

Architecture-Aware LDPC Code Design For Multi-Processor Software Defined Radio Systems

Yuming Zhu, *Member, IEEE*, and Chaitali Chakrabarti, *Senior Member, IEEE*

Abstract— This paper presents a general procedure for designing low density parity check (LDPC) codes for multi-processor software defined radio platforms. Our approach is to design the LDPC code to match the constraints imposed by the hardware architecture, without compromising on the communication performance. The proposed architecture-aware code design procedure involves feature identification, code construction and verification. We demonstrate the effectiveness of our procedure for three cases. If the local memory of the processor is small and it can only process one horizontally partitioned sub-matrix at a time, we show how the code can be constructed so that the traffic to the global memories is reduced by 2X. If the row weight of the matrix is large and each processor processes a vertically partitioned sub-matrix, we show how the matrix can be constructed so that the computational load is evenly distributed among the processors. If the processors have no storage capability and all data is stored in global memories, then for the case when all traffic is through a multi-stage interconnection network, we show how code construction can be used to significantly reduce the number of routing conflicts. In all three cases, the resulting LDPC codes can not only be mapped efficiently onto the multi-processor platform but also have very good frame error performance.

I. INTRODUCTION

Low-Density Parity-Check (LDPC) [1] codes are linear block codes with sparse parity check matrices. Their asymptotic performance can be as close to one tenth dB away from the Shannon limit [2]. Another advantage of LDPC codes is that the decoding algorithm is inherently parallel and so a wide variety of hardware implementations can be derived to exploit this feature. Because of their extraordinary performance, LDPC codes have been adopted in the physical layer of many recent communication standards such as DVB-S2, 10GBase-T, 802.16e and 802.11n.

Recently, there has been a lot of work done on LDPC decoder implementation [3], [4], [5], [6], [7], [8], [9], [10]. The fully parallel ASIC implementation of LDPC decoder presented in [3] achieves very high throughput with large area overhead. Starting with the pioneering work in [5], many of the work consider decoder complexity in the design of the LDPC codes. For instance, [7] presented an architecture-aware LDPC code that utilizes pseudo-permutation sub-matrices and views the LDPC code as a concatenation of super-codes. This facilitates Turbo decoding like operations in the LDPC

decoder and results in an implementation that is significantly simpler in terms of both interconnection network complexity and memory requirement. This has been demonstrated in a real chip implementation in [11]. Co-design approaches have also been proposed in [9], [10]. The block LDPC design method presented in [9] jointly considers code design, decoder design and encoder design. It is shown that the encoder and decoder complexity can be reduced without significant performance loss by using circulant sub-matrices in the parity check matrix. In [10], we showed that by including weight-2 circulant sub-matrices in the parity check matrix, we can achieve higher throughput without any performance loss. While all these architecture-aware LDPC code design studies are targeted for ASIC implementations, there are LDPC decoding studies for multi-processors as well [4]. For instance, randomly constructed LDPC codes are considered in [4]. The focus there is on balancing the computation among the processors and reducing communication cost by clustering computation.

In this paper, we consider the problem of systematically designing LDPC codes that exploit the architectural features of multi-processor architectures, such as those used in existing SDR platforms. This is an extension of the work presented in [12] and [13]. Fig. 1 shows the design flow that we first proposed in [12]. First, the code features based on (i) system specifications, such as frame error rate (FER) performance, codeword size, code rate, and (ii) architectural features of the target platform, such as the number of processing units (PU), the width of the single-instruction multiple-data (SIMD) unit and arithmetic complexity, are obtained. The LDPC code is then constructed based on code features, which include number of block columns, row weight and level of parallel processing. The last step is verifying that the LDPC code meets both the system constraints and the architectural constraints. In some cases, several iterations may be necessary to ensure that all the constraints are met. Note that the code design process is done off-line and does not affect the run-time performance.

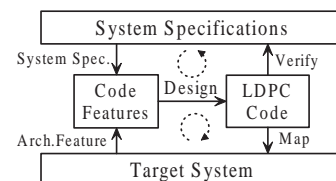


Fig. 1. Architecture-aware LDPC code design flow.

We demonstrate the effectiveness of the procedure for three cases. In all cases, we ensure that the FER performance is not compromised. First, we present the case when the local mem-

Copyright (c) 2008 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.

Yuming Zhu is with the DSPS R&D Center, Texas Instruments, Dallas, TX, 75243, USA (email: yuming.zhu@ieee.org). This work is based on his Ph.D research at Arizona State University.

Chaitali Chakrabarti is with Electrical Engineering Department of Arizona State University, Tempe, AZ, 85287, USA (email: chaitali@asu.edu).

ory in the PU is not large enough to support the computations on the whole parity check matrix. We present a scheme where the parity check matrix is partitioned horizontally into super-codes [7], [14] (Section IV). We find that super-code decoding introduces degree-one nodes which hinder its performance. So we derive code design constraints that minimize the number of such nodes. Our simulation results show that the LDPC codes constructed with these constraints converge twice as fast compared to the non-optimized codes. This reduces the memory traffic by a factor of 2 which result in the decoding throughput increasing by a factor of 2.

Next we consider the case when the parity check matrix has a large row weight and implementation on a single PU results in low throughput. We present a scheme where the parity check matrix is partitioned vertically and each of the sections is mapped to a PU in a multi-processor architecture (Section V). We derive code design constraints that help in distributing the computational load evenly among the PUs.

Finally, we present an extreme case where the local memory in each PU is relatively very small and the bulk of the information is stored in the global (shared) memories. Thus there is significant traffic between the PUs and the global memory, and the throughput is closely related to the number of routing conflicts. We demonstrate the LDPC code design procedure for a SDR platform where the PUs communicate to the global memory units via a multi-stage interconnection network (MIN). The combinations of routing paths that cause conflicts in the MIN are identified and mechanisms to avoid them are translated into constraints for the code construction step. The resulting LDPC code can be mapped very efficiently onto the SDR platform and has very high decoding throughput.

The rest of this paper is organized as follows. Section II provides a brief review of LDPC codes. Section III outlines the SDR platform and mapping of LDPC decoding onto the platform. Sections IV and V describe the schemes based on horizontal partitioning and vertical partitioning of the parity check matrix. Section VI describes the scheme that has been optimized for MIN-based interconnection. Section VII concludes the paper.

II. BACKGROUND

A. Basics of LDPC Codes

LDPC codes can be represented by bipartite graphs in which a set of nodes (variable nodes) corresponds to the elements of a codeword and another set of nodes (check nodes) correspond to the parity-check constraints of the code. Figure 2 shows the parity check matrix and the corresponding bipartite graph of a very short (2,3)-regular LDPC code. It is *regular* since each variable node is connected to 2 check nodes and each check node is connected to 3 variable nodes. LDPC codes that do not have the regularity property are called *irregular*. Irregular LDPC codes usually achieve better performance than the regular LDPC codes [15], [2], and thus are considered here.

The performance of the LDPC codes is often predicted in terms of the *ensemble* average of codes, a consequence of the random choice of edges between check nodes and variable nodes for a given length and a given degree distribution.

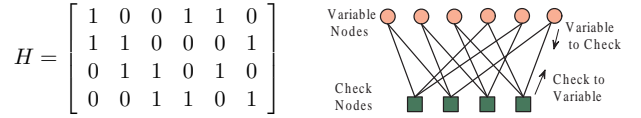


Fig. 2. Bipartite graph of a (2,3)-regular LDPC code.

The optimal degree distribution for asymptotic cases (infinite codeword size, infinite iteration number) can be computed based on density evolution (DE) [16]. An online computing program based on DE is available in [17]. Let d_{min}^v (d_{max}^v) be the minimum (maximum) variable node degree and let d_{min}^c (d_{max}^c) be the minimum (maximum) check node degree. A study of the asymptotic performance shows the following features for degree distribution:

- $d_{min}^v \geq 2$.
- The check node degrees for optimal distribution are limited to only a few (one or two) values close to each other. Thus a simple notation $d_c \triangleq d_{max}^c \cong d_{min}^c$ is used to represent the maximum (average) check node degree.

B. The Iterative Decoding Algorithms

LDPC decoding is done by either the bit flipping hard-decision algorithm [1], or by the soft-decision iterative decoding algorithms such as the belief propagation (BP) algorithm and variations of Min-Sum algorithm. All of them involve two kinds of operations: variable node processing (VP) and check node processing (CP). The operations for soft-decision iterative decoding are summarized in Alg. 1.

Algorithm 1 Layered iterative decoding algorithm

- 1: {Initialization:}
Set iteration number $i = 0$, and for $\forall n \in [1, N], \forall m \in M(n)$ set $E_{m,n} = 0, L_n = I_n$ (1)
 - 2: **while** $i \leq i_{max}$ and parity check equation not met **do**
 - 3: **for** $s = 1$ to m_b **do**
 - 4: **for** $t = 1$ to Z **do**
 - 5: $m = (s - 1) \times Z + t$ {Index of check node}
 - 6: **for all** variable node $n \in N(m)$ **do**
 - 7: {Variable Node Processing (VP):}
 $L_{n,m} = L_n - E_{m,n}$ (2)
 - 8: {Check Node Processing (CP):}
 $E_{m,n}^{New} = f(L_{n',m} |_{n' \in S \subseteq N(m)})$ (3)
 - 9: {Variable Node Update:}
 $L_n^{New} = L_{n,m} + E_{m,n}^{New}$ (4)
 - 10: **end for**
 - 11: **end for**
 - 12: **end for**
 - 13: **end while**
 - 14: Output the soft/hard decision of $L_n, \forall n \in [1, N]$.
-

In the description in Alg. 1, I_n is the intrinsic information of variable node n from the received signal, $E_{m,n}$ and $E_{m,n}^{New}$ are the extrinsic information from check node m to variable node n , $L_{n,m}$ is the information from variable node n to check node m , L_n is the total information about the variable node n .

All the information are in form of log-likelihood ratio (LLR). $N(m)$ is the set of variable nodes which is connected with check node m in the bipartite graph. Similarly, $M(n)$ is the set of check nodes which is connected with variable node n . The different decoding algorithms differ in how the function $f(\cdot)$ is evaluated. For BP and Min-Sum algorithm, the set $S = N(m) \setminus \{n\}$. The $f(\cdot)$ function in Eqn. (3) for several common algorithms are listed below:

For Min-Sum algorithm:

$$E_{m,n}^{New} = \prod_{n' \in N(m) \setminus \{n\}} \text{sign}(L_{n',m}) \cdot \min_{n' \in N(m) \setminus \{n\}} |L_{n',m}| \quad (5)$$

For BP algorithm:

$$E_{m,n}^{New} = \prod_{n' \in N(m) \setminus \{n\}} \text{sign}(L_{n',m}) \cdot \Psi \left(\sum_{n' \in N(m) \setminus \{n\}} \Psi(L_{n',m}) \right) \quad (6)$$

where the function $\Psi(x) = -\log(|\tanh(x/2)|)$.

The LLR information (L_n) is the sum of the intrinsic and extrinsic information of the variable node n

$$L_n = I_n + \sum_{m \in M(n)} E_{m,n} \quad (7)$$

After the final iteration, a hard decision based on L_n can be made. The number of iterations required for decoding varies with the SNR. In practice, the maximum number of iterations can be set to 10 - 15. Early termination algorithms are often adopted to stop the decoding process after all the parity check equations have been met.

C. Finite Length LDPC code design

Unlike the asymptotic cases, the performance of finite length LDPC code is greatly affected by the code structure as well as the degree distribution. This is because the finite length LDPC codes are no longer cycle-free [18], [19]. Many different criteria have been proposed to optimize the performance of finite length LDPC codes, for example, large girth (length of the smallest cycles in the bipartite graph) [20], [21], selective avoidance of small cycles [22] and optimization based on stopping set [23] or trapping set [24].

LDPC codes with structured sub-matrices [25] are widely used in practice to reduce implementation overhead and is the focus of this paper. The parity check matrix of size $M \times N$ can be represented as a block matrix H_b of size $m_b \times n_b$; each element in H_b matrix is a $Z \times Z$ circulant shifted identity matrix or zero matrix, where Z is an integer. Each row of the block parity matrix (which is equivalent to Z rows with $n_b \cdot Z$ elements per row of the original parity check matrix) is referred to as *block row*. Similarly, each column of the block parity matrix (which is equivalent to Z columns with $m_b \cdot Z$ elements per column) is referred to as *block column*. The non-zero item in H_b is written as $H_b(i, j) = I_x$, where I_x is the circulant identity matrix with shift value of x . For simplicity, we use '1' or '0' to represent the I_x or zero matrix.

To design finite length LDPC codes that can be efficiently implemented on the target multi-processor architecture, we utilize a design process that combines the optimal degree distribution for asymptotic performance, characteristics of the

structured sub-matrices, finite length code optimization criteria and architectural constraints such as number of processors and the width of the SIMD unit. The design process is performed at two levels: the block matrix level design which constructs the H_b matrix, and the sub-matrix level design which assigns shift values to the sub-matrices.

The steps for designing LDPC codes based on the code features can be summarized as follows:

- 1) First, obtain the code features based on system requirement and architectural specifications. The code features include: n_b , m_b , Z , d_c , d_{min}^v , d_{max}^v , level of parallel processing, etc.

While system requirements determine the code rate R and the range of codeword size ($n_b \times Z$), the SIMD width and number of PUs are used to determine the extent of parallelism and the row weight (d_c). The specific decoding algorithm is selected based on the arithmetic operation supported by the processor.

- 2) Compute the degree distribution based on the targeted channel condition, code rate (obtained from m_b and n_b), d_c , d_{min}^v and d_{max}^v . For the asymptotic case, the optimal degree distribution can be computed based on density evolution [16]. To use these results for finite sized blocks, adjustments have to be made such as rounding off the actual node numbers to integers.
- 3) Construct the H_b matrix based on the degree distribution result in previous step. The construction procedure can be briefly written as:

(i) Initialize the H_b of size $m_b \times n_b$ with '0'.

(ii) Randomly place d_c '1's in each row.

(iii) Move the '1's along rows to satisfy the variable node degree distribution.

(iv) Switch columns to facilitate further modification. For example, some lower degree columns are moved to the right side of the H_b matrix before step (v) in order to make the encoder efficient [26].

(v) Move the '1's in the H_b matrix in pairs to meet additional constraints that have been imposed due to architectural or performance considerations. The pair-wise movement is such that if a '1' is moved from location (i, j) to location (k, j) , then another '1' needs to move from location (k, l) to location (i, l) , provided that location (k, l) had a '1' and location (i, l) had a '0'. Note that this pair-wise movement of '1's does not affect the check node or variable node distributions.

(vi) Output the H_b matrix.

Note that steps (iv) and (v) may have to be processed multiple times until all the constraints have been met or the maximum iteration number has been reached.

- 4) Assign shift values to all the non-zero elements in H_b matrix. This is done by assigning shift values based on GF(Z) [25] or assigning shift values in a more flexibly way such that the trapping set is minimized and small cycles are avoided selectively [22], [24].

A summary of the key notations that are used in this paper is included in Table I for easy reference.

TABLE I
DEFINITIONS FOR SOME KEY NOTATIONS

Notation	Definition
C	Number of super-codes
d_c	Check node degree
d_v	Variable node degree
$E_i(X)$	i -th level ENS of a node X in MIN
$E_{m,n}$	Extrinsic info. from check node m to variable node n
$G_i(X)$	i -th level neighborhood group for a node X in MIN
H	Parity check matrix
H_b	Block parity check matrix, each element is a $Z \times Z$ matrix
H_b^i	Block parity check matrix for i -th super-code
$H_{b_j}^i$	j -th vertically partitioned H_b matrix
I_n	Intrinsic information of variable node n
J	Number of vertical partitions
L_n	LLR for a variable node n
$L_{n,m}$	Information from variable node n to check node m
M	Number of rows in H matrix (number of check nodes)
m_b	Number of block rows in H_b
$M(n)$	Set of check nodes connected with variable node n
N	Number of columns in H matrix (number of variable nodes)
n_b	Number of block column in H_b matrix
$N(m)$	Set of variable nodes connected with check node m
P	SIMD width in the PU
Q_1	Number of inner iterations for super-code based decoding
Q_2	Number of outer iterations for super-code based decoding
Q_m^j	Partial sum of j -th PU in vertically partitioned decoding
$Q_{m,n}^j$	CP result of j -th PU in vertically partitioned decoding
$r(d_c)$	Memory required by check-to-variable info. of a row
W	Bit precision of the values in PU
Z	Size of the regularly structured sub-matrix in H_b
$f(\cdot)$	Function used to calculate $E_{m,n}^{New}$ based on $L_{n,m}$
$g(\cdot)$	Function used to calculate $Q_{m,n}^j$
$\zeta(\cdot)$	Function used to calculate Q_m^j
$\xi(\cdot)$	Function to calculate $E_{m,n}^{New}$ based on $Q_{m,n}^j$ and Q_m^j

III. LDPC DECODING ON AN SDR PLATFORM

A. SDR platform

There are many different definitions of SDR systems. In the context of this paper, it is a hardware platform that processes the physical layer of multiple protocols. We only consider the digital processing engine, thus ADC/DAC and RF parts are not studied here.

A typical SDR platform consists of multiple processing units (PU) and multiple global memory units as shown in Fig. 3. The PUs and the global memory units communicate with each other via an interconnection network. The interconnection network could be one or more shared buses or even a multistage interconnection network. Each PU consists of a local memory, a processing element (PE), which consists of a scalar unit and a SIMD unit, and an application specific element (ASE). The combination of a scalar unit and a SIMD unit enables a large class of algorithms to be mapped very efficiently as shown in [27]. The ASEs are typically included to enhance the performance of some target protocols. For example, the ASE could be an optimized shuffle network as in [27] or a sorter. The local memory of a PU is typically quite small – of the order of a few kilobytes. The software control unit coordinates all the PUs in the system.

B. LDPC decoding

We first consider the case where all the information related to LDPC decoding can be stored in the local memory of a PU.

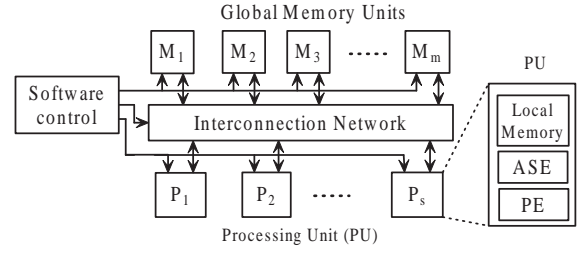


Fig. 3. Software defined radio (SDR) platform.

This puts an obvious limitation on the size of the codeword or equivalently, the size of the parity check matrix that can be processed.

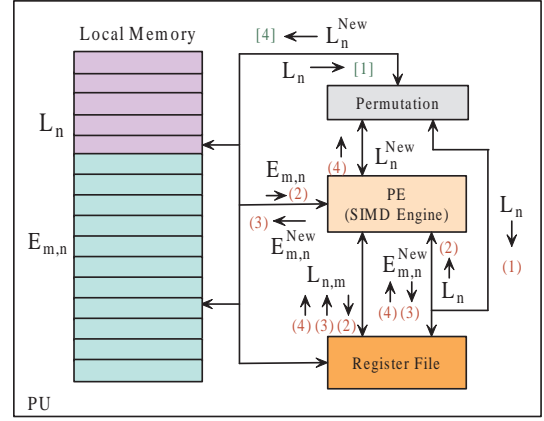


Fig. 4. LDPC decoding data flow on a single PU in the SDR platform.

Assume that the PE is a SIMD engine with P slices. For example, the PE in [27] has $P = 32$ slices. Fig. 4 describes such a PU. Each SIMD slice can efficiently compute addition/subtraction and comparison. For LUT based implementations that are required for BP based decoding, approximate computations can be used [7]. The local memory in the PU stores the intrinsic and extrinsic information. The local memory and the register file feed data to the P slices. In addition, there is a permutation unit that is used to permute P values to/from the P slices of the SIMD engine.

We assume the scheduling is in check node order. Each SIMD slice is responsible for the computations in a check node. In each step, only the check-to-variable or variable-to-check information corresponding to the scheduled check nodes are updated. The general process for processing check nodes in a single PU is shown in Fig. 4. The decoding includes the following steps:

- 1) Load L_n from memory, permute L_n to the correct slice and store to register file;
- 2) Load $E_{m,n}$ from memory, load L_n from register file, calculate $L_{n,m}$ (VP), store the $L_{n,m}$ to register file;
- 3) Load $L_{n,m}$ from register file, calculate $E_{m,n}^{New}$ (CP), store $E_{m,n}^{New}$ to memory and register file;
- 4) Load $E_{m,n}^{New}$ and $L_{n,m}$ from register file, calculate L_n^{New} , permute L_n^{New} to its original order and store L_n^{New} to memory.

These steps have been marked along the edges in Fig. 4. The round parenthesis denotes the data that can directly be fed

to the SIMD slices and square parenthesis denotes the data that is organized for local memory and require permutation before/after being processed by the SIMD slices.

1) *Number of Cycles*: We estimate the number of cycles that are required for the decoding steps outlined above. Each of the steps (except Step 3) requires d_c cycles. This is because the steps are repeated for each of the d_c non-zero elements in a row. The CP operation in Step 3 (described in Eqn.(5)-(6)) requires the all-but-one operation which can be implemented very efficiently in ASICs using adder-trees and XOR-trees. In an SDR platform, $2d_c$ cycles are required to compute the all-but-one operation. Thus the total number of cycles to process P check nodes using the SDR platform is $5d_c$ cycles, ignoring the overhead due to the pipelines. The precise cycle count is platform dependent. For example, the permutation step that is used to convert the L_n or L_n^{New} values between variable node order and check node order, can be done in one cycle in [27] but may not be true for other platforms.

2) *Memory Requirement*: Because the precision of operands in processors are limited to a few choices (for example, 8 bits, 16 bits, 32 bits), we will evaluate the memory requirement in terms of number of values in the SDR platform. For BP and Min-Sum (and its variation) algorithms, 8 bits are more than sufficient to achieve decent performance in hardware [6], [7].

The information that needs to be stored for BP decoding include the LLR value L_n and the check-to-variable information ($E_{m,n}$). The total memory requirement is

$$\text{MEM}_{Total} = N + M \cdot d_c \quad (8)$$

where the first term (N) is equal to the number of nodes and the second term is equal to the total number of edges in the bipartite graph.

If the Min-Sum algorithm is used then the memory requirement can be reduced. For each check node, it is sufficient to only store the minimum (together with its position), the second minimum, and the signs for all check-to-variable information. Because the sign information can be packed together to save memory, the total memory required for iterative Min-Sum decoding is

$$\text{MEM}_{Total} = N + M \cdot \left(3 + \lceil \frac{d_c}{W} \rceil \right) \quad (9)$$

where W is the bit width of each word in memory. We can introduce a new notation $r(d_c)$ to represent the memory required by check-to-variable information of each row.

$$r(d_c) = \begin{cases} d_c, & \text{BP algorithm} \\ 3 + \lceil \frac{d_c}{W} \rceil, & \text{Min-Sum algorithm} \end{cases} \quad (10)$$

For a SIMD architecture with P slices, the local memory is laid out such that each memory word contains P values. The local memory required for decoding is

$$\text{MEM}_{Total}^{SIMD} \cong N_p \cdot (n_b + m_b \cdot r(d_c)) \quad (11)$$

where $N_p = \lceil \frac{N}{P} \rceil$.

The LDPC codes require large codeword sizes to achieve good performance. For example, DVB-S2 standard specifies a

maximum size of 16200 bits, the new Chinese DTV standard DMB-T specifies a maximum size of 7493 bits. On the other hand, the local memory size of PU is limited. For example, the SDR architecture proposed in [27] has only 8KB+4KB local memory in each PU (which is significantly less than the memory in general purpose processors). Algorithmic innovations are required to handle such cases.

IV. LDPC CODE DESIGN FOR HORIZONTALLY-PARTITIONED MATRIX

In this section, we study the case when only a single PU can be assigned for LDPC decoding. The PU has P SIMD slices (as described in Section III-A) and limited local memory. To satisfy the memory constraints, the parity check matrix can be partitioned horizontally into super-codes [7], [14], and the PU operates on the super-codes one after the other. Super-code based decoding could results in degradation of FER performance if not careful. We show how to avoid this from happening by proper code construction.

A. Super-code based LDPC decoding

Fig. 5 shows the decomposition of a LDPC code into two super-codes. The parity check matrices of super-codes (H_b^1 and H_b^2) are sub-matrices of the original H_b matrix.

Fig. 5. Decomposition of the parity check matrix H_b into two super-codes with block parity check matrices H_b^1 and H_b^2 .

LDPC decoding based on super-codes is summarized in Alg. 2. There are two levels of iteration in the decoding – inner and outer iteration. Each super-code is decoded with fixed number of iterations (inner iteration). The output LLR values (L_n) from one super-code serve as the input to the next super-code. The super-codes are decoded one after another and in an iterative fashion (outer iteration). Let C denote the number of super-codes, each of which has a block parity check matrix H_b^i , $i \in [1, C]$. The outer iteration number is Q_2 and the inner iteration number of the i -th super-code is Q_1^i .

The AA-LDPC codes in [7] is equivalent to the case $C = m_b$, $Q_1^i = 1$, $\forall i \in [1, C]$. The scheme shown in [14] is equivalent to the case $C = 2$ and $Q_1^i = Q_1^j$, $\forall i, j \in [1, C]$. Our method is more general and provides more choices for the LDPC code designer.

B. Memory Analysis

In this section, we analyze the memory requirements for super-code based BP decoding. The procedure for the Min-Sum based decoding is identical and has not been included here. In super-code based BP decoding, the local memory requirement is smaller than that given in Eqn. (11). Here the

Algorithm 2 Super-code based iterative decoding

```

1: {Initialization:}
   Set iteration number  $i = 0$ , and for  $\forall n \in [1, N], \forall m \in M(n)$  set  $E_{m,n} = 0, L_n = I_n$ 
2: while  $i \leq Q_2$  and parity check equation not met do
3:   for  $j = 1$  to  $C$  do
4:     Load Check-to-Variable information  $E_{m,n}$  for all edges in  $H_b^j$ 
5:     for  $k = 1$  to  $Q_1^j$  do
6:       Perform one iteration of BP decoding for  $H_b^j$ 
7:     end for
8:     Save updated  $E_{m,n}$  values to global memory.
9:   end for
10: end while
11: Output the soft/hard decision of  $L_n, \forall n \in [1, N]$ .

```

local memory keeps a complete copy of all L_n values and only the $E_{m,n}$ values for the super-code to be decoded; the global memory stores all the $E_{m,n}$ values. There is another possible storage scheme described in [13] which stores only a subset of the L_n values. It results in smaller local memory but larger global memory and increased global memory traffic. Here the memory traffic is defined as the clock cycles used to transfer data among memory and PU.

The memory requirement is summarized below:

$$\text{MEM}_{Local}^{SIMD} = N_p \cdot \max_{i \in [1, C]} (n_b + m_b^i \cdot r(d_c)) \quad (12)$$

$$\text{MEM}_{Global}^{SIMD} = N_p \cdot m_b \cdot r(d_c) \quad (13)$$

where $N_p = \lceil Z/P \rceil$, m_b^i is the number of block rows in H_b^i . The m_b^i values are usually set equal for all super-codes, i.e., $m_b^i = \lceil \frac{m_b}{C} \rceil$, to minimize the local memory.

The local memory requirement reduces with the increase in the number of super-codes, as expected. However, the reduction in local memory comes with the price of increase in memory traffic between the global memory and the PU.

The number of access cycles for global memory is

$$T_{mem} = 2 \sum_{i=1}^C m_b^i \cdot r(d_c) \cdot Q_2 \cdot T_1 = 2m_b \cdot r(d_c) \cdot Q_2 \cdot T_1 \quad (14)$$

where T_1 is the number of cycles for transmitting Z values between the global memory and the local memory.

The number of decoding cycles is

$$T_{dec} = \left(\sum_{i=1}^C m_b^i \cdot Q_1^i \right) \cdot Q_2 \cdot T_2 \quad (15)$$

where T_2 is the number of cycles for one decoding iteration (including VP and CP operation) for a block row in H_b . The Q_1^i can be chosen independently for each super-code. For example, it can be chosen proportional to m_b^i . However, for simplicity of control and performance comparison, we choose $Q_1^i = Q_1, \forall i \in [1, C]$. In this case, $T_{dec} = m_b \cdot Q_1 \cdot Q_2 \cdot T_2$.

A comparison between T_{mem} and T_{dec} shows that while T_{mem} is proportional to the outer iteration number Q_2 , T_{dec} is proportional to the product of the outer and inner iteration

numbers, $Q_1 \cdot Q_2$. Thus we need to reduce Q_2 to minimize the memory traffic and keep certain number of $Q_1 \cdot Q_2$ to maintain the FER performance. In the following sections, we describe how this can be achieved by appropriate code construction.

C. Study of FER Performance

In [14], the authors proposed a method where all super-codes have the same degree distribution and the degree distribution of the super-codes is derived directly from the overall degree distribution. The problem with this method is that it results in a large number of degree one variable nodes in the super-codes. It is easy to verify that the variable-to-check information from such variable nodes do not get updated during the inner iterations. Thus it does not help in the convergence process within one super-code.

For example, Fig. 5 shows H_b matrix generated by the block level random placing step in the LDPC code construction procedure (Section II-C). A blind construction of super-codes by directly partitioning an existing parity check matrix is likely to create many degree-one variable nodes in super-codes, as shown in the shaded boxes in Fig. 5. As a result, the performance of super-code based decoding is dominated by the number of outer iterations Q_2 [13]. However, increasing Q_2 increases the memory traffic and is detrimental to the overall throughput. In the next section, we show how to unleash the potential of super-code based decoding with small Q_2 . We achieve this by constructing the codes such that the number of degree-one nodes in super-codes is reduced.

D. Code constraints

In order to reduce the number of degree one nodes in super-code based decoding, we impose an additional constraint in the LDPC code construction procedure (Section II-C), which is given as follows:

- Divide H_b into C parts, each of which corresponds to a super-code. The number C is chosen so that the local memory is able to accommodate all the L_n and $E_{m,n}$ values of one super-code.
- If $m_b^i \geq 2$ and there exists degree-one nodes in the i th super-code, perform exchange operation to make it at least degree 2 or degree 0. The only exception is where the degree-one node is critical in maintaining a certain structure of the parity check matrix.
- Reduce the outer iteration number Q_2 till the FER performance is no longer acceptable.

In step (3v) of the algorithm in Section II-C, the exchange procedure removes degree-one variable nodes in super-codes except in some special cases where they are kept to preserve the structure of the parity check matrix. If all the degree-one variable nodes in super-codes can not be removed, the variable nodes with lower degree in final codeword (especially those with degree 2 or 3) should be handled with higher priority.

Fig. 6 shows the H_b matrix of Fig. 5 after undergoing the exchange procedure. Almost all the degree-one nodes in super-codes have been removed. There are only two degree-one nodes left in H_b , which are purposely kept to maintain the lower triangle shape of the H_b matrix. The FER performance of this new code is no longer dependent on Q_2 .

Algorithm 3 Iterative decoding based on vertically partitioned H_b

```

1: {Initialization:}
   Set iteration number  $i = 0$ , and for  $\forall n \in [1, N], \forall m \in M(n)$  set  $E_{m,n} = 0, L_n = I_n$ 
2: while  $i \leq i_{max}$  and parity check equation not met do
3:   for  $s = 1$  to  $m_b$  do
4:     for  $t = 1$  to  $Z$  do
5:        $m = (s - 1) \times Z + t$  {Index of check node}
6:       {First Round: Calculate partial update}
7:       for all vertically-partitioned  $H_{bv}^j$  do
8:         for all variable node  $n \in N^j(m)$  do
9:           {Variable Node Processing (VP):}
10:             $L_{n,m} = L_n - E_{m,n}$  (16)
11:           {Partial Check Node Processing:}
12:             $Q_{m,n}^j = g(L_{n',m} |_{n' \in S \subseteq N^j(m)})$  (17)
13:          end for
14:          {Update Partial Sum: }
15:            $Q_m^j = \zeta(Q_{m,n} |_{n \in N^j(m)})$  (18)
16:        end for
17:        {Second Round: CP and variable-update based on partial update }
18:        for all vertically-partitioned  $H_{bv}^j$  do
19:          for all variable node  $n \in N^j(m)$  do
20:            {Check Node Processing (CP):}
21:             $E_{m,n}^{New} = \xi(Q_{n,m}^j, Q_m^{j'} |_{j' \neq j})$  (19)
22:          {Bit Update:}
23:            $L_n^{New} = L_{n,m} + E_{m,n}^{New}$  (20)
24:        end for
25:      end for
26:    end for
27:  end while
28: Output the soft/hard decision of  $L_n, \forall n \in [1, N]$ .

```

$$\xi(Q_{m,n}^j, Q_m^{j'}) = \text{sign}(Q_{m,n}^j) \cdot \prod_{j' \neq j} \text{sign}(Q_m^{j'}) \cdot \min_{j' \neq j} \left\{ |Q_{m,n}^j|, |Q_m^{j'}| \right\} \quad (23)$$

For BP algorithm:

$$g(L_{n',m}) = \prod_{n' \in N^j(m) \setminus \{n\}} \text{sign}(L_{n',m}) \cdot \sum_{n' \in N^j(m) \setminus \{n\}} \Psi(L_{n',m}) \quad (24)$$

$$\zeta(L_{n',m}) = \prod_{n' \in N^j(m)} \text{sign}(L_{n',m}) \cdot \sum_{n' \in N^j(m)} \Psi(L_{n',m}) \quad (25)$$

$$\xi(Q_{m,n}^j, Q_m^{j'}) = \text{sign}(Q_{m,n}^j) \cdot \prod_{j' \neq j} \text{sign}(Q_m^{j'}) \cdot \Psi \left(|Q_{m,n}^j| + \sum_{j' \neq j} |Q_m^{j'}| \right) \quad (26)$$

It is easy to verify that the algorithm is equivalent to the original Min-Sum and BP algorithm. Furthermore, the memory

requirement for a PU is given by

$$\text{MEM}_{Local}^{SIMD} = N_p \cdot \max_{j \in [1, J]} \left(\sum_{s=1}^{m_b} r(\text{NUM}^j(s)) + n_b^j \right) \quad (27)$$

Note that there are several other LDPC decoding schemes that vertically partition the H_b matrix for different purpose [28], [29], [30]. In [28], the authors also proposed to partition the H_b matrix vertically to reduce the decoder complexity. The decoding algorithm in [28], which is based on a variation of the BP algorithm, uses heuristic multiplicative factors and results in performance loss. While the decoding is sub-optimal, in [28], the inter-processor communication is significantly reduced. In [29], the H matrix is vertically partitioned into groups and the groups updated sequentially from left to right. Our work differs from the scheme in [29] in that the update of vertically partitioned groups are performed simultaneously in a cooperative way. Vertical partitioning of H matrix has been used in a recently published paper [30] to achieve multiple code rates.

B. Code Constraint

A closer look at Alg. 3 indicates that although the different PUs work independently, they need to exchange the partial sum value Q_m^j for each check node. Thus the throughput performance is determined by the partition with the largest row weight. This leads us to establishing the following LDPC code requirement: all partitions, $H_{bv}^j, \forall j = 1, \dots, J$ should have approximately the same row weight. Besides, H_{bv}^j should have approximately the same number of block columns, which will minimize the memory requirement of each PU.

To characterize this requirement quantitatively, we introduce a parameter called the unbalance factor (UF).

$$UF = \sum_{s=1}^{m_b} \left(\max_j \{ \text{NUM}^j(s) \} - \left\lceil \frac{d_c}{J} \right\rceil \right) \quad (28)$$

Essentially UF is the number of extra cycles required for decoding a specific partition compared to the optimal partition (which corresponds to $UF = 0$). So during code design, the distribution of 1s should be such that UF is as small as possible. This is achieved by applying the constraint to minimize UF defined in Eqn. (28) in Step 3(v) in the code construction procedure in Section II-C. Specifically, the exchange procedure in Step 3(v) would switch block columns of the original H_b matrix such that UF is as small as possible.

C. Performance Analysis

FER: There is no change in the FER performance since the proposed algorithm is equivalent to the original BP or Min-Sum algorithms.

Throughput: In order to evaluate the increase in the throughput performance, we calculate the number of cycles that are required in this scheme. Partial check node processing is equivalent to the original CP computation (outlined in Section III-B) with $\lceil \frac{d_c}{J} \rceil$ non-zero elements. Another $\lceil \frac{d_c}{J} \rceil + J$ cycles is required for the second round of operation during CP processing. This overhead is due to the J partial sum values

that have to be sent to all the PUs and the updated Q_m^j values that we assume can be broadcast to the other PUs in one cycle. For a simple bus interconnection network, it takes J cycles to broadcast the results. Thus the number of cycles is given by

$$\text{Number of cycles} = \begin{cases} 5 \cdot d_c, & J = 1 \\ 6 \cdot \lceil \frac{d_c}{J} \rceil + J, & J > 1 \end{cases} \quad (29)$$

Figure 8 shows the throughput of vertical partition schemes for different d_c values. In each case, the throughput values have been normalized to the $J = 1$ case. We see that for small d_c values, the throughput increases sub-linearly with J , while for large d_c values, the throughput increases linearly with J . Thus this method can be used to increase the throughput for matrices with large d_c values by effectively distributing the computational load among the PUs.

The code with codeword size 6176, code rate 3/4, $Z = 193$ and $d_c=13$ was mapped onto the 4 SODA PUs. The resulting throughput was 24.8 Mbps which is lower than the 42.8 Mbps that is achievable if horizontal partitioning is used and all 4 PUs are activated. However, for codes with larger d_c values, the relative difference in throughput is very small and is easily out weighted by the advantage of the smaller memory requirement in PUs.

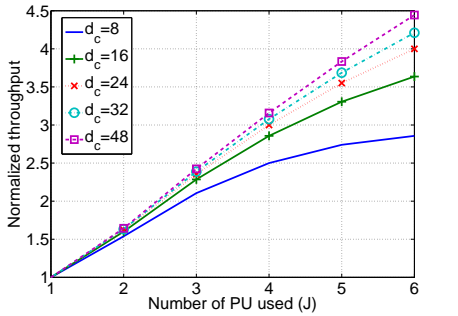


Fig. 8. Normalized throughput as a function of number of PUs for vertically partitioned matrix.

VI. LDPC CODE DESIGN FOR MIN BASED SYSTEMS

When the LDPC decoder is required to support very large codeword sizes and the PU local memory that is required for processing vertically partitioned or horizontally partitioned parity check matrix is not sufficient, we consider a different approach. Assume that the whole SDR platform can be utilized for LDPC decoding. We also assume that the PUs and the global memory units are interconnected with an MIN. The MIN network can be of different type such as butterfly, perfect k -shuffle. We consider the butterfly MIN here though the proposed framework can be applied to other classes of MIN in the similar way. Because many of the known MINs, such as butterfly, perfect k -shuffle, belong to the class of Delta networks [31] and can be proved to be topologically and functionally equivalent [32].

Assume that there are d_c PUs. Each PU consists of P SIMD slices, and thus can process P data simultaneously. Since the local memories are small and store only temporary data, the

L_n values and the $E_{n,m}$ values are stored in global memories. In each iteration, the L_n and $E_{n,m}$ values have to be read from the global memories and after processing, the updated values have to be stored back into the global memories. Thus there is significant traffic between the PUs and the global memories, and routing conflicts through the MIN network can seriously affect the throughput. In this section we describe how the LDPC code can be designed to reduce the routing conflicts and thereby increase the decoding throughput.

A. Feature Identification

To minimize the routing conflicts through the MIN network during LDPC decoding, we need to first characterize the constraints imposed by the MIN and translate the constraints to the corresponding code features.

1) *MIN-based Constraints*: Let the number of global memory units be n_b . The MIN interconnection network of size N connects the d_c PUs to the n_b global memory units. Clearly, $d_c, n_b \leq N$. When multiple data have to be routed through the network at the same time, certain routing paths (between the input and output nodes) cannot be activated simultaneously. The relative positions of the input/output nodes play an important role in determining the conflicting paths. To make the problem tangible, we focus on reduction of “pair-wise” conflicts. These conflicts are used to derive explicit constraints that are then used in the code construction process.

For any given input node X , the remaining $N - 1$ input nodes are divided into different *neighborhood groups* $G_i(X)$:

$$G_i(X) \triangleq \{ \text{all the input nodes that can be routed to the same } 2 \times 2 \text{ switch as } X \text{ in stage } i \} \quad (30)$$

It is obvious that $Y \in G_i(X) \Leftrightarrow X \in G_i(Y)$.

If the input node X is represented by $X = (x_{n-1}, \dots, x_1, x_0)$, $x_i \in \{0, 1\}$, the necessary and sufficient condition that an input node $Y = (y_{n-1}, \dots, y_1, y_0)$ belongs to the i -th neighborhood group $G_i(X)$ for butterfly MIN is given in Eqn. (31). Fig. 9 shows the neighborhood groups of $(010)_b$ in an 8-point butterfly MIN: $G_0(010) = \{(011)_b\}$, $G_1(010) = \{(000)_b, (001)_b\}$ and $G_2(010) = \{(100)_b, (101)_b, (110)_b, (111)_b\}$.

$$Y \in G_i^{butterfly}(X) \Leftrightarrow \begin{cases} y_j = x_j, & j > i \\ y_j \neq x_j, & j = i \\ \text{don't care,} & \text{otherwise} \end{cases} \quad (31)$$

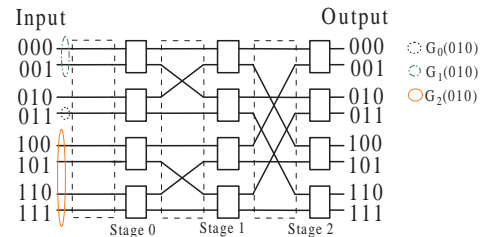


Fig. 9. Butterfly MIN: neighborhood groups of $(010)_b$.

To derive a method to reduce the probability of conflict, we introduce the concept of *exclusive node set* (ENS). Let $E_i(X)$

be a set of input nodes corresponding to i th-level ENS which includes X and all the elements in $G_j(X), \forall j \leq i$, i.e.:

$$E_i(X) \triangleq \{X\} \cup G_0(X) \cup \dots \cup G_i(X) \quad (32)$$

For the butterfly MIN in Fig. 9, $E_0(010) = \{010, 011\}$. $E_i(X)$ has several properties: (1) $|E_i(X)| = 2^{i+1}$. (2) $\forall X' \in E_i(X)$, $E_i(X') = E_i(X)$. (3) $\forall X' \in \overline{E_i(X)} \Rightarrow \forall X'' \in E_i(X')$, $E_i(X'') \cap E_i(X) = \phi$.

From these properties, we can see that the N input nodes can be divided into $\frac{N}{2^{i+1}}$ non-intersecting groups in i -th level ENS. In Fig. 9, the groups in 0-th level ENS are $\{000, 001\}$, $\{010, 011\}$, $\{100, 101\}$, $\{110, 111\}$. To reduce the possibility of routing conflict, no more than one node from each ENS should be selected as input node. Thus the maximum number of input nodes that can be routed simultaneously is the same as the cardinality of i th level ENS, i.e., $N/2^{i+1}$. Our goal is to have the largest number of input nodes participate in routing at any given time so we start with the 0-th level ENS.

2) *Code Features*: $\frac{N}{2}$ input nodes from the nodes in the 0th level ENS can be routed simultaneously. Since the number of input nodes equals the maximum row weight (d_c), this translates to the constraint $d_c \leq \frac{N}{2}$. In addition, the number of block columns $n_b \leq N$. Because of the sparseness of the parity check matrix for LDPC codes, we can easily find configurations that satisfy these constraints.

Since the two nodes in the same group in 0-th level ENS should not be routed simultaneously, the H_b block matrix should have the following structure. In any row, there is no more than one '1' element belonging to the same ENS group.

The constraints on n_b and d_c can be further refined by considering the encoding complexity. As shown in [12], the constraints for encoder efficient codes are as follows. If $n_b = N$, the maximum value of $d_c = \frac{N-m_b-1}{2} + d_{min}^v$; if $n_b < N$, the result remains the same since we can amortize the un-used $N - n_b$ output nodes for the left part of H_b . Also, it is clear from the pigeonhole principle, that the maximum variable node degree $d_{max}^v \leq m_b - d_{min}^v$.

The positions of the d_c input nodes can be chosen arbitrarily as long as the two nodes do not belong to the same ENS group. The positions of the output nodes are constrained by the positions of the input nodes. These constraints can be derived from Theorem 6.1. We will first present a lemma.

Lemma 6.1: If the routing conflict between two input nodes (X and Y) in an $N = 2^n$ -point butterfly MIN starts at switch stage- s and ends at switch stage- t ($t > s$), then

- The input nodes $X \in G_s(Y)$ and $Y \in G_s(X)$.
- The absolute value of difference in indices of the destination nodes of X and Y is in the range of $\Delta(t, k) = \sum_{i=t+1}^{n-1} 2^i \cdot u(k-i) + \delta$, where $\forall k \in [t+1, n-1]$ is an integer, $u(\cdot)$ is a unit step function and $\delta \in \{-1, 0, 1\}$.

The first statement is obvious from the definition of ENS. The proof of the second statement can be derived from the structure of MIN and is omitted here. We will only give an example. In a 32-point butterfly MIN ($n = 5$), if $t = 2$, then for routing conflict, the difference in indices of the destination nodes are $\Delta(2, 3) = 2^3 + \delta = 8 + \delta$ for $k = 3$, and $\Delta(2, 4) = 2^3 + 2^4 + \delta = 24 + \delta$ for $k = 4$.

Theorem 6.1: If two input nodes X and Y are such that $X \in G_s(Y)$ (equivalently $Y \in G_s(X)$) and the absolute value of difference of the destination nodes' indices is not $\Delta(t, k)$, $\forall t, k \in [s+1, n-1]$, then there is no conflict when nodes X and Y are routed simultaneously.

Theorem 6.1 can be derived directly from Lemma 6.1.

The constraints are thus summarized as follows:

- $n_b = N$, $d_c \leq \frac{N-m_b-1}{2} + d_{min}^v$.
- $d_{min}^v \geq 2$, $d_{max}^v \leq m_b - d_{min}^v$.
- No more than one '1' within an ENS group in any row.
- For two input nodes in G_s , avoid assigning their output nodes with difference of $\Delta(t, k)$ (absolute value), $\forall t, k \in [s+1, n-1]$.

By applying the constraints presented above in step 3(v) in Section II-C, the routing conflicts greatly reduce. Our experimental results show that the algorithm converges very well. Next we will show a code design example following these rules.

B. LDPC Code Design Example

In this section, we illustrate our procedure in the design of a rate $\frac{3}{4}$ LDPC code which fully exploits the characteristics of an SDR platform equipped with a $N=32$ -point butterfly MIN. From the discussion above, we can derive the design parameters as follow:

- $n_b = 32$, $m_b = 8$, code rate $R = \frac{3}{4}$
- $d_{min}^v = 2$, $d_{max}^v \leq 6$, check node degree $d_c = 13$

With these constraints, we can design a finite length LDPC code as described in Section II-C (for details see [12]). In a single run, we obtained two H_b matrixes with the same degree distribution – one is randomly generated as shown in Fig. 10(a) and the other is optimized for MIN interconnection as shown in Fig. 10(b).

In the H_b block matrices in Fig. 10, columns $2j$ and $2j+1$ belong to the same ENS group. In this figure every other ENS group has been highlighted. Notice that in the random H_b matrix of Fig. 10(a), there are multiple cases where there are two '1's that belong to the same 0-th level ENS group in the same row. In contrast, in the H_b matrix of Fig. 10(b), all such cases are eliminated.

After the LDPC codes are constructed, we map the original and optimized LDPC codes to the target architecture. Here the number of input nodes is $d_c = 13$ and $n_b = 32$. The mapping will result in a look-up table that stores the switch states for each stage of the MIN. For each block row there are $16 \times \log_2(32) = 80$ bits that need to be stored. There are $m_b = 8$ block rows in the H_b matrix thus result in a total of 640 bits switch states information.

We see that when the original H_b matrix is mapped to the architecture, there are on an average 4 conflicts out of the 80 switch states per block row, causing a slow down of 47.5%. The slowing down factor is calculated by dividing the additional number of cycles, which are introduced by separately routing the data that caused conflicts, by the overall number of cycles. The interconnection-aware LDPC code had no conflicts in any switch and thus no routing conflicts. This translates to a very high decoding throughput.

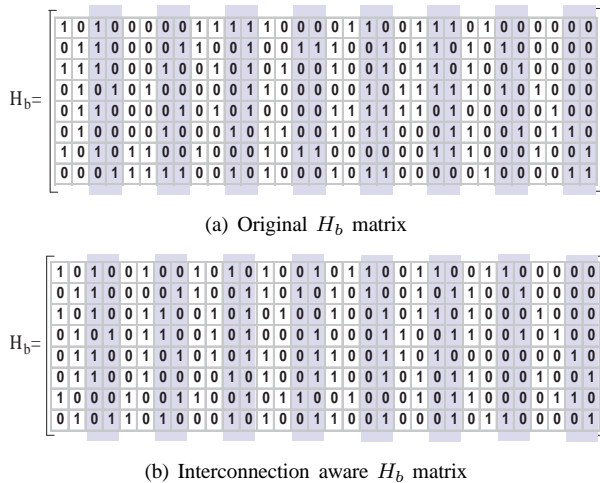


Fig. 10. An example of interconnection aware LDPC code with rate $\frac{3}{4}$.

FER Performance: We verify the performance of the code before and after interconnection-aware optimization. Fig. 11 shows the FER performance for AWGN channel when the LDPC encoded data is modulated with binary phase shift keying (BPSK). Each codeword is decoded with maximum of 15 iterations and each FER point represents the average of 25 error codewords. The proposed code generation procedure did not introduce any degradation in the FER performance. For better comparison, we also included simulation results for WiMax LDPC codes.

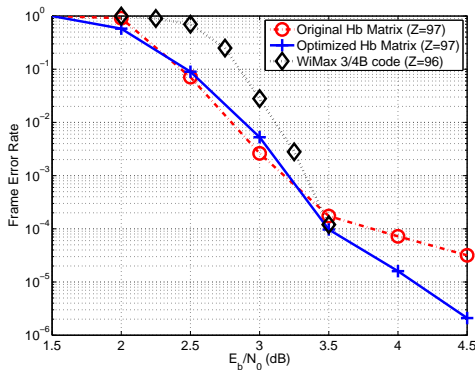


Fig. 11. FER of the rate $\frac{3}{4}$ LDPC codes shown in this section and the WiMax 3/4A LDPC code [33].

VII. CONCLUSION

In this paper, we presented a procedure to design LDPC codes that can be mapped efficiently onto multi-processor SDR platforms. The general design flow for architecture-aware code design was presented along with three case studies. In all cases, the FER performance of the modified LDPC code was not compromised.

First, we considered the case when only one PU is available and the local memory of the PU is not sufficient to support decoding of the entire parity check matrix. Here the matrix is partitioned horizontally into super-codes and the PU performs super-code based decoding. We show how addition of certain constraints during the code design phase results in enhanced

FER performance compared to that of typical super-code based decoding, as well as a 2X reduction in the memory traffic. Next, we consider the case where the parity check matrix has a large weight and multiple PUs are available for computation. We show how certain constraints can be used to create a matrix which supports even distribution of computation load among the PUs. Finally, we consider the case where the PU has very limited local memory and so there is a large volume of traffic between the PU and the global memory. We show how addition of code constraints can be used to reduce the number of routing conflicts in a MIN based interconnection network, resulting in substantial improvement in the throughput.

Code construction principles presented in this paper can also be applied to some of the LDPC codes used in standards. Since the LDPC codes are linear block codes, we can exchange the whole block row and block column of the parity check matrix for efficient decoding. This does not change the code as long as the codeword bits are restored to their original order after decoding.

ACKNOWLEDGEMENT

This work was supported in part by grants NSF-ITR 0325761 and NSF CSR-EHS 0615135. The authors also thank S. Seo for help with the SODA implementation.

REFERENCES

- [1] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8, no. 1, pp. 21–28, Jan. 1962.
- [2] S.-Y. Chung, G. Forney, T. 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, no. 2, pp. 58–60, Feb. 2001.
- [3] C. Howland and A. Blanksby, "A 220 mW 1 Gb/s 1024-bit rate-1/2 low density parity check code decoder," in *IEEE Conf. on Custom Integrated Circuits (CICC)*, May 2001, pp. 293–296.
- [4] G. Al-Rawi, J. Cioffi, and M. Horowitz, "Optimizing the mapping of low-density parity check codes on parallel decoding architectures," in *Intl. Conf. on Inform. Tech.: Coding and Computing*, April 2001, pp. 578 – 586.
- [5] E. Boutillon, J. Castura, and F. R.Kschischang, "Decoder-first code design," in *Proc. of the 2nd Intl. Symp. on Turbo Codes & Related Topics*, Sept. 2000, pp. 459–462.
- [6] Y. Chen and D. Hocevar, "A FPGA and ASIC implementation of rate 1/2, 8088-b irregular low density parity check decoder," in *Proc. of Global Telecom. Conf. (GlobeCom)*, vol. 1, Dec. 2003, pp. 113–117.
- [7] M. Mansour and N. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. VLSI*, vol. 11, no. 6, pp. 976–996, 2003.
- [8] D. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *IEEE Workshop on Signal Processing Systems (SIPS)*, 2004, pp. 107–112.
- [9] H. Zhong and T. Zhang, "Block-LDPC: a practical LDPC coding system design approach," *IEEE Trans. Circuits and Systems I*, vol. 52, no. 4, pp. 766–775, April 2005.
- [10] Y. Zhu and C. Chakrabarti, "Aggregated Circulant Matrix based LDPC codes," in *Proc. of IEEE ICASSP*, vol. 3, May 2006, pp. 916–919.
- [11] M. Mansour and N. Shanbhag, "A 640-Mb/s 2048-bit programmable LDPC decoder chip," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 684 – 698, March 2006.
- [12] Y. Zhu and C. Chakrabarti, "Architecture-aware LDPC code design for software defined radio," in *IEEE Workshop on Signal Processing Systems (SIPS)*, 2006, pp. 405–410.
- [13] —, "Memory Efficient LDPC Code Design for High Throughput Software Defined Radio (SDR) systems," in *Proc. of IEEE ICASSP*, vol. 2, May 2007, pp. 9–12.
- [14] H. Sankar and K. R. Narayanan, "Memory-efficient sum-product decoding of LDPC codes," *IEEE Trans. Commun.*, vol. 52, no. 8, pp. 1225–1230, Aug. 2004.

- [15] M.Luby, M.Mitzenmacher, A.Shokrollahi, D.Spielman, and V.Stemann, "Practical loss-resilient codes," in *Proc. of 29th Annual ACM Symp. Theory of Computing*, May 1997, pp. 150–159.
- [16] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. on Inform. Theory*, vol. 47, no. 2, pp. 619–637, Feb. 2001.
- [17] R. Urbanke, "LDPC optimization program," Available at: <http://lthcwww.epfl.ch/research/ldpcopt/>.
- [18] M.Chiani and A.Ventura, "Design and performance evaluation of some high-rate irregular low-density parity-check codes," in *Proc. of IEEE GlobeCom Conf. 2001*, vol. 2, Nov. 2001, pp. 990–994.
- [19] M.Yang, W.E.Ryan, and Y.Li, "Design of efficiently encodable moderate-length high-rate LDPC codes," *IEEE Trans. on Commun.*, vol. 52, no. 4, pp. 564–571, April 2004.
- [20] Y. Mao and A. Banihashemi, "A heuristic search for good low-density parity-check codes at short block lengths," in *Proc. of IEEE ICC 2001*, vol. 1, June 2001, pp. 41–44.
- [21] X. Hu, E. Eleftheriou, and D. Arnold, "Progressive edge-growth Tanner graphs," in *IEEE Global Telecom. Conf. (GLOBECOM)*, vol. 2, Nov. 2001, pp. 995–1001.
- [22] T. Tian, C. R. Jones, J. D. Villasenor, and R. D. Wesel, "Selective avoidance of cycles in irregular LDPC code construction," *IEEE Trans. on Commun.*, vol. 52, no. 8, pp. 1242–1247, August 2004.
- [23] C. Di, D. Proietti, I. Telatar, T. Richardson, and R. Urbanke, "Finite-length analysis of low-density parity-check codes on the binary erasure channel," *IEEE Trans. on Inform. Theory*, vol. 48, no. 6, pp. 1570–1579, June 2002.
- [24] T. Richardson, "Error floors of LDPC codes," in *Proc. of Allerton conf.*, 2003, pp. 1426–1435.
- [25] R. Tanner, D. Sridhara, A. Sridharan, T. Fuja, and D. Costello, "LDPC block and convolutional codes based on circulant matrices," *IEEE Trans. Inform. Theory*, vol. 50, no. 12, pp. 2966–2984, Dec. 2004.
- [26] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. on Inform. Theory*, vol. 47, no. 2, pp. 638–656, Feb. 2001.
- [27] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: a low-power architecture for software radio," in *The 33rd Ann. Int. Symp. on Computer Arch. (ISCA)*, June 2006, pp. 89–101.
- [28] T. Mohsenin and B. M. Baas, "Split-row: A reduced complexity, high throughput LDPC decoder architecture," in *IEEE International Conference of Computer Design (ICCD)*, 2006, pp. 320–325.
- [29] J. Zhang and M. P. C. Fossorier, "Shuffled Iterative Decoding," *IEEE Trans. Commun.*, vol. 53, no. 2, pp. 209–213, Feb. 2005.
- [30] J. Doré, M. H. Hamon, and P. Pénard, "On Flexible Design and Implementation of structured LDPC codes," in *IEEE Intl. Symp. on Personal, Indoor and Mobile Radio Commun.*, Sept. 2007.
- [31] J. H. Patel, "Performance of processor-memory interconnections for multiprocessors," *IEEE Trans. on computers*, vol. C-30, pp. 771–780, Oct. 1981.
- [32] C. Wu and T. Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. on computers*, vol. C-29, pp. 694–702, Aug. 1980.
- [33] T. Brack, M. Alles, F. Kienle, and N. Wehn, "A Synthesizable IP Core for WiMAX 802.16E LDPC Code Decoding," in *Proc. of IEEE PIMRC*, Sept. 2006.



Chaitali Chakrabarti (SM'02) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1984, and the M.S. and Ph.D. degrees in electrical engineering from the University of Maryland, College Park, in 1986 and 1990, respectively. She is a Professor with the Department of Electrical Engineering, Arizona State University (ASU), Tempe. Her research interests include the areas of low power embedded systems design including memory optimization, high level synthesis and compilation, and VLSI architectures and algorithms for signal processing, image processing, and communications.

Prof. Chakrabarti was a recipient of the Research Initiation Award from the National Science Foundation in 1993, a Best Teacher Award from the College of Engineering and Applied Sciences from ASU in 1994, and the Outstanding Educator Award from the IEEE Phoenix Section in 2001. She served as the Technical Committee Chair of the DISPS subcommittee, IEEE Signal Processing Society (2006-2007). She is now an Associate Editor of the Journal of VLSI Signal Processing Systems and the IEEE Transactions on Very Large Scale Integration Systems.

Yuming Zhu (S'00, M'07) received the B.E. and M.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 1999 and 2002 respectively, and the Ph.D. degree in electrical engineering from Arizona State University, in 2006.

During the summer of 2005, he worked at the Communication and Medical Systems Laboratory, DSP Solution R&D Center, Texas Instruments Inc., Dallas, TX, where he was involved in WiMax modem development. Since September 2006, he has been working on WiMax and 3GPP long term evolution (LTE) modem development at the Communication and Medical Systems Laboratory, DSP Solution R&D Center, Texas Instruments Inc., Dallas, TX.

His research interests include VLSI architectures for communication and signal processing systems.

