Low-Density Parity-Check Code Design Techniques to Simplify Encoding

J. M. Perez[1] and K. Andrews[2]

This work describes a method for encoding low-density parity-check (LDPC) codes based on the accumulate–repeat–4–jagged–accumulate (AR4JA) scheme, using the low-density parity-check matrix H instead of the dense generator matrix G. The use of the H matrix to encode allows a significant reduction in memory consumption and provides the encoder design a great flexibility.

Also described are new hardware-efficient codes, based on the same kind of protographs, which require less memory storage and area, allowing at the same time a reduction in the encoding delay.

I. Introduction

The NASA proposal for the Consultative Committee for Space Data Systems (CCSDS) experimental specification CCSDS 131.1-O-2 [1] describes a set of low-density parity-check (LDPC) codes for near-Earth and deep-space applications. That experimental specification describes two kinds of codes. The second one is a set of accumulate–repeat–4–jagged–accumulate (AR4JA) codes, which have characteristics particularly well suited to deep-space applications. One drawback of these codes is that, unlike the codes in other standards (such as Digital Video Broadcasting (DVB)-S2 [3], 802.11n [4], and 802.16e), the parity-check matrices have not been defined to be encoding-efficient, but to improve the bit-error rate (BER) performance as much as possible. This fact allows the AR4JA codes to show better BER performance than some other standard codes. On the other hand, it makes necessary the use of dense generator matrices to encode the AR4JA codes defined in the proposal. In Figs. 1 and 2, an example parity-check matrix, H, and corresponding generator matrix, G, are displayed to show the difference between these two in terms of sparseness. The example in these figures is for a code of rate 1/2 with $M = 8$ and $k = 64$, where M is the size of the circulants that form the H matrix, and k is the information block length. The matrices shown accommodate $n = 160$ code symbols, but the last fifth of these are punctured (not transmitted over the channel) to yield a rate 1/2 code. We call this Code 1.

The objective of this work is to show that, by doing some simple operations that do not alter the BER performance of the proposed AR4JA codes, a sparse H matrix can be used for encoding.

[1] Microelectronics Engineering Group of the University of Cantabria, Spain, visiting scholar in the Communications Systems and Research Section.

[2] Communications Systems and Research Section.

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, and the University of Cantabria through the Spanish Ministerio de Educación y Ciencia (MEC) project TEC-2005-03301/MIC.

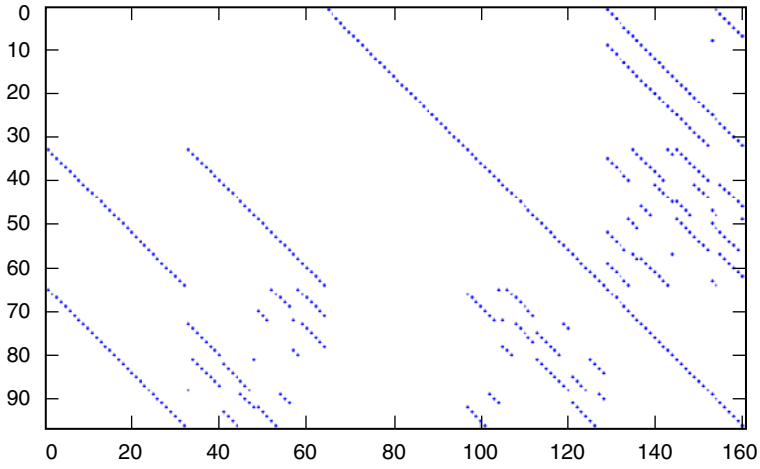

Fig. 1. Parity-check matrix for Code 1.

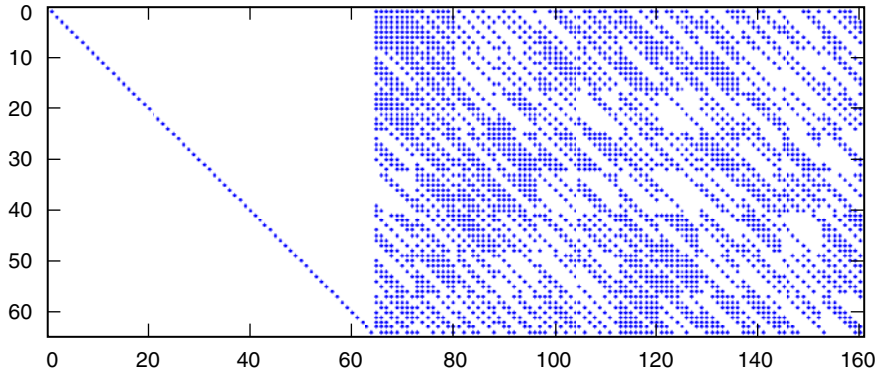

Fig. 2. Systematic generator matrix for Code 1.

In Section II, we describe a modification of the H matrix (without modifying the code) required to encode the current AR4JA codes using the sparse H matrix. In Section III, an alternative method will be described to construct encoder-efficient AR4JA codes. Some examples of these codes will be presented. So far, the codes designed to be encoding-aware do not exhibit the same BER performance as those defined in the current CCSDS specification, but the design of such codes using the progressive edge growth (PEG) or approximate cycle extrinsic–message–degree (ACE) algorithms [7,8] with certain restrictions may be a good choice for future designs. Finally, in Section IV, three hardware architectures will be compared in terms of area requirements (including XORs, ANDs, flip-flops, and memory requirements).

II. Encoders for the Current AR4JA Codes

Permuting the code symbols does not alter the BER performance of the code. Reordering of the code symbols is accomplished by permuting the columns of the H matrix and by permuting the columns of the G matrix to match. This technique can be used to reveal a desired structure in the H matrix.

With this idea in mind, we first define three groups (designated 1, 2, and 3 in Fig. 3) of $4M$ columns in the original H matrix, as shown in Fig. 3. If we now permute these groups, the result shown in Fig. 4 is obtained. We call this Code $1'$. The matrix displayed in Fig. 4 has an approximate lower triangular structure, i.e., above a certain diagonal on the right side of the matrix, all the elements are zeros. In [2], Richardson and Urbanke partition an H matrix into six sub-matrices, as shown in Fig. 5, with dimensions

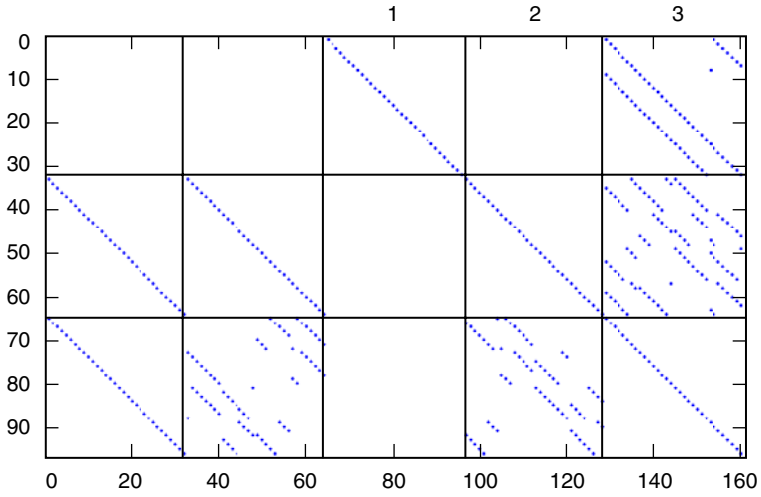

Fig. 3. *H* **matrix for Code 1.**

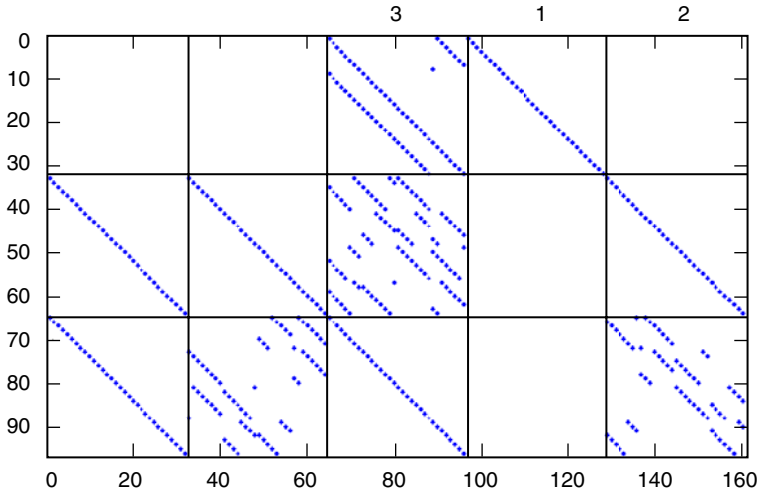

Fig. 4. Modified *H* **matrix for Code 1′.**

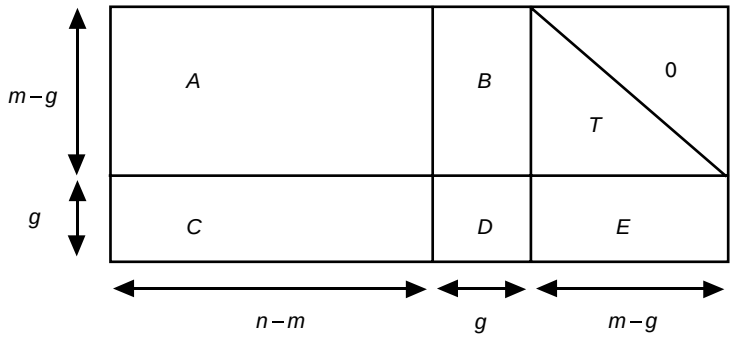

Fig. 5. Richardson and Urbanke's partition of an approximate lower triangular parity-check matrix into sub-matrices [2].

as indicated. The upper-right sub-matrix T is a lower triangular matrix, and all of the elements on the diagonal are ones. If we take a look at the H matrix defined in Fig. 4, we can see that it can be partitioned to match the block matrix shown in Fig. 5. The resulting sub-matrices A, B, C, D, E, and T are displayed in Fig. 6.

It is not the aim of this document to go through the equations of Richardson and Urbanke's encoding method, and we refer the reader to [2] as a guide for these equations. Here we only summarize the main expressions to calculate the parity bits, keeping the same notation used in [2].

Let

$$\phi = ET^{-1}B + D \tag{1}$$

Due to the design of AR4JA codes, T is the identity matrix, which simplifies the calculations, and

$$\phi = EB + D \tag{2}$$

Let

$$p_1^T = \phi^{-1}(ET^{-1}A + C)s^T \tag{3}$$

$$= \phi^{-1}(EA + C)s^T \tag{4}$$

where p_1 represents the punctured parity symbols and s represents the vector of input bits to be encoded.

Finally, we can compute the parity symbols to be transmitted as

$$p_2^T = T^{-1}(As^T + Bp_1^T) = As^T + Bp_1^T \tag{5}$$

Encoding by this method involves several sparse matrices but only one dense matrix, ϕ^{-1}, of size $4M \times 4M$. This matrix can be precomputed, and encoding requires multiplying this matrix by a vector.

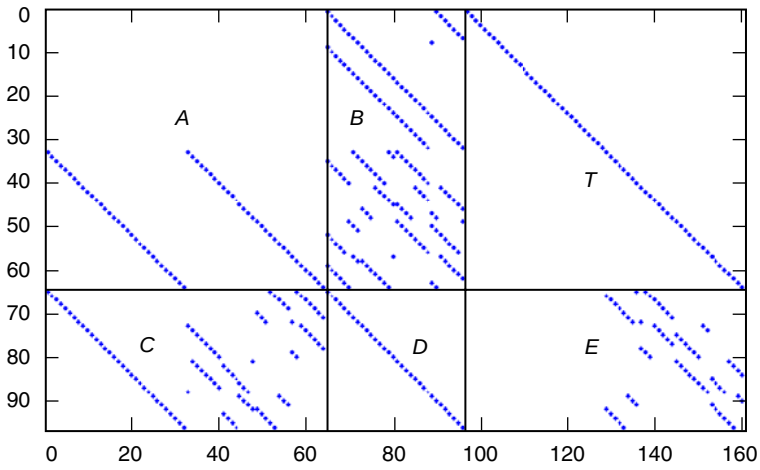

Fig. 6. *H* **matrix for Code 1′ partitioned into the sub-matrices defined in [2].**

III. Encoding-Aware Code Design

In this section, we describe a method for designing hardware-aware codes similar to the AR4JA code, and encoders for the codes. We will also provide two examples and the steps for obtaining the desired H matrix.

Some code standards, such as 802.16e and 802.11n, include LDPC codes that are easily encodable by using the Richardson–Urbanke method. They provide an easy way of calculating the circulant matrices on the right side of the parity-check matrix (the parity part) in such a way that ϕ is the identity matrix, so that its inverse is also.

The scheme proposed is displayed in Fig. 7 [6]. On the right, the architecture used for these codes is displayed. In this figure, shaded squares represent circulant sub-matrices and blank squares zero matrices. The shaded squares filled with a "0" represent the identity matrix, and those filled with an "a" represent the identity matrix circularly shifted to the right "a" times. From Eq. (1), with B, D, E, and T as shown, we find ϕ^{-1} is the identity matrix, and this greatly simplifies Eq. (3). On the other hand, the fixed architecture of the right side and the number of degree-two variable nodes degrade the BER performance.

In the remainder of this section, we try to obtain the same benefits in terms of encoder complexity without significant degradation in the BER performance of the code. The idea is simple: we wish to keep the code's protograph unchanged, keep T as the identity matrix, and obtain a matrix $\phi = EB + D$ that is either the identity or that has a sparse inverse, without reducing E, B, and D to simple expressions (degree 1 and/or identity matrices), as was done in previous work. We present two examples of this idea as well as the steps to follow in order to achieve a sparse ϕ^{-1}. For each example, we then consider the resulting BER performance.

A. Example 1: ϕ Matrix Orthogonal

As a first example of the idea, we modify Code 1′, shown in Fig. 6, to form Code 2. We aim to keep E and D unchanged and to modify only the last $4M$ rows of B; an alternative would be to modify E as well.

For notational convenience, circulant matrices can be represented as powers of the indeterminate x [1]. The identity matrix is denoted $x^0 = 1$, its first right circular shift is x^1, and so on, with $x^M = x^8 = x^0$. Using this notation,

$$E = \begin{bmatrix} 0 & 0 & 0 & 0 & x^7 & x^1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x^3 & x^6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & x^4 \\ 0 & 0 & 0 & 0 & x^5 & 0 & 0 & x^6 \end{bmatrix}$$

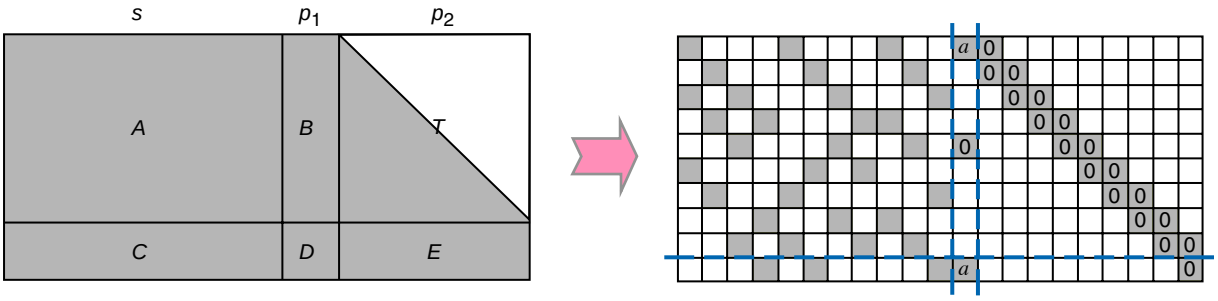

Fig. 7. *H* **matrix suitable for Richardson and Urbanke encoding.**

and

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For B, we keep the matrix dimensions and the first $4M$ rows the same as in the original matrix. In the last $4M$ rows, we keep the locations of the nonzero sub-matrices, but modify the particular choices of these circulants to achieve a sparse ϕ^{-1}. That is,

$$B_2 = \begin{bmatrix} 1 & 0 & 0 & x^1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ x^a & x^b & x^c & 0 \\ 0 & x^d & x^e & x^f \\ x^i & 0 & x^g & x^h \\ x^k & x^l & 0 & x^j \end{bmatrix}$$

where the unknowns a through l are to be determined. In this example, we aim to pick those unknowns so that ϕ has the form

$$\phi_2 = EB_2 + D = \begin{bmatrix} 0 & 0 & 0 & x^\delta \\ x^\alpha & 0 & 0 & 0 \\ 0 & x^\beta & 0 & 0 \\ 0 & 0 & x^\gamma & 0 \end{bmatrix}$$

where α, β, γ, and δ are also to be determined. Substituting,

$$EB_2 = \begin{bmatrix} x^{7+a} & x^{7+b} + x^{1+d} & x^{7+c} + x^{1+e} & x^{1+f} \\ x^{6+i} & x^{3+d} & x^{3+e} + x^{6+g} & x^{3+f} + x^{6+h} \\ x^i + x^{4+k} & x^{4+l} & x^g & x^h + x^{4+j} \\ x^{5+a} + x^{6+k} & x^{5+b} + x^{6+l} & x^{5+c} & x^{6+j} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x^\delta \\ x^\alpha & 1 & 0 & 0 \\ 0 & x^\beta & 1 & 0 \\ 0 & 0 & x^\gamma & 1 \end{bmatrix}$$

From the diagonal elements, we find $a = 1$, $d = 5$, $g = 0$, and $j = 2$. From these, the rest of the values are found to be $b = 7$, $c = 5$, $e = 3$, $f = 1$, $h = 6$, $i = 4$, $k = 0$, and $l = 6$.

With these choices, $\phi_2 = EB_2 + D$ is an orthogonal permutation matrix, so $\phi_2^{-1} = \phi_2^T$ and the hardware requirements needed for an encoder using ϕ_2^{-1} are greatly reduced. In Fig. 8, ϕ_2^{-1} is compared with ϕ^{-1}.

1. BER Performance. The problem with this example is that to obtain a main diagonal full of 1's we have created a dependency between B_2 and E in which many of the circulants of B_2 are transposes of circulants in E. This causes a large number of short cycles (length-4 cycles) between B_2, E, D, and T that lead to a great degradation of BER performance.

B. Example 2: ϕ Matrix Avoiding Length-4 Cycles

In this example, we add design constraints to eliminate length-4 cycles in H. This comes at the cost of an increased complexity in the resulting matrix ϕ, although it still possesses a sparse inverse. As in the previous example, we choose to modify only the choice of circulants in the bottom half of B.

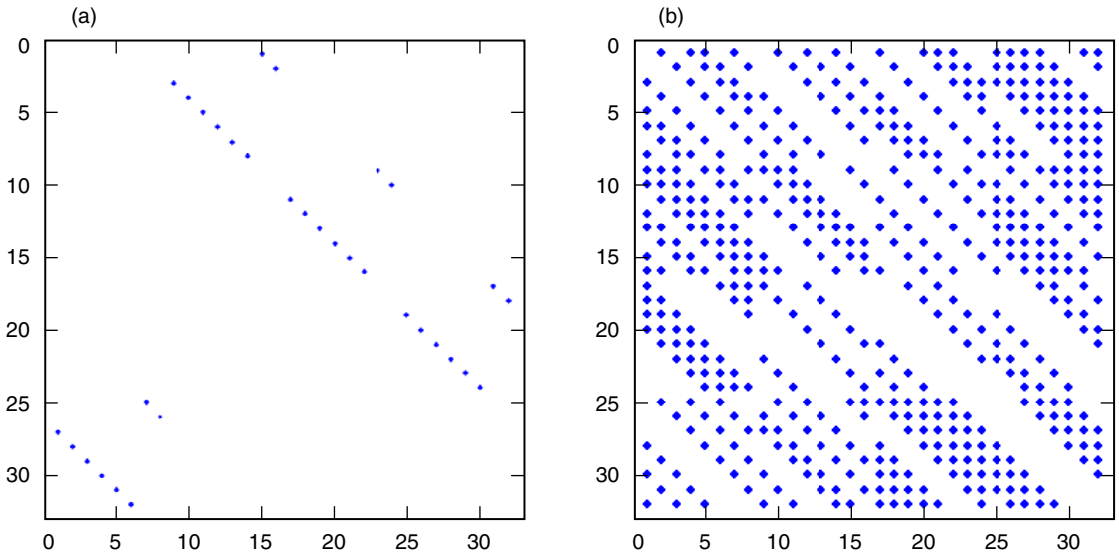

Fig. 8. Comparison of (a) ϕ_2^{-1} of the modified code and (b) ϕ^{-1} of Code 1.

In order to avoid length-4 cycles, we have to check where the modification of the circulants of B can cause these cycles. The design of the bottom half of B can introduce three categories of length-4 cycles, in the locations shown by rectangles X, Y and Z in Fig. 9. Length-4 cycles in category X are caused by relationships between the circulants in the bottom half of B, the lower-right quarter of T, the right half of E, and D. For analysis, we collect those four $4M \times 4M$ matrices into a new matrix, X:

$$X = \begin{bmatrix} x^a & x^b & x^c & 0 & 1 & 0 & 0 & 0 \\ 0 & x^d & x^e & x^f & 0 & 1 & 0 & 0 \\ x^i & 0 & x^g & x^h & 0 & 0 & 1 & 0 \\ x^k & x^l & 0 & x^j & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & x^m & x^n & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & x^o & x^p & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & x^q & x^r \\ 0 & 0 & 0 & 1 & x^t & 0 & 0 & x^s \end{bmatrix}$$

To avoid length-4 cycles in X, it turns out that there are constraints only on the circulants that lie on the main diagonal: $(a+m) \bmod M \neq 0$, $(d+o) \bmod M \neq 0$, $(g+q) \bmod M \neq 0$, and $(j+s) \bmod M \neq 0$.

Similarly, length-4 cycles in category Y are caused by relationships between the circulants in the bottom half of B, the lower-right quarter of A, the right half of C, and D. We collect those into the matrix Y:

$$Y = \begin{bmatrix} x^a & x^b & x^c & 0 & 1 & 0 & 0 & 0 \\ 0 & x^d & x^e & x^f & 0 & 1 & 0 & 0 \\ x^i & 0 & x^g & x^h & 0 & 0 & 1 & 0 \\ x^k & x^l & 0 & x^j & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & x^u & x^v \\ 0 & 1 & 0 & 0 & x^w & 0 & 0 & x^x \\ 0 & 0 & 1 & 0 & x^y & x^z & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & x^\psi & x^\omega & 0 \end{bmatrix}$$

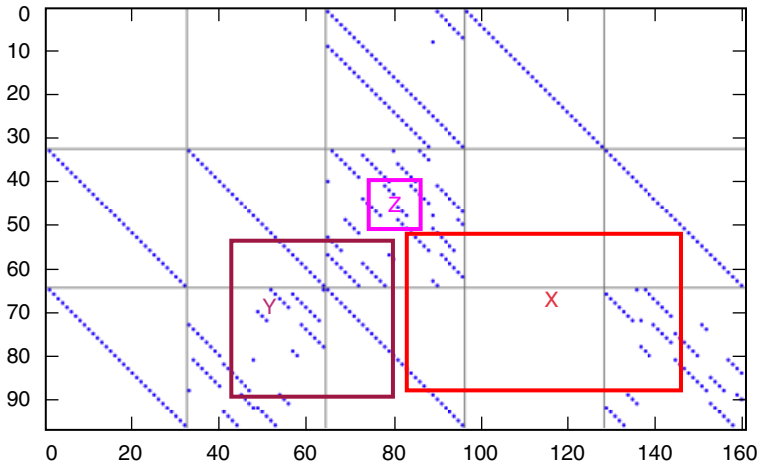

Fig. 9. Locations of length-4 cycles in the parity-check matrix for Code 2.

To avoid length-4 cycles in Y, there are eight restrictions on the circulants that may be chosen: $(b+w) \bmod M \neq 0$, $(c+y) \bmod M \neq 0$, and so on.

Length-four cycles in category Z are contained entirely within the bottom half of B, which we denote Z:

$$Z = \begin{bmatrix} x^a & x^b & x^c & 0 \\ 0 & x^d & x^e & x^f \\ x^i & 0 & x^g & x^h \\ x^k & x^l & 0 & x^j \end{bmatrix} \tag{6}$$

It is necessary that this matrix not be composed of circulants that form short cycles between them. This fact results in six restrictions: $(a-k+l-b) \bmod M \neq 0$, $(b-d+e-c) \bmod M \neq 0$, and so on.

In total, we have 18 constraints on the circulants that can be chosen for the bottom half of B in order to avoid adding length-4 cycles to H. Much more demandingly, we wish to choose those circulants so that $\phi = EB + D$ has a sparse inverse. In this example, we aim to choose circulants so that ϕ has the form

$$\phi_3 = EB_3 + D = \begin{bmatrix} P & 0 & Px^\eta & x^\alpha \\ x^\beta & P & 0 & Px^\theta \\ Px^\varepsilon & x^\gamma & P & 0 \\ 0 & Px^\zeta & x^\delta & P \end{bmatrix} \tag{7}$$

where $P = 1 + x^{M/2}$. This matrix has the sparse inverse

$$\phi_3^{-1} = \begin{bmatrix} Px^{\theta-\alpha-\beta} & x^{-\beta} & Px^{\beta-\gamma} & 0 \\ 0 & Px^{\varepsilon-\beta-\gamma} & x^{-\gamma} & Px^{\gamma-\delta} \\ Px^{\delta-\alpha} & 0 & Px^{\zeta-\gamma-\delta} & x^{-\delta} \\ x^{-\alpha} & Px^{\alpha-\beta} & 0 & Px^{\eta-\delta-\alpha} \end{bmatrix}$$

We start with the AR4JA code of size $(n = 2048, k = 1024)$ given in the CCSDS standard, for which $M = 128$, D_{AR4JA} is the identity matrix, and

$$E_{\mathrm{AR4JA}} = \begin{bmatrix} & x^m & x^n & 0 & 0 \\ 0_{512\times512} & 0 & x^o & x^p & 0 \\ & 0 & 0 & x^q & x^r \\ & x^t & 0 & 0 & x^s \end{bmatrix} = \begin{bmatrix} & x^{115} & x^{30} & 0 & 0 \\ 0_{512\times512} & 0 & x^{59} & x^{102} & 0 \\ & 0 & 0 & x^1 & x^{69} \\ & x^{94} & 0 & 0 & x^{99} \end{bmatrix}$$

Substituting these values of D and E into Eq. (7), we can solve to find a unique solution for the bottom half of B with the form given in Eq. (6). It is

$$Z_3 = \begin{bmatrix} x^{M/2-m} & x^{M/2-o+n-m} & x^{-q+p-o-m+n} & 0 \\ 0 & x^{M/2-o} & x^{M/2-q+p-o} & x^{-s+r=q-o+p} \\ x^{-m+t-s-q+r} & 0 & x^{M/2-q} & x^{M/2-s+r-q} \\ x^{M/2-m+t-s} & x^{-o+n-m-s+t} & 0 & x^{M/2-s} \end{bmatrix} = \begin{bmatrix} x^{77} & x^{48} & x^{85} & 0 \\ 0 & x^5 & x^{106} & x^{12} \\ x^{76} & 0 & x^{63} & x^{33} \\ x^{72} & x^{107} & 0 & x^{93} \end{bmatrix}$$

We make this modification to the sub-matrix B_{AR4JA} from the CCSDS code and call the result Code 3. In Fig. 10, $\phi^{-1}_{\mathrm{AR4JA}}$ and ϕ_3^{-1} are compared. Because ϕ_3^{-1} remains sparse, the encoding procedure is simplified in terms of memory consumption, hardware consumption, or both, depending on the architecture used to encode.

1. BER Performance. With this architecture, the BER performance of Code 3 is 0.5-dB worse than the original AR4JA code, but much better than Code 2. A further study of this code revealed that there were some codewords of low weight caused by information frames of weight 2 (the minimum codeword weight found was 28, far from the 78 that is believed to be the minimum weight of the original AR4JA code).

C. General Design Method

The previous example suffered a 0.5-dB performance loss compared to the original AR4JA code. This may be the result of leaving most of the parity-check matrix H unchanged and of restricting our

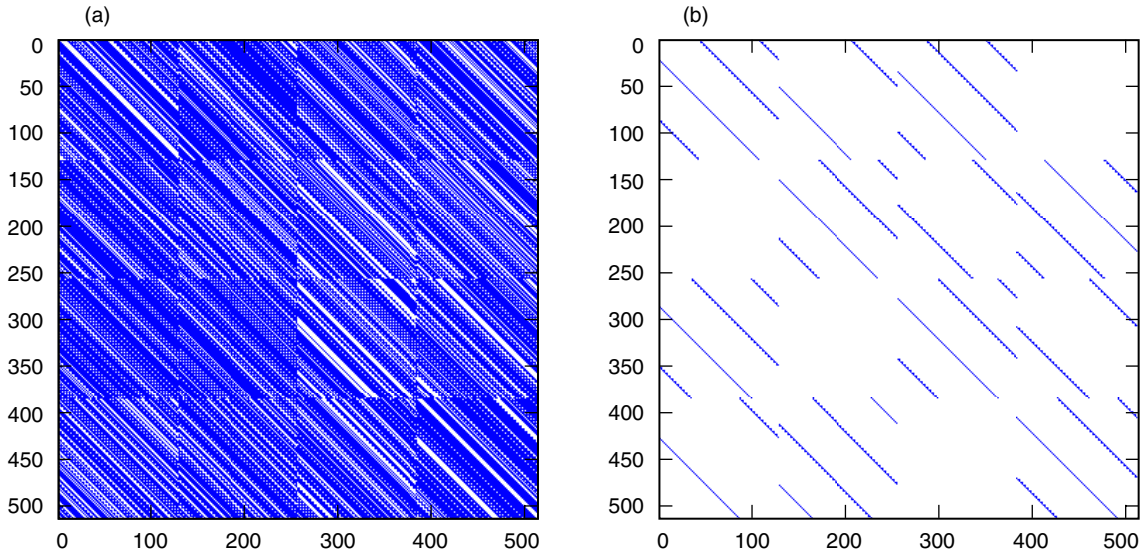

Fig. 10. Comparison of (a) $\phi^{-1}_{\mathrm{AR4JA}}$ and (b) ϕ_3^{-1}.

modifications to only a $4M \times 4M$ sub-matrix of H. In order to improve the BER performance, we suggest the following method for designing a complete parity-check matrix, while keeping the same protograph:

(1) Choose T to be the identity matrix.

(2) Choose B, E, and D so that $\phi^{-1} = (EB + D)^{-1}$ is sparse and easy to obtain from ϕ.

(3) Select the rest of the circulant matrices using the PEG algorithm [7] and the ACE metric [8] to minimize the presence of short cycles and stopping sets in H.

IV. Hardware Architectures

We now present a comparison between three different encoder architectures in terms of latency and the amount of circuitry required, measured as the number of flip-flops and AND and XOR logic gates and the amount of memory used. The first encoder performs a matrix multiplication by the dense generator matrix G, the second uses the algorithm developed in Section II, and the third is an encoder that takes advantage of the sparse ϕ^{-1} of the codes in Section III.

A circuit to multiply a dense matrix of circulants by a vector was developed in [5] and is shown in Fig. 11. This circuit consists of one recursive convolutional encoder per row of circulants. The boxes along the top of the figure are configured with the first column of the matrix (perhaps stored in a memory), and as the vector elements are shifted in serially, each is multiplied by this column, and the result is accumulated and circularly shifted. After the multiplication has proceeded through the first column of circulants, the boxes along the top are reconfigured for the second column of circulants, and the process is repeated until the entire matrix multiplication is complete. To multiply a matrix of size $J \times K$ by a vector of length K, this method requires J registers, J XOR gates, and J AND gates.

On the other hand, when multiplying a sparse matrix by a vector, the sparseness of the matrix can be exploited to reduce the logic required. Moreover, for a matrix that is composed of circulants, there is only one nonzero entry in each row or column of each sub-matrix. Therefore, the following method uses only one XOR gate for each one of the circulants.

As in the dense matrix multiplier described earlier, the sparse matrix multiplier is based upon a set of recirculating shift registers, one per row of circulants. Unlike the dense matrix multiplier, we use a single XOR gate per shift register, and connect it to the position necessary to achieve the desired circulant multiplication. For each successive circulant, the XOR gate may need to be connected to a different position in the shift register, and this list of positions may be tabulated in a memory.

An example of sparse matrix-by-vector multiplication is shown in Fig. 12, and the corresponding circuit is shown in Fig. 13. As the multiplication process proceeds column by column, the multiplexer below each

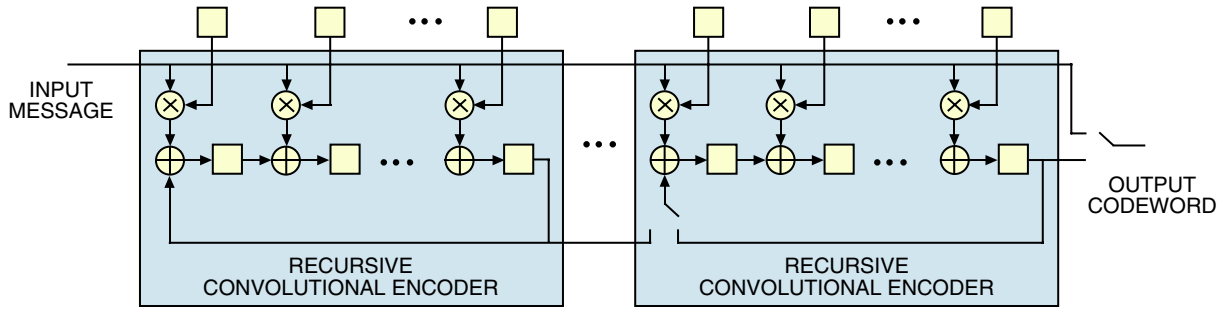

Fig. 11. Circuit to multiply a dense matrix of circulants by a vector.

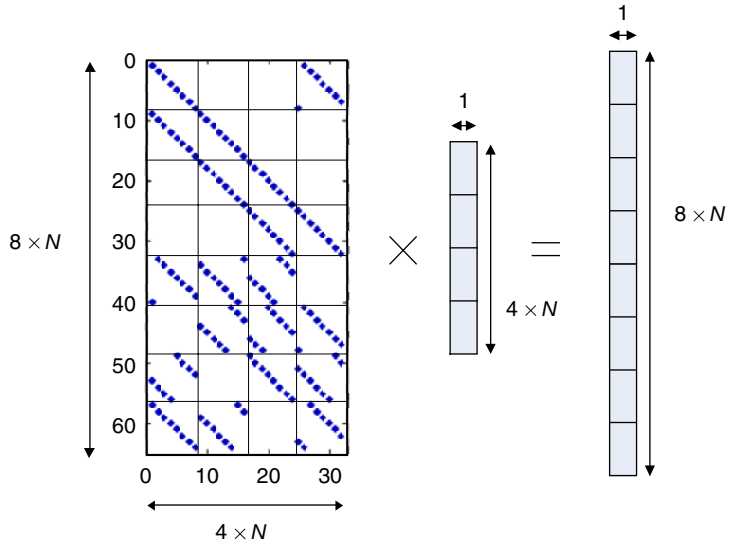

Fig. 12. Example showing the multiplication of a sparse matrix by a vector.

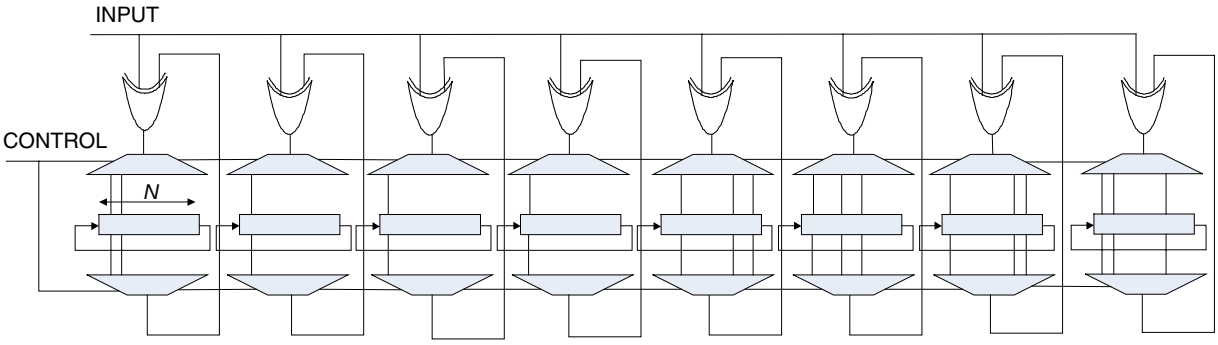

Fig. 13. Circuit for performing the sparse matrix multiplication in Fig. 12.

shift register, and the demultiplexer above it, are used to connect the XOR gate to the appropriate stage of the shift register. The sizes of the multiplexer and demultiplexer are determined by the number of distinct non-zero circulants in the matrix row that they implement. They are controlled from a memory in which the circulant offsets are tabulated.

The sparse matrix consists of macro-rows, and we call the number of distinct circulants in a macro-row its degree. The amount of memory required to control the multiplexers in this multiplier is equal to $btc \times mr \times dg$ bits, where mr is the number of macro-rows, dg is the maximum degree of the macro-rows, and btc is the number of bits required to select the proper circulant from the dg choices: $btc = \text{ceil}\big(\log_2(dg)\big)$. If the macro-rows have different degrees, the equation is adjusted accordingly.

Table 1 compares the complexities of three encoders. Each takes a vector of k information bits and computes $2m/3$ parity symbols for transmission, and perhaps $m/3$ additional punctured parity symbols if necessary as an intermediate result. The first encoder directly computes the $2m/3$ transmitted parity symbols by multiplying the vector of information bits by the dense $k \times (2m/3)$ generator matrix, as the $m/3$ punctured symbols do not have to be calculated. Using the multiplier of Fig. 11, this requires $2m/3$ flip-flops, $2m/3$ AND gates, and $2m/3$ XOR gates, as shown in the first row of Table 1. The multiplier also requires knowledge of the first row of each $M \times M$ circulant, thus requiring $2mk/3M$ bits of memory.

Table 1. Complexity comparison between three different encoder architectures.

Encoder architecture	Step	MUXs	Flip-flops	ANDs	XORs	Memory
1. Multiply by dense G		—	$\frac{2m}{3}$	$\frac{2m}{3}$	$\frac{2m}{3}$	$\frac{2mk}{M}$
2. Richardson–Urbanke	Step 1. $\left[\phi^{-1}(EA+C)\right] s^T$	—	$\frac{m}{3}$	$\frac{m}{3}$	$\frac{m}{3}$	$\frac{m}{3}\frac{k}{M}$
	Step 2. As^T	2×8	$\frac{2m}{3}$	—	8	h
	Step 3. Bp_1^T	2×8	$\frac{2m}{3}$	—	8	
	Total	32	$\frac{5}{3}m$	$\frac{m}{3}$	$\frac{m}{3}+16$	$h+\frac{m}{3}\frac{k}{M}$
3. Sparse ϕ^{-1}		40	$\frac{5}{3}m$	—	20	h

As described in Section II, the second encoder proceeds in three steps. First, with the precomputed dense matrix $\phi^{-1}(EA + C)$ of size $m/3 \times k$, it computes p_1 from the matrix multiplication in Eq. (4). Second, it performs the sparse matrix multiplication As^T, where A is of size $2m/3 \times 2m/3$. Third, it computes Bp_1^T, where B is a sparse matrix of size $2m/3 \times m/3$, and adds this to As^T to find p_2 according to Eq. (5). Each sparse multiplication requires eight XOR gates, eight multiplexers and eight demultiplexers (collectively MUXs), $2m/3$ flip-flops, and some small amount of memory, simply denoted by h in Table 1. The first and second steps could be performed simultaneously.

The codes developed in Section III are encoded in the same way as described in Section II, but they have a sparse ϕ^{-1}. This means that the encoder's first step can be performed as a sparse matrix multiplication rather than a dense one. The sparse multiplier requires four XOR gates, four multiplexers and demultiplexers, $m/3$ flip-flops (as in the dense case), and a small amount of memory. The second and third steps of the encoding process remain unchanged, giving the total in the last row of Table 1.

Several variations on the second and third encoders are possible, so there may be improvements possible to those encoders described here. For the third option, no dense matrices need be stored, which greatly reduces the memory consumption. This even allows the possibility of storing only H, and computing ϕ^{-1} in hardware, if H is required for some other application, such as a decoder in the same field programmable gate array.

V. Conclusions

An alternative way of encoding AR4JA codes proposed for CCSDS 131.1-O-2 [1] has been shown. No changes to the code are made, and the method allows a low-memory-consumption architecture, saving around 33 percent of the memory used by the encoder suggested in that document. It also reduces the number of gates used by increasing only slightly the number of registers used. This proposal is quite flexible and allows some other architectural alternatives.

Moreover, methods are proposed for constructing modified versions of the rate-1/2 AR4JA code with performance close to the original code, while allowing the use of encoders with far less memory consumption.

Acknowledgments

The authors would like to express their thanks to Jon Hamkins for his valuable comments during the realization of this work. Also, thanks go to Dariush Divsalar for the simulations that led to the improvements in the proposed method. Finally, special thanks go to Victor Fernandez and Pablo Sanchez for their invaluable contributions as advisors and their continuous checking.

References

[1] CCSDS 131.1-O-2, "Low Density Parity Check Codes for Use in Near-Earth and Deep Space Applications," The Consultative Committee for Space Data Systems, Orange book, Issue 2, September 2007.
http://public.ccsds.org/publications/archive/131x1o2.pdf

[2] T. J. Richardson and R. L. Urbanke, "Efficient Encoding of Low Density Parity Check Codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 638–656, 2001.

[3] "Digital Video Broadcasting (DVB) Second Generation Framing Structure for Broadband Satellite Applications," European Telecommunications Standards Institute (ETSI), EN 302 307 V1.1.1.

[4] "Draft IEEE Standard for Local Metropolitan Networks—Specific Requirements. Part 11: Wireless LAN Medium Access Control (MAC), and Physical Layer (PHY) Specifications: Enhancements for Higher Throughput," IEEE P802.11n/D10, March 2006.

[5] K. Andrews, S. Dolinar, and J. Thorpe, "Encoders for Block-Circulant LDPC Codes," *Proc. IEEE International Symposium on Information Theory*, Adelaide, Australia, pp. 2300–2304, 2005.

[6] S. Myung, K. Yang, and J. Kim, "Quasi-Cyclic LDPC Codes for Fast Encoding," *IEEE Trans. Inf. Theory*, vol. 51, no. 8, pp. 2894–2901, 2005.

[7] X.-Y. Hu, E. Eleftheriou, and D.-M. Arnold, "Progressive Edge-Growth Tanner Graphs," *IEEE Proc. Globecom'2001*, San Antonio, Texas, November 2001.

[8] T. Tian, C. Jones, J. Villasenor, and R. Wesel, "Selective Avoidance of Cycles in Irregular LDPC Code Construction," *IEEE Trans. Communications*, vol. 52, pp. 1242–1247, August 2004.