0.182 - 0.310$	**00**	$[0, 8)$	–	$[2, 8)$
	$0.310 - 0.437$	**0**	$[0, 6)$	–	$[3, 8)$
	$0.437 - 0.563$	**0**	$[0, 8)$	**1**	$[0, 8)$
	$0.563 - 0.690$	–	$[0, 5)$	**1**	$[2, 8)$
	$0.690 - 0.818$	–	$[0, 6)$	**11**	$[0, 8)$
	$0.818 - 1.000$	–	$[0, 7)$	**111**	$[0, 8)$
$[0,6)$	$0.000 - 0.244$	**000**	$[0, 8)$	–	$[1, 6)$
	$0.244 - 0.415$	**00**	$[0, 8)$	**f**	$[0, 8)$
	$0.415 - 0.585$	**0**	$[0, 6)$	**f**	$[2, 8)$
	$0.585 - 0.756$	**0**	$[0, 8)$	**10**	$[0, 8)$
	$0.756 - 1.000$	–	$[0, 5)$	**101**	$[0, 8)$
$[2,8)$	$0.000 - 0.244$	**010**	$[0, 8)$	–	$[3, 8)$
	$0.244 - 0.415$	**01**	$[0, 8)$	**1**	$[0, 8)$
	$0.415 - 0.585$	**f**	$[0, 6)$	**1**	$[2, 8)$
	$0.585 - 0.756$	**f**	$[0, 8)$	**11**	$[0, 8)$
	$0.756 - 1.000$	–	$[2, 7)$	**111**	$[0, 8)$

Table 6 Example of operation of quasi-arithmetic coding

Start in state $[0, 8)$.
$\text{Prob}\{a_1\} = \frac{2}{3}$, so choose range $0.563 - 0.690$ in $[0, 8)$. First event is b_1, so choose right (b) input. Output **1**. Next state is $[2, 8)$.
$\text{Prob}\{a_2\} = \frac{1}{2}$, so choose range $0.415 - 0.585$ in $[2, 8)$. Second event is a_2, so choose left (a) input. **f** means increment **follow** count. Next state is $[0, 6)$.
$\text{Prob}\{a_3\} = \frac{3}{5}$, so choose range $0.585 - 0.756$ in $[0, 6)$. Third event is b_3, so choose right (b) input. Indicated output is **10**. Output **1**; output **0** to account for **follow** bit; output **0**. Next state is $[0, 8)$.

A. Development of binary quasi-arithmetic coding

We have seen that doing arithmetic coding with large integers instead of real or rational numbers hardly degrades compression performance at all. In Table 4 we show the encoding of the same file using small integers: the full interval $[0, N)$ is only $[0, 8)$.

The number of possible states (after applying the interval expansion procedure) of an arithmetic coder using the integer interval $[0, N)$ is $3N^2/16$. If we can reduce the number of states to a more manageable level, we can precompute all state transitions and outputs and substitute table lookups for arithmetic in the coder. The obvious way to reduce the number of states is to reduce N. The value of N should be a power of 2; its value must be at least 4. If we choose $N = 8$ and apply the arithmetic coding algorithm in a straightforward way, we obtain a table with 12 states, each state offering a choice of between 3 and 7 probability ranges. Portions of the table are shown in Table 5.

Table 6 shows how Table 5 is used to encode our sample file. Before coding each input event the coder is in a certain current state, corresponding to the current subinterval. For each state there are a number of probability ranges;

we choose the one that includes the estimated probability for the next event. Then we simply select the input event that actually occurs and perform the indicated actions: outputting bits, incrementing the **follow** count, and changing to a new current state. In the example the encoded output file is **1100**. Because we were using such low precision, the subdivision probabilities became distorted, leading to a lower probability for the file (1/16), but one which ends in the full interval $[0, 8)$, requiring no disambiguating bits. We usually use a somewhat larger value of N; in practice the compression inefficiency of a binary quasi-arithmetic coder (neglecting very large and very small probabilities) is less than one percent for $N = 32$ and about 0.2 percent for $N = 128$.

In implementations, the coding tables can be stripped down so that the numerical values of the interval endpoints and probability ranges do not appear. Full details and examples appear in [13]. The next section explains the computation of the probability ranges. Decoding uses essentially the same tables, and in fact is easier than encoding.

B. Analysis of binary quasi-arithmetic coding

We now prove that using binary quasi-arithmetic coding causes an insignificant increase in the code length compared with pure arithmetic coding. We mathematically analyze several cases.

In this analysis we exclude very large and very small probabilities, namely those that are less than $1/W$ or greater than $(W-1)/W$, where W is the width of the current interval. For these probabilities the relative excess code length can be large.

First we assume that we know the probability p of the left branch of each event, and we show both how to minimize the average excess code length and how small the excess is. In arithmetic coding we divide the current interval (of width W) into subintervals of length L and R; this gives an effective coding probability $q = L/W$ since the resulting code length is $-\log_2 q$ for the left branch and $-\log_2(1-q)$ for the right. When we encode a sequence of binary events with probabilities p and $1-p$ using effective coding probabilities q and $1-q$, the average code length $L(p, q)$ is given by

$$L(p, q) = -p \log_2 q - (1-p)\log_2(1-q).$$

If we use pure arithmetic coding, we can subdivide the interval into lengths pW and $(1-p)W$, thus making $q = p$ and giving an average code length equal to the entropy, $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$; this is optimal.

Consider two probabilities p_1 and p_2 that are adjacent based on the subdivision of an interval of width W; in other words, $p_1 = (W - \Delta_1)/W$, $p_2 = (W-\Delta_2)/W$, and $\Delta_2 = \Delta_1 - 1$. For any probability p between p_1 and p_2, either p_1 or p_2 should be chosen, whichever gives a shorter average code length. There is a cutoff probability p^* for which p_1 and p_2 give the same average code length. We can compute p^* by solving the equation $L(p^*, p_1) = L(p^*, p_2)$, giving

$$p^* = \frac{1}{1 + \dfrac{\log \dfrac{p_2}{p_1}}{\log \dfrac{1-p_1}{1-p_2}}} \tag{1}$$

We can use Equation (1) to compute the probability ranges in the coding tables. As an example, we compute the cutoff probability used in deciding whether to subdivide interval $[0, 6)$ as $\{[0,3),[3,6)\}$ or $\{[0,4),[4,6)\}$; this is the number 0.585 that appears in Table 5. In this case $p_1 = \frac{1}{2}$ and $p_2 = \frac{2}{3}$. We compute $p^* = \log(3/2)/\log 2 \approx 0.585$. Hence when we encode the third event in the example (with $p\{a_3\} = \frac{3}{5}$), we use the $\{[0,4),[4,6)\}$ subdivision.

Probability p^* is the probability between p_1 and p_2 with the worst average quasi-arithmetic coding performance, both in excess bits per event and in excess bits relative to optimal compression. This can be shown by monotonicity arguments.

Theorem 1 *If we construct a quasi-arithmetic coder based on full interval $[0, N)$, and use correct probability estimates, the number of bits per input event by which the average code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder is at most*

$$\frac{4}{\ln 2}\left(\log_2 \frac{2}{e \ln 2}\right)\frac{1}{N} + O\left(\frac{1}{N^2}\right) \approx \frac{0.497}{N} + O\left(\frac{1}{N^2}\right),$$

and the fraction by which the average code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder is at most

$$\left(\log_2 \frac{2}{e\ln 2}\right)\frac{1}{\log_2 N} + O\left(\frac{1}{(\log N)^2}\right) \approx \frac{0.0861}{\log_2 N} + O\left(\frac{1}{(\log N)^2}\right).$$

Proof: For a quasi-arithmetic coder with full interval $[0, N)$, the shortest terminal state intervals have size $W = N/4 + 2$. The worst average error occurs for the smallest W and the most extreme probabilities, that is, for $W = N/4 + 2$, $p_1 = (W-2)/W$, and $p_2 = (W-1)/W$ (or symmetrically, $p_1 = 1/W$ and $p_2 = 2/W$). For these probabilities, we find the cutoff probability p^*. Then for the first part of the theorem we take the asymptotic expansion of $L(p_2, p^*) - H(p_2)$, and for the second part we take the asymptotic expansion of $(L(p_2, p^*) - H(p_2))/H(p_2)$. □

If we let $\overline{p} = (p_1 + p_2)/2$, we can expand Equation (1) asymptotically in W and obtain an excellent rational approximation for p^*:

$$\widetilde{p^*} = \overline{p} + \frac{1}{6W^2}\frac{\overline{p} - 1/2}{\overline{p}(1-\overline{p})}.$$

The compression loss introduced by using $\widetilde{p^*}$ instead of p^* is completely negligible, never more than 0.06%. In the example above with $p_1 = \frac{1}{2}$, $p_2 = \frac{2}{3}$, and $W = 6$, we find that $p^* = \log(3/2)/\log 2 \approx 0.58496$ and $\widetilde{p^*} = 737/1260 \approx 0.58492$. As we expect, only for small values of W (the short intervals that occur when using very low precision arithmetic) do we need to be careful about roundoff when subdividing intervals; for larger values of W we can practically round the interval endpoints to the nearest integer.

We can prove a similar theorem for a more general case, in which we compare quasi-arithmetic coding with arithmetic coding for a single worst-case event. We assume that both coders use the same estimated probability, but that the estimate may be arbitrarily bad. (The proof is omitted to save space.)

Theorem 2 *If we construct a quasi-arithmetic coder based on full interval $[0, N)$, and use arbitrary probability estimates, the number of bits per input event by which the code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder in the worst case is at most*

$$\log_2 \frac{N+8}{N+4} \sim \frac{4}{N \ln 2} \approx \frac{5.771}{N}.$$

IV. Parallel coding

In [14] we present general-purpose algorithms for encoding and decoding using both Huffman and quasi-arithmetic coding. We are considering the case where we wish to do both encoding and decoding in parallel, so the location of bits in the encoded file must be computable by the decoding processors before they begin decoding. The main requirement for our parallel coder is that the model for each event is known ahead of time; it is not necessary for each event to have the same model. We use the PRAM model of parallel computation; the number of processors is p. We allow concurrent reading of the parallel memory, and limited concurrent writing, to a single one-bit register, which we call the *completion register*. It is initialized to **0** at the beginning of each time step; during the time step any number of processors (or none at all) may write **1** to it. We assume that n events are to be coded. The main issue is in dealing with the differences in code lengths of different events. For simplicity we do not consider input data routing.

We describe the algorithm for the Huffman coding case, since prefix codes are easier to handle, and then extend it to quasi-arithmetic coding. First the n events are distributed as evenly as possible among the p processors. Then each processor begins encoding its assigned events in a deterministic sequence, outputting one bit in each time step; the bits output in each time step are contiguous in the output stream. As soon as one or more processors complete all of their assigned events, they indicate this fact by writing **1** to the completion register. When the completion register is **1**, the output process is interrupted. Events whose processing has not begun are distributed evenly among the processors, using a prefix operation; events whose processing has begun but not finished remain assigned to their current processors. Processing of the reallocated events then continues until the next time that one or more processors finishes all their allocated events. Toward the end of processing, the number of remaining events will become less than the number of available processors. At this time we deactivate processors, making the number of active processors equal to the number of remaining events; we must redirect the output bits from the remaining processors. No further event reallocations are needed, but we still have to deactivate processors whenever any of them finish.

We analyze the time required by the parallel algorithm. Assuming that in one time unit each processor can output one bit, we can easily show that the time required for bit output is between $\lceil nH/p \rceil$ and $\lceil nH/p \rceil + L$. The more interesting analysis concerns the number of prefix operations required. We define a *phase* to be the period between prefix operations. *Early phases* are those which take place while the number of events is greater than the number of processors; each is followed by an event reallocation. *Late phases* are those needed to code the final p or fewer events; each late phase requires a prefix operation to redirect output bits. We bound the number of prefix operations needed in the following theorem.

Theorem 3 *When coding n events using p processors, the number of prefix operations needed by the reallocation coding algorithm is at most $L \log_2(2n/p)$ in the worst case.*

Proof: We define a *superphase* to be the time needed to halve the number of remaining events. Consider the first superphase. The number of events assigned to each processor ranges from n/p in the first phase down to $n/2p$ in the last. At least one processor completes all its events in each phase; such a processor must output at least one bit per event, since all code words in a Huffman code have at least one bit. This processor (and hence all processors) thus output at least $n/2p$ bits in each phase, making a total of at least $n/2$ bits output in each phase. The total number of bits that must be output in the first superphase is at most $nL/2$, so the number of phases in the first superphase is at most L.

The same reasoning holds for all superphases. The number of superphases needed to reduce the number of remaining events from n to p is $\log_2(n/p)$, so the number of phases needed is just $L \log_2(n/p)$. Once p or fewer events remain, at most L late phases are required, so the total number of phases needed is at most $L \log_2(2n/p)$. □

The extension of the parallel Huffman algorithm to quasi-arithmetic coding is fairly straightforward. The only complication arises when the last event of a processor's allocation leaves the processor in a state other than the starting state. We deal with this by outputting the smallest possible number of bits (0, 1, or 2) needed to identify a subinterval that lies within the final interval; this is the end-of-file disambiguation problem that we have seen in Section II.

V. Modeling

The goal of modeling for statistical data compression is to provide probability information to the coder. The modeling process consists of structural and probability estimation components; each may be adaptive, semi-adaptive, or static. In addition there are two strategies for probability estimation. The first is to estimate each event's probability individually based on its frequency within the data set. The other strategy is to estimate the probabilities collectively, assuming a probability distribution of a particular form and estimating the parameters of the distribution, either directly or indirectly. For direct estimation, we simply compute an estimate of the parameter (the variance, for instance) from the data. For indirect estimation [15], we start with a small number of possible distributions, and compute the code length that we would obtain with each; then we select the one with the smallest code length. This method is very general and can be used even for distributions from different families, without common parameters.

For lossless image compression, the structural component is usually fixed, since for most images pixel intensities are close in value to the intensities of nearby pixels. We code pixels in a predetermined sequence, predicting each pixel's intensity using a fixed linear combination of a fixed constellation of nearby pixels, then coding the prediction error. Typically the prediction errors have a symmetrical exponential-like distribution with zero mean, so the probability estimation component consists of estimating the variance and possibly some other parameters of the distribution, either directly or indirectly. A collection of only a few dozen distributions is sufficient to code most images with minimal compression loss.

For text, on the other hand, the best methods involve constantly changing structures. In text compression the events to be coded are just the letters in the file. Typically we begin by assuming only that some unspecified sequences of letters will occur frequently. (We may also specify a maximum length for the frequently occurring sequences.) As encoding proceeds we determine which sequences are most frequent. In the Ziv-Lempel dictionary-based methods [16, 17], the sequences are placed directly in the dictionary. The advantage of Ziv-Lempel methods is their speed, obtained by coding directly from the dictionary data structure, bypassing the explicit probability estimation and statistical coding stages. The PPM method [18] obtains better compression by constructing a Markov model of moderate order, proceeding to higher orders as more of the file is encoded. Complicated data structures are used to build and update the context information.

Most models for text compression involve estimating the letter probabilities individually, since there is no obvious mathematical relationship between the probabilities of different letters. (Numerical proximity of ASCII representations does not imply anything about probabilities.) We usually estimate the probability p of a given letter by

$$p = \frac{\text{weight of letter}}{\text{total weight of all letters}}.$$

The weight of a letter is usually based on the number of occurrences of the letter in a particular conditioning context.

Since we are often dealing with contexts that have occurred only a few times, we have to deal with letters that have never occurred. We cannot give a weight of 0 to such letters because that would lead to probability estimates of 0, which arithmetic coding cannot handle. This is the *zero-frequency problem*, thoroughly investigated by Witten and Bell [19]. It is possible to assign an initial weight of 1 to all possible letters, but a better strategy is to assign initial 0 weights and to include a special *escape* event indicating "some letter that has never occurred before in this context"; this event has its own weight, and must be encoded like an ordinary letter whenever a new letter occurs. There are a number of methods of dealing with the zero-frequency problem, differing mainly in the computation of the escape probability.

A second issue that arises in text compression is *locality of reference*: strings tend to occur in clusters within a text file. One way to take advantage of locality is to scale the counts periodically, typically by halving all weights when the total weight reaches a specified number. This effectively gives more weight to more recent occurrences of each letter, and has the additional benefit of keeping the counts small enough to fit into small registers. Analysis [4] shows that scaling often helps compression, and can never hurt it by very much.

One technique for probability estimation when only two events are possible is to use small scaled counts, considering each count pair to be a probability state. We can then precompute both the corresponding probabilities and the state transitions caused by the occurrence of additional events. The states can be identified by index numbers, which in turn can be used to index into the probability range lists in quasi-arithmetic code tables like those in Table 5.

VI. Conclusion

The main advantages of arithmetic coding for statistical data compression are its optimality and its inherent separation of coding and modeling. Pure arithmetic coding, as described in Section II, is strictly optimal for a stochastic data source whose probabilities are accurately known, but it relies on relatively slow arithmetic operations like multiplication and division. The quasi-arithmetic coding method described in Section III, which uses table lookup as a low-precision alternative to full-precision arithmetic, is nearly optimal and fairly fast, even when the probabilities are close to 1 or 0.

The separation of coding and modeling is important because it permits any degree of complexity in the modeler without requiring any change to the coder. In particular, the model structure and probability estimates can change adaptively. (In Huffman coding, by contrast, the probability information must be built into the coding tables, making adaptive coding difficult.) Even totally disjoint data streams can be intertwined; the only requirement is that the decoder must be able to track the model structure of the encoder. One occasionally useful advantage to arithmetic coding is that it is easy to maintain lexicographic order without any loss in compression efficiency, so that encoded strings can have the same order as the original unencoded data; maintaining lexicographic order with a prefix code requires a more complicated algorithm (the Hu-Tucker algorithm [20] and entails some sacrifice in efficiency.

The disadvantages of arithmetic coding are that it runs slowly, it is fairly complicated to implement, and it does not produce prefix codes. It can be speeded up with only a slight loss in compression efficiency by using approximations like quasi-arithmetic coding, the patented IBM Q-Coder, or Neal's low-precision coder [21]. The implementation difficulties are not insurmountable, but arithmetic coding will not be available as an off-the-shelf package until a fast, efficient, patent-free method is agreed upon as a standard. The non-prefix-code property leads to some technical difficulties: error resistance can be a serious problem, especially with adaptive models, and the output delay can be unbounded in the Witten-Neal-Cleary version described here, although not in the Q-Coder.

For binary alphabets, various forms of run-length encoding can be used instead of arithmetic coding, even if the probabilities are highly skewed. Golomb coding [22] and the related Rice coding [23] are based on exponentially distributed run lengths. The Golomb and Rice methods each consist of a family of codes parameterized by a single parameter; the parameter can be estimated adaptively [15] giving good compression efficiency. These codes are extremely fast prefix codes and are easily implemented in software or hardware. Other non-parameterized run-length codes like Elias codes [24] are less flexible and hence less useful.

When more than two input events are possible, the main alternatives to arithmetic coding are Huffman coding and coding based on splay trees [25]. Moffat *et al.* [26] compare arithmetic coding with these methods. Huffman coding is very effective in conjunction with a semi-adaptive model; the probability information can be built into the coding tables, leading to fast execution. Splay trees are even faster, since the data structure for coding *is* the "statistical" model; compression efficiency suffers somewhat. Golomb and Rice coding can be used when many events are possible, maintaining lists of the events in approximate probability order and coding the positions of the events within the lists. This is an especially useful technique for very large alphabets [27].

The main usefulness of arithmetic coding is in obtaining maximum compression in conjunction with an adaptive model, or when the probability of one event is much larger than 1/2. Arithmetic coding gives optimal compression, but its slow execution can be problematical. Approximate versions of arithmetic coding give almost optimal compression at improved speeds. Probabilities can be estimated approximately too, again leading to only slight degradation of compression performance.

Acknowledgments

The authors would like to acknowledge helpful suggestions made by Marty Cohn.

References

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.*, 27, pp. 398–403, July 1948.

[2] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the Institute of Radio Engineers*, 40, pp. 1098–1101, 1952.

[3] T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression*. Englewood Cliffs, NJ, Prentice-Hall, 1990.

[4] P. G. Howard and J. S. Vitter, "Analysis of Arithmetic Coding for Data Compression," *Information Processing and Management*, 28, no. 6, pp. 749–763, 1992.

[5] I. H. Witten, R. M. Neal and J. G. Cleary, "Arithmetic Coding for Data Compression," *Comm. ACM*, 30, no. 6, pp. 520–540, June 1987.

[6] R. Pasco, "Source Coding Algorithms for Fast Data Compression," Stanford Univ., Ph.D. Thesis, 1976.

[7] J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop.*, 20, no. 3, pp. 198–203, May 1976.

[8] F. Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers," *IEEE Trans. Inform. Theory*, IT–25, no. 6, pp. 672–675, Nov. 1979.

[9] J. J. Rissanen and G. G. Langdon, "Arithmetic Coding," *IBM J. Res. Develop.*, 23, no. 2, pp. 146–162, Mar. 1979.

[10] M. Guazzo, "A General Minimum-Redundancy Source-Coding Algorithm," *IEEE Trans. Inform. Theory*, IT–26, no. 1, pp. 15–25, Jan. 1980.

[11] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon and R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Develop.*, 32, no. 6, pp. 717–726, Nov. 1988.

[12] P. G. Howard and J. S. Vitter, "Practical Implementations of Arithmetic Coding," in *Image and Text Compression*, J. A. Storer, Ed. Norwell, MA: Kluwer Academic Publishers, pp. 85–112, 1992.

[13] ———, "Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, Eds. Snowbird, Utah: pp. 98–107, Mar. 30-Apr. 1, 1993.

[14] ———, "Parallel Lossless Image Compression Using Huffman and Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, Eds. Snowbird, Utah: pp. 299–308, Mar. 24-26, 1992.

[15] ———, "Fast and Efficient Lossless Image Compression," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, Eds. Snowbird, Utah: pp. 351–360, Mar. 30-Apr. 1, 1993.

[16] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Inform. Theory*, IT–23, no. 3, pp. 337–343, May 1977.

[17] ———, "Compression of Individual Sequences via Variable Rate Coding," *IEEE Trans. Inform. Theory*, IT–24, no. 5, pp. 530–536, Sept. 1978.

[18] J. G. Cleary and I. H. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Trans. Comm.*, COM–32, no. 4, pp. 396–402, Apr. 1984.

[19] I. H. Witten and T. C. Bell, "The Zero Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression," *IEEE Trans. Inform. Theory*, IT–37, no. 4, pp. 1085–1094, July 1991.

[20] T. C. Hu and A. C. Tucker, "Optimal Computer-Search Trees and Variable-Length Alphabetic Codes," *SIAM J. Appl. Math.*, 21, no. 4, pp. 514–532, 1971.

[21] R. M. Neal, "Fast Arithmetic Coding Using Low-Precision Division," Unpublished manuscript, 1987.

[22] S. W. Golomb, "Run-Length Encodings," *IEEE Trans. Inform. Theory*, IT–12, no. 4, pp. 399–401, July 1966.

[23] R. F. Rice, "Some Practical Universal Noiseless Coding Techniques," Jet Propulsion Laboratory, Pasadena, California, JPL Publication 79–22, Mar. 1979.

[24] P. Elias, "Universal Codeword Sets and Representations of Integers," *IEEE Trans. Inform. Theory*, IT–21, no. 2, pp. 194–203, Mar. 1975.

[25] D. W. Jones, "Application of Splay Trees to Data Compression," *Comm. ACM*, 31, no. 8, pp. 996–1007, Aug. 1988.

[26] A. M. Moffat, N. Sharman, I. H. Witten and T. C. Bell, "An Empirical Evaluation of Coding Methods for Multi-Symbol Alphabets," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, Eds. Snowbird, Utah: pp. 108–117, Mar. 30-Apr. 1, 1993.

[27] A. M. Moffat, "Economical Inversion of Large Text Files," presented at Computing Systems, 1992.

Paul G. Howard is a Member of Technical Staff in the Visual Communications Research Department of AT&T Bell Laboratories. He received the B.S. degree in computer science from M.I.T. in 1977 and the M.S. and Ph.D. degrees in computer science from Brown University in 1989 and 1993 respectively. His research interests are in data compression, including coding, image modeling, and text modeling. He is a member of Sigma Xi and an associate member of the Center of Excellence in Space Data and Information Sciences. He was briefly a Research Associate at Duke University before joining AT&T in 1993.

Jeffrey Scott Vitter (Fellow, IEEE) is the Gilbert, Louis, and Edward Lehrman Professor of Computer Science and the Chair of the Department of Computer Science at Duke University. Previously he was Professor at Brown University, where he was on the faculty since 1980. He received the B.S. degree in mathematics with highest honors from the University of Notre Dame in 1977 and the Ph.D. degree in computer science from Stanford University in 1980. He is a Guggenheim Fellow, an NSF Presidential Young Investigator, and an IBM Faculty Development Awardee. He is coauthor of the book *Design and Analysis of Coalesced Hashing* (Oxford University Press, 1987) and is coholder of a patent in the area of external sorting. He has written numerous articles and has consulted often in the areas of combinatorial algorithms, I/O efficiency, parallel and incremental computation, computational geometry, and machine learning. He serves on several editorial boards and is a frequent editor of special issues and member of program committees. He has served ACM SIGACT as Member-at-Large from 1987 to 1991 and as Vice Chair since 1991. He is currently an associate member of the Center of Excellence in Space Data and Information Sciences.