A Finite-Precision Adaptation of Bit Recycling to Arithmetic Coding

Ahmad Al-Rababa’a
Université Laval, Canada
Ahmad.Al-Rababaa.1@ulaval.ca

Danny Dubé
Université Laval, Canada
Danny.Dube@ift.ulaval.ca

Abstract—The bit recycling compression technique has been introduced to minimize the redundancy caused by the multiplicity of encodings present in many compression techniques. It has achieved about 9% as a reduction in the size of the files compressed by Gzip. In prior work, we have proposed an arbitrary-precision technique to adapt bit recycling to arithmetic code instead of Huffman code. We have shown that this adaptation enables bit recycling to achieve better compression and a much wider applicability. We have also presented a theoretical analysis that estimates the average amount of data compression that can be achieved by this adaptation. In this paper, we propose the finite-precision version of this adaptation so that it can be implemented efficiently using conventional computer registers.

I. INTRODUCTION

Data compression aims to reduce the size of data so that it requires less storage space and less bandwidth of the communication channels. Data compression has recently started to be used to reduce the energy consumption in many wireless applications, such as wireless-networked handheld devices [1] and wireless sensor networks [2], since wireless transmission of a bit can require over 1000 times more energy than a single 32-bit computation [3]. Many compression techniques suffer from a problem that we call the redundancy caused by the multiplicity of encodings (ME). ME means that the source data may be encoded in more than one way. In its simplest form, it occurs when a compression technique with ME has the freedom, at certain steps during the encoding process, to encode the next symbol in different ways; i.e. different codewords for the same symbol can be sent to the decoder and any one of these codewords can be decoded correctly. Upon occurrence of such a situation, the default behavior of most techniques is to encode the symbol using the shortest codeword and, possibly, the least computation. Many applications suffer from ME, such as LZ77 (Lempel and Ziv, 1977) and its variants, some variants of the Prediction by Partial Matching (PPM) technique, Volf and Willems switching compression technique [4], and Knuth’s algorithm [5] for the generation of balanced codes.

The Bit Recycling (BR) technique has been introduced to reduce the redundancy caused by ME [6]. It reduces that kind of redundancy by harnessing ME in a certain way, so that it is not always necessary to select the shortest codeword, but instead, so that all the appropriate codewords are taken into account with some agreement between the encoder and the decoder. Variants of BR have been applied on LZ77 algorithm by Dubé and Beaudoin. The experimental results showed that BR has achieved better compression (a reduction of about 9% in the size of files that has been compressed by Gzip) by exploiting ME rather than systematically selecting the shortest codeword [7], [8].

The authors of BR have pointed out that their technique could not minimize the redundancy perfectly since it is built on Huffman codes (HC), which does not have the ability to deal with codewords of fractional lengths; i.e. it is constrained to generate codewords of integral lengths. Moreover, Huffman-Coding-based BR (HCBR) has imposed additional burdens to avoid some situations that affect its performance negatively. Unlike HC, Arithmetic Coding (AC) does have the capability of manipulating codewords of fractional lengths. Furthermore, it has attracted the researchers in the last few decades since it is more powerful and flexible than HC. Consequently, a new technique named Arithmetic-Coding-based BR (ACBR) has been proposed to resolve the weakness of HCBR by adapting it to AC [9]. A theoretical analysis showed that ACRB achieves perfect recycling in all cases whereas HCBR achieves perfect recycling only in very specific cases. Accordingly, significantly better compression could be obtained using ACRB.

The problem of ACRB, as proposed by the authors, is that it uses arbitrary-precision calculations, which require unbounded (or infinite) resources [9]. Hence, in order to benefit from ACRB in practice, ACRB needs to be adapted so that it can perform finite-precision calculations instead of arbitrary-precision calculations. This would make it efficiently applicable on computers with conventional fixed-size registers. This work aims to address the problems of arbitrary-precision ACRB (APACBR).

The outline of the next sections are as follows. In Section II, we briefly review the LZ77 technique, ME, the objectives of HCBR and its weakness, the principle of APACBR, and the problems of APACBR. In Section III, we present a finite-precision technique that addresses the problems of APACBR. The new proposed technique consists of two algorithms: the coder and the decoder. Finally, the conclusion and future work are given in Section IV.

II. BACKGROUND

A. The principle of LZ77, HCBR, and ME

The reasons of the remaining redundancy in compressed data are: inappropriate modeling of the data, incorrect (or inaccurate) random source statistics, and ME. The concern of this work is the third reason: ME. Let us show an instance of redundancy caused by ME by using the following LZ77 example. LZ77 is a compression technique that compresses a
string of characters, S, by transmitting a sequence of messages. A message is either a literal message, denoted by [c], which means that the next character is c, or a match message, denoted by [l, d], which means that the next l characters are identical to those located at distance d prior to the current position in S. For example, let S be “abbaaabbababbb”, the underlined substring is the prefix that has been encoded so far. The next character to be encoded is ”a”, which can be encoded by transmitting the literal message [a]. However, the encoder can also encode, for instance, ”abb” by transmitting any one of the match messages [3, 3], [3, 6], and [3, 11], since ”abb” has three copies at the distances 3, 6, and 11 in the underlined substring. LZ77 typically selects the longest match (”abb”) at the closest distance (d = 3), therefore the match message (3, 3) will be transmitted and the encoder proceeds to the last ”b”. It is clear that LZ77 has the freedom to encode ”abb” by selecting any message from the set of the equivalent messages \( M = \{ M_1, M_2, M_3 \} \), where \( M_1 = (3, 3), M_2 = (3, 6), \) and \( M_3 = (3, 11) \). These messages are called the equivalent messages since any message in \( M \) can be used to encode ”abb”. Accordingly, many different sequences of messages may be transmitted to describe S, and any possible sequence will be decoded correctly. It is clear that this property represents an instance of ME.

BR aims at improving code efficiency by exploiting the redundancy caused by ME [6]. In BR, the compressor is not restricted to select the default choice, i.e. the shortest codeword, and to ignore the other choices. Instead, thanks to some agreement between the compressor and decompressor, it uses ME to implicitly carry information from the compressor to the decompressor. Let us illustrate this using the following example, which will be used as a running example in the remainder of this paper. Assume that, at time t, the alphabet \( \alpha \) is \( \{ m_i \}_{i=0}^{5} \) and the corresponding distribution and Huffman codewords for \( \alpha \) are as shown in Table I. Let \( \mathcal{M}_\alpha = \{ M_1, M_2, M_3 \} \) be the set of equivalent messages at time t, where \( M_1 = m_1, M_2 = m_3, \) and \( M_3 = m_5 \) (similar to the equivalent messages for ”abb” in the above example). Suppose that the string to be encoded, S, can be described first using any message in \( \mathcal{M}_\alpha \) at time t, and then, at time t + 1, using solely message \( m_3 \) (without equivalents). The Huffman codeword Huffi corresponding to message \( m_i \) is generated using HC, based on the count of occurrences (Cnti) of \( m_i \). The default behavior (without recycling) is that the equivalent message with the shortest codeword, \( M_1 = m_1 \), gets selected and Huff1 (i.e. 11) is transmitted to the decoder.

<table>
<thead>
<tr>
<th>Message</th>
<th>Count (Cnt)</th>
<th>Probability (p)</th>
<th>Cumulative probability (( Q_i ))</th>
<th>Huffman codeword (Huffi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_0 )</td>
<td>3</td>
<td>0.064</td>
<td>0.064</td>
<td>101</td>
</tr>
<tr>
<td>( m_1 )</td>
<td>15</td>
<td>0.319</td>
<td>0.064</td>
<td>11</td>
</tr>
<tr>
<td>( m_2 )</td>
<td>12</td>
<td>0.255</td>
<td>0.383</td>
<td>10</td>
</tr>
<tr>
<td>( m_3 )</td>
<td>3</td>
<td>0.064</td>
<td>0.638</td>
<td>0100</td>
</tr>
<tr>
<td>( m_4 )</td>
<td>8</td>
<td>0.120</td>
<td>0.792</td>
<td>000</td>
</tr>
<tr>
<td>( m_5 )</td>
<td>6</td>
<td>0.128</td>
<td>0.872</td>
<td>011</td>
</tr>
<tr>
<td>Total</td>
<td>47</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let us evaluate the performance of HCBR in the previous example. The average net cost NC of the set of equivalent messages, \( \mathcal{M}_\alpha = \{ M_i \}_{i=1}^{n} \), is given by:

\[
NC = \sum_{i=1}^{n} (c_i - |r_i|) \times \frac{1}{2^{|r_i|}},
\]

where \( n \) is the number of the equivalent messages, \( c_i \) is the cost of \( M_i \), i.e. the length of its codeword \( (c_i = |\text{Huff}_i|) \), and \( |r_i| \) is the length of the recycled codeword. So NC for \( \mathcal{M}_\alpha \) is 1.25 bits, which is less than the default cost \( (C_{\text{default}} = 2 \text{ bits}) \). This benefit is brought by HCBR but did BR achieve the perfect (maximum) recycling by this average net cost? To answer this question, we first need to know what is the ideal (minimum) average net cost of \( \mathcal{M}_\alpha \) according to the associated probabilities. The self-information of \( \mathcal{M}_\alpha \), say T, represents the minimum average net cost of \( \mathcal{M}_\alpha \), and T is given by:

\[
T = -\log \sum_{i=1}^{n} p_i.
\]

The value of T in our example is 0.97 bit, so HCBR did not achieve perfect recycling. The reason behind this is that HC is constrained to generate recycled codewords of integer lengths, which correspond to probabilities that are powers of \( \frac{1}{2} \) only. Therefore, HCBR could not achieve perfect recycling due to the nature of HC. Moreover, HCBR has imposed the additional burden to avoid some situations that affect its performance negatively. AC is more flexible and it does have the ability to utilize the ratio between the messages’ probabilities fractionally and to recycle fractions of bits. Accordingly, the authors have proposed APACBR to resolve this weakness and to improve the code efficiency and the flexibility of BR [9]. Next, we explain the principle of APACBR.
B. The principle of APACBR

We consider the same running example to describe the principle of APACBR. Let us first describe the model that will be used in our technique. The alphabet shown in Table I can be decomposed into the corresponding binary tree depicted in Figure 1, which represents the statistical model at time $t$. This decomposition enables the model to accommodate a larger and more skewed alphabets. The root node in Figure 1-A, contains the total counts ($47$) of the alphabet messages, the leaves of the tree represent $\alpha$. The shaded leaves represent $M_{\Xi}$ at time $t$. The skeleton tree depicted in Figure 1-B is derived from the main tree and describes the relative weights and locations of equivalent messages. The trees in Figure 1-A and Figure 1-B will be used by the model to provide the encoder/decoder with necessary information to perform coding/decoding and recycling, respectively.

The statistical model describes the message to be encoded by transmitting a sequence of binary events $B$ associated with the corresponding probability, $P_0$ or $P_1$ (we assume $P_0 + P_1 = 1$), that are formed by traversing from the root to the leaf that corresponds to the message to be encoded. For example, to encode $M_2 = m_3$ as one of the equivalent messages at time $t$, the model has to send the following sequence of coding orders of the form $(B, P_0, P_1)$ to the encoder: $(1, \frac{11}{20}, \frac{11}{20})$, $(0, \frac{11}{20}, \frac{11}{20})$, and $(1, \frac{11}{20}, \frac{11}{20})$. Since ACBR uses AC to encode $S$, the encoder starts executing each order by gradually dividing the unit interval $[0, 1]$ into two subintervals according to $P_0$ and $P_1$. The interval of event 0 is $I_{E0t}$ and that of event 1 is $I_{E1t}$. Only one subinterval is kept, from time $t$ to time $t'$, according to $B$, as shown in Figure 2. Similarly, the skeleton tree is used to transmit a sequence of recycling orders to perform recycling as we will show later in this paper.

Let the subinterval corresponding to the equivalent message $M_i$ at time $t$, denoted by $I_{t}$, APACBR aims to let any $M_i$ from $M_{\Xi}$ be selected according to the next message, providing that the next message, at $t + 1$, will be encoded using the total of $M_{\Xi}$ sub-intervals $I_{t+1}$ instead of the subinterval of the selected message $I_{t+1}$ as shown in Figure 2, which results in fewer bits to be sent to the decoder to describe any message at $t + 1$. The length of the interval at time $t$ represents the self-information of the available equivalent messages; i.e. $\#I_{t+1} = \#I_{t+1}^1 + \#I_{t+1}^2 + \#I_{t+1}^3$.

Let us assume that the location (cumulative probability) of the messages at time $t + 1$ is indicated by the arrow in Figure 2. According to the location of the arrow, one and only one of the equivalent messages will be selected (and encoded) as follows. Since the arrow points to $I_{t+1}$ and to $I_{t+1}^1$ at the same time and $I_{t+1}^2$ belongs to $M_2$ (in its scope), then the encoder selects message $M_2$. The encoder provides the new interval $I_{t+1}$ for the next message to be encoded instead of $I_{t+1}^1$. The same thing for $M_1$ and $M_3$ as follows. If the arrow points above $I_{t+1}^2$, so it points to $I_{t+1}^1$, which is the part of $I_{t+1}$ related to $M_3$, accordingly, the encoder selects $M_3$ and so on. Hence, each $M_i$ in $M_{\Xi}$ has an opportunity to be encoded using $I_{t+1}$ instead of $I_{t+1}$ but one and only one of them will be used according to the cumulative probability of next message to be encoded. Widening the interval from $I_{t+1}^2$ to $I_{t+1}$ represents the arithmetic recycling according to equation (2).

On the other side, the decoder undoes what the encoder did based on the aforementioned arrangement as follows. The decoder at time $t$ will have the same information represented by the tree in Figure 1 based on the already decoded string (before $t$). The decoder starts decoding using the value represented by the position of the arrow with respect to $I_t$. The model at this point has to provide the decoder with $P_0$ ($\frac{11}{20}$) so that the decoder can accordingly split $I_t$ into $I_{E0t}$ and $I_{E1t}$, now the decoder can determine that the arrow is pointing to the upper part, i.e. $I_{E1t}$, accordingly the decoder tells the model that the first decoded binary event $B$ is 1, which tells the model that the message to be encoded is located to the right in the tree in Figure 1. The decoder continues using the same procedure until the model reaches the leaf ($m_3$) at time $t'$, where the decoder can realize that the decoded message ($M_2 = m_3$) has two other equivalents, $M_1$ and $M_3$ (recall LZ77), accordingly, it has to rebuild $I_{t+1}$ based on the decoder’s implicit knowledge about $M_{\Xi}$ represented by the skeleton tree Figure 1-B; i.e. the decoder can retrieve the necessary information that enables the decoder to decode the next message according to $I_{t+1}$ instead of $I_{t+1}^2$. At time $t + 1$, the same position of the arrow with respect to $I_{t+1}$ will be used to decode the next message as described above and so on.

The authors have compared the performance of ACBR and HCBR in terms of the compression efficiency and time complexity [9]. We found that the average net cost, NC, of any message in $M_{\Xi}$, that can be achieved by HCBR, NC$_H$, is bounded by $T \leq NC_H \leq C_{default}$, and $NC_H \approx T$ only when the messages in $M_{\Xi}$ have equal probabilities, while NC of ACBR, NC$_A \approx T$ in all cases, regardless of the ratio between the probabilities of the messages in $M_{\Xi}$.
Furthermore, in recent work, the authors have used both HCBR and APACBR as the means to reduce the redundancy caused by ME in plurally parsable dictionaries designed by Savari [10], [11]. We were able to reduce this redundancy significantly, and we have theoretically shown that ACBR (in general) is more efficient and flexible than HCBR, but due to some shortcomings in the applicability of APACBR, we could not evaluate its performance in practice. However, we next discuss these shortcomings in detail.

C. The problems and the associated needs of APACBR

The main problem of the technique described above is that it uses arbitrary-precision AC (APAC) to encode each message at a time, which entails too many resources and makes it impractical, therefore we need to adapt the encoder to finite-precision AC (FPAC) so that we can reduce the computational requirements and it can then be applied in practice. Next, we discuss (using the same example) the basic needs, settings, and consequences that are required to address this problem.

Traditional FPAC encodes one message, \( m_i \), at a time, as we have shown above, the ACBR coder has to be able to encode and keep track of more than one message \( (M_{\infty}) \) at a time, so that the ACBR coder can exploit the self information of \( M_{\infty} \), and consequently, achieve more compression. Hence, the ACBR coder has to proceed non-deterministically (ND) by assigning a separate thread, \( \Theta_t \) for each \( M_i \in M_{\infty} \).

In FPAC, the unit interval \([0, 1]\) gets mapped into \([0, N] = [0, 2^b]\), where AC uses \( b \)-bit registers to encode \( S \), \((L, H)\) is defined as \( \{x \mid L \leq x < H\} \). The interval \([L, H]\) is updated for each encoding step, when the width of the interval shrinks to a certain limit, the encoder according to the values of \( L \) and \( H \), performs one of the three types of upscaling as follows: E1, E2 or E3 when \((L, H)\) lies in \([0, \frac{N}{4})\), \([\frac{N}{4}, \frac{N}{2})\), or \([\frac{N}{2}, N]\) respectively. According to FPAC, the encoder at time \( t \), in the above example, needs first to upscale (enlarge) \( I^2_0 \) so that it becomes large enough to encode the message at time \( t + 1 \). At time \( t + 1 \), the encoder needs to merge the intervals of the other equivalent messages, but it has to consider the new length, \#\( I^2_{t+1} \), as a reference after upscaling. As we assumed above that \#\( I^1_t = 5 \times \#I^2_t \) and \#\( I^3_t = 2 \times \#I^2_t \), then the total length of merged interval at time \( t + 1 \) would exceed the used finite-precision limits. To address this raised problem, we need to downscale the interval so that we can accommodate the intervals of the other equivalent messages within the used finite limits. Hence, we propose next a finite-precision variant of ACBR (FPACBR) that addresses the aforementioned problem and the associated needs.

III. Description of FPACBR

In this section, we propose FPACBR, which has the following main features. It has to be based on AC and to proceed ND for the reasons explained above, and it can be easily interfaced with the models of several different compression techniques that suffer from the redundancy caused by ME. We use the same running example to explain the pseudo-code shown in Figure 3 for the encoder algorithm, on lines 1–63, and the decoder algorithm, on lines 65–84, with the help of Figure 4, which illustrates the steps of encoding \( M_2 = m_3 \) at time \( t \) followed by \( m_3 \) at time \( t + 1 \) based on a sequence of encoding and recycling orders received from the model. The encoder interprets the received ordered into a compressed binary stream \( \sigma \) that will be sent to decoder.

In APACBR, the limits of the intervals and the arrow were regarded as points (real numbers) in the unit interval \([0, 1]\). Here, we need to map these points into tiles, and the arrow into pointed tile. The FPACBR coder uses a convention different from that used by FPAC, which will be explained throughout this section.

Let \#\( I \) be the number of tiles occupied by an interval \( I \). \#\( I \) is now an integer. The coder uses a global interval \( G \) of at most \( 2^b \) tiles, and inside \( G \), the encoder uses a fixed central interval, \( F = \{\text{Bot, Top}\} \) of length \#\( F = \{\text{Top} - \text{Bot}\} = 2^b - 2 \) tiles. The working interval \( W = \{L, H\} \) is initialized to \([\text{Bot, Top}] \). The size of \( W \) increases and decreases according to the coding steps and it is bounded by \( 1 = B \leq \#W \leq 2^b - 1 \). L and H are allowed to slide outside of \( F \) for a limited number of tiles, \( B = 2^b - 1 \). The interval \( F \) contains the active interval, \( A = \{a_1, a_2\} \subseteq W \), which is the portion of \( W \) that belongs to the current thread \( \Theta_{\text{cur}} \). The ratio of \( A \) to \( W \) in \( \Theta_{\text{cur}} \) represent the relative weight and location of \( M_i \) to the weights and locations of \( M_{\infty} \). Accordingly, the inactive (rest of) interval will be \( I_{\text{inact}} = W - A \), which is the portion of \( W \) that belongs to other concurrent threads. Note that \( I_{\text{inact}} \) need not be an interval.

To encode the first message \( (M_2 = m_3) \) at time \( t \), the model sends the same sequence of coding orders that has been sent to the APAC encoder: \((1, \frac{18}{17}), (0, \frac{15}{207}), (1, \frac{12}{17})\). The encoder and decoder follow the same principle described in APACBR, but using a finite-precision coding/decoding. Thus, the encoder gradually divides the initial interval \( I_{\text{init}} = \{L, H\} = [0, 16] \) into two subintervals, \( I_{\text{Ext}_0} \) and \( I_{\text{Ext}_1} \), of integer lengths according to \( P_0 \) using the rounding rule stated in line 2 in the pseudo-code, the encoder then considers either \( I_{\text{Ext}_0} \) or \( I_{\text{Ext}_1} \), according to \( B \) to continue executing the next coding orders as stated on lines 3–7 in the pseudo-code. The steps of encoding the aforementioned sequence of coding orders are illustrated in Figure 4, from steps 1 to 5. The decoder uses the same procedure followed by the encoder to split the current interval according to \( P_0 \), and according to the arrow location, it decodes the associated binary event. The
Let us show this by executing the first coding order Figure 4, are kept temporarily in register

```c
while #W ≤ Half do
  if A ⊆ [Bot, Mid] then
    W ← E1(W)
  else if A ⊆ [Mid, Top] then
    W ← E2(W)
  else
    fork Θcur into Θ0, Θ1:
    In Θ0: trim A to [L, Mid]
    In Θ1: trim A to [Mid, H]

Emit(w)

procedure E1([l, h]):
return \([2 \times l - \text{Bot}, 2 \times h - \text{Bot}]\)

procedure E2([l, h]):
return \([2 \times l - \text{Top}, 2 \times h - \text{Top}]\)
```

Fig. 3. Pseudo-code for the algorithms of the encoder and the decoder.

arrow location here is represented by a finite number \((b - 2)\)
of bits of the compressed stream \(σ\) as we will show next. Let
us show this by executing the first coding order \((1, 18)\)
at time \(t\). Initially, at time \(t\), \(I_{init} = 16\), and \(P_0 = \frac{18}{47}\), so
\(I_{Evt_0} = I_{init} \times P_0 = 6.13\), the result \((6.13)\) gets
rounded to 6, and therefore \(I_{Evt_1} = 16 - 6 = 10\). According
to \(B = 1\), the encoder ignores \(I_{Evt_0}\) and considers \(I_{Evt_1}\) as
the current interval to execute the next orders. Thereby \(W\) is
updated to \([L, H] = (6, 16)\).

During the encoding/decoding process, if \(W\) shrinks to less
than or equal to \(\text{half}\) the length of \(F\), that is \(\text{Half} = \frac{\text{Top} - \text{Bot}}{2}\),
the coder is triggered to upscale \(W\) so that it is scaled up
above \(\text{Half}\) tiles as stated on lines 11–23 in the pseudo-
code, the encoder can then continue executing the next orders.
The coder performs only two types of upscaling, \(E1\) and \(E2\)
according to the location of \(A\) and the mid point of \(F, \text{Mid} = \frac{\text{Top} + \text{Bot}}{2}\), as follows. \(E1\) is performed if \(A \subseteq [\text{Bot, Mid}]\),
yielding 0 to \(σ, E2\) is performed if \(A \subseteq [\text{Mid, Bot}]\), yielding 1
to \(σ\). Such yielded bits (1010), indicated at the lowest row in
Figure 4, are kept temporarily in register \(w\). Since the coder
proceeds ND, we choose to avoid \(E3\) upscaling completely.
When \(A\) straddles \(\text{Mid}\), instead of using \(E3\), we take advantage of the ND
process, as shown at steps 3 and 4 in Figure 4 and as
stated on lines 21–23 in the pseudo-code. This is achieved by
splitting \(Θ_{cur}\) into two new separate threads, \(Θ_0\) and \(Θ_1\),
each with their specific active interval. The encoder will eventuall
kill the non-proper thread according to the upcoming coding
orders.

The decoder uses the same two types of upscaling, \(E1\)
and \(E2\), but in different way as described in the pseudo-code
on lines 73–79 and as follows. Initially, the decoder loads
register \(v\) of size \(b - 2\) (i.e. 4) bits with the first 4 bits of
\(σ (v = 1010)\). The decoder performs \(E1\) or \(E2\) upscaling
according to the Most Significant Bit of \(v, \text{MSB}(v)\). For each
upscaling, the contents of \(v\) is shifted left by one bit and one
bit is consumed from \(σ\) to become the Least Significant Bit
of \(v, \text{LSB}(v)\).

The situations at step 8 in Figure 4, represent the greatest
challenge of FPACBR, since the encoder needs to merge the
corresponding intervals of the other equivalent messages to the
current interval \(W = [0, 16]\) after upscaling, and the whole
merged interval should be accommodated within the limits of
\(G\). To do so, another procedure named \(\text{RecycleE}\) is required.
The main purpose of \(\text{RecycleE}\) is to downscale \(W\) until it
becomes small enough to accommodate the whole merged
interval within \(G\) as described in the pseudo-code on lines 40–
58. Therefore, the model has to provide \(\text{RecycleE}\) with a sequence of \textit{recycling orders} that describe the relative weights and
directions of \(M\) using the \textit{skeleton} tree of \(M\) shown in
Figure 1-B as follows. Let us describe the recycling orders
using our example. Starting from the internal node labeled \(9/24\)
\((9/24\) in the picture), which connects the current message \(m_3\)
with the first (closest) neighbor \(m_5\), the position of the first
neighbor \(m_3\) is to the right, let us say \(R = 1\), of the internal
node labeled \(9/24\), and the weight of the left branch is \(P_0 = 3/9\).
In other words, \(R\) tells the encoder/decoder if the interval to
be merged is above \((R = 1)\) or below \((R = 0)\) of the current
interval, and the associated \(P_0\) tells the encoder/decoder about
the ratio of the current interval to the interval to be merged. So
the recycling order \((1, \frac{3}{2})\) of the form \((R, P_0)\) describes \(m_5\), as the first equivalent message, and similarly, the second recycling order \((0, \frac{15}{22})\) describes the internal node labeled \(\frac{9}{22}\) as the current message and \(m_1\) as the closest neighbor node.

The encoder and decoder use two types of downsizing: \(S_2\) and \(S_1\). The \(S_2\) downsizing undoes \(E_2\) and \(S_1\) undoes \(E_1\) upsizing. Let \(\rho\) be a variable that keeps the recycled bits, that need not be sent to \(\sigma\). The variable \(\rho\) is initialized to \(\epsilon\), where \(\epsilon\) is the empty string. Each \(S_2\) and \(S_1\) yields 1 and 0 to \(\rho\), respectively. Accordingly, from steps 9 to 11, \(\text{RecycleE}\) interprets the recycling orders \((1, \frac{3}{2})\) then \((0, \frac{15}{22})\) into the following sequence of downsizing: \(S_1, S_1,\) and \(S_2\), yielding the corresponding binary sequence, 100, to \(\rho\). The \(\text{RecycleD}\) procedure is identical to \(\text{RecycleE}\) except that it inserts the bits that have been removed by the coder from \(\sigma\), just to the left of \(\sigma\), this entails to change only two lines of the programming of \(\text{RecycleE}\), as described in the pseudo-code on lines 82–84.

In addition to the main function of the recycling procedure (accommodating the whole merged interval within \(G\)), let us explain why the coder proceeds in this certain way. If we look at \(W\) at step 11 and try to upscale the active interval of the message being encoded, \(A = [a_1, a_2] = [8, 10]\), from right to left; i.e. upscale \(A\) in reverse order from step 11 to step 8, then the yielded bits that need to be sent to \(\sigma\) are the same as the bits kept in \(\rho\)! Which means that if \(\Theta\) is the proper thread for the next message, then the first three bits of the next message will be \(100 = \rho\), accordingly, these bits can be recycled, because it can be inferred implicitly by the decoder as described above.

We assumed that the next message to be encoded at time \(t + 1\) is \(m_3\) without equivalents. The coder encodes \(m_3\) as described above from steps 12 to 18. At step 18, procedure \(\text{Emit}\) is called to check if the current thread is to be continued (the proper thread of the message being encoded) or to be killed (not the proper thread). The thread is to be continued if the first bit of \(\rho\) (100) matches the first bit of \(w\) (1000) and, accordingly, the matched bits are removed (recycled) from \(\rho\) and \(w\). (The first bit of a string \(w\) is extracted using \(\text{head}(w)\) and the rest of \(w\) is extracted using \(\text{tail}(w)\).) Otherwise, if no match is found, the current thread is then terminated since it will not be the proper thread to encode \(m_3\) at time \(t + 1\). As a result, the encoder will send the compressed stream \(\sigma = 10100\) to the decoder, and the decoder can decode \(\sigma\) into the string \(S = M_2m_3\).

The pseudo-code, on lines 60 and 63, uses two types of rounding in procedures \(S_1\) and \(S_2\): floor and ceiling. The floor and ceiling rounding operations are used to round down \(L\) and round up \(H\), respectively. The reason behind using these two types of optimistic rounding is to ensure that there is no portion of the downscaled interval that will not be used (covered) by any concurrent thread, which would lead to the existence of a specific message that can not be encoded permanently.

The pseudo-code, on line 3, covers a special (highly skewed) case when the ratio of the received order occupies less than one tile or the entire available space (\(#I_{cut}\)), to avoid assigning zero or one probability to any highly skewed event. Similarly, the pseudo-code, on line 54, handles another special (highly skewed) case for \(M_E\), where the recycling procedures stop downsizing if \(#I_{cut}\) in \(A\) reaches the minimum, \(B = 1\), and allocates the maximum possible size to the highly probable message. Notice that the proposed technique was able to encode/decode \(M_2m_3\) and to improve the code efficiency using fixed-size (\(b\)-bit) registers, which is the main goal of FPACBR.

IV. Conclusion

In prior work, we have theoretically shown that ACBR can achieve a significant amount of better compression than HCBR. In this work, we have proposed FPACBR, which can be implemented and applied in practice. FPACBR is easy to interface with the compression techniques that suffer from the problem of redundancy caused by ME. As a future work, we intend to implement and apply FPACBR on the proper applications mentioned in this paper, in order to evaluate and measure its performance in practice comparatively with the results obtained by HCBR.

REFERENCES


ABBREVIATIONS

ME Multiplicity of Encodings
ACBR AC-based BR
HC Huffman Coding
APAC Arbitrary-Precision AC
AC Arithmetic Coding
FPAC Finite-Precision AC
BR Bit Recycling
APACBR Arbitrary-Precision ACBR
HCBR HC-based BR
FPACBR Finite-Precision ACBR
ND Non-Determinism (or Non-Deterministic or Non-Deterministically)