Abstract—Emerging applications such as flash-based storage systems and 10 gigabit Ethernet require that there is no error floor even at bit error rates as low as $10^{-12}$ or so. It has been found that trapping sets are responsible for the error floors of many LDPC codes with AWGN channels. This paper presents a hardware based backtracking scheme to break the trapping sets at runtime for lowering the error floor of quasi-cyclic LDPC codes. Backtracking is implemented as a self-contained module that can be interfaced to any generic reconfigurable iterative decoder for QC-LDPC codes. The backtracking module and a reconfigurable decoder are implemented with a FPGA and an 180 nm standard cell library. The results indicate that the overhead of backtracking is modest—about 5% in terms of logic and 13% in terms of memory for the first level backtracking and 14% in terms of logic and 46% in terms of memory for a two-level backtracking scheme. Furthermore, it is shown that the increase in latency due to backtracking is modest in the average case and can be controlled by the system designer by choosing the appropriate values for the number of trials and the number of iterations of the backtracking module.

Index Terms— error floor, low-density parity-check (LDPC) codes, reconfigurable computing, trapping set, very large scale integration.

I. INTRODUCTION

LOW-DENSITY parity-check (LDPC) codes, discovered by Gallager in 1962 [1], were rediscovered and shown to approach Shannon capacity in the late 1990s [2]. LDPC codes are being considered for a wide range of applications such as high-density flash memory [3], satellite broadcasting, WiFi, and mobile WiMAX. With iterative decoding, most LDPC codes (including many standard codes for wireless communications) perform poorly at very low error rates. It is found that suddenly, the error probability of a code in the high signal-to-noise (SNR) region drops at a rate much slower than that in the region of low to moderate SNR (or even stops to drop). This is referred to as an error floor [4]. It is very difficult to construct codes that do not exhibit an error floor at bit error rates as low as $10^{-12}$ or so. However, many emerging applications such as optical communication, 10 gigabit Ethernet, and NASA Landsat data continuity mission [5], require bit-error rates below $10^{-12}$. Data storage systems require even lower bit-error rates, say $10^{-15}$ or below. Even in applications that do not require very low bit error rates, such as wireless communications, high error floors are very undesirable because they degrade quality of service [6]. Thus, there is a compelling need to lower the error floor in LDPC decoders.

It has been found that trapping sets are responsible for the error floors of many LDPC codes with AWGN channels [4]. Constructing codes to avoid harmful trapping sets is a hard combinatorial problem, so decoder-based strategies are explored to lower down the error floor of codes [7]–[9]. In [7], a variety of decoder-based strategies have been proposed, including bit-pinning, generalized-LDPC decoding, concatenation, bi-mode decoding, and combinations of these methods. Although these methods work well with some codes, they have limitations and disadvantages. Concatenation not only requires a second decoder but also results in a loss of code rate and coding gain in the waterfall region. Bit-pinning and bi-mode decoding are usually customized for specific codes, hence they need a detailed knowledge of the dominant (the most harmful) trapping sets, which may be difficult to obtain for long or even moderate long codes or codes whose parity-check matrices that are not so sparse. Furthermore, [7] does not address hardware implementation issues underlying these schemes. A postprocessing technique with selectively biased messages is proposed in [8], [9]. However, this method works well only when detailed knowledge of dominant absorbing sets or trapping sets are obtained.

Instead of devising methods customized to a specific code, the backtracking algorithm proposed in [10], [11] does not require detailed knowledge of dominant trapping sets and can be used to decode codes constructed based on a wide range of pseudorandom and algebraic methods. It is shown that the backtracking algorithm can break most of the trapping sets in the decoding process and push the error floor down to a level very close to that limited by the minimum distance of a code [11]. However, the challenges of implementing such the backtracking scheme efficiently in hardware has not been addressed.

Over the years, there has been a plethora of work on the hardware implementation of iterative decoders for LDPC codes [12]–[24]. However, most (if not all) of the existing work contribute in improving the throughput and reducing the area (including memory requirements) of the decoder. Decoders that...
lower down the error floor of LDPC codes is rarely implemented in hardware, except for [6]. However, the decoder described in [6] is designed specifically for the (2048, 1723) code (adopted by the IEEE 802.3an standard) with the (8, 8) trapping set as the dominant trapping set and thus lacks flexibility. The implementation proposed in this paper is integrated with a reconfigurable (or flexible) quasi-cyclic LDPC decoder, therefore the same decoder can be used to lower the error floor of a family of structurally compatible quasi-cyclic LDPC codes. Furthermore, the backtracking based implementation used in this paper does not require detailed knowledge of the dominant trapping sets, so it can be used with codes constructed by a variety of methods.

It is impractical to implement the algorithm described in [11] for the following two reasons. First, there is a computation-intensive correlation operation in the first backtracking stage of the algorithm proposed in [11], which requires multi-input addition of multibit messages. This correlation operation increases the area required for implementation significantly and reduces the throughput. Second, for the decoder to be of practical use, the additional latency imposed by backtracking has to be minimized. The original algorithm does not address the latency issue. Furthermore, the hardware implementation of the backtracking module should be amenable for integration with a generic high-performance iterative decoder and should be independent of the specific code being decoded, i.e., the same backtracking module should be capable of being used with any code.

This paper proposes a hardware-efficient architecture that addresses these challenges and thereby develops a hardware oriented variation of the backtracking algorithm that can be used by a system designer. This paper makes two major contributions—First, a Hamming distance criteria is introduced as a replacement for correlation function, which decreases the hardware complexity significantly without compromising the performance; Second, techniques are developed to reduce the number of iterations by as much as 75% compared to the direct implementation of the algorithm in [11]. We demonstrate that though the worst case latency of the backtracking scheme is very high, the average case latency can be modest and practical for many applications. Furthermore, two key parameters of backtracking—the number of iterations and number of trial nodes, can be selected by the system designer to meet the overall system level latency requirements of an application.

The rest of the paper is organized as follows. An overview of quasi-cyclic LDPC codes and iterative decoding with backtracking is presented in Section II. The techniques to reduce complexity and latency are described in Section III and Section IV, respectively. The architecture of a reconfigurable LDPC decoder with backtracking is described in Section V and the results are presented in Section VI. Section VII concludes the paper.

II. BACKGROUND

A. Quasi Cyclic LDPC Codes

A binary LDPC code \( C \) of length \( N \) is given by the null space of an \( M \times N \) sparse parity-check matrix \( H = [h_{i,j}] \) over GF(2).

If each column has constant weight \( d_c \) (the number of 1-entries in a column, also known as check node degree) and each row has constant weight \( d_r \) (the number of 1-entries in a row, also known as variable node degree), the LDPC code \( C \) is referred to as a \((d_c, d_r)\)-regular LDPC code. If the columns and/or rows of the parity-check matrix \( H \) have multiple weights, then the null space of \( H \) gives an irregular LDPC code. A binary \( N \)-tuple \( u = (u_0, u_1, \ldots, u_{N-1}) \) is a codeword in \( C \) if and only if \( uH^T = 0 \).

The LDPC code \( C \) is represented graphically by a Tanner graph which consists of \( N \) variable nodes (VNs) and \( M \) check nodes (CNs).

In terms of encoding and decoding implementation, the most advantageous structure of an LDPC code is the quasi-cyclic (QC) structure. QC-LDPC codes are adopted by many communications standards, e.g. the IEEE 802.11n (Wi-Fi), the IEEE 802.16e (WiMAX), and the (8176,7156) code for the NASA Landsat data continuity mission [5]. The parity-check matrix for a QC-LDPC code is a \( N_c \times N_r \) two-dimensional array (or block) of circulants or circulant permutation matrices (CPMs) and/or zero matrices of the same size, say \( n \times m \). Fig. 1 shows the parity-check matrix structure of the rate-3/4 (1944,1458) QC-LDPC code from the IEEE 802.11n standard.

B. Iterative Decoding With Backtracking

Iterative message passing algorithms, such as the sum-product algorithm (SPA) [2] and normalized min-sum algorithm (NMSA) [25], are widely used to decode LDPC codes. With these iterative decoding algorithms, VNs in the Tanner graph are usually initialized with intrinsic messages (also known as log likelihood ratios (LLRs)) sampled at the channel. For each decoding iteration, the CNs and VNs exchange and compute extrinsic messages on the edge of Tanner graph. The decoding process is terminated when either \( uH^T = 0 \) or the maximum number of iterations allowed has been reached.

Iterative decoding algorithms are based on exchanging messages computed by CNs and VNs using the local information available to those CNs and VNs, and thus are susceptible to trapping sets, which in turn could lead to poor performance at very low error rates. A trapping set \( TS(a,b) \) is a set \( T \) of \( a \) VNs in a code’s Tanner graph which induces a subgraph with \( b \) odd-degree CNs. Suppose, in transmission of a codeword, an error pattern with \( a \) errors at the locations of the \( a \) VNs of a trapping set \( TS(a,b) \) occurs. This error pattern will cause \( b \) parity-check failures (i.e., the check-sums are not equal to zeros). In expressing a trapping set, it is sometimes desirable to display both its set \( T \) of VNs and its set \( C \) of odd-degree CNs. Hence, a trapping set is also denoted by \( (T; C) \). Fig. 2 shows a (7,1) trapping set with \( T = \{v_0, v_1, \ldots, v_6\} \) and \( C = \{c_0\} \). The CN \( c_0 \) is adjacent only to one VN \( v_0 \). For the (7,1) trapping set shown in Fig. 2, if an error-pattern with 7 errors at the locations of the 7 VNs, \( v_0, v_1, \ldots, v_6 \) occurs, then a single check-sum corresponding to the CN \( c_0 \) will fail.

Let \( K_{\text{MAX}}^{(0)} \) be the maximum number of iterations to be performed in the original decoding. Since there may be oscillations in decoding iterations when the decoder falls into a small trapping set, we keep track of the minimum set of unsatisfied CNs, \( C_0 \), of all iterations up to \( K_{\text{MAX}}^{(0)} \) and its corresponding decoder output \( u_0 \). If \( |C_0| \leq \tau \), where \( \tau \) is predetermined threshold, we say that the decoder falls into a small trapping set. The choice
of \( \tau \) depends on the specific LDPC code to be decoded, the required decoding performance and the acceptable additional latency. Usually, \( \tau \) is no greater than ten based on the simulation results shown in [11].

Let \( N(C_0) \) be the set of VNs connected to the unsatisfied CNs in \( C_0 \). We refer to the VNs in \( N(C_0) \) as the candidate VNs. It is very likely that some candidate VNs are also contained in the trapping set. The backtracking scheme is a trial-and-error process to identify these VNs and is shown as follows. For each VN \( v \in N(C_0) \), let \( u_{0,v} \) be the bit in \( u_0 \) that corresponds to \( v \). In order to perturb the error-inducing pattern in the decoder input as much as possible, we set the input LLR at the VN \( v \) to the maximum possible LLR value with a polarity opposite to that of its corresponding bit \( u_{0,v} \) in \( u_0 \). Then, we start the decoding process up to \( K_{\text{max}}^{(1)} \) iterations with \( K_{\text{max}}^{(1)} \leq K_{\text{max}}^{(0)} \). If the assumption that \( v \) is in the trapping set is wrong, the error-inducing pattern in the decoder input LLR vector is most likely not affected and hence the redecoding will most likely fail again; otherwise, we expect a decoding success with high probability. If redecoding fails, we choose another node \( v' \in N(C_0) \) as a trial node and start another redecoding process. This flipping and redecoding process continues until either a codeword is found or all the VNs in \( N(C_0) \) are tested. Each testing of a VN in \( N(C_0) \) is called a trial.

This first backtracking works very well for many LDPC codes but cannot always correct a failure of the original decoding. If, in a trial, a bit corresponding to a trapping set node is flipped, we say the trial is a hit-trial. It is found in [11] that the information contained in the hit trials can be used to repeat the backtracking to overcome the failures in the first backtracking. The second round of backtracking is carried out as follows.

Assume that all the \( |N(C_0)| \) trials in the first backtracking fail, and the \( i \)-th trial fails with decoder output \( u_i \) and minimum set of unsatisfied CNs \( C_i \), \( 1 \leq i \leq |N(C_0)| \). It is shown that a trial whose unsatisfied CNs are different from \( C_0 \) could be the hit-trial for most of the time [11]. Let \( \Gamma = \{ i : 1 \leq i \leq |N(C_0)| \text{ and } u_i \neq u_0 \} \). Assume BPSK transmission, each code bit \( u \) is mapped into \( \chi(u) = 1 - 2u \), i.e., \( \chi(0) = +1 \) and \( \chi(1) = -1 \). Since a hit-trial is very likely to result in the minimum number of error bits among all the trials, the index of the trial that is most likely a hit-trial is given by the inner product \( \langle y, \chi(u) \rangle = \sum_{i=1}^{|N(C_0)|} y_i \chi(u_i) \), \( y \) and \( x \) are used as the candidates for a hit-trial. Let \( \{ C_i^* : 1 \leq i \leq \gamma \} \) be the set of sets of unsatisfied CNs and the set of decoder outputs associated to the \( \gamma \) most likely trials found in the first backtracking. Then, in the second backtracking, for each of the \( \gamma \) most likely hit-trials, we perform a decoding process similar to the first backtracking process. Each trial is performed with up to \( K_{\text{max}}^{(2)} \) iterations. The parameter \( \gamma \) should be chosen to ensure a high successful decoding rate of the second backtracking and a further error floor performance improvement, while minimizing the latency overhead.

### Fig. 1.
The parity-check matrix of the rate-3/4 (1944,1458) QC-LDPC code (from the IEEE 802.11n standard) consists of a \( 6 \times 24 \) two-dimensional array of 81 \( \times \) 81 CPMs and zero matrices. The row weight for each block row (also known as VN degree) is 14. The column weight for each block column (also known as VN degree) varies from 2, 3, and 6. The 84 CPMs are represented by shadowed square box. The number inside each box denotes the CPM offset, which is the position of nonzero entry in the first CPM row. The 60 zero matrices are represented by white square box.

### Table 1
The parity-check matrix of the rate-3/4 (1944,1458) QC-LDPC code (from the IEEE 802.11n standard) consists of a \( 6 \times 24 \) two-dimensional array of 81 \( \times \) 81 CPMs and zero matrices. The row weight for each block row (also known as VN degree) is 14. The column weight for each block column (also known as VN degree) varies from 2, 3, and 6. The 84 CPMs are represented by shadowed square box. The number inside each box denotes the CPM offset, which is the position of nonzero entry in the first CPM row. The 60 zero matrices are represented by white square box.
III. HAMMING DISTANCE BASED REDUCED-COMPLEXITY HIT-TRIAL SELECTION CRITERIA

The correlation function \( \langle \mathbf{y}, \mathbf{c}(\mathbf{u}) \rangle \) introduced in Section II-B requires multi-input addition of multibit messages in implementation, and thus is not cost-effective in terms of both silicon area and critical path delay. To reduce the complexity of the backtracking algorithm, we propose to use the Hamming distance between the hard decision bits of the input LLR and the decoder output of each trial in the first backtracking, i.e., \( d(\mathbf{sgn}(\mathbf{y}), \mathbf{u}) \), as the hit-trial selection criteria. For \( 0 \leq i < N \), when \( y_i \geq 0 \), \( \mathbf{sgn}(y_i) = 0 \); otherwise, \( \mathbf{sgn}(y_i) = 1 \). \( \gamma \) trials with minimal \( d(\mathbf{sgn}(\mathbf{y}), \mathbf{u}) \) are considered as the most likely hit-trials. This new criteria greatly reduces the implementation cost of backtracking algorithm, since the multibit additions in the original backtracking scheme are replaced by single-bit addition.

To show the efficacy of this heuristic, we evaluate the error performance of every code from the 802.11n and 802.16e standard using FPGA-based emulation [18]. The input LLR, the extrinsic message, and the updated LLR of the fixed-point FPGA-based decoder are of 6-bit, 6-bit, and 8-bit, respectively. Normalized min-sum algorithm is used for implementation. For each simulation point (SNR), 500 block errors and their associated decoder input LLRs are collected. We choose the codes from these wireless standards for three reasons. First, although these codes are not designed for low-error floor applications, their error performance are significantly affected by small trapping sets. Second, these codes have higher error floors (usually at block error rate (BLER) of \( 10^{-6} \)) and short code length (up to 2304), thus their error performance can be simulated by FPGA within hours. However, emulation of codes designed specifically for low-error floor application (BLER of \( 10^{-12} \) or so) will take months. Third, the 802.11n and 802.16e standards provide 126 codes with various lengths and code rates, a comprehensive simulation on these codes constitutes a no-bias study.

We apply the original backtracking algorithm and the improved backtracking algorithm to these captured block errors with the same simulation parameters \( (K_{\text{max}} = 10, \tau = 10, \gamma = 3) \), and find that these two algorithms result in exactly the same number of corrected block errors. We summarize the error performance of four codes in Table I.

<table>
<thead>
<tr>
<th>Code Parameter</th>
<th>SNR</th>
<th>No. of Corrected Block Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Rate</td>
<td>Length</td>
<td>Correlation</td>
</tr>
<tr>
<td>802.11n 1/2</td>
<td>1944</td>
<td>2.6</td>
</tr>
<tr>
<td>802.11n 2/3</td>
<td>1944</td>
<td>3.3</td>
</tr>
<tr>
<td>802.16c 1/2</td>
<td>1056</td>
<td>3.5</td>
</tr>
<tr>
<td>802.16c 1/2</td>
<td>2304</td>
<td>2.4</td>
</tr>
</tbody>
</table>

IV. LATENCY REDUCTION TECHNIQUES FOR THE BACKTRACKING ALGORITHM

Obviously, iterative decoding with backtracking requires more iterations to decode a received codeword compared to the original decoding. The simulation data in [11] indicates that in the worst case, tens of trials are required for successful decoding. As each trial consists of tens of iterations, the backtracking latency could be in the order of hundreds of iterations. This overhead could result in a high latency for decoding certain codewords, which in turn complicates the system level design of receiver because it may entail additional buffering and implementation of flow control.

We propose two techniques to reduce the latency introduced by backtracking, which are described next. The first technique is early trap detection, which effectively reduces the number of iterations for each trial. The second technique is variable node degree based flipping scheme, which decreases the number of trials when decoding irregular codes. We also explore the trade-off between the number of iterations and the number of trial nodes used on the performance, so that the additional latency incurred by backtracking can be customized to a given application.

A. Early Trap Detection

When a LDPC decoder is trapped, there are two observable patterns in terms of the number of unsatisfied CNs over iterations: this number either stops to change, or oscillate around a certain minimum value. For example, if we record the number of unsatisfied CNs on the iteration basis, the pattern would be either like \{\ldots, 36, 18, 14, 10, 10, 10, \ldots\}, or like \{\ldots, 36, 18, 14, 10, 11, 12, 10, 13, 11, 10, \ldots\}. For the first case, the minimum number of unsatisfied CNs (10) is repeated; while for the latter case, the number of unsatisfied CNs oscillates around the minimum value (10).

Based on this observation, we propose an early trap detection method. For the original decoding or one trial in the first backtracking, let \( C_{\text{min}} \) be the set of unsatisfied CNs with the minimum \( |C_{\text{min}}| \); let \( C_{\text{cur}} \) be the set of unsatisfied CNs for the current iteration, let \( N_{\text{rep}} \) be the number of repetition times of \( |C_{\text{min}}| \). In the first iteration, \( C_{\text{min}} \) is set by \( C_{\text{cur}} \), and \( N_{\text{rep}} \) is set by one. Afterwards, \( C_{\text{min}} \) and \( N_{\text{rep}} \) are updated on the iteration basis: if \( |C_{\text{cur}}| = |C_{\text{min}}| \), \( N_{\text{rep}} \) is increased by one; if \( |C_{\text{cur}}| < |C_{\text{min}}| \), \( C_{\text{min}} \) is set by \( C_{\text{cur}} \), and \( N_{\text{rep}} \) is reset to one; otherwise, no change is applied. If \( N_{\text{rep}} \) reaches \( N_{\text{rep}}^{\text{max}} \), a predetermined threshold, the decoder is declared as been trapped, and thus is stopped early.

B. Variable Node Degree Based Flipping Scheme

The original backtracking algorithm does not specify the order to flip the candidate VNs. A straightforward flipping scheme is to flip the candidate VNs from leftmost column to the rightmost column. However, for irregular codes which contains VNs of various degrees, the small-degree VNs are more likely to fall into small trapping sets than the large-degree VNs, since they are not well informed from their limited number of connected CNs. Based on this observation, we propose a VN degree based flipping scheme that gives priority to the small-degree VNs when choosing the candidate VN to flip. For example, consider the code shown in Fig. 1. When the
proposed flipping scheme is applied, the candidate VNs in the 19th block column are flipped first, since it is the first block column with smallest VN degree two. After all the candidate VNs with degree two are tested with selective flipping, we flip the candidate VNs with degree three, and so on.

C. Simulation Results

To show the effectiveness of the proposed techniques, we evaluate the error performance of three codes from the IEEE 802.11n standard and three codes from the IEEE 802.16e standard. Let $N_{\text{max}}^{\text{imp}}$ and $N_{\text{max}}^{\text{tri}}$ be the maximum number of permitted trials and overall iterations, respectively. The simulation parameters are $K_{\text{max}}^{(0)} = K_{\text{max}}^{(1)} = 20$, $\tau = 10$, $\gamma = 0$ (the second backtracking is disabled), $N_{\text{max}}^{\text{tri}} = 15$ (up to 15 candidate VNs can be selectively flipped), $N_{\text{max}}^{\text{det}} = \infty$ (no constraint for the number of overall iterations), $N_{\text{rep}} = 5$. As stated in the previous section, 500 block errors are captured for each simulation point.

The results of error performance and latency are shown in Table II. The error performance is measured by the number of corrected block errors. The latency is measured by the number of iterations performed. Then, $K_{\text{max}}^{(1)}$ is close to $G_0$. Since we require $|G_0| \leq \tau$ for $G_0$ to be considered small in the first backtracking, then for a given LDPC code, the choice of the parameters $\tau$ and $\gamma$ has an impact on the computational overhead. Since backtracking is only invoked at a relatively low block error rate $P(B)$, say $P(B) \leq 10^{-5}$. Hence, for moderate $\tau$ and $\gamma$, we have $P(B)(M_1 + M_2) \ll 1$ and hence $K_{\text{lat,ave}}$ is very close to $K_{\text{ave}}$. This means that the multiple trials in the proposed double-backtracking setup in iterative decoding gives only a marginal overall computational overhead.

Next we show that the result above holds in practice. We pick four codes from wireless standards, and apply the backtracking algorithm on them with the following simulation parameters: $K_{\text{max}}^{(0)} = K_{\text{max}}^{(1)} = K_{\text{max}}^{(2)} = 20$, $\tau = 10$, $\gamma = 3$, $N_{\text{max}}^{\text{tri}} = \infty$ (no constraint for the number of trials), $N_{\text{max}}^{\text{det}} = \infty$ (no constraint for the number of overall iterations). The results of average decoding latencies before and after backtracking are shown in Table III. Notice, that the average increase in latency is very modest, almost negligible, for this application.

However, in the worst case the additional latency due to backtracking can be quite high. We show that limiting the number of trials and the number of overall iterations for these trials provides an effective tradeoff between latency and error performance. Let $P_B$ and $P_{B,\text{im}}$ be the number of BLER for decoding with and without backtracking. Then, $P_{B,\text{im}}/P_B$ represents the BLER improvement due to backtracking. We examine the relationship between the BLER improvement and the number of trials and number of overall iterations. The BLER improvement is plotted against the number of trials and the number of overall iterations in Fig. 3 and Fig. 4, respectively. In Fig. 4, the number of overall iterations is plotted at an interval of 20. Notice that when $N_{\text{iter}}^{\text{max}} = 50$ and $N_{\text{trial}}^{\text{max}} = 5$, the improvement in performance is quite significant. In general, this shows us that the

<table>
<thead>
<tr>
<th>code parameter</th>
<th>SNR</th>
<th>no. of corrected block error</th>
<th>latency (no. of iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>rate</td>
<td>length (dB)</td>
<td>conventional backtracking</td>
</tr>
<tr>
<td>802.11n</td>
<td>2/5</td>
<td>1944</td>
<td>[11]</td>
</tr>
<tr>
<td></td>
<td>2/3</td>
<td>1944</td>
<td>early trap detection</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>early trap detection &amp; VN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>degree based flipping</td>
</tr>
<tr>
<td>802.16e</td>
<td>1/2</td>
<td>2304</td>
<td>403 (91%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>107.4 (100%)</td>
</tr>
<tr>
<td>802.16e</td>
<td>2/3A</td>
<td>2304</td>
<td>325 (99%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>92.8 (100%)</td>
</tr>
<tr>
<td>802.16e</td>
<td>3/4A</td>
<td>2304</td>
<td>427 (100%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>93.2 (75%)</td>
</tr>
</tbody>
</table>

Let $M_2$ be the average degree of CNs in the Tanner graph of the code to be decoded. Then $M_2 = N(C_0) \approx |G_0| / d_c$ and

$$M_2 \approx \sum_{l=1}^{\gamma} N(C_l^1) \cdot d_c.$$  

(2)

Usually $|C_l^1|$ is close to $|G_0|$. Since we require $|G_0| \leq \tau$ for $G_0$ to be considered small in the first backtracking, then for a given LDPC code, the choice of the parameters $\tau$ and $\gamma$ has an impact on the computational overhead. Since backtracking is only invoked at a relatively low block error rate $P(B)$, say $P(B) \leq 10^{-5}$. Hence, for moderate $\tau$ and $\gamma$, we have $P(B)(M_1 + M_2) \ll 1$ and hence $K_{\text{lat,ave}}$ is very close to $K_{\text{ave}}$. This means that the multiple trials in the proposed double-backtracking setup in iterative decoding gives only a marginal overall computational overhead.

D. Latency Analysis

As we have seen, iterative decoding with backtracking requires more iterations to decode a codeword compared to the traditional iterative decoding. Suppose, for conventional iterative decoding (without backtracking) at a specific SNR, the block error rate is $P(B)$ and the average numbers of iterations to decode a received word is $K_{\text{ave}}$. Let $M_1$ and $M_2$ be number of trials required in the first and second backtracking, respectively. Assume that for each trial in backtracking, the maximum number of iterations performed is set to $K_{\text{max}}$, same as the original decoding. Then, the average number of iterations required for the proposed iterative decoding scheme with double-backtracking setup is upper bounded by

$$K_{\text{lat,ave}} \leq K_{\text{ave}} + P(B)(M_1 + M_2)K_{\text{max}}.$$  

(1)
parameters for the backtracking algorithms can be tuned to meet the system-level requirements. The decoding module is usually a part of a larger system, so depending on what else exists in the system, there may be a slack in timing that can be used to pick the appropriate values for the number of trials and number of iterations.

The hardware efficient backtracking algorithm that incorporates the techniques introduced in this paper is shown in Algorithm 1. Next, we compare the performance of the improved backtracking algorithm tonal algorithm described in [11]. Table II shows the benefit of just early trap detection and VN node based flipping. It is evident that the original algorithm did not address latency optimization while the algorithm proposed here is specifically geared to address this problem. Note that the latency can be reduced by as much as 75% in some cases.

**Algorithm 1** The Improved Backtracking Algorithm

1: Initialization: (from the original decoding with $K_{\text{max}}^{(1)}$ as the maximum number of iterations)
2: $C_0$: the minimum set of unsatisfied CNs
3: $N(C_0)$: VNs neighboring $C_0$
4: $u_0 = (u_{0,0}, \ldots, u_{0,N-1})$: decoder output corresponding to $C_0$
5: $y = (y_0, \ldots, y_{N-1})$: decoder input LLR vector
6: $\eta$: the largest possible positive LLR value
7: $T_{\text{trial}}^{\text{max}}$: the maximum number of trial times
8: (First Backtracking)
9: $i \leftarrow 0; \Gamma \leftarrow \emptyset$
10: for each $v \in N(C_0), 0 \leq v < n$ do
11: $i \leftarrow i + 1$
12: $y' \leftarrow y; y_0' \leftarrow -\chi(u_{0,v}) \cdot \eta$
13: redecode using $y'$ as input LLR and $K_{\text{max}}^{(1)}$ as the maximum number of iterations, which outputs $u_i$ with unsatisfied CNs in $C_i$
14: if $|C_i| = 0$ then
15: stop and exit, output $u_i$ as the decoded codeword
16: else if $i = T_{\text{trial}}^{\text{max}}$ then
17: stop and exit, claim uncorrectable error
18: else if $u_i \neq u_0$ then
19: $\lambda_i \leftarrow d(\text{sgn}(y), u_i)$; insert $i$ into set $\Gamma$
20: end
21: end
22: if second backtracking is disabled then
23: stop and exit, claim uncorrectable error
24: end
25: find the $\gamma$ smallest in $\{\lambda_i\}_{i \in \Gamma}$, with the $\gamma$ corresponding $C_i$'s as $\{C_i^\gamma\}_{i=1}^\gamma$, and the $\gamma$ corresponding decoder outputs $\{u_i^\gamma\}_{i=1}^\gamma$.

---

**Table III** The Average Decoding Latency Overhead is Very Small When Compared with the Original Decoding. Also, a High Rate of Successful Decoding is Observed from Our Experiments if Backtracking is Invoked

<table>
<thead>
<tr>
<th>code length</th>
<th>802.11n</th>
<th>802.11n</th>
<th>802.11n</th>
<th>802.11n</th>
</tr>
</thead>
<tbody>
<tr>
<td>code rate</td>
<td>1/2</td>
<td>2/3</td>
<td>1/2</td>
<td>2/3</td>
</tr>
<tr>
<td>SNR (dB)</td>
<td>2.6</td>
<td>3.3</td>
<td>3.5</td>
<td>2.4</td>
</tr>
<tr>
<td>No. of sim. blocks</td>
<td>$4.4 \times 10^8$</td>
<td>$1.7 \times 10^8$</td>
<td>$6.7 \times 10^8$</td>
<td>$3.2 \times 10^8$</td>
</tr>
<tr>
<td>No. of block errors</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>

**Fig. 3** The BLER improvement in terms of the number of trials.

**Fig. 4** The BLER improvement in terms of the number of iterations.
This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

CHEN et al.: HARDWARE IMPLEMENTATION OF A BACKTRACKING-BASED RECONFIGURABLE DECODER 7

Fig. 5. The overall architecture for backtracking-based reconfigurable decoder that supports CPM size of up to $P_M = 96$. Note that the backtracking module includes the whole logic part denoted as “backtracking module” at the bottom.

V. BACKTRACKING-BASED RECONFIGURABLE DECODER

In this section, we describe a backtracking-based reconfigurable decoder, which contains a generic interface named backtracking module that interacts with a reconfigurable QC-LDPC decoder. The proposed architecture can decode a family of structurally compatible codes with much lower error floor than conventional decoder.

A. Overall Architecture

The high-level architecture for the backtracking-based reconfigurable decoder is shown in Fig. 5. The depicted decoder supports CPM size of up to $P_M = 96$ and thus accommodates all the codes in the IEEE 802.11n and 802.16e standards. Note that it is basically a reconfigurable decoder interfaced to a backtracking module. The backtracking module reads the LLR from the input buffer and is responsible for configuring and controlling the reconfigurable decoder.

The backtracking algorithm is mapped to the backtracking module as follows. The decoder input LLRs are generated by the generator. In the original decoding, the initial LLRs from input buffer are bypassed directly to the decoder. In each decoding trial, one VN is flipped to the largest possible LLR magnitude with opposite sign bit. The 96-bit signals representing hard decision bits from the $u$ memory and the 96-bit selection signals from the controller are passed to the $y'$ generator to form

26: (Second Backtracking)
27: for $l = 1$ to $\gamma$
28: for each $v \in N(C^n_i)$, $0 \leq v < n$
29: $i \leftarrow i + 1$
30: $y' \leftarrow y; y'_0 \leftarrow -\chi(t_{0,1}^l)$, $\eta$
31: redecode using $y'$ as input LLR and $K_{\text{max}}^{(2)}$ as the maximum number of iterations, which outputs $u_i$ with unsatisfied CNs in $C_i$
32: if $|C_i| = 0$ then
33: stop and exit, output $u_i$ as the decoded codeword
34: else if $i = T_{\text{trial}}^{\text{max}}$ then
35: stop and exit, claim uncorrectable error
36: end
37: end
38: end
the modified input LLR $y'$, as shown in Fig. 6. The trial node selection unit (TNSU) takes the output of tag memory and helps the controller generates the selection signals of $y'$ generator.

The difference detector compares the hard decision bits generated in the original decoding and current decoding trial, as shown in line 18 of Algorithm 1. To compute the $\text{diff}$ signal, the difference detector firstly performs an exclusive-OR operation on the two versions of hard decision bits, and then executes a 96-bit OR operation on the resultant 96-bit vector. The result of the 96-bit OR operation is computed iteratively over $N_c$ cycles to form the $\text{diff}$ signal. The Hamming distance unit (HDU) computes the Hamming distance, as shown in line 19 of Algorithm 1. An implementation of HDU is shown in Fig. 7. The 12-bit signal $\text{dist}$ representing the Hamming distance is passed to the controller to determine the most likely hit-trials.

The controller generates the handshaking signals for the input buffer and the reconfigurable decoder. The $\text{ui}$ memory and tag memory are used to store the required information for backtracking, i.e., $N(C_i)$ and $\mu_k$ in Algorithm 1. The memory management unit (MMU) allocates and reclaim the $\text{ui}$ memory and tag memory pages as described in Section V-C.

B. Reconfigurable Decoder With Optimum State Storage

The reconfigurable decoder interfaced with the backtracking module is similar to the configurable or programmable decoder with layered decoding [26] as described in [27], [28], which simultaneously processes the extrinsic messages of one CPM in the parity-check matrix of a QC-LDPC code and uses a permutation network to reorder and deliver messages to corresponding serial-in serial-out processing units. Configuration memory is built into the decoder to store the code structure such as the offsets for all CPMs and the CN degrees for each block row. An optimized circular shift network [29] is used to realize the permutations. A $P_{M'}$-parallel reconfigurable decoder supports any QC-LDPC code with CPM size $m \leq P_M$.

The backtracking algorithm requires the decoder to identify its minimum set of unsatisfied CNs and to store the associated hard decision bits and candidate VNs, which are updated on the iteration basis. The iteration corresponding to the minimum set of unsatisfied CNs is referred to as the optimum iteration. The three-tuple of hard decision bits, candidate VNs, and the number of unsatisfied CNs in each iteration is referred to as a state. The state in the optimum iteration is named as the optimum state, which is initialized as the state in the first iteration and is updated by the state of the current iteration when the number of unsatisfied CNs in the current iteration is smaller than the current minimum number. Thus, at any given time, the state of the current iteration and that of the optimum iteration must be stored.

To compute the number of unsatisfied CNs, the $m \leq P_M$ syndrome checks generated during the message update of each block row is sent to a $P_{M'}$-input parallel adder, whose result is accumulated with a register storing the partial sum of syndrome checks. When a decoding iteration is finished, the register value is the number of unsatisfied CNs in the current iteration.

The hard decision bits are stored in the $\text{ui}$ memory, which is organized as a set of pages containing $N_c P_{M'}$-bit words. Each memory word corresponds to a group of VNs in the same block column of the parity-check matrix. In layered decoding, the syndrome checks in the current iteration are computed by the hard decision bits generated in the previous iteration. Therefore, three versions of the hard decision bits—corresponding to that of the current, previous and the optimum iteration—are stored in the $\text{ui}$ memory. In implementation, three index registers are used to keep track of these pages.

The candidate VNs information are stored in the the tag memory, which is organized as a set of pages containing $N_C P_{M'}$-bit words. The structure of each page is designed in accordance with the decoder, since the messages of one $m \times m$ CPM is processed simultaneously and there are $N_c$ block columns in the underlying parity-check matrix. Each bit in the tag memory corresponds to a VN in the Tanner graph. If the VN neighbors an unsatisfied CN, the bit content is one; otherwise the bit content is zero. This information can be viewed as meta information about a given VN, and hence we refer to the memory as tag memory. Two versions of the hard decision bits—corresponding to that of the current and the optimum iteration—are stored in the tag memory. In implementation, two index registers are used to keep track of these pages.

The tag information is generated by performing a circular shift operation on the syndrome bits generated at the end of each block row processing through existing QSN in the decoder. However, there is a subtle problem. A given VN neighbors multiple CNs and contributes in multiple syndrome calculations. Without proper protection scheme, it is very likely that a VN tagged as one is updated to zero later since it is connected with an satisfied CN. To solve this problem, an additional bit for each CPM is appended to each word of the configuration memory. If
the CPM is the first CPM in that block column, the bit is configured as zero, and the tag information is written directly to the tag memory; otherwise, the bit is configured as one, and the tag information is performed an OR operation with the existing value before being written to the tag memory.

**Example Illustrating Tag Memory Update:** Fig. 8(a) shows the configuration memory of a typical reconfigurable decoder configured to decode the (1944, 1458) code as described in Fig. 1. Each configuration memory word has three fields. As noted above, the most significant bit (MSB) denotes whether the VN has been tagged or not in a prior syndrome calculation, the five bit field denotes the block column index and the 7-bit field denotes the offsets that describe the CPM structure. Each word in the configuration memory corresponds to one of the 84 CPMs in the underlying parity-check matrix.

Suppose we are interested in updating the tag information for the 14th CPM, i.e., the CPM located in the 1st block row and 0-th block column. Note that $s_0, s_1, \ldots, s_8$ denote the syndrome bits corresponding to this block row. Given that the tag information for the VNs in the 0-th block column starts from the 0th CPM, the selection signals for the multiplexors array are set to 1. The syndrome register is shifted right by $81 - 4 = 77$ with the help of the QSN. Fig. 8(b) (which indicates the high level view of the code structure) shows that the $i$-th VN is connected to $(i + 77) \mod 81$-th CN. So, if the syndrome corresponding to $(i + 77) \mod 81$ is 1 (which indicates that it is not satisfied), the corresponding VN is tagged as 1.

**C. Memory Management Unit**

To implement first backtracking only, the $\mathbf{u}$ memory and tag memory require three pages and two pages respectively, as shown in Section V-B. However, two design challenges arise for the second backtracking, which requires the storage of $\gamma$ optimum states corresponding to the $\gamma$ most likely hit-trials during the first backtracking. First, the capacity of $\mathbf{u}$ memory and the tag memory are extended to $3 + 1 + \gamma = \gamma + 4$ pages and $2 + 1 + \gamma = \gamma + 3$ pages, respectively. There is a constant term 1 in the above two equations since the memory page storing the optimum state of the original decoding is still in use when generating optimum states for the second backtracking. Second, since the hit-trials are very likely to be replaced during the first backtracking, we design a memory management unit (MMU) to allocate memory pages as requested by the decoder and to reclaim the invalidated pages for memory efficiency.

As stated in Section V-B, the original decoding or each trial in the first backtracking requires three pages of $\mathbf{u}$ memory and two pages of tag memory. When the decoder is in the original decoding phase, the MMU will allocate three unused $\mathbf{u}$ memory pages (shown as uid0, uid1, and uid2 in Fig. 5) indexed as $\{0, 1, 2\}$ and two unused tag memory pages (shown as tag_id0 and tag_id1 in Fig. 5) indexed as $\{0, 1\}$ to decoder. After the original decoding or each trial is complete, the MMU will receive the two memory indices (shown as uid_opt and tag_id_opt in Fig. 5) corresponding to the $\mathbf{u}$ memory and tag memory of the optimum iteration. If the optimum state does not satisfy the $\gamma$ constraint in the original decoding and is not among the $\gamma$ states with the smallest Hamming distance in the first backtracking, the pages used in the previous decoding are resent to the decoder by the MMU for the next trial. Otherwise two new pages (one page for the $\mathbf{u}$ memory and tag memory each) will be allocated for the next trial along with the pages that do not represent the optimum state in the current trial.

If no greater than $\gamma$ valid optimum states are found in the first backtracking, the indices of newly allocated pages are generated linearly. For example, for the $\mathbf{u}$ memory, after the original decoding, the indices of newly generated pages are in the order of $\{3, 4, \ldots, \gamma + 3\}$. If more than $\gamma$ valid optimum states are found, the page indices corresponding to the current maximum
Hamming distance will be fetched from the controller and invalidated by the MMU to form the new allocated page indices. Fig. 9(a) and Fig. 9(b) illustrate how the memory is managed for this case.

D. Trial Node Selection Unit

For each CPM in the parity-check matrix, there may be zero or multiple (no less than one) candidate VNs. In other words, each word of the tag memory consists of all 0-bits or multiple 1-bit and remaining 0-bits. Before each trial, a candidate VN is located by searching for the 1-bit in the tag memory. Since the number of candidate VNs \(N(C)\) is much smaller than the cardinality of the set of all VNs \(N\), a linear search method requiring to \(O(N)\) clock cycles test all the bits in the tag memory is extremely inefficient. We developed a parallel search scheme instead that reduces the time complexity from \(O(N)\) to \(O([N(C)])\).

Fig. 10 shows a trial node selection unit (TNSU) which supports CPM size of up to \(P_M = 96\). The controller generates read and write address for the tag memory. The tag information are fetched from tag memory and passed to the TNSU as the input signals (tag[95:0]). The TNSU has two functionalities. First, it identifies whether the corresponding 96 VNs of the tag memory word contains candidate VNs. If the signal \(\text{all_zero}\) is low, the accessed memory word contains 0-bits only, which means no candidate VNs are found. Thus, the controller will generate the read address for the next row. Second, the TNSU select convert the tag information containing multiple 1-bits to a modified tag information (tag[95:0] in Fig. 10) with only one 1-bit, i.e., only one candidate VN is selected. The modified tag information will be performed an exclusive-or operation with the associated tag memory word to update its content, thus the previous accessed candidate VN will not be tested for a new trial. For example, suppose the contents of a 8-bit tag memory word \(P_M = 8\) are tag[7:0] = 0010_1010, then the output vector should be tag[7:0] = 0010_0000. Update the modified tag information is generated, the tag memory word is updated by tag[7:0] = 0000_1010.

To realize these functions with reduced pipeline stages, we employ a divide-and-conquer strategy, as shown in Fig. 10. The 8-bit or 12-bit one detector in Fig. 10 can be realized with 8-bit or 12-bit OR-gate. The priority encoder is designed with simple logic gates. For example, for a 4-input priority encoder with input \(x_3x_2x_1x_0\), output \(y_3 = x_3, y_2 = x_3x_2, y_1 = x_3x_2x_1, \) and \(y_0 = x_3x_2x_1x_0\).

VI. IMPLEMENTATION RESULTS

Based on the proposed architecture, we implemented two backtracking modules, one that realizes only the first backtracking algorithm and the other that realizes the double backtracking. The backtracking module is interfaced to a reconfigurable QC-LDPC decoder that can decode any QC-LDPC code that has up to 24 block columns and up to 96 CPMs with \(m \leq P_M = 96\). The input LLR, the extrinsic message, and the updated LLR are of 6-bit, 6-bit, and 8-bit, respectively. The designs are synthesized targeting a TSMC 0.18 μm CMOS ASIC cell library using the Synopsys design tools and flow. The realized reconfigurable decoder and the two backtracking modules achieve an operating frequency of 200 MHz and 400 MHz, respectively. Thus, there is no penalty in clock rate by integrating the backtracking module with the reconfigurable decoder. The critical path of the reconfigurable decoder exists in the QSN. The critical paths of both two backtracking modules exist in the controller component.

The backtracking module realizing the first backtracking only and the backtracking module realizing double backtracking algorithm require 7.5 K and 10.3 K of equivalent NAND2 gates respectively. If only the first backtracking is implemented, only three pages of \(\Pi\) memory and two pages of tag memory are required. Since each page contains twenty-four 96-bit words, the memory requirement to implement only the first backtracking is \(5 \times 24 \times 96 = 11520\) bits. As shown in Section V-C, an implementation of double backtracking algorithm requires \(\gamma + 4\) pages of \(\Pi\) memory and \(\gamma + 3\) pages of tag memory. Our implementation uses \(\gamma = 3\), thus the memory requirement for double backtracking is \((7 + 6) \times 24 \times 96 = 29952\) bits.
TABLE IV

<table>
<thead>
<tr>
<th>Decoder</th>
<th>[30]</th>
<th>[31]</th>
<th>[28]</th>
<th>this work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip Area (mm$^2$)</td>
<td>8.29 (Total)</td>
<td>6.25 (Total)</td>
<td>10.12 (Total)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>420K (Total)</td>
<td>380K (Logic)</td>
<td>198K (Logic)</td>
<td>185K (Logic)</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>590K (Memory)</td>
<td>265K (Memory)</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory Bits</td>
<td>N/A</td>
<td>89,856</td>
<td>82,752</td>
<td>54,492</td>
</tr>
<tr>
<td>Shift Network Based</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallelism</td>
<td>N/A</td>
<td>92</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>Cycles per Iter.</td>
<td>384</td>
<td>N/A</td>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td>Mode</td>
<td>19</td>
<td>114</td>
<td>≥ 126</td>
<td>≥ 126</td>
</tr>
<tr>
<td>Technology</td>
<td>130nm, 1.2V</td>
<td>90nm, 1.0V</td>
<td>180nm, 1.8V</td>
<td>180nm, 1.8V</td>
</tr>
<tr>
<td>$f_{CLK}$ (MHz)</td>
<td>83.3</td>
<td>150</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>Low Error Floor</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Fig. 11. Block error performance for 802.16e standard code with code length 2304 when backtracking is applied.

We integrate the two backtracking modules with the reconfigurable decoder with optimum state storage as shown in Section V-B. Table IV shows the post-synthesis results of the conventional decoder realizing the original decoding, the decoder realizing the first backtracking only, and the decoder realizing double backtracking. We also present the implementation results of three decoders that were recently published for comparison. The difference in terms of memory usage and logic gates is mainly caused by the choices of quantization and parallelism. The results clearly demonstrate that the area overhead (measured in terms of gates) of backtracking module is not very significant—it is about 5% if only one backtracking is implemented and about 7% if the second backtracking is also implemented. The additional memory requirements for the backtracking implementation is also not very significant—about 13% and 46% for the first and double backtracking respectively.

Furthermore, it is important to note that the backtracking module does not change the time (measured in clock cycles) for a single iteration. This means, there is no throughput penalty if the backtracking is not invoked. It requires exactly $N_r + 2N_{CPM}$ clock cycles to complete one iteration. Thus, if the average number of iterations is $\bar{K}$ and if takes $N_r$ clock cycles to load the intrinsic messages, the throughput would be

$$\frac{mN_r f_{CLK}}{(\bar{K} \times (N_r + 2N_{CPM})) + N_c}.$$  

For example, for the (2304, 1152) code in the mobile WiMAX standard, the throughput would be 344 Mbps at five iterations. We evaluate the code performance of the 802.16e standard code with code length 2304 and 802.11n standard code with code length 1944, and show their block error performance in Figs. 11 and 12, respectively. For each simulation point, we collect 500 block errors. The simulation parameters are $(K = 20, i \in \{0, 1, 2\}, \tau = 10, \gamma = 3)$. No latency constraints are applied for the decoder. Based on the simulation, the first backtracking algorithm lowers the error floor by 0.5 to 1.5 order of magnitudes, while the second backtracking algorithm further lowers the error floor by another 0.5 order of magnitudes. For some codes, e.g., rate-5/6 code in the 802.16e standard, the improvement of double backtracking algorithm is within half the order of magnitude, however the curve also approach the limit
of undetected block error rate. Thus, for errors resulted by trapping sets, the backtracking algorithm is exceptionally useful.

VII. CONCLUSIONS

We described the design and implementation of a hardware-based backtracking scheme to detect and break trapping sets at runtime to improve the error floor performance of QC-LDPC codes at very low bit error rates. The main advantage of our approach is that it is not customized to one specific code. So the same hardware decoder can be used to lower the error floor of many codes instead of building a customized decoder for each code. This is accomplished by creating a self-contained backtracking module that can be plugged to any generic reconfigurable iterative decoder for QC-LDPC codes. Because of the optimized techniques to reduce the complexity and latency of the backtracking algorithm, we demonstrated that the additional area and latency overhead is small, especially for the first backtracking. Also, there is no impact on the throughput of the decoder if the backtracking is not invoked, since the backtracking module is implemented at a much higher clock rate than the decoder.

REFERENCES


Xiaoqiang Chen received the B.S. and M.S. degrees in electrical engineering from Zhejiang University, Hangzhou, China, in 2005 and 2007, respectively, and the Ph.D. degree in electrical and computer engineering from the University of California, Davis, in 2011.

He is currently with Microsoft, Redmond, WA as a software engineer. His research interests include compiler architecture and parallel computing, embedded systems, cloud computing, reconfigurable computing, FPGA, and error control coding.

Jingyu Kang received the B.S. and M.S. degrees in electronic engineering from Tsinghua University, Beijing, China, in 2002 and 2007, respectively, and the Ph.D. degree in electrical and computer engineering from the University of California, Davis, in 2009.

He is currently with Augusta Technology USA, Inc., Santa Clara, CA, as a System Engineer. His research interests include error control coding and signal processing for data storage and communication systems.
Shu Lin (S’62–M’65–SM’78–F’80–LF’00) received the B.S.E.E. degree from the National Taiwan University, Taipei, in 1959, and the M.S. and Ph.D. degrees in electrical engineering from Rice University, Houston, TX, in 1964 and 1965, respectively.

In 1965, he joined the Faculty of the University of Hawaii, Honolulu, as an Assistant Professor of Electrical Engineering. He became an Associate Professor in 1969 and a Professor in 1973. In 1986, he joined Texas A&M University, College Station, as the Irma Runyon Chair Professor of Electrical Engineering. In 1987, he returned to the University of Hawaii. From 1978 to 1979, he was a Visiting Scientist at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, where he worked on error control protocols for data communication systems. He spent the academic year of 1996–1997 as a Visiting Professor at the Technical University of Munich, Munich, Germany. He retired from University of Hawaii in 1999 and he is currently an Adjunct Professor at University of California, Davis. He has published numerous technical papers in IEEE Transactions and other refereed journals. He is the author of An Introduction to Error-Correcting Codes (Englewood Cliff, NJ: Prentice-Hall, 1970). He also coauthored (with D. J. Costello) Error Control Coding: Fundamentals and Applications (Upper Saddle River, NJ: Prentice-Hall, 1st ed., 1982, 2nd ed., 2004), and (with T. Kasami, T. Fujiwara, and M. Fossorier) Trellises and Trellis-Based Decoding Algorithms, (Boston, MA: Kluwer, 1998). His current research areas include algebraic coding theory, coded modulation, error control systems, and satellite communications. He has served as the principal investigator on 36 research grants.

Dr. Lin is a Member of the IEEE Information Theory Society and the Communication Society. He served as the Associate Editor for Algebraic Coding Theory for the IEEE TRANSACTIONS ON INFORMATION THEORY from 1976 to 1978, the Program Co-Chair of the IEEE International Symposium of Information Theory held in Kobe, Japan, in June 1988, and a Co-Chair of the IEEE Information Theory Workshop held in Chengdu, China, October 2006. He was the President of the IEEE Information Theory Society in 1991. In 1996, he was a recipient of the Alexander von Humboldt Research Prize for U.S. Senior Scientists, a recipient of the IEEE Third-Millennium Medal, 2000, and a recipient of the IEEE Communications Society 2007 Stephen O. Rice Prize in the Field of Communication Theory.

Venkatesh Akella received the Ph.D. degree in computer science from the University of Utah, Salt Lake City.

He is a Professor of Electrical and Computer Engineering at the University of California, Davis. His current research encompasses computer architecture and embedded systems.

He is a member of ACM.