Block-LDPC: A Practical LDPC Coding System Design Approach

Hao Zhong, *Student Member, IEEE*, and Tong Zhang, *Member, IEEE*

***Abstract*— This paper presents a joint low-density parity-check (LDPC) code-encoder-decoder design approach, called Block-LDPC, for practical LDPC coding system implementations. The key idea is to construct LDPC codes subject to certain hardware-oriented constraints that ensure the effective encoder and decoder hardware implementations. We develop a set of hardware-oriented constraints, subject to which a semi-random approach is used to construct Block-LDPC codes with good error-correcting performance. Correspondingly, we develop an efficient encoding strategy and a pipelined partially parallel Block-LDPC encoder architecture, and a partially parallel Block-LDPC decoder architecture. We present the estimation of Block-LDPC coding system implementation key metrics including the throughput and hardware complexity for both encoder and decoder. The good error-correcting performance of Block-LDPC codes has been demonstrated through computer simulations. With the effective encoder/decoder design and good error-correcting performance, Block-LDPC provides a promising vehicle for real-life LDPC coding system implementations.**

***Index Terms*— LDPC, Encoder, Decoder, VLSI architecture.**

I. Introduction

LOW-DENSITY parity-check (LDPC) codes have recently attracted tremendous research interest because of their excellent error-correcting performance and highly parallel decoding scheme. LDPC codes have been lately selected by the DVB (digital video broadcasting) standard and are being seriously considered in various real-life applications such as magnetic storage, 10Gigabit Ethernet, and high-throughput wireless LAN (local area network). Invented by Gallager [1] in 1962, LDPC codes have been largely neglected by the scientific community for several decades until the remarkable success of Turbo codes that invoked the re-discovery of LDPC codes, pioneered by MacKay and Neal [2] and Wiberg [3]. The past a few years experienced significant improvement on LDPC code construction and performance analysis. For the practical LDPC coding system implementations, it has been well recognized that the conventional code-to-encoder/decoder design approach, i.e., first construct the code and then develop the encoder/decoder hardware implementations, is not appropriate and we must *jointly* consider the code construction and encoder/decoder hardware implementation. This is referred to as *joint LDPC coding system design*. Following the theme of joint design, we developed a design solution, called **Block-LDPC**, for practical LDPC coding system implementations.

Manuscript received April, 2004; revised July, 2004. This work was supported in part by SRC contract No. 2004-HJ-1192.

The authors are with the department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY.

An LDPC code is defined as the null space of an $M \times N$ sparse parity check matrix. It can be represented by a bipartite graph, between M check (or constraint) nodes in one set and N variable (or message) nodes in the other set. An LDPC code can be decoded by the message-passing decoding algorithm [4], [5] that directly matches the code bipartite graph as illustrated in Fig. 1: After variable nodes are initialized with the channel messages, the decoding messages are iteratively computed by all the variable nodes and check nodes and exchanged through the edges between the neighboring nodes.

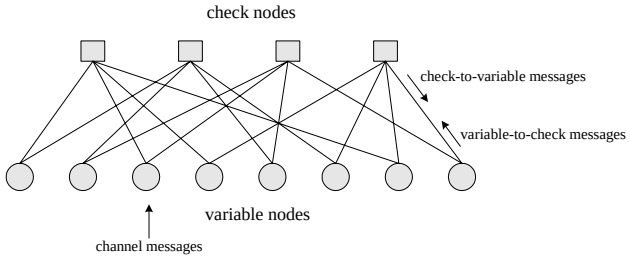

Fig. 1. Message-passing decoding based on LDPC code bipartite graph.

In the context of LDPC decoder hardware implementation, the challenge is how to realize the parallel message passing. There are two decoder implementation styles: (1) fully parallel decoder that realizes fully parallel message passing by directly instantiating the entire code bipartite graph into the hardware, and (2) partially parallel decoder that realizes partially parallel message passing by mapping a certain number of variable nodes or check nodes to a single hardware unit in time-division multiplexed mode. Fully parallel decoder can achieve very high decoding throughput, e.g., a 1 Gbps decoder for 1024-bit, rate 1/2 LDPC code has been physically demonstrated [6]. However, due to the typically large code length (at least few thousand bits) and widespread code bipartite graph connectivity, fully parallel decoder suffers from prohibitive implementation complexity, especially the routing overhead with a large number of global routing wires. This restricts the applications of fully parallel decoder to a very limited extent.

Aiming to achieve appropriate trade-off between implementation complexity and decoding throughput, partially parallel decoder is of practical interest to most real-life applications and becomes the target of most prior work on LDPC decoder hardware design. In this context, how to realize message passing is *more* challenging because the fully parallel bipartite graph connectivity has to be realized *part-by-part* in cooperation among a small-size interconnection network, a

reduced number of decoding hardware units, and a decoding message storage fabric. It has been well recognized that the code construction and partially parallel decoder hardware design must be considered jointly to facilitate the partially parallel LDPC decoder implementation, which is the basic principle underlying all the recent work on LDPC decoder design. Interestingly but not surprisingly, nearly all the recently proposed LDPC decoder design schemes [7]–[13] employ the essentially same joint design approach: The LDPC code parity check matrix is a *block structured* matrix that can be partitioned into an array of square block matrices, where each block matrix is either a zero matrix or a permuted identity matrix, even though the specific decoder architectures may largely differ among different proposed design solutions.

In the context of LDPC encoder design, the straightforward method is to multiply the information bits with the dense generator matrix derived from the sparse parity check matrix. The denseness of the generator matrix and typically large code length make this straightforward method impractical due to very high implementation complexity. Richardson and Urbanke [14] demonstrated that, if the parity check matrix is approximate lower (or upper) triangular, the encoding complexity can be largely reduced by performing the encoding directly based on the sparse parity check matrix. Most recently proposed low-complexity encoder hardware design schemes [15]–[17] essentially follow this idea.

The above summarized state-of-the-art LDPC encoder/decoder design practice shows that the essence of joint design is to apply certain *implementation-oriented* constraints on LDPC code construction to facilitate the LDPC encoder/decoder hardware implementations. A complete joint design solution should successfully address the following three interleaved questions:

1) What constraints should be used in LDPC code construction to facilitate the hardware implementation?
2) How to preserve the good error-correcting performance under those code construction constraints?
3) What are the appropriate encoder and decoder architectures?

The state-of-the-art LDPC coding system design solutions in the open literature mainly have two weaknesses: (1) most prior work addressed the above questions only in terms of decoder design and left encoder design unconsidered, and (2) most prior work did not address how to further optimize the code error-correcting performance under the corresponding implementation-oriented constraints. The authors of [8], [16], [17] considered both encoder and decoder design for regular LDPC codes that are typically worse than irregular ones in terms of error-correcting performance. Hocevar [10], [15] developed encoder/decoder design solution for irregular LDPC codes, but the specific code construction largely relies on hand-craft code template in lack of systematic construction approach. Moreover, the encoder design [15] process involves an off-line Gaussian elimination that will increase the denseness of the matrix based on which the encoding is performed, leading to higher encoding computational complexity.

The contribution of this paper is to provide a complete joint code-encoder-decoder design approach, called Block-LDPC, to address the above weaknesses. The Block-LDPC consists of three integral parts: (1) a semi-random implementation-oriented code construction approach, (2) a low-complexity encoding process and pipelined partially parallel encoder hardware architecture, and (3) a partially parallel decoder hardware architecture. The entire joint design methodology can be visualized as Fig. 2, where the appropriate LDPC code construction plays the essential role by determining both the error-correcting performance and feasibility/efficiency of the encoder/decoder hardware implementations.

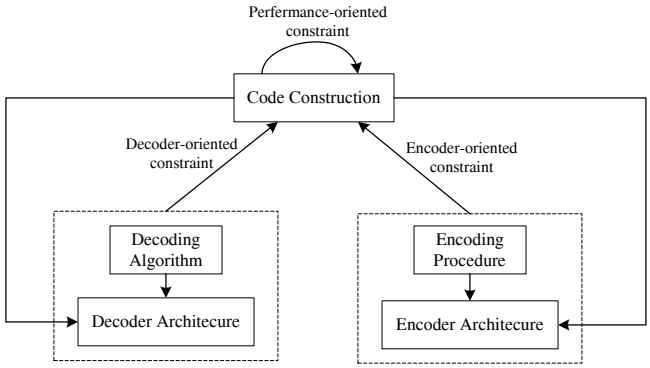

Fig. 2. Joint code-encoder-decoder design methodology.

Block-LDPC code is constructed subject to two types of constraints that ensure the effective encoder and decoder hardware implementations, respectively, and leave enough space for code error-correcting performance optimization. Our computer simulations show that Block-LDPC codes can achieve good error-correcting performance with little degradation compared with the codes constructed without any implementation-oriented constraints. We develop a low-complexity pipelined partially parallel Block-LDPC encoder architecture that can realize high encoding throughput with rather low implementation complexity. The partially parallel decoder architecture is relatively straightforward from the Block-LDPC code construction and the decoder architecture presented in this paper basically follows the one we presented in [11].

The remainder of this paper is organized as follows. Section II discusses the Block-LDPC code construction. In Section III, we present the efficient encoding strategy and its corresponding encoder architecture. Section IV addresses the Block-LDPC decoder design. Conclusions are drawn in Section V.

II. Block-LDPC Code Construction

A Block-LDPC code is constructed subject to two types of constraints, referred to as *implementation-oriented* constraints and *performance-oriented* constraints that ensure the efficient encoder/decoder hardware implementations and good error-correcting performance, respectively.

A. Implementation-oriented Constraints

The implementation-oriented code construction constraints consist of a *decoder-oriented* constraint and an *encoder-oriented* constraint that demand the code parity check matrix should have the structure as shown in Fig. 3.

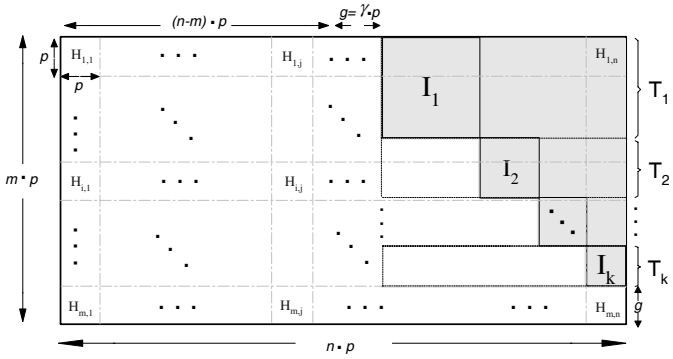

Fig. 3. The parity check matrix **H** subject to implementation-oriented constraints.

Decoder-oriented constraint forces the code parity check matrix to be block structured with circular block matrices, i.e., the entire parity check matrix can be partitioned into an array of $p \times p$ block matrices, each one denoted as $H_{i,j}$. Each block matrix $H_{i,j}$ is either a zero matrix or a right cyclic shift of an identity matrix. The value of p is an important parameter determining the decoding parallelism. Such block-structured constraint is the key in the development of several different partially parallel decoder architectures [7]–[12] with different trade-offs among decoding throughput, hardware complexity, and code structure re-configurability. As we will see in Section IV, the decoder architecture presented in this paper is more suitable for the applications demanding very high decoding throughput with minimal requirement on code structure re-configurability.

Encoder-oriented constraint forces the code parity check matrix to be lower *macro-block* triangular as shown in the shaded region of Fig. 3. The k identity matrices $\mathbf{I}_1$, $\mathbf{I}_2$, $\cdots$, $\mathbf{I}_k$ are called *macro-block identity matrices*. The size of each $\mathbf{I}_i$ is a multiple of p and decreases as the index i increases from 1 to k. The value of g is also a multiple of p ($g = \gamma \cdot p$ as shown in Fig. 3). Let $\mathbf{T}$ denote the $(m \cdot p - g) \times (m \cdot p - g)$ square matrix containing $\mathbf{I}_1$, $\mathbf{I}_2$, $\cdots$, $\mathbf{I}_k$, and $\mathbf{T}_i$ denote the sub-matrix of $\mathbf{T}$ that contains $\mathbf{I}_i$, as illustrated in Fig. 3. Moreover, the maximum column weight of each sub-matrix $\mathbf{T}_i$ is 1. The encoder-oriented constraint ensures a low-complexity pipelined partially parallel encoder that only performs a few sparse matrix-vector multiplications and one small dense matrix-vector multiplication. The detailed encoding process and encoder architecture design will be discussed in Section III.

B. Performance-Oriented Constraints

To achieve good error-correcting performance, LDPC codes should have the following properties: (a) *Large code length*: The performance improves as the code length increases, and the code length cannot be too small (typically at least few thousand bits); (b) *Carefully designed node degree distribution*: The node degree distribution, particularly variable node degree distribution, heavily affects the error-correcting performance. Design of appropriate node degree distribution is largely application dependent. LDPC codes with carefully designed irregular node degree distributions typically outperform regular ones[1]; (c) *Not too many small cycles*: Too many small cycles in the code bipartite graph will seriously degrade the effectiveness of the message-passing decoding algorithm, which will result in worse error-correcting performance; (d) *Widespread bipartite graph connectivity*: Any subset of the nodes in the LDPC code bipartite graph should have a large number of neighbors so that the messages generated by each node can distribute more quickly throughout the graph to improve the decoding performance.

The performance-oriented constraints in Block-LDPC code construction directly originate from the above last three rules of thumb (note that the code length is typically determined by the specific applications). The appropriate node degree distributions are obtained by a well-known standard design technique, i.e, density evolution [19]. To avoid too many small cycles, we explicitly set a constraint on the girth [2] of the code bipartite graph during the code construction in a similar way as the bit-filling approach [20].

Moreover, we note that the variable nodes with higher degree tend to converge more quickly than those with lower degree during the message-passing iterative decoding. This suggests that, with *finite number* of decoding iterations, not all the small cycles in the code bipartite graph are equally harmful, i.e., those small cycles passing low-degree variable nodes degrade the performance more seriously than the others. Thus, it is intuitive that we should put more effort to prevent small cycles from passing low-degree variable nodes. To this end, we introduce a concept, the degree of a cycle, which is similar to the concept of ACE proposed in [18]:

Definition 2.1: We define the *degree of a cycle* to be the sum of degrees of all variable nodes found along the path of a cycle.

It is intuitive that the error-correcting performance can be improved if we make the degree of the unavoidable small cycles as large as possible, which has been verified through our computer simulations.

The widespread graph connectivity is not explicitly used as a code construction constraint, but we implicitly ensure the widespread graph connectivity by incorporating randomness in the overall code construction that can guarantee the widespread connectivity almost for sure. Hence, based on the above discussion, we explicitly use the following three performance-oriented constraints in the code construction to ensure the good error-correcting performance:

1) *Node degree distribution constraint*: The code construction must comply with the desired node degree distribution.
2) *Girth constraint*: The code bipartite graph does not contain too many small cycles and is free of 4-cycle if possible.
3) *Degree of cycle constraint*: It prevents the variable nodes with lower degree from passing small cycles to further

[1]We note that some recent results suggest that irregular codes may be more seriously subject to error floor and how to deal with this issue has been discussed in [18].

[2]The girth is defined as the minimum cycle length in a graph.

preserve the effectiveness and fast convergence of the message-passing iterative decoding algorithm.

We note that, in practice, the typical values of small cycle length are 4 and 6, and the typical degree of low-degree variable nodes is 2.

C. Block-LDPC Code Construction

The overall Block-LDPC code construction process consists of three steps: (1) determine the code parity check matrix parameters, (2) construct a group of code parity check matrices, and (3) select one code from the code group for real application.

Step 1: We first determine the parameters of the code parity check matrix, including the size of code parity check matrix, the value of p (the size of each block matrix), node degree distribution, the value of g (or γ), the value of k (the number of macro-block identity matrices $\mathbf{I}_i$s), and the size of each $\mathbf{I}_i$. The size of the parity check matrix should be determined by the desired code length and code rate of the specific application, only subject to the constraint of being multiples of p, i.e., $m \cdot p \times n \cdot p$. The value of p is determined by the desired encoder/decoder throughput (as we will see in Sections III and IV, the encoder/decoder throughput directly depends on p). The node degree distribution is obtained by the density evolution. Strictly speaking, the selection of g involves the trade-off between encoding complexity reduction and code performance optimization space, i.e., the smaller the value of g, the less encoding complexity but the smaller space left for code performance optimization. Nevertheless, our computer simulations show that even the minimum value, i.e., $g = p$, seems to be enough for constructing Block-LDPC codes with good error-correcting performance. Hence we suggest to simply set $g = p$ in practice, which means the right-most block column[3] always has the weight of 2.

As we will see in Section III, the value of k affects the trade-off between encoder throughput and code performance optimization space, i.e., small value of k will lead to high encoder throughput but leave less space for code performance optimization. Our computer simulations show that $4 \sim 6$ should be the appropriate range for k. The size of each $\mathbf{I}_i$ should decrease as the index i increases from 1 to k in order to leave more code performance optimization space. Our experience is to consecutively scale down the size by 2 with the increase of i.

Step 2: We apply a *random block flipping/shifting* (RBFS) method to construct a group of LDPC code parity check matrices. The basic principle of RBFS is to randomly construct the code parity check matrix *block-by-block* (the size of each block matrix is $p \times p$) subject to the hardware-oriented constraints and performance-oriented constraints. RBFS starts from a nearly zero block structured $m \cdot p \times n \cdot p$ matrix in which the only non-zero portion is the k macro-block identity matrices $\mathbf{I}_1, \cdots, \mathbf{I}_k$ in the shaded region as illustrated in Fig. 3. RBFS proceeds by *flipping* a zero block matrix once a time in the ungrayed region to a right cyclic shift of an identity matrix while keeping the gray region untouched. The position of each flipped zero block matrix and the cyclic shift value are chosen *randomly* subject to the constraints on the degree of a cycle and girth. The number of flipped zero block matrices on each block column and block row is determined by the node degree distribution. The constraints on degree of a cycle and girth are initialized as reasonably large numbers such as 12 and 10. During the code construction process, the constraint on degree of a cycle or girth is relaxed (reduced) if the RBFS can not proceed with the current value after trying certain number of random choices. Repeating the RBFS process with different random number seeds, we can obtain a group of Block-LDPC codes.

Step 3: From the codes obtained in step 2, we select the code for real application based on a metric called *cycle effect* metric, which is also known as loopiness [21]. The cycle effect is defined as:

$$L = \sum_{i=6,8,\ldots} N_i \cdot \alpha^i,$$

where N_i is the number of cycles with the length of i and $\alpha < 1$ is a value chosen for the sum to converge. The code with smaller value of cycle effect L tends to have less small cycles and better error-correcting performance. Thus, we simply pick the code leading to the minimum value of L.

D. Code Examples

To demonstrate the error-correcting performance, we constructed two rate-1/2 and two rate-7/8 Block-LDPC codes. The specific code parameters are given in Table I. The sizes of the $k = 5$ macro-block identity matrices ($\mathbf{I}_1, \cdots, \mathbf{I}_5$) scale down approximately by factor of 2 as the index i increases from 1 to 5. The two codes with the rate of 1/2 have the node degree distribution as follows: (a) check nodes: degree of 6 $\Rightarrow$ 69%, degree of 7 $\Rightarrow$ 31%; (b) variable nodes: degree of 2 $\Rightarrow$ 27%, degree of 3 $\Rightarrow$ 45%, degree of 4 $\Rightarrow$ 14%, degree of 5 $\Rightarrow$ 14%. The two codes with the rate of 7/8 have the node degree distribution as follows: (a) check nodes: degree of 24 $\Rightarrow$ 100%; (b) variable nodes: degree of 2 $\Rightarrow$ 26%, degree of 3 $\Rightarrow$ 55%, degree of 4 $\Rightarrow$ 12%, degree of 5 $\Rightarrow$ 7%.

TABLE I
PARAMETERS OF EXAMPLE CODES

	rate	length	m	n	p	g	k	MDC	girth
code 1	1/2	4096	64	128	32	32	5	8	6
code 2	1/2	8192	64	128	64	64	5	8	6
code 3	7/8	4096	32	256	16	16	5	8	6
code 4	7/8	8192	32	256	32	32	5	8	6

[4]MDC: minimum degree of cycle

We simulate the code error-correcting performance with the assumption that each code is modulated by BPSK and transmitted over AWGN channel. Fig. 4 shows the simulated BER (bit error rate) vs. SNR (signal to noise ratio). For the purpose of comparison, we also constructed four LDPC codes without any implementation-oriented code construction

[3]We use *block column* and *block row* to represent each successive p columns and rows in the block structured parity check matrix.

constraints, i.e., setting $p = 1$ and $g = m \cdot p$ (eliminating the lower triangular part) and then using the same code construction process as described above. As shown in Fig. 4, the performance degradation incurred by the implementation-oriented constraints is not significant.

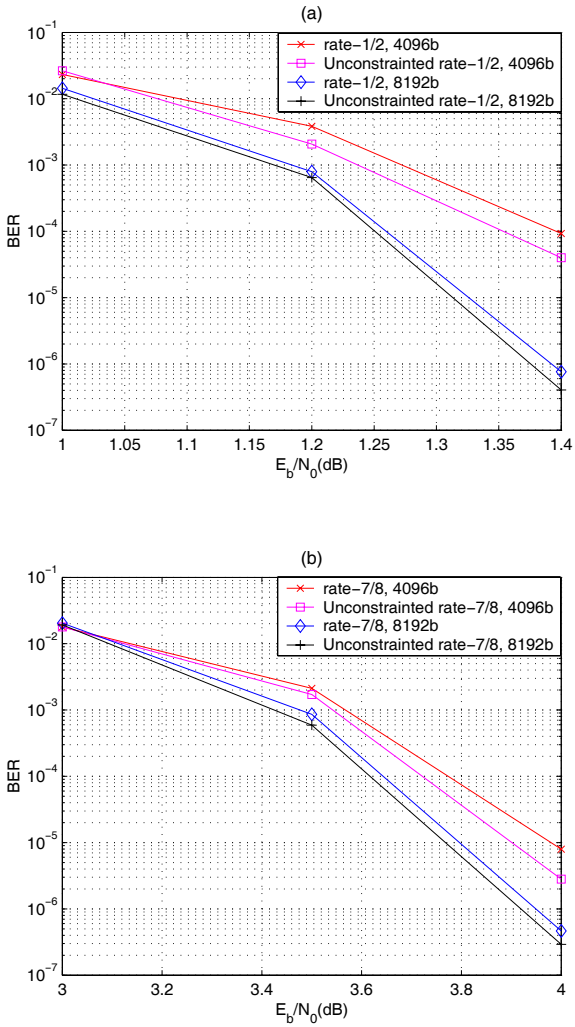

Fig. 4. BER vs. SNR simulation results.

III. Block-LDPC Encoder Design

Exploiting the structural properties of the Block-LDPC code parity check matrix, we developed a low-complexity pipelined partially parallel encoder hardware architecture. As we will show later, the *low complexity* comes from two aspects: (1) the encoding is performed based on the sparse parity check matrix and mainly involves a few sparse matrix-vector multiplications and a small dense matrix-vector multiplication, hence the overall computational complexity is not significant, and (2) encoding carries out in a partially parallel fashion via hardware resource sharing for further complexity reduction. In the following, we first describe the encoding process, then present a hardware architecture design for the sparse matrix-vector multiplication involved in the encoding, finally show the overall encoder architecture and the estimated implementation metrics.

A. Encoding Process

Following the idea of encoding based on approximate lower (or upper) triangular parity check matrix, Block-LDPC encoding process has the same data flow as the algorithm described in [14]. According to Fig. 3, we can write the Block-LDPC code parity check matrix[5] as

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{T} \\ \mathbf{C} & \mathbf{D} & \mathbf{E} \end{bmatrix}, \tag{1}$$

where $\mathbf{A}$ is $(m \cdot p - g) \times ((n-m) \cdot p)$, $\mathbf{B}$ is $(m \cdot p - g) \times g$, the lower triangular matrix $\mathbf{T}$ is $(m \cdot p - g) \times (m \cdot p - g)$, $\mathbf{C}$ is $g \times ((n-m) \cdot p)$, $\mathbf{D}$ is $g \times g$, and $\mathbf{E}$ is $g \times (m \cdot p - g)$. Let $[\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3]$ be a codeword decomposed according to (1), i.e, $\mathbf{z}_1$ is the information bit vector with the length of $(n-m) \cdot p$, redundant parity check bit vectors $\mathbf{z}_2$ and $\mathbf{z}_3$ have the length of g and $m \cdot p - g$, respectively. The encoding process performs the computation of the vectors $\mathbf{z}_2$ and $\mathbf{z}_3$ as follows [14]:

$$\begin{aligned} \mathbf{z}_2^T &= \mathbf{\Phi}^{-1} \cdot (\mathbf{E} \cdot (\mathbf{T}^{-1} \cdot (\mathbf{A} \cdot \mathbf{z}_1^T)) + \mathbf{C} \cdot \mathbf{z}_1^T), \\ \mathbf{z}_3^T &= \mathbf{T}^{-1} \cdot (\mathbf{A} \cdot \mathbf{z}_1^T + \mathbf{B} \cdot \mathbf{z}_2^T), \end{aligned}$$

where $\mathbf{\Phi} = \mathbf{E} \cdot \mathbf{T}^{-1} \cdot \mathbf{B} + \mathbf{D}$. In the entire encoding process, the multiplications with $\mathbf{T}^{-1}$ and $\mathbf{\Phi}^{-1}$ may lead to significant computational complexity overhead since they are most likely dense matrices. To reduce the computational complexity of the multiplication with $\mathbf{\Phi}^{-1}$, which is linearly proportional to g^2, we mainly rely on the minimization of g in the code construction (note that g is a multiple of the block size p, i.e., $g = \gamma \cdot p$). As we pointed out in Section II-C, our simulations show that $g = p$ (or $\gamma = 1$), is typically enough for constructing Block-LDPC codes with good error-correcting performance. Hence we can fix $g = p$ (p is typically a small number such as 16 and 32) to minimize the computational complexity of the multiplication with $\mathbf{\Phi}^{-1}$.

To reduce the computational complexity of the multiplication with $\mathbf{T}^{-1}$, we replace the direct multiplication with a k-*stage back substitution*. Recall that the lower triangular matrix $\mathbf{T}$ contains k macro-block identity matrices along the diagonal as illustrated in Fig. 3 and can be written as:

$$\mathbf{T} = \begin{bmatrix} \mathbf{I}_1 & \mathbf{O} & \cdots & \mathbf{O} \\ \mathbf{T}_{2,1} & \mathbf{I}_2 & \cdots & \mathbf{O} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{T}_{k,1} & \mathbf{T}_{k,2} & \cdots & \mathbf{I}_k \end{bmatrix}, \tag{2}$$

where $\mathbf{I}_1, \mathbf{I}_2, \cdots, \mathbf{I}_k$ are the k macro-block identity matrices; each $\mathbf{T}_{i,j}$ is a block structured matrix composed of an array of $p \times p$ zero and right cyclic shifted identity matrices; $\mathbf{O}$ represents zero matrix. Given the matrix $\mathbf{T}$ and input vector $[\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k]^T$, instead of directly computing the output vector as

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_k \end{bmatrix} = \begin{bmatrix} \mathbf{I}_1 & \mathbf{O} & \cdots & \mathbf{O} \\ \mathbf{T}_{2,1} & \mathbf{I}_2 & \cdots & \mathbf{O} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{T}_{k,1} & \mathbf{T}_{k,2} & \cdots & \mathbf{I}_k \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_k \end{bmatrix}, \tag{3}$$

[5]We assume that the parity check matrix is full rank, i.e., the $m \cdot p$ rows are linearly independent. In our simulations, all the matrices constructed are full rank.

where $\mathbf{x}_i$ and $\mathbf{y}_i$ ($i = 1, 2, \cdots, k$) are p-bit sub-vectors, we can solve each $\mathbf{y}_i$ consecutively by using k-stage back substitution as

$$\mathbf{y}_i = \mathbf{T}_{i,1}\mathbf{y}_1 + \mathbf{T}_{i,2}\mathbf{y}_2 + \cdots + \mathbf{T}_{i,i-1}\mathbf{y}_{i-1} + \mathbf{x}_i. \quad (4)$$

Hence the multiplication with $\mathbf{T}^{-1}$ is replaced by a series of sparse matrix-vector multiplications and vector additions, leading to significant reduction of the computational complexity.

B. Block Structured Matrix-Vector Multiplication

From the above discussion, we know that the overall encoding process mainly consists of a certain number of sparse matrix-vector multiplications and one small dense matrix-vector multiplication. The complexity and speed metrics of the encoder is largely determined by how to implement these sparse matrix-vector multiplications. We note that each sparse matrix involved in the multiplication is block structured and the corresponding matrix-vector multiplications can be written as:

$$\begin{bmatrix} \mathbf{U}_{1,1} & \mathbf{U}_{1,2} & \dots & \mathbf{U}_{1,s} \\ \mathbf{U}_{2,1} & \mathbf{U}_{2,2} & \dots & \mathbf{U}_{1,s} \\ \vdots & \vdots & \dots & \vdots \\ \mathbf{U}_{t,1} & \mathbf{U}_{t,2} & \dots & \mathbf{U}_{t,s} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_s \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_t \end{bmatrix}, \quad (5)$$

where each $p \times p$ block matrix $\mathbf{U}_{i,j}$ is either a zero matrix or a right cyclic shift of an identity matrix, and each $\mathbf{x}_j$ and $\mathbf{y}_i$ are $p \times 1$ sub-vectors.

We can leverage the circular structure of all the non-zero block matrices to develop efficient matrix-vector multiplication hardware architecture. Define a set $\mathcal{P} = \{(i,j) | \forall\ \mathbf{U}_{i,j}$ is non-zero.$\}$. Since each non-zero $\mathbf{U}_{i,j}$ is a right cyclic shift of an identity matrix, we have $\mathbf{y}_i = \sum_{(i,j)\in\mathcal{P}} \mathbf{x}_j[\uparrow d_{i,j}]$, where $d_{i,j}$ is the right cyclic shift value of $\mathbf{U}_{i,j}$ and $\mathbf{x}_j[\uparrow d_{i,j}]$ represents cyclic shifting up the sub-vector $\mathbf{x}_j$ by $d_{i,j}$ positions. Thus the matrix-vector multiplication reduces to a set of vector XORs (modulo 2 summations) $\{\mathbf{y}_i = \sum_{(i,j)\in\mathcal{P}} \mathbf{x}_j[\uparrow d_{i,j}],\ 1 \le i \le t\}$. Although the direct parallel implementation of theses vector XORs can easily achieve extremely high speed, such high speed is not necessary for most applications, and appropriate trade-off between implementation complexity and speed in encoder design is desirable.

Such a trade-off can be realized by using an *inter-vector-parallel/intra-vector-serial* computational style: compute all the t sub-vectors $\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_t$ in parallel, but only 1 out of the p bits in each sub-vector is computed at once. Since the value of p is typically small, e.g., 16 and 32, the overall matrix-vector multiplication can still achieve high speed. Clearly, because of the inter-vector-serial operation, the computations of all the p bits in the same sub-vector $\mathbf{y}_i$ can share the same hardware resource in a time-division multiplexed mode for implementation complexity reduction.

To compute $\mathbf{y}_i = \sum_{(i,j)\in\mathcal{P}} \mathbf{x}_j[\uparrow d_{i,j}]$ bit by bit consecutively, we only need to retrieve each involved sub-vector $\mathbf{x}_j$ bit by bit consecutively. Hence, it can be easily conceived that, in the hardware implementation, we can store each sub-vector $\mathbf{x}_j$ in an individual register file and use a binary counter to generate the access address, where the binary counter is initialized to the value of $d_{i,j}$. However, since each $\mathbf{x}_j$ may participate in the computations of more than one $\mathbf{y}_i$'s and the corresponding $d_{i,j}$'s have different values, in order to support the parallel computation of all $\mathbf{y}_i$'s, we have to use either multi-port register file or several single-port register file blocks for the storage of each $\mathbf{x}_j$, both of which will increase the implementation complexity. In the following, we will present a method to further trade the speed for the storage complexity reduction and present the corresponding matrix-vector multiplication hardware architecture.

1) Storage Complexity Reduction Via Task Scheduling: With the goal of reducing implementation complexity, we store each input sub-vector $\mathbf{x}_j$ only in one p-bit single-port register file associated with one binary counter for address generation. Hence each input sub-vector at most can participate in the computation of one output sub-vector at once. Nevertheless, since the weight of each block column in the block structured matrix might be larger than one, each input sub-vector may participate in the computation of more than one output sub-vectors. Therefore, instead of computing all the t output sub-vectors in fully parallel, we have to *schedule* the computations in a *partially parallel* fashion to ensure each input sub-vector at once participate in the computations of at most one output sub-vector, which can be interpreted as the following task scheduling problem:

Task Scheduling Problem: *Schedule the computations of all the t output sub-vectors into l time slots subject to two constraints: (1) If the computations of two output sub-vectors need the same input sub-vector, then we must compute these two output sub-vectors in different time slots; (2) The value of l should be minimized in order to maximize the speed.*

To solve the above scheduling problem, we first represent the block structured matrix $\mathbf{U}$ with a graph G as follows: (1) represent each block row as a node in the graph, and (2) if two block rows have non-zero block matrices in the same block column position, then connect the two corresponding nodes with an edge. Clearly, two nodes connected with an edge in the graph G indicate that the computations of the two output sub-vectors corresponding to the two block rows share the same input sub-vector, and hence cannot be performed in the same time slot. If we use different color to represent different time slot, the above task scheduling problem can be directly transformed into the following classic graph coloring problem:

Coloring Problem: *Color the nodes in the graph G with the minimum number of colors such that adjacent nodes have different colors.*

There are many well established algorithms that can effectively solve the above coloring problem. Interested readers are referred to [22]. Given the solution to this coloring problem, we simply schedule the computations of the output sub-vectors corresponding to the block rows with the same color in the same time slot. In this way, the implementation complexity can be reduced by storing each input sub-vector in one p-bit single-port register file.

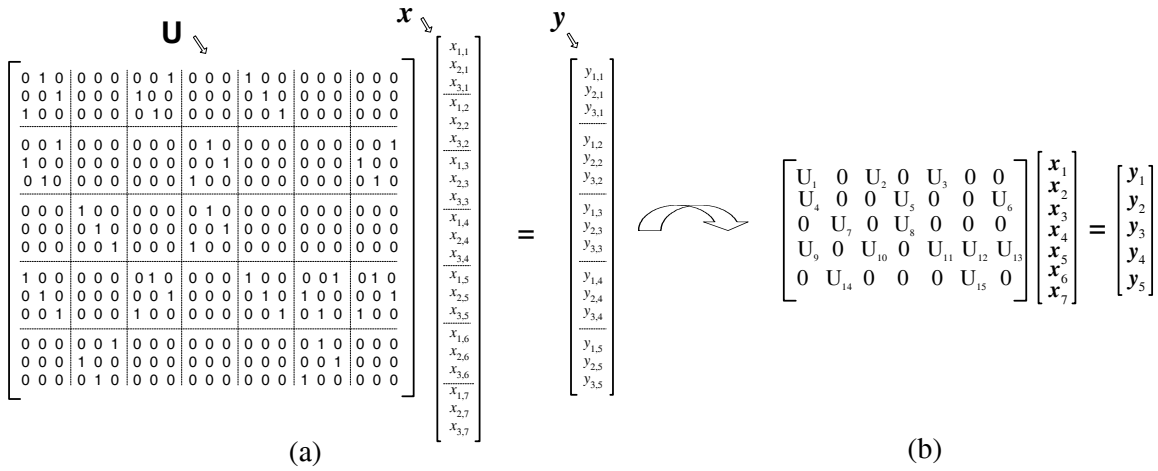

Fig. 5. Block structured matrix-vector multiplication example.

Example 3.1: Let's consider the multiplication between the block structured matrix $\mathbf{U}$ and vector $\mathbf{x}$ as illustrated in Fig. 5, where all the fifteen non-zero block matrices, $\mathbf{U}_1, \mathbf{U}_2, \cdots, \mathbf{U}_{15}$, are right cyclic shift of an identity matrix. The block structured matrix $\mathbf{U}$ is represented by a graph for which three different colors are enough to solve the coloring problem, as illustrated in Fig. 6. Thus, we compute the five output sub-vectors in three time slots: in the first time slot, we compute $\mathbf{y}_1$ and $\mathbf{y}_3$ bit by bit, which involves five input sub-vectors $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_5$. Similarly, we compute $\{\mathbf{y}_2, \mathbf{y}_5\}$ and $\{\mathbf{y}_4\}$ in the second and third time slot, respectively.

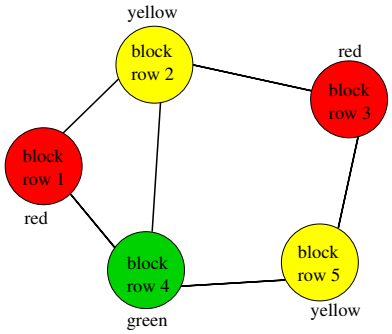

Fig. 6. The graph with the solution to the coloring problem.

2) Matrix-Vector Multiplication Hardware Architecture: Fig. 7 shows the proposed hardware architecture to implement the sparse block structured matrix-vector multiplication in the partially parallel style as described in the above. Each input sub-vector $\mathbf{x}_j$ and output sub-vector $\mathbf{y}_i$ are stored in register files X_j and Y_i, respectively. The read address of each register file X_j is generated by a binary counter RAG_j. The write address of all the register files Y_i's is generated by a binary counter WAG. The output of X_j is routed through the hardwired interconnection network to the input of the XOR tree XT_i if $\mathbf{x}_j$ participates in the computation of $\mathbf{y}_i$.

Suppose the size of each block matrix is $p \times p$ and l different colors are used in the task scheduling, we arrange all the t output sub-vectors into l groups, and the sub-vectors in the same group are computed bit-by-bit in the same p clock cycles.

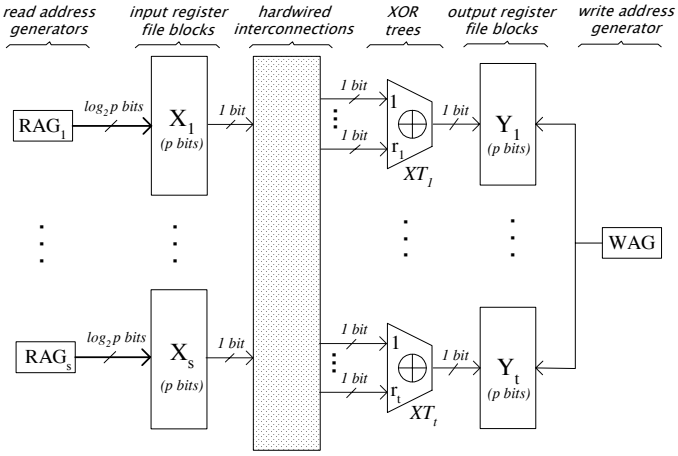

Fig. 7. Hardware architecture for block structured sparse matrix-vector multiplication.

The entire matrix-vector multiplication requires $l \cdot p$ clock cycles. At the beginning of each p clock cycles, we have:

- If sub-vector $\mathbf{x}_j$ participates in the computation of sub-vector $\mathbf{y}_i$ in the following p clock cycles, the binary counter RAG_j is initialized to the cyclic shift value $d_{i,j}$;
- If sub-vector $\mathbf{x}_j$ does not participate in any computation in the following p clock cycles, the register file X_j will be disabled;
- The binary counter WAG is always initialized to 0;
- The register file Y_i will be disabled if sub-vector $\mathbf{y}_i$ is not computed in the following p clock cycles.

C. Overall Encoder Structure

Fig. 8 shows the structure of the pipelined partially parallel Block-LDPC encoder that mainly contains 7 function blocks: The four function blocks A, B, C, and E realize the multiplications with the block structured sparse matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{E}$, respectively. These four function blocks are directly designed as described in section III-B. The function block Φ realizes the multiplication with the small dense matrix

TABLE II

THE NUMBER OF REGISTERS REQUIRED IN THE ENCODER.

	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
Block A output	$2(m \cdot p - g)$	$2(m \cdot p - g)$	$2(m \cdot p - g)$	$2(m \cdot p - g)$	$2(m \cdot p - g)$
Block B output	-	-	-	-	$2(m \cdot p - g)$
Block C output	$2g$	$2g$	$2g$	-	-
Block E output	-	-	$2g$	-	-
Block Φ output	-	-	-	$2g$	$2g$
Block T1 output	-	$2(m \cdot p - g)$	-	-	-
Total	$2m \cdot p$	$4m \cdot p - 2g$	$2m \cdot p + 2g$	$2m \cdot p$	$4m \cdot p - 2g$

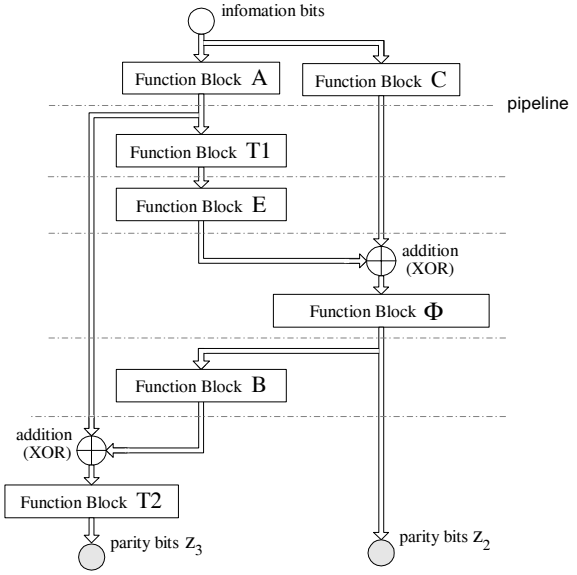

Fig. 8. Pipelined Block-LDPC encoder structure.

$\mathbf{\Phi}^{-1}$. The two identical function blocks T1 and T2 realize the multiplication with $\mathbf{T}^{-1}$ by using the k-stage back substitution as described in section III-A. Each dashed horizontal line in Fig. 8 represents one pipeline stage. Fig. 9 shows the structure of function blocks T1 and T2, where each sub-block $\mathrm{T}_{i,j}$ performs the multiplication with the sub-matrix $\mathbf{T}_{i,j}$ in the lower triangular matrix $\mathbf{T}$. Since all the sub-matrices $\mathbf{T}_{i,j}$ are also block structured, we can again use the architecture described in section III-B for each multiplication.

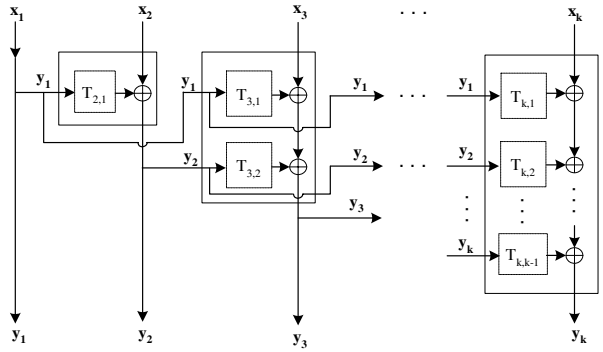

Fig. 9. Structure of function blocks T1 and T2.

Pipelining is realized by the input/output register file banks in each sparse matrix-vector multiplication block as illustrated in Fig. 7. Each sparse matrix-vector multiplication block requires two sets of register files to store the input vector, i.e., the two sets of input register files alternatively receive the output from the preceding pipeline stage and provide the data for the current computation. The register file complexity in terms of number of bits needed for each pipeline stage are listed in the Table II.

Let l_{max} denote the maximum number of colors used in the task scheduling in all the sparse matrix-vector multiplications. Except the pipeline stages for function blocks Φ, T1 and T2, any other pipeline stage at most takes $l_{max} \cdot p$ clock cycles to complete the present computation. Because of the small size (i.e., $p \times p$) of the dense matrix $\mathbf{\Phi}^{-1}$, it is feasible to implement the function block Φ (multiplication with $\mathbf{\Phi}^{-1}$) in fully parallel, i.e., implemented as a p-bit input p-bit output XOR array. The latency of such fully parallel implementation of function block Φ will be much less than $l_{max} \cdot p$ clock cycles. As for the function block T1 and T2, since the maximum collum weight of each sub-matrix $T_{i,j}$ is 1 (according to the encoder-oriented constraint as described in section II-C), each of these sub-matrix and vector multiplications can be performed in 1 time slot, i.e. p clock cycles. Thus each of T1 and T2 requires $(k-1) \cdot p$ clock cycles to complete the present computation. Therefore, the pipeline stage latency of this pipelined encoder is $\max(l_{max}, k-1) \cdot p$ clock cycles, i.e., each pipeline stage takes $\max(l_{max}, k-1) \cdot p$ clock cycles to complete the present computation and moves the output to the next pipeline stage. Let f_E denote the clock frequency of the encoder, the encoding throughout would be $(n-m) \cdot p \cdot f_E / (p \cdot \max(l_{max}, k-1)) = (n-m) \cdot f_E / \max(l_{max}, k-1)$.

To estimate the encoder logic gate complexity in terms of the number of 2-input NAND gates, we count each 2-input XOR gate as three 2-input NAND gates and each z-bit binary counter as $8z$ 2-input NAND gates. Let $|\mathcal{P}|$ denote the total number of nonzero blocks in the parity check matrix. Let θ denote the ratio of the number of nonzero blocks in the lower triangular sub-matrix $\mathbf{T}$ divided by $|\mathcal{P}|$. For the function block A, B, C, E, T1 and T2, we need approximately $(1+\theta) \cdot (|\mathcal{P}| - m)$ XOR gates and $(1+\theta) \cdot |\mathcal{P}|$ counters in total. Thus the number of NAND gates is about $3(1+\theta) \cdot (|\mathcal{P}| - m) + 8\lceil \log_2 p \rceil \cdot (1+\theta) \cdot |\mathcal{P}| \approx (1+\theta) \cdot |\mathcal{P}| \cdot (3 + 8\lceil \log_2 p \rceil)$. We assume the function block Φ consumes $\frac{1}{6}g^2$ XOR gates, i.e. $\frac{1}{2}g^2$ NAND gates. Furthermore, we need approximately $|\mathcal{P}| \cdot \lceil \log_2 p \rceil$ bits of ROM to store the initialization values for all the counters. We summarize the estimation of the key metrics of the encoder in the Table III.

TABLE III

THE SPEED AND COMPLEXITY ESTIMATION OF THE ENCODER

User Data Rate	ROM (bits)	Register Storage (bits)	Number of Logic Gates
$\frac{(n-m)\cdot f_E}{\max(l_{max},k-1)}$	$\lvert\mathcal{P}\rvert \cdot \lceil \log_2 p \rceil$	$14m \cdot p - 2g$	$(1+\theta)\cdot\lvert\mathcal{P}\rvert\cdot(3+8\lceil\log_2 p\rceil)+\frac{1}{2}g^2$

IV. LDPC DECODER DESIGN

In this section, we briefly present the partially parallel Block-LDPC code decoder architecture and its implementation metrics estimation. Fig. 10 shows the decoder architecture for a Block-LDPC code with an $m \cdot p \times n \cdot p$ parity check matrix. It contains m check node computation units (CNUs) and n variable node computation units (VNUs), where each CNU_i and VNU_j perform the computations, respectively, for the p rows (or check nodes) and p columns (or variable nodes) in the same block row and block column in time-division multiplexing fashion. It contains $2\cdot n$ channel message memory blocks (CMMBs), each CMMB stores the messages associated with p columns in the same block column. Two sets of n CMMBs alternatively store the channel messages for current decoding and receive the channel messages of the next block to be decoded.

Let $\mathbf{H}_{i,j}$ denote the block matrix with the position (i,j) in the Block-LDPC code parity check matrix, and set $\mathcal{P} = \{(i,j)|\forall\ \mathbf{H}_{i,j}$ is non-zero.$\}$. The total number of non-zero block matrices in the parity check matrix is $|\mathcal{P}|$. The $|\mathcal{P}| \times p$ decoding messages are stored in $|\mathcal{P}|$ decoding message memory blocks (DMMBs), each $\text{DMMB}_{i,j}$ (where $(i,j) \in \mathcal{P}$) stores the p messages associated with the p 1's in $\mathbf{H}_{i,j}$. The decoder contains n p-bit hard decision memory blocks (HDMBs) to store the hard decision bits. The access address of each CMMB_j, HDMB_j or $\text{DMMB}_{i,j}$ is generated by an individual binary counter. Each CNU_i connects with all the $\text{DMMB}_{i,j}$s with the same index i. Each VNU_j connects with all the the CMMB_j, HDMB_j and $\text{DMMB}_{i,j}$ with the same index j.

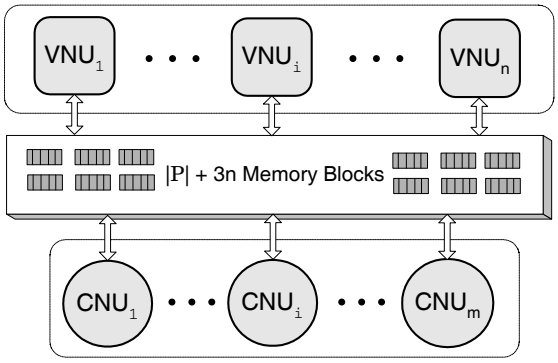

Fig. 10. Decoder architecture.

The decoding process consists of an initialization phase and an iterative decoding phase as described in the following.

Initialization: Upon the received code block data, CMMBs are initialized to store the channel messages associated with all the variable node. The content of each CMMB_j is copied to all the $\text{DMMB}_{i,j}$s with the same index j as the initial variable-to-check messages.

Iterative Decoding: Each decoding iteration is completed in $2p$ clock cycles, and the decoder works in check node processing mode and variable node processing mode during the 1st and 2nd p clock cycles, respectively.

(1) During the **check node processing**, the decoder performs the computations of all the check nodes and realizes the message passing between neighboring nodes in the code bipartite graph. In each clock cycle, each CNU retrieves one variable-to-check message from each connected DMMB, convert it to one check-to-variable message, and send the check-to-variable message back to the same DMMB. The memory access address of each $\text{DMMB}_{i,j}$ is generated by a binary counter that is initialized to the right cyclic shift value $d_{i,j}$ of the non-zero block matrix $\mathbf{H}_{i,j}$ at the beginning of check node processing.

(2) During the **variable node processing**, the decoder performs the computations of all the variable nodes. In each clock cycle, each VNU retrieves one check-to-variable message from each connected DMMB and one channel message from the connected CMMB, convert each check-to-variable message to variable-to-check message, and send it back to the same DMMB. The memory access addresses of each memory block are generated by the counters that are set to zero at the beginning of variable node processing.

The number of node computation units (CNU and VNU) in this partially parallel decoder is reduced by the factor p compared with its fully parallel counterpart. This partially parallel decoder is well suited for high speed hardware implementations because of the regular structure and simple control logic.

Given each decoding message uniformly quantized to q bits[6], we estimate that each CNU and VNU require $320\cdot q$ and $250\cdot q$ gates (in terms of 2-input NAND gate), respectively. The total sizes of DMMBs, CMMBs, and HDMBs are $|\mathcal{P}|\cdot p\cdot q$ bits, $2n\cdot p\cdot q$, and $n\cdot p$ bits, respectively. Let f_D denote the clock frequency of the decoder and the number of iterations is D, the decoding throughput can be up to $\frac{(n-m)\cdot f_D}{2D}$. We summarize the estimated key metrics of the decoder in Table IV.

TABLE IV

THE SPEED AND COMPLEXITY ESTIMATION OF THE DECODER.

User Data Rate	Memory (bits)	# of Gates
$\frac{(n-m)\cdot f_D}{2D}$	$(2n+\lvert\mathcal{P}\rvert)\cdot p\cdot q + n\cdot p$	$(320m+250n)\cdot q$

[6]We note that recent results suggest that uniform quantization with too small value of q may lead to error floor in the decoding. Interested readers are referred to [23].

V. Conclusions

This paper presented a joint code-encoder-decoder design solution, called Block-LDPC, for practical LDPC coding system implementations. The key is to construct LDPC codes subject to certain implementation-oriented constraints and performance-oriented constraints, simultaneously. We presented the code construction constraints and developed a semi-random approach for Block-LDPC code construction. Computer simulation showed that the Block-LDPC codes have insignificant error-correcting performance degradation compared with LDPC codes constructed without any implementation-oriented constraints. Leveraging the implementation-oriented constraints, we developed a pipelined partially parallel Block-LDPC code encoder and a partially parallel Block-LDPC code decoder. We believe the Block-LDPC design approach will provide communication system designers an unique opportunity to explore the attractive merits of LDPC codes in many real-life applications.

References

[1] R. G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[2] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, pp. 1645–1646, Aug. 1996.

[3] N. Wiberg, "Codes and decoding on general graphs," Ph.D. Dissertation, Linkoping University, Sweden, 1996. available at http://www.essrl.wustl.edu/~jao/itrg2000/.

[4] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, pp. 399–431, Mar. 1999.

[5] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, pp. 498–519, Feb. 2001.

[6] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, March 2002.

[7] J. Thorpe, "Low-density parity-check (LDPC) codes constructed from protographs," in *IPN Progress Report*, http://ipnpr.jpl.nasa.gov/tmo/progress_report/42-154/154C.pdf, Aug. 2003, pp. 42–154.

[8] S. Olcer, "Decoder architecture for array-code-based LDPC codes," in *Global Telecommunications Conference*, Dec. 2003, pp. 2046 – 2050.

[9] M. M. Mansour and N. R. Shanbhag, "Architecture-aware low-density parity-check codes," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, Bangkok, Thailand, May 2003, pp. 57–60.

[10] D. E. Hocevar, "LDPC code construction with flexible hardware implementation," in *IEEE International Conference on Communications*, 2003, pp. 2708 –2712.

[11] H. Zhong and T. Zhang, "Design of VLSI implementation-oriented LDPC codes," in *IEEE Semiannual Vehicular Technology Conference (VTC)*, Oct. 2003.

[12] Flarion Technologies, "Methods and Apparatus for Decoding LDPC Codes," *US patent number: 6,633,856 B2*, Oct. 2003.

[13] Jose M. F. Moura, J. Lu, , and H. Zhang, "Structured low-density parity-check codes," *IEEE signal processing magazine*, pp. 42–55, Jan 2004.

[14] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 638–656, Feb. 2001.

[15] D.E Hocevar, "Efficient encoding for a family of quasi-cyclic LDPC codes," *Global Telecommunications Conference*, vol. 7, pp. 3996 – 4000, 2003.

[16] E. Eleftheriou and S. Olcer, "Low-density parity-check codes for digital subscriber lines," in *Proc. IEEE International Conference on Communications*, 2002, pp. 1752–1757.

[17] T. Zhang and K. K. Parhi, "Joint $(3, k)$-regular LDPC code and decoder/encoder design," *IEEE Transactions on Signal Processing*, vol. 52, no. 4, pp. 1065-1079, April 2004.

[18] T. Tian, C. Jones, J. D. Villasenor, and R. D. Wesel, "Construction of irregular LDPC codes with low error floors," in *Proc. IEEE International Conference on Communications*, 2003, pp. 3125–3129.

[19] T. Richardson, A. Shokrollahi, and R. Urbanke, "Design of capacity-approaching low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, Feb. 2001.

[20] J. Campello and D. S. Modha, "Extended bit-filling and LDPC code design," in *Proc. IEEE Global Telecommunications Conference*, 2001, pp. 985–989.

[21] J. Thorpe, "Design of LDPC graphs for hardware implementation," in *Proc. IEEE International Symposium on Information Theory*, 2002, pp. 483–483.

[22] The Boost Graph Library, http://www.boost.org/libs/graph/.

[23] D. Declercq and F. Verdier, "Optimization of LDPC finite precision belief propagation decoding with discrete density evolution," *3rd International Symposium on Turbo Codes and Related Topics Brest, France*, September 2003.