Memory-Efficient Recursive Interleaved Entropy Coding

A. B. Kiely[1] and M. Klimesh[1]

Recursive interleaved entropy coding is a promising new adaptable binary entropy coding technique that offers fast encoding and decoding at high compression efficiency [1]. However, with the encoding methods presented in [1], the maximum memory required by the encoder is proportional to the source sequence length. In this article, we present a new encoding technique, and corresponding decoding technique, that limits encoder memory usage without limiting the source sequence length. We illustrate the improvement in compression efficiency over the alternative memory-efficient technique that partitions the source sequence into smaller blocks. We present a generalized encoding framework that encompasses both memory-efficient encoding methods and includes additional variations as well.

I. Introduction

Recursive interleaved entropy coding, introduced in [1], is a novel binary entropy coding technique that offers the same functionality as arithmetic coding: it accommodates an estimate of the source bit probability distribution that can be updated with each source bit, and it can achieve arbitrarily small redundancy at the expense of increasing complexity. In addition, simulation results in [1] suggest that it has significant speed advantages over arithmetic coding.

The first encoding scheme described in [1], which we refer to as the "basic" encoder, loads the entire source sequence into memory before further processing occurs. This leads to two potential limitations that may be unacceptable in some data compression applications:

(1) The maximum amount of memory required by the encoder is proportional to the source sequence length, which presents obvious limitations in encoding long sequences.

(2) Encoding latency is large since no output is produced until the entire source sequence is loaded into memory.

Note that neither of the above limitations occurs in decoding—decoder memory use and latency are both very low.

[1] Communications Systems and Research Section.

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The recursive encoder of [1, Fig. 5] could easily be modified to load source bits into memory only as they are needed to process codewords, but it is always possible that the entire source sequence must be loaded before any processing can occur. Thus, the above issues are still present in this encoder, although the *average* encoding latency and memory use can be reduced somewhat.

A straightforward method of overcoming these encoding limitations is to partition the source sequence into smaller blocks that are encoded separately. We refer to this as the "partitioning" method of encoding.

In this article we describe an alternative method of memory-efficient recursive interleaved entropy coding that yields more efficient compression than the partitioning method. We assume throughout that the reader is familiar with [1].

The key to our technique is the judicious flushing of partial codewords in the encoder. In the basic encoder, flush bits are added to prevent partial codewords from being left in the encoder at the end of the encoding process. Here flushing is also used to dislodge partial codewords that cannot be completed normally because the memory constraint prevents the encoder from loading source bits that might lead to completion of the codeword.

Note that the addition of the extra flush bits during encoding means more bits must be encoded, and therefore compression efficiency is reduced. However, with the method we introduce, we will see that this cost can be quite small.

It is essential for the decoder to recognize bits that were inserted as flush bits during encoding; otherwise these bits could be mistaken for source bits or for bits that arose from processing codewords, and decoding would fail. Thus, we must add flush bits in a way that can be recognized by the decoder. Note that this is not an issue in the basic encoding scheme, since in this case flush bits are simply extra bits remaining in the decoder after the source sequence has been decoded.

We introduce a method that constrains the encoder to consider no more than M source bits at a time. That is, source bit i is completely encoded before source bit $i + M$ enters the encoder. We refer to this technique as "sliding source window" encoding with source window length M. A sliding source window encoder inserts flush bits less frequently than a partitioning encoder with the same memory constraint and thus gives better compression. In Section V we present a generalization of the sliding source window technique that encompasses other memory-efficient variations as well.

For either a sliding source window encoder with source window length M or a partitioning encoder with partition length M, the maximum memory[2] needed by the encoder depends on the coder design and is essentially linear in M.

II. Encoding

In the sliding source window encoder, as in the basic encoder, bits together with the indices of their corresponding bins are maintained in encoder memory in a list arranged in order of priority. The sliding source window encoder processes data in "waves." During the ith wave, the encoder produces the minimal number of additional output bits necessary to reconstruct the value of the ith source bit, given that the output bitstream already includes the information needed to reconstruct the first $i - 1$ source bits.

We ensure that bits containing information about source bit i do not remain in the encoder when the ith wave is complete by recording for each bit a *freshness index*, which acts like an expiration date for

[2] We assume that once the encoder calculates the next bit to be transmitted, this bit is not counted in encoder memory usage, since even if the bit cannot be transmitted immediately, it can often be stored in a separate buffer. In any case, managing these compressed bits is necessary for any compression algorithm.

the bit. The freshness index of a bit indicates the latest wave during which the bit may be processed. Wave i is complete when all bits in the encoder have freshness index greater than i. Freshness indices are assigned according to the following rules:

(1) The ith source bit, b_i, is assigned freshness index i.

(2) When bits in a bin are taken to form a codeword, the resulting output bits are all assigned a freshness index equal to the minimum freshness index of any of the bits that formed the codeword, excluding flush bits.

A consequence of these rules is that the freshness index of any bit is no larger than the freshness index of the bit with next highest priority; however, in contrast to the situation with priority values, several bits may have the same freshness index.[3] The rules give two properties that can be exploited in the encoding process:

(1) To determine whether wave i is complete, we need only check that the freshness index of the first (i.e., highest-priority) bit in the encoder is greater than i.

(2) When we form a codeword, the resulting output bits are all assigned a freshness index equal to that of the first bit of the codeword.

Note that at any time the freshness indices of bits in encoder memory can take on only M different values, so we need only store freshness indices modulo M.

During wave i, the encoder has not yet observed source bits $b_{i+M}, b_{i+M+1}, \cdots$. Once the wave is complete, we can load the next source bit, b_{i+M}, into memory. Often more than one wave will be completed simultaneously. If after processing wave i the highest priority bit in the encoder has freshness index $i + \ell + 1$, then bits $b_{i+1}, \cdots, b_{i+\ell}$ can also be reconstructed from the current output bitstream, and we can load ℓ new source bits into the encoder.

Sliding source window encoding follows the five priority rules given in [1, Section II.B]. For convenience, the encoding algorithms given in [1] abide by a stricter version of the fourth rule:

(4′) A codeword cannot be processed unless the last bit in the codeword has higher priority than all bits in higher-indexed bins.

We also adopt Rule (4′) for sliding source window encoding, as it makes encoding much simpler. Note, however, that in [1] use of this stricter version does not affect the output bitstream, while here the stricter rule can affect the output bitstream.

The priority rules allow some flexibility in the order in which processing of some codewords is performed, so the specific operations performed during a given wave are not strictly defined.

Flush bits can be added only in the highest-indexed nonempty bin, and only if the priority rules prevent any other operation from being performed. In other words, we cannot add flush bits if we can output another encoded bit, load another source bit, or process another codeword.

Figures 1 through 3 outline two different sliding source window encoding procedures that conform to the priority rules. Both encoders produce the same encoded bitstream. In these figures, and elsewhere, source bits are numbered from 1.

[3] With the section-number-like priority labels described in [1, Section II.B], the freshness index of a bit is equivalent to the first number of the priority label.

```
initialization: clear memory and assign nextbit $= 1$

repeat
    repeat
        if one of the following conditions holds, then perform the associated operation:[a]
        (a) if memory is empty or the highest priority bit has freshness index $>$ nextbit $- M$, then
                1. load source bit $b_{\texttt{nextbit}}$ into memory
                2. assign nextbit $=$ nextbit $+1$
        (b) if the highest priority bit in memory is in the first bin, then remove it and output the value of
            this bit[b]
        (c) if a codeword can be formed in some bin (subject to the priority rules, and without adding any
            flush bits), then process this codeword
    until (none of these conditions holds)

    add flush bits as needed to form a complete codeword in the highest-indexed nonempty bin and process
        this codeword.
until (all source bits have been loaded and memory is empty)
```

[a] We can choose the operation arbitrarily from those for which the corresponding condition is satisfied.

[b] If the bit with the next highest priority has a different freshness index or if memory is now empty, then this operation marks the end of a wave. Depending on the encoder implementation, there may be no reason to track the wave number.

Fig. 1. Outline of one implementation of the sliding source window encoding algorithm.

```
initialization: clear memory and assign nextbit $= 1$

repeat
    repeat
        if one of the following conditions holds, then perform the associated operation:[a]
        (a) if memory is empty or the highest priority bit has freshness index $>$ nextbit $-M$, then
                1. load source bit $b_{\texttt{nextbit}}$ into memory
                2. assign nextbit $=$ nextbit $+1$
        (b) if the highest priority bit in memory is in the first bin, then remove it and output the value of
            this bit
    until (none of these conditions holds)

    if memory is not empty, then
        let bitpointer point to highest priority bit in memory
        while (bin(bitpointer) $\neq 1$)
            let bitpointer $=$ MakeCodeword(bitpointer)
until (memory is empty)
```

[a] If both conditions hold, then either operation can be performed.

Fig. 2. An alternative implementation of the sliding source window encoding algorithm using the `MakeCodeword` procedure of Fig. 3.

```
MakeCodeword(bitpointer):
   thebin = bin(bitpointer)
   bitstring = bit(bitpointer)

   while (bitstring is not a complete codeword)
      if bitpointer points to the lowest priority bit in memory
         append flush bits as needed to make bitstring a complete codeword
      else
         assign bitpointer to the bit with next lower priority
         while (bin(bitpointer) $>$ thebin)
            bitpointer = MakeCodeword(bitpointer)
         if (bin(bitpointer) = thebin)
            append bit(bitpointer) to bitstring

   delete from memory the bits that formed bitstring
   insert the corresponding output bits at the position occupied by the first bit in bitstring
   return a pointer to first output bit generated
```

Fig. 3. The recursive `MakeCodeword` procedure from [1]. This procedure forms a codeword starting with a given bit (and if needed forms codewords in higher-indexed bins), producing the corresponding output bits in encoder memory.

The following example illustrates sliding source window encoding.

Example 1. We demonstrate sliding source window encoding using the procedure in Fig. 1 with $M = 3$ and coder design C5 of [1]. For convenience, coder design C5 is shown in Fig. 4. Our example source sequence is $0, 0, 1, 1, 1$, with these bits arriving in bins $2, 4, 4, 1, 2$, respectively. In this example, we'll store encoder information in a linked list. Each record in the linked list stores a bit value, the bin to which the bit is assigned, and the freshness index of the bit. Initially we load the first three source bits, so the encoder memory is as shown in step (a) of Fig. 5.

In the first wave of encoding, we must generate the encoded information required to reconstruct the first source bit. Here, the first source bit is in bin 2, and no complete codeword is present in this bin. However, a complete codeword exists in a higher-indexed bin (bin 4), so we process this codeword by replacing it with the output bits as shown in step (b). At this point, bin 2 contains a complete codeword, but our priority rules do not allow us to process it because bin 3 has a bit with higher priority than the last bit in the codeword. So we process the codeword in bin 3 (which consists of a single bit), producing the encoder state shown in step (c). Finally, we can process a codeword in bin 2; the resulting encoder state is shown in step (d). The first two bits in memory have freshness index 1 and are assigned to bin 1, so we output these bits, and the first wave is complete since no bits remain in the encoder with freshness index 1. We add the next source bit to memory to arrive at the encoder state shown in step (e).

In the second wave of encoding, all bits in the encoder are in the first bin, with the exception of a single bit in the second bin. Since this bit has freshness index 2, we must process it to complete the second wave. This bit does not form a complete codeword, so we add flush bits (in this case a single zero will suffice) as shown in step (f). The resulting codeword is processed, producing the encoder state of step (g). At this point, the first three bits in the encoder have freshness index 2 and are in the first bin, so these bits are the next encoder output bits, and we have completed the second wave. The fifth source bit is loaded into memory.

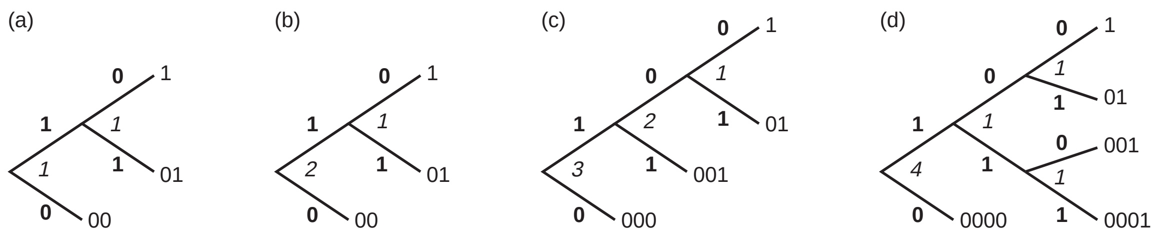

Fig. 4. The 5-bin coder design C5 reproduced from [1]: (a) bin 2, (b) bin 3, (c) bin 4, and (d) bin 5. The first bin does not have an associated tree. Output bits are shown in boldface, and the corresponding bin indices are in italics. The input codewords are shown at terminal nodes of the trees.

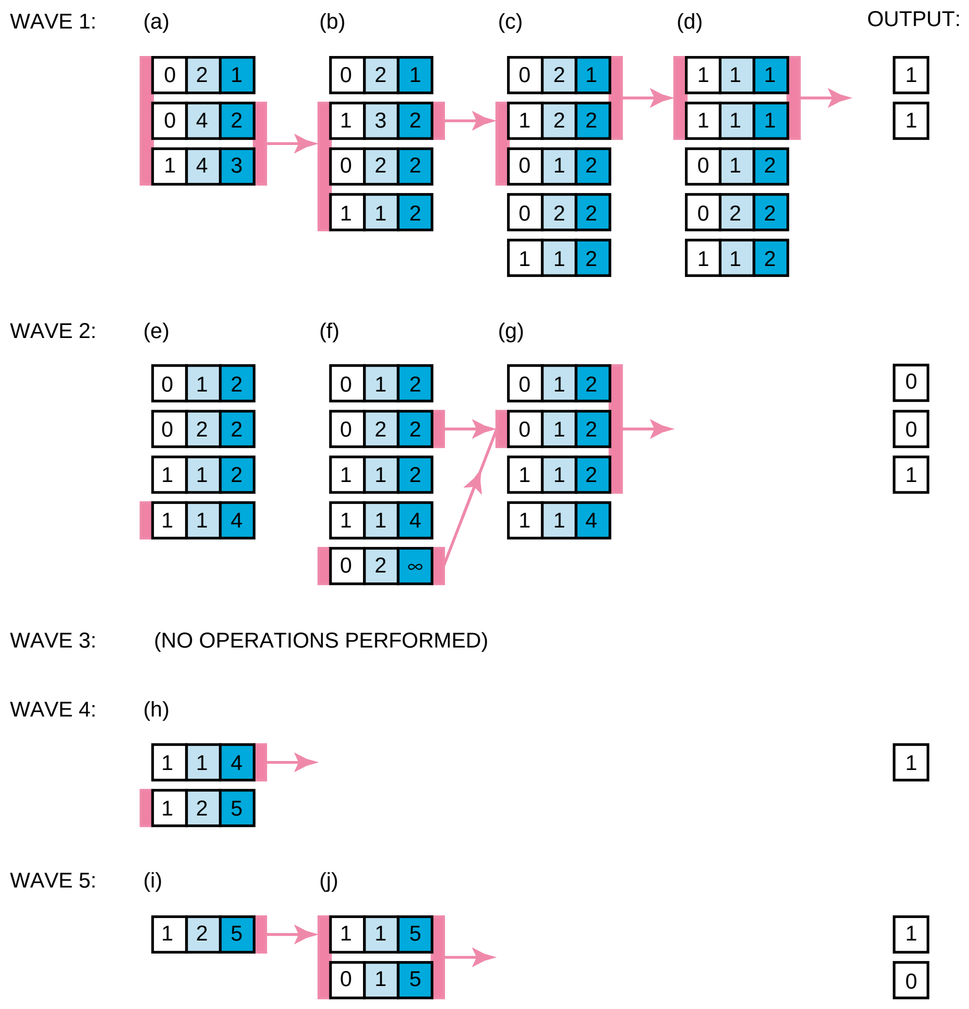

Fig. 5. Example of the sliding source window encoding procedure. In each triple, the first box indicates bit value, the second box shows bin index, and the third box gives the freshness index of the bit. (Steps (a) through (j) are discussed in Example 1.)

We observe in step (g) that no bits in the encoder have freshness index 3. Thus, the third source bit was encoded as part of the first and second waves, and no additional processing is needed to complete the third wave. If there were a sixth source bit to encode, it would be loaded into memory now.

For the fourth wave, shown in step (h), the only bit with freshness index 4 is already in the first bin, so we output this bit and the wave is complete.

In the fifth and final wave, shown in steps (i) and (j), we process the codeword formed in bin 2 by the last remaining bit. The resulting output bits fall in the first bin and become the last two encoder output bits.

The final output bitstream is shown in the right-hand column of Fig. 5. $\triangle$

III. Decoding

The sliding source window decoder reconstructs the source sequence in the same manner as the basic decoder described in [1], with the key modification that it must routinely identify and delete the flush bits inserted by the encoder.[4]

Although the priority rules allow flexibility in the order in which some codewords are processed, flush bits can be added only if no other encoding operation can be performed. Thus, the wave during which a codeword containing flush bits will be processed is unambiguous. For each bit in memory, the decoder calculates the *wave number*, which we define as the last encoding wave during which the codeword containing the bit could have been processed. A bit in the decoder with wave number $i - M$ was part of a codeword formed before or during encoding wave $i - M$, at which time the encoder had no observation of source bit b_i. Consequently, if we are about to decode the ith source bit, any bits in memory with wave number less than $i - M + 1$ are flush bits and must be discarded.

At any time during decoding, all the bits in a given bin have the same wave number, so in practice it's more convenient to record the wave number for each bin rather than for each bit, and we will refer to both bits and bins as having wave numbers.

When we reconstruct a codeword in a given bin, the wave number assigned to the bin is equal to the minimum wave number over all bits in lower-indexed bins, including bits that are used and removed when the codeword is reconstructed. Here we treat bits in the first bin as having wave number equal to the index of the source bit being reconstructed. This wave number assignment method is necessitated in part by the fact that, during encoding, a complete codeword in a bin must be processed before any of the incomplete codewords in lower-indexed bins can be completed and processed.

This construction ensures that the wave number in the nonempty bins is a nonincreasing function of bin index. This fact simplifies decoding in two ways:

(1) When reconstructing a codeword in bin j, to calculate the wave number we need only find the wave number of the highest-indexed nonempty bin with index less than j. Calculation of wave numbers can naturally be included as part of a modified version of the `GetCodeword` procedure described in [1].

(2) When searching decoder memory for flush bits to delete, we can start our search at the highest-indexed nonempty bin, and stop searching once we find a nonempty bin that does not contain flush bits.

[4] The flush bits added after the entire source sequence has been loaded into encoder memory will simply be leftover in the decoder memory after the source sequence is decoded, so we don't explicitly need to check for them.

There are several possible methods a decoder can use to identify flush bits to be removed:

Method 1. Modify the `GetBit` procedure given in [1] to first find and remove any flush bits from the requested bin. (Note that decoding one source bit can require several `GetBit` operations.)

Method 2. Check for and remove flush bits before decoding each source bit. To do this, we check bins starting with the highest-indexed bin. We ignore empty bins, clear bins with a wave number equal to $i - M$ (where i is the index of the source bit being decoded), and are finished when we reach a bin with a wave number greater than $i - M$. This method may be slow.

Method 3. Keep track of the minimum wave number among the nonempty bins. Before decoding the ith source bit, if the minimum wave number is equal to $i - M$, then clear bins with a wave number equal to $i - M$ (as in Method 2) and recalculate the minimum wave number. The minimum wave number is also recalculated when a bin with the minimum wave number becomes empty. Recalculation of the minimum wave number need not occur often, and this method should be faster than Method 2, but there is no guarantee that recalculation will be rare.

Method 4. *The wave number bound method:* Maintain a lower bound on the lowest wave number of any bit in the encoder. This bound is initialized to be 1 and is not changed by normal decoding operations. Before decoding the ith source bit, check whether this bound is equal to $i - M$. If it is, then check for and remove bits with wave number $i - M$ (there may not be any), and reset the bound to equal the actual lowest wave number. Theorem 1 below asserts that the fraction of source bits for which a recalculation is triggered cannot be greater than B/M, where B is the number of bins. Note that flush-bit removal occurs only when the wave number bound is computed, and thus for most source bits only an increment and compare is needed.

The wave number bound method (Method 4) is probably the fastest choice for a software implementation of the decoder. Under Method 1, flush bits sometimes remain in decoder memory long after we are able to identify them; all other methods remove flush bits as soon as they expire. Consequently, all but Method 1 allow wave numbers to cycle; e.g., after reaching wave M, the next wave number is 1.

We observe in Section VI that the above techniques for identifying flush bits are just as applicable for the special case of non-recursive interleaved entropy coding [2,3]. An adapted version of the wave number bound method is quite useful in this case and does not appear to have been previously identified.

Increased decoding speed can also be achieved by modifying the encoder so that several source bits are loaded at a time, at widely spaced intervals. This idea is developed in more detail and generality in Section V; for now we remark that with this modification the decoder needs to check for flush bits only at these widely spaced intervals. Any of the above methods can be used for this check, and the speed of this operation becomes relatively unimportant.

The following theorem establishes that computation of the bound used in the wave number bound method occurs relatively infrequently.

Theorem 1. *In decoding a source sequence using the wave number bound method, the number of times the wave number bound must be computed is at most B/M times the number of source bits, where B is the number of bins and M is the sliding source window size.*

In the Appendix, we prove this theorem, and we also show that the fraction of source bits for which the bound is recomputed can approach $B/(M + 1)$.

IV. Performance

Since compression performance depends on the particular source encountered, it's difficult to precisely quantify the improvement that sliding source window encoding offers over the partitioning method. However, as an indication of this improvement, we have measured the redundancy resulting from both methods for a source in which the probability-of-zero is uniformly distributed on $[0, 1]$. Specifically, we generated sequences of probability values, produced random bits according to these values, and compressed the sequence using the optimal bin assignment for each source bit given the probability-of-zero. Compression was performed using the 6-bin coder design of [1, Table C-2] and with various values of the sliding window or partition length. The results are shown in Fig. 6, along with the performance of the basic encoder, which achieves essentially the asymptotic redundancy offered by the two memory-efficient techniques. This figure demonstrates the noticeable performance improvement offered by the sliding source window encoding method compared to the partitioning method when the same encoder memory constraint is imposed.[5]

For large M, the extra redundancy (the increase in redundancy over that of the basic encoder) of the partitioning encoder decreases as $1/M$, while the extra redundancy of the sliding source window encoder becomes essentially zero at some window length. The window length at which this occurs varies depending on the number of bins and the distribution of source bits into these bins.

The amount of memory that must be available to an encoder is determined by the coder design and the window or partition length M. We quantify this amount with the number of bit values that might be contained in bins in the encoder at one time. (Additional information such as the bin numbers and freshness indices of the bits may also need to be stored, but for a particular encoder type the memory used is still essentially proportional to the number of bit values stored.) As examples of memory requirements, we have determined that coder design C5 and the 6-bin coder design of [1, Table C-2] have maximum

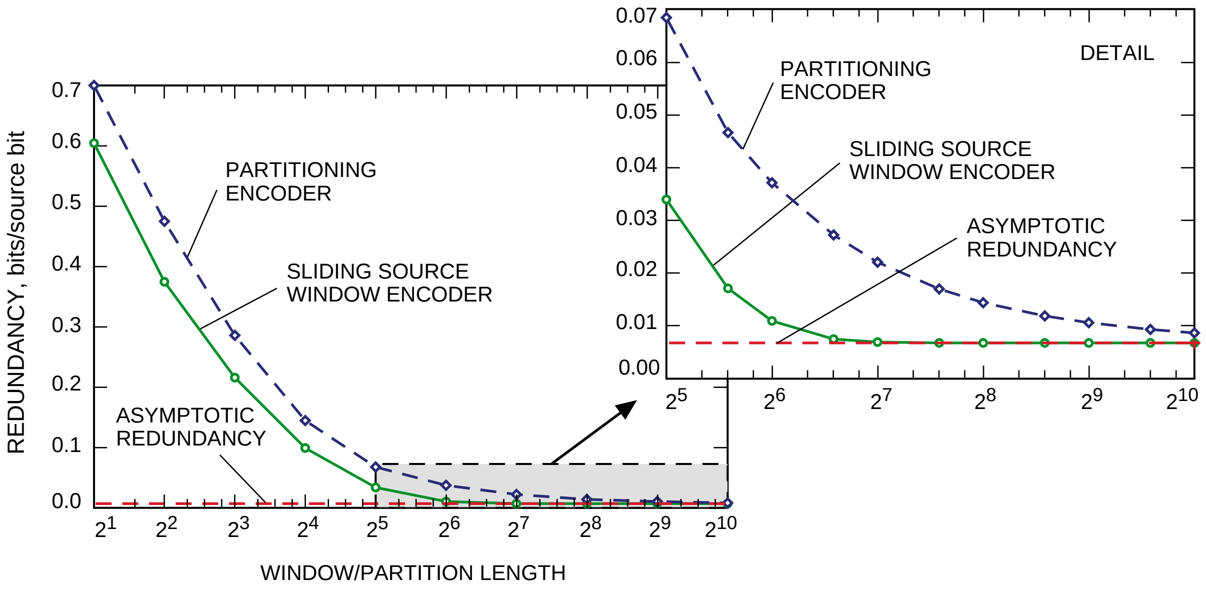

Fig. 6. Redundancy as a function of window or partition length for the source described in the text, compressed using the 6-bin coder design of [1, Table C-2]. Each point was generated using 500 sequences, each of length 2^{20} bits.

[5] Note, however, that a partitioning encoder has less data to store with each bit than a sliding source window encoder, so for a given value of M, the partitioning encoder uses slightly less memory.

memory usages of $M + 8$ and $6M - 3$, respectively. For coder design C5, a source sequence that uses maximum memory is the all-ones sequence assigned to the fifth bin. For the 6-bin encoder, maximum memory is used by the all-ones sequence with the first bit in the third bin, and all remaining bits in the sixth bin.

Maximum encoder memory usage is essentially linear in M. It is equal to M plus a constant when the coder design has the property that each codeword in any given bin produces at least one bit in the bin with the next lower index (coder design C5, for example, has this property).

V. A Unifying Generalization

We've seen three types of encoders—basic, partitioning, and sliding source window—that differ in the rules specifying when source bits can be loaded into encoder memory. We now describe a generalized encoding technique that encompasses all three of these encoders and includes other encoders as well by allowing more general schedules for loading source bits. Alternative rules for loading source bits can allow the decoder to check for flush bits less frequently (and thus increase decoding speed), and may allow more efficient use of encoder memory.

We use a function w to describe a schedule for loading source bits, where $w(i)$ equals the index of the wave that must be completed before loading source bit b_i. Obviously source bits must be loaded in order, and a source bit cannot be encoded until it is loaded into encoder memory, so we require that $w(i)$ is nondecreasing and $w(i) < i$, but apart from these restrictions, w can be an arbitrary function of i. In fact, w can also be a function of source bits $b_{i-1}, b_{i-2}, \cdots$ since the encoder and decoder both have access to the values of these bits when $w(i)$ needs to be computed. In particular, we could use the values of previous source bits to calculate or estimate memory usage and load bits in a way that keeps encoder memory nearly full. Such a technique would achieve better compression performance by making more efficient use of encoder memory, although decoding might be slower.

Example 2. The following choices of w illustrate how this generalization encompasses various encoding techniques (recall that source bits are numbered beginning with index 1):

(a) The basic encoder uses $w(i) = 0$ for all i since all source bits are loaded into memory before encoding.

(b) The partitioning method loads M source bits at a time, which corresponds to

$$w(i) = M \cdot \left\lfloor \frac{i-1}{M} \right\rfloor$$

(c) The sliding source window encoder uses $w(i) = \max(i - M, 0)$.

(d) An encoder that initially loads M source bits into memory, and then encodes groups of K source bits at a time, can be obtained using

$$w(i) = \max\left(K \cdot \left\lfloor \frac{i+K-M-1}{K} \right\rfloor, 0\right) \qquad \triangle$$

An arbitrary schedule for loading source bits is accommodated by modified versions of the sliding source window encoder and decoder. In the modified encoder, as in the sliding source window encoder, flush bits cannot be added if the encoder is permitted to load another source bit. But now the function

$w(i)$ determines whether a source bit can be loaded—if the highest priority bit in memory has freshness index larger than $w(i)$, then we can load source bit b_i.

The modified decoder is similar to the sliding source window decoder: before decoding source bit b_i, all bits with a wave number less than or equal to $w(i)$ are flush bits and should be deleted. In the encoder, source bit b_i is not loaded until wave $w(i)$ is complete, so the encoder did not have access to source bits with index less than or equal to $w(i)$ while encoding bit b_i. In part (d) of Example 2 above, this means we need only check for flush bits before decoding source bit b_i if $i = M + 1 + jK$ for some integer $j \geq 0$.

For a given w, the maximum source window size (i.e., the maximum number of source bits that can reside in encoder memory at a time) is $M = \max_i\{i - w(i)\}$. This maximum must also be taken over all possible source sequences when w is allowed to depend on the values of source bits.

VI. Memory-Efficient Non-Recursive Encoding

For the special case of non-recursive interleaved entropy coding, memory-efficient encoding can be accomplished using a substantially different—and simpler—technique, described in [2,3]. Instead of limiting the number of source bits considered by the encoder at one time, we limit the number of codewords and partial codewords that can reside in encoder memory at one time. For completeness we present some details of the technique here.

In non-recursive interleaved entropy coding, the encoder can build up codewords from source bits one bit at a time. The encoder contains a list of codewords and prefixes of codewords (rather than individual bits), implemented as a circular buffer with room for some maximum number of words. When a source bit arrives, if there is a partial codeword for the bit's bin, the bit is appended to the word. Otherwise, the bit begins a new word at the end of the list. Here we can treat the first bin like the other bins, with each source bit comprising a complete codeword. When the codeword at the beginning of the list is complete, the encoder outputs the corresponding output bits, and the codeword is removed from the list. If the new front position of the list contains a complete codeword, it is also processed, and so on. If the buffer becomes full, we add flush bits to complete the (necessarily partial) codeword at the front of the list, so that the resulting codeword can be processed.

The decoder reconstructs codewords in the same order that they formed in the encoder's list. Thus, the decoder can easily keep track of the position that each codeword occupied in the encoder, and in fact this is easier than keeping track of wave numbers in the recursive case. Therefore, the decoder can determine when the encoder added flush bits and can remove these bits accordingly. With "wave numbers" replaced by "word numbers," any of the techniques of Section III can be used to check for flush bits, and the generalized schedules of Section V can be adapted as well. In particular, the equivalent of the wave number bound method allows efficient flush-bit detection and a generalized schedule that adds flush bits only at certain points (analogous to part (d) of Example 2 in Section V) could achieve the same result.

VII. Conclusion

We have modified the encoding and decoding techniques presented in [1] in a way that allows memory-efficient encoding. The new techniques offer improved compression performance compared to the simpler alternative of partitioning long source sequences into segments of manageable size. The key to memory-efficient encoding is the addition of flush bits in a way that can be tracked by the decoder and that limits the delay in waiting for complete codewords to be formed. We have described an encoder called a sliding source window encoder that accomplishes this by considering a limited number of source bits at one time. We have described the corresponding decoder and identified efficient new techniques that allow the decoder to identify flush bits; these techniques are also useful in the more straightforward case of non-recursive interleaved entropy coding. We presented a generalization of the encoding technique that encompasses the basic encoding method, sliding source window encoding, and other techniques as well.

A number of directions for future research are suggested by this work. These include evaluating the encoding and decoding speeds of our new techniques and their variations, and determining a simple method of computing the memory requirement as a function of M for an arbitrary coder design. We observe that in general the amount of memory used during encoding may still be substantially lower than the amount of memory that must be reserved; thus, the prospect exists of producing an even more memory-efficient encoder.

References

[1] A. B. Kiely and M. Klimesh, "A New Entropy Coding Technique for Data Compression," *The InterPlanetary Network Progress Report 42-146, April–June 2001*, Jet Propulsion Laboratory, Pasadena, California, pp. 1–48, August 15, 2001. http://ipnpr.jpl.nasa.gov/progress_report/42-146/146G.pdf

[2] P. G. Howard, "Interleaving Entropy Codes," *Proc. Compression and Complexity of Sequences 1997*, Salerno, Italy, pp. 45–55, 1998.

[3] F. Ono, S. Kino, M. Yoshida, and T. Kimura, "Bi-Level Image Coding with MELCODE—Comparison of Block Type Code and Arithmetic Type Code," Proc. IEEE Global Telecommunications Conference (GLOBECOM '89), Dallas, Texas, pp. 0255–0260, November 1989.

Appendix
The Wave Number Bound Method

In this appendix we prove Theorem 1 of Section III, which states that under the wave number bound method the number of times the bound must be computed is at most B/M times the number of source bits, where B is the number of bins and M is the sliding source window size. We also show that the fraction of source bits for which a bound computation occurs can approach $B/(M+1)$.

Proof. We say that the lower bound on the wave number is "triggered" by bin η at "time" t if the bound is recomputed just before decoding the tth source bit and this recomputation causes the new bound to be equal to the wave number of the bits in bin η at this time. In other words, after removing any flush bits, bin η contained the oldest bits. (Note that the bound might be triggered by more than one bin at a given time.)

Suppose that the sequence of times at which recomputations occur includes times $t_1, \cdots, t_{B+1}$, where $t_1 < \cdots < t_{B+1}$. For each i, let η_i be a bin that triggered the bound at time t_i, and let a_i be the resulting new bound.

The uncoded bin never contains flush bits and can never trigger a recomputation. Thus, the η_i can take on $B-1$ possible values, and among $\eta_1, \cdots, \eta_B$ there must be a repeated value. Let r_1 and r_2 be indices in $\{1, \cdots, B\}$ for which $\eta_{r_1} = \eta_{r_2}$ and $r_1 < r_2$.

At time t_{r_1}, the wave number of bin η_{r_1} was a_{r_1}, and all new bits arriving in bin η_{r_1} at time t_{r_1} or later will have a wave number of at least t_{r_1}. Therefore, since a_{r_2} is the wave number of bin η_{r_2} at time t_{r_2}, we must have either $a_{r_2} \geq t_{r_1}$ or $a_{r_2} = a_{r_1}$. But $a_{r_2} > a_{r_1}$, since each recomputation of the bound must increase it; therefore,

$$a_{r_2} \geq t_{r_1}$$

Also observe that

$$t_{r_2+1} \geq a_{r_2} + M$$

since otherwise it would not have been necessary to recompute the bound at time t_{r_2+1}.

Combining inequalities yields

$$t_{B+1} \geq t_{r_2+1} \geq a_{r_2} + M \geq t_{r_1} + M \geq t_1 + M$$

or simply $t_{B+1} \geq t_1 + M$. In other words, at most B recomputations can occur in a given length M window. Noting also that no recomputations can occur for the first M source bits, we see that the total fraction of source bits for which a recomputation occurs cannot exceed B/M. ❐

We now show by means of an example that the fraction of source bits for which a recalculation occurs can approach $B/(M+1)$, assuming that $M > 2B-2$. For this example, we describe a sequence of codewords arriving in the bins, where for each codeword we specify the bin it arrives in, the wave number ("start time" of the codeword), and the time at which the last bit of the codeword is used ("end time"). No flush bits are added in this example.

The sequence of codewords is indexed by i starting at $i = 0$. Let codeword i have start time $1 + \lfloor i/B \rfloor (M+1) + 2(i \bmod B)$ and end time equal to M plus the start time of codeword $i-1$. (We let the end time of codeword 0 be $2B-2$.) Let the bin indices of the sequence of codewords cycle through $\{2, \cdots, B\}$ in any fixed order. When $B = 4$ and for bin index order 2, 3, 4, the beginning of this sequence is as shown in Table A-1.

Table A-1. Bin indices, start times, and end times for the codeword sequence example.

i	Bin	Start	End
0	2	1	6
1	3	3	$M+1$
2	4	5	$M+3$
3	2	7	$M+5$
4	3	$M+2$	$M+7$
5	4	$M+4$	$2M+2$
6	2	$M+6$	$2M+4$
7	3	$M+8$	$2M+6$
8	4	$2M+3$	$2M+8$
9	2	$2M+5$	$3M+3$
10	3	$2M+7$	$3M+5$
11	4	$2M+9$	$3M+7$
		$\vdots$	

For general B, our sequence could occur, for example, if the coder design maps all bits into bin 1 (i.e., is non-recursive) and the codewords received all contain 2 source bits. The first of the source bits would be returned at the start time of the codeword, and the second would be returned at the end time of the codeword. The remaining source bits can be assumed to all be from bin 1. Many other scenarios are possible, including some with recursive coder designs.

It is straightforward to check that for $i \geq B-1$, the start time of codeword i is equal to one plus the end time of codeword $i-(B-1)$, which is the previous codeword in the same bin. Thus, our list is consistent in that there is no overlap of codewords in the same bin.

For all of the codewords beyond codeword 0, a recalculation occurs at the end time of the codeword and results in the new bound being equal to the start time of the codeword. (This is easily verified by induction.) Thus, after retrieving source bit $2B-2+k(M+1)$, the number of recalculations that have occurred is kB, or in other words the fraction of source bits for which a recalculation was performed is

$$\frac{B}{M+1+(2B-2)/k}$$

This quantity approaches $B/(M+1)$ as k becomes large.