Overlapped Message Passing for Quasi-Cyclic Low-Density Parity Check Codes

Yanni Chen, *Member, IEEE,* and Keshab K. Parhi, *Fellow, IEEE*

***Abstract*—In this paper, a systematic approach is proposed to develop a high throughput decoder for quasi-cyclic low-density parity check (LDPC) codes, whose parity check matrix is constructed by circularly shifted identity matrices. Based on the properties of quasi-cyclic LDPC codes, the two stages of belief propagation decoding algorithm, namely, check node update and variable node update, could be overlapped and thus the overall decoding latency is reduced. To avoid the memory access conflict, the maximum concurrency of the two stages is explored by a novel scheduling algorithm. Consequently, the decoding throughput could be increased by about twice assuming dual-port memory is available.**

***Index Terms*—High throughput, low-density parity check (LDPC) codes, overlapped message passing (MP), quasi-cyclic codes.**

I. Introduction

SINCE its recent rediscovery [1], low-density parity check (LDPC) codes first introduced by Gallager [2] have been of great research interest in terms of exploring good code construction methods [3]–[7] as well as efficient VLSI implementation architectures [8]–[10]. LDPC codes are considered a serious competitor to the turbo codes [11], [12] because they push the performance closer to the Shannon limit [7], [13]. On the other hand, like block turbo codes [14], parallel decoder structure could also be developed for LDPC codes to achieve high decoding throughput. However, due to its typical sparse, random, and large parity check matrix used to construct LDPC code, the data routing and large memory requirement imposes great design challenges for the efficient hardware implementations.

In the current literature, most LDPC code designs rely on random construction of the parity check matrix [1]. However, the resulting lack of structure makes the code difficult to describe efficiently, hard to implement and also requires extensive testing to assure the performance. Recently, as opposed to random construction of LDPC codes, the group-structured quasi-cyclic LDPC codes are proposed [15], [16]. This particular type of codes is of interest because they provide good performance and are hardware friendly. It has been shown that the regular quasi-cyclic LDPC codes can achieve comparable performance to randomly constructed codes if the codeword length is less than 10000 bits [5]. The performance could be further improved by optimizing the check node as well as variable node degrees [6]. Moreover, the decoder implementation of quasi-cyclic LDPC codes significantly simplifies the memory address generation and wire interconnections [17].

For many real-world applications, the high-speed fully parallel decoder architecture such as [8], where the Tanner graph corresponding to the parity check matrix is exactly instantiated to the hardware, is too complicated due to the large number of functional units and prohibitive routing. To reduce the hardware complexity, a partly parallel architecture, where certain logic devices have to be utilized in a time-multiplexed manner, is also implemented using either application specified integrated circuit (ASIC) [9] or field programmabel gate array (FPGA) [18]. Among several different partly parallel decoders, the approach in [19] achieves higher speed with larger memory requirement compared to the design in [10]. The latter architecture requires simpler control logic, minimum memory size, namely, the total nonzero elements of parity check matrix, and hence is of our interest in this paper.

The message passing (MP) (also called belief propagation or sum product) algorithm [20] is commonly used to iteratively decode the LDPC codes. By using the MP algorithm, two different types of messages, i.e., variable-to-check message and check-to-variable message, are computed and exchanged along the edges in the corresponding Tanner graph. Conventionally, these two stages of computation cannot be overlapped because one stage needs the updated information passing from the other stage. Otherwise, conflicts are inevitable during the memory access for information exchange. However, due to the inherent characteristics of quasi-cyclic LDPC codes, certain concurrency exists between the two stages, which could be utilized to reduce the decoding latency.

In this paper, based on [17], the maximum concurrency between the two stages for quasi-cyclic LDPC codes is explored by a novel scheduling algorithm, which systematically determines the memory address generation and perfectly resolves the memory access conflict for the partly parallel decoder architecture. Consequently, the major disadvantage of low hardware utilization efficiency (HUE) for the design presented in [10] is overcome and the decoding throughput could be increased by about two times assuming dual-port memory is available. To evaluate our scheme, a case study with the (155, 64, 31) code is employed to verify the correctness of the proposed scheduling algorithm.

The structure of this paper is as follows. In Section II, we first briefly review the quasi-cyclic LDPC code construction and its top-level decoder structure. Section II also presents the MP

Manuscript received June 11, 2003; revised December 6, 2003. This work was supported by the Army Research Office under Grant DA/DAAD19-01-1-0705. This paper was presented in part at the ACM Great Lake Symposium on VLSI, April 26–29, 2003, Washington, DC. This paper was recommended by Associate Editor Y. Wang.

Y. Chen is with DSP Solutions R & D Center, Texas Instruments Incorporated, Dallas, TX 75243 USA (e-mail: ynchen@ece.umn.edu).

K. K. Parhi is with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA (e-mail: parhi@ece.umn.edu).

Digital Object Identifier 10.1109/TCSI.2004.826194

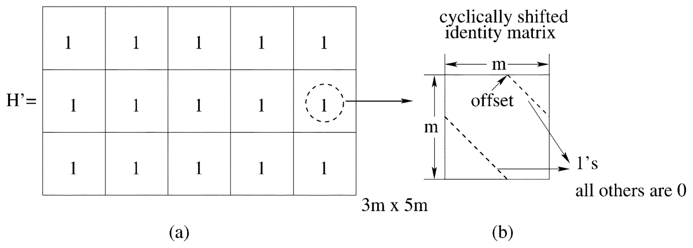

Fig. 1. Code construction of regular (3, 5) quasi-cyclic LDPC codes. (a) Block matrix H'. (b) $m \times m$ circularly shifted identity matrix with certain offset.

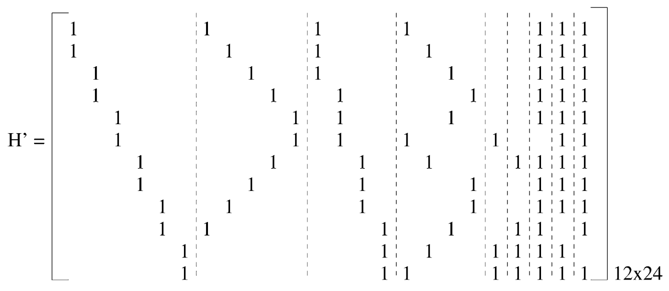

Fig. 2. Block matrix H' of irregular quasi-cyclic LDPC code with rate 1/2.

iterative decoding algorithm and block diagrams of functional units. The proposed overlapped MP is presented in Section III. All the details on how to solve the memory access conflict as well as a case study are also explained in this section. Finally, some conclusions are drawn in Section IV.

II. Quasi-Cyclic LDPC Codes

In this section, the code construction and their performance curves of quasi-cyclic LDPC codes will be briefly reviewed. Its decoder architecture will then be illustrated.

A. Code Construction

The LDPC code may be described in terms of a parity check matrix H, which satisfies Hx modulo $2 = 0$ for all codewords x. The matrix H for quasi-cyclic LDPC codes can be constructed as follows: for a desired (j, k) code, first construct the block all-one matrix H' with size of $j \times k$; then, replace each element in H' with $m \times m$ cyclically shifted identity matrix with certain offsets, where m is a prime number and j, k are among the prime factors of $m-1$. One example of (3, 5) code is shown in Fig. 1. The offsets are given by $\left(b^{(s-1)} \times a^{(t-1)}\right)$ modulo m, where $1 \leq s \leq j$, $1 \leq t \leq k$ and a, b have multiplicative orders of k, j, respectively. Dependent on the (s, t) locations, the offsets might be different for different identity matrices within H', for the sake of clarity, a subscript denoting as $\text{offset}_{s,t}$ will be adopted in later sections.

Following the construction method described above, the obtained (j, k) code has parity check matrix H with size of $jm \times km$ and code rate $R \geq 1 - (j/k)$ (there are at least $j - 1$ dependent rows in H). This code is regular code since both the variable node degrees and check node degrees are constants. To achieve better decoding performance, irregular quasi-cyclic LDPC codes could also be constructed by optimizing the node degrees. The block matrix H' for one irregular quasi-cyclic LDPC code as an example of rate 1/2 code is shown in Fig. 2 [6]. It is also in the category of quasi-cyclic codes since each element within H' is actually a $m \times m$ circularly shifted identity matrix with certain offset as regular quasi-cyclic LDPC codes.

B. MP Decoding Algorithm

The MP algorithm is generally employed to decode LDPC codes. Following the original MP algorithm, in every iteration, two types of messages passed between variable nodes and check nodes have to be updated. The check-to-variable message R_{cv} for the check node c and variable node v using the incoming variable-to-check messages L_{cn} is computed by the check node functional unit (CNFU) as follows:

$$S_{cv} = \Pi_{n \in N(c), n \neq v} sign(L_{cn}) \tag{1}$$

$$R_{cv} = -S_{cv}\Psi\left(\sum_{n \in N(c), n \neq v} \Psi(L_{cn})\right) \tag{2}$$

where S_{cv} is the sign part of R_{cv} and $N(c)$ denotes the set of variable nodes connected to the check node c. The function $\Psi(x) = \log(\tanh(|x/2|))$ can be implemented by look-up-table (LUT) operations and each LUT normally has 32 5-bit entries [9].

On the other hand, the variable-to-check message L_{cv} for the check node c and variable node v using the incoming check-to-variable messages R_{cv} and received channel information r_v is computed by the variable node functional unit (VNFU)

$$L_v = \sum_{c \in M(v)} R_{cv} - \frac{2r_v}{\sigma^2} \tag{3}$$

$$L_{cv} = L_v - R_{cv} \tag{4}$$

where $M(v)$ is the set of check nodes connected to variable node v and $2r_v/\sigma^2$ is the intrinsic information while σ stands for the estimated standard deviation of the additive white Gaussian noise (AWGN) channel. The soft output L_v for the variable node v is later sliced to check whether all the parity check equations are satisfied, i.e., the decoded output is a codeword or not.

1) Simulated Results: According to the above MP decoding algorithm, the simulation results of some (N, K, m) codes are shown in Fig. 3, where N stands for the codeword length, K is the information length, and m is the prime number used to construct the circularly shifted identity matrix. The solid lines are for regular codes with code rate of around 0.4 while the dashed line is for the irregular code with code rate of 0.5. All the performance are evaluated over the AWGN channel with BPSK modulation.

As expected, we can easily see that in Fig. 3, the bit error rate (BER) performance is significantly improved with larger codeword length N. Furthermore, the performance of regular codes could be ameliorated by optimizing the node degrees to construct irregular LDPC codes [6]. For the (4632, 2316, 193) code, the BER of 10^{-5} could be achieved at the signal-to-noise ratio of about 1.4 dB. Using the inherent early termination feature of LDPC codes, namely, the iterative process will be stopped earlier if the codeword is found based on the sliced soft outputs at the end of every iteration, the average number of iteration for

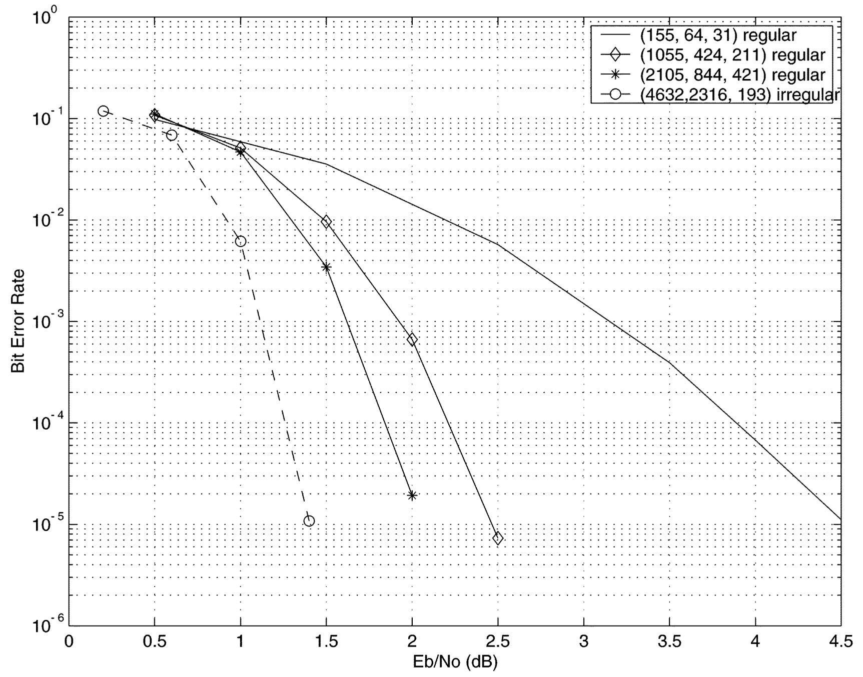

Fig. 3. Simulated performance of quasi-cyclic LDPC codes.

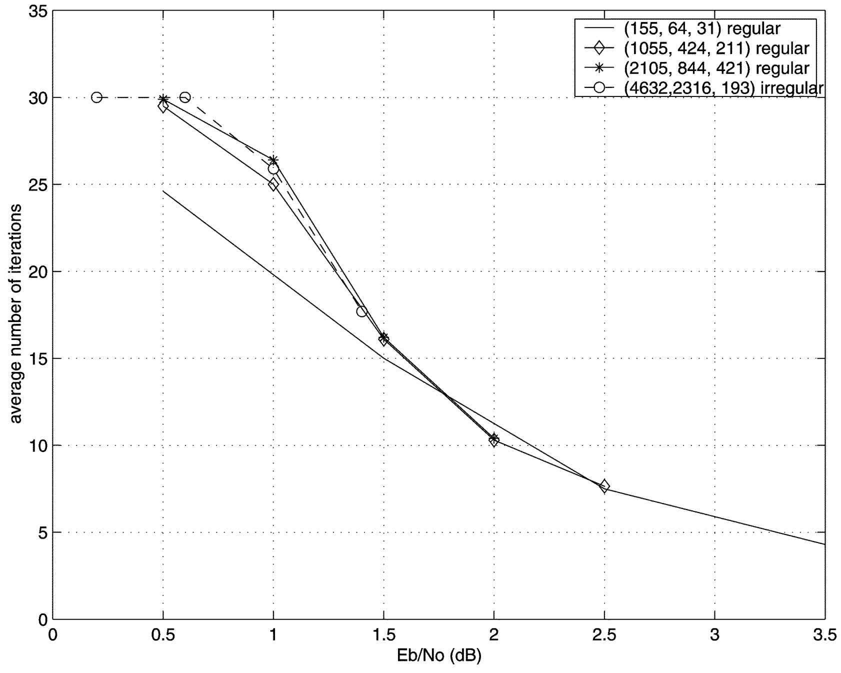

Fig. 4. Average number of iterations for quasi-cyclic LDPC codes.

all the four considered codes are depicted in Fig. 4 assuming the maximum number of iterations is set to 30.

From Fig. 4, it is observed that both regular and irregular quasi-cyclic LDPC codes almost converge at the same speed in spite of better coding gain for irregular code. In later sections, the (3, 5) regular code will be our example due to its relatively smaller j, k values for easier explanations.

2) Node Functional Units: In terms of hardware implementation, the structures of CNFU and VNFU for the (3, 5) regular code could be illustrated as in Figs. 5 and 6, respectively.

For the sake of clarity, the parity checking part is not shown in Fig. 5 and the intrinsic information $2r_v/\sigma^2$ is represented by z_v in Fig. 6. It is also worth noting that the data format transformation block, either from sign magnitude (SM) to two's (2's) complement format or *vice versa*, exists in both types of functional units. The major advantage of using SM format for LUT operations is that each LUT size can be reduced by half by making use of the symmetry properties of $\Psi(x)$ function. However, it is still more convenient to use 2's complement format in VNFU computations.

C. Decoder Structure

For the (j, k) regular quasi-cyclic LDPC code, one straightforward approach similar to [18] adopts j k-input CNFUs, k j-input

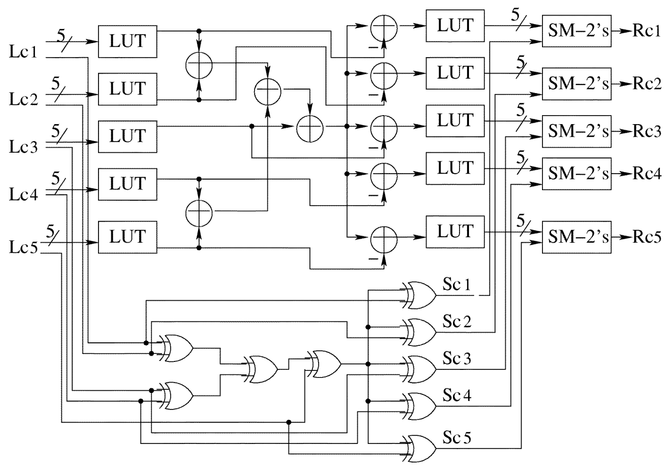

Fig. 5. CNFU.

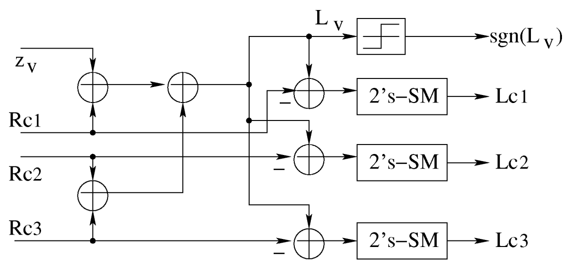

Fig. 6. VNFU.

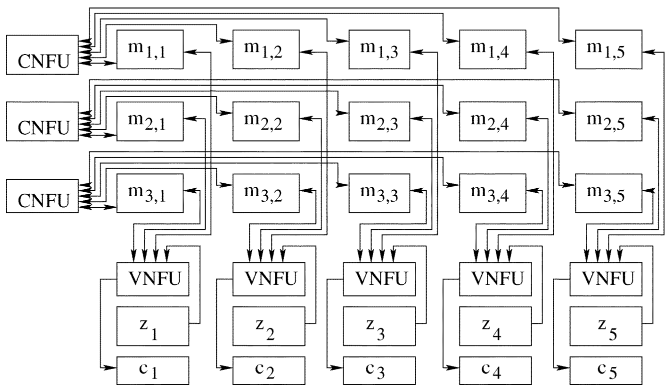
CNFU
$m_{1,1}$ $m_{1,2}$ $m_{1,3}$ $m_{1,4}$ $m_{1,5}$
$m_{2,1}$ $m_{2,2}$ $m_{2,3}$ $m_{2,4}$ $m_{2,5}$
$m_{3,1}$ $m_{3,2}$ $m_{3,3}$ $m_{3,4}$ $m_{3,5}$
VNFU
z_1 z_2 z_3 z_4 z_5
c_1 c_2 c_3 c_4 c_5

Fig. 7. Partly parallel decoder for quasi-cyclic (3, 5) LDPC code.

VNFUs and $j \times k$ extrinsic information memories $m_{s,t}$, where $1 \leq s \leq j$ and $1 \leq t \leq k$. The case of $j = 3$ and $k = 5$ is depicted in Fig. 7. Note that the intrinsic information is retrieved in z memories while c memories store the hard decisions of soft outputs $sgn(L_v)$ used for parity checking in the subsequent iteration, where $1 \leq v \leq N$. In this partly parallel decoder structure, each memory contains m memory words. The wordlength is dependent on whether it is extrinsic information, intrinsic information or 1-bit hard decision bit. Furthermore, there is one modulo-m counter associated with each memory to generate the corresponding memory address, which always counts up to $m-1$ starting from certain initial value and then wraps around to zero.

For the considered decoder structure in Fig. 7, the decoding process can be carried out as follows.

- *Initialization*: Flush the received intrinsic information to both the z memories and the corresponding $j \times k$ extrinsic information memories $m_{s,t}$. The data are stored column-wise in the z memories and row-wise in $m_{s,t}$ memories. However, in order to store the incoming intrinsic information row-wise in the $m_{s,t}$ memories while they are received column-wise, during the initialization process, the starting memory addresses for the $m_{s,t}$ memories have to be different for different block rows and dependent on the offsets computed by $\left(b^{(s-1)} \times a^{(t-1)}\right)$ modulo m. Here, *block row* means those m rows sharing the same s value.
- *Check node update*: In each subsequent iteration, the updated variable-to-check messages are simultaneously read from all the $j \times k$ $m_{s,t}$ memories by all the CNFUs, each CNFU read k memories in a block row. For instance, memories $m_{2,1}, m_{2,2}, \ldots, m_{2,5}$ are read by the same second CNFU in Fig. 7. Then, after CNFU computation the updated check-to-variable messages are written back to the same address r in the order of $0, \ldots, m-1$ for all the k $m_{s,t}$ memories in a block row. Therefore, in one clock cycle j rows are updated simultaneously, namely, rows $r, m+r, 2m+r, \ldots, (j-1)m+r$. Consequently, in one iteration totally m clock cycles are required to complete the updating process of all the $j \times m$ rows.
- *Variable node update*: Similarly, in the same iteration, the updated check-to-variable messages are simultaneously read from all the $j \times k$ $m_{s,t}$ memories by all the VNFUs, each VNFU read j memories in a block column. Here, *block column* means those m columns sharing the same t value. For instance, memories $m_{1,1}$, $m_{2,1}$, $m_{3,1}$ are read by the same first VNFU in Fig. 7. Then, after the VNFU computation, the updated variable-to-check messages are written back to the *same* address as read operations in the order of $m - \text{offset}_{s,t}, \ldots, m-1, 0, \ldots, m - \text{offset}_{s,t} - 1$. Consequently, in one clock cycle k columns are updated simultaneously, namely, columns $l, m+l, 2m+l, \ldots, (k-1)m+l$, where $0 \leq l \leq m-1$. In one iteration, totally m clock cycles are required to complete the updating process of all the $k \times m$ columns. It is worth noting that for the check node update, all the k memories in one block row share the same memory address while this is not the case for the variable node update since all the j memories in one block column have different $\text{offset}_{s,t}$ values thus different addresses.
- *Parity checking*: At the end of every iteration, all the soft outputs computed during the variable node update are sliced to check the parity equations for all the $N - K$ rows. The iterative process will be terminated when either all the parity check equations are satisfied, that is to say, one codeword x satisfying $Hx = 0$ is found, or the pre-assigned maximum number of iterations is reached. This step could also be concurrently proceeded with the check node update as well.

The decoder structure in Fig. 7 has the obvious advantage of memory requirement. Let the wordlengths of the extrinsic and intrinsic information are W_e and W_z, respectively. The total required data storage is for all the extrinsic information (size equal to the number of nonzero elements in the parity check matrix H multiplied by W_e), the intrinsic information (size equal to the codeword length N multiplied by W_z) and the hard decision

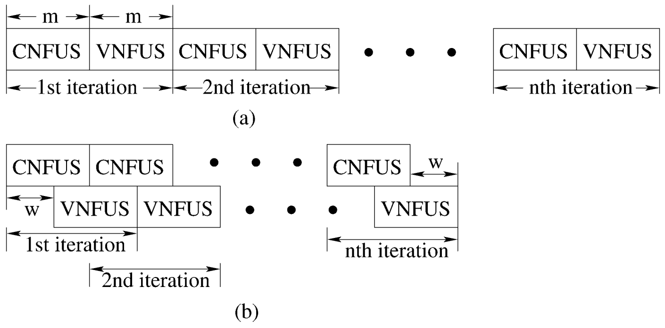

Fig. 8. Scheduling of CNFUs and VNFUs.

bits of soft outputs (size equal to the codeword length N). This is the least memory requirement if the original MP decoding algorithm is strictly followed without introducing any performance degradation. Furthermore, this structure possesses some other nice features including straightforward memory address generation, localized memory access and simple routing.

From the decoding process outlined above, we know that in one iteration both check node update and variable node update operations have to be performed, one after another. This leads to the merely 50% HUE of the logic core for the decoder in Fig. 7 because all the VNFUs are idle when CNFUs are busy during the check node update and *vice versa* during the variable node update. To improve the HUE of logic core, the approach we considered is to overlap the check node update and variable node update operations by making use of inherent properties of quasi-cyclic LDPC codes to get higher throughput design.

III. Overlapped MP Decoding

In this section, the proposed overlapped MP for the considered partly parallel decoder structure will be explained in detail.

A. Scheduling of CNFU and VNFU

Due to the randomness characteristics of parity check matrix H, the computation by CNFUs and VNFUs generally could not be overlapped; thus, they are always computed one after another in one iteration. This sequential process can be illustrated as in Fig. 8(a). Totally, n iterations are assumed.

As discussed in Section II-C, both CNFUs and VNFUs need m clock cycles to finish computing one block row and one block column, respectively. Therefore, in Fig. 8(a) for one iteration, altogether $2 \times m$ clock cycles are required to update all the $N-K$ rows and N columns if CNFUs and VNFUs are sequentially executed. By realizing the data independence among the rows, not all the CNFUs have to start from the same initial address as long as the cyclic order is assured, this sequence tie between CNFUs and VNFUs can be naturally broken.

If assuming dual-port memories are available, CNFUs and VNFUs can be computed as shown in Fig. 8(b). Instead of waiting for m clock cycles, VNFUs can begin to update the variable-to-check messages only after w clock cycles' computation of CNFUs, here w is referred to as waiting time in later sections and $0 < w \leq m$. Likewise, CNFUs in the subsequent iteration can start after $m - w$ clock cycles' computation of VNFUs. In this way, the number of clock cycles of the entire iterative process for the two cases in Fig. 8(a) and (b) are $(2 \times m \times n)$ and $(m \times n + w)$, respectively. The throughput gain through the overlapped decoding compared to the original decoding is

$$\text{throughput gain} = \frac{2 \times m \times n}{m \times n + w} \approx 2. \tag{5}$$

To achieve this, the following three constraints need to be satisfied.

- No performance degradation is introduced, which means the data flow should not be changed.
- No memory access conflict since both CNFUs and VNFUs exchange message through the same $m_{s,t}$ memories.
- We minimize waiting time w to achieve throughput gain closer to two according to (5).

B. Waiting Time Minimization

An algorithm is developed to systematically minimize the waiting time w for the scheduling scheme depicted in Fig. 8(b). Intuitively, the minimal waiting time w could be obtained through the following five steps.

1) Choose the first block row as a reference basis, where memory addresses of k $m_{s,t}$ memories start from 0 and counts up to $m-1$. On the other hand, for all the other $(j-1)$ block rows except for the reference basis, the memory addresses start from constants $c_1, c_2, \ldots, c_{j-1}$, respectively.
2) Independently determine these $(j-1)$ constants $c_1, c_2, \ldots, c_{j-1}$ such that they individually minimize $\max\{d_{s,t}\}$ values in a block row, namely, the maximum of the k values $\{d_{s,0}, d_{s,1}, \ldots, d_{s,k-1}\}$ is minimized by one constant. Here, $d_{s,t}$ stands for the column index difference between the block row $s(0 < s \leq j-1)$ and the reference basis block row (RB) for block column $t(0 \leq t \leq k-1)$; thus, $d_{s,t} = (\text{offset}_{s,t} - \text{offset}_{RB,t} + c_s)$ modulo m. It can be easily observed that in order to obtain the minimum $\max\{d_{s,0}, d_{s,1}, \ldots, d_{s,k-1}\}$ value, c_s should be equal to $\min_{0 \leq t \leq k-1}\{\text{offset}_{RB,t} - \text{offset}_{s,t}\}$ modulo m. As a result, the $\max\{d_{s,0}, d_{s,1}, \ldots, d_{s,k-1}\}$ for different s values are denoted as $d_1, d_2, \ldots, d_{j-1}$, respectively.
3) The waiting time for this first reference basis W_{b1} is thus the maximum column index difference among all the $(j-1)$ block rows, i.e., $W_{b1} = \max\{d_1, d_2, \ldots d_{j-1}\}$.
4) Change the reference basis to the other $(j-1)$ block rows, repeat the steps from step 1 to step 3 for each reference basis, and denote the obtained waiting times as $W_{b2}, W_{b3}, \ldots, W_{bj}$.
5) The minimum waiting time w is thus given by $\min\{W_{b1}, W_{b2}, W_{b3} \ldots W_{bj}\}$. For the winner basis (WB) giving w, its corresponding CNFUs memory address is still in the order of $0, \ldots, m-1$ as the original decoding while for all the other CNFUs the starting memory addresses are merely those constants $c_1, c_2, \ldots c_{j-1}$ for the chosen basis BS, respectively. On the other hand, during the variable-node update operations, starting memory addresses for the overlapped decoding should be $(\text{offset}_{WB,t} - \text{offset}_{s,t} + w)$ modulo m instead of $(m - \text{offset}_{s,t})$ modulo m for the original decoding.

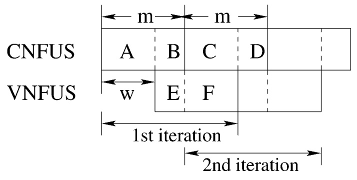

Fig. 9. Constraint model for CNFUs and VNFUs.

Since all the five steps above could be pre-computed, that is to say, for any specified code the minimal waiting time w, the winner basis and the constants for other block rows other than the winner basis can be determined before the decoding process is actually started, no hardware overhead is introduced.

C. Constraint Satisfaction

To verify the validity of the algorithm described in Section III-B, a constraint model for the overlapped MP decoding is developed as depicted in Fig. 9 based on the Fig. 8(b). For the sake of clarity, only the first two iterations are shown here.

In the first iteration, CNFUs are divided into region A and B while VNFUs are divided into regions E and F. Similarly, C and D are the two regions of CNFUs in the second iteration. To meet the constraint of no memory access conflict, both, the so-called *intra-iteration constraint*, and the *inter-iteration constraint* have to be satisfied. Here, the former constraint means that all the starting addresses in region E (F) should have already been updated in region A (A or B), which guarantees that VNFUs have to start only after all the check-to-variable messages in the same column passed by CNFUs are already available. On the other hand, the latter constraint states that all the starting addresses in region C (D) should have been computed in region E (E or F), which implies that CNFUs start only after all the variable-to-check messages in the same block row passed by VNFUs have already been updated.

D. Slightly Modified Memories Addressing

The algorithm of computing w itself assures there is no memory access conflict for the very first iteration. However, in the case of large w values, some conflicts might happen if in the following iteration all the CNFUs still use the same memory starting addresses as in the previous iterations. Therefore, some modifications of the initial addresses in later iterations are necessary. For even iterations, the worst case conflict happens if the first memory word need to be updated by any of CNFUs in the current iteration is the last memory word utilized by any of VNFUs in the previous iteration. In this case, the maximum memory miss is w clock cycles. This problem can be solved like this: subtract all the memory starting addresses during CNFUs and VNFUs operations in even iterations by w. Likewise, subtract all the memory starting addresses during CNFUs and VNFUs operations in odd iterations by $m - w$, which actually return to the same value as the first iteration.

Following the scheme described above, both the inter- and intra- iteration constraints are satisfied. There is no memory access conflict, and the old messages are always utilized before they are updated. Unfortunately, in a small amount of clock cycles there is such an undesirable case where CNFU and VNFU have to update the same memory word in the same clock cycle.

1	2	4	8	16
5	10	20	9	18
25	19	7	14	28

3 x 5

Fig. 10. Offsets of identity matrices for the (155, 64, 31) code.

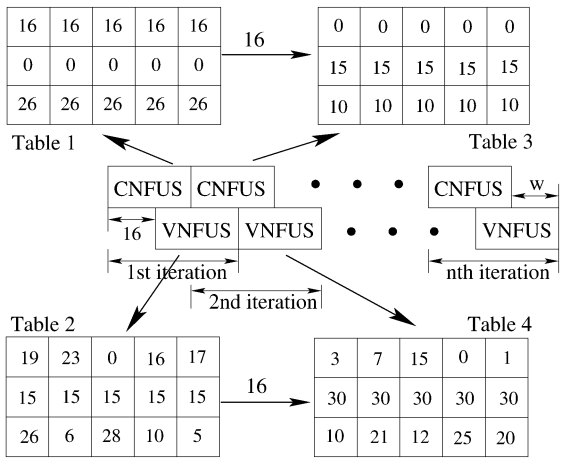

Table 1

16	16	16	16	16
0	0	0	0	0
26	26	26	26	26

Table 3

0	0	0	0	0
15	15	15	15	15
10	10	10	10	10

Table 2

19	23	0	16	17
15	15	15	15	15
26	6	28	10	5

Table 4

3	7	15	0	1
30	30	30	30	30
10	21	12	25	20

Fig. 11. Memory starting addresses for CNFUs and VNFUs.

To avoid the contradiction, the operations order in that clock cycle has to be arranged like this: the CNFU first fetches that memory word, then updates the check-to-variable message. Before the new message is written back to the same memory address, it is *immediately* utilized by the corresponding VNFU, which updates its variable-to-check message and store it back to that memory address. To completely eliminate this case, the slightly modified overlapped scheduling is proposed and summarized as follows.

1) Based on the five steps in Section III-B, pre-compute the waiting time w and other related information including winner basis, memories starting addresses for CNFUs and VNFUs in the first iteration. Let the new minimal waiting time $w' = w + 1$.
2) At the end of check node or variable node computations in every iteration, either CNFUs or VNFUs are idle for one clock cycle. Consequently, the new throughput gain becomes $\{(m+1) \times n + w\}/(2 \times m \times n) \approx 2$.
3) Taking the worst case memory miss into consideration, all the memories starting addresses have to be subtracted accordingly at the end of *each* iteration by w'.

E. Case Study of the (155, 64, 31) Code

Applying the proposed overlapped decoding in Section III-D to the regular (155, 64, 31) code, we found that the required number of clock cycles is indeed reduced by about two times without introducing any memory access conflict or performance degradation.

For the (155, 64, 31) code, its prime number is $m = 31$ and $(j, k) = (3, 5)$. The code generators b, a are equal to 5 and 2, respectively. According to the code construction method described in Section II-A, the following offsets listed in Fig. 10 for each of 3×5 identity matrices could be obtained by computing $(5^s \times 2^t)$ modulo 31.

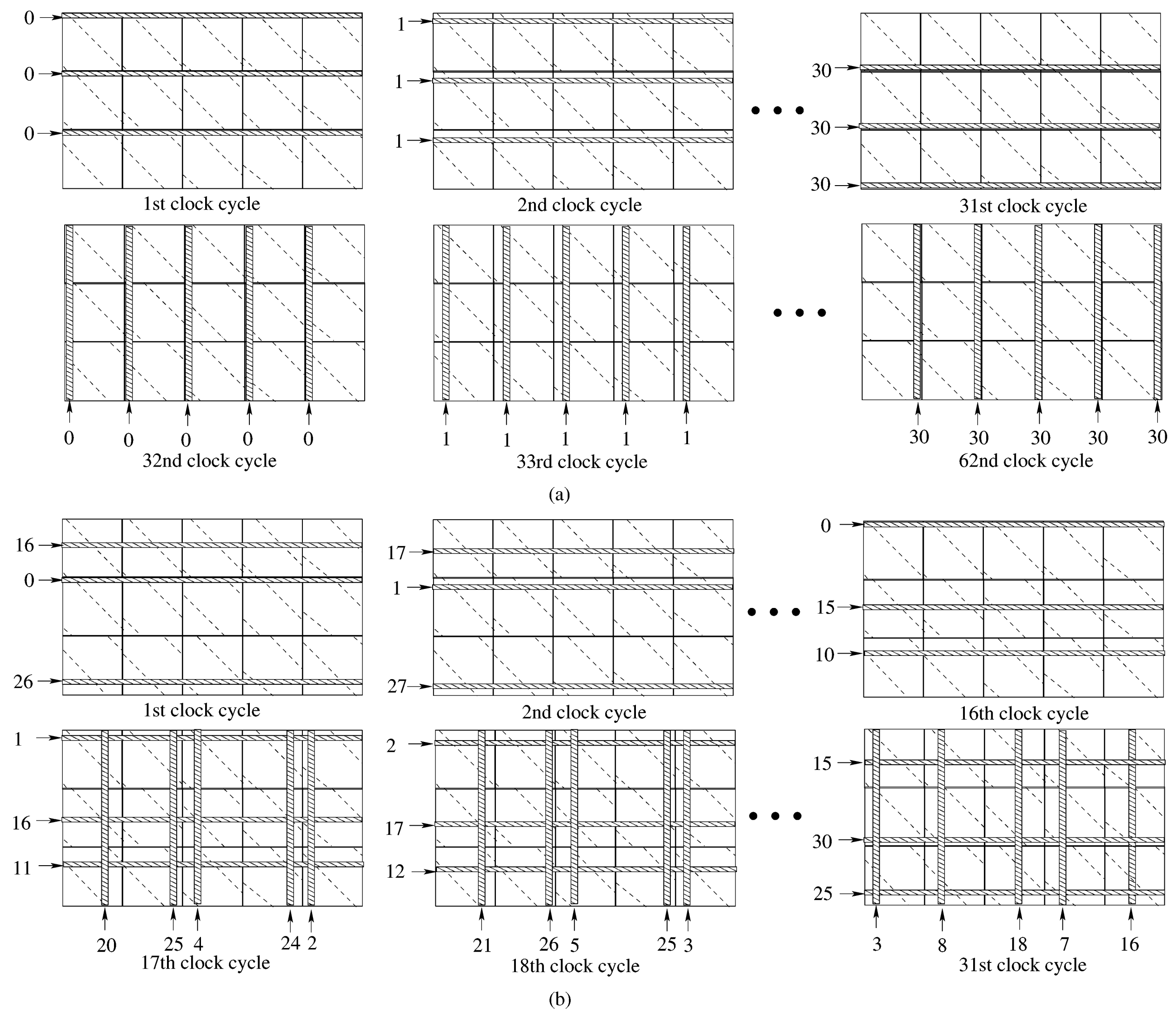

Fig. 12. Alternatively interpretation of partly parallel decoder. (a) Original decoding. (b) Overlapped decoding.

Then, by following the five steps explained in Section III-B, we can find that the winner basis is the second block row, the waiting time w is equal to 15 and, thus, w' is 16 clock cycles. Furthermore, the constants for the first c_1 and third block rows c_3 are 16 and 26, respectively. Based on the iteration flow in Fig. 8(b), the corresponding starting addresses of each individual memory for the first two iterations, during either CNFUs or VNFUs operations, are listed in the four tables shown in Fig. 11.

From Fig. 11, we know that the entries in Table 1 in Fig. 11 directly reflect the values obtained from the pre-computation, where all zeros are in the second row since it is the winner basis. Obviously, the values in the first row and third row are the constants c_1 and c_3, respectively. For Table 2 in Fig. 11, one easy way to prove the correctness of our method is that by adding the entries with their corresponding offsets in Fig. 10, the exactly same value is obtained for each column, which flags the same starting column of VNFUs operations for each block column. It is perfectly synchronized without any memory access conflict. Furthermore, it is easily seen that those columns have already been updated by CNFUs; thus, the intra-iteration constraint is met. For later iterations, it is very straightforward to determine the memories starting addresses by simply subtracting 16 from the previous iteration for both CNFUs and VNFUs. A similar verification method could be adopted to check whether inter-iteration and intra-iteration constraints are met. For the sake of brevity, the details are omitted here.

Our proposed overlapped decoding could also be alternatively interpreted in time order as illustrated in Fig. 12. The numbers beside the arrows denote the corresponding memories addresses in the current clock cycle.

For the considered partly parallel decoder for the (3, 5) regular code, three check nodes or/and five variable nodes are processed in one clock cycle. In the original decoding in Fig. 12(a), the check node update and variable node update are processed one after another. During the check node update, in the first clock cycle, the first check nodes in the three block rows are first updated, and then the three second check nodes are updated in the second clock cycle. Therefore, the last three check codes are updated in the the 31st clock cycle. In a similar manner, during the variable node update, the first variable nodes in the five block columns are updated in the 32nd clock cycle, and then another subsequent five variable nodes are updated. Finally, in the 62nd clock cycle the last five variable nodes are updated, which completes one iteration.

In contrast, for overlapped decoding shown in Fig. 12(b), in the first clock cycle, three check nodes, which are not necessarily the first check nodes in the respective block row except the winner basis, from three block rows are updated. Then, as original decoding, another three neighbor check nodes are updated in the next clock cycle. However, after 16 (minimal waiting time) clock cycles, three check nodes *and* five variable nodes are updated simultaneously in the s17th clock cycle. This is exactly what we are trying to achieve for overlapped decoding. Certainly, those check and variable nodes have to be chosen by strictly following the steps described in Sections III-B and D to avoid memory access conflict. Obviously, in the 18th clock cycle another immediate three check nodes and five variable nodes are updated until the 31st clock cycle when the check node update is finished for the first iteration. Then, starting from the 33rd clock cycle, the variable node update in the first iteration and check node update in the second iteration will be overlapped. This process continues until the iterative process is terminated.

IV. Conclusion

In this paper, for the so-called quasi-cyclic LDPC codes, a new scheme is proposed on how to generate memory address for the overlapped MP decoding. The considered decoder architecture significantly simplifies the memory address generation and wire interconnections. The new scheme exploits the maximum concurrency between two stages of belief propagation algorithm and entirely eliminate the memory access conflict. The application of the proposed approach to the real example verifies its correctness. If dual-port memory is available, the decoding throughput could be increased almost twice, which provides a good tradeoff between area and speed. This approach can also be easily extended to irregular quasi-cyclic LDPC codes as long as the similar decoder structure is adopted.

References

[1] D. J. C. MacKay and R. M. Neal, “Near Shannon limit performance of low-density parity check codes,” *Electron. Lett.*, vol. 32, p. 1645, 1996.

[2] R. G. Gallager, “Low-density parity check codes,” *IRE Trans. Info. Theory*, vol. IT-8, pp. 21–28, 1962.

[3] Y. Kou, S. Lin, and M. P. C. Fossorier, “Low-density parity-check codes based on finite geometries: A rediscovery and new results,” *IEEE Trans. Inform. Theory*, vol. 47, pp. 2711–2736, Nov. 2001.

[4] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, “Design of capacity approaching irregular low-density parity-check codes,” *IEEE Trans. Inform. Theory*, vol. 47, pp. 619–637, 2001.

[5] D. Sridhara, T. Fuja, and R. M. Tanner, “Low-density parity check codes from permutation matrices,” in *Proc. Conf. Information Sciences and Systems*, Baltimore, MD, Mar. 2001.

[6] D. Hocevar, “LDPC code construction with flexible hardware implementation,” in *Proc. IEEE Int. Conf. Communications*, vol. 4, May 11–15, 2003, pp. 2708–2712.

[7] S.-Y. Chung, G. D. Forney Jr, T. J. Richardson, and R. Urbanke, “On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit,” *IEEE Commun. Lett.*, vol. 5, pp. 58–60, Feb. 2001.

[8] 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 J. Solid-State Circuits*, vol. 37, pp. 404–412, Mar. 2002.

[9] Y. Chen and D. Hocevar, “A FPGA and ASIC implementation of rate 1/2, 8088-b irregular low-density parity check decoder,” in *Proc. Global Telecommunications Conf. GLOBECOM’03*, vol. 1, Dec. 1–5, 2003, pp. 113–117.

[10] T. Zhang and K. K. Parhi, “VLSI implementation oriented $(3, k)$-regular low-density parity-check codes,” in *Proc. IEEE Workshop Signal Processing Systems*, Sept. 26–28, 2001, pp. 25–36.

[11] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error correcting coding and decoding: Turbo codes,” in *Proc. IEEE Int. Conf. Communications*, vol. 2, May 1993, pp. 1064–1070.

[12] R. Pyndiah, A. Glavieux, A. Picart, and S. Jacq, “Near-optimum decoding of product codes,” in *Proc. Global Telecommunications Conf. GLOBECOM’94*, vol. 1, Nov. 1994, pp. 339–343.

[13] C. E. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, pp. 379–423 , 1948.

[14] R. Pyndiah, “Near-optimum decoding of product codes: Block turbo codes,” *IEEE Trans. Commun.*, vol. 46, pp. 1003–1010, Aug. 1998.

[15] R. M. Tanner, “A class of group-structured LDPC codes,” in *Proc. ICSTA*, Ambleside, U.K., July 2001.

[16] A. Sridharan, D. J. Costello Jr, D. Sridhara, T. Fuja, and R. M. Tanner, “A construction for low-density parity check convolutional codes based on quasi-cyclic block codes,” in *Proc. IEEE Int. Symp. Information Theory*, 2002, p. 481.

[17] Y. Chen and K. K. Parhi, “High throughput overlapped message passing for low-density parity check codes,” in *Proc. IEEE/ACM GLSVLSI*, 2003, pp. 245–248.

[18] T. Zhang and K. K. Parhi, “A 54 MBPS (3, 6)-regular FPGA LDPC decoder,” in *Proc. IEEE Int. Symp. Information Theory*, Oct. 16–18, 2002, pp. 127–132.

[19] E. Boutillon, J. Castura, and F. R. Kschischang, “Decoder-first code design,” in *Proc. Int. Symp. Turbo Codes and Related Topics*, Sept. 2000, pp. 459–462.

[20] J. Chen and M. P. C. Fossorier, “Near optimum universal belief propagation based decoding of low-density parity check codes,” *IEEE Trans. Commun.*, vol. 50, pp. 406–414, Mar. 2002.

Yanni Chen (S’00–M’03) received the B.S. and M.S. degrees from Tongji University, Shanghai, China, and the Ph.D. degree from the University of Minnesota, Minneapolis, all in electrical engineering, in 1997, 1999, and 2003, respectively.

She is with DSP Solutions R & D Center, Texas Instruments Incorporated, Dallas, TX. Her current research interests are efficient very large-scale integrated architecture designs for various building blocks in communication systems.

Keshab K. Parhi (S’85–M’88–SM’91–F’96) received the B.Tech., M.S.E.E., and Ph.D. degrees from the Indian Institute of Technology, Kharagpur, India, the University of Pennsylvania, Philadelphia, and the University of California at Berkeley, in 1982, 1984, and 1988, respectively.

Since 1988, he has been with the University of Minnesota, Minneapolis, where he is currently a Distinguished McKnight University Professor in the Department of Electrical and Computer Engineering. His research addresses VLSI architecture design and implementation of physical layer aspects of broadband communications systems. He is currently working on error-control coders and cryptography architectures, high-speed transceivers, ultra wideband systems, and quantum error-control coders and quantum cryptography. He has published over 350 papers, has authored the textbook *VLSI Digital Signal Processing Systems* (New York: Wiley, 1999) and coedited the reference book *Digital Signal Processing for Multimedia Systems* (New York: Marcel Dekker, 1999).

Dr. Parhi is the recipient of numerous awards including the 2003 IEEE Kiyo Tomiyasu Technical Field Award, the 2001 IEEE W.R.G. Baker prize paper award, and a Golden Jubilee award from the IEEE Circuits and Systems Society in 1999. He has served on Editorial Boards of IEEE Transactions on VLSI Systems, IEEE Transactions on Signal Processing, IEEE Signal Processing Letters, IEEE Transactions on Circuits and Systems, and IEEE Transactions on Circuits and Systems—II, currently serves on Editorial Boards of the *IEEE Signal Processing Magazine* and *Journal of VLSI Signal Processing Systems*, and is the current Editor-in-Chief of the IEEE Transactions on Circuits and Systems—I: Regular Papers, for 2004–2005. He served as Technical Program Cochair of the 1995 IEEE VLSI Signal Processing Workshop and the 1996 ASAP Conference, and as the General Chair of the 2002 IEEE Workshop on Signal Processing Systems. He was a Distinguished Lecturer for the IEEE Circuits and Systems Society from 1997 to 1999.