Context-adaptive Binary Arithmetic Coding with Fixed-length Codewords

Francesc Aulí-Llinàs, *Senior Member, IEEE*

***Abstract*—Context-adaptive binary arithmetic coding is a widespread technique in the field of image and video coding. Most state-of-the-art arithmetic coders produce a (long) codeword of a priori unknown length. Its generation requires a renormalization procedure to permit progressive processing. This paper introduces two arithmetic coders that produce multiple codewords of fixed length. Contrarily to the traditional approach, the generation of fixed-length codewords does not require renormalization since the whole interval arithmetic is stored in the coder's internal registers. The proposed coders employ a new context-adaptive mechanism based on variable-size sliding window that estimates with high precision the probability of the symbols coded. Their integration in coding systems is straightforward as demonstrated within the framework of JPEG2000. Experimental tests indicate that the proposed coders are computationally simpler than the MQ coder of JPEG2000 and the M coder of HEVC while achieving superior coding efficiency.**

***Index Terms*—Context-adaptive binary arithmetic coding, fixed-length arithmetic codes.**

I. Introduction

ARITHMETIC coding is among the most popular entropy coding techniques employed nowadays. The codeword generated by the arithmetic coder is a number within an interval arithmetic that represents the coded symbols. Briefly described, the coder begins by segmenting the interval of real numbers $[0, 1)$ in as many subintervals as there are symbols in the alphabet. The size of the subintervals is commonly selected according to the probabilities of the symbols, more precisely, as $[0, F(x = \alpha))$, $[F(x = \alpha), F(x = \beta))$, $\ldots$ with $\{\alpha, \beta, ...\}$ representing the alphabet of symbols and $F(x)$ being the cumulative mass function of x. The first symbol of the message is coded by selecting its corresponding subinterval. Then, this procedure is repeated within the selected subintervals for the following symbols. The transmission of any number within the range of the final subinterval (i.e., the codeword), guarantees that the reverse procedure decodes the original message losslessly.

The computational complexity of context-adaptive binary arithmetic coders has always been a concern since they are intensively used in image and video codecs. The first ideas to reduce their complexity aimed at multiplication-free implementations that perform the interval division using bit shifts and adds [1]. Subsequently, the Q coder [2] approached the interval division by means of lookup tables (LUTs). Some of the descendants of the Q coder were introduced in the JPEG, JBIG2, and JPEG2000 standards. Standards of video coding such as H.264/AVC and HEVC employ variants of the M coder [3], which was introduced in the 2000s employing a reduced range of possible subinterval sizes together with LUTs. Among others, enhancements to the M coder have been proposed in [4].

Most arithmetic coders employed for image and video coding produce variable-to-variable length codewords. This is, a variable number of input symbols are coded with a codeword of a priori unknown length. Practical realizations of arithmetic coders operate with hardware registers of at most 64 bits, so the generation of a single –and commonly very long– codeword is carried out progressively. The main idea to do so is the following. Let $[L, U)$ denote the current interval of the coder, with L and U being the fractional part of the lower and upper bound of the interval stored in hardware registers. Assume that the leftmost bits of the binary representation of L and U are not equal in the current interval. When a new symbol is coded, this interval is further reduced to $[L', U')$. If the leftmost bits of L' and U' are then equal, all following segmentations of the interval will also start with those same bit(s) since $L \leq L' \leq \ldots \leq U' \leq U$. This permits to dispatch the leftmost bits of L' and U' that are identical and to shift the remaining bits of the registers to the left. This procedure is called renormalization. It represents a non-negligible part of the coder's workload since these operations are executed intensively.

The operations carried out by the renormalization procedure can be avoided if, instead of producing a single codeword, the coder produces short codewords of fixed length. To this end, the coder uses an integer interval of range $[0, 2^{\mathcal{W}} - 1]$, with $\mathcal{W}$ denoting the length of the codewords. The coding of symbols is carried out by segmenting this interval as it is previously described. When the size of the last selected subinterval is 1, the number that it contains (which represents the fixed-length codeword) is dispatched and the interval is reset. We note that the codewords produced by such a method can not be regarded as portions of a single codeword generated by a variable-to-variable arithmetic coder since each fixed-length codeword holds the complete representation of some symbols of the message.

Arithmetic coding with fixed-length codewords was first proposed in the nineties [5], [6] with the aim to address some of the disadvantages of conventional arithmetic coding such as poor recovery from channel errors or lack of random access and partial decoding. Such technique has also been used in [7] to limit error propagation, and in [8] to compress machine instructions. In the field of image coding, only [9] utilizes arithmetic coding with codewords of fixed length for the compression of bilevel images.

Dr. Francesc Aulí-Llinàs is with the Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain (phone: +34 935811861; fax: +34 935813443; e-mail: fauli@deic.uab.cat). This work has been partially supported by the Spanish Government (MINECO), by FEDER, and by the Catalan Government, under Grants RYC-2010-05671, TIN2012-38102-C03-03, and 2014SGR-691.

This work introduces two arithmetic coders employing fixed-length codewords that are devised for image/video coding systems. The first coder aims at low computational complexity. It utilizes one integer interval that is reset when exhausted. The second coder aims at high coding efficiency. It utilizes two integer intervals in which the symbols are selectively coded depending on their probabilities. A novel variable-size sliding window mechanism that estimates the probabilities of the symbols is included in both coders. The main difference between the proposed method and previous fixed-length arithmetic coders [5]–[9] (not including the MQ or the M coder) is the use of a binary alphabet, adaptive mechanisms for probability estimation, low-complexity instructions for the interval division, and a selective interval coding technique to enhance efficiency. Experimental results indicate that the proposed coders achieve superior performance to that achieved by the MQ coder of JPEG2000 and by the M coder of HEVC –both in terms of coding efficiency and computational throughput. They are introduced in a JPEG2000 codec to illustrate their performance and ease of integration.

Section II of this paper describes the proposed coders. Their performance is evaluated in Section III through experimental results. The last section provides concluding remarks.

II. Proposed Coders

A. Low-complexity coding

The proposed coder codes binary symbols using codewords of $\mathcal{W}$ bits. With some abuse of notation, let the lower bound of the interval be denoted by L. The size of the interval *minus one* is denoted by S. Both L and S are stored in integer registers. Initially, $L = 0$ and $S = 2^{\mathcal{W}} - 1$. The operation to partition the interval uses integer arithmetic since the latency of integer multiplications in modern processors is (almost) one clock cycle [10]. When the coded bit is 0 (i.e., $x = 0$), the size of the interval is reduced to

$$S \leftarrow (S \cdot P) \gg \mathcal{B} \ , \tag{1}$$

and L is left unmodified. $\gg$ denotes a bit shift to the right. P denotes the probability of the symbol to be 0, expressed in the range $[0, 2^{\mathcal{B}} - 1]$. More precisely, $P = \lfloor f(x = 0) \cdot 2^{\mathcal{B}} \rfloor$, with $f(x)$ denoting the probability mass function of x and $\lfloor \cdot \rfloor$ denoting the floor operation. $\mathcal{B}$ is the number of bits employed to express the symbol's probability. The result of the multiplication in (1) must not cause arithmetic overflow, so $\mathcal{W} + \mathcal{B} \leq 64$. In our implementation $\mathcal{B} = 15$, whereas $\mathcal{W}$ ranges from 8 to 48 (see below).

When $x = 1$, the interval is reduced according to

$$\begin{aligned} S &\leftarrow S - ((S \cdot P) \gg \mathcal{B}) - 1 \ , \\ L &\leftarrow L + ((S \cdot P) \gg \mathcal{B}) + 1 \ . \end{aligned} \tag{2}$$

In this case, four more additions (or three in the algorithm below) than those necessary in (1) are required. The execution of (2) can be minimized by performing a conditional exchange between symbols 0 and 1 when $f(x = 1) > f(x = 0)$ so that the most probable symbol is always coded as 0. The interval is exhausted when $S = 0$. Then, L (which represents the codeword) is dispatched and the registers are reset to $L \leftarrow 0$ and $S \leftarrow 2^{\mathcal{W}} - 1$.

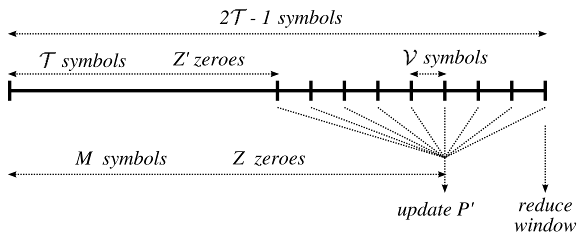


Fig. 1: Illustration of the variable-size sliding window mechanism employed to estimate the probabilities of the symbols.

In general, $f(x)$ is not known during coding, so it is estimated considering the distribution of the last symbols coded. The estimation of $f(x)$ is commonly carried out by context-adaptive mechanisms and probability models. Our coder uses a variable-size sliding window [4] that utilizes between $\mathcal{T}$ and $2\mathcal{T} - 1$ symbols except at the beginning of coding. This technique has been devised to minimize computational costs without affecting coding efficiency. More precisely, it has low memory requirements since it does not hold the symbols coded, reduces the times that some variables are updated, and computes the probability estimate only once every $\mathcal{V}$ symbols coded. Fig. 1 illustrates the variable-size sliding window employed. The thick horizontal line represents the symbols coded. The probability estimate, denoted by P', is updated every $\mathcal{V}$ symbols according to

$$P' \leftarrow \min\left(\frac{Z \ll \mathcal{B}}{M}, \ 2^{\mathcal{B}} - 1\right) \ , \tag{3}$$

with M being the number of symbols within the variable-size window and Z being the number of zeroes coded within the window, this is, during the last M symbols. $\ll$ denotes a bit shift to the left. The $\min(\cdot)$ operation is employed to make sure that $P' \in [0, 2^{\mathcal{B}} - 1]$ even when $Z = M$. The division in (3) is carried out in the integer domain. As depicted in Fig. 1, when $M = 2\mathcal{T} - 1$ the window is reduced to $\mathcal{T}$ symbols and the number of zeroes within the window is updated according to

$$\begin{aligned} M &\leftarrow \mathcal{T} \ , \\ Z &\leftarrow Z - Z' \ , \\ Z' &\leftarrow Z \ , \end{aligned} \tag{4}$$

with Z' being the number of zeroes coded during the first $\mathcal{T}$ symbols of the window.

To simplify the conditional that checks if the window size and the probability estimate have to be updated, $2\mathcal{T} - 1$ and $\mathcal{V}$ are forced to be of the form $2\mathcal{T} - 1 = 2^{\mathcal{E}} - 1$ and $\mathcal{V} = 2^{\mathcal{E}'} - 1$, so that a bit-wise AND operation between two integer registers can be used instead of a modulo operation (i.e., $M \ \& \ \mathcal{V} = \mathcal{V}$ is used instead of $M \ \% \ \mathcal{V} = 0$, with & and % denoting the AND and the modulo operation, respectively). Obviously, $\mathcal{E} \geq \mathcal{E}'$. The result of the bit shift in (3) must not cause arithmetic

Algorithm 1 FLW encode (x bit to encode, c context)
Initialization: $L \leftarrow 0, S \leftarrow 2^{\mathcal{W}} - 1, Z'[c] \leftarrow -1 \ \forall \ c$

```
1: if $M[c]$ & $\mathcal{V} = \mathcal{V}$ then
2:     $P'[c] \leftarrow \min\big((Z[c] \ll \mathcal{B})/M[c],\ 2^{\mathcal{B}} - 1\big)$
3:     if $M[c]$ & $(\mathcal{T} - 1) = (\mathcal{T} - 1)$ then
4:         if $Z'[c] \geq 0$ then
5:             $M[c] \leftarrow \mathcal{T}$
6:             $Z[c] \leftarrow Z[c] - Z'[c]$
7:         end if
8:         $Z'[c] \leftarrow Z[c]$
9:     end if
10: end if
11: if $x = 0$ then
12:     $S \leftarrow (S \cdot P'[c]) \gg \mathcal{B}$
13:     $Z[c] \leftarrow Z[c] + 1$
14: else
15:     $k \leftarrow ((S \cdot P'[c]) \gg \mathcal{B}) + 1$
16:     $L \leftarrow L + k$
17:     $S \leftarrow S - k$
18: end if
19: $M[c] \leftarrow M[c] + 1$
20: if $S = 0$ then
21:     dispatchCodeword($L$)
22:     $L \leftarrow 0$
23:     $S \leftarrow 2^{\mathcal{W}} - 1$
24: end if
```

overflow, so $\mathcal{E} + \mathcal{B} \leq 64$. We note that other mechanisms of probability estimation such as [4], that avoids the use of an integer division, might also be employed. Empirical evidence indicates that (3) increases only slightly the complexity of the coder.

Algorithm 1 details the encoding procedure of the proposed arithmetic coder with fixed-length codewords (FLW). The notation is that employed in the previous discussion except for the variables of probability estimation, which are arrays accessed via the context c for which they are computed [11]. The probability estimation is carried out in lines 1-10. Z' is initialized to -1 so that the size of the window is extended to $2\mathcal{T} - 1$ at the beginning of the coding. We note that after this point, the actual number of symbols within the window is $M - 1$, though it is not considered in line 2 since it does not affect coding efficiency. The interval division is performed in lines 11-18, whereas the dispatching of the codeword is carried out in lines 20-24. The decoder has a structure similar to that of the encoder (not shown due to page constraints). As well as most context adaptive coders, the bitstream generated by this (and the following) algorithm does not have error recovery properties. If needed, they could be included by using segment markers.

B. Improving coding efficiency

The main drawback behind the use of codewords of fixed-length is that the coding efficiency is penalized when the size of the interval is small. Let us illustrate this point with an example. Assume that the size of the current interval is 2 and that the next symbol to code has a high probability estimate, say 90%. Although the coding of the most probable symbol should spend a fraction of a bit, its actual coding spends a full bit because the interval can only be divided in two subintervals of equal size. So, in practice, it is like if the coding of this symbol had employed a probability estimate of 50%.

The proposed arithmetic coder with two fixed-length codewords (FL2W) addresses this drawback by using two intervals. They are stored in the integer registers $L[0], S[0]$ and $L[1], S[1]$. All symbols are coded employing the first interval while its size is greater than a predefined threshold, i.e., while $S[0] > \mathcal{Q}$. When $S[0] \leq \mathcal{Q}$, then the symbol's probability estimate $P'[c]$ is tested to check whether it fits well in the first interval or not. The closest probability to $P'[c]$ that can be employed in this interval is

$$P'' \leftarrow \frac{(((S[0] \cdot P'[c]) \gg \mathcal{B}) + 1) \ll \mathcal{B}}{S[0] + 1} \ , \tag{5}$$

with the division carried out in the integer domain. P'' is expressed in the same range employed for $P'[c]$, i.e., $P'' \in [0, 2^{\mathcal{B}} - 1]$. The absolute difference between P'' and the probability estimate (i.e., $|P'[c] - P''|$) determines whether the symbol is coded in the first interval or not. If the difference is smaller than a predefined threshold, say $\mathcal{R}$, then the symbol is coded in the first interval. Otherwise it is coded in the second. When the first interval is exhausted, its codeword is dispatched and replaced by that of the second interval, which is reset. This selective interval coding increases the efficiency of the coder since it avoids coding symbols in intervals in which the probability estimates do not fit well. We found that $\mathcal{Q} = 16$ and $\mathcal{R} = \lfloor 0.05 \cdot 2^{\mathcal{B}} \rfloor$ are good choices for a large variety of sources.

Evidently, the procedure described before can only be performed if the size of the second interval is $S[1] > \mathcal{Q}$. If not, both intervals may not have an appropriate size to code the symbol without loss in coding efficiency. When both $S[0]$ and $S[1]$ are $\leq \mathcal{Q}$, the interval with a closest P'' to the probability estimate is chosen. The implementation of FL2W must also take into account that if the second interval is exhausted (i.e., $S[1] = 0$), then all symbols are coded in the first until it is exhausted too. This happens rarely in practice.

To use the immediately next codeword to alleviate the impact in coding efficiency is also employed in [6], [7], though these methods only code the last symbol employing bits of two consecutive codewords. Limited to pages, the algorithm of FL2W is not detailed herein. It can be found together with the implementation of all coders employed in this paper in [12].

III. Experimental Results

A. Simulations

The first set of experimental tests assess the coding efficiency and computational throughput when coding artificially generated symbols. The symbols are generated assuming that they are independent and identically distributed. A generalized Gaussian distribution (GGD) with parameter $\sigma = 0.2$ and support in the range $(0, 1)$ is employed to generate the probabilities of the symbols. Through this method, the probabilities of the symbols are from almost 0 to almost 1, though most symbols have a probability close to μ. The experiments below report the performance achieved with $\mu = 0.55$ and $\mu = 0.85$

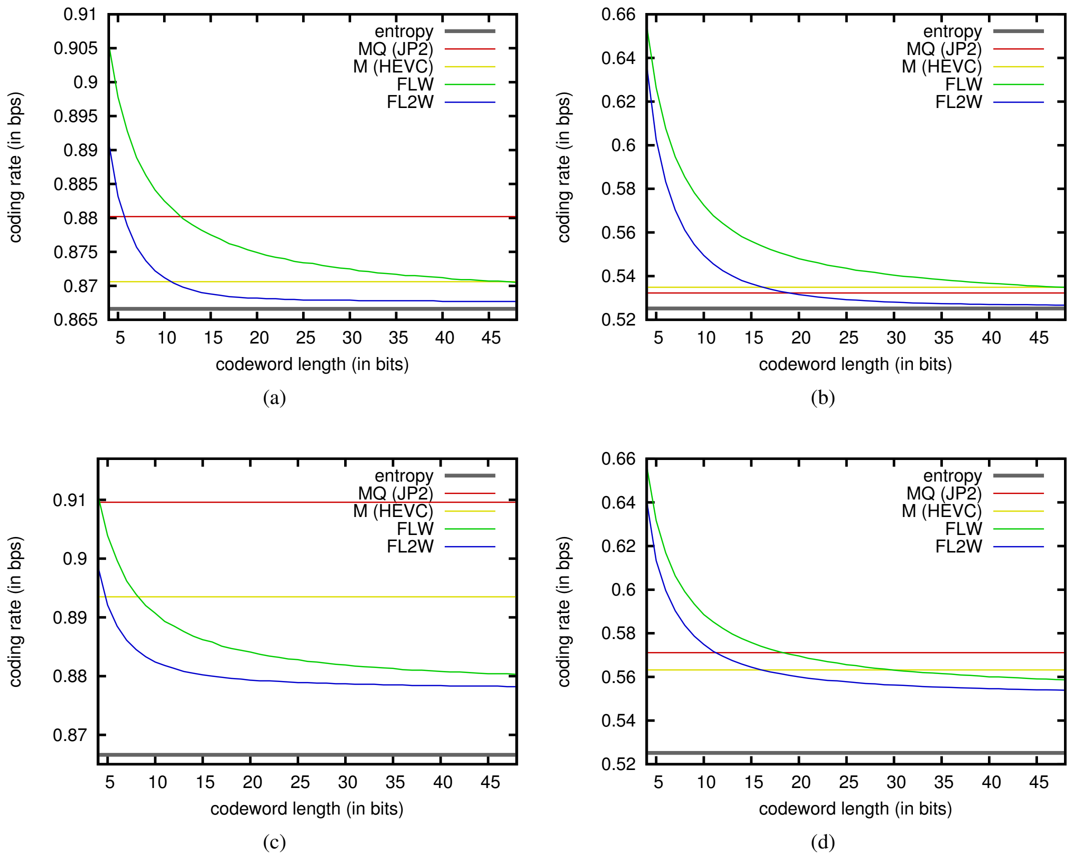


Fig. 2: Evaluation of coding efficiency. (a) and (c) report results for a GGD with $\mu = 0.55$, and (b) and (d) with $\mu = 0.85$. (a) and (b) do *not* use context-adaptive mechanisms to estimate the probability in any of the coders.

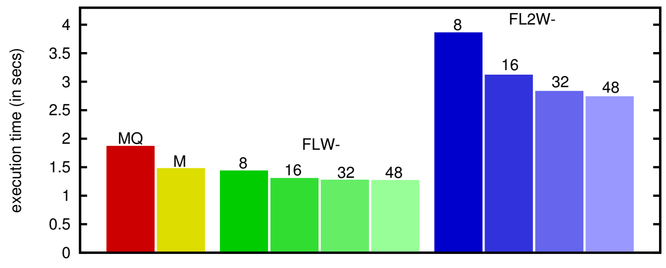
separately to appraise the coders in different conditions. The sequences employed in the tests have $5 \cdot 10^5$ and 10^8 symbols to evaluate coding efficiency and computational throughput, respectively. All coders are programmed in Java. All tests are performed with an Intel Core i7-3770 @ 3.40 GHz using a Java Virtual Machine v1.7.

Fig. 2 evaluates the coding efficiency achieved by the proposed coders when using codewords of different length. The figure also reports the entropy of the source and the performance achieved by the MQ coder of JPEG2000 and the M coder of HEVC. The vertical axis of the figures is the coding rate, expressed in bits per sample (bps), whereas the horizontal axis is the codeword length (i.e., $\mathcal{W}$). Fig. 2(a) and (b) report results when the real probability of the symbol is fed directly to the coder, i.e., when $f(x)$ is used instead of probability estimates. Fig. 2(c) and (d) report results when the probabilities of the symbols are estimated through the context-adaptive mechanisms described before. The experiments that employ context-adaptive mechanisms utilize eight contexts. The probability of a symbol is always in the range $(0, 1)$. This range is divided into eight uniform intervals and each one is assigned to a context. All symbols whose probabilities fall within an interval are coded with the corresponding context. FLW and FL2W use $2\mathcal{T} - 1 = 255$ and $\mathcal{V} = 7$ (see below).

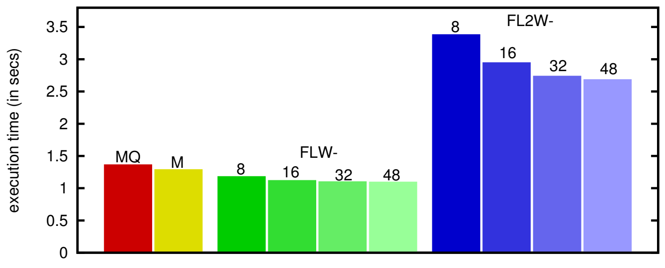
The results of Fig. 2 indicate that the longer the codeword employed by FLW/FL2W, the higher the coding efficiency achieved. This is because the longer the codeword, the less often the interval is reset. When $f(x)$ is utilized, the coding rate achieved by FL2W for codewords of 32 bits or longer is almost that of the source's entropy. For long codewords, FLW (FL2W) achieves a coding efficiency 1% (1.4%) higher than that of the MQ coder when $\mu = 0.55$ and almost equal when $\mu = 0.85$. Compared to the M coder, FLW and FL2W achieve virtually the same coding efficiency. These results suggest that arithmetic coders can be implemented with*out* the renormalization procedure while achieving high efficiency.

With regard to the efficiency of the context-adaptive mechanisms, the results of Fig. 2 indicate that the variable-size sliding window employed by FLW/FL2W is competitive. For long codewords, FLW (FL2W) achieves a coding efficiency 3.3% (3.6%) higher than that of the MQ coder when $\mu = 0.55$, and 2.2% (3.1%) higher when $\mu = 0.85$. Compared to the M coder, the performance of FLW and FL2W is 0.8% and 1.6% higher, respectively. Another observation of Fig. 2 is that the selective interval coding employed by FL2W works well. The coding efficiency achieved by FL2W when using two codewords of length $\mathcal{W}$ is higher than that achieved by FLW when using a codeword of length $2\mathcal{W}$.

Fig. 3 evaluates the computational throughput. The vertical axis of the figure is the execution time, whereas each column



(a)



(b)

Fig. 3: Evaluation of execution time, for the encoder. (a) reports results for a GGD with $\mu = 0.55$, and (b) $\mu = 0.85$.

reports the results for one coder. In this test, the coders employ the context-adaptive mechanisms for probability estimation. The results indicate that the FLW coder always achieves a higher computational throughput than that of the MQ coder. Compared to the M coder, the computational throughput of FLW is, in general, slightly superior. As seen in the figure, the longer the codeword employed by FLW, the lower the execution time. A similar behavior is obtained when context-adaptive mechanisms are *not* employed, though the execution time of all coders is slightly lower (not shown in the figure). Though it depends on the probability distribution of the source, approximately 20% of the total time spent by the proposed coders is devoted to probability estimation. Similar results are obtained for the decoder (not shown due to page constraints).

The results achieved by FL2W in Fig. 3 indicate that the computational complexity of FL2W is higher than that of the MQ and M coders. The use of long codewords helps to improve the throughput of FL2W slightly. Clearly, the competitive coding efficiency achieved by FL2W comes at the expense of high computational costs.

B. Image Coding with JPEG2000

The next set of tests appraises the performance achieved by the proposed coders when they are integrated in a JPEG2000 codec. Evidently, the resulting codestream is not compliant with the standard, though it keeps all its features. All images of the ISO12640-1 corpus are employed. They are grayscale, 8 bps, and of size 2560×2048. The codeword lengths employed by the FLW and FL2W coder are 48 and 32, respectively, whereas $2\mathcal{T} - 1 = 255$ and $\mathcal{V} = 7$. These parameters work well for a large variety of images. Smaller window sizes than 255 achieve the same coding performance, whereas larger penalize the coding performance in 0.1 dB or more. To update the probability for every symbol (i.e., $\mathcal{V} = 1$) enhances coding performance in less than 0.05 dB. JPEG2000 coding parameters are: lossy mode, 64×64 codeblocks, no precincts, and single quality layer codestreams.

Fig. 4 reports the coding efficiency. Results are reported as the peak signal to noise ratio (PSNR) difference between FLW and MQ. Each image is coded at 50 rates uniformly distributed between 0.01 to 5 bps. The straight horizontal line in the figure is the performance achieved by the JPEG2000 implementation that uses the MQ coder. The results indicate

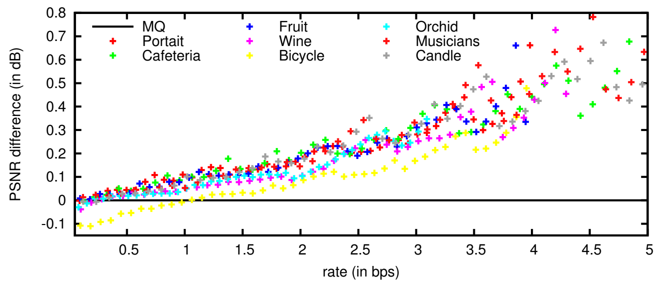


Fig. 4: Evaluation of the coding efficiency achieved by a JPEG2000 implementation employing different arithmetic coders. Results are the difference between FLW and MQ.

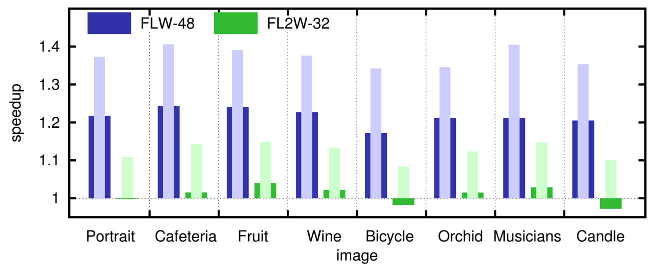
that the FLW coder achieves a higher coding performance than that of the MQ. In terms of rate, FLW generates a codestream that is approximately 1.5%, 1.75%, 2%, and 2.1% shorter than that produced by the MQ coder at 1, 2, 3, and 4 bps, respectively, on average for all images. These percentages are computed as the difference between the codestream length generated by the MQ coder at the reported rate and that generated by the FLW coder to reach the same image quality. The progressive coding gain achieved by the proposed coder as the rate increases is partially caused by the context-adaptive mechanisms of the MQ coder, which are less effective in low bitplanes than at high bitplanes [13]. FL2W achieves a coding performance very similar to that of the FLW (not shown).

Table I reports the performance for the lossless mode of JPEG2000 when codewords of different length are employed. Both FLW and FL2W achieve a coding rate approximately 0.1 bps (or 2%) lower than that of the MQ coder, on average. Differences between the use of different codeword lengths are in the order of 0.01 bps.

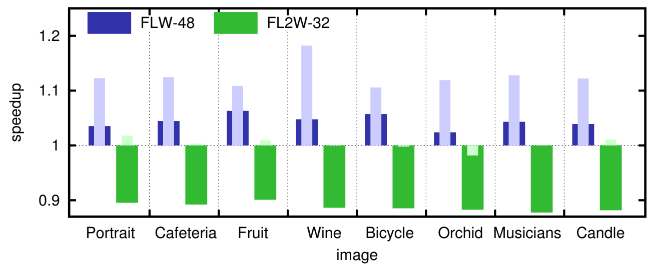
The evaluation of the computational throughput considers the speedup achieved in the tier-1 coding stage of a conventional JPEG2000 codec. The tier-1 implements the bitplane coding engine and the entropy coder. It spends 60∼70% of the total execution time. The widest columns of Fig. 5 report the speedup achieved by FLW/FL2W with respect to MQ. The speedup is computed as the execution time spent by the MQ coder divided by that of the FLW/FL2W coder. The FLW coder achieves speedups around 1.2 for the encoder and around 1.05 for the decoder. For encoding, FL2W spends a similar

TABLE I: Evaluation of the lossless coding performance achieved by JPEG2000 when using different arithmetic coders. Results are reported in bps.

		FLW-			FL2W-	
image	**MQ**	**16**	**32**	**48**	**16**	**32**
"Portrait"	4.38	4.31	4.30	4.30	4.30	4.30
"Cafeteria"	5.28	5.18	5.17	5.16	5.17	5.16
"Fruit"	4.29	4.22	4.21	4.20	4.21	4.20
"Wine"	4.57	4.50	4.49	4.48	4.49	4.48
"Bicycle"	4.37	4.33	4.31	4.31	4.31	4.31
"Orchid"	3.58	3.54	3.53	3.52	3.53	3.52
"Musicians"	5.56	5.44	5.42	5.41	5.42	5.41
"Candle"	5.65	5.55	5.53	5.52	5.53	5.52
average	*4.71*	*4.63*	*4.62*	*4.61*	*4.62*	*4.61*



(a)



(b)

Fig. 5: Evaluation of the computational throughput achieved by FLW/FL2W when using the context-adaptive mechanisms (widest columns) and a static model of probabilities (thinnest columns). (a) and (b) report results for the encoder and the decoder, respectively.

computation time as that of the MQ, with slight variations depending on the image. For decoding, FL2W slows down the decoding process by approximately 10%. These results slightly differ from those obtained with artificially generated symbols due to the different probability distribution of the source.

The previous test employs context-adaptive mechanisms to estimate the probabilities of the symbols. Recently, a probability model that avoids the use of adaptive mechanisms has been introduced in [14], [15]. Its main idea is that the probability of the symbols can be estimated depending on the bitplane and the context in which they are emitted. The thinnest columns of Fig. 5 report the speedup achieved when FLW/FL2W is combined with such a stationary probability model. The throughput is significantly improved for all images. The coding performance achieved with such model is similar to that of JPEG2000 (not shown in the figure).

IV. Conclusions

Context-adaptive binary arithmetic coders employed in image codecs are commonly implemented with variable-to-variable length codes. This paper introduces two context-adaptive binary arithmetic coders that employ codewords of fixed length. They are referred to as FLW and FL2W. FLW employs one interval arithmetic, whereas FL2W employs two to enhance coding efficiency. The proposed coders avoid renormalization, which simplifies their implementation and reduces their computational complexity. An important point disclosed in the experimental results is that the renormalization procedure of arithmetic coders can be removed without affecting their coding efficiency. The context-adaptive mechanism integrated in these coders employs a low-complexity technique based on a variable-size sliding window.

Acknowledgment

The author thanks the associate editor and the anonymous reviewers for their comments.

References

[1] D. Chevion, E. D. Karnin, and E. Walach, "High efficiency, multiplication free approximation of arithmetic coding," in *Proc. IEEE Data Compression Conference*, Apr. 1991, pp. 43–52.
[2] M. Slattery and J. Mitchell, "The Qx-coder," *IBM Journal of Research and Development*, vol. 42, no. 6, pp. 767–784, Nov. 1998.
[3] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 620–636, Jul. 2003.
[4] E. Belyaev, A. Turlikov, K. Egiazarian, and M. Gabbouj, "An efficient adaptive binary arithmetic coder with low memory requirement," *IEEE J. Sel. Topics Signal Process.*, vol. 7, no. 6, pp. 1053–1061, Dec. 2013.
[5] C. G. Boncelet, "Block arithmetic coding for source compression," *IEEE Trans. Inf. Theory*, vol. 39, no. 5, pp. 1546–1554, Sep. 1993.
[6] J. Teuhola and T. Raita, "Arithmetic coding into fixed-length codewords," *IEEE Trans. Inf. Theory*, vol. 40, no. 1, pp. 219–223, Jan. 1994.
[7] D.-Y. Chan, J.-F. Yang, and S.-Y. Chen, "Efficient connected-index finite-length arithmetic codes," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 5, pp. 581–593, May 2001.
[8] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression using variable-to-fixed coding based on arithmetic coding," in *Proc. IEEE Data Compression Conference*, Mar. 2003, pp. 382–391.
[9] M. D. Reavy and C. G. Boncelet, "An algorithm for compression of bilevel images," *IEEE Trans. Image Process.*, vol. 10, no. 5, pp. 669–676, May 2001.
[10] Intel, "Intel 64 and IA-32 architectures optimization reference manual," Tech. Rep. Order Number: 248966-030, Sep. 2014. [Online]. Available: http://www.intel.com/products/processor/manuals/
[11] F. Auli-Llinas and M. W. Marcellin, "Scanning order strategies for bitplane image coding," *IEEE Trans. Image Process.*, vol. 21, no. 4, pp. 1920–1933, Apr. 2012.
[12] F. Auli-Llinas. (2014, Dec.) Context-adaptive binary arithmetic coding with fixed-length codewords. [Online]. Available: http://www.deic.uab.cat/~francesc/research/ac_flw
[13] ——, "Stationary probability model for bitplane image coding through local average of wavelet coefficients," *IEEE Trans. Image Process.*, vol. 20, no. 8, pp. 2153–2165, Aug. 2011.
[14] F. Auli-Llinas, M. W. Marcellin, J. Serra-Sagrista, and J. Bartrina-Rapesta, "Lossy-to-lossless 3D image coding through prior coefficient lookup tables," *ELSEVIER Information Sciences*, vol. 239, no. 1, pp. 266–282, Aug. 2013.
[15] F. Auli-Llinas and M. W. Marcellin, "Stationary probability model for microscopic parallelism in JPEG2000," *IEEE Trans. Multimedia*, vol. 16, no. 4, pp. 960–970, Jun. 2014.