Automatic Generation of C++/Java Code for Efficient Binary Arithmetic Coding

Danny Hong, *Student Member, IEEE,* Alexandros Eleftheriadis, *Senior Member, IEEE*

Abstract

We first present a set of simple parameters for describing any binary arithmetic coder. We then propose a highly efficient, lookup table-based, renormalization method that can be used by any binary arithmetic coder with the follow-on procedure. It replaces the time-consuming branching operations in the renormalization process with table lookups and some simple bit-wise operations. We show that our new renormalization method outperforms the currently used ones. We have modified Flavor, a software tool that automatically generates C++ or Java code for parsing and generating bitstreams, so that the code for efficient binary arithmetic coding can also be generated. The code is automatically generated based on the parameters and the proposed renormalization method.

Index Terms

Arithmetic codes, software tools, optimization methods

This material is based upon work supported in part by the National Science Foundation under Grant ACI-0313116.

Automatic Generation of C++/Java Code for Efficient Binary Arithmetic Coding

I. Introduction

Huffman coding [1] is arguably the most widely used statistical compression mechanism for media representation (e.g., Group 3 [2] and Group 4 [3] fax, MPEG-1 [4], MPEG-2 [5], and etc.). It is proven to be optimal among instantaneous codes, as it can represent any given random variable within 1 bit of its entropy (tighter bounds are shown in [6], [7], [8]). Arithmetic coding [9], [10], derived from Elias coding [11], [12], is another statistical coding method proven to yield better compression than Huffman coding when the probabilities of the symbols are not in powers of 0.5; however, it has not been widely used for media representation due to its complexity and patent issues. The very first, practical arithmetic coder was developed for compressing bi-level images (the Skew coder [13], [14] and the Q-Coder [15]), as Huffman coding cannot compress binary symbols, unless groups of symbols are coded at a time. Run-length coding (e.g., Golomb coding [16]) is a good alternative coding method for binary symbols, when the probability of one symbol is much higher than the other. Nevertheless it is an optimal coding method for sources with geometric distribution, and for the best result for all possible binary source sequences, using an adaptive binary arithmetic coder (BAC) is preferred.

Even today, most of the practical arithmetic coders deal solely with binary alphabets: binary arithmetic coding is computationally simple and it makes using higher-order conditioning models feasible. In some cases it is only natural to assume a binary source. For instance, JBIG [17], JBIG2 [18], and MPEG-4 shape coding [19] focus on coding of bi-level images/videos (JBIG and JBIG2 can also be used for grayscale images where bit-plane by bit-plane coding is applied), and for JPEG2000 [20] and MPEG-4 texture coding [19], bit-plane coding is ultimately applied. On the other hand, arithmetic coding can optionally be used in JPEG [21] and H.264 [22], and in these cases, each symbol is first binarized so that a BAC can be used. Despite such wide use of BACs, different BACs are generally incompatible with each other, e.g., a code string generated by a JBIG arithmetic encoder cannot be correctly decoded by an arithmetic decoder designed for MPEG-4 shape coding. As a remedy, we present a unique solution that unifies BACs; we define a set of parameters that can be used to automatically generate different variants of BACs.

Arithmetic coding can be separated into two main units: *modeling* and *coding*. The modeling unit appropriately selects one or more structures for conditioning events, and gathers the relative frequencies of the conditioned events [13], [23], which correspond to the event probabilities. The latter part of the modeling unit can also be regarded as a unit of its own (e.g., the *statistics* unit in [24]), and sometimes it is incorporated as part of the coding unit. Modeling, by itself, is a huge topic and there are numerous effective models that have been introduced. The H.264 standard alone defines more than 300 models to account for different structures each bit of the binarized syntactic elements might have. Consequently, unifying modeling is an extremely difficult task (if not impossible), and models often depend on the semantics of the data. Here, we focus only on the coding part. For a very good introduction on arithmetic coding – including its history – refer to the 1984 paper by Langdon [9].

Flavor [25], [26] is a language that has been developed to describe the syntax of any compressed bitstream so that the bitstream parsing and generation code can be automatically generated. Flavor already has a construct for describing variable-length codes and we complement it by introducing a new construct `bac`, with a set of parameters, for describing BACs. Using the `bac` construct any BAC can be easily described, and the Flavor translator automatically generates the corresponding C++/Java code. As a result, application developers can solely concentrate on the modeling part, which has been shown to have higher impact on the compression effectiveness of the BAC compared to the coding part. In the next section we briefly describe the main concept behind binary arithmetic coding, and in Section III we present the set of parameters needed to describe any BAC. Most of the work described in Section III has been introduced in [27].

The quasi-coder introduced by Howard and Vitter [28], [29] is the fastest BAC, as it replaces all operations with simple table lookups. Nevertheless, it requires too much memory and is impractical to build a quasi-coder that is compatible with, for example, the M coder used in the CABAC entropy coding scheme of the H.264 video coding standard [22], [30], [31]. In Section IV we introduce a novel, highly efficient, lookup table-based, renormalization method (semi-quasi-renormalization), that requires much less memory than the renormalization method used by the quasi-coder. Then in Section V we discuss some experimental results, which show that our proposed renormalization method is much faster than the ones currently being used. Finally, we conclude with Section VI.

II. Background

A high-level pseudo-code describing the basic concept of binary Elias coding is depicted in Figure 1. The variable R represents the current interval length (initially 1) and it is divided into two subintervals with lengths R_0 and $R_1 = R - R_0$, according to the probabilities (P_0 and $P_1 = 1 - P_0$) of the two possible symbols 0 and 1. The variable L represents the lower bound of the current interval $[L, L+R)$ and if the symbol 0 is being coded, then assuming that the top interval corresponds to the symbol, the new interval is $[L + R_1, L + R_1 + R_0)$; likewise, for the symbol 1, the new interval is $[L, L + R_1)$. For coding of each symbol, the current interval gets subdivided and at the end, the minimum number of bits that can uniquely represent the final interval gets output as the code string. For decoding, the variable V represents the code string and the decoder essentially mimics the encoding process to deduce the original symbols. The variable x represents the current symbol being encoded/decoded.

```
Encoding                 Decoding
1 R0=R*P0                1 R0=R*P0
2 R1=R-R0                2 R1=R-R0
3 if(x==1)               3 if(V-L<R1)
4   R=R1                 4   R=R1, x=1
5 else                   5 else
6   L=L+R1, R=R0         6   L=L+R1, R=R0, x=0
```

Fig. 1. Binary Elias coding.

A. Integer Binary Arithmetic Coding

To overcome the precision problem inherent in Elias coding, most of the practical arithmetic coders are implemented using integer arithmetic with renormalization [10], [15]. Although it is possible to use floating-point numbers, integer arithmetic is preferred for its simplicity and better portability. As a consequence of using integers, the probabilities of the symbols are represented by their respective counts (C_0 and C_1), and a pseudo-code for the integer binary arithmetic coding process is shown in Figure 2.

```
Interval subdivision
1  R0=R*C0/(C0+C1)
2  R1=R-R0
3  if(x==1)
4    R=R1
5  else
6    L=L+R1, R=R0
Renormalization
7  for( ; ; )
8    if(L>=HALF)                       //HALF=2^(p-1)
9      L-=HALF, bpf(1)
10   else if(H<=HALF)                  //H=L+R
11     bpf(0)
12   else if(L>QTR1 && H<=QTR3)        //QTR1=2^(p-2)
13     L-=QTR1, F++                    //QTR3=3*QTR1
14   else
15     break
16   L<<=1, R<<=1
```

Fig. 2. Integer binary arithmetic coding.

As shown in Figure 2, arithmetic coding consists of interval subdivision and renormalization. The subdivision process continuously decreases the R value, and to prevent R from getting too small, it gets renormalized. Here p, referred to as the precision of the BAC, corresponds to the register size for L and R, where $[0, 2^p)$ is equivalent to the full interval $[0, 1)$. All renormalization methods consist of $b > 0$ doublings of L and R and output (or input, in the case of decoding) of b code bits. After subdivision, if the interval lies in the upper or lower half of the full interval (Line 8 and Line 10 of Figure 2), then the leading bit of the code string representing the interval must be 1 or 0, respectively. If the interval, however, straddles the midpoint of the full interval (Line 12 of Figure 2), then there is no way of knowing what the leading bit is. This is known as the carry-over problem [15], and the renormalization method shown in Figure 2 resolves this problem with the follow-on procedure [10]. The procedure basically keeps track of the number of times L and R have been doubled, without outputting a code bit (we refer to this number as the number of follow bits F). Once the interval escapes the midpoint, and lies in the upper or lower half of the full interval, the retained bits are output. The `bpf()` function is equivalent to the `bit_plus_follow()` function described in [10].

The set of parameters needed to describe any integer BAC is described in Section III-A.

B. Fast Binary Arithmetic Coding

The biggest drawback of using arithmetic coding has been its high complexity (too many operations performed per symbol), and many fast BACs (FBACs) that replace some operations with table lookups have been proposed. Most of the FBACs use a set of discrete values for the subinterval lengths, which can be pre-calculated; this avoids the two multiplicative operations shown in Line 1 of Figure 2. For example, the Q-Coder [15] and its variants [17], [18], [21], [20], [32], [33] work with a table, the R table, of pre-calculated values for R_{LPS}, the subinterval length for the least probable symbol (LPS). More precisely, the table contains the possible probability values for the LPS – P_{LPS}, and the current interval length R is maintained to be in the range $[0.75, 1.5)$. Consequently, R is assumed to be approximately 1 and $R_{LPS} = R * P_{LPS} = P_{LPS}$; for the most probable symbol (MPS), $R_{MPS} = R - R_{LPS}$. As a result, the coder no longer needs to know the probabilities of the symbols (or the frequencies of the symbols), but an index i into the R table so that an appropriate R_{LPS} value can be used.

The index i corresponds to the current probability state, i.e., the current probability values for the two symbols. Here, the statistics unit, which updates i, is incorporated into the Q-Coder, and consequently, the probability transition rules for obtaining the next i has to be specified.

A general version of the Q-Coder is the M coder [34] specified in the H.264 standard. In the M coder, rather than assuming R to be always equal to 1 (substituting $0.75 \leq R < 1.5$ by 1 is rather a crude approximation), R is better approximated by allowing $U > 1$ different values. For example, the M coder in H.264 allows R to take $U = 4$ different values. Then the values of the next R_{LPS} for, not one, but four different values of R are pre-specified. As a result, the R table is bigger than the one used by the Q-Coder; but, the M coder yields better compression. In the M coder, the R table is accessed by two indices – q and i. The index $q \in [0, U)$ approximates the current interval length, and the index i determines the LPS subinterval length R_{LPS}. To maximize the coding speed of the M coder, U is restricted to be a power of 2, i.e., $U = 1 << u$ (u is a non-negative integer), so that q can be simply calculated as $q = (R >> (p - 2 - u))\&((1 << u) - 1)$ [34].

The set of additional parameters needed to describe FBACs is defined in Section III-B.

C. Quasi-Coding

The FBACs described in the previous subsection simplify the subdivision process by replacing the multiplicative operations in Line 1 of Figure 2 with a table lookup. The quasi-coding concept introduced by Howard and Vitter [28], [29] further decreases the complexity of the BACs by also simplifying the renormalization process. Hence a quasi-coder is the fastest BAC, as it replaces all arithmetic operations with table lookups. The key idea is to limit the precision by allowing only small number of states; using integer arithmetic, the full interval $[0, 1)$ is represented by $[0, N)$, and if $N = 2^p$ is small, the number of states is small, and thus table-based arithmetic coding is feasible. We refer to the current interval $[L, H)$, where $H = L + R$, as the state of the coder, and a unique state number q can be assigned to each valid state. By definition, a valid state is any state where $H > L$, and there are a total of $2^{2p-1} + 2^{p-1}$ valid states.

The valid states can be divided into *terminal states* and *non-terminal states*. The terminal states are the states that do not require renormalization, and the non-terminal states are the ones that do. After encoding/decoding of each symbol, due to renormalization, we are always at a terminal state. If we assume the renormalization method shown in Figure 2, then for a given precision p, there are a total of $3(2^{2p-4})$ terminal states and $2^{2p-4} + 2^{2p-2} + 2^{p-1}$ non-terminal states.

The most straightforward way (that yields the fastest coding time, but requires the most memory) to implement a quasi-coder is to consider only the terminal states. Figure 3 shows a pseudo-code for quasi-coding, where q represents a terminal state and i, a probability state. For each triplet (q, i, x), where x is the binary symbol to be coded, the actual bits to output (`bits`), the number of bits to output (b) and the number of incurred follow bits (f) can be obtained via simple table lookups. Then the actual code bits can be output based on the `bits`, b and f values. Note that in this very simple implementation, there is no longer a clear distinction between interval subdivision and renormalization.

```
bits=tblBits[q][i][x]
b=tblNumBits[q][i][x]
f=tblNumFBits[q][i][x]
output(bits,b,f)
q=tblNextq[q][i][x]
i=tblNexti[i][x]
```

Fig. 3. Straightforward implementation of quasi-coding.

Assuming there are N_t terminal states and N_p probability states, all the tables require $2N_tN_p$ entries, except for the `tblNexti` table that requires $2N_p$ entries. If $N_t = 3(2^{2p-4})$, then for $p = 10$, which is the precision used by the M coder in the CABAC of H.264, $N_t = 3(2^{16})$. Then, with $N_p = 2^6$, we need a total of more than $3(2^{25})$ table entries. Definitely not practical.

An alternative implemenation that requires less memory is introduced in [29]. Contrary to the previous implementation, this implementation deals with both terminal and non-terminal states, where the encoding/decoding process consists of selecting a new state, and if the new state is non-terminal, a second transition to a terminal state is performed, with possible output/input of some code bits. The process of selecting a new state corresponds to the interval subdivision process, and the second transition corresponds to the renormalization process. The interval subdivision process consists of a lookup into an R table, and the concept is the same as the simplified subdivision process used by the Q-Coder and the M coder. The key difference between this implementation and those of the FBACs described in the previous subsection is in the renormalization part. Even in this implemenation, however, the renormalization method that is compatible with the M coder renormalization, would still require about 2^{21} table entries. In Section IV, we introduce a comparably fast renormalization method, semi-quasi-renormalization, that requires only about 2^p table entries for any precision p.

III. BAC Parameters

The following subsection introduces the set of parameters that can be used to describe any integer BAC (IBAC), similar to the one shown in Figure 2. Subsequently we define additional parameters needed for describing FBACs that replace the time consuming multiplicative operations in Line 1 of Figure 2 with a simple table lookup.

A. IBAC

1) Precision. This parameter is used to indicate the number of bits p used to represent L and R, or the current interval. Using integer arithmetic with p-bit precision is the same as mapping the $[0, 1)$ interval into $[0, 2^p)$.

2) Operation ordering convention (OOC). Arithmetic coding is based on interval subdivision, and for binary arithmetic coding, finding one of the two subintervals requires at least one multiplication and one division (e.g., Line 1 of Figure 2). Due to the use of integer arithmetic, the order of multiplicative operations is important (see [24] for more information). This parameter is used to set the operation order.

3) Symbol ordering convention (SOC). For proper decoding of coded symbols, both the encoder and decoder must agree on the SOC. For example, we can always assign the upper subinterval to the symbol 0 and the lower subinterval to 1 (0 over 1 SOC) as shown in Figure 2. Alternatively, it has been shown to be optimal, for software binary arithmetic coding, to have the upper subinterval assigned to the LPS and the lower one to the MPS [15]. This results in one less operation when coding the MPS than coding the LPS. On the other hand, when parallel processing is possible, the MPS over the LPS SOC is more efficient. This parameter allows to set the SOC.

```
const int rTbl[64][4]={ {128,176,208,240}, {128,167,197,227}, ... };
const int iTbl[64][2]={ {0,1}, {0,2}, {1,3}, ... };
bac MCoder {
  "prec",  10,              // 10-bit precision
  "ooc",   2,               // No multiplicative operation --> fast BAC
  "init",  {(1<<9)-2,9},    // Init: R=(1<<9)-2; D consists of first 9 code stream bits
  "end",   3,               // Output 3 bits to disambiguate the last symbol
  "rtable",{0,64,4,"rTbl"},// The rTbl table contains possible R values for the LPS
  "nexti", "iTbl",          // The iTbl table contains probability transition rules
}
```

Fig. 4. Flavor description of the M coder used in CABAC of H.264.

4) Truncation excess (TE). In Figure 2, due to the integer division applied, R_0 gets truncated and the excess range is assigned to R_1. The best way to distribute the TE due to integer arithmetic is to assign the TE to the two subintervals according to the corresponding symbol probabilities. However, this can be too complex (none of the currently availabe BACs supports this), and the next best choice is to assign the TE to the MPS. This parameter allows to specify the assignment of the TE.

5) When to renormalize. Using a fixed-precision arithmetic coding requires a renormalization process, which prevents R from getting too small (so that R can be represented by a fixed-bit integer and that *underflow* [10] can be prevented). At the same time, renormalization fixes the retention problem, allowing sequential coding, by outputting (in the case of encoding) the known bits of the code string. Renormalization can be applied as soon as the leading bit of the code string is known (e.g., the CACM renormalization [10]), or when R falls below a certain value. Each renormalization corresponds to $b > 0$ doublings of L and R, and output of b code bits. It is also possible to speed up the coding process by outputting a byte at a time [35], or more generally, to output $n_b \geq 1$ bits at a time [36]; when renormalizing, it is necessary that $R \leq 2^{p-n_b}$. This parameter allows to specify when to renormalize as well as the minimum number of bits n_b to output at a time.

6) Carry-over (CO). In general, the CO problem is resolved by one of the following two procedures: follow-on [10] and bit-stuffing [15]. By default, the follow-on procedure is used; however, when bit-stuffing is used, this paramter allows to specify when and what to stuff. For example, in the Q-Coder a 0 bit is stuffed whenever a `0xFF` byte is encountered [15].

7) Initialization. Though it makes the most sense to initially set R to the biggest value possible, there are cases where R is set to a smaller value due to compatibility reasons. Also for decoding, an extra variable V is maintained, which always contains the first p un-decoded bits of the code string; however, in certain cases, it may initially contain less than p bits (e.g., MPEG-4 shape coding). Alternatively, instead of maintaining $\{L, R \text{ and } V\}$ in the decoder, just $\{R \text{ and } D\}$ can be maintained where $D = V - L$. Using D instead of V and L makes the decoder simpler [24]; however, less option is given in terms of disambiguating the last symbol (see the next paragraph).

8) Disambiguating the last symbol. When terminating, some information that is still left in the encoder must be output to enable the decoder to unambiguously decode the last symbol. We call this disambiguating the last symbol and this is different from the actual termination problem, which is notifying the decoder when to stop. In the latter problem, we can use a special symbol, an end-of-file (EOF) symbol, or the total number of the symbols to be decoded can be made known to the decoder. If after encoding the last symbol, $R \geq 2^{p-n}$, where $n < p$, then it can be shown that $n+1$ bits are always enough to uniquely identify the last interval. Usually an encoded file (e.g., an H.264 video data) contains multiple components, where each component is separately coded. Thus, in the case where $n+1$ bits are used to disambiguate the last symbol of a component, the decoder must recant $p-(n+1)$ bits, which are needed to decode the symbols of the subsequent component. A simpler and less effective (compression-wise) method is to always output p bits to disambiguate the last symbol so that the decoder does not have to recant any bits. It is also possible to calculate the exact bits needed, and in this case the decoder has to maintain the L variable; hence, the decoding method using only the D variable cannot be used.

B. FBAC

9) Specifying the lookup tables. An R table can be specified to contain the values for one of the two possible subintervals. The probability transition rules can also be specified for each symbol. For example, the M coder specifies two tables for the the transition rules pertaining to the MPS and the LPS.

10) When to apply a probability state transition. In addition to specifying the above-mentioned tables, we also have to specify when to apply the probability state transition, i.e., when to adapt the probabilities of the symbol events. For example, in the Q-Coder, a transition is applied whenever renormalization takes place, and in the M coder, a transition is applied whenever a symbol is coded.

11) Whether to apply a conditional exchange. As we are using only given, approximate values for the subinterval lengths, it is possible to have $R_{LPS} > R_{MPS}$. In such condition, to improve compression, the two subintervals can be exchanged. For example, the QM-Coder [32] and the MQ-Coder [33] apply the conditional exchange.

Figure 4 shows an example of a Flavor description for the M coder used in the CABAC. Corresponding C++ or Java implementation of the coder can be automatically generated by feeding the descripiton into the Flavor translator.

Note that the description does not include all the parameters described in this section because each parameter has a default value. For more information on Flavor, refer to `http://flavor.sourceforge.net`.

IV. Semi-Quasi-Renormalization

Except the Q-Coder and the MQ-Coder, all of the currently known BACs use a renormalization method with the follow-on procedure. As shown in Figure 2, we assume that the follow bits get incurred only if $2^{p-2} \le L < 2^{p-1}$ and $2^{p-1} < H \le 3(2^{p-2})$. This guarantees that for each renormalization, if $f \le b$ follow bits are incurred, then they are incurred due to the last f doublings of L and R. The M coder of H.264, however, incurs a follow bit whenever $2^{p-2} \le L < 2^{p-1}$ and $R < 2^{p-2}$, and in this case there is no way of knowing which f of the b doublings caused the accumulation of the follow bits; hence quasi-coding is impossible. Note that the difference in the follow bits accumulation does not affect their compatibility. On the other hand, the former accumulation approach has the benefit of retaining less bits, as there are less cases for the follow bits to get incurred.

For every renormalization, the `bits`, b and f values (described in Subsection II-C) should be obtained for correct output of code bits. Additionally, after renormalization, the coder should be in a terminal state. The goal is to efficiently calculate the values for `bits`, b, f and the next terminal state – L_t and R_t. We first introduce several theorems that we use to come up with our semi-quasi-renormalization method, and then we describe the actual algorithms used to obain the above-mentioned values. As described in Section III, renormalization can be applied as soon as the leading bit of the code string is known, or when R falls below a certain value. Our method works almost the same for both renormalizations, with a small difference in finding the b value.

Any BAC can be described using the `bac` construct with the parameters described in the previous section, and Flavor can automatically generate C++/Java code for the described BAC. The generated code can be automatically optimized with our proposed semi-quasi-renormalization method, and in the last subsection we describe how the lookup tables needed by the method can be automatically generated.

A. Useful Theorems

The following theorems hold for the CACM renormalization method shown in Figure 2, as well as for the method that applies renormalization whenever R is below a certain value.

Theorem 1: Given a non-terminal interval $[L, H)$ that yields b output bits and f follow bits in precision p, the same interval in precision $p+1$ yields $b+1$ output bits and f follow bits.

Proof: A non-terminal interval $[L, H)$ in $[0, 2^p)$ is an interval that lies in the lower half of the full interval $[0, 2^{p+1})$ for $(p+1)$-bit precision. The interval just needs to be doubled without incurring any follow bit, leading to the interval $[2L, 2H)$. But an interval $[L, H)$ in p-bit precision corresponds to the interval $[2L, 2H)$ in $(p+1)$-bit precision. Hence, if the interval $[L, H)$ yields b output bits and f follow bits in p-bit precision, then it yields $b+1$ output bits and f follow bits in $(p+1)$-bit precision. □

Theorem 2: In p-bit representation, an interval $[L, H)$ in $[0, 2^{p-1})$ and a corresponding interval $[L + 2^{p-1}, H + 2^{p-1})$ in $[2^{p-1}, 2^p)$ yield the same number of output bits b and the same number of follow bits f.

Proof: If an interval in $[2^{p-1}, 2^p)$ is non-terminal, since $L \ge 2^{p-1}$, it needs to be doubled after being translated by -2^{p-1}. Hence both the intervals, $[L, H)$ and $[L + 2^{p-1}, H + 2^{p-1})$, if they are non-terminal, become $[2L, 2H)$ after one doubling. If they are terminal, then $b = f = 0$. □

Theorem 3: Let $b(I_A)$ denote the number of output bits yielded by an interval I_A in A, and let $f(I_A)$ denote the number of follow bits incurred from the same inteval. Then for the regions A and B shown in Figure 5 and for the corresponding intervals I_A and I_B, if they are non-terminal, then $b(I_A) = b(I_B)$ and $f(I_A) = f(I_B)+1$. Note that if I_B is $[b_1, b_2)$, then the corresponding interval I_A is $[b_1 + 2^{p-2}, b_2 + 2^{p-2})$.

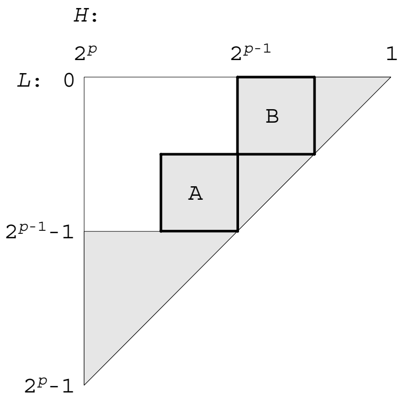

Fig. 5. The number of output bits yielded by a non-terminal state in region A is the same as that yielded by a corresponding state in B, i.e., $b(I_A) = b(I_B)$. For the incurred follow bits, $f(I_A) = f(I_B) + 1$.

Proof: If an interval $[L, H)$ in B is non-terminal, then it gets doubled to $[2L, 2H)$, without incurring any follow bit. The corresponding interval in A is $[L + 2^{p-2}, H + 2^{p-2})$. It is also non-terminal and it needs to be doubled after being translated by -2^{p-2}, incurring a follow bit. Hence, both the intervals in A and B converge to the interval $[2L, 2H)$ after one doubling, but the one in A incurs one more follow bit. □

For the CACM renormalization, the b values for the valid intervals in $p = 3$ and 4-bit precisions are shown in Figure 6 and the corresponding f values are shown in Figure 7. It can be seen from the figures that the above theorems hold. The non-shaded region represents terminal states, and hence no renormalization takes place, and $b = f = 0$.

B. Number of output bits b

Given the theorems in the previous subsection and the b values for $p = 3$ and 4 in Figure 6, the number of output bits for non-terminal states follows the pattern shown in Table I, in the case of the CACM renormalization. The table shows that the b value depends on L, R, and p. For example, if $R = 5$,

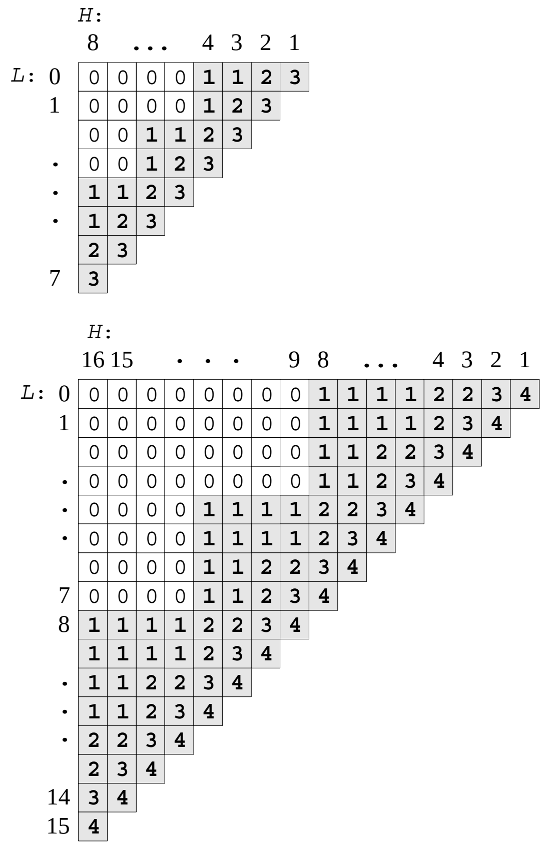

Fig. 6. The sample b values for the valid intervals in 3 and 4-bit precisions when the CACM renormalization method is applied. The shaded region corresponds to non-terminal states.

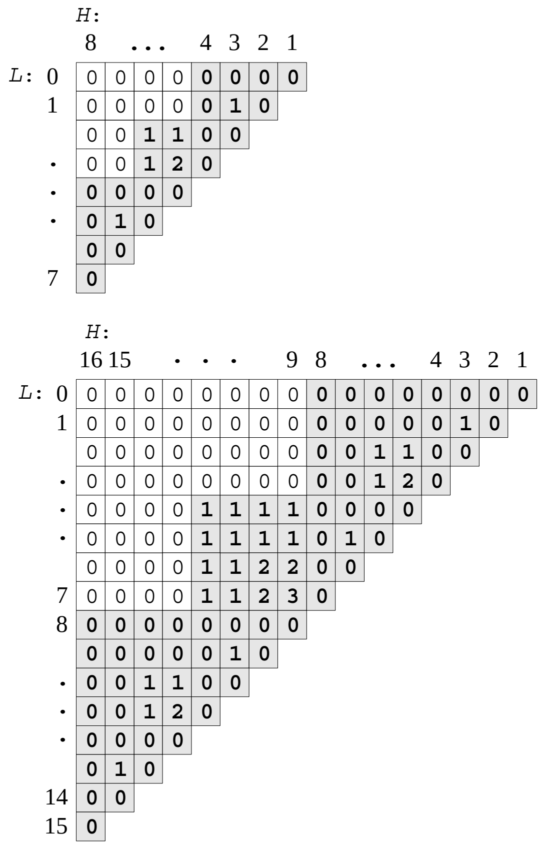

Fig. 7. The sample f values for the valid intervals in 3 and 4-bit precisions when the CACM renormalization method is applied. The shaded region corresponds to non-terminal states.

then $b = \overline{b} = p - 3$. In the case $R = 6$, if the two least significant bits (LSBs) of L are set, then $b = \overline{b} - 1 = p - 4$; otherwise $b = \overline{b} = p-3$. Let $\text{lsb}_n(L)$ be a function that returns the n LSBs of L for $n > 0$, and returns 0 for $n \leq 0$, Then, the general equation for b is as follows:

$$b = \overline{b} - [((\text{lsb}_n(L) + \text{lsb}_n(R-1)) >> n)\&1],$$

$$\text{where } n = (p - \overline{b} - 1)$$

The $\overline{b}$ value can be easily obtained via a lookup table with 2^{p-1} entries, the $\text{lsb}_n()$ function can be easily implemented via simple bit-wise operations, and consequently the b value can be very efficiently obtained.

In the case where renormalization is applied whenever R falls below a certain value, b simply equals to $\overline{b}$; Table II lists the $\overline{b}$ values for the case where renormalization is applied whenever $R < 2^{p-2}$.

TABLE I

IN THE CASE OF THE CACM RENORMALIZATION, THE b VALUE IS A FUNCTION OF L, R AND p.

R	$\overline{b}$	b
1	p	$\overline{b}$
2	p-1	$\overline{b}$
3	p-2	$\overline{b}$
4		$\overline{b}$-1, if 0bL_0 = 0b1; else $\overline{b}$
5	p-3	$\overline{b}$
6		$\overline{b}$-1, if 0bL_1L_0 = 0b11; else $\overline{b}$
7		$\overline{b}$-1, if 0bL_1L_0 = 0b11,0b10; else $\overline{b}$
8		$\overline{b}$-1, if 0bL_1L_0 = 0b11,0b10,0b01; else $\overline{b}$
9	p-4	$\overline{b}$
10		$\overline{b}$-1, if 0b$L_2L_1L_0$ = 0b111; else $\overline{b}$
11		$\overline{b}$-1, if 0b$L_2L_1L_0$ = 0b111,0b110; else $\overline{b}$
⋮	⋮	⋮
2^n+1	p-n-1	$\overline{b}$
⋮	⋮	⋮

C. Number of follow bits f

Given the theorems in Subsection IV-A and the f values for $p = 3$ and 4 in Figure 7, for non-terminal states, f can be simply obtained as follows. For $R = 2$ (i.e., $\overline{b} = p - 1$ from Table I), $f > 0$ if H is in the form $2n{+}1$, where n is a positive integer. Then $f = \text{ls}_1(n)$, where the $\text{ls}_1()$ function returns the bit position of the least significant 1-bit, with the LSB position corresponding to 1, the next LSB position corresponding to 2, and so on. For example, $\text{ls}_1(6) = 2$ and $\text{ls}_1(15) = 1$. In the case $n = 0$, $\text{ls}_1(n)$ simply returns 0. For $R = 3$ (i.e., $\overline{b} = p - 2$), the states with $H = \{4n+1, 4n+2\}$ yield $f > 0$ follow bits, and again, the actual f value can be obtained using the $\text{ls}_1()$ function with n as its argument. For $R = 4$ (again, $\overline{b} = p - 2$), the states with $H = \{4n+1, 4n+2, 4n+3\}$

TABLE II
IF RENORMALIZATION IS APPLIED WHENEVER $R < 2^{p-2}$, THEN THE b VALUE IS A FUNCTION OF R AND p.

R	1	[2,3]	[4,7]	[8,15]	...
$\overline{b}$	p-2	p-3	p-4	p-5	...

yield $f > 0$ follow bits. Note that in the case $R = 4$, the states with $H = \{4n+1, 4n+3\}$ yield $f = \text{ls}_1(n) - 1$. This happens whenever $b = \overline{b} - 1$. Generally, given R for a non-terminal state, we can obtain the corresponding $\overline{b}$ value, as discussed in the previous subsection, and the states with $H = \{2^{p-\overline{b}}n + 1, 2^{p-\overline{b}}n + 2, \ldots, 2^{p-\overline{b}}n + (R-1)\}$ yield $f > 0$ follow bits. The actual f value can be obtained as follows:

$$f = \text{ls}_1(n) - \overline{b} + b$$

The $\text{ls}_1()$ function can be implemented using a lookup table with 2^{p-1} entries. Note that in the case where renormalization is applied whenever R falls below a certain value, $b = \overline{b}$ and $f = \text{ls}_1(n)$.

D. The next terminal state $[L_t, H_t) = [L_t, L_t + R_t)$

Given b from Subsection IV-B, we know that the next terminal interval length $R_t = R << b$. Now, for the next terminal lower bound L_t, consider the following.

Before doubling of L, if its most significant bit $L_{p-1} = 1$, it is set to 0. Hence after one doubling, L becomes $2L_{(p-2)..0}$, where $L_{(p-2)..0}$ represents the $p-1$ LSBs of L. If, however, a follow bit has incurred, then 2^{p-2} has been subtracted from L and it becomes $2L_{(p-3)..0}$. Note that if a renormalization yields $f > 0$ follow bits, then it must be the last $f \le b$ doublings that yield the follow bits. Hence, after b doublings, the terminal interval lower bound becomes:

$$L_t = \begin{cases} (L << b)\&(N-1) & \text{, if } f = 0 \\ (L << b)\&(N-1) - 2^{p-1} & \text{, otherwise} \end{cases}$$

The pseudo-code for the semi-quasi-renormalization method is shown in Figure 8, which is equivalant to the CACM renormalization method shown in Figure 2. Note that the `for` and `if-else` branching operations are omitted. The `output()` function in Line 8 requires one extra `if` statement than the `bit_plus_follow()` function [10], but overall, our renormalization method requires less coding time. In Section V we show that this is indeed the case.

E. Generating lookup tables

The table for $\overline{b}$ (`tblNumBits` in Line 1 of Figure 8) can be automatically generated for any given p value as follows. In the case where renormalization is applied as soon as the leading code bit is known, we need 2^{p-1} entries. Then, the table entry values can be otained according to Table I. In the other case where renormalization is applied whenever R is below is certain value, e.g., 2^n, we need 2^n entries for the table, where $n < p$, and its entry values are obtained similarly. In the latter case, the first value must be n, rather than p, and

```
1  b̄=tblNumBits[R], lsb=tblLSB[b], n=p-b̄
2  b=b̄-(((L&lsb)+((R-1)&lsb))>>(n-1))
// index into tblNumFBits
3  i_F=H>>n
4  i_1=((L-(i_F<<n))>>31)&((i_F<<n)-H)>>31)&1
// index into tblOne
5  f=tblOne[i_1]&(tblNumFBits[i_F]-b̄+b);
6  i_1=((L|~(H-1)|((L&~(H-1))<<1))>>(p-1))&1
7  b&=tblOne[i_1], f&=tblOne[i_1]
8  output(L,b,f)
9  L=(L<<b)&(N-1)-(tblOne[(-f)>>31)&1]&HALF)
10 R<<=b
```

Fig. 8. The semi-quasi-renormalization method.

the next 2^1 values are $n-1$, the subsequent 2^2 values are $n-2$, and so on.

The table for the $\text{ls}_1()$ function (`tblNumFBits` in Line 5 of Figure 8) can also be automatcally generated, where for the k-th entry in the table, the value is simply $\text{ls}_1(k)$. In the case where renormalization is applied as soon as the leading bit is known, we need 2^{p-1} entries, and in the other case where renormalization is applied whenever $R < 2^n$, we need $2^n - 1$ entries.

The `tblLSB` table in Line 2 of Figure 8 includes the values used by the $\text{lsb}_n()$ function described in Subsection IV-B, and it only consists of $p+1$ entries. For example, for $p = 10$, the values stored in the table are: {`0x00, 0xFF, 0x7F, 0x3F, 0x1F, 0x0F, 0x07, 0x03, 0x01, 0x00, 0x00`}. The `tblOne` table contains only two values `0x00000000` and `0xFFFFFFFF`.

We show in the next section that the Flavor-generated BACs with our proposed renormalization method outperforms corresponding BACs that use renormalization with branching operations.

V. EXPERIMENTAL RESULTS

The experiments described in this section are all performed on a Pentium M, 1.6 GHz machine with 512 MB RAM. The first experiment shows that our semi-quasi-renormalization method outperforms the currently used renormalization methods that use branching operations. We have two methods: Renorm 1 renormalizes the current interval whenever the leading bit of the current interval is known, and Renorm 2 performs renormalization whenever $R < 2^{p-2}$. Note that Renorm 1 is the method shown in Figure 2, and Renorm 2 is the one used by the M coder in H.264. To compare the speed of the two methods with our corresponding semi-quasi-renormalization methods (e.g., the corresponding semi-quasi-renormalization method for Renorm 1 is shown in Figure 8), we timed 10 million renormalization processes for 8 different cases. The results are shown in Table III, where the second and third columns correspond to the processing times of Renorm 1 and Renorm 2, and the last two columns, to those of their corresponding semi-quasi-renormalization methods. First, we consider the case where there is no need for a doubling of L and R, i.e., $b = 0$. This is equivalent to applying no renormalization, however, there is the need for a conditional branch operation that checks to see if a renormalization is required. In the second case, each renormalization process consists of 1 doubling, i.e., $b = 1$; in the third case, there is only 2 doublings, i.e., $b = 2$; and so on. Note that the processing time for the semi-quasi-renormalization methods is constant, whereas both Renorm 1 and Renorm 2's processing time increases linearly with b. Even when $b = 1$, we see that the semi-quasi-renormalization methods are more than 5 times faster than the ones that use branching operations.

TABLE III
THE TIME (IN MS) OF RUNNING VARIOUS RENORMALIZATION CASES.

			semi-quasi	
b	Renorm 1	Renorm 2	Renorm 1	Renorm 2
0	140 ms	30 ms	40 ms	30 ms
1	210	160	40	30
2	290	300	40	30
3	350	340	40	30
4	390	400	40	30
5	460	450	40	30
6	490	500	40	30
7	540	540	40	30

In many practical cases, most of the renormalizations will consist of only 1 doubling, however, it is possible to have renormalizations consisting of $b > 1$ doublings. The second experiment compares the actual coding time of the 8 test images generated by the CCITT for developing and testing the Group 3 fax standard. For this experiment, we used the fast interval subdivision process used by the M coder, with the four renormalization methods listed in Table III; the results are shown in Table IV. MC 1 in the second column corresponds to the M coder that uses Renorm 1, MC 2 uses Renorm 2, and so on. Note that MC 2 is the actual M coder specified in the H.264 standard, and the semi-quasi-MC 2 (in the last column of Table IV) is 100% compatible with MC 2. We see that even for practical cases, the coders with semi-quasi-renormalization outperforms the other ones. Additionally, the semi-quasi-M coders are automatically generated by the Flavor translator from the description shown in Figure 4.

TABLE IV
THE TIME (IN MS) OF CODING THE 8 CCITT TEST IMAGES.

			semi-quasi	
image	MC 1	MC 2	MC 1	MC 2
ccitt1.pbm	280 ms	270 ms	240 ms	190 ms
ccitt2.pbm	280	280	240	190
ccitt3.pbm	290	290	250	200
ccitt4.pbm	300	270	270	230
ccitt5.pbm	290	280	250	210
ccitt6.pbm	280	280	250	190
ccitt7.pbm	310	310	260	220
ccitt8.pbm	380	320	310	270

VI. CONCLUSION

We have introduced a set of simple parameters for describing any BAC. Then, using the Flavor translator, optimized C++ or Java code can be automatically generated for the described coder. If bit-stuffing is not used, the generated coder can use our novel, memory-efficient, semi-quasi-renormalization method that significantly outperforms the currently used renormalization methods. Similar to the quasi-coder introduced by Howard and Vitter, it replaces time-consuming operations (the branching operations) with table lookups. The BAC with our proposed renormalization method yields comparable coding time as the quasi-coder, but with much less memory.

REFERENCES

[1] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, vol. 40, pp. 1098–1101, 1952.

[2] *Standardization of Group 3 Facsimile Apparatus for Document Transmission*, CCITT (ITU Recommendation T.4), 1980, amended in 1984 and 1988.

[3] *Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus*, CCITT (ITU Recommendation T.11), 1984, amended in 1988.

[4] *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbits/s*, ISO/IEC 11172 International Standard (MPEG-1), 1993.

[5] *Information technology – Generic coding of moving pictures and associated audio information*, ISO/IEC 13818 International Standard (MPEG-2), 1996.

[6] R. G. Gallager, "Variations on a Theme by Huffman," *IEEE Trans. on Info. Theory*, vol. 24, pp. 668–674, 1978.

[7] R. M. Capocelli and A. de Santis, "Tight Upper Bounds on the Redundancy of Huffman Codes," *IEEE Trans. on Info. Theory*, vol. 35, pp. 1084–1091, 1989.

[8] ——, "New Bounds on the Redundancy of Huffman Codes," *IEEE Trans. on Info. Theory*, vol. 37, pp. 1095–1104, 1991.

[9] G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.*, vol. 28, pp. 135–149, 1984.

[10] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, pp. 520–540, 1987.

[11] N. Abramson, *Information Theory and Coding*. McGraw-Hill, 1963.

[12] F. Jelinek, *Probabilistic Information Theory*. McGraw-Hill, 1968.

[13] G. G. Langdon and J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. on Communications*, vol. 29, pp. 858–867, 1981.

[14] ——, "A Simple General Binary Source Code," *IEEE Trans. on Info. Theory*, vol. 28, pp. 800–803, 1982.

[15] W. B. Pennebaker and et al., "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Develop.*, vol. 32, pp. 717–726, 1988.

[16] S. W. Golomb, "Run-Length Encodings," *IEEE Trans. on Info. Theory*, vol. 12, pp. 399–401, 1966.

[17] *Information technology – Coded representation of picture and audio information – Progressive bi-level image compression*, ISO/IEC 11544 International Standard (JBIG), 1993.

[18] *Information technology – Lossy/lossless coding of bi-level images*, ISO/IEC 14492 International Standard (JBIG2), 2001.

[19] *Information technology – Coding of audio-visual objects – Part 2: Video*, ISO/IEC 14496-2 International Standard, 1999.

[20] *Information technology – JPEG 2000 image coding system*, ISO/IEC 15444 International Standard (JPEG 2000), 2000.

[21] *Information technology – Digital compression and coding of continuous-tone still images*, ISO/IEC 10918 International Standard (JPEG), 1994.

[22] *Advanced video coding for generic audiovisual services*, ITU-T Recommendation (H.264), 2003.

[23] J. Rissanen and G. G. Langdon, "Universal Modeling and Coding," *IEEE Trans. on Info. Theory*, vol. 27, pp. 12–23, 1981.

[24] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic Coding Revisited," *ACM Transactions on Information Systems*, vol. 16, pp. 256–294, 1998.

[25] A. Eleftheriadis, "Flavor: A Language for Media Representation," in *ACM Int. Conf. on Multimedia*, ser. Proceedings, 1997, pp. 1–9.

[26] A. Eleftheriadis and D. Hong, "Flavor: A Formal Language for Audio-Visual Object Representation," in *ACM Int. Conf. on Multimedia*, ser. Proceedings, 2004, pp. 816–819.

[27] D. Hong and A. Eleftheriadis, "Automatic Generation of C++/Java Code for Binary Arithmetic Coding," in *Picture Coding Symposium*, ser. Proceedings, 2004.

[28] P. G. Howard and J. S. Vitter, “Practical Implementations of Arithmetic Coding,” in *Image and Text Compression*, ser. Kluwer Academic, 1992, pp. 85–112.

[29] ——, “Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding,” in *IEEE Data Compression Conference*, ser. Proceedings, 1993, pp. 98–107.

[30] D. Marpe, H. Schwartz, G. Blattermann, G. Heising, and T. Wiegand, “Context-Based Adaptive Binary Arithmetic Coding in JVT/H.26L,” in *IEEE Int. Conf. on Image Processing*, ser. Proceedings, 2002, pp. 513–516.

[31] D. Marpe, H. Schwartz, and T. Wiegand, “Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard,” *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, pp. 620–636, 2003.

[32] W. B. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*. Chapman and Hall, 1992.

[33] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic, 2001.

[34] D. Marpe and T. Wiegand, “A Highly Efficient Multiplication-Free Binary Arithmetic Coder and Its Application in Video Coding,” in *IEEE Int. Conf. on Image Processing*, ser. Proceedings, 2003, pp. 263–266.

[35] M. Schindler, “A Fast Renormalisation for Arithmetic Coding,” in *IEEE Data Compression Conference*, ser. Proceedings, 1998, p. 572.

[36] L. Stuiver and A. Moffat, “Piecewise Integer Mapping for Arithmetic Coding,” in *IEEE Data Compression Conference*, ser. Proceedings, 1998, pp. 3–12.