Using Bit Recycling to Reduce the Redundancy in Plurally Parsable Dictionaries

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

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

***Abstract*—Tunstall proposed an efficient algorithm for constructing the optimal dictionary of any particular size to obtain a variable-to-fixed code. More accurately, the algorithm constructs the optimal *uniquely parsable* dictionary. In fact, Savari showed that, if one allows herself to consider *plurally parsable* dictionaries, better codes may be constructed. Savari found a class of plurally parsable dictionaries that outperform the Tunstall code for memoryless, highly skewed, binary sources. This work addresses the redundancy in plurally parsable dictionaries and proposes the use of bit recycling as the means to reduce this redundancy, extending the range of random binary sources that may benefit from a plurally parsable dictionary at the same time. We present a theoretical analysis that evaluates the performance of variable-to-fixed codes based on the Tunstall dictionary and ones based on plurally parsable dictionaries, using Savari's coding on the one hand and coding with bit recycling on the other hand.**

I. Introduction

Many techniques used for lossless data compression, such as LZ77 [1], LZ78 [2], and Tunstall codes [3], are dictionary based and process the input string of symbols by first *parsing* it. The input string s, which is drawn from some alphabet $\mathcal{A}$, is turned into the concatenation of a number n of dictionary entries, $s = s_1 s_2 \dots s_n$. The entries may have variable lengths. Each entry is selected from the dictionary that is available at the corresponding step. In the case of Tunstall coding, the dictionary $\mathcal{D}_T$ remains constant for the whole process. It has size M and all entries get mapped to codewords of the same length. Tunstall coding is said to be *variable-to-fixed* (*VF*). The dictionary used in any dictionary-based compression technique has to be *exhaustive* (or *complete*). This means that, for any infinite string σ drawn from $\mathcal{A}$, it is possible to find a prefix of σ that is an entry of the dictionary. Tunstall devised a simple and efficient algorithm to construct an optimal *uniquely parsable* dictionary of size M for strings generated by identically and independent random variables [3]. The fact that $\mathcal{D}_T$ is *uniquely parsable* means that s can only be parsed in a unique way. A dictionary is uniquely parsable if, for any infinite string σ drawn from $\mathcal{A}$, there exists one and only one entry of the dictionary that is a prefix of σ. Said differently, no entry of the dictionary may be a prefix of another one. Note that we ignore the issue of handling the end of s, which might involve partial matching with a dictionary entry.

Many techniques, such as LZ77 and LZ78, use a *plurally parsable* dictionary; i.e. there usually exists a dictionary entry that is a prefix of another entry. This implies that there exists some string s that can be parsed in two or more ways. As a consequence, s may be encoded in more than one way; i.e. different sequences of codewords may be transmitted to the decoder and any one of these sequences can be decoded back into s. We call this property *the multiplicity of encodings* (*ME*). Necessarily, *ME* adds redundancy to a coding technique. Still, Savari [4] showed that it was possible to design a plurally parsable dictionary that outperforms the Tunstall dictionary when facing a very skewed memoryless binary source.

The *bit recycling* technique [5], [6] has been introduced to reduce the redundancy caused by *ME*. The idea behind the expression "bit recycling" is that bits that would have been wasted due to *ME* get turned into useful information. Variants of bit recycling have been applied on the LZ77 algorithm. Experimental results showed that bit recycling achieves a reduction of about 9% in the size of files compressed by `Gzip` by exploiting *ME* in LZ77 [7].

This works aims to show and calculate the redundancy caused by *ME* in plurally parsable dictionaries designed by Savari, and to show how this redundancy is an opportunity for bit recycling to outperform the Tunstall and Savari encodings for highly skewed, memoryless, binary sources. At the same time, bit recycling extends the range of random binary sources that may benefit from a plurally parsable dictionary.

The rest of the paper is organized as follows. The Tunstall code and a plurally parsable dictionary are briefly reviewed in Section II. The principle of bit recycling and how it can be used to lessen the redundancy in the plurally parsable dictionaries are presented in Section III. Section IV contains a theoretical analysis and the results obtained by applying bit recycling on the plurally parsable dictionaries. Finally, the conclusion is given in Section V.

II. Tunstall's and Savari's dictionaries

In this work, we consider a random binary source $\mathbf{X}$ with alphabet $\mathcal{A} = \{0, 1\}$. A string $s = b_1 b_2 \dots b_n$ produced by the source has each bit b_i generated identically and independently by X_i. Variable X_i generates 0 with probability p_0 and 1 with probability p_1. Let M be the desired size of the dictionaries. Let $N = M - 2$ be the number of entries that will be identical in both the uniquely parsable dictionary and the plurally parsable one. We are interested in sources that are skewed and, in particular, we assume that the source obeys $p_0^N \geq p_1$.

Tunstall's algorithm for constructing an optimal uniquely parsable dictionary of *M* entries for a source like $\mathbf{X}$ is well known [3]. We merely state that, since the source is skewed, the dictionary $\mathcal{D}_T$ constructed by Tunstall's algorithm has to

Entry	$\mathcal{D}_T$	$\mathcal{D}_S$	Codeword
1	1	1	000
2	01	01	001
3	001	001	010
4	0001	0001	011
5	00001	00001	100
6	000001	000001	101
7	0000001	000000	110
8	0000000	000000000000	111

TABLE I. THE TUNSTALL AND SAVARI DICTIONARIES FOR A SKEWED BINARY SOURCE

be as follows.

$$\mathcal{D}_T = \{0^l 1 \mid 0 \leq l \leq N\} \cup \{0^{N+1}\} \tag{1}$$

It is easy to verify that $\mathcal{D}_T$ is uniquely parsable.

Savari considered a family of dictionaries, for $i \geq 2$ [4]:

$$\{0^l 1 \mid 0 \leq l < N\} \cup \{0^N, 0^{iN}\}. \tag{2}$$

It is clear that these dictionaries are plurally parsable. In this work, we consider the simplest of these dictionaries: the one ($\mathcal{D}_S$) for which $i = 2$.

$$\mathcal{D}_S = \{0^l 1 \mid 0 \leq l < N\} \cup \{0^N, 0^{2N}\} \tag{3}$$

For example, suppose that we want to construct $\mathcal{D}_T$ and $\mathcal{D}_S$ for a binary source with the probabilities $p_0 = 0.9$ and $p_1 = 0.1$. Suppose the desired dictionary size is $M = 8$ (so $N = 6$). Note that the source obeys the aforementioned assumption: $p_0^N \geq p_1$. Table I presents $\mathcal{D}_T$ and $\mathcal{D}_S$. One may observe that, in $\mathcal{D}_S$, entry 7 is a prefix of entry 8, which definitely makes $\mathcal{D}_S$ a plurally parsable dictionary. Despite the fact that this introduces *ME* and, consequently, incurs some redundancy, Savari has shown that $\mathcal{D}_S$ outperforms $\mathcal{D}_T$ for the skewed source. To illustrate the presence of *ME*, let us see that an encoder using $\mathcal{D}_S$ may parse the example string $s = 0^{21}1$ in three ways: (i) 000000-000000-000000-0001; (ii) 000000000000-000000-0001; (iii) 000000-000000000000-0001. We say that the encoder is offered three *choices*. These three choices result in the following codewords, respectively: (i) 110-110-110-011; (ii) 111-110-011; (iii) 110-111-011. The source and the two dictionaries presented in this very paragraph are used as a running example in the remainder of the paper.

From the dictionaries defined by both Tunstall and Savari, we can derive the corresponding codes. Let us start with the one for Tunstall. First, let us define d_T as the (partial) function that merely maps entries of $\mathcal{D}_T$ to their respective codewords. Next, let us recursively define c_T as the function that parses and encodes an input string according to $\mathcal{D}_T$.

$$c_T(u \cdot v) = d_T(u) \cdot c_T(v), \text{ where } u \in \mathcal{D}_T \tag{4}$$

We observe that c_T is deterministic in the way it parses its argument since there is always a unique entry in $\mathcal{D}_T$ that matches a prefix of the argument. In a strict sense, this definition of c_T is incomplete because we do not describe how to handle arguments that are too short for a match of occur. However, as stated above, we choose to ignore the issue of encoding the end of the input string.

Now, we define the code that derives from $\mathcal{D}_S$. First, we analogously define d_S as a mere mapping of entries of $\mathcal{D}_S$ to codewords. Next, we turn to defining c_S. Notice that it is possible to define c_S in many different ways as $\mathcal{D}_S$ is plurally parsable. Savari forces c_S to be deterministic by adopting a greedy rule: whenever the argument starts with enough '0's, entry M is used.[1] The definition is the following.

$$c_S(0^l 1 \cdot w) = \begin{cases} d_S(0^{2N}) \cdot c_S(0^{l-2N} 1 \cdot w), & \text{if } l \geq 2N \\ d_S(u) \cdot c_S(v), & \text{otherwise,} \\ \quad \text{where } u \cdot v = 0^l 1 \cdot w \text{ and } u \in \mathcal{D}_S & \end{cases} \tag{5}$$

This rule is sufficient to make c_S deterministic since $\mathcal{D}_S - \{0^{2N}\}$ forms a uniquely parsable dictionary. Once again, we have ignored the issue of the end of the input string.

III. BIT RECYCLING AND THE REDUNDANCY DUE TO *ME*

In the running example, we saw an instance of *ME* in that $s = 0^{21}1$ can be parsed in three ways using $\mathcal{D}_S$. Although Savari deliberately forbids alternative parses by imposing a greedy parse rule, she does acknowledge that there is *ME*. However, *ME* adds redundancy and merely forbidding the use of all but one desired parse does not remove that redundancy. So Savari's technique is still able to feature an improvement due to the fact that the advantage brought by removing the unique-parsability constraint more than compensates for the disadvantage caused by introducing *ME*.

Bit recycling adopts a more opportunistic approach. Instead of ignoring *ME* or trying to avoid the generation of *ME* in the first place, which might be a computationally daunting task, bit recycling *uses ME* to carry information implicitly from the coder to the decoder. Let us illustrate this using the running example. The encoder is offered three choices for the encoding of $s = 0^{21}1$. Let us denote these choices by Ч_1, Ч_2, and Ч_3, respectively. Encoding s according to these choices would cost 12, 9, and 9 bits, respectively. Let us suppose that the encoder rejects Ч_1 due to its higher cost. The encoder is left with Ч_2 and Ч_3, both having the same cost. While c_S would systematically select Ч_2, there is no *economical* reason to prefer Ч_2 over Ч_3 and vice versa. However, by adding a few extra operations in the programming of both the coder and the decoder, it would be possible to use this situation as an opportunity to transmit one bit of information for free. If we let b be a bit we want to transmit for free, then the coder ought to choose Ч_{b+2}. The decoder, after receiving this particular encoding of s, would be able to acknowledge that the selection of Ч_{b+2} among Ч_2 and Ч_3 was intentional and it would then be able to recover the free bit b. Thanks to the free bit, our two choices Ч_2 and Ч_3, taken together, can be viewed as costing 8 bits since after encoding one of them using 9 bits, we obtain the transmission of 1 bit for free, resulting in a "net cost" of 8 bits. Why should the coder/decoder pair take the trouble of transmitting free bits, like this? Because, by doing so each time opportunities arise, a valuable amount of information may get transmitted. This information may describe, say, the end of s and, this way, allow the very transmission of s to terminate "ahead of time" due to the opportunistic transmission of the rest via free bits.

The example of bit recycling above illustrates the essence of *Huffman bit recycling* (*HuBR*). *HuBR* is the first form of bit

[1] Yamamoto and Yokoo also adopt a greedy-parsing rule to make their plurally parsable *VF* codes deterministic [8].

recycling which, in the presence of multiple choices offered to the encoder, allows to recycle one or many bits. *HuBR* is sub-optimal in recovering a compensation for the redundancy incurred by *ME*. Fundamentally, this is due to the fact that only (one or many) *whole* bits may get recycled. This fact constrains the recycling to be performed on an *integer* number of bits at a time. So, in *HuBR*, either 1 or 2 bits may get recycled in a situation where, "ideally", it is somehow 1.5 bits that should have been recycled. The wholeness of the recycled bits forces the encoder/decoder pair to manipulate all the choices as a set. It also leads to the dropping of choices, when they are too costly, and to other inefficiencies. We keep the presentation of *HuBR* to this superficial description. The construction of optimal whole-bit recycling codes was presented in 2009 [9].

Later, the authors have proposed a more efficient bit recycling technique, arithmetic code bit recycling (*ACBR*) [10], which has the ability to achieve perfect recycling[2] and to exploit even the costly choices (like Ч_1), since it is built on arithmetic code. *ACBR* features the ability to assign fractional numbers of recycled bits to options. Moreover, it is always possible to handle multiple choices two at a time. Thus, to any two choices Ч_1 and Ч_2, of respective costs c_1 and c_2, we may assign recycled codewords of length r_1 and r_2, with no need for special care for the respective magnitudes of the costs. In *ACBR*, instead of assigning a specific recycled bit sequence to a choice as in *HuBR*, a sub-interval $I_i = [L,\, H)$ contained within the unit interval $[0.0,\, 1.0)$ gets assigned. The amount of information that gets recycled from the selection of a choice Ч_i is the self-information specified by the associated sub-interval I_i. *ACBR* cumulatively divides the unit interval into sub-intervals proportionally according to the selection probabilities of the choices as follows. The average cost of two choices Ч_1 and Ч_2, of respective costs c_1 and c_2, is given by:

$$\begin{aligned} c_1 \otimes c_2 \quad &= \quad \min_{r_1,\, r_2}(c_1 - r_1)\tfrac{1}{2^{r_1}} + (c_2 - r_2)\tfrac{1}{2^{r_2}} \\ &\text{subject to } \tfrac{1}{2^{r_1}} + \tfrac{1}{2^{r_2}} \geq 1 \ . \end{aligned} \tag{6}$$

Accordingly, the unit interval (or the current recycling interval, with all proportions kept, if the unit interval has already been subdivided) is divided into two sub-intervals: I_1, of length $\frac{1}{2^{r_1}}$, and I_2, of length $\frac{1}{2^{r_2}}$. Since *ACBR* is not forced to constrain r_1 and r_2 to be integers, then it has the ability to achieve the minimum (optimal) value of $c_1 \otimes c_2$. We have found that the minimum value for $c_1 \otimes c_2$ is reached at:

$$r_1 = \lambda - c_2 \qquad \text{and} \qquad r_2 = \lambda - c_1 \tag{7}$$

where $\lambda = \log_2(2^{c_1} + 2^{c_2})$. In conclusion, Ч_1 has the probability $p_1 = \frac{1}{2^{r_1}}$ (which is the length of I_1) to recycle r_1 bits, and Ч_2 has the probability $p_2 = \frac{1}{2^{r_2}}$ (which is the length of I_2) to recycle r_2 bits.

For the sake of illustration, let us consider the choices (i), (ii), and (iii) of the running example again. *ACBR* will exploit the three of them, even if choice (i) is much more expensive than the others. The three choices are joined in two steps, two choices being joined at each step. In the first step, let us consider choices (ii) and (iii) to be Ч_1 and Ч_2 and their costs are $c_1 = c_2 = 9$. The value of $c_1 \otimes c_2 = 8$ bits is obtained by assigning equal numbers of recycled bits to the choices; i.e. $r_1 = r_2 = 1$. The interpretation of *ACBR* for this assignment is that the encoder and decoder will divide the unit interval $[0,\, 1)$ into two sub-intervals proportionally according to the choices probabilities. Therefore, each of Ч_1 and Ч_2 have the same probability, $\frac{1}{2}$, to be selected and each of them would cause $\log_2 \frac{1}{1/2} = 1$ bit to be recycled. We may now consider that choices (ii) and (iii) have been joined into a new choice (ii*) that costs 8 bits. In the second step, let Ч_1 be the new choice (ii*) and Ч_2 be choice (i). We have $c_1 = 8$ and $c_2 = 12$. Now, the value of $c_1 \otimes c_2 = 7.91$ bits is achieved by assigning r_1 to 0.09 and r_2 to 4.09. Then the new division of the unit interval $[0.0,\, 1.0)$ will be as follows: choice (i) is assigned a sub-interval of length $\frac{1}{2^{r_2}} = 0.06$ and a sub-interval of length $(1 - 0.06 = 0.94)$ is divided equally between choices (ii) and (iii), so we can say that the following intervals: $[0.0,\, 0.47)$, $[0.47,\, 0.94)$, and $[0.94,\, 1.0)$ are assigned to choices (ii), (iii), and (i), respectively. The appropriate sub-interval will be selected by the coder according to the information that needs to be transmitted "for free". For more details about *ACBR* we refer the reader to [7] and [10].

[2]That is, *ACBR* is able to recycle as many bits as there are redundant bits caused by *ME*, provided the list of the available choices can be established efficiently, computationally speaking.

IV. Theoretical analysis and results

In this section, and based on the aforementioned assumptions, we aim to evaluate the performance of constructing *VF* codes for a string, s, taken from a binary source, $\mathcal{A} = \{0, 1\}$, using the Tunstall code and plurally parsable dictionaries without and with *ACBR*. For convenience, we view s as a concatenation of regular chunks, r, of the form 0^*1, followed by an irregular chunk of the form 0^*. For the sake of simplicity, we assume that a '1' bit gets added to the end of s at coding time and gets stripped at decoding time. Therefore, s can be viewed as a sequence of only regular chunks. Then only regular chunks need be taken into consideration in the analysis, below. Asymptotically, the single irregular chunk has a negligible impact on the coding efficiency.

In order to be able to evaluate the performance of each technique, we need to calculate the average number $A_{\boxdot}$ of bits (the expected length of the codeword) produced per regular chunk. These calculations are based on the number $C_{\boxdot}(0^l1)$ of bits produced for the chunk 0^l1, for $l \geq 0$. Let us start with the Tunstall code. The probability of 0^l1 to be the next chunk to encode is $p_0^l \cdot p_1$. Chunk 0^l1 needs $C_T(0^l1)$ bits to be encoded. If we let Z be the length of the fixed-length codeword of a dictionary with M entries (i.e. $Z = \log_2 M$ bits), we have:

$$C_T(0^l1) = |c_T(0^l1)| = Z \cdot \left(\left\lfloor \frac{l}{M-1} \right\rfloor + 1 \right) \ . \tag{8}$$

Accordingly, A_T is the expectation of C_T. We were able to reduce the infinite sum and obtain a closed formula for A_T.

$$A_T = \sum_{l=0}^{\infty} C_T(0^l1) \cdot p_0^l \cdot p_1 = Z \cdot \left(1 + \frac{p_0^{M-1}}{1 - p_0^{M-1}} \right) \tag{9}$$

In the same way, let us consider the Savari code. Each chunk 0^l1 needs $C_S(0^l1)$ bits to be encoded.

$$C_S(0^l1) = |c_S(0^l1)| = Z \cdot \left(\left\lfloor \frac{l}{2 \cdot (M-2)} + \frac{1}{2} \right\rfloor + 1 \right) \tag{10}$$

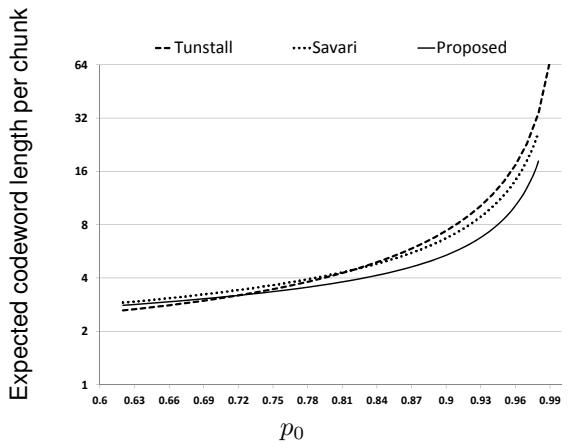

Fig. 1. A comparison of A_T, A_S, and A_R for $M = 4$

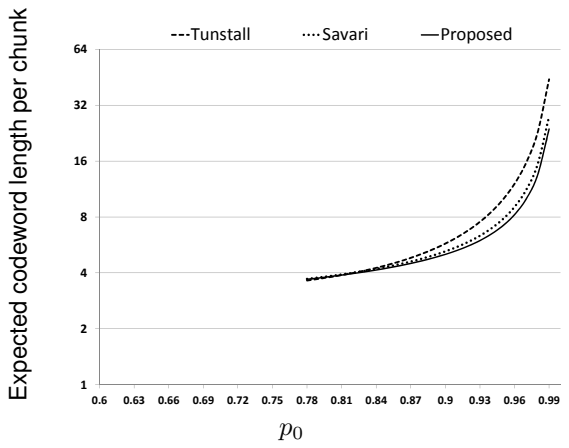

Fig. 2. A comparison of A_T, A_S, and A_R for $M = 8$

Accordingly, A_S is the expectation of C_S and we were once again able to derive a closed formula.

$$A_S = \sum_{l=0}^{\infty} C_S(0^l 1) \cdot p_0^l \cdot p_1 = Z \cdot \left(1 + \frac{p_0^{M-2}}{1 - p_0^{2(M-2)}}\right) \quad (11)$$

Now, let us calculate A_R by using *ACBR* on $\mathcal{D}_S$ as follows. The cost of encoding chunk $0^l 1$ is calculated with the help of Equation (6) and according to the following recurrence:

$$C_R(0^l 1) = \begin{cases} Z, & \text{if } l < N \\ 2Z, & \text{if } N \le l < 2N \\ \begin{pmatrix} Z + C_R(0^{l-N}1) \otimes \\ Z + C_R(0^{l-2N}1) \end{pmatrix}, & \text{otherwise.} \end{cases} \quad (12)$$

The idea is that, when the chunk starts with at least $2N$ '0's, we may as well extract entry 0^N or entry 0^{2N} as a prefix of the chunk and then continue with the coding of the remainder. Now A_R can be calculated as follows.

$$A_R = \sum_{l=0}^{\infty} C_R(0^l 1) \cdot p_0^l \cdot p_1 \quad (13)$$

In order to be able to compare A_T, A_S, and A_R for dictionaries of different sizes and for different binary sources, we have computed the averages for $4 \le M \le 8$ and for p_0 high enough to obey the condition $p_0^N \ge p_1$ (which is $p_0 \ge 0.62$, for $M = 4$, and $p_0 \ge 0.78$, for $M = 8$). Unfortunately, since we could not derive a closed formula for A_R, as we did for A_T and A_S, we have computed A_R numerically using the aforementioned recurrence. We computed A_R with a precision of 10^{-8} by considering the summation from $l = 0$ to large-enough l's for all values of M and p_0. The computed values for $M = 4$ is plotted in Figure 1. It is clear that a significant improvement (less redundancy), represented by the gap between the curves, is achieved by *ACBR*. Notice that *ACBR* extends the range of the binary sources that may benefit from plurally parsable dictionaries from $p_0 \in [0.8, 0.99]$ to $p_0 \in [0.72, 0.99]$. We found that, as M increases, the gap between the curves decreases and the value of p_0 at which *ACBR* starts to outperform the Tunstall and Savari codes increases. Therefore a smaller improvement has been achieved for $M = 8$, as shown in Figure 2.

V. Conclusion

We have proposed the principle of *ACBR* as a solution for the redundancy caused by *ME* in plurally parsable dictionaries. The theoretical analysis showed that applying *ACBR* on plurally parsable dictionaries, for memoryless, highly skewed, binary sources, achieves a significant improvement (less redundancy), especially for small dictionaries. Moreover, it has been shown that *ACBR* widens the class of binary sources that may benefit from a plurally parsable dictionary.

References

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–342, 1977.

[2] ——, "Compression of individual sequences via variable rate coding," *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, 1978.

[3] B. P. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Institute of Technology, 1967.

[4] S. Savari, "Variable-to-fixed length codes and plurally parsable dictionaries," in *Proceedings of Data Compression Conference, 1999. DCC'99*, March 1999, pp. 453–462.

[5] D. Dubé and V. Beaudoin, "Recycling bits in LZ77-based compression," in *Proceedings of Sciences Électroniques, Technologies de l'Information et des Télécommunications (SETIT 2005)*, March 2005.

[6] ——, "Improving LZ77 data compression using bit recycling," in *Proceedings of ISITA*, October 2006.

[7] D. Dubé and V. Beaudoin, "Improving LZ77 bit recycling using all matches," in *Proceedings of the IEEE International Symposium in Information Theory*, Toronto, Ontario, Canada, July 2008, pp. 985–989.

[8] H. Yamamoto and H. Yokoo, "Average-sense optimality and competitive optimality for almost instantaneous VF codes," *IEEE Transactions on Information Theory*, vol. 47, no. 6, pp. 2174–2184, September 2001.

[9] D. Dubé and V. Beaudoin, "Constructing optimal whole-bit recycling codes," in *Proceedings of the IEEE Information Theory Workshop*, Volos, Greece, June 2009, pp. 27–31.

[10] A. Al-Rababa'a and D. Dubé, "Adaptation of bit recycling to arithmetic coding," in *Proceedings of the International Workshop on Systems, Signal Processing, and their Applications*, Algiers, Algeria, May 2013.