Hardware Implementation and Performance Analysis of Resource Efficient Probabilistic Hard Decision LDPC Decoders

Burak Unal, *Student Member, IEEE*, Ali Akoglu, *Member, IEEE*, Fakhreddine Ghaffari, *Member, IEEE*, and Bane Vasić, *Fellow, IEEE*

***Abstract*— The Gallager B (GaB), among the hard-decision class of low-density-parity-check (LDPC) algorithms, is an ideal candidate for designing high-throughput decoder hardware. However, GaB suffers from poor error-correction performance. We introduce a probabilistic GaB (PGaB) algorithm that disturbs the decisions made during the decoding iterations randomly with a probability value determined based on experimental studies. We propose a heuristic that switches the decoding from GaB to PGaB after certain number of iterations and show that our heuristic reduces the average iteration count by up to 62% compared with GaB. We evaluate the hardware performance and resource requirement trends of PGaB over three quasicyclic codes using the Xilinx Virtex-6 field programmable gate array. We extend this analysis to performance comparison over our implementations of gradient descent bit flipping (GDBF) and probabilistic GDBF (PGDBF) algorithms for each code studied in this paper. We achieve up to four orders of magnitude better error correction performance than the GaB with less than 1% loss in throughput performance. Our heuristic consistently results with an improvement in maximum operational clock rate across all codes compared with the GDBF and PGDBF.**

***Index Terms*— High-performance LDPC decoders, FPGA architectures, low complexity implementation, low-density parity-check codes.**

I. Introduction

LOW-DENSITY Parity Check (LDPC) codes offer performance improvement and implementation cost saving for long codeword lengths compared to Reed-Solomon (RS) and Bose-Chaudhuri-Hocquenghem (BCH) codes as they are theoretically proven to be asymptotically good family of codes [1]. Therefore, for a sufficiently high codeword length, LDPC will outperform a BCH or RS code of a comparable rate. Binary LDPC codes are adopted in many applications and standards [2], such as digital video broadcasting, 10GBase-T Ethernet, WiMAX wireless communications, as well as data storage systems. LDPC codes have also been selected as the data channel coding scheme for the 3GPP new radio access technology of the fifth generation (5G) mobile communication standard [3]. In the literature, we have seen soft-decision and hard-decision decoders as two main classes of LDPC decoding algorithms. Soft-decision decoders such as Belief Propagation (BP), Min-Sum (MS) [4], and Offset MinSum [5] offer high error correction performance with the cost of high computation complexity. On the other hand, hard-decision decoders such as Gallager B (GaB) [6], [7], Bit-Flipping (BF) [8], Gradient Descent Bit-Flipping (GDBF) [9], have much less hardware requirements than soft-decision decoders, and achieve higher throughput with a trade-off in the error correction performance. Message passing decoders of LDPC codes have algorithmic complexity that is linear in codeword length compared to the quadratic complexity of algebraic decoding RS and BCH codes rely on. In addition, LDPC codes handle soft channel outputs which is essential in numerous applications even in optical communications and data storage channels, especially in flash memories [10].

Among the hard-decision class of LDPC algorithms, hardware realization of the GaB [7] has not been favorable due to its poor decoding performance. On the other hand, GaB is an ideal candidate for designing a high-throughput decoder due to its simplicity of computations requiring combinational circuits at the scale of only 2-bit multiplication operations. In this study, our aim is to answer the question of whether it is feasible or not to bridge the gap between GaB and better performing hard-decision (bit flipping) based algorithms in terms of decoding performance without sacrificing its suitability for hardware implementation. For this, we introduce a Probabilistic GaB (PGaB) algorithm by applying a probabilistic stimulation function over the iterative decoding process. We present the details of our incremental approach to designing and implementing the PGaB hardware architecture.

In order to improve the GaB decoding performance, we first analytically study the cases for which a message bit received from the channel becomes the determining factor in GaB for a decision made during the iterative decoding process. We then introduce an algorithm that disturbs those decisions with a predefined probability, which we refer to as p_v. We experimentally identify the p_v that results with preferable decoder performance.

In order to reduce the hardware cost and improve the throughput of the implementation, we first show that, rather

Manuscript received October 1, 2017; revised January 18, 2018 and March 3, 2018; accepted March 5, 2018. This work was supported in part by NSF under Grant ECCS-1500170 and in part by the Indo-US Science and Technology Forum through the Joint Networked Center for Data Storage Research under Grant JC-16-2014-US. This paper was recommended by Associate Editor F. J. Kurdahi. This paper was presented in [6]. *(Corresponding author: Burak Unal.)*

B. Unal, A. Akoglu, and B. Vasić are with the Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721 USA (e-mail: burak@email.arizona.edu; akoglu@email.arizona.edu; vasic@email.arizona.edu).

F. Ghaffari is with the ETIS, ENSEA/University of Cergy-Pontoise/CNRS, 95014 Cergy-Pontoise, France (e-mail: fakhreddine.ghaffari@ensea.fr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2018.2815008

than using a complex and hardware demanding random number generator, using a less sophisticated random number generator based on the linear feedback shift register (LFSR), which requires fewer hardware resources, is sufficient to improve the decoding performance. We then propose a heuristic that allows switching to PGaB only when GaB is not able to correct the errors in predetermined number of iterations. We investigate the impact of switching from GaB to PGaB at a specific iteration, which we refer to as s_i. We experimentally identify the s_i that results with preferable decoder performance, and show that when s_i is set to fifteen, we also drastically reduce the average iteration count by up to 62% compared to GaB.

After determining the p_v and s_i values, for hardware performance analysis, we implement GaB, PGaB, Gradient Descent Bit Flipping (GDBF) [9], and Probability based Gradient Descent Bit Flipping (PGDBF) [11] algorithms on the Xilinx Virtex- 6 Field Programmable Gate Array (FPGA). We conduct a detailed robustness analysis that involves evaluating the impact of a change in code rate and codeword length over the FPGA based implementations of GaB, PGaB, GDBF and PGDBF. We show that PGaB consistently results with higher maximum operational clock rate over the GDBF and PGDBF by up to a factor of 3.3×. To the best of our knowledge, there is no prior work on the FPGA based implementation of the GDBF and PGDBF on the LDPC codes studied in this paper.

We finally compare the decoding performance of the PGaB with GaB, GDBF, PGDBF, and MinSum based on frame error rate (FER) for each code studied in this paper. We show that GaB architecture delivers the best throughput while using fewest FPGA resources, however performs the worst in terms of decoding performance. The PGaB results with up to four orders of magnitude decoding performance improvement over the GaB, exceeding the performance of GDBF over the codes studied in this paper, with a negligible loss (less than 1%) in throughput performance compared to the GaB. We conclude that the PGaB is able to bridge the gap between GaB and complex decoding algorithms such as GDBF and PGDBF without sacrificing the throughput advantage of the GaB by consistently exceeding FER performance of the GDBF.

The rest of the paper is organized as follows. In Section II, we give an overview of the baseline GaB, along with the GDBF and PGDBF decoding algorithms. In Section III, we present our methodology for introducing the probabilistic behavior to the GaB and determining the critical parameters for the PGaB implementation. We discuss the hardware implementations for GaB, PGaB, GDBF, and PGDBF in Section IV. After giving an overview of our simulation environment in Section V, we evaluate the decoding performance and hardware performance of PGaB in Section VI. We present our robustness analysis of the PGaB over changes in code rate and codeword length in Section VII. Finally, in Section VIII, we present our conclusions and future work.

II. Overview of Decoding Algorithms

An LDPC code is defined by a sparse parity-check matrix H [12], with size (M, N), where $N > M$. A codeword is a vector $\mathbf{x} = (x_1, x_2, \ldots, x_N) \in \{0,1\}^N$, which satisfies $H\mathbf{x}^T = 0$. We denote by $\mathbf{r} = \{r_1, r_2, \ldots, r_N\} \in \{0,1\}^N$ the output of a Binary Symmetric Channel (BSC), in which the bits of the transmitted codeword $\mathbf{x}$ have been flipped with crossover probability α. The graphical representation of an LDPC code is a bipartite graph called Tanner graph [13] composed of two types of nodes including N number of Variable Node Units (VNUs, v_n, $n = 1, \ldots, N$) and M number of Check Node Units (CNUs, c_m, $m = 1, \ldots, M$). In the Tanner graph, a VNU v_n is connected to a CNU c_m when $H(m,n) = 1$. An example Tanner graph and its H matrix are shown in 1. Let us also denote $\mathcal{N}(v_n)$ the set of CNUs connected to the VNU v_n, with a connection degree $d_v = |\mathcal{N}(v_n)|$, and denote $\mathcal{N}(c_m)$ the set of VNUs connected to the CNU c_m, with a connection degree $d_c = |\mathcal{N}(c_m)|$. Based on a decision function applied over the received messages from each adjacent vertex, each CNU and VNU sends a message back to its adjacent vertices. This iterative message processing between nodes recover the original data, which may have been exposed to channel noise.

A. Gallager B (GaB)

Binary messages are exchanged between CNUs and VNUs during each iteration of the decoding process and new messages are computed in an extrinsic manner. A VNU excludes the message received from a CNU, when the VNU is calculating the message to be sent back to that specific CNU. This is valid for the message calculation for the CNU as well. Each message represents an estimation on the correctness of the received word from the channel. Eventually, VNUs and CNUs accumulate gradually more information with each new iteration, which increasingly improves the codeword correction capacity. The estimation of the codeword is called posteriori decision information and is represented by $d^{(i)}_{n,m}$. Let $\mathrm{E}(x)$ represent a set of edges connected to a node x in the Tanner graph. The $v^{(i)}_{n,m}(e)$ denotes the extrinsic messages sent on edge e from a VNU v_n to a CNU c_m at iteration i and the $c^{(i)}_{m,n}(e)$ represents the extrinsic messages sent on edge e from a CNU c_m to a VNU v_n at iteration i. The received word from the channel at a VNU v_n is denoted as r_n. We express the operation of VNU and CNU using equations 1 and 2 respectively.

$$v^{(i)}_{n,m}(e) = \begin{cases} 1, & if\ r_n + (\sum_{e' \in \mathcal{N}(v_n)\setminus e} c^{(i)}_{m,n}(e')) > b_n \\ 0, & if\ r_n + (\sum_{e' \in \mathcal{N}(v_n)\setminus e} c^{(i)}_{m,n}(e')) < b_n \\ r_n, & otherwise \end{cases} \quad (1)$$

where i is the iteration count, e' is the set of extrinsic edges, and b_n is the threshold calculated as $b_n = \lceil d_v/2 \rceil$.

$$c^{(i)}_{m,n}(e) = (\sum_{e' \in \mathcal{N}(c_m)\setminus e} v^{(i)}_{n,m}(e'))mod2 \quad (2)$$

At each iteration, a new value of posteriori decision $d^{(i)}_{n,m}$ is computed as follows

$$d^{(i)}_{n,m} = \begin{cases} 1, & if\ r_n + (\sum_{e \in \mathcal{N}(v_n)} c^{(i)}_{m,n}(e)) > b_n \\ 0, & if\ r_n + (\sum_{e \in \mathcal{N}(v_n)} c^{(i)}_{m,n}(e)) < b_n \\ r_n, & otherwise \end{cases} \quad (3)$$

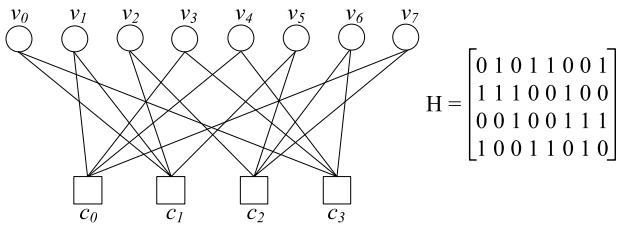

Fig. 1. Tanner graph (left) and its parity check matrix (right).

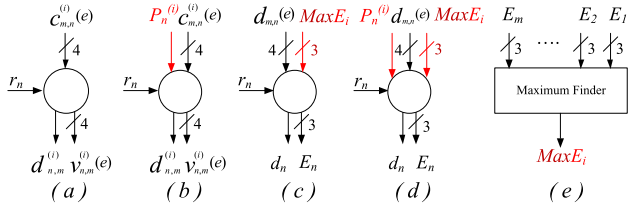

Fig. 2. General architecture of VNUs for (a) GaB, (b) PGaB, (c) GDBF, and (d) PGDBF for dv = 4 and N = 1296. (e) Maximum finder unit for GDBF and PGDBF decoders.

The VNU for GaB can be implemented using *Majority* gates (based on *and* and *or* logic functions only), and does not require complex operations such as the maximum finder required by the GDBF and PGDBF, along with the additional random number generator required by the PGDBF, which will be described in the following subsection.

B. GDBF and PGDBF VNU Analysis

The Gradient Descent formulation of Bit Flipping (BF) algorithm for the Binary Symmetric Channel (BSC) [9] sets a threshold for each VNU unit to determine whether the output of the VNU should be flipped or not based on an energy objective function. Energy objective is an integer value that varies between 0 and $d_v + 1$ and results with fewer number of flips in the successive iterations of the decoding process. Due to the integer representation of energy function, several VNUs may share the same maximum of energy value resulting with several bits to be flipped in one iteration. This may induce a negative impact on the convergence of the algorithm [11]. The Probabilistic GDBF (PGDBF) has been proposed to flip the outputs of only a random number of those VNUs with the maximum energy value. Energy calculations for the GDBF and PGDBF are governed by expressions similar to equations 1 and 2 [9], but they involve finding the maximum value across all VNUs in each iteration of the decoding process as illustrated in Figure 2. This gradient descent algorithm used in the PGDBF increases hardware complexity of PGDBF On the other hand, the VNU for GaB can be implemented using majority logic and *xor* gates and does not require complex operations. Later we will show that the maximum energy computation is the main bottleneck on the throughput performance of GDBF and PGDBF implementations.

III. Probabilistic GaB Algorithm

During the decoding process, the interactions between CNUs and VNUs may result in an oscillation phenomena due to the n^{th} order dependencies between CNUs and VNUs. In such cases, the decoding process may get trapped in a cyclic behavior. For example, in the Tanner graph [14] given in Figure 1, c_0 transmits message to v_1 and v_3. After receiving their inputs from all CNUs, v_1 and v_3 send their messages back to their designated CNUs. In this example, there is a third order dependency between c_0 and v_0 based on the message passing in the order of $(c_0 - v_7 - c_2 - v_2 - c_1 - v_0)$. If we count each CNU-VNU interaction as one iteration, then it would take three iterations for the message of c_0 to propagate to v_0. Similarly, there is also a second order dependency in the order of $(c_0 - v_1 - c_1 - v_0)$. During each iteration, CNUs and VNUs update their states. The sequence of states observed for a given VNU may show repeating pattern, which is called a trapping set [15]. Trapping means that the decoder cannot correct the error, and then it remains in the cyclic sequences of states. One way to break this cyclic behavior is to disturb the VNU when such a pattern is detected. One may introduce large memory to keep track of the states, but that would not be hardware friendly, since the trapping set size is unknown and there can be many thousands of different trapping sets. Therefore, we randomly disturb the state of each VNU to be able to escape from the trapping set. Of course, one may question that such disturbance could adversely affect the normal behavior of the VNU, but theoretical results indicate that this side effect does not significantly increase the number of iterations [16]–[18]. If the GaB decoder does not converge within user-defined number of (k) iterations, then we apply this probabilistic strategy (Probabilistic GaB) to escape from trapping set. We modify Equation 1 by introducing a probability function $p_n^{(i)}$ as shown in Equation 4. The PGaB flow is shown in Algorithm 1.

$$v_{n,m}^{(i)}(e) = \begin{cases} 1, & if\ \ p_n^{(i)} \oplus r_n + (\sum_{e' \in \mathcal{N}(v_n)\backslash e} c_{m,n}^{(i)}(e')) > b_n \\ 0, & if\ \ p_n^{(i)} \oplus r_n + (\sum_{e' \in \mathcal{N}(v_n)\backslash e} c_{m,n}^{(i)}(e')) < b_n \\ r_n, & otherwise \end{cases}$$

A. Determining How to Disturb the VNU

The truth table shown in Table I captures how we propose to modify the VNU function with an example on calculating only one of the output messages ($v_{n,m}^{(i)}(4)$). In this example we assume that the d_v is four where each VNU has five inputs including the received word (r_n) and four CNU messages ($c_{m,n}^{(i)}(1,2,3,4)$). Since in this example we are calculating the message for the fourth output of the VNU, message ($c_{m,n}^{(i)}(4)$) received from CNU is not used in the calculation.

In GaB algorithm, $v_{n,m}^{(i)}(e)$ is calculated by Equation 1 and is illustrated in Table I. CNU messages ($c_{m,n}^{(i)}(1,2,3)$) represent whether the previous decision of VNU is correct or not. We take a close look at the GaB VNU logic for the cases where there is a tie over the three inputs ($c_{m,n}^{(i)}(1)$, $c_{m,n}^{(i)}(2)$, $c_{m,n}^{(i)}(3)$) and the received message (r_n). In such cases, shown with rows in bold in Table I, the VNU output is determined by the r_n input. We argue that when the decoder is stuck in the trapping set, we should not use the r_n as a determining factor. Looking closely, when we express $v_{n,m}^{(i)}(4)$ for the PGaB (column 5) of Table 1, we see that function is equivalent to $c_{m,n}^{(i)}(2).c_{m,n}^{(i)}(3) + c_{m,n}^{(i)}(1).c_{m,n}^{(i)}(3) + c_{m,n}^{(i)}(1).c_{m,n}^{(i)}(2)$, and shows

Algorithm 1 Probabilistic Gallager B

Initialization $i = 0,\ v_{n,m}^{(0)}(e)_{e\in\mathcal{N}(v_n)} \leftarrow r_n,\ n = 1,\ldots,N.$
$d_{n,m}^{(0)} \leftarrow r_n,\ n = 1,\ldots,N.$
$s = H\mathbf{v}^{(0)^T} \mod 2$
while $s \neq 0 \quad and \quad i \leq i_{max}$ **do**
  Generate $p_n^{(i)}$, $n = 1,\ldots,N$, from $\mathcal{B}(p_v)$.
  for $n = 1,\ldots,N$ **do**
    Compute
    $c_{m,n}^{(i+1)}(e)_{e\in\mathcal{N}(c_m)}$ using Equation 3
    $v_{n,m}^{(i+1)}(e)_{e\in\mathcal{N}(v_n)}$ using Equation 4
    $d_{n,m}^{(i+1)}$ using Equation 2
  end for
  $s = H\mathbf{v}^{(i+1)^T} \mod 2$
  $i = i + 1$
end while
Output: $\mathbf{v}^{(i)}$

TABLE I
TRUTH TABLE FOR PGAB ALGORITHM

Inputs for VNU				GaB	PGaB
r_n	$c_{m,n}^{(i)}(1)$	$c_{m,n}^{(i)}(2)$	$c_{m,n}^{(i)}(3)$	$v_{n,m}^{(i)}(4)$	$v_{n,m}^{(i)}(4)$
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	**0**	**1**	**1**	**0**	**1**
0	1	0	0	0	0
0	**1**	**0**	**1**	**0**	**1**
0	**1**	**1**	**0**	**0**	**1**
0	1	1	1	1	1
1	0	0	0	0	0
1	**0**	**0**	**1**	**1**	**0**
1	**0**	**1**	**0**	**1**	**0**
1	0	1	1	1	1
1	**1**	**0**	**0**	**1**	**0**
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

that we ignore the received message for all input scenarios. If we ignore the received messages completely, decoder will fail. If we force all VNUs to rely on the received messages from the channel for the tie cases, then for the trapping set cases the decoder may not converge. The decoder cannot ignore the received messages, however during the decoding we do not know which VNU is in the trapping set. For this we introduce a mechanism that selects a predefined percentage of VNUs to ignore the received message and operate as the PGaB column of Table I. We refer to predefined percentage of VNUs as the p_v term in our implementation. The subset of VNUs that ignore the received message is randomly chosen based on the p_v value. In the following section we present our experimental approach for determining the value of p_v. The probability function can be applied to the decoder in various positions. For example, in the PGDBF decoder [19], the probabilistic function is applied randomly during the final output decision of a VNU to decide whether to flip the channel value or not. In the Noisy GaB [20], the randomness effect acts arbitrarily on both messages exchanged mutually between VNU and CNU. The main objective of these these studies is to distract the decoder by adding noise. Our approach to utilization of randomness is different from these studies as we attempt to use randomness in a more deterministic way. Rather than disturbing the outcome of decisions made during each iteration, we incorporate randomness directly into the message computation only for the cases when a tie occurs among the received messages of a VNU. Our Monte-Carlo simulations show better decoding performance, for the studied LDPC codes, when we introduce randomness as a tie-breaker for the VNU function when computing only messages sent from VNU to CNU.In the following subsection we present our approach for determining the p_v value.

B. Determining the p_v Value

Before proceeding to the hardware implementation, we need to determine Bernoulli distribution p_v, which represents the probability of $p_n^{(i)}$ taking the value of 1 ($P(p_n^{(i)} = 1)$). This will indicate the proportion of VNUs that will be disturbed in the hardware architecture. In this case, setting p_v to 0 would mean no disturbance for all VNUs. We conduct experiments, as shown in Figure 3, for four values of channel crossover probability α (0.025, 0.02, 0.03, and 0.035) by sweeping the p_v between 0 and 1. We choose four α values in order to check the consistency on the FER performance. In the figure, x-axis shows the range of p_v being 1 and y-axis shows the frame error rate for the LDPC codeword length of 1296 with degrees of VNU and CNU set to 4 and 8 respectively. As shown in Figure 3, for the case of alpha 0.02, the simulation point labeled with *A* indicates no stochastic behavior (GaB) where p_v is 0 and the point labeled with *B* shows the case where all VNUs operate as PGaB where p_v is 1.

Based on the plots in Figure 3, we conclude that disturbing all VNUs results with an improvement over the GaB (point *B*). We observe two trends in the figure that reveal important insights for determining the p_v. As the p_v value reduces to 0.4, the FER is almost insensitive to this change for both α values. We also observe a flood region between 0.1 and 0.2 where FER performance is the best for both α values. Based on this observation we set the p_v value to 0.2 for the hardware implementation. This leads us to selecting a random number generator (RNG) for generating the p_v with Bernoulli distribution. Random number generators have been studied in terms of their quality and complexity in the literature extensively [21]. For example, the Park-Miller [22] algorithm is one of the high quality random number generators that relies on linear congruential method, which would require complex hardware components. Therefore this type of RNG even though generates strong random numbers is not hardware friendly. Our design choice favors simplicity with the objective of a light-weight decoder architecture in terms of its hardware resource requirement. We argue that, in our case there is no need for a sophisticated RNG in the hardware implementation that gives precise distribution for a given p_v. This is because, for all cases where p_v is set to a non-zero value, we observe improvement over the GaB and the preferable performance occurs in a window ranging between 0.1 and 0.2. We use

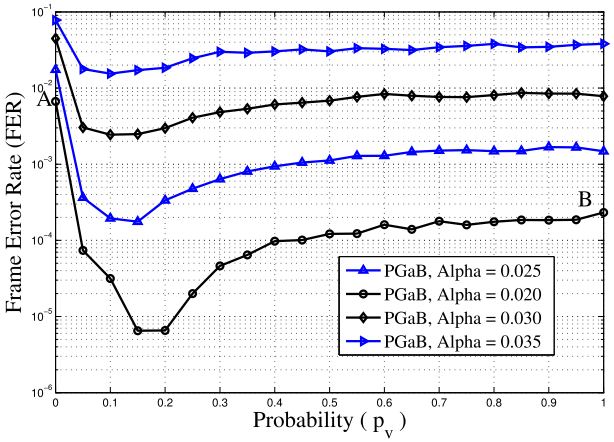

Fig. 3. Frame Error Rate versus p_v ($\alpha = 0.02, 0.025, 0.03$, and 0.035). LDPC code ($d_v = 4, d_c = 8, Z = 54$), ($N = 1296, M = 648$) when switching iteration s_i is set to 15.

this conclusion as basis for choosing a simpler and hardware friendly linear feedback shift register (LFSR) based RNG.

C. Determining the s_i Value

In section III-A we discussed the way we introduce probabilistic behavior to GaB to overcome trapping sets. Based on our simulations, we observe that GaB when successfully decodes a code, typically resolves the errors in less than ten iterations. Therefore we believe that a hybrid implementation that switches to PGaB only when GaB is not able to correct the errors in predetermined number of iterations would be a better approach than executing only PGaB in terms of FER performance. We conduct two experiments to validate our claim.

In the first experiment, we evaluate the impact of switching from GaB to PGaB after a specific number of iterations (s_i) for three regular LDPC codes with (N, d_v, d_c, R) configurations of (155, 3, 5, 0.5), (1296, 3, 6, 0.5) and (1296, 4, 8, 0.5). We vary the switching point from 5 to 50 and show the FER performance for different α values for each code shown in Figure 4. In the same plot we also plot the average number of iterations for three α values. For all experiments, we set the maximum number of iterations to 300. We evaluate the impact of change in codeword length on switching iteration using codeword length of 155 and 1296; and the impact of change in VNU and CNU degree using codes with connection degree d_v set to 3 and 4, and d_c set to 5, 6, and 8. All of these experiments indicate that setting the switching point between 15 and 20 iterations would be preferable for achieving better FER performance. Since the average number of iterations for three α values show an increasing trend as the switching point moves from 5 to 50, we conclude that 15 is the ideal point to make the switching from GaB to PGaB. A side benefit of switching after 15 iterations is the reduced power consumption since we dont use PGaB for all iterations and in hardware implementation we turn on the the RNG unit only after iteration count 15 has been reached.

In the second experiment, we set the switching point to 15 and p_v value to 0.2, and evaluate the impact of disturbing VNUs on average number of iterations. In Figure 5, we compare average number of iterations for the baseline GaB,

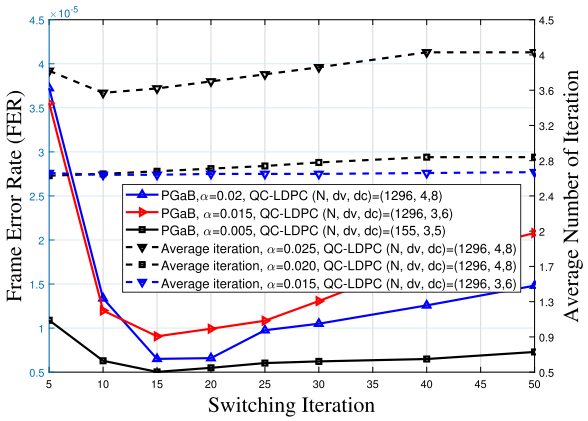

Fig. 4. Frame Error Rate and Average iteration versus iteration number to switch from GaB to PGaB (QC-LDPC codes with (N, d_v, d_c, R) configurations of (155, 3, 5, 0.5), (1296, 3, 6, 0.5) and (1296, 4, 8, 0.5)).

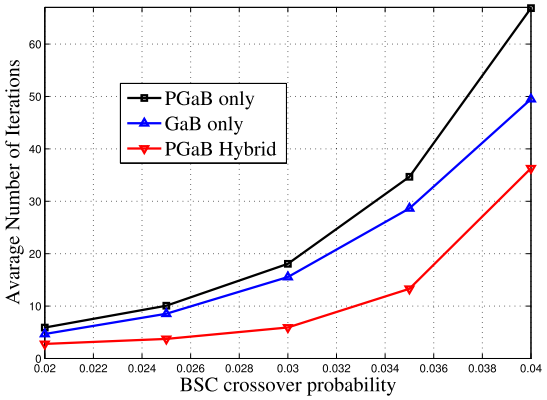

Fig. 5. Comparison of average number of iterations for PGaB, GaB, PGaB Hybrid (GaB for the first 15 iterations, and PGaB onwards) ($d_v = 4, d_c = 8$, $N = 1296, M = 648$).

the PGaB, and the hybrid implementation that relies on the execution of GaB for the first 15 iterations of the decoding process and PGaB afterwards. In the figure, x-axis shows the α range and y-axis shows the average number of iterations for three simulations. Figure shows that when we start using the PGaB after 15 iterations, the average number of iterations is always better than the deterministic GaB. We reduce the average iteration count by 40%, 56%, 62%, 54%, and 26% compared to the GaB for the α values studied in this experiment respectively. The PGaB only approach consistently results with larger number of iterations compared to the GaB only method. Disturbing the decoder starting with the first iteration results with adding more noise and therefore leads to increase in the average number of iterations. The hybrid PGaB on the other hand reduces the average number of iterations consistently with respect to the GaB only method. We believe that disturbing the GaB decoder after 15 iterations helps resolve some of the trapping set cases as shown theoretically by Ivanis and Vasic [20], which contribute to the increase in average number of iterations for the GaB only method. Reducing the maximum iteration count has also direct impact on the power consumed by the decoder. By reducing average iteration number, we also increase the throughput of the decoder. PGaB spends fewer iterations in average compared to the GaB to correct the errors.

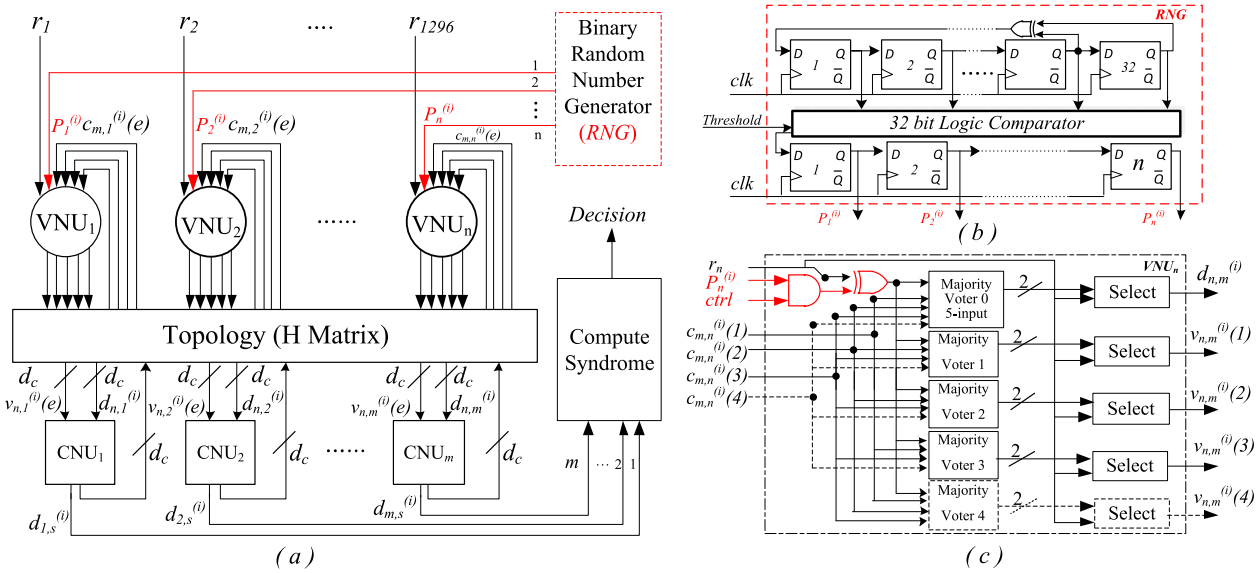

Fig. 6. Overall decoder architecture (a) for PGaB, VNU architecture for dv4 (c) and LFSR-32bits based Random Number Generator (b).

In the following section we will present our hardware results and decoding performance based on the hybrid PGaB implementation where we set p_v value to 0.2 and the s_i to 15. For the remainder of the paper we refer to the hybrid PGaB as the PGaB.

IV. Hardware Design

A. GaB and PGaB Hardware Design

We first show the generic architecture for GaB and PGaB with VNUs, CNUs and the $HMatrix$ based on a regular QC-LDPC code for codeword length (N) in Figure 6(a). The *Compute Syndrome* unit in the figure checks whether all of the CNUs are satisfied or not.

We show the details of the VNU architecture for d_v equals to 4 in Figure 6(c). The colored arrows along with the *and* and xor gates in the VNU architecture are used by the PGaB implementation. When d_v is set to 4, besides the control input ($ctrl$), there are six 1-bit inputs for each VNU. The first two inputs from top to bottom are 1-bit data received from the channel (r_n) and 1-bit random value ($P_n^{(i)}$) generated by the LFSR based RNG. Remaining four inputs ($c_{m,n}^{(i)}(1)$, $c_{m,n}^{(i)}(2)$, $c_{m,n}^{(i)}(3)$, $c_{m,n}^{(i)}(4)$) are the 1-bit messages received from CNUs (as $d_v = 4$). There are four 4-input majority voter units (labeled as 1-4) and one 5-input majority voter unit (labeled as 0). The majority voter generates a 2-bit output representing majority of 1s, majority of 0s, or tie cases. The *select* unit acts as a selection of the majority output, which generates 1 for the majority of 1s case, and 0 for the majority of 0s case. In the case of a tie, the *select* unit passes the received word to its output. In this generic architecture, if the d_v changes than the number of *Majority Voters* and the number of inputs need to be adjusted properly. For example, when d_v is set to three, the input ($c_{m,n}^{(i)}(4)$), output ($v_{n,m}^{(i)}(4)$), and components (*Majority Voter 4* and *Select* for $v_{n,m}^{(i)}(4)$ output) marked with dotted lines are excluded from the VNU. We implement a regular majority voter based on Table I. We do not show the details of the *Majority Voter* architecture, since it is straightforward to implement. The VNU operation is modified with the red marked lines and glue logic to adopt its function to PGaB. The control bit ($ctrl$) sets the first input of the xor gate to 0 if the algorithm is GaB, in which the xor gate passes the r_n input to its output, otherwise the output becomes a function of r_n and the $P_n^{(i)}$ for implementing the PGaB. A state machine controls switching between GaB and PGaB. After iteration number 15, if the decoder does not converge, the control bit ($ctrl$) is set to 1 by the state machine to switch to PGaB. The controller allows us to use VNU architecture of GaB to implement PGaB. Based on our conclusion about the RNG type to utilize in Section III.B, we implement a regular LFSR based RNG, shown in Figure 6(b) to feed a 1-bit random value to each VNU. We implement 32-bit LFSR to generate a 32-bit random number. The *32 bits Logic Comparator* compares 32-bit random number with the user defined $Threshold$ value determined in Section III-B. Finally, the output of the comparator, a one bit random number, is stored in the shift register. If the codeword length is N, then the RNG will take N number of cycles to generate the bits needed by all the VNUs. This N cycle overhead is applied only once during the first iteration of the decoding. During the subsequent iterations between the CNUs and VNUs, we simply generate one bit and use a shift register of size N to distribute the values to each VNU.

We do not show the details of the CNU architecture, since it is straightforward to implement. When d_c is set to 8, inputs are eight bit messages $v_{n,m}^{(i)}(e)$, and eight bit decision information ($d_{n,m}^{(i)}$) received from the VNUs. The outputs of a CNU are an eight bit message ($c_{m,n}^{(i)}(e)$) and one bit decision information ($d_{m,s}^{(i)}$). The $d_{m,s}^{(i)}$ is the output of the xor operation on the 8-bit input $d_{n,m}^{(i)}$. Decision information is sent to the $ComputeSyndrome$ unit to decide whether the decoder has converged or not. Message calculation is different than decision information calculation as it is executed in an extrinsic manner. The $c_{m,n}^{(i)}(e)$ is calculated by Equation 2. For instance, $c_{m,n}^{(i)}(1)$ is determined by calculating the xor of

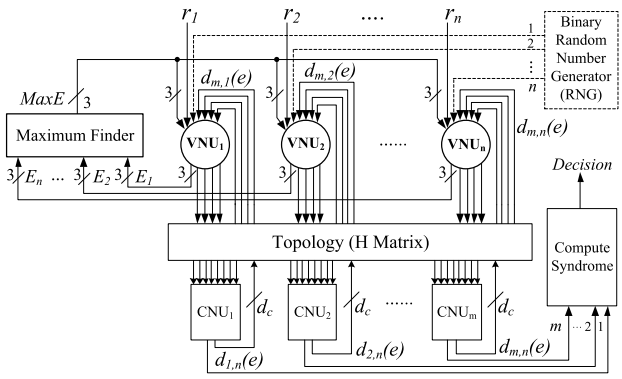

Fig. 7. Architecture of GDBF and PGDBF for $d_v = 4$, and N = 1296, where dc is determined by the code rate.

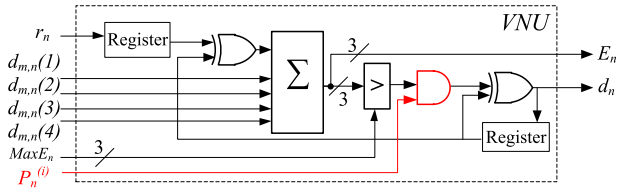

Fig. 8. Architecture of VNU for GDBF and PGDBF for $d_v = 4$, and N = 1296.

messages $v_{n,m}^{(i)}(2)$,..,$v_{n,m}^{(i)}(8)$ and excludes the $v_{n,m}^{(i)}(1)$. In summary, the CNU implementation requires one 8-bit *xor* gate and eight 7-bit *xor* gates.

B. GDBF and PGDBF Hardware Design

In order to present a comprehensive analysis on the decoding and hardware performance of PGaB, we implement two hard-decision based algorithms (GDBF and PGDBF). In this section, we present hardware implementations for these two algorithms. High level architectures for the GDBF and PGDBF decoders are shown in Figure 7 for a QC-LDPC code with codeword length of 1296 bits ($d_v = 4$, $d_c = 8$). The only difference between GDBF and PGDBF architectures is the binary RNG indicated with the dotted lines in Figure 7. The RNG generates 1296 binary 1-bit random numbers ($P_n^{(i)}$) to distribute to each VNU. Detailed architecture for the VNU is shown in Figure 8. The 1-bit received message (r_n) from the channel, the 1-bit decision estimations from the four CNUs ($d_{m,n}(e)$), and the 3-bit maximum energy value for the current iteration ($MaxE_n$) are common inputs for the VNU in GDBF and PGDBF. The summation operation in the VNU calculates the output energy value (E_n), which can be between zero and five. Therefore bit-width for the E_n and $MaxE_n$ are set to three bits. The *Maximum Finder* unit shown in Figure 7 computes the maximum of the E_n values received from each VNU in the current iteration i labeled as $MaxE_n$ in the figure. Each VNU uses the $MaxE_n$ and E_n to generate a 1-bit decision value (d_n). In the same iteration, if the E_n of a VNU is equal to the $MaxE_n$, then the output message d_n is flipped. If the E_n is less than the $MaxE_n$, then the d_n is not flipped. Additionally, for the PGDBF, a VNU receives a 1-bit random value generated by the LFSR based RNG ($P_n^{(i)}$). The d_n is a new message for all CNUs connected to the VNU. This iterative process continues till all CNUs are satisfied. A 1-bit message is sent by CNU to *Compute Syndrome* unit indicating whether a CNU it is satisfied or not. A state machine controls the *Compute Syndrome* unit to make a decision on whether the decoder has converged or not.

TABLE II
HARDWARE RESOURCE UTILIZATION, THROUGHPUT AND CLOCK RATE OF DECODING ALGORITHMS IMPLEMENTED FOR TANNER CODE ON VIRTEX6 FPGA

Algorithm	1-bit register	Slice LUTs	Fmax (MHz)	Throughput (Mbps)
GDBF* [23]	946	2151	132.7	4114.3
PGDBF* [23]	9161	3545	135.6	4202.5
MinSum [23]	13694	15350	237.2	197.5
GDBF	502	1630	137.5	4263.4
PGDBF	687	1802	138.2	4285.8

The hardware implementations for the GDBF and PGDBF have been studied based on the Tanner code ($N = 155$, $M = 93$, $d_v = 3$, $d_c = 5$) in [23]. We first implement these two algorithms based on the same code and compare their hardware resource usage and throughput performance with the published results (indicated as * in the table) on the Xilinx Virtex6 FPGA using Table II. With this comparison, our aim is to show that our implementations form a credible baseline for our extensive performance evaluations in the following section across GaB, PGaB, GDBF and PGDBF over various code lengths and code rates. We include MinSum in the table just to highlight the hardware efficiency of the hard-decision based algorithms with respect to this best performing soft-decision decoder. As shown in the table, we reduce the 1-bit register usage significantly by 92% compared to the PGDBF*. We also reduce the Slice LUT usage by 24% and 49% with our implementations of the GDBF and PGDBF respectively. The study by Le *et al.* [23] reveals limited amount of information about the hardware implementation approach for the GDBF* and PGDBF*. We believe that there are two factors contributing to significant reduction on resource usage for our implementations. First, our CNU implementation does not require any register, as we implement it as a combinatorial logic and each CNU sends its output message $d_{m,n}(e)$ back to the VNU without having to store it. Secondly, we take advantage of a resource efficient implementation of maximum finder logic as shown by [24] based on parallel tree structure for calculating the $MaxE_n$. Earlier we claimed that maximum finder unit was a critical factor on throughput performance of the GDBF. When we replaced the "Maximum Finder" logic with a hard coded maximum value in our version of the GDBF implementation, we observed a reduction in logic block resource usage by 14.7% and an increase in the maximum clock rate by a factor of 2.99× for this hypothetical implementation. Nevertheless, with our implementation by reducing the resource usage for GDBF and PGDBF significantly with a slight improvement in the maximum clock rate, we are setting a tighter constraint on measuring the hardware performance in terms of resource usage and throughput for the PGaB implementation.

V. SIMULATION ENVIRONMENT

Our simulation environment includes the GaB, PGaB, GDBF, and PGDBF implementations in C programming language. The simulation flow is shown in Algorithm 2.

Algorithm 2 Simulation Flow for Generating FER Plots

Input : *Decoding Algorithm* (GaB, PGaB, GDBF, PGDBF), *Codeword length of* 1296 (code rate 0.5 and 0.75) and *Codeword length of* 2212 (code rate 0.857) and *Crossover Probability* (α)

Output : FER plot of each algorithm over α

```
foreach Decoding Algorithm do
    foreach Code do
        FrameCounter = 0;
        foreach α do
            # α ∈ [0.001, 0.07] for N = 1296, rate 0.5
            # α ∈ [0.02, 0.03] for N = 1296, rate 0.75
            # α ∈ [0.005, 0.6] for N = 2212, rate 0.857
            ErrorCount = 0;
            while (ErrorCount < 100) do
                Generate a random codeword (Frame);
                FrameCounter = FrameCounter + 1;
                Add noise to Frame using α;
                Value = Decoding Algorithm();
                # Value from Compute Syndrome
                if (Value == 0) then
                    ErrorCount = ErrorCount + 1;
                if ErrorCount == 100 then
                    FER = 100/FrameCounter;
            Mark FER for α on FER plot;
```

We evaluate the impact of change in code rate on performance using codeword length of 1296 [25] with rates of 0.5 and 0.75; and the impact of change in codeword length using codeword length of 2212 [26]. For all algorithms, the d_v is equal to 4. For the FER analysis we include the FER performance of the flooding scheduling MinSum (MS) [4] and Offset MinSum (OMS) based decoders even though they belong to a different class of decoder algorithm, where CNUs and VNUs exchange messages of multi-bit granularity, as opposed to the bit flip class of algorithms with single-bit granularity that are considered in this paper. We include MS and OMS just to set the stage on where the hard-decision (bit flipping) based algorithms stand with respect to these best performing soft decision decoders. The MS and OMS used in this work are the quantized decoders with 4 bits for passed messages and 6 bits for A posteriori Log Likelihood Ratios (AP-LLR). We set the number of iterations to 20, the channel gain factor to 2, and the offset factor to 1 for OMS. We design and implement each decoder for each code (total of 12 architectures) on the Xilinx Virtex-6 FPGA (vc6vlx240t-2ff1156) and conduct post placement and routing analysis over hardware cost in terms of logic and register usage, and hardware performance in terms of maximum clock rate, and throughput. For each algorithm, FER curves are plotted as a function of the cross-over probability (α) over the BSC channel based on the simulation flow shown in Algorithm 2. Similar to other studies ([4], [23]), we calculate the system throughput using Equation 4. All designs have been implemented in VHDL. Functional verification is conducted by validating iteration by iteration post-routing CNU and VNU values against the C equivalent bit accurate implementation. We implemented the PGaB hardware architecture after completing a preliminary analysis and confirming the decoding performance of the PGaB based on the FER plots generated using the C simulation. We measured the total simulation time for PGaB on codeword length of 1296 (code rate 0.5) as 116 days on the Intel Xeon (2.33GHz, 8GB RAM) processor. The same simulation takes slightly over 5 minutes on our FPGA based testbed. Therefore, for certain cross-over probability values, since the simulation times for the C code are extremely long, we used the FPGA based simulations to generate the points on the FER plots. For example, in the case of cross-over probability value of 0.01, we reached up to processing 10^{10} codewords with PGaB to generate the point that represents the 10^{-8} frame error rate.

$$SystemThroughput = \frac{CodeLength \times MaxClockRate}{AvgIteration \times CyclesPerIteration} \quad (4)$$

VI. Performance Analysis

Figure 9 shows the FER performance comparison between the GaB, PGaB, GDBF and PGDBF algorithms as a function of the cross-over probability over the binary symmetric channel (BSC) on LDPC code with the rate 0.5. This chart shows that, with the probabilistic execution, we are able to bridge the gap between the GaB and the better performing decoding algorithms through PGaB. Another remarkable conclusion is our ability to perform better than the GDBF with the PGaB. The gap between PGaB and GaB in the error floor region where α is 0.01 quantifies the dramatic improvement (up to four orders of magnitude) achieved by disturbing randomly the state of the decoder.

In Table III, we present the resource usage, maximum clock rate, and throughput for the GaB and PGaB based on their FPGA implementations with 0.5 code rate. The percentage sign (%) in Table III indicates the change in resource usage, FMax, and Throughput for PGaB, GDBF, and PGDBF with respect to the GaB implementation. It takes 2 clock cycles to complete one iteration of the decoder, one cycle for VNU and one cycle for CNU. Average number of iterations and throughput of the decoder vary at different FERs. Throughput values in Table III are calculated based on the average number of iterations set to 2.5. With the probabilistic execution, the improvement in the error floor over the GaB comes with a negligible amount (0.85%) of loss in throughput performance. Even though modification to GaB involved including a RNG and an additional input to each VNU, the clock rate difference between the two designs is negligible. However, the decoding performance improvement is achieved with an increase on register and slice LUT usage by 17% and 24% respectively. Register overhead of the PGaB implementation includes the 1296 bits to store the 1-bit random number for each VNU and the 32-bit shift register to implement the random number generator. The increase in resource usage is a reasonable trade-off for improving the decoding performance. The maximum

TABLE III

RESOURCE USAGE, MAXIMUM CLOCK RATE AND THROUGHPUT OF GaB, PGaB, GDBF, AND PGDBF BASED ON THE FPGA IMPLEMENTATIONS FOR QC-LDPC $(N, d_v, R) = (1296, 4, 0.5)$, QC-LDPC $(N, d_v, R) = (1296, 4, 0.75)$, AND QC-LDPC $(N, d_v, R) = (2212, 4, 0.857)$. (HARDWARE IMPLEMENTATION RESULTS FOR GaB AND PGaB FOR QC-LDPC $(N, d_v, R) = (1296, 4, 0.5)$ AND QC-LDPC $(N, d_v, R) = (1296, 4, 0.75)$, APPEAR IN [6]) (% INDICATES THE DIFFERENCE WITH RESPECT TO GaB)

	1-bit Register			Slice LUTs			FMax (MHz)			Throughput (Gbps)		
Codeword	R=0.50	R=0.75	R=0.857	R=0.50	R=0.75	R=0.857	R=0.50	R=0.75	R=0.857	R=0.50	R=0.75	R=0.857
GaB	7812	4596	13308	11784	6097	23292	147	114	59	38224	29575	26410
PGaB	9141	5601	15552	14605 (24%)	7133 (17%)	28895 (24%)	146 (-1%)	113 (0%)	59 (0%)	37900 (-1%)	29471 (0%)	26281 (0%)
GDBF	3923	3923	6871	14822 (26%)	12024 (97%)	25037 (7%)	44 (-70%)	43 (-62%)	32 (-46%)	11423 (-70%)	11270 (-62%)	14322 (-46%)
PGDBF	5251	5251	8915	16091 (37%)	15224 (150%)	29613 (27%)	45 (-70%)	41 (-62%)	32 (-46%)	11583 (-70%)	10765 (-64%)	14348 (-46%)

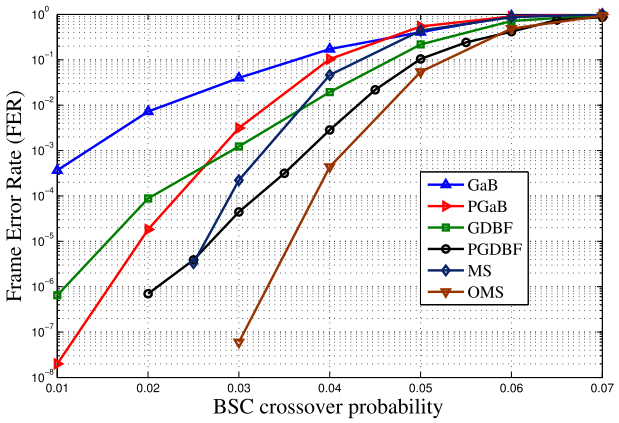

Fig. 9. GaB, PGaB, GDBF, PGDBF, MS, OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4, d_c = 8$, $M = 648$, and Rate $= 0.5$ (presented in [6]).

clock rate for the PGaB is 3.3 times better than GDBF and PGDBF for this code rate. Since our optimized versions of the GDBF and PGDBF implementations do not use registers for the CNUs, PGaB has larger register foot print. However, the slice LUT (logic block) usage is comparable with PGDBF.

VII. ROBUSTNESS ANALYSIS

In the following series of experiments we evaluate the impact of changes in code rate and codeword length on decoding performance and hardware cost over the four algorithms, and demonstrate that PGaB consistently outperforms the GDBF and PGDBF in terms of throughput. Code rate indicates the ratio of data to the length of the codeword that includes the data and parity bits. The higher the code rate, the higher the probability of noise over the communication medium effecting the data portion of the codeword. This increases the stress on the decoding algorithm on correcting errors. As the bandwidth for communications systems increase, the packet lengths (codewords) become longer. Therefore ability to process longer codewords encoded with higher code rates are important criteria for evaluating the efficiency of a decoding algorithm. Furthermore, a change in code rate or codeword length involves modification to the decoder hardware architecture and imposes routability and critical path delay constraints from FPGA implementation point of view. In this section, we summarize the hardware modifications, present performance analysis, and correlate resource requirement and throughput trends with respect to changes in code rate and codeword length.

A. Effect of Code Rate

In this experiment, we change the code rate from 0.5 to 0.75 when the codeword length remains as 1296 bits and the degree of a VNU is four. From hardware implementation perspective, the number of VNUs depend on the codeword length, whereas the number of CNUs and the number of connections per CNU (degree of CNU) depend on the code rate. With fewer number of CNUs, the number of connections per CNU increases. As the code rate increases from 0.5 to 0.75, based on Table III, we observe that the register and Slice LUT resources reduce for the PGaB and GaB. Reduction in the CNU count is the primary reason for reduction in the total logic block usage and register. The maximum clock rate for the new PGaB design is 113 MHz, which is 22% slower. We believe that the degree of the CNU is the primary reason for this performance loss.

Each signal that is generated by the CNU for GaB and PGaB implementations are stored in a register. On the other hand, in our GDBF and PGDBF implementations, the CNU does not include any register. Given that the number of connections per VNU remains the same, even though the code rate changes, there is no additional register demand. Therefore the number of registers used by these implementations remain the same. We observe less than one percent reduction in Slice LUTs for the GDBF and PGDBF implementations with code rate of 0.75 over code rate of 0.5. This results with slight change in maximum clock rate and throughput performance. In overall, the PGaB implementation results with a maximum clock rate that is around 2.6 times better than GDBF and PGDBF. In Figure 10, we compare the FER performance of the GaB and PGaB for the code rate 0.75. The PGaB consistently outperforms the GaB decoder especially in the error-floor region (more than two orders of magnitude at crossover probability of 2×10^{-3}). The PGaB catches the GDBF at α value of 0.04 and performs better than GDBF beyond this point. Considering the FER performance in the error floor region shown in Figure 10, and the negligible loss (0.004%) in throughput performance, we conclude that PGaB decoding and throughput performance is consistent across two code rates.

B. Effect of Codeword Length

Next, we implement the GaB, PGaB, GDBF, and PGDBF for the QC-LDPC code constructed by [26], which has a length of 2212 bits and ($d_v = 4$, $d_c = 28$) with a higher code rate of 0.857. Each CNU has a degree of 28 for all four hardware

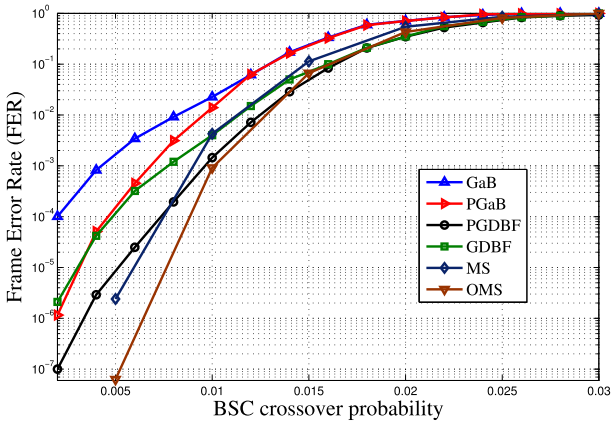

Fig. 10. GaB, PGaB, GDBF, PGDBF, MS, OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4, d_c = 16$, $M = 324$, and Rate $= 0.75$ (presented in [6]).

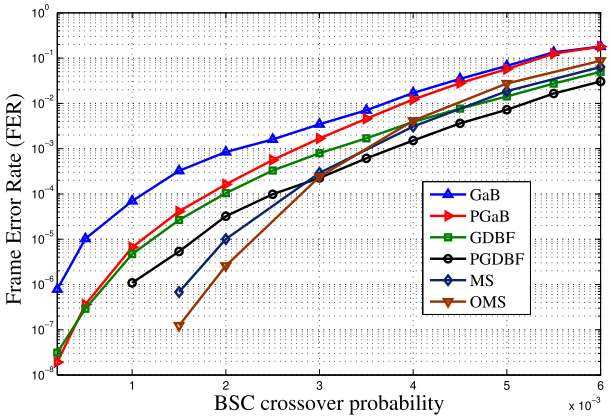

Fig. 11. GaB, PGaB, GDBF, PGDBF, MS, OMS: FER vs. crossover probability for the QC-LDPC code ($d_v = 4, d_c = 28$, $N = 2212$, $M = 312$, and Rate $= 0.857$).

implementations. Note that the total number of inputs per CNU for the GaB and PGaB implementations are 56 as illustrated in Figure 6.

As shown in Table III, resource usage for all design increases significantly compared to the implementations based on the codeword length of 1296 primarily due to the increase in the VNU count. The size of LDPC code affects the complexity of interconnection network [27]. This is reflected in the maximum clock rate for the PGaB implementation, which drops from 113.7MHz to 59.4MHz. However, the rate of the throughput loss is much smaller (by 10.8%), because, throughput is linearly proportional to the length of the codeword as shown in Equation 4 independent from the design. The longer codeword compensates for the reduction in clock rate. In overall for the longer codeword we observe that PGaB achieves higher clock rate and throughout performance with respect to the GDBF and PGDBF.

The slice count for an implementation is widely used as the area metric by FPGA researchers. In Table IV, we show the throughput-to-area ratio (TAR) based on slice count for each algorithm for each code rate. The PGaB consistently results with better TAR performance than both GDBF and PGDBF. Even though GaB and PGaB throughput performances are similar for each code rate, due to higher resource usage of the PGaB, the TAR performance is worse than the GaB. In Table IV, we also compare the normalized throughput (T_p) that is calculated based on fixing the α value for each code rate studied in this paper and using the average number of iterations for that α value. As seen in Table IV, average number of iterations for the PGaB implementation is consistently lower than GaB, GDBF and PGDBF implementations for each code rate. In Table III, 3 we notice that GaB resulted with better throughput than the PGaB implementation. However when we take the actual iteration number into account we observe that the PGaB achieves better throughput than the GaB and the throughput gap between PGaB implementation with respect to GDBF and PGDBF further improves.

Figure 11 shows the FER performance comparison between the GaB, PGaB, GDBF, and PGDBF decoders as a function of the cross-over probability on QC-LDPC $(N, d_v, d_c, R) = (2212, 4, 28, 0.857)$ code with rate of 0.857. We conclude

TABLE IV
THROUGHPUT-TO-AREA RATIO (TAR) AND NORMALIZED THROUGHPUT (T_p) FOR GaB, PGaB, GDBF, AND PGDBF WHEN THE CROSSOVER PROBABILITY IS FIXED

	N, d_v, R=(1296,4,.5)			N, d_v, R=(1296,4,.75)			N, d_v, R=(2212,4,.857)		
	I_{ave} ($\alpha = 0.02$)	T_p ($Mbps$)	TAR	I_{ave} ($\alpha = 0.002$)	T_p ($Mbps$)	TAR	I_{ave} ($\alpha = 0.001$)	T_p ($Mbps$)	TAR
GaB	3.37	28362	2.41	1.14	64857	10.64	1.12	58954	2.53
PGaB	2.78	34078	2.33	1.09	67594	9.48	1.09	60272	2.07
GDBF	3.70	7723	0.52	1.18	23888	1.99	1.16	30892	1.23
PGDBF	5.78	5011	0.31	1.62	16600	1.09	1.44	24885	0.84

that for longer codewords the PGaB can surpass the GaB by more than one order of magnitude at crossover probability of 2×10^{-4} without sacrificing the throughput advantage of the GaB. We observe that, as the code rate increases and as the codeword length increases, the gap between GaB and PGaB on the FER plots shrinks. This trend is expected, since each case stresses the decoder. The important observation here is the superiority of the PGaB for all scenarios considered in this study in terms of decoding performance without sacrificing the throughput performance of the GaB.

VIII. CONCLUSION AND FUTURE WORK

In this paper we quantified the hardware cost and performance of the GaB with probabilistic execution (PGaB decoder). We showed that without a performance loss in throughput, we improved the decoding performance of the GaB significantly and bridged the gap between GaB and other hard decision bit flipping decoding algorithms. Our simulation results showed that the PGaB now even has a better decoding performance than GDBF. In our current designs, we are generating a 1296-bit random number register. We will investigate ways to reduce this register footprint by sharing 1-bit register among a cluster of VNUs. This would reduce the size of the shift register on the datapath and have considerable impact on resource usage. Our evaluations rely on implementation of a new architecture for each codeword and core rate combination. Ability to switch the context from one implementation to another at runtime would allow us to evaluate multi-rate LDPC codes. For future work we plan

to expand the capabilities of our FPGA based simulation testbed with partial reconfiguration and run time configuration to conduct a run-time flexible analysis.

REFERENCES

[1] H. D. Pfister, I. Sason, and R. Urbanke, "Capacity-achieving ensembles for the binary erasure channel with bounded complexity," *IEEE Trans. Inf. Theory*, vol. 51, no. 7, pp. 2352–2379, Jul. 2005.

[2] S. Myung, K. Yang, and J. Kim, "Quasi-cyclic LDPC codes for fast encoding," *IEEE Trans. Inf. Theory*, vol. 51, no. 8, pp. 2894–2901, Aug. 2005.

[3] *NR; Multiplexing and Channel Coding*, document 3GPP TS 38.212 V15.0.0, 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Dec. 2017.

[4] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Trans. Commun.*, vol. 47, no. 5, pp. 673–680, May 1999.

[5] X. Wu, Y. Song, M. Jiang, and C. Zhao, "Adaptive-normalized/offset min-sum algorithm," *IEEE Commun. Lett.*, vol. 14, no. 7, pp. 667–669, Jul. 2010.

[6] B. Ünal, F. Ghaffari, A. Akoglu, D. Declercq, and B. Vasić, "Analysis and implementation of resource efficient probabilistic Gallager B LDPC decoder," in *Proc. 15th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2017, pp. 333–336.

[7] R. G. Gallager, *Low Density Parity Check Codes*. Cambridge, MA, USA: MIT Press, 1963.

[8] V. V. Zyablov and M. S. Pinsker, "Estimation of the error-correction complexity for Gallager low-density codes," *Problems Inf. Transmiss.*, vol. 11, no. 1, pp. 18–28, Jul. 1975.

[9] T. Wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, and I. Takumi, "Gradient descent bit flipping algorithms for decoding ldpc codes," *IEEE Trans. Communu.*, vol. 58, no. 6, pp. 1610–1614, Jun. 2010.

[10] F. Ghaffari, A. Akoglu, B. Vasic, and D. Declercq, "Multi-mode low-latency software-defined error correction for data centers," in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Aug. 2017, pp. 1–8.

[11] O. Al Rasheed, P. Ivaniš, and B. Vasić, "Fault-tolerant probabilistic gradient-descent bit flipping decoder," *IEEE Commun. Lett.*, vol. 18, no. 9, pp. 1487–1490, Sep. 2014.

[12] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.

[13] R. M. Tanner, D. Sridhara, and T. Fuja, "A class of group-structured ldpc codes," in *Proc. Int. Conf. Space-Time Absoluteness (ICSTA)*, Jul. 2001, pp. 365–370.

[14] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. 27, no. 5, pp. 533–547, Sep. 1981.

[15] B. Vasić, S. K. Chilappagari, and D. V. Nguyen, "Failures and error floors of iterative decoders," in *Channel Coding: Theory, Algorithms, and Applications: Academic Press Library in Mobile and Wireless Communications*. New York, NY, USA: Academic, Jun. 2014, pp. 299–341.

[16] B. Vasić, P. Ivaniš, K. L. Trung, and D. Declercq, "Approaching maximum likelihood performance of LDPC codes by stochastic resonance in noisy iterative decoders," in *Proc. Inf. Theory Appl. Workshop (ITA)*, Feb. 2016, pp. 1–9.

[17] F. Leduc-Primeau, S. Hemati, S. Mannor, and W. J. Gross, "Dithered belief propagation decoding," *IEEE Trans. Commun.*, vol. 60, no. 8, pp. 2042–2047, Aug. 2012.

[18] C. Leroux, S. Hemati, S. Mannor, and W. J. Gross, "Stochastic chase decoding of Reed-Solomon codes," *IEEE Commun. Lett.*, vol. 14, no. 9, pp. 863–865, Sep. 2010.

[19] K. Le, F. Ghaffari, D. Declercq, and B. Vasić, "Efficient hardware implementation of probabilistic gradient descent bit-flipping," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 64, no. 4, pp. 906–917, Apr. 2017.

[20] P. Ivaniš and B. Vasić, "Error errore eicitur: A stochastic resonance paradigm for reliable storage of information on unreliable media," *IEEE Trans. Commun.*, vol. 64, no. 9, pp. 3596–3608, Sep. 2016.

[21] N. Deák, T. Györfi, K. Márton, L. Vacariu, and O. Cret, "Highly efficient true random number generator in FPGA devices using phase-locked loops," in *Proc. 20th Int. Conf. Control Syst. Comput. Sci.*, May 2015, pp. 453–458.

[22] J. Lan, W. L. Goh, Z. H. Kong, and K. S. Yeo, "A random number generator for low power cryptographic application," in *Proc. Int. SoC Design Conf.*, Nov. 2010, pp. 328–331.

[23] K. Le *et al.*, "Efficient realization of probabilistic gradient descent bit flipping decoders," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 1494–1497.

[24] B. Yuce, H. F. Ugurdag, S. Gören, and G. Dündar, "Fast and efficient circuit topologies for finding the maximum of n k-bit numbers," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 1868–1881, Aug. 2014.

[25] T. Nguyen-Ly *et al.*, "FPGA design of high throughput LDPC decoder based on imprecise offset min-sum decoding," in *Proc. IEEE 13th Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2015, pp. 1–4.

[26] J. Zhang, J. Wang, S. G. Srinivasa, and L. Dolecek, "Achieving flexibility in LDPC code design by absorbing set elimination," in *Proc. 45th Asilomar Conf. Signals, Syst. Comput. (ASILOMAR)*, Nov. 2011, pp. 669–673.

[27] A. Darabiha, A. C. Carusone, and F. R. Kschischang, "Block-interlaced LDPC decoders with reduced interconnect complexity," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 55, no. 1, pp. 74–78, Jan. 2008.

Burak Unal (S'07) received the B.Sc. degree (Hons.) in electrical and electronics engineering from Erciyes University, Kayseri, Turkey, in 2008, the M.Sc. degree in electrical and computer engineering from the University of Arizona, Tucson, AZ, USA, in 2013, where he is currently pursuing the Ph.D. degree. His current research interests include design, development, and analysis of low-complexity iterative decoding algorithms for error correction codes (low-density parity check) and their efficient hardware architectures for wireless communication systems.

Ali Akoglu (M'06) received the Ph.D. degree in computer science from the Arizona State University in 2005. He is currently an Associate Professor with the Department of Electrical and Computer Engineering and the BIO5 Institute, University of Arizona. He is also the Site-Director of the National Science Foundation, Industry-University Cooperative Research Center on Cloud and Autonomic Computing. His research interests lie in the fields of high performance computing, reconfigurable computing, and cloud computing with the goal of solving the challenges of bridging the gap between the domain scientist, programming environment, and highly-parallel hardware architectures.

Fakhreddine Ghaffari (M'12) received the degree in electrical engineering and the master's degree from the National School of Electrical Engineering (ENIS), Tunisia, in 2001 and 2002, respectively, and the Ph.D. degree in electronics and electrical engineering from the University of Sophia Antipolis, France, in 2006. He is currently an Associate Professor with the University of Cergy-Pontoise, France.

His research interests include VLSI design and implementation of reliable digital architectures for wireless communication applications in ASIC/FPGA platform and the study of mitigating transient faults from algorithmic and implementation perspectives for high-throughput applications.

Bane Vasić (M'91–F'12) is currently a Professor of electrical and computer engineering and mathematics with the University of Arizona and the Director of the Error Correction Laboratory. He is a da Vinci Fellow. He is a past Chair of the IEEE Data Storage Technical Committee.

He is a Co-Founder of Codelucida, a startup company developing advanced error correction solutions for communications and data storage. He is an inventor of the soft error-event decoding algorithm and the key architect of a detector/decoder for Bell Labs data storage read channel chips, which were regarded as the best in industry.