Arithmetic Coding Revisited

ALISTAIR MOFFAT
The University of Melbourne
RADFORD M. NEAL
University of Toronto
and
IAN H. WITTEN
The University of Waikato

Over the last decade, arithmetic coding has emerged as an important compression tool. It is now the method of choice for adaptive coding on multisymbol alphabets because of its speed, low storage requirements, and effectiveness of compression. This article describes a new implementation of arithmetic coding that incorporates several improvements over a widely used earlier version by Witten, Neal, and Cleary, which has become a *de facto* standard. These improvements include fewer multiplicative operations, greatly extended range of alphabet sizes and symbol probabilities, and the use of low-precision arithmetic, permitting implementation by fast shift/add operations. We also describe a modular structure that separates the coding, modeling, and probability estimation components of a compression system. To motivate the improved coder, we consider the needs of a word-based text compression program. We report a range of experimental results using this and other models. Complete source code is available.

Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*data compaction and compression*; E.1 [**Data**]: Data Structures

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Approximate coding, arithmetic coding, text compression, word-based model

This investigation was supported by the Australian Research Council and the Natural Sciences and Engineering Research Council of Canada. A preliminary presentation of this work was made at the 1995 IEEE Data Compression Conference.
Authors' addresses: A. Moffat, Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia; email: alistair@cs.mu.oz.au; R. M. Neal, Department of Statistics and Department of Computer Science, University of Toronto, Canada; email: radford@cs.utoronto.ca; I. H. Witten, Department of Computer Science, The University of Waikato, Hamilton, New Zealand; email: ihw@waikato.ac.nz.

1. INTRODUCTION

During its long gestation in the 1970's and early 1980's, arithmetic coding [Rissanen 1976; Rissanen and Langdon 1979; Rubin 1979; Rissanen and Langdon 1981; Langdon 1984] was widely regarded more as a curiosity than a practical coding technique. This was particularly true for applications where the alphabet has many symbols, as Huffman coding is usually reasonably effective in such cases [Manstetten 1992]. One factor that helped arithmetic coding gain the popularity it enjoys today was the publication of source code for a multisymbol arithmetic coder by Witten et al. [1987] in *Communications of the ACM*, which we refer to as the CACM implementation. One decade later, our understanding of arithmetic coding has further matured, and it is timely to review the components of that implementation and summarize the improvements that have emerged. We also describe a novel, previously unpublished, method for performing the underlying calculation needed for arithmetic coding. Software is available that implements the revised method.

The major improvements discussed in this article and implemented in the software are as follows:

—Enhanced models that allow higher-performance compression.

—A more modular division into modeling, estimation, and coding subsystems.

—Data structures that support arithmetic coding on large alphabets.

—Changes to the coding procedure that reduce the number of multiplications and divisions and which permit most of them to be done with low-precision arithmetic.

—Support for larger alphabet sizes and for more accurate representations of probabilities.

—A reformulation of the decoding procedure that greatly simplifies the decoding loop and improves decoding speed.

—An extension providing efficient coding for binary alphabets.

To motivate these changes, we examine in detail the needs of a word-based model for text compression. While not the best-performing model for text (see, for example, the compression results listed by Witten et al. [1994]), word-based modeling combines several attributes that test the capabilities and limitations of an arithmetic coding system.

The new implementation of arithmetic coding is both more versatile and more efficient than the CACM implementation. When combined with the same character-based model as the CACM implementation, the changes that we advocate result in up to two-fold speed improvements, with only a small loss of compression. This faster coding will also be of benefit in any other compression system that makes use of arithmetic coding (such as the block-sorting method of Burrows and Wheeler [1994]), though the percent-

age of overall improvement will of course vary depending on the time used in other operations and on the exact nature of the hardware being used.

The new implementation is written in C, and is publicly available through the Internet by anonymous ftp, at `munnari.oz.au`, directory `/pub/arith_coder`, file `arith_coder.tar.Z` or `arith_coder.tar.gz`. The original CACM package [Witten et al. 1987] is at `ftp.cpsc.ucalgary.ca` in file `/pub/projects/ar.cod/cacm-87.shar`. Software that implements the new method for performing the arithmetic coding calculations, but is otherwise similar to the CACM version, can be found at `ftp.cs.toronto.edu` in the directory `/pub/radford`, file `lowp_ac.shar`.

In the remainder of this introduction we give a brief review of arithmetic coding, describe modeling in general, and word-based models in particular, and discuss the attributes that the arithmetic coder must embody if it is to be usefully coupled with a word-based model. Section 2 examines the interface between the model and the coder, and explains how it can be designed to maximize their independence. Section 3 shows how accurate probability estimates can be maintained efficiently in an adaptive compression system, and describes an elegant structure due to Fenwick [1994]. In Section 4 the CACM arithmetic coder is reexamined, and our improvements are described. Section 5 analyzes the cost in compression effectiveness of using low precision for arithmetic operations. Low-precision operations may be desirable because they permit a shift/add implementation, details of which are discussed in Section 6. Section 7 describes the restricted coder for binary alphabets, and examines a simple binary model for text compression. Finally, Section 8 reviews the results and examines the various situations in which arithmetic coding should and should not be used.

1.1 The Idea of Arithmetic Coding

We now give a brief overview of arithmetic coding. For additional background the reader is referred to the work of Langdon [1984], Witten et al. [1987; 1994], Bell et al. [1990], and Howard and Vitter [1992; 1994].

Suppose we have a message composed of symbols over some finite alphabet. Suppose also that we know the probability of appearance of each of the distinct symbols, and seek to represent the message using the smallest possible number of bits. The well-known algorithm of Huffman [1952] takes a set of probabilities and calculates, for each symbol, a code word that unambiguously represents that symbol. Huffman's method is known to give the best possible representation when all of the symbols must be assigned discrete code words, each an integral number of bits long. The latter constraint in effect means that all symbol probabilities are approximated by negative powers of two. In an arithmetic coder the exact symbol probabilities are preserved, and so compression effectiveness is better, sometimes markedly so. On the other hand, use of exact probabilities means that it is not possible to maintain a discrete code word for each symbol; instead an overall code for the whole message must be calculated.

The mechanism that achieves this operates as follows. Suppose that p_i is the probability of the ith symbol in the alphabet, and that variables L and R are initialized to 0 and 1 respectively. Value L represents the smallest binary value consistent with a code representing the symbols processed so far, and R represents the product of the probabilities of those symbols. To encode the next symbol, which (say) is the jth of the alphabet, both L and R must be refined: L is replaced by $L + R\sum_{i=1}^{j-1} p_i$ and R is replaced by $R \cdot p_j$, preserving the relationship between L, R, and the symbols so far processed. At the end of the message, any binary value between L and $L + R$ will unambiguously specify the input message. We transmit the shortest such binary string, c. Because c must have at least $-\lceil \log_2 R \rceil$ and at most $-\lceil \log_2 R \rceil + 2$ bits of precision, the procedure is such that a symbol with probability p_j is effectively coded in approximately $-\log_2 p_j$ bits, thereby meeting the entropy-based lower bound of Shannon [1948].

This simple description has ignored a number of important problems. Specifically, the process described above requires extremely high precision arithmetic, since L and R must potentially be maintained to a million bits or more of precision. We may also wonder how best to calculate the cumulative probability distribution, and how best to perform the arithmetic. Solving these problems has been a major focus of past research, and of the work reported here.

1.2 The Role of the Model

The CACM implementation [Witten et al. 1987] included two driver programs that coupled the coder with a static zero-order character-based model, and with a corresponding adaptive model. These were supplied solely to complete a compression program, and were certainly not intended to represent excellent models for compression. Nevertheless, several people typed in the code from the printed page and compiled and executed it, only—much to our chagrin—to express disappointment that the new method was inferior to widely available benchmarks such as *Compress* [Hamaker 1988; Witten et al. 1988].

In fact, all that the CACM article professed to supply was a state-of-the art coder with two simple, illustrative, but mediocre models. One can think of the model as the "intelligence" of a compression scheme, which is responsible for deducing or interpolating the structure of the input, whereas the coder is the "engine room" of the compression system, which converts a probability distribution and a single symbol drawn from that distribution into a code [Bell et al. 1990; Rissanen and Langdon 1981]. In particular, the arithmetic coding "engine" is independent of any particular model. The example models in this article are meant purely to illustrate the demands placed upon the coder, and to allow different coders to be compared in a uniform test harness. Any improvements to the coder will

primarily yield better compression *efficiency*, that is, a reduction in time or space usage. Improvements to the model will yield improved compression *effectiveness*, that is, a decrease in the size of the encoded data. In this article we are primarily interested in compression efficiency, although we will also show that the approximations inherent in the revised coder do not result in any substantial loss of compression effectiveness.

The revised implementation does, however, include a more effective word-based model [Bentley et al. 1986; Horspool and Cormack 1992; Moffat 1989], which represents the stream as a sequence of words and nonwords rather than characters, with facilities for spelling out new words as they are encountered using a subsidiary character mode. Since the entropy of words in typical English text is around 10–15 bits each, and that of nonwords is around 2–3 bits, between 12 and 18 bits are required to encode a typical five-character word and the following one-character nonword. Large texts are therefore compressed to around 30% of their input size (2.4 bits per character)—a significant improvement over the 55%–60% (4.4–4.8 bits per character) achieved by zero-order character-based models of English. Witten et al. [1994] give results comparing character-based models with word-based models.

A word-based compressor can also be faster than a character-based one. Once a good vocabulary has been established, most words are coded as single symbols rather than as character sequences, reducing the number of time-consuming coding operations required.

What is more relevant, for the purposes of this article, is that word-based models illustrate several issues that do not arise with character-based models:

—An efficient data structure is needed to accumulate frequency counts for a large alphabet.

—Multiple coding contexts are necessary, for tokens, characters, and lengths, for both words and nonwords. Here, a coding context is a conditioning class on which the probability distribution for the next symbol is based.

—An escape mechanism is required to switch from one coding context to another.

—Data structures must be resizable because there is no a priori bound on alphabet size.

All of these issues are addressed in this article.

Arithmetic coding is most useful for adaptive compression, especially with large alphabets. This is the application envisioned in this article, and in the design of the new implementation. For static and semistatic coding, in which the probabilities used for encoding are fixed, Huffman coding is usually preferable to arithmetic coding [Bookstein and Klein 1993; Moffat and Turpin 1997; Moffat et al. 1994].

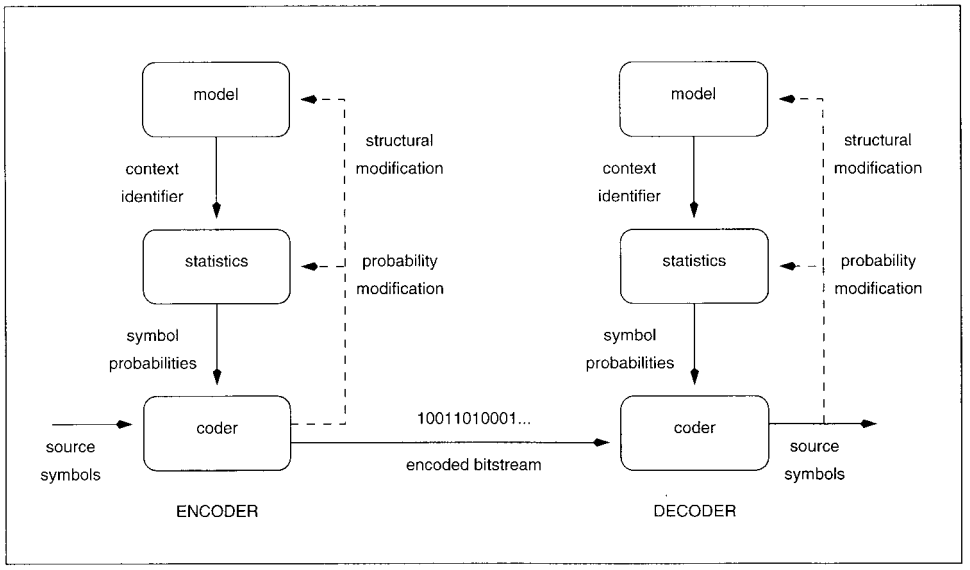

Fig. 1. Modeling, statistics, and coder modules.

2. COOPERATING MODULES

It is useful to divide the process of data compression into three logically disjoint activities: modeling, statistics-gathering, and coding. This separation was first articulated by Rissanen and Langdon [1981], although the CACM implementation of Witten et al. [1987] combined statistics gathering with modeling to give a two-way split. This section describes the three-way partition, which is reflected in our implementation by three cooperating modules. Examples are given that show how the interfaces are used.

2.1 Modeling, Statistics, and Coding

Of the three modules, the most visible is that which performs the modeling. Least visible is the coder. Between these two is the statistics module, which manipulates a data structure to record symbol frequency counts (or some other estimate of symbol probabilities). In detail, a statistics module used with an arithmetic coder must be able to report the cumulative frequency of all symbols earlier in the alphabet than a given symbol, and to record that this symbol has occurred one more time. Both the model and the coder are oblivious to the exact mechanism used to accomplish this: the model is unaware of the probability attached to each symbol; and the coder is unaware of symbol identifiers and the size of the alphabet. This organization is shown in Figure 1.

The CACM implementation [Witten et al. 1987] has just one interface level, reflecting the two-way modeling/coding division of activity. An array *cumfreq* containing cumulative frequencies and an actual symbol identifier *s* are passed from model to coder to achieve the transmission of each

Table I. Module Interface Functions

Module	Encoder	Decoder
Statistics	$C \leftarrow$ *create_context*()	$C \leftarrow$ *create_context*()
	encode(C, s)	$s \leftarrow$ *decode*(C)
	install_symbol(C, s)	*install_symbol*(C, s)
	purge_context(C)	*purge_context*(C)
Coder	*start_encode*()	*start_decode*()
	arithmetic_encode(l, h, t)	*target* $\leftarrow$ *decode_target*(t)
		arithmetic_decode(l, h, t)
	finish_encode()	*finish_decode*()

symbol. This forces both modules to use an array to maintain their information—an unnecessarily restrictive requirement. By divorcing the statistics module from both model and coder, any suitable data structure can be used to maintain the statistics. Section 3 below considers some alternatives.

The main routines required to interface the modules are listed in Table I. (The implementation includes several other routines, mainly for initializing and terminating compression.) The routine *install_symbol*() in both encoder and decoder has the same functionality as *encode*() except that no output bits are transmitted or consumed: its purpose is to allow contexts to be primed, as if text had preceded the beginning of the transmission.

The routine *purge_context* removes all records for that context, leaving it as if it had just been created. This allows "synchronization points" to be inserted in the compressed output using a *finish_encode* and *start_encode* pair, from which points adaptive decoding can commence without needing to process the entire compressed message. Purging model frequencies and inserting synchronization points does, of course, reduce the compression rate.

A zero-order character-based model requires just one context and relatively simple control structures, as shown in the psuedocode of Algorithm Zero-Order Character-Based (Figure 2), which closely corresponds to the adaptive model described by Witten et al. [1987]. A context C is created, *install_symbol*() is used to make each valid character available, and *encode*() is called once for each character in the input. The compressed message is terminated by an *end_of_message* symbol which has also been previously installed in C. The method of Algorithm Zero-Order Character-Based can easily be extended to a first-order character-based model using an array of contexts, one for each possible conditioning character. This would require considerably more memory, but would improve the compression effectiveness without impacting execution speed. Many other modifications are possible.

Complex models require the use of multiple contexts. The word-based model described in Section 1.2 uses six contexts: a zero-order context for words, a zero-order context for nonwords (sequences of spaces and punctu-

```
encode_file()
    /* Use a simple zero-order character model to compress a file. In both encoder
    and decoder, the context C is initialized to contain each valid character, plus
    the extra symbol end_of_message */
(1) Set C ← create_context(), and install_symbol(C, end_of_message)
(2) For each valid character c do
        install_symbol(C, c)
(3) start_encode()
(4) While characters remain do
        Set c ← read_one_character()
        encode(C, c)
(5) encode(C, end_of_message)
(6) finish_encode()


decode_file()
    /* Perform the corresponding decoding operations */
(1) Set C ← create_context(), and install_symbol(C, end_of_message)
(2) For each valid character c do
        install_symbol(C, c)
(3) start_decode()
(4) Repeat
        Set c ← decode(C)
        If c = end_of_message then
            break
        else
            write_one_character(c)
(5) finish_decode()
```

Fig. 2. Algorithm Zero-Order Character-Based.

ation), a zero-order character context for spelling out new words, a zero-order character context for spelling out new nonwords, and contexts for specifying the lengths of words and of nonwords. The encoder for that model is sketched as Algorithm Zero-Order Word-Based (Figure 3), except that for brevity the input is treated as a sequence of words rather than alternating "word, nonword" pairs and so only three contexts, denoted W, C, and L, are used. To cater for nonwords as well requires additional contexts W', C', and L', along with an outer loop that alternates between words and nonwords by using each set of contexts in turn. Note that Algorithm Zero-Order Word-Based assumes that the length of each word is bounded, so that context L can be initialized. In our implementation the actual definition of a word was "a string of at most 16 alphanumeric characters"; long symbols are handled by breaking them into shorter ones with zero-length opposite symbols between.

```
encode_file()
    /* Use read_one_word() to read a list of words. Compress the list adaptively,
    spelling out character-by-character each word the first time it is encountered,
    and establishing a word code to use for any subsequent appearances. Makes use
    of three context data structures: W, the set of valid word numbers; C, the set
    of valid characters used to compose words; and L, the set of legal word lengths.
    Structure D is a dictionary that converts a word into the corresponding word
    number, with word numbers assigned in order of first appearance in the input
    list */
(1) Set W ← create_context(), and install_symbol(W, end_of_message)
(2) Set D ← make_empty_dictionary()
(3) Set C ← create_context()
    For each valid character c do
        install_symbol(C, c)
(4) Set L ← create_context()
    For each valid length l do
        install_symbol(L, l)
(5) start_encode()
(6) While words remain do
        Set word ← read_one_word()
        Set w ← lookup_word_number(D, word)
        If (encode(W, w) = escape_transmitted) then
            encode(L, length(word))
            For each character c in word do
                encode(C, c)
            insert_into_dictionary(D, word)
            install_symbol(W, w)
(7) encode(W, end_of_message)
(8) finish_encode()
```

Fig. 3. Algorithm Zero-Order Word-Based.

The decoder, omitted in Figure 3, is the natural combination of the ideas presented in Algorithms Zero-Order Character-Based (Figure 2) and Zero-Order Word-Based (Figure 3).

2.2 Coding Novel Symbols

The character-based model of Algorithm Zero-Order Character-Based (Figure 2) codes at most 257 different symbols (256 different eight-bit characters plus the *end_of_message* symbol), all of which are made available in that context by explicit *install_symbol*() calls. In contrast to this, in the word-based model there is no limit to the number of possible symbols—the number of distinct word tokens in an input stream might be hundreds, thousands, or even millions. To cater for situations such as this in which the alphabet size is indeterminate, the function call *encode*(C, s) returns a flag *escape_transmitted* if the symbol s is not known in context C, or if, for some other reason, s has zero probability. In this event, the word is

encoded using the length and character models, and is then installed into the lexicon. As well as returning a flag to indicate a zero-probability symbol, the encoder must also explicitly transmit an escape code to the decoder so that the corresponding call *decode*(C) can similarly return an exception flag.

This raises the vexatious question as to what probability should be assigned to this escape code—the so-called *zero-frequency problem*. Of the methods described in the literature (summarized by Moffat et al. [1994]), we chose a modified version of method XC [Witten and Bell 1991] which we call method AX, for "approximate X." Method XC approximates the number of symbols of frequency zero by the number of symbols of frequency one. If t_1 symbols have appeared exactly once, symbol s_i has appeared c_i times, and $t = \Sigma_i c_i$ is the total number of symbols that have been coded to date, then the escape code probability p_{escape} is given by t_1/t and the probability of symbol s_i is estimated as $p_i = (1 - t_1/t) \cdot (c_i/t)$.

The drawback of method XC is the need for a two-stage coding process when the symbol is *not* novel—one step to transmit the information "not novel" (probability $1 - t_1/t$), and a second to indicate which nonnovel symbol it is (probability c_i/t). It is not feasible to calculate the exact probability for use in a single coding step, since the need to represent the product p_i restricts t to a very small range of values if overflow is to be avoided (see also the discussion in Section 4 below). Instead, for method AX we advocate

$$p_{escape} = (t_1 + 1)/(t + t_1 + 1)$$

$$p_i = c_i/(t + t_1 + 1).$$

The "+1" allows novel events to be represented even when there are no events of frequency one (in method XC this exception is handled by reverting to another method called "method C" in the literature); and t_1 is incorporated additively in the denominator by taking a first-order binomial approximation to $(1 - t_1/t)^{-1}$. With this method a single coding step suffices, as $t + t_1 + 1$ can be represented in roughly half the number of bits as the denominator t^2 required by method XC. The difference is crucial, since for flexibility we desire t to be similar in magnitude to the largest integer that can be represented in a machine word. The change distorts the probabilities slightly, but all zero-frequency methods are heuristics anyway, and the effect is small.

Once the escape code has been transmitted, the new token can be spelled out and added to the W context. Both encoder and decoder assign it the next available symbol number, so there is no need to specify an identifier.

2.3 Storage Management

An industrial-strength compression system must provide some mechanism to bound the amount of memory used during encoding and decoding. For

example, some compressors reclaim list items using a least-recently-used policy, so that the model structure continues to evolve when memory is exhausted. Others purge the model when memory is full, but retain a sliding window buffer of recent symbols so that a smaller replacement model can be rebuilt (using *install_symbol*) immediately. The exact mechanism used in any application will depend upon the needs of that application, in particular, on the amount of memory consumed by the structural model (Figure 1). Because of this dependence, the only facility we provide in our implementation is the routine *purge_context*(), which removes all records for that context, as if it had just been created. One rationalization for this abrupt "trash and start again" approach is that memory is typically so plentiful that trashing should be rare enough to cause little deterioration in the average compression rate. Furthermore, in the particular case of the word-based model the impact is softened by the fact that the underlying character-based models do not need to be purged, so transmission of novel words while the lexicon is rebuilt is less expensive than it was originally. In practice, purging of the lexicon in the word-based compressor is rare. A memory limit of one megabyte is ample to process texts with a vocabulary of about 30,000 distinct words, such as *The Complete Works of Shakespeare*.

3. DATA STRUCTURES

We now turn to the statistics module, which is responsible for managing the data structure that records cumulative symbol frequencies. In particular, a call *encode*(C, s) to encode symbol s in context C must result in a call to the coder module of *arithmetic_encode*($l_{C,s}$, $h_{C,s}$, t_C), where $l_{C,s}$ and $h_{C,s}$ are the cumulative frequency counts in context C of symbols respectively prior to and including s, according to some symbol ordering, and t_C is the total frequency of all symbols recorded in context C, possibly adjusted upward by the inclusion of a "count" for the escape symbol. To avoid excessive subscripting, we suppress explicit references to the context C whenever there is no ambiguity and use l_s, h_s, and t to denote the values that must be calculated.

3.1 Ordered List

In the CACM implementation [Witten et al. 1987] cumulative frequency counts are stored in an array, which is summed backward from the end of the alphabet. The alphabet is ordered by decreasing symbol frequency so as to place common symbols near the beginning. Symbols are indexed through a permutation vector, allowing l_s and h_s to be calculated very quickly. In order to increment the count of symbol s in the cumulative frequency array, a loop is used to increment each symbol to the left of—that is, of greater frequency than—s, thereby keeping the cumulative counts correct. To maintain this structure, $3n$ words of storage are needed for an alphabet of n symbols.

Although compact compared to the demands of adaptive Huffman coders, this mechanism is only effective for small alphabets, or for highly skewed distributions. An alternative structure has been described which bounds the number of loop iterations required to calculate the coding parameters for each symbol by the number of bits produced when that symbol is entropy-coded [Moffat 1990b]. The structure is asymptotically optimal for both uniform and skew distributions, taking time linear in the number of input symbols and output bits; it is also efficient in practice [Moffat et al. 1994]. About $4n$ words are required for an alphabet of n symbols.

3.2 Implicit Tree Structure

More recently, Fenwick [1994] proposed an implicit tree organization for storing an array of cumulative frequency counts that preserves the symbol ordering and so requires only n words to represent an n-symbol alphabet. Fenwick's method calculates and updates $l_s = \sum_{i=1}^{s-1} f_s$ and $h_s = \sum_{i=1}^{s} f_s$, where f_s is the observed frequency of symbol s, in $\Theta(\log n)$ time per symbol, irrespective of the underlying entropy of the distribution.[1] Although this is suboptimal for very skewed distributions (see, for example, the results of Moffat et al. [1994]), Fenwick's method works well for most of the probability distributions encountered in practice, and is employed in our revised arithmetic coding implementation because of its low space requirements.

Fenwick proposed that a single array $F[1 \ldots n]$ be used to record partial cumulative frequencies in a regular pattern based upon powers of two. At positions that are one less than a multiple of two the array records the corresponding symbol frequency; at positions that are two less than a multiple of four the value stored is the sum of the frequency for two symbols; at positions that are four less than a multiple of eight the value stored is the sum of the immediately preceding four symbol frequencies; and so on.

Cumulative counts can be updated in logarithmic time in this structure by using the following device to impose an implicit tree structure on the array. Suppose that s is some symbol number in the range $1 \le s \le n$. Define $backward(s)$ to be the integer obtained from s by subtracting the binary number corresponding to the rightmost one-bit in s. For example, the binary representation of 13 is 1101, and so $backward(13) = 1100_2 = 12$; $backward(12) = 1000_2 = 8$; and $backward(8) = 0$. Similarly, define $forward(s)$ to be $s + 2^i$ where i is again the position of the rightmost one-bit in s. For example, $forward(13) = 14$, $forward(14) = 16$, and $forward(16) = 32$. Both *backward* and *forward* can be readily implemented using bitwise operations if integers are represented in two's-complement

[1]We make use of the standard functional comparators $O()$, $\Theta()$, and $\Omega()$ to mean "grows no faster than," "grows at the same rate as," and "grows at least as quickly as," respectively. For precise definitions of these comparators see, for example, Cormen et al. [1990].

fenwick_get_frequency(C, s)
/* Returns the cumulative frequency of the symbols prior to and including s in the alphabet. Array F and alphabet size n are assumed to be components of C, the context data structure used. Once the symbol is located and h_s has been calculated, the symbol's frequency count is incremented by the amount *increment* (which is also a component of C) in the second while loop */
(1) Set $i \leftarrow s$ and $h_s \leftarrow 0$
(2) While $i \neq 0$ do
    Set $h_s \leftarrow h_s + F[i]$ and $i \leftarrow$ *backward*(i)
(3) Set $i \leftarrow s$
(4) While $i \leq n$ do
    Set $F[i] \leftarrow F[i] +$ *increment* and $i \leftarrow$ *forward*(i)
(5) Return h_s

Fig. 4. Algorithm Fenwick Get Frequency.

form:[2] *backward*(i) as either "$i - (i \text{ AND } -i)$" or "i AND $(i - 1)$"; and *forward*(i) as "$i + (i \text{ AND } -i)$", where AND is a bitwise logical "and" operator.

Array F is assigned values so that $F[s] = h_s - h_{backward}(s)$, where $h_0 = 0$. Table II shows an example set of f_s values, and the corresponding $F[i]$ values stored by Fenwick's method.

Given the array F, calculating h_s and incrementing the frequency count of symbol s is achieved by the loop shown in Algorithm Fenwick Get Frequency (Figure 4), which calculates h_s using F, where $1 \leq s \leq n$.

For example, h_{13} is calculated as $F[13] + F[12] + F[8]$, which by definition of F is the telescoping sum $(h_{13} - h_{12}) + (h_{12} - h_8) + (h_8 - h_0) = h_{13}$, as required. This is achieved in steps (1) and (2) of Algorithm Fenwick Get Frequency (Figure 4). It is also straightforward to increment a frequency count, using a process that updates the power-of-two aligned entries to the right of $F[s]$, as shown by steps (3) and (4) in Algorithm Fenwick Get Frequency (Figure 4). The value of variable *increment* can for the moment be taken to be one, but, as is explained below, may be larger than one if some further constraint—such as that the frequencies be partially normalized in some range—is also to be maintained. Since $l_s = h_{s-1}$, the same process can be used to calculate l_s—without, of course, steps (3) and (4). (In practice, the two values are calculated jointly, since much of the work is common.) The decoding process, which must determine a symbol s whose cumulative frequency range straddles an integer *target*, is sketched in Algorithm Fenwick Get Symbol in Figure 5 (see Fenwick [1994] for details). The important point to note is that all access and

[2]Slightly altered calculations are necessary for one's-complement or sign-magnitude representations [Fenwick 1994].

Table II. Example of Fenwick's Tree Structure

s	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f_s	1	1	1	4	3	5	2	3	6	5	4	1	1	9
$F[s]$	1	2	1	7	3	8	2	20	6	11	4	16	1	10
$l_s = \sum_{i=1}^{s-1} f_i$	0	1	2	3	7	10	15	17	20	26	31	35	36	37
$h_s = \sum_{i=1}^{s} f_i$	1	2	3	7	10	15	17	20	26	31	35	36	37	46

increment operations in both encoder and decoder are carried out in $O(\log n)$ time, using just a single array F.

The other operation required of the statistics module is scaling the frequency counts by some fixed multiplier (usually 1/2, as discussed further in Section 4). Fenwick [1994] gives a mechanism that requires $O(n \log n)$ time for an alphabet of n symbols; in our implementation we include an improved method that requires $O(n)$ time per call.

3.3 Dynamic Alphabets

Both of Algorithms Fenwick Get Frequency and Fenwick Get Symbol (Figures 4 and 5) make use of arrays rather than dynamic linked structures. If there is an a priori bound on alphabet size, the necessary space can be reserved in advance. This is the case in the character compressor of Algorithm Zero-Order Character-Based (Figure 2). However, with the adaptive word-based model it cannot be known in advance how many distinct words will appear in the input. It is therefore necessary to resize the array whenever the alphabet grows beyond its current size. This can conveniently be accomplished by allocating a new block of memory and copying the contents of the old block to it (e.g., using the C *realloc*() function). The expansion should be by a multiplicative factor k, so that if the current size is I, the next size is $I' = kI$, then $I'' = kI'$, and so on. When implemented this way the amortized copying cost is $O(k/(k - 1)) = O(1)$ per symbol and the space used, including all previous deallocated arrays—which because of fragmentation might not be reusable—is at most $nk/(k - 1) + nk$. This latter expression is minimized at $4n$ when $k = 2$. That is, a model with a dynamic alphabet uses as much as four times as much space as an equivalent static model. Using Fenwick's data structure for a dynamic alphabet of n symbols might therefore require as many as $4n$ words of memory.

Use of $k = 2$ is also helpful from a throughput perspective, since it allows the decoder loop (Algorithm Fenwick Get Symbol, Figure 5) to avoid the test "$s + mid \le n$" in each loop iteration.

3.4 Sparse Alphabets

So far we have assumed that the symbols are numbered from one up to the alphabet size. In some applications, however, they are scattered sparsely throughout some much larger range. For example, in many contexts of the

```
fenwick_get_symbol(C, target)
    /* Returns the symbol number that, if passed as argument to function
    fenwick_get_frequency, would return the least value that exceeds the parameter
    target. Array F, alphabet size n, and increment amount increment are assumed
    to be components of C, the context data structure used */
(1) Set s ← 0 and mid ← 2^⌊log n⌋
(2) While mid > 0 do
        If s + mid ≤ n and F[s + mid] ≤ target then
            Set target ← target − F[s + mid] and s ← s + mid
        Set mid ← mid/2
(3) Increment appropriate elements in F by increment, as for encoding (Algorithm
    FENWICK GET FREQUENCY, Figure 4)
(4) Return s + 1
```

Fig. 5. Algorithm Fenwick Get Symbol.

character-based PPM method [Cleary and Witten 1984] there will only be a handful of possible "next" characters—for example, for ordinary English text the only symbols appearing in the context of "qu" will be vowels, just five values in a range of 256. In such applications it is better to use an explicit search structure than to store the frequency counts in an array indexed by symbol number. For the PPM method, where most contexts have only a small alphabet, a linked list at each context is appropriate [Moffat 1990a]. For larger (but still sparse) alphabets, more complex structures are required. Possibilities include balanced search structures such as AVL trees and Splay trees [Jones 1988], both of which also allow a symbol to be located and l_s and h_s calculated in $O(\log n)$ time, or better.

4. PERFORMING THE ARITHMETIC

We turn now to the coding module. First, the process carried out by the CACM implementation [Witten et al. 1987] is introduced and critiqued. We then describe a reorganization that avoids most of those problems.

4.1 The CACM Coder

The basic coding step is performed by *arithmetic_encode*(l, h, t), which encodes a symbol implicitly assumed to have occurred $h - l$ times out of a total of t, and which is allocated the probability range $[l/t, h/t)$. The internal state of the coder is given by the values of two variables L and R, the current lower bound and range of the coding interval, and at each stage the message so far can be represented by any value in $[L, L + R)$.[3] The

[3]The description of the CACM coder in Witten et al. [1987] recorded the state of the coder using variables L and H, with $R = H - L + 1$ calculated explicitly at each coding step. Here we prefer the L, R description, which has also been used by Jiang [1995] (see also Moffat [1997]).

arithmetic_encode(l, h, t)
/* Arithmetically encode the range $[l/t, h/t)$. The state variables R and L are modified to reflect the new range, and then renormalized to restore the initial and final invariants $2^{b-2} < R \leq 2^{b-1}$, $0 \leq L < 2^b - 2^{b-2}$, and $L + R \leq 2^b$ */
(1) Set $T \leftarrow (R \text{ times } l) \text{ div } t$
(2) Set $L \leftarrow L + T$
(3) Set $R \leftarrow ((R \text{ times } h) \text{ div } t) - T$
(4) While $R \leq 2^{b-2}$ do
Renormalize R, adjust L, and output one bit

decode_target(t)
/* Returns an integer *target*, $0 \leq target < t$ that is guaranteed to lie in the range $[l, h)$ that was used at the corresponding call to *arithmetic_encode*() */
(1) Return $(((V - L + 1) \text{ times } t) - 1) \text{ div } R$

arithmetic_decode(l, h, t)
/* Adjusts the decoder's state variables L and R to reflect the changes made in the encoder during the corresponding call to *arithmetic_encode*() */
(1) Set $T \leftarrow (R \text{ times } l) \text{ div } t$
(2) Set $L \leftarrow L + T$
(3) Set $R \leftarrow ((R \text{ times } h) \text{ div } t) - T$
(4) While $R \leq 2^{b-2}$ do
Renormalize R, adjust L and V, and shift the next input bit into V

Fig. 6. Algorithm CACM Coder.

process followed by the CACM coder is illustrated (in anticipation of the new procedure, in a somewhat altered form) in Figure 6 as Algorithm CACM Coder. The values of L and R are represented by b-bit integers. L is initially 0, and takes on values between 0 and $2^b - 2^{b-2}$. R is initially 2^{b-1}, and takes on values between $2^{b-2} + 1$ and 2^{b-1}, inclusive. We assume that $0 \leq l < h \leq t$, and that t, which is the sum of the frequency counts of all symbols available in the specified context, lies in the range $t \leq 2^f$, where f is the number of bits used to store symbol frequencies, and may not exceed a value dictated by the choice of b and the word size of the machine being used.[4] Finally, note that if both b and f are fixed in advance (that is, at compilation time) many of the operations that involve them can be implemented more efficiently than if they are true run-time variables.

The basic operation of the encoder, in steps (1)–(3), is to reduce the $[L, L + R)$ interval to the subinterval that is represented by the new symbol,

[4]Strictly speaking, Algorithm CACM Coder as first presented [Witten et al. 1987] requires $t < 2^f$. If $t \leq 2^f$ is desired in Algorithm CACM Coder then the range on R must be slightly tightened to $2^{b-2} \leq R < 2^{b-1}$. This is not done here to allow consistency with the presentation of the improvements described below.

$[l/t, h/t)$, yielding a new interval $[L + Rl/t, L + Rh/t)$. The values of L and R are updated to reflect this new interval. The interval must be renormalized periodically to prevent it from becoming too small to be represented in the b bits of precision available. This is accomplished by step (4), which is considered in more detail in the next subsection along with the constraints on R implied by the assertions in the encoder and decoder.

Algorithm CACM Coder also illustrates the actions of the decoder, in which V is the current window extending b bits into the compressed bitstream. There are two functions, *decode_target*(t) and *arithmetic_decode* (l_s, h_s, t), corresponding to two basic steps that are required to decode a symbol. The first determines a *target* value based upon V and the value of t used in the corresponding encoding step. This is returned to the statistics module, which is responsible for searching its data structure and determining which symbol s corresponds to that integer—that is, finding the symbol s for which $l_s \leq \ target \ < h_s$. Then a call is made to *arithmetic_decode* (l_s, h_s, t), which carries out the bounds-scaling and bitshifting phases of the operation, mimicking the effect of the encoder.

4.2 Renormalization

To minimize loss of compression effectiveness due to imprecise division of code space, R should be kept as large as possible, and for correct operation, it must certainly be at least as large as t, the total count of symbols. This is done by maintaining R in the interval $2^{b-2} < R \leq 2^{b-1}$ prior to each coding step,[5] and making sure that $t \leq 2^f$, with $f \leq b - 2$. That is, R must be periodically renormalized and one or more bits output. The details of this process, and of the corresponding renormalization of L, R, and V at step (4) of *arithmetic_decode*(), are shown as Algorithm Encoder Renormalization in Figure 7 and Algorithm Decoder Renomalization in Figure 8.

In Algorithm Encoder Renormalization, when L and $L + R$ are both less than or equal to 2^{b-1}, a zero-bit can be output and L and R adjusted accordingly. Similarly, a one bit can be output when both are larger than or equal to 2^{b-1}. When $R \leq 2^{b-2}$ and neither of these two cases applies, the interval $[L, L + R)$ must straddle 2^{b-1}. In this third case the variable *bits_outstanding* is incremented, so that the next zero-bit or one-bit output will be followed by one more opposite bit. Function *bit_plus_follow*() checks for outstanding opposite bits each time it is called upon to output a bit of known polarity. The correctness of this arrangement is justified by Witten et al. [1987]; an alternative arrangement in which the effect of *bits*

[5]Again, the presentation here differs slightly from the CACM code [Witten et al. 1987] so as to allow a succinct description of the major improvements proposed below. In the CACM implementation R was bounded as $2^{b-2} < R \leq 2^b$, and renormalization was done whenever $L + R \leq 2^{b-1}$ or $2^{b-1} \leq L$ or $2^{b-2} \leq L \leq L + R \leq 2^b - 2^{b-2}$.

```
In arithmetic_encode()
    /* Reestablish the invariant on R, namely that 2^{b-2} < R ≤ 2^{b-1}. Each doubling
    of R corresponds to the output of one bit, either of known value, or of value
    opposite to the value of the next bit actually output */
(4) While R ≤ 2^{b-2} do
        If L + R ≤ 2^{b-1} then
            bit_plus_follow(0)
        else if 2^{b-1} ≤ L then
            bit_plus_follow(1)
            Set L ← L − 2^{b-1}
        else
            Set bits_outstanding ← bits_outstanding + 1 and L ← L − 2^{b-2}
        Set L ← 2L and R ← 2R


bit_plus_follow(x)
    /* Write the bit x (value 0 or 1) to the output bit stream, plus any outstanding
    following bits, which are known to be of opposite polarity */
(1) write_one_bit(x).
(2) While bits_outstanding > 0 do
        write_one_bit(1 − x)
        Set bits_outstanding ← bits_outstanding − 1
```

Fig. 7. Algorithm Encoder Renormalization.

```
In arithmetic_decode()
    /* Mimic the renormalization undertaken in the encoder (Figure 7) at the cor-
    responding stage. Function read_one_bit() is assumed to return either 0 or 1,
    being the value of the next unprocessed bit in the input stream */
(4) While R ≤ 2^{b-2} do
        If L + R ≤ 2^{b-1} then
            /* no action */
        else if 2^{b-1} ≤ L then
            Set L ← L − 2^{b-1} and V ← V − 2^{b-1}
        else
            Set L ← L − 2^{b-2} and V ← V − 2^{b-2}
        Set L ← 2L, R ← 2R, and V ← 2V + read_one_bit()
```

Fig. 8. Algorithm Decoder Renomalization.

_outstanding is achieved by carry propagation in an output buffer is also possible [Jiang 1995; Langdon 1984]. Schindler [1998] describes a further method that allows byte-by-byte output from the renormalisation loop, based upon the loop guard $R < 2^{b-9}$ that delays any action until eight bits are determined. Bytes that are all ones must be buffered until a byte containing a zero-bit is produced, the logical equivalent of *bits _outstanding*. The elimination of bit-at-a-time processing makes execution

faster, but has the drawback of restricting the range in which symbol frequency counts must be maintained.

All of the output operations in Figure 7 result in R being doubled, so after some number of output operations R will again be in range, ready for the next coding step. Indeed, a symbol coded according to parameters l, h, and t must automatically cause either $\lfloor -\log((h - l)/t) \rfloor$ bits or $\lceil -\log((h - l)/t) \rceil$ bits to be generated in the renormalization loop. Any remaining information content of the symbol is recorded by the altered coding state, reflected in the new values of L and R.

After each symbol is identified, the decoder must adjust its interval in the same way. Furthermore, as each bit is processed by moving it out of the high-order end of V, a new bit is shifted into the low-order end.

4.3 Drawbacks

There are several drawbacks to the CACM implementation. The first is that the multiplication and division operations (marked as "times" and "div" in Algorithm CACM Coder in Figure 6, to make them obvious) are slow on some architectures compared to addition, subtraction, and shifting operations. To *arithmetic_encode* a symbol requires two multiplications and two divisions; and to *decode_target* plus *arithmetic_decode* requires three of each.

The second problem concerns arithmetic overflow. Suppose that the machine being used supports w-bit arithmetic. For example, on most current workstations, $w = 32$. Then, if overflow is to be avoided at steps (1) and (3) of *arithmetic_encode*, at which "R times l" and "R times h" are calculated, we must have $2^w - 1 \geq 2^{b-1}t$, which is satisfied when $w \geq b - 1 + f$ and $t < 2^f$. Moreover, if underflow and $R = 0$ is not to occur after step 3, $R \geq t$ is required, that is, $2^{b-2} \geq 2^f$. In combination, these two constraints can only be satisfied when $w \geq 2f + 1$. Hence, for $w = 32$ the largest value of f that can be supported is 15, and the sum of the frequency counts in any given context must not exceed 32,767. For word-based compression, and other models with a large alphabet, this is a severe restriction indeed. The advent of 64-bit computers alleviates the problem in some environments, and other work-arounds are also possible. Nevertheless, a more general solution is worthwhile.

Finally, there is some small risk of arithmetic overflow in the calculation of *bits_outstanding*. Imagine a static model of three equiprobable symbols A, B, and C; and suppose a file of a billion or so B's is to be coded. Then *bits_outstanding* will increase without any bits ever actually being emitted. The risk of overflow is small—the probability of 2^{32} bits all the same polarity is, in an adaptive model, of the order of one in $2^{2^{32}}$; nevertheless, it is possible. There are three ways this problem can be handled. The first, and most economical, is to monitor *bits_outstanding* in the encoder, and should it become very large, write an error message and abort the program.

If this is unpalatable—if the compression process must be robust to the extent that even a one in a billion chance of failure cannot be tolerated—then the decoder should be modified to be aware of the current value of *bits_outstanding*, thereby allowing both encoder and decoder to synchronously make use of a *finish_encode* and *start_encode* pair to flush outstanding bits and reset the state of the coder whenever *bits_outstanding* approaches its limiting value. This does, however, have the disadvantage of somewhat slowing the decoder, and in our implementation we adopt the first approach. The third alternative is to suppose an a priori upper bound on the length of any file that will be processed, and use extended precision arithmetic for *bits_outstanding* to ensure that overflow is impossible. Again, this problem is greatly alleviated by the use of 64-bit architectures.

4.4 An Improved Coder

Algorithm Improved Coder (Figure 9) describes our new approach to arithmetic coding. The major change is in the order of calculation at steps (1) and (2) of *arithmetic_encode*(). By doing the division first, one multiplicative operation is immediately saved. The trade-off is in compression performance, since rounding the ratio $r = R/t$ to an integer adversely affects compression effectiveness. In step (3), in the case where the top symbol is the one being encoded ($h = t$), care is taken to ensure that the new top-point of the range is the same as it was before, that is, that $L + R$ remains unchanged. This minimizes the loss by ensuring that the whole range R is allocated to one symbol or another, but nevertheless some inaccuracy arises. The amount of inaccuracy, and techniques that reduce it, are discussed below in Section 5. Two multiplicative operations are saved in the decoder, which is also described in Algorithm Improved Coder. Note that the common value "r times l" need not be recalculated when $h = t$, saving a further operation in both encoder and decoder when the remainder of the range is being allocated to the last symbol in the alphabet. As we shall see below, this is a nontrivial saving, since it is usually desirable to place the most probable symbol at the top of the alphabet.

The rearrangement of the multiplicative operations also allows larger frequency counts to be manipulated, and this a second important benefit of the change. Now the only constraint is that $t \leq R$, that is, that $f \leq b - 2$ and that $t \leq 2^f$. The revised coder can operate effectively with $b = 32$ and $f = 30$, and streams of up to $2^{30} \approx$ one billion symbols can be processed before count scaling is required.

As an independent change, we also advocate a reorganization of the decoder. In Algorithm Improved Coder (Figure 9) we make use of the transformation $D = V - L$. This allows a substantially simpler decoder renormalization loop (step (3) of function *arithmetic_decode*()), since in Algorithm Decoder Renomalization (Figure 8) L and V undergo identical

arithmetic_encode(l, h, t)
/* Arithmetically encode the range $[l/t, h/t)$ using low-precision arithmetic. The state variables R and L are modified to reflect the new range, and then renormalized to restore the initial and final invariants $2^{b-2} < R \leq 2^{b-1}$, $0 \leq L < 2^b - 2^{b-2}$, and $L + R \leq 2^b$ */
(1) Set $r \leftarrow R$ div t
(2) Set $L \leftarrow L + r$ times l
(3) If $h < t$ then
set $R \leftarrow r$ times $(h - l)$
else
set $R \leftarrow R - r$ times l
(4) While $R \leq 2^{b-2}$ do
Use Algorithm ENCODER RENORMALIZATION (Figure 7) to renormalize R, adjust L, and output one bit

decode_target(t)
/* Returns an integer *target*, $0 \leq$ *target* $< t$ that is guaranteed to lie in the range $[l, h)$ that was used at the corresponding call to *arithmetic_encode*() */
(1) Set $r \leftarrow R$ div t
(2) Return $(\min\{t - 1, D \text{ div } r\})$

arithmetic_decode(l, h, t)
/* Adjusts the decoder's state variables R and D to reflect the changes made in the encoder during the corresponding call to *arithmetic_encode*(). Note that, compared with Algorithm CACM CODER (Figure 6), the transformation $D = V - L$ is used. It is also assumed that r has been set by a prior call to *decode_target*() */
(1) Set $D \leftarrow D - r$ times l
(2) If $h < t$ then
set $R \leftarrow r$ times $(h - l)$
else
set $R \leftarrow R - r$ times l
(3) While $R \leq 2^{b-2}$ do
Set $R \leftarrow 2R$ and $D \leftarrow 2D +$ *read_one_bit*()

Fig. 9. Algorithm Improved Coder.

changes in each of the three cases of the renormalization loop. A simpler loop, in turn, means faster execution, and this change alone improves decoding speed by approximately 10%. Note that use of this transformation is not linked in any way to the changes to the calculation of R at steps (1) and (2) of *arithmetic_encode*().

Finally, we reiterate the changes already assumed in the presentation of Algorithm CACM Coder: the use of L and R to describe the coder's state rather than L and H; and the use of the overriding guard $R \leq 2^{b-2}$ on the renormalization loop.

4.5 Initialization and Termination

We now consider the initialization and termination routines needed to support the coding functions of Algorithm Improved Coder. The initialization of R differs slightly from the CACM code, which had the effect of initializing R to 2^b. Here we use 2^{b-1} instead, to ensure that the asserted preconditions of *arithmetic_encode*() and *arithmetic_decode*() are always correct. The change means that the first output bit from the coder will always be a zero, introducing a small redundancy into the output stream. This extra output bit can either be filtered out by the encoder and assumed by the decoder, or left in the bitstream and inspected by the decoder as a one-bit validity check. Our source code implementation allows either of these alternatives. Use will be made of the assertions governing R in Section 6.

Termination involves the encoder calculating and transmitting sufficiently many bits that the final interval can be identified by the decoder unambiguously, irrespective of what other bits follow on from there. For example, the next bit in the input stream might represent—completely uncompressed—whether there is another compressed stream to be decoded, or might be another coded message started from a new initial model. There are several ways in which this termination can be done. In our implementation we offer two alternatives.

The first alternative is to simply emit the b bits that describe the current value of L. While less than optimal in terms of compression, this approach means that there is no need to push any bits in the window (variable D) back to the input bitstream if another message (compressed or uncompressed) follows in the immediately next bit of the input file. This alternative also allows the $D = V - L$ transformation to be fully exploited, since there is never any need for the decoder to maintain a value for V or L.

It does, however, cause as many as $b - 1$ unnecessary bits to be emitted. In a "frugal bits" environment in which decoding speed is less important than economy of transmission, a minimal number of further output bits can be generated so that, regardless of what bits are allowed to follow these final bits, the message will be decoded correctly. This is achieved with a loop that first tests the smallest one-bit-of-precision number greater than or equal to L, then the smallest two-bits-of-precision number greater than or equal to L, and so on. When a value is found for which any possible successor bits still yield a value less than $L + R$, the loop exits, and those bits are emitted. This is the second alternative provided in our implementation; and for details the reader is referred to the source code. In this case the need for both L and R to be known means that the decoder must maintain a value for either L or V during the renormalization process, partially negating the speed advantage of the $D = V - L$ transformation.

With the constraints on L and R enforced by the various loop guards this second approach requires at least one (when either $L = 0$ or $L = 2^{b-1}$ and

```
start_encode()
    /* Initialize the encoder's state variables */
(1) Set L ← 0, R ← 2^(b−1), and bits_outstanding ← 0

finish_encode()
    /* Flush the encoder so that all information is in the output bit stream */
(1) Use bit_plus_follow() to transmit the b bits of L

start_decode()
    /* Initialize the decoder's state variables */
(1) Set R ← 2^(b−1) and D ← 0
(2) For i ← 1 to b do
        Set D ← 2D + read_one_bit()

finish_decode()
    /* Push any unconsumed bits back to the input bit stream. For the version of
    finish_encode() described here, no action is necessary */
```

Fig. 10. Algorithm Support Routines.

$R = 2^{b-1}$) and at most three final bits to be emitted. Readers very familiar with the CACM implementation may recall that for it, two disambiguating bits were always sufficient: the difference is caused by the altered renormalization loop guards. The extra third bit sometimes output by the new implementation is one that would already have been output in the original implementation, and the net effect is identical.

These functions are described in Figure 10 as Algorithm Support Routines. Note that the CACM interface [Witten et al. 1987] has been extended slightly by including function *finish_decode*(); when function *finish_encode*() emits L (which is the case described in Algorithm Support Routines) *finish_decode*() does nothing. On the other hand, in a "frugal bits" environment *finish_decode*() must reconstitute values for L and V, repeat the calculations performed in *finish_encode*() as to how many termination bits should be emitted, and then push the correct number of residual bits from V back to the front of the input bitstream so that they are available to whatever reading process is expecting to find them there.

4.6 Partial Normalization of Frequency Counts

In Section 6, we will describe how the new organization of the coding operations allows multiplicative operations to be replaced by shifts and adds. To support this development, it is necessary to maintain the fre-

```
In create_context()
(1) Set increment_C ← 2^f

encode(C, s)
    /* Use the context data structure C to determine the observed probability of
    symbol s; code s using the arithmetic coder; update the observed probability of
    symbol s; and renormalize the frequency counts for context C if required */
(1) Use the statistics data structure (Algorithm FENWICK GET FREQUENCY) to de-
    termine l_{C,s}, h_{C,s}, and t_C
(2) arithmetic_encode(l_{C,s}, h_{C,s}, t_C)
(3) Adjust the data structure so that increment_C is added to the frequency of symbol
    s (also described in Algorithm FENWICK GET FREQUENCY)
(4) Set t_C ← t_C + increment_C
(5) if t_C > 2^f then
        Halve each frequency count stored in C, making sure each is at least 1
        Set t_C ← the new sum of the frequency counts, and
            increment_C ← (increment_C + 1) div 2
```

Fig. 11. Algorithm Count Scaling.

quency counts in partially normalized form, so that $2^{f-1} < t \leq 2^f$.[6] In an adaptive coder this is achieved by requiring the statistics module to scale the counts for context C by a factor of 1/2 whenever $t_C > 2^f$. To ensure that the statistics accumulation remains fair, the increment used during frequency count updates should also be scaled, but bounded below by one. Algorithm Count Scaling (Figure 11) illustrates the process of coding a symbol and then incrementing its frequency count, including scaling to maintain $2^{f-1} < t \leq 2^f$.

The $t \leq 2^f$ bound on the frequency counts means that the inaccuracy of the rounding process is greatest when f is large relative to b, and when f is small the arithmetic is more precise. For example, when $f = b - 2$ then r must be either 1, 2, or 3; and, in general, $2^{b-f-2} \leq r < 2^{b-f}$.

It is also possible to constrain t even more tightly by maintaining $2^{f-1} < t \leq 2^{f+1}/3$. When t exceeds the upper limit each frequency count is multiplied by 3/4, which can be achieved with two shifts and an addition. The analysis of the next two sections demonstrates that the smaller the value of t, the lower the rounding inefficiency; a more constrained scaling process thus reduces the inefficiency for a given lower limit 2^{f-1} on the frequencies that sum to t. The trade-off is in increased execution time, as scaling will take place approximately three times more often.

[6]Partial normalization is not necessary if multiplications and divisions are done the usual way. However, by always performing these normalizations, one preserves the option of ultimately using a shift/add decoder, even if the encoding was done using normal multiplies and divides.

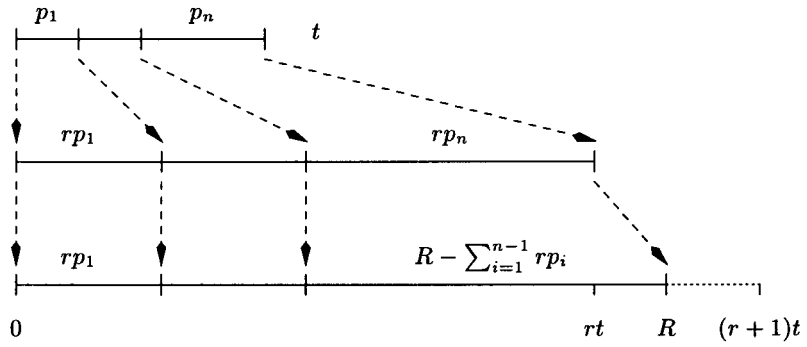

Fig. 12. Approximate calculation and truncation excess.

5. ANALYSIS

In this section we bound the loss of compression effectiveness caused by the use of approximate symbol probabilities.

5.1 Worst-Case Analysis

Consider the compression loss caused by the use of approximate arithmetic. Call the last symbol in the alphabet s_n, and its probability p_n. The effect of the rounding in *arithmetic_encode*() in Algorithm Improved Coder (Figure 9) is to allocate too much code space to s_n at the expense of the remaining symbols. In the worst case, instead of the complete range R, only a fraction $r/(r + 1)$ of it is apportioned fairly between the symbols according to their probabilities. Figure 12 illustrates the situation.

The excess, which can be as much as $1/(r + 1)$, is allocated to s_n over and above its rightful share, giving it a total of as much as $p_n r/(r + 1) + 1/(r + 1)$. The consequent increase in per-symbol code length can be calculated by considering the two events s_n and *not* s_n, because the symbols $s_1, s_2, \cdots, s_{n-1}$ receive the correct proportion of code space within that allocated to *not* s_n. This excess E is as follows (all logarithms are binary):

$$\begin{aligned} E &= (p_n \log p_n + (1 - p_n) \log (1 - p_n)) \\ &\quad - \left(p_n \log \frac{1 + p_n r}{r + 1} + (1 - p_n) \log \frac{r - p_n r}{r + 1} \right) \\ &= p_n \log \frac{p_n (r + 1)}{1 + p_n r} + (1 - p_n) \log \frac{r + 1}{r}. \end{aligned}$$

The error bound is proportionately at its greatest when $r_* = 2^{b-f-2}$, which is the smallest value of R/t permitted by the constraints on R and t, because at this point the ratio $(r + 1)/r$ is maximized. As p_n tends to one,

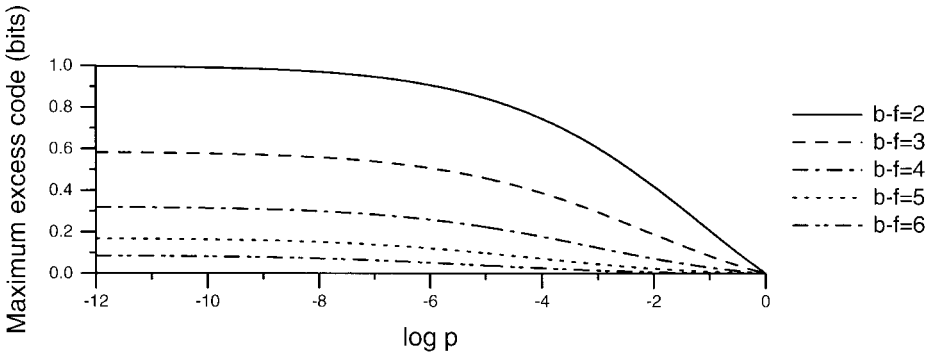

Fig. 13. Upper bound on excess bits per symbol, as a function of p_n and $b - f$.

E tends to zero, and as p_n tends to zero, E tends to $E_* = \log[(r_* + 1)/r_*] \approx \log e/2^{b-f-2}$. Figure 13 plots E as a function of p_n for various values of $b - f$; each of the curves approaches the corresponding value of E_* at the left-hand side of the graph.

The analysis shows that, for any given symbol probability distribution, the loss of compression effectiveness can be minimized by making p_n the largest probability in the distribution; that is, by arranging the symbol ordering so that it is the most probable symbol (MPS) that is allocated the extra probability created during the rounding process. If s_n is the MPS, the entropy of the alphabet must be at least $-\lfloor 1/p_n \rfloor p_n \log p_n - (1 - \lfloor 1/p_n \rfloor p_n) \log (1 - \lfloor 1/p_n \rfloor p_n)$, a lower bound achieved only when as many as possible of the other symbols also have this probability.[7] Figure 14 shows the upper bound on the relative inefficiency obtained using this lower bound on the entropy. We see, for example, that for $b - f \geq 6$ the relative inefficiency is always less than 1%.

5.2 Average-Case Analysis

These compression degradations have been calculated assuming only that the source is true to the probabilities being used. The assumption that R always takes on its minimal value is unduly pessimistic, however, as R will in practice vary in a random manner within its permitted range.

To exploit this variation it is tempting to assume that R is randomly distributed uniformly between its lower and upper bounding values—that is, that each value z in the range $2^{b-2} < z \leq 2^{b-1}$ occurs with probability $2^{-(b-2)}$. Unfortunately, the assumption of uniformity is too optimistic—the correct probability density function is proportional to $1/R$, with the probability $Pr(R)$ of R being approximately $(\log e)/R$. (Recall that all logs in this article are to base 2.)

[7]It is assumed here that $x \log_2 x = 0$ when $x = 0$.

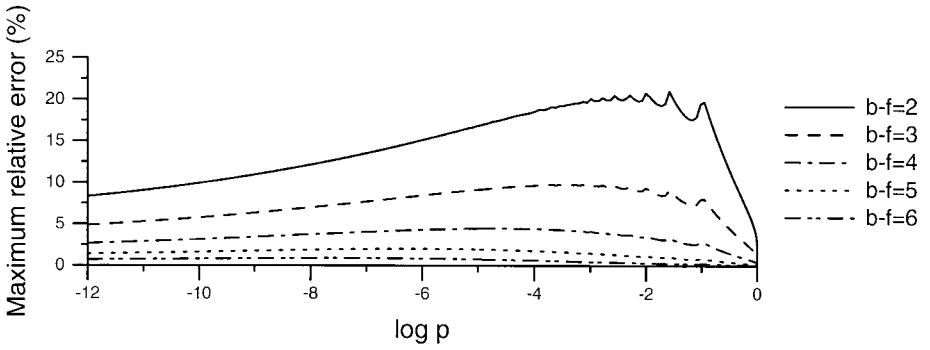

Fig. 14. Upper bound on relative inefficiency, percentage of entropy as a function of p_n and $b - f$, assuming s_n is the most probable symbol.

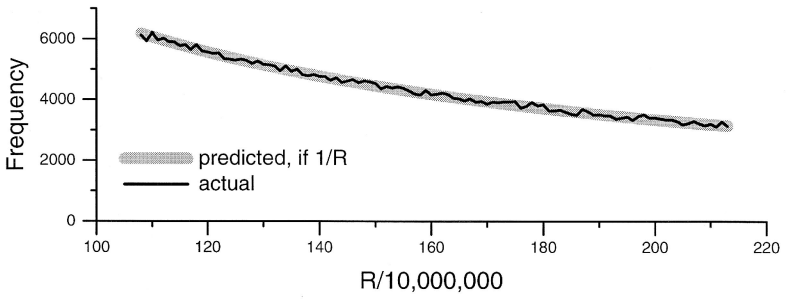

Fig. 15. Distribution of $\lfloor R/10{,}000{,}000 \rfloor$, accumulated over 465,349 coding steps with $b = 32$.

To see why this is so, consider the quantity $I = -\log(R/2^{b-1})$, which is the amount of information from previously encoded symbols that has not yet been transmitted to the decoder. At the end of an encoding step, I will be in the interval [0,1). When a symbol with probability p_i is encoded, I increases by approximately $-\log p_i$. When a bit is transmitted, I decreases by 1. If symbol probabilities vary randomly to at least some small degree, the value of I at any given time will also be random.

Unless symbol probabilities are highly restricted (for example, all $p_i = 1/2$, always), I will vary over its full range and can be analyzed as a continuous random variable (even though it of course discrete, given finite precision arithmetic). In particular, I will usually have a distribution that is uniform over [0,1). This distribution is invariant with respect to the operation of encoding a symbol with any given probability, p_i, and hence also with respect to the encoding of a symbol with p_i selected according to any distribution. When many p_i are used, the uniform distribution will generally be the only one that is invariant, and will therefore be the equilibrium distribution reached after many encoding operations.

This uniform distribution for I, with density $Pr(I) = 1$, can be transformed to find the distribution for $R = 2^{b-1-I} = e^{(b-1-I)/\log e}$, using the general rule for transforming densities:

$$Pr(R) = \frac{Pr(I)}{|dR / dI|} = \frac{\log e}{e^{(b-1-I)/\log e}} = \frac{\log e}{R}$$

The probabilities for the actual discrete distribution are the same as the above density, since the interval between discrete values is one.

Figure 15 shows the measured distribution of R over a compression run involving 465,349 calls to *arithmetic_encode*() with $b = 32$ and thus $1{,}073{,}741{,}824 < R \leq 2{,}147{,}483{,}648$. The wide gray line is the behavior that would be expected if the probability of R taking on any particular value was proportional to $1/R$; the crisp black line is the observed frequency of $\lfloor R/10^7 \rfloor$, that is, R when quantized into buckets of 10 million. As can be seen, the distribution is certainly not uniform, and the $1/R$ curve fits the observed distribution very well. Similar results were observed for other input files and other models.

Suppose then that R takes on values in the range $[2^{b-2}, 2^{b-1})$ with probability density proportional to $1/R$. If $t = 2^f$, then after the division takes place the "true" quotient $x = R / t$ is similarly random in $[2^{b-f-2}, 2^{b-f-1})$, again with a $1/R$ probability distribution. The true x is then approximated in the coder by $r = \lfloor x \rfloor$. Hence, the maximum expected coding excess, $\bar{E}_*$, which occurs as p_n tends to zero, is given by

$$\bar{E}_* = \frac{1}{K} \cdot \sum_{R=2^{b-2}}^{2^{b-1}-1} \frac{1}{R} \cdot \log \frac{R/2^f}{\lfloor R/2^f \rfloor},$$

where

$$K = \sum_{R=2^{b-2}}^{2^{b-1}-1} \frac{1}{R} \approx \frac{1}{\log e}.$$

When $b = 20$ and $f = 18$ this evaluates to 0.500; when $b = 20$ and $f = 17$, it is 0.257; and, in general, it is about 40% of the pessimal values plotted in Figures 13 and 14. Table III lists the worst-case and average-case error bounds for various values of $b - f$ when $b = 20$. Note that the exact value of b has very little effect on these values unless b is small.

If t may also be assumed to be random in its range then the expected inefficiency is further reduced, since the quotient R/t becomes proportionately larger. Note, however, that t is an attribute of the model rather than the coder and so no particular distribution may in general be assumed. All of the values in Table III assume a pessimal value of $t = 2^f$.

Table III. Limiting Worst-Case and Average-Case Errors, Bits per Symbol, as $p_n \to 0$

$b - f$	Worst-Case Error (bits per symbol) E_*	Average-Case Error (bits per symbol) $\bar{E}_*$
2	1.000	0.500
4	0.322	0.130
6	0.087	0.033
8	0.022	0.008
10	0.006	0.002

5.3 Practical Use

From Figure 13 it can be seen that if minimization of coding inefficiency is important, the MPS should be identified in the statistics module, and maintained at the top of the alphabet. Our implementation of Fenwick's structure does just that. The alternative is to simply use a large value of $b - f$, minimizing the compression loss by using better approximations.

Table III also allows estimation of the relative effectiveness irrespective of p_n, provided only that the entropy of the distribution being coded can be approximated. For example, the zero-order entropy of a word frequency distribution usually exceeds 10 bits per symbol and so even if p_n is the smallest probability, the average error is less than 5% for $b - f = 2$ and about 0.3% when $b - f = 6$. Similarly, the entropy of the nonword distribution is usually greater than 2 bits per symbol, and so with $b - f = 6$ the combined inefficiency when coding the word-based model described earlier in this article is not more than $(0.03 + 0.03)/(10 + 2) \approx 0.5\%$.

Finally, this reorganization of the coding operations permits most of the multiplications and divisions to be performed with a small number of shifts and adds, provided the symbol frequencies are maintained partially normalized. On some architectures the use of shift/add calculation will provide substantial additional improvement in compression throughput. Methods for doing these calculations are discussed in Section 6.

5.4 Related Work

Other multisymbol approximate arithmetic coding approaches have been proposed in the literature. Rissanen and Mohiuddin [1989] describe and analyze a method not dissimilar to the one described here in the special case when $b - f = 2$, but with r taken to be either 1 or 2, which makes the decoder straightforward. They stipulate a slightly different normalization regime for R, and also suppose that the symbol frequencies are shifted at each modeling step rather than maintained partially normalized as is the case here. Points in common (compared to the case when $b - f = 2$) are the use of an approximation to r, and the observation that the coding error is minimized if the MPS is allocated the excess probability caused by the rounding process. The distinguishing feature of our proposal is the ability

to vary $b - f$, allowing a smooth tradeoff between time requirements (or hardware complexity) and compression effectiveness. Chevion et al. [1991] (see also Feygin et al. [1994]) analyze another method; they approximate R by a value $R' = 2^{b-2} + 2^{b-2-L}$, which means that the multiplications can be performed using two shift/add operations. This method achieves good compression effectiveness, but as described requires fully normalized probabilities, possibly making it less suitable for general purpose adaptive compression.

One important point worth noting is that this previous work has assumed for the purpose of average case analysis that R is distributed uniformly in its range. As demonstrated above, this is not correct, so the average case error bounds of these papers require revision.

6. REMOVING MULTIPLICATIVE OPERATIONS

In this section we further refine the description of the improved method, showing how it can be implemented with a small number of additive operations, and giving experimental results showing it to be fast in practice.

6.1 Shift-Add Implementation

The description of Algorithm Improved Coder allows a further improvement—the use of additive and logical (shift and mask) operations to calculate the results of the remaining multiplications and divisions. Since the quotient r has at most $b - f$ bits of precision, when $b - f$ is small it may be faster to use shift/add operations to calculate $l \times r$ and $h \times r$, and shift/subtract to effect "div" operations, than it is to use the hardware multiply and divide operations, which provide 32 or 64 bits of precision and are often implemented in firmware using shift/add operations anyway. The loop in Algorithm Shift/Add Coder (Figure 16), for example, achieves the same adjustment of L and R as do steps (1), (2), and (3) of Algorithm Improved Coder.

Using this approach, the three multiplicative operations in the encoder can be replaced by $O(b - f)$ additive and logical operations.

A similar rearrangement can be effected in *arithmetic_decode*(). If it is assumed that r is already known, then $l \times r$ and $h \times r$ can be calculated in the same manner as in the encoder. There is, however, a problem with *decode_target*(). Since the *target* value that is calculated once r is determined has f bits of precision, it would appear that f shift/subtract loop iterations are required, making $O(b)$ iterations in total. In fact, this need for an f-bit target is an artifact of the modular approach espoused here, and can be reduced using the following mechanism.

Suppose that r is calculated by *decode_target*() in $O(b - f)$ shift/subtract iterations, but that no further computation is performed. Let *target* be the target value that would be computed were the computation completed. To determine the symbol number that corresponds to *target*, the statistics

```
arithmetic_encode(l, h, t)
    /* Arithmetically encode the range [l/t, h/t) using shift-add calculation rather
    than multiplicative operations. Variables rl and rh end up with the values
    (R div t) times l and (R div t) times h respectively that are calculated in Algo-
    rithm IMPROVED CODER in Figure 9; and L and R end up with the same values
    as they do in Algorithm IMPROVED CODER */
(1) Set rl ← 0, rh ← 0, numerator ← R, and denominator ← t × 2^(b−f−1)
(2) for nbits ← b − f − 1 downto 0 do
        Set rl ← 2 × rl and rh ← 2 × rh
        if numerator ≥ denominator then
            Set rl ← rl + l, rh ← rh + h, and
                numerator ← numerator − denominator
        Set numerator ← 2 × numerator
(3) Set L ← L + rl
(4) If h < t then
        set R ← rh − rl
    else
        set R ← R − rl
```

Fig. 16. Algorithm Shift/Add Coder.

module makes use of a further function *less_than_target*(*value*) that returns true if *value* $<$ *target*, and false otherwise. All that is necessary now is to write *less_than_target*(*value*) in such a way that it incrementally calculates sufficiently many more significant bits of *target* that the outcome of the comparison can be known. That is, *less_than_target*(*value*) determines only sufficiently many bits of *target* for it to return the correct result of the comparison. The statistics module can then branch to the appropriate alternative, and will make further calls to *less_than_target*() until a symbol number is known. Once the symbol number is known no further bits of *target* are required.

Clearly, all f bits of *target* might in some situations need to be evaluated. But on average, fewer bits suffice. Suppose that the symbol in question has probability p_i. Then the bounding cumulative probabilities of symbol i will differ in magnitude by tp_i, which means they will differ in about $\log t + \log p_i$ of their low order bits. Since *target* is essentially a uniformly random value between these two bounds, it can in turn be expected to differ from both bounds on average by $\log t + \log p_i - 1$ bits. If the calculation of bits of *target* from most-significant to least-significant is halted as soon as *target* differs from both bounds, then on average $\log t - (\log t + \log p_i - 1)$ bits must be calculated, that is, $1 + \log(1/p_i)$. In most cases the constant overhead of calling a function per comparison outweighs any speed advantage, and we do not include this facility in our package. An alternative implementation demonstrating incremental calculation of *target*, coupled with the original linear array statistics data structure [Witten

Table IV. Multiplicative- and Shift-Based Approximate Coding

Method	Statistics Structure	b	f	Compression (bits/char)	Encoding (MB/min.)	Decoding (MB/min.)
CACM Coder	list	32	14	4.89	4.83	3.93
Improved Coder	list	32	14	4.89	6.64	5.26
Improved Coder	tree	32	14	4.89	6.52	5.26
Shift/Add Coder	tree	32	14	4.89	7.89	6.75
Shift/Add Coder	tree	20	14	4.89	9.85	8.32
Shift/Add Coder	tree	16	14	5.12	10.59	8.90

et al. 1987], is available at `ftp://ftp.cs.toronto.edu` in the directory `/pub/radford`, file `lowp_ac.shar`.

6.2 Performance

Table IV shows throughput rates for both "multiply" and "shift" versions of the improved method, for various values of b and f, using an adaptive zero-order character-based model of the form described in Section 2 (as was also used for the experiments of Witten et al. [1987]).[8] The "shift/add" coder calculates a full f-bit *target* in a single call to *decode_target*(). Also listed is the compression ratio obtained by each version. The speed and compression of Algorithm CACM Coder for the same experiment is listed in the first row of the table, using the source code reproduced in the CACM article. In order to gauge the extent to which the speed improvements resulted from the data structure changes (from a sorted list to Fenwick's tree) we also coupled Algorithm Improved Coder to a sorted list structure, and these results are shown in the second line of the table.

The test file was 20MB of text taken from the *Wall Street Journal* (part of the *TREC* collection [Harman 1995]). This file has 92 distinct characters, a zero-order character entropy of 4.88 bits per characters, and has an MPS probability of 16% (for the space character). For Algorithm Improved Coder and Algorithm Shift/Add Coder, the data structure manipulations were such that the MPS was allocated all of the truncation excess, and so the compression results can be compared directly with those predicted by Table III. All experiments were performed on an 80 MIP Sun Sparc 10 Model 512, using a compiler that generated software function calls for integer multiplication and division operations.

Comparing the first two lines of Table IV, it is clear that the reduced number of arithmetic operations brings the expected performance gain. Comparing the second and third lines of the table, it is equally clear that for this small alphabet the tree-based statistics structures is no faster than the sorted list structure used in the CACM implementation. This is plausible, since for this alphabet the tree has eight levels, and Witten et al. [1987] reported that the sorted-list frequency structure averaged about 10 loop iterations when processing English text. The tree implementation is

[8]This is the program `char` distributed as part of our software package.

measurably better than the list structure for files that contain a less skew character distribution, such as object or data files.

The difference between the third and fourth lines shows the speed advantage obtainable with the shift/add implementation. Note that for this particular architecture the advantage accrues even when $b - f$ is relatively large. Finally, the fifth and sixth lines show how further speed improvements can be obtained if compression effectiveness is sacrificed. With $b - f = 6$ the compression loss is negligible, and even with $b - f = 2$ the loss is small—0.23 bits per character, well within the 0.50 average case bound reported above in Table III. (The difference arises because it was assumed in that analysis that t is always 2^f, whereas in the experiments it varies within the range $2^{f-1} < t \leq 2^f$.) Overall, comparing the first and the fifth rows of the table provides the justification for the earlier claim that both encoding and decoding speed are doubled—with no discernible loss of compression effectiveness—by the techniques advocated in this article.

As expected, the word-based compressor of Algorithm Zero-Order Word-Based (Figure 3) yields much better compression. With a 5MB memory limit in which the words and nonwords must be stored, and a shift/add coder using $b = 32$ and $f = 20$, it attains a compression ratio for the same 20MB test file of 2.20 bits per character, encodes at 10.8 MB/min., and decodes at 10.0 MB/min. Note that this includes the cost of learning the words and nonwords; there is no external vocabulary used and only a single pass is made over the file. As a reference point we used the same test harness on the well-known `Gzip` utility,[9] which uses a LZ77 [Ziv and Lempel 1977] sliding-window model and encodes pointers and literals using a variant of Huffman coding. `Gzip` compresses at 10.9 MB/min., decompresses at 92.3 MB/min., and attains a compression rate of 2.91 bits per character.

6.3 Hardware Considerations

These results are, of course, for one particular software implementation and one particular machine. Different relativities will arise if different hardware with different characteristics is used, or if a full hardware implementation is undertaken. Indeed, in many ways the Sparc architecture is the ideal machine for us to have carried out this work, since there are no integer multiplicative operations provided at the hardware level—all such operations are implemented at the machine-language level by a sequences of shift and add operations. Experimentally it appears that integer multiply operations take about 10–15 times longer than integer adds, and that integer divides take as much as 50 times longer, so it is not at all surprising that the shift/add implementation is faster.

[9]Gailly, J. 1993. Gzip program and documentation. The source code is available from `ftp://prep.ai.mit.edu/pub/gnu/gzip-*.tar`.

Table V. Module Interface Functions for Binary Arithmetic Coding

Module	Encoder	Decoder
Statistics	$C \leftarrow$ *create_binary_context*() *binary_encode*(C, s)	$C \leftarrow$ *create_binary_context*() $s \leftarrow$ *binary_decode*(C)
Coder	*binary_arithmetic_encode*(c_0, c_1, *bit*)	*bit* $\leftarrow$ *binary_arithmetic_decode*(c_0, c_1)

On the other hand, some architectures offer integer multiply and divide instructions in hardware, and execute them in the same time as other operations. For example, on an Intel Pentium Pro (200MHz), which provides single-cycle integer multiply and divide operations, encoding using Algorithm Shift/Add Coder is still slightly faster than encoding using Algorithm Improved Coder (25.5 MB/min. compared with 23.9 MB/min., both with $b = 32$ and $f = 27$, and with the CACM implementation [Witten et al. 1987] operating at 19.6 MB/min., all on the same 20MB test file), but decoding is slower (15.2 MB/min. for Algorithm Shift/Add Coder versus 20.7 MB/min. for Algorithm Improved Coder and 20.8 MB/min. for Algorithm CACM Coder). This difference in relative behavior between encoding and decoding is a result of the f-bit division in function *decode_target*(), which on the Pentium Pro is expensive in the shift/add version compared with single-cycle division used in Algorithms CACM Coder amd Improved Coder.[10] Hence, on machines such as the Pentium Pro, the best combination is to reduce the range using the shift/add approach in both encoder and decoder, but for *decode_target*() to make use of a full division. None of the other benefits of using the low-precision approach are compromised by such a combination.

7. BINARY ALPHABETS

A variety of compression applications deal with a binary alphabet rather than the multisymbol alphabets considered in the previous section. For example, both the context-based bi-level image compression methods [Langdon and Rissanen 1981] upon which the JBIG standard is based and the Dynamic Markov Compression method of Cormack and Horspool [1987] rely for their success upon the use of binary arithmetic coding. Binary coding can certainly be handled by the routines described in Table I, but specially tailored routines are more efficient. Table V shows the binary arithmetic coding routines that we support.

Binary coding allows a number of the components of a coder for a multisymbol alphabet to be eliminated, which is why there are efficiency gains from treating it specially. There is no need for the statistics data structure, since cumulative frequency counts are trivially available. It is

[10]Similar results were observed on the Sun hardware when explicit use was made of the in-built SuperSparc hardware multiplication and division operations. These fast operations are accessed via a compiler option that was discovered only after the bulk of the experimentation had been completed.

binary_arithmetic_encode(c_0, c_1, *bit*)
/* Arithmetically encode binary value *bit*, where where zero and one bits have previously been observed c_0 and c_1 times, respectively */
(1) If $c_0 < c_1$ then
    Set $LPS \leftarrow 0$ and $cLPS \leftarrow c_0$
else
    Set $LPS \leftarrow 1$ and $cLPS \leftarrow c_1$
(2) Set $r \leftarrow R$ div $(c_0 + c_1)$ and $rLPS \leftarrow r$ times $cLPS$
(3) If *bit* $= LPS$ then
    Set $L \leftarrow L + R - rLPS$ and $R \leftarrow rLPS$
else
    Set $R \leftarrow R - rLPS$
(4) Renormalize as for function *arithmetic_encode*() in Algorithm IMPROVED CODER (Figure 9)

binary_arithmetic_decode(c_0, c_1)
/* Arithmetically decode and return a binary value *bit*, where zero and one bits have previously been observed c_0 and c_1 times, respectively. Note that for binary coding there is no need to explicitly calculate a *target* */
(1) If $c_0 < c_1$ then
    Set $LPS \leftarrow 0$ and $cLPS \leftarrow c_0$
else
    Set $LPS \leftarrow 1$ and $cLPS \leftarrow c_1$
(2) Set $r \leftarrow R$ div $(c_0 + c_1)$ and $rLPS \leftarrow r$ times $cLPS$
(3) If $D \geq (R - rLPS)$ then
    Set *bit* $\leftarrow LPS$, $D \leftarrow D - (R - rLPS)$, and $R \leftarrow rLPS$
else
    Set *bit* $\leftarrow 1 - LPS$ and $R \leftarrow R - rLPS$
(4) Renormalize as for function *arithmetic_decode*() in Algorithm IMPROVED CODER (Figure 9)
(5) Return *bit*

Fig. 17. Algorithm Binary Coder.

also straightforward to arrange for the MPS (which for a binary coder must have probability of at least 0.5) to be allocated the excess probability caused by the truncation during the division. Finally, one multiplicative operation can be eliminated in the encoder when the less probable symbol (LPS) is transmitted, and one multiplicative operation avoided in the decoder for the MPS and two for the LPS. Algorithm Binary Coder (Figure 17) details the operation of a binary arithmetic encoder and decoder; c_0 and c_1 are assumed to be the frequencies of symbols zero and one respectively, and *bit* is the bit to be actually transmitted. Note that the MPS is allocated the truncation excess, but is placed at the lower end of the range R; this is to minimize the number of additive operations incurred when the MPS occurs.

To validate the procedures described in Algorithm Binary Coder, we implemented a simple bit-based compression model that predicts the next

Table VI. Binary Arithmetic Coding

Method	b	f	Compression (bits/char)	Encoding (MB/min.)	Decoding (MB/min.)
Improved Coder	20	14	3.22	1.22	0.93
Shift/Add Coder	20	14	3.22	1.49	1.28
Binary Coder, mult/div	20	14	3.22	1.84	1.90
Binary Coder, shift/add	20	14	3.22	3.19	3.36
Binary Coder, shift/add	16	14	3.35	3.71	3.72
Q-Coder	—	—	3.28	5.37	4.50

bit in an input stream based upon a context established by the k immediately preceding bits.[11] For example, with $k = 16$ each bit is predicted in a context of (roughly speaking) two eight-bit characters. The results of executing this model with the two coding methods—Algorithm Improved Coder and Algorithm Binary Coder—and $k = 16$ are shown in Table VI. The same test file was used as for Table IV, so the values are directly comparable. It should, however, be noted that approximately nine times as many arithmetic coding steps are performed in the bit-based model (one for each bit of each byte, plus a ninth to indicate whether there is a subsequent byte to be coded) than are required using the character-based model used for Table IV, so it is not surprising that throughput suffers.

Note that the shift/add binary arithmetic decoder does not suffer the problems described above for the multialphabet decoder, and there is no requirement for any f-bit values to be calculated. In both encoder and decoder all multiplicative operations can be evaluated to $b - f$ bits of precision.

The Q-Coder [Pennebaker et al. 1988] is the benchmark against which all binary arithmetic coders must be judged. The Q-Coder combines probability estimation and renormalization in a particularly elegant manner, and implements all operations as table lookups. The last row of Table VI shows the effect of coupling a Q-Coder (using the implementation of Kuhn [1996]) with the bit-based model with $k = 16$. This combination attains better throughput for both encoding and decoding, and compression midway between the results obtained with $b - f = 2$ and $b - f = 6$. Compared to the Q-Coder, the binary coding method described here does, however, have the advantage of allowing binary symbols to be interleaved in a compression stream containing multialphabet symbols. For example, some implementations of the PPM compression scheme [Cleary and Witten 1984] interleave binary novel/not-novel escape symbols with actual character predictions.

8. CONCLUSION

We have detailed from top to bottom the various components required in any adaptive compression scheme that uses arithmetic coding. We have

[11]This is the program `bits` distributed as part of our software package.

also described an improved implementation of arithmetic coding that, compared to previous approaches, uses fewer multiplicative operations and allows a much wider range of symbol probabilities. We have validated our claims with a full software implementation, and made that implementation available to others through the Internet. That implementation includes a word-based model that obtains good compression at reasonable speed; a simple character-based model that obtains only moderate compression, whose primary purpose is to serve as a test-harness for our comparative compression and speed results; and a bit-based model that is slow, but illustrates the key ideas of binary arithmetic coding and obtains compression midway between the word and character-based models. We hope that subsequent researchers will make use of these various software components when describing their own techniques for coding, and that it will be possible for results from disparate papers to be compared.

Finally, we conclude with a caveat: despite our advocacy of arithmetic coding there are also many situations in which it should *not* be used. For binary alphabets, if there is only one state and if the symbols are independent and can be aggregated into runs, then Golomb or other similar codes should be used [Gallagher and Van Voorhis 1975; Golomb 1966]. For multisymbol alphabets in which the MPS is relatively infrequent, the error bounds on minimum-redundancy (Huffman) coding are such that the compression loss compared to arithmetic coding is very small [Capocelli and De Santos 1991; Manstetten 1992]. If static or semistatic coding (that is, with fixed probabilities) is to be used with such an alphabet, a Huffman coder will operate several times faster than the best arithmetic coding implementations, using very little memory [Bookstein and Klein 1993; Moffat and Turpin 1997; Moffat et al. 1994].

On the other hand, adaptive minimum-redundancy (Huffman) coding is expensive in both time and memory space, and is handsomely outperformed by adaptive arithmetic coding even ignoring the difference in compression effectiveness. Fenwick's structure requires just n words of memory to manage an n-symbol alphabet, whereas the various implementations of dynamic Huffman coding [Cormack and Horspool 1984; Gallager 1978; Knuth 1985; Vitter 1987] consume more than 10 times as much memory [Moffat et al. 1994]. They also operate at between a half and a quarter of the speed of the adaptive arithmetic coder described here. Indeed, in many applications, the better compression of the arithmetic coder is almost incidental. It is this fact that inspires our enthusiasm and the continued investigation reported in this article.

ACKNOWLEDGMENTS

John Carpinelli, Wayne Salamonsen, and Lang Stuiver carried out the bulk of the programming work described here, and we thank them for their efforts. We are also grateful to Mahesh Naik and Michael Schindler, who provided extensive comments on our work. Finally we thank the referees for their perceptive and helpful comments.

REFERENCES

BELL, T. C., CLEARY, J. G., AND WITTEN, I. H. 1990. *Text Compression*. Prentice-Hall, Inc., Upper Saddle River, NJ.

BENTLEY, J. L., SLEATOR, D. D., TARJAN, R. E., AND WEI, V. K. 1986. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr.), 320–330.

BOOKSTEIN, A. AND KLEIN, S. T. 1993. Is Huffman coding dead?. *Computing 50*, 4, 279–296.

BURROWS, W. AND WHEELER, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. Rep. 124. Digital Equipment Corp., Maynard, MA.

CAPOCELLI, R. M. AND DE SANTIS, A. 1991. New bounds on the redundancy of Huffman codes. *IEEE Trans. Inf. Theor. IT-37*, 1095–1104.

CHEVION, D., KARNIN, E. D., AND WALACH, E. 1991. High efficiency, multiplication free approximation of arithmetic coding. In *Proceedings of the 1991 IEEE Data Compression Conference*, J. A. Storer and J. Reif, Eds. IEEE Computer Society Press, Los Alamitos, CA, 43–52.

CLEARY, J. G. AND WITTEN, I. H. 1984. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun. COM-32*, 396–402.

CORMACK, G. V. AND HORSPOOL, R N. 1984. Algorithms for adaptive Huffman codes. *Inf. Process. Lett. 18*, 3 (Mar.), 159–165.

CORMACK, G. V. AND HORSPOOL, R. N. 1987. Data compression using dynamic Markov modelling. *Comput. J. 30*, 6 (Dec.), 541–550.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

FENWICK, P. M. 1994. A new data structure for cumulative frequency tables. *Softw. Pract. Exper. 24*, 3 (Mar.), 327–336.

FEYGIN, G., GULAK, P. G., AND CHOW, P. 1994. Minimizing excess code length and VLSI complexity in the multiplication free approximation of arithmetic coding. *Inf. Process. Manage. 30*, 6, 805–816.

GALLAGER, R. G. 1978. Variations on a theme by Huffman. *IEEE Trans. Inf. Theor. IT-24*, 6 (Nov.), 668–674.

GALLAGER, R. G. AND VAN VOORHIS, D. C. 1975. Optimal source codes for geometrically distributed integer alphabets. *IEEE Trans. Inf. Theor. IT-21*, 2 (Mar.), 228–230.

GOLOMB, S. W. 1966. Run-length encodings. *IEEE Trans. Inf. Theor. IT-12*, 3 (July), 399–401.

HAMAKER, D. 1988. Compress and compact discussed further. *Commun. ACM 31*, 9 (Sept.), 1139–1140.

HARMAN, D. K. 1995. Overview of the second text retrieval conference (TREC-2). *Inf. Process. Manage. 31*, 3 (May–June), 271–289.

HORSPOOL, R. N. AND CORMACK, G. V. 1992. Constructing word-based text compression algorithms. In *Proceedings of the 1992 IEEE Data Compression Conference*, J. Storer and M. Cohn, Eds. IEEE Computer Society Press, Los Alamitos, CA, 62–71.

HOWARD, P. G. AND VITTER, J. S. 1992. Analysis of arithmetic coding for data compression. *Inf. Process. Manage. 28*, 6 (Nov.–Dec.), 749–763.

HOWARD, P. G. AND VITTER, J. S. 1994. Arithmetic coding for data compression. *Proc. IEEE 82*, 6, 857–865.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng. 40*, 9 (Sept.), 1098–1101.

JIANG, J. 1995. Novel design of arithmetic coding for data compression. *IEE Proc. Comput. Dig. Tech. 142*, 6 (Nov.), 419–424.

JONES, D. W. 1988. Application of splay trees to data compression. *Commun. ACM 31*, 8 (Aug.), 996–1007.

KNUTH, D. E. 1985. Dynamic Huffman coding. *J. Alg. 6*, 2 (June), 163–180.

KUHN, M. 1996. Implementation of JBIG (including a QM-coder). ftp://ftp.informatik.uni-erlangen.de/pub/doc/ISO/JBIG/jbigkit-0.9.tar.gz

LANGDON, G. G. 1984. An introduction to arithmetic coding. *IBM J. Res. Dev. 28*, 2 (Mar.), 135–149.

LANGDON, G. G. AND RISSANEN, J. 1981. Compresson of black-and-white images with arithmetic coding. *IEEE Trans. Commun. COM-29*, 858–867.

MANSTETTEN, D. 1992. Tight upper bounds on the redundancy of Huffman codes. *IEEE Trans. Inf. Theor. 38*, 1 (Jan.), 144–151.

MOFFAT, A. 1989. Word-based text compression. *Softw. Pract. Exper. 19*, 2 (Feb.), 185–198.

MOFFAT, A. 1990a. Implementing the PPM data compression scheme. *IEEE Trans. Commun. 38*, 11 (Nov.), 1917–1921.

MOFFAT, A. 1990b. Linear time adaptive arithmetic coding. *IEEE Trans. Inf. Theor. 36*, 2 (Mar.), 401–406.

MOFFAT, A. 1997. Critique of "Novel design of arithmetic coding for data compression". *IEE Proc. Comput. Digit. Tech.*

MOFFAT, A. AND TURPIN, A. 1997. On the implementaiton of minimum-redundancy prefix codes. *IEEE Trans. Commun. 45*, 10 (Oct.), 1200–1207.

MOFFAT, A., SHARMAN, N., WITTEN, I. H., AND BELL, T. C. 1994. An empirical evaluation of coding methods for multi-symbol alphabets. *Inf. Process. Manage. 30*, 6, 791–804.

PENNEBAKER, W. B., MITCHELL, J. L., LANGDON, G. G., AND ARPS, R. B. 1988. An overview of the basic prinicples of the Q-coder adaptive binary arithmetic coder. *IBM J. Res. Dev. 32*, 6 (Nov.), 717–726.

RISSANEN, J. 1976. Generalised Kraft inequality and arithmetic coding. *IBM J. Res. Dev. 20*, 198–203.

RISSANEN, J. AND LANGDON, G. G. 1979. Arithmetic coding. *IBM J. Res. Dev. 23*, 2 (Mar.), 149–162.

RISSANEN, J. AND LANGDON, G. G. 1981. Universal modeling and coding. *IEEE Trans. Inf. Theor. IT-27*, 1 (Jan.), 12–23.

RISSANEN, J. AND MOHIUDDIN, K. M. 1989. A multiplication-free multialphabet arithmetic code. *IEEE Trans. Comm. 37*, 2 (Feb.), 93–98.

RUBIN, F. 1979. Arithmetic stream coding using fixed precision registers. *IEEE Trans. Inf. Theor. IT-25*, 6 (Nov.), 672–675.

SCHINDLER, M. 1998. A fast renormalisation for arithmetic coding. In *Proceedings of the 1998 IEEE Data Compression Conference* (Snowbird, UT), J. A. Storer and M. Cohn, Eds. IEEE Computer Society Press, Los Alamitos, CA, 572.

SHANNON, C. E. 1948. A mathematical theory of communication. *Bell Syst. Tech. J. 27*, 79–423.

VITTER, J. S. 1987. Design and analysis of dynamic Huffman codes. *J. ACM 34*, 4 (Oct.), 825–845.

WITTEN, I. H. AND BELL, T. C. 1991. The zero frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inf. Theor. 37*, 4 (July), 1085–1094.

WITTEN, I. H., MOFFAT, A., AND BELL, T. C. 1994. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold Co., New York, NY.

WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. 1987. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June), 520–540.

WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. 1988. Authors' response to "Compress and Compact discussed further". *Commun. ACM 31*, 9 (Sept.), 1140–1145.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor. IT-23*, 3, 337–343.

Received: June 1996; revised: December 1996 and May 1997; accepted: August 1997