HIGHLY EFFICIENT, LOW COMPLEXITY ARITHMETIC CODER FOR JPEG2000

Francesc Aulí-Llinàs

Department of Information and Communications Engineering
Universitat Autònoma de Barcelona, Spain

ABSTRACT

Arithmetic coding is employed in image and video coding schemes to reduce the statistical redundancy of symbols emitted by coding engines. Most arithmetic coders proposed in the literature generate variable-length codes, i.e., they produce one long codeword of variable size. This requires renormalization operations to control the internal registers of the coder and the propagation of carry bits. This paper introduces an arithmetic coder that generates fixed-length codewords. The main advantage of the proposed coder is that it avoids renormalization procedures, which reduces computational complexity. Also, it uses a variable-size sliding window mechanism to estimate with high precision the probability of the emitted symbols. Experimental results indicate that the proposed coder achieves coding efficiency superior to those coders employed in JPEG2000 and HEVC while having lower computational costs. When integrated in a JPEG2000 implementation, the proposed coder achieves coding gains between 0.5 to 1 dB at medium and high rates, and speedups between 1.1 to 1.3 in the bitplane coding stage.

Index Terms— Arithmetic coding, context-adaptive models.

1. INTRODUCTION

Entropy coding is an indispensable tool in image and video coding to reduce the statistical redundancy of the symbols produced by coding engines. Currently, the most popular entropy coding technique is arithmetic coding. Briefly described, the arithmetic coder begins by segmenting the interval of real numbers $[0, 1)$ in as many subintervals as symbols has the alphabet. The size of the subintervals is chosen according to the probability mass function (pmf) of the symbols. 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 of the message. The transmission of any number within the final subinterval guarantees that the reverse procedure decodes the original message losslessly.

In image and video coding schemes, the arithmetic coder is employed to code every symbol emitted. The reduction of its computational complexity has always been an issue of concern. The first ideas to do so aimed at multiplication-free implementations that perform the subinterval division using bit shifts and adds. The shift coder [1] and the CKW coder [2] are representatives of such ideas. Subsequently, most works approached the subinterval division by means of lookup tables (LUTs). The quasi-arithmetic coder [3], the ELS coder [4], the Z coder [5], and the Q coder [6] belong to this type of implementations. Descendants of the Q coder, namely, the QM coder [7] and the MQ coder [8] 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 [9], which was introduced in the 2000s employing a reduced range of possible subinterval sizes together with LUTs. Enhancements to the M coder to improve its context-adaptive mechanisms and to reduce its computational load have been proposed in [10, 11] and in [12, 13], respectively. More recently, the PIPE coder [14] suggests the replacement of arithmetic coding by a technique of probability interval partitioning to further reduce complexity. Also, architectures to decrease the power consumption [15, 16], to allow parallel threads of execution [17], or to accelerate the execution pipeline [18, 19] have been proposed to minimize the computational costs of arithmetic coding.

A common characteristic of all arithmetic coders mentioned above is that they produce variable-to-variable length codes. This is, a variable number of input symbols are coded with a codeword of a priori unknown length. In terms of implementation, this requires a renormalization procedure to reposition internal registers, and conditionals to control the propagation of carry bits. These operations are needed to produce a single and, commonly, long codeword. They can be avoided if, instead of producing a long codeword of variable length, the coder produces short codewords of fixed length.

Fixed-length arithmetic codes are not new. First approaches were proposed in the nineties [20, 21] with the aim to address some of the disadvantages of conventional arithmetic coding such as poor recovery to channel errors, or lack of random access and partial decoding. Variable-to-fixed length arithmetic coding has also been used in [22, 23] to limit error propagation, and in [24] to compress machine instructions. In the context of image coding, only [25] employs arithmetic coding with codewords of fixed-length, though its application is limited to bilevel images only.

The goal of this work is to explore the use of arithmetic coding with fixed-length codewords in modern image codecs. Our objective is to replace the MQ coder of JPEG2000 with a coder that provides same features as those of the MQ while increasing its computational throughput and coding efficiency. The proposed coder uses codewords of fixed length and a variable-size sliding window mechanism to adjust the probability estimates of the symbols. Experimental results indicate coding performance gains between 0.5 to 1 dB at medium and high rates, and speedups in the JPEG2000 bitplane coding stage between 1.1 to 1.15. When the proposed coder is employed together with a recently introduced stationary probability model, the speedup raises to 1.3.

The paper is organized as follows. Section 2 describes the proposed coder and Section 3 assesses its coding performance and computational throughput with experimental results. The last section concludes with some remarks and outlines lines of future research.

2. ARITHMETIC CODING WITH FIXED-LENGTH CODEWORDS

The main idea behind arithmetic coding with fixed-length codewords (FLW) is to use a codeword of W bits. The subinterval division is carried out into the codeword until it is exhausted, which happens when the size of the subinterval is 0. Then, the subinterval stored in

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 2009-SGR-1224.

the codeword is dispatched and the codeword is reset. Algorithm 1 details the proposed method. The subinterval division is carried out in lines 12 and 15-17. The subinterval is stored in the integer registers L and S, which are the left boundary and the size minus 1 of the subinterval, respectively. The coding of a 0 reduces the subinterval size according to the context's probability estimate. The coding of a 1 increases the left boundary and reduces the subinterval size. When $S = 0$ (line 20 in Algorithm 1), the codeword is exhausted and L and S are reset after dispatching the subinterval that they contain.

The subinterval division is carried out through an integer multiplication and a bit shift to the right, which is denoted by $\gg$. The context's probability estimate, accessed via $\mathcal{P}[c]$, is an integer in the range $\mathcal{P}[c] \in [0, 2^{\widehat{\mathcal{P}}} - 1]$. The context of the symbol coded is provided by the coding engine and is referred to as c. $\widehat{\mathcal{P}}$ is the number of bits employed to represent the probability. The division of the subinterval is implemented as $(S \cdot \mathcal{P}[c]) \gg \widehat{\mathcal{P}}$ in lines 12 and 15. This is faster than using floating point arithmetic or an integer division. This operation should not exceed the size of the registers of the processor, so $W + \widehat{\mathcal{P}} \leq 64$. In our implementation $\widehat{\mathcal{P}} = 15$, whereas W ranges from 8 to 48 (see below).

Lines 1-10 and 13, 19 embody the mechanism to estimate the probabilities of the symbols coded with each context. The main idea is that the probability estimate is updated every $\widehat{\mathcal{U}}$ symbols are coded with the corresponding context. The probability estimate is computed employing a window that contains between $\widehat{\mathcal{W}}$ to $2\widehat{\mathcal{W}}$ symbols except in the beginning of the coding process. Such a variable-size sliding window reduces computational costs and enhances coding efficiency. This is in accordance with other probability models for bitplane image coding [26, 27, 28] and has been implemented in different forms in the literature [29, 30, 13]. The proposed mechanism employs two counters per context, referred to as $\mathcal{T}$ and $\mathcal{Z}$. They are implemented as arrays accessed via c. $\mathcal{T}[c]$ and $\mathcal{Z}[c]$ count the number of symbols, and the number of 0s, coded for context c since the last time that the window was updated, respectively. The conditional in line 1 checks whether $\widehat{\mathcal{U}}$ symbols are coded for context c. $\widehat{\mathcal{U}}$ is of the form $\widehat{\mathcal{U}} = 2^U - 1$, so that a bit-wise AND operation, denoted by $\&$, is employed to check whether so many symbols have been coded. The conditional in line 3 checks whether $\widehat{\mathcal{W}}$ symbols have been coded for that context. If so, subtracts $\widehat{\mathcal{W}}$ from $\mathcal{T}[c]$, and $\mathcal{Z}'[c]$ from $\mathcal{Z}[c]$. $\mathcal{Z}'[c]$ keeps the number of 0s coded since the last time that the window of symbols was updated. This is carried out every time $\widehat{\mathcal{W}}$ symbols are coded with context c except for the first $\widehat{\mathcal{W}}$ symbols. As seen in Algorithm 1, $\widehat{\mathcal{W}}$ is also of the form $\widehat{\mathcal{W}} = 2^{U'} - 1$. Obviously, $\widehat{\mathcal{U}} \leq \widehat{\mathcal{W}}$. For the coding of images with JPEG2000, we found that $\widehat{\mathcal{U}} = 7$ and $\widehat{\mathcal{W}} = 127$ achieve competitive coding efficiency, so they are employed in the experiments of Section 3 except when indicated.

The MQ coder of JPEG2000 utilizes a bit-stuffing mechanism to simplify the resolution of carry bits. A carry bit is produced due to the intrinsic operations performed during subinterval division in conventional arithmetic coding. Without the bit-stuffing mechanism, a carry bit may propagate to already dispatched bytes. The MQ coder stuffs a 0 bit just after dispatching a FF_h byte, ensuring that carry bits in the current registers can not propagate into previous bytes. This also causes that bytes following a FF_h are always in the range 00_h to $8F_h$, which is employed in JPEG2000 to assign markers that can be unequivocally identified in the codestream. The same bit-stuffing procedure as that of the MQ coder is implemented in the "dispatchSubinterval(·)" function of Algorithm 1. The proposed coder does *not* require this bit-stuffing procedure since there are no carry bits to control. Nonetheless, it is included so that the markers in the JPEG2000 codestream can still be unequivocally identified. Also, comparisons of coding efficiency between MQ and FLW can disregard this aspect as the cause behind their differences.

Algorithm 1 FLW encode (x bit to encode, c context)
Initialization: $L \leftarrow 0, S \leftarrow 2^W - 1$

```
 1: if T[c] & Û = Û then
 2:     P[c] ← (Z[c] / T[c]) · 2^P̂
 3:     if T[c] & Ŵ = Ŵ then
 4:         if Z'[c] ≠ 0 then
 5:             Z[c] ← Z[c] − Z'[c]
 6:             T[c] ← T[c] − Ŵ
 7:         end if
 8:         Z'[c] ← Z[c]
 9:     end if
10: end if
11: if x = 0 then
12:     S ← (S · P[c]) ≫ P̂
13:     Z[c] ← Z[c] + 1
14: else
15:     k ← ((S · P[c]) ≫ P̂) + 1
16:     L ← L + k
17:     S ← S − k
18: end if
19: T[c] ← T[c] + 1
20: if S = 0 then
21:     dispatchSubinterval(L)
22:     L ← 0
23:     S ← 2^W − 1
24: end if
```

The decoder has a structure similar to that of the encoder. Algorithm 2 details its procedure. In this algorithm, I is the subinterval stored in the codeword. The probability estimation (lines 1-10 and 25,27) is the same as that of the encoder. The main difference with the encoder is that the decoder needs to compute the left boundary and the size of the new subinterval (lines 16, 17) for every symbol decoded. The encoder computes the left boundary of the new subinterval only when a 1 bit is encoded. This increases slightly the computational complexity of the decoder.

Algorithms 1 and 2 provide an idea of the structure of the FLW coder. In practice, however, the conditionals dictate the operations that are actually carried out to encode each symbol. To provide a precise evaluation of its computational complexity, Table 1 reports the average number of operations per symbol coded carried out by the MQ coder[1] and the proposed FLW coder when encoding and decoding an image. The FLW coder employs codewords of 48 bits in this test (labeled as "FLW-48"), though other lengths do not change results significantly. The results also take into account the operations required to dispatch bytes and to initialize and terminate the codeword (not shown in the algorithms). The operations are classified depending on their type. Assignments, accesses to LUTs/arrays, and conditionals are, in general, computationally simpler than the remaining types shown in the table, which are arithmetic-like operations. The number of computationally simple operations performed by the proposed coder is significantly lower than that of the MQ coder, whereas the number of computationally

[1]The MQ coder employed is that described in [8, Ch. 17.1], which minimizes the number of operations executed.

Algorithm 2 FLW decode (c context)
Initialization: $L \leftarrow 0, S \leftarrow 0, C \leftarrow 0$

1: **if** $\mathcal{T}[c]$ & $\widehat{\mathcal{U}} = \widehat{\mathcal{U}}$ **then**
2: $\quad \mathcal{P}[c] \leftarrow (\mathcal{Z}[c] \,/\, \mathcal{T}[c]) \cdot 2^{\widehat{\mathcal{P}}}$
3: $\quad$ **if** $\mathcal{T}[c]$ & $\widehat{\mathcal{W}} = \widehat{\mathcal{W}}$ **then**
4: $\quad\quad$ **if** $\mathcal{Z}'[c] \neq 0$ **then**
5: $\quad\quad\quad \mathcal{Z}[c] \leftarrow \mathcal{Z}[c] - \mathcal{Z}'[c]$
6: $\quad\quad\quad \mathcal{T}[c] \leftarrow \mathcal{T}[c] - \widehat{\mathcal{W}}$
7: $\quad\quad$ **end if**
8: $\quad\quad \mathcal{Z}'[c] \leftarrow \mathcal{Z}[c]$
9: $\quad$ **end if**
10: **end if**
11: **if** $S = 0$ **then**
12: $\quad I \leftarrow$ getSubinterval()
13: $\quad L \leftarrow 0$
14: $\quad S \leftarrow 2^W - 1$
15: **end if**
16: $k \leftarrow \left((S \cdot \mathcal{P}[c]) \gg \widehat{\mathcal{P}} \right) + 1$
17: $l \leftarrow L + k$
18: **if** $I \geq l$ **then**
19: $\quad x \leftarrow 1$
20: $\quad L \leftarrow l$
21: $\quad S \leftarrow S - k$
22: **else**
23: $\quad x \leftarrow 0$
24: $\quad S \leftarrow k - 1$
25: $\quad \mathcal{Z}[c] \leftarrow \mathcal{Z}[c] + 1$
26: **end if**
27: $\mathcal{T}[c] \leftarrow \mathcal{T}[c] + 1$
28: **return** x

complex operations is higher (see the rows labeled "TOTAL a+b+c" and "TOTAL d+e+f+g" in the table). This is due to the proposed implementation carries out, at least, one multiplication every time a symbol is coded (line 12 or 15 in Algorithm 1 and line 16 in Algorithm 2). This multiplication may be removed through the use of LUTs, similarly as how it is done in [3, 4, 5, 6, 7, 8]. In our implementation, the attempts to do so have always decreased the computational throughput. This may be caused because the access to several memory positions and the addition of assignments increases the computational load more than to carry out one multiplication that can be rapidly processed in the processor pipeline. The division carried out to compute the probability estimates (line 2 in both algorithms) increases negligibly the computational complexity of the coder because it is *not* performed every time a symbol is coded. See in Table 1 that this operation increases in only 0.1 the number of multiplications/divisions performed per symbol coded. The total number of operations carried out by the proposed coder is approximately 15% less than those required by the MQ coder.

3. EXPERIMENTAL RESULTS

Rather than employing real data, the first experimental test assesses coding efficiency and computational throughput when coding artificially generated symbols. This appraises the performance of the coder without intervention of the other mechanisms of the coding engine. The symbols are generated assuming that they are independent and identically distributed. A generalized Gaussian distribution (GGD) with parameters $\sigma = 0.2, \mu = 0.75$ and support in the range $(0, 1)$ is employed to generate the symbols' probability. The GGD

Table 1: Evaluation of the average number of operations per symbol coded carried out by the MQ and the FLW-48 coder when coding with JPEG2000 (lossy mode and 64×64 codeblocks) the "Cafeteria" image of the ISO12640-1 corpus.

	ENCODER		DECODER	
operation type	**MQ**	**FLW-48**	**MQ**	**FLW-48**
a) assignment	7.6	4.4	9.8	6.1
b) LUT/array access	5	3.3	5	3.3
c) conditional	5.4	4.8	5.6	5.1
d) bit-wise	1.9	2.5	5.2	2.6
e) increment/decrement	0.9	2.1	0.9	3.4
f) add/subtract	1.7	1.3	2.2	1.9
g) multiplication/division	-	1.1	-	1.1
TOTAL a+b+c	*18*	*12.5*	*20.4*	*14.5*
TOTAL d+e+f+g	*4.5*	*7*	*8.3*	*9*
TOTAL a+b+c+d+e+f+g	*22.5*	*19.5*	*28.7*	*23.5*

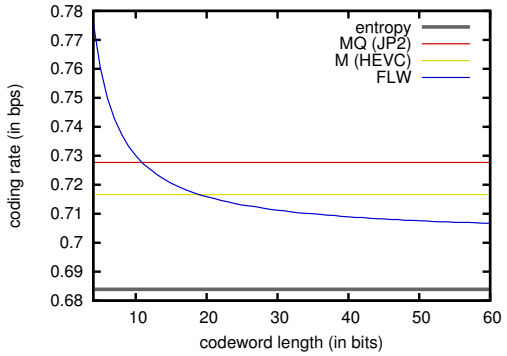

Fig. 1: Evaluation of the coding efficiency when coding artificially generated symbols.

serves to simulate the real probabilities of symbols generated by image codecs, which are commonly centered about a central value (parameter μ in the GGD). Other parameters achieve similar results. The sequences employed in the tests have $5 \cdot 10^5$ and 10^8 symbols to appraise coding efficiency and computational throughput, respectively. Tests of throughput employ sequences with more symbols to measure computational time with more precision. All coders evaluated are programmed in Java. All tests are performed with an Intel Core i7-3770 CPU at 3.40 GHz employing a Java Virtual Machine v1.7 and GNU/Linux v3.5.

Fig. 1 evaluates the coding efficiency achieved by the JPEG2000 arithmetic coder (labeled "MQ") and the FLW coder when using different codeword lengths. For comparison purposes, the arithmetic coder of the HEVC standard (labeled "M"), and the entropy of the source are also included in this figure. All coders use 8 contexts to adapt the probabilities of the symbols. 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 one context. A context codes all symbols whose probabilities fall within its interval. The vertical axis of the figure is the coding rate, expressed in bits per sample (bps), whereas the horizontal axis is the codeword length, expressed in bits. Results indicate that the longer the codeword employed by FLW, the lower the coding rate. Compared to the MQ and M coder, FLW achieves higher efficiency for codewords of 20 bits or longer.

Fig. 2 reports the computational time spent to encode and to decode the sequence of artificially generated symbols. The longer the codeword, the lower the computational time spent by the FLW coder,

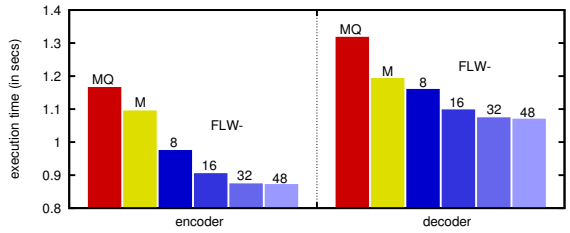

Fig. 2: Evaluation of the computational throughput when coding artificially generated symbols.

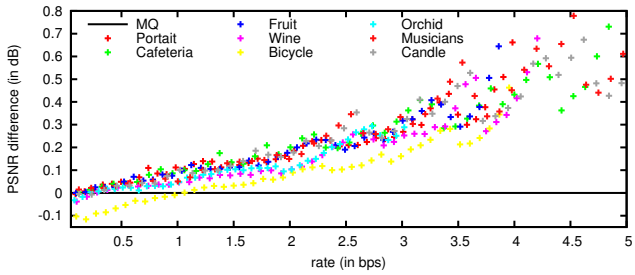

Fig. 3: Evaluation of the coding efficiency achieved by FLW-48. Results are reported as the difference between FLW and MQ.

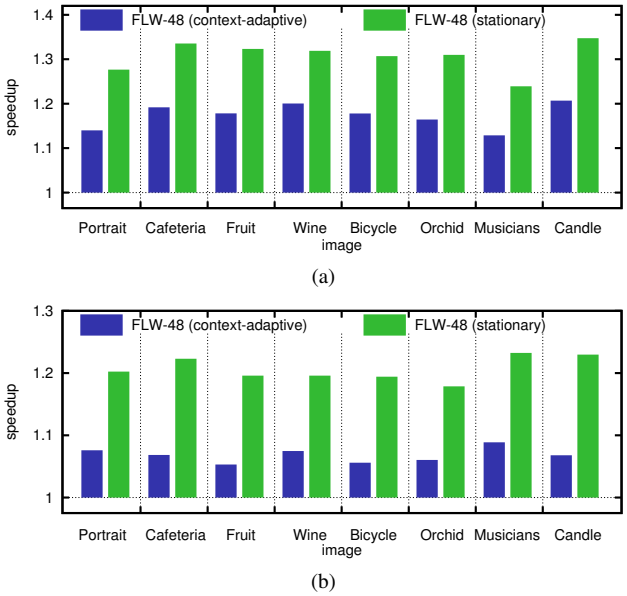

Fig. 4: Evaluation of the computational throughput achieved by FLW when using the context-adaptive mechanisms (blue columns) and a stationary model of probabilities (green columns). (a) reports results for the encoder and (b) for the decoder.

with codewords of 32 and 48 bits achieving the lowest times. This is caused because the use of long codewords calls the dispatching procedure less frequently than when using short codewords. Compared to the MQ and M coder, the proposed coder is between 10% to 25% faster depending on the codeword length.

The following experimental tests employ the images of the ISO12640-1 corpus. The images are grayscale, 8 bps, and of size 2560×2048. The codeword length employed by FLW is 48. The experiments evaluate the coding efficiency and the computational throughput achieved by a JPEG2000 codec when it employs the conventional MQ coder and the proposed FLW coder. JPEG2000 coding parameters are: lossy mode, 64×64 codeblocks, no precincts, and single quality layer codestreams. Our JPEG2000 implementation BOI [31] is employed to carry out these experiments. Evidently, only the codec that employs the MQ coder produces a compliant codestream, though the FLW coder produces a codestream with a syntax that does not undermine any feature of the standard.

Fig. 3 evaluates the coding efficiency achieved when coding the eight images of the corpus. Results are reported as the quality difference, in peak signal to noise ratio (PSNR), between FLW and MQ when coding each image at 50 rates uniformly distributed between 0.01 to 5 bps. The straight horizontal line in the figures is the performance achieved by the JPEG2000 implementation that uses the MQ coder. Plots above this line indicate that FLW achieves higher PSNR than that. Results for each image are depicted with a different color. The results indicate that the FLW coder improves the coding efficiency achieved by MQ significantly. Only for the "Bicycle" image coded at low rates, the MQ coder achieves slightly superior efficiency. At medium and high rates, FLW improves coding efficiency from 0.5 to almost 1 dB.

The evaluation of the computational throughput considers the speedup achieved in the tier-1 coding stage of a conventional JPEG2000 codec. The tier-1 coding stage implements the bitplane coding engine and the entropy coder and spends 60∼70% of the total coding time. The blue columns of Fig. 4(a) and (b) report the speedup achieved by FLW with respect to MQ, respectively for the encoder and the decoder. These results indicate that the FLW coder achieves speedups between 1.1 to 1.2 for the encoder and between 1.05 to 1.1 for the decoder.

All experimental results above employ context-adaptive mechanisms to estimate the symbols' probability. As seen in Algorithms 1 and 2, these mechanisms moderately consume computational resources. Recently, a model of probabilities that avoids the use of adaptive mechanisms to determine the probabilities of the symbols coded in each context has been introduced [28]. 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. Similar ideas are also employed in [32]. The green columns of Fig. 4 report the speedup achieved when the proposed coder is combined with the stationary probability model described in [28]. The use of such a probability model improves the throughput of the proposed coder more. Speedups of approximately 1.3 and 1.2 are respectively achieved for the encoder and the decoder. We note that such speedups are significant in the context of bitplane image coding [27].

4. CONCLUSIONS

This paper introduces an arithmetic coder that uses fixed-length codewords (FLW). Contrarily to most coders in the literature, the proposed coder does not require renormalization procedures to control internal registers and carry bits. This simplifies its implementation, reducing computational complexity. Also, the proposed coder employs a new variable-size sliding window mechanism to estimate with high precision the probability of symbols emitted. FLW is introduced in the framework of JPEG2000. Experimental results indicate coding gains over the conventional MQ coder between 0.5 to 1 dB. In terms of computational throughput, the FLW coder accelerates the bitplane coding stage between 10% to 30%.

5. REFERENCES

[1] G. G. Langdon and J. Rissanen, "Compression of black-white images with arithmetic coding," *IEEE Trans. Commun.*, vol. 29, no. 6, pp. 858–867, Jun. 1981.

[2] 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.

[3] P. Howard and J. S. Vitter, "Design and analysis of fast text compression based on quasi-arithmetic coding," in *Proc. IEEE Data Compression Conference*, Mar. 1992, pp. 98–107.

[4] W. D. Wither, "The ELS-coder: a rapid entropy coder," in *Proc. IEEE Data Compression Conference*, Mar. 1997, pp. 475–475.

[5] L. Bottou, P. G. Howard, and Y. Bengio, "The Z-Coder adaptive binary coder," in *Proc. IEEE Data Compression Conference*, Mar. 1998, pp. 1–10.

[6] M. Slattery and J. Mitchell, "The Qx-coder," *IBM Journal of Research and Development*, vol. 42, no. 6, pp. 767–784, Nov. 1998.

[7] W. Pennebaker and J. Mitchell, *JPEG still image data compression standard*. New York: Van Nostrand Reinhold, 1993.

[8] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image compression fundamentals, standards and practice*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.

[9] 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.

[10] T. Nguyen, H. Schwarz, H. Kirchhoffer, D. Marpe, and T. Wiegand, "Improved context modeling for coding quantized transform coefficients in video compression," in *Proc. IEEE Picture Coding Symposium*, Dec. 2010, pp. 378–381.

[11] K. Vermeirsch, J. Barbarien, P. Lambert, and R. V. de Walle, "Region-adaptive probability model selection for the arithmetic coding of video texture," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2011, pp. 1537–1540.

[12] D. Hong and A. Eleftheriadis, "Memory-efficient semi-quasi renormalization for arithmetic coding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, no. 1, pp. 106–110, Jan. 2007.

[13] 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.

[14] H. Kirchhoffer, D. Marpe, C. Bartnik, A. Henkel, M. Siekmann, J. Stegemann, H. Schwarz, and T. Wiegand, "Probability interval partitioning entropy coding using systematic variable-to-variable length codes," in *Proc. IEEE International Conference on Image Processing*, Sep. 2011, pp. 341–344.

[15] Y. Li and M. Bayoumi, "A three-level parallel high-speed low-power architecture for EBCOT of JPEG 2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 9, pp. 1153–1163, Sep. 2006.

[16] S. Lei, C. Lo, C. Kuo, and M. Shieh, "Low-power context-based adaptive binary arithmetic encoder using an embedded cache," *IET Image Processing*, vol. 6, no. 4, pp. 309–317, Jun. 2012.

[17] S. Chen, S. Chen, and S. Sun, "P3-CABAC: A nonstandard tri-thread parallel evolution of CABAC in the manycore era," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 20, no. 6, pp. 920–924, Jun. 2010.

[18] M. Dyer, D. Taubman, S. Nooshabadi, and A. K. Gupta, "Concurrency techniques for arithmetic coding in JPEG2000," *IEEE Trans. Circuits Syst. I*, vol. 53, no. 6, pp. 1203–1212, Jun. 2006.

[19] M. Rhu and I.-C. Park, "Optimization of arithmetic coding for JPEG2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 20, no. 3, pp. 446–451, Mar. 2010.

[20] C. G. Boncelet, "Block arithmetic coding for source compression," *IEEE Trans. Inf. Theory*, vol. 39, no. 5, pp. 1546–1554, Sep. 1993.

[21] J. Teuhola and T. Raita, "Arithmetic coding into fixed-length codewords," *IEEE Trans. Inf. Theory*, vol. 40, no. 1, pp. 219–223, Jan. 1994.

[22] 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.

[23] H. Chen, "Joint error detection and vf arithmetic coding," in *Proc. IEEE International Conference on Communications*, Jun. 2001, pp. 2763–2767.

[24] 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.

[25] 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.

[26] F. Auli-Llinas, "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.

[27] 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.

[28] ——, "Stationary probability model for microscopic parallelism in JPEG2000," *IEEE Trans. Multimedia*, vol. 16, no. 4, pp. 960–970, Jun. 2014.

[29] B. Ryabko and A. Fionov, "Fast and space-efficient adaptive arithmetic coding," in *Proc. IMA International Conference on Cryptography and Coding*, Dec. 1999, pp. 270–279.

[30] E. Belyaev, M. Gilmutdinov, and A. Turlikov, "Binary arithmetic coding system with adaptive probability estimation by virtual sliding window," in *Proc. IEEE World Congress on Intelligent Control and Automation*, Jun. 2006, pp. 194–198.

[31] F. Auli-Llinas. (2014, May) BOI codec. [Online]. Available: http://www.deic.uab.cat/~francesc/software/boi

[32] 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.