Spatially-coupled low-density parity check codes: Zigzag-window decoding and code-family design considerations

Shadi Abu-Surra, Eran Pisek, and Rakesh Taori
Samsung Research America - Dallas
Samsung Electronics
Richardson, Texas, USA
Email: {shadi.as, e.pisek, rakesh.taori}@samsung.com

Abstract—Spatially-coupled low-density parity-check (SC-LDPC) codes are new capacity-achieving codes, which combine the advantages of both convolutional-turbo codes and LDPC block codes. SC-LDPC codes can provide scalable code-length, high granularity, convolutional-gain, low-complexity decoding, and high parallel processing. To overcome the limitations of the current SC-LDPC decoders which have essentially hindered the widespread adoption of SC-LDPC, we propose a new decoder which we call the Zigzag Decoder, that essentially leverages the convolutional-gain of the sliding window decoder. We show that the zigzag-window decoder reduces the average number of iterations and improves the performance relative to the conventional sliding-window decoder. Furthermore, for facilitating the use of SC-LDPC codes in practice, we have explored how to generate SC-LDPC code families and compare the performance and complexity between shortening-based and puncturing-based SC-LDPC code families supporting different code rates.

Keywords—LDPC; spatially-coupled, iterative decoder; sliding window; zigzag window; shortening; puncturing.

I. INTRODUCTION

Low Density Parity Check (LDPC) codes [1] have received a great deal of attention in recent years. This is due to their ability to achieve performance close to the Shannon limit [2], the ability to design codes which facilitate high parallelization in hardware, and their support of high data rates. The most commonly deployed form of the LDPC codes are the block LDPC codes. However, in highly dynamic wireless communication systems, where the channel conditions and the data allocation per user are continuously changing, block LDPC codes offer rather limited flexibility.

Using block LDPC codes requires allocating data in multiples of the code’s block-length to avoid unnecessary padding, which reduces the link efficiency. Amongst the wireless standards that have adopted LDPC as a part of the specification, the following three approaches can be observed to handle the granularity limitation of block LDPC codes: 1) Use codes with one very short block-length (e.g., IEEE 802.11ad [3]): the smaller the block length the finer the granularity of the code. However, block LDPC codes with short block lengths are lacking in performance [4], which also reduces the link efficiency. 2) Use block LDPC codes with multiple block lengths (e.g., IEEE 802.11n [5]): This approach mitigates the performance degradation at the expense of implementing a more complex decoder due to the requirement to support multiple codes. 3) Use turbo codes (e.g., 3GPP [6]). The convolutional structure of turbo codes can provide a scalable code-length with high granularity without increasing the decoder’s complexity. However, turbo codes do not provide enough parallel processing capability, which in turn limits their capability to achieve multiple Giga bits per second throughput.

Spatially-coupled low-density parity-check (SC-LDPC) codes [7]-[11] are new capacity-achieving codes, which combine the advantages of both turbo codes and LDPC block codes. SC-LDPC codes form a special class of LDPC codes which have relatively small syndrome former memory (see, Section II). Hence, SC-LDPC codes have parity check matrices with convolutional structure [7]. This structure allows for scalable code-length with high granularity compared to the other block LDPC codes. In addition, SC-LDPC codes inherit the high parallel processing capabilities of LDPC codes, and are therefore capable of supporting multiple Giga bits per second throughput.

Although, it is possible to decode terminated SC-LDPC codes [8] as any other block LDPC code, it diminishes the advantages of the flexible length and the fine granularity of the SC-LDPC codes. In addition, the latency and hardware complexity of this approach becomes impractical for SC-LDPC codes with very large block lengths. Due to the convolutional structure of the SC-LDPC codes parity check matrix, this decoder also requires large number of iterations before it converges. Alternatively, SC-LDPC codes can be decoded using a sliding-window decoder [7] [8]. This approach seems to mitigate most of the full block decoder problems. However, there are still several critical performance issues to be resolved:

1) The current SC-LDPC decoder average number of iterations is high (e.g., running a sliding-window decoder on the code in Section II with window size 15 and fixed 10 iteration per window execute 150 iterations per Variable node (VN)). It is required to reduce to the average number of iterations per VN in a sliding-window decoder to increase the throughput. It is also necessary to use a simple early stopping rule to terminate decoding of the current window and start decoding the next window. An early
stopping rule, which estimates the bit error rate (BER) within the decoding window based on a soft-bit indicator for the VN, was proposed in [11]. However, the evaluation of this rule requires a lookup table and the additions of $N_r$ real-numbers, where $N_r$ is the number of target variable nodes (VNs) in the decoding window. Instead, in this paper we propose and evaluate a simple stopping rule based on a partial syndrome computation (see, Section III).

2) While the full block decoder utilizes both termination ends of the SC-LDPC code to improve the decoding threshold [9], the sliding-window decoder utilizes only the start termination end. Thus, the full block decoder outperforms the sliding-window decoder. To further close the gap between the two decoders, we introduce the zigzag-window decoder in Section IV. We also show that the zigzag-window decoder requires lesser number of iterations per VN on average than the sliding-window decoder.

3) To adapt to the dynamic wireless channel, the coding scheme of a wireless system has to support multiple code rates. In Section V we address the problem of designing an SC-LDPC code family, which supports multiple code rates. In particular, we compare two approaches in term of BER performance and decoding complexity. The first approach uses code puncturing, and the second uses code shortening. The results show that the SC-LDPC code family based on shortening has a superior error performance and converges faster than that based on puncturing.

The remainder of the paper is arranged as follows: In Section II, we give a brief review of SC-LDPC codes, explain terminology used in the paper, and present an example SC-LDPC code to be used in following sections. In Section III, we review the sliding-window decoder, and propose a simple early stopping rule for window decoding. In Section IV, we propose the zigzag-window decoder, and present simulation results to demonstrate the performance of this new decoder. In Section V, we compare shortening versus puncturing approach to design SC-LDPC code families supporting multiple rates.

II. SC-LDPC CODES: REVIEW AND EXAMPLE

In this section we shortly introduce SC-LDPC codes (for more details refer to [7] and [8]), and then we describe an example SC-LDPC code, which is used to generate the simulation results in the following sections.

![Example of a protograph-based SC-LDPC code structure](image)

Fig. 1: Example of a protograph-based SC-LDPC code structure.

![Example of a (3, 6, 720) protograph-based SC-LDPC code with lifting factor Z = 112.](image)

Protograph-based SC-LDPC codes starts with a protograph, which is a Tanner graph [12] with relatively small number of VNs and check nodes (CNs). Example of a regular (3, 6) LDPC code protograph is shown in Fig. 1a. A (3, 6, L) SC-LDPC code ensemble is created using L copies of the protograph and coupling them into a chain as shown in Fig. 1b. Each protograph in the chain is coupled to at most m protograph copies to its left and at most m protograph copies to its right, where m is called the syndrome former memory, or simply the code’s memory. In contrast to protograph-based block LDPC codes with multiple steps lifting [13], SC-LDPC codes have small code’s memory (i.e., $m < L$), which dictates the parity check matrix of the SC-LDPC codes its convolutional structure. Note that the CNs in the first and last m protograph copies have lower degree than their corresponding CNS in the other protograph copies. This implies a stronger CNs at the start and the end of the SC-LDPC coupled chain, and this improves the iterative decoding threshold of SC-LDPC codes [9] at the cost of a small reduction in the code rate. The first m protograph copies in the coupled chain are called the start-termination, and the last m ones are called the end-termination, these are shown in shaded color in Fig 1b.

As an example, we designed a (3, 6, 720) SC-LDPC code using the protograph shown in Fig. 1a. First the code ensemble is constructed by coupling a chain of 720 protograph copy as in Fig. 1b, then the code is constructed by lifting the coupled chain with a lifting factor $Z = 112$. The parity check matrix for the designed code is shown in Fig. 2. A number x in this matrix indicates the $Z \times Z$ identity matrix cyclically shifted x positions to the right. The blank entries in the matrix are the $Z \times Z$ all-zeroes matrices. Each two columns of the parity check matrix are labeled by a number, which indicates a protograph section to which these 2Z VNs belong. In each protograph section, the first Z VNs are information bits and the second Z VNs are parity bits. The designed code is a time-variant quasi-cyclic (QC) SC-LDPC code with periodicity equals three [10]. The permutations matrices in this parity check matrix were chosen to maximize the girth of the first nine protograph sections. Because the periodicity of this code is three, the girth of the full code is the same as the girth of the first nine protograph sections.

This SC-LDPC code will be used in generating the performance results in the next two sections.
III. SLIDING-WINDOW DECODER AND EARLY STOPPING RULE

In this section we review the sliding window decoder, and then introduce the partial-syndrome early stopping rule.

The sliding-window decoder [7] is depicted in Fig. 3. In this decoder, a decoding window of size $W$ moves along the protograph sections of the code. The position, $p_w$, of the decoding window refers to the first protograph section in the window. The decoder runs as follows:

- For all positions $p_i = p_{i-1} + s$, where $s$ is the window step size, $i = 1, \ldots, L$, and $p_1 = 1$:
  1. For a number of $I_w$ iterations:
     a. Update all VNs and CNs in the decoding window according to an iterative LDPC code decoding algorithm.
    2. Make a decision on the value of the VNs in the first $s$ protograph sections. These VNs are called target VNs.

Within the decoding window, one can use any of the iterative decoding algorithms used for LDPC codes such as sum-product [14], min-sum [15], dual-quantization domain [16], etc. Also, note that updating the CNs within the decoding window requires access to extrinsic information for VNs outside the decoding window. In particular, the VNs in the $m$ protograph sections just before the decoding window. These VNs send their extrinsic information based on the last update from the previous window. The number of iterations should be chosen to achieve targeted error probability within the window. This can be done using density evolution techniques [7].

![Fig. 3: Sliding window decoder with decoding window of size $W = 10$ and partial syndrome-check window of size $W_s = 5$.](image)

The use of a pre-determined number of iterations is not very efficient, that as it has to be set for the worst channel condition. Instead, in [11] a real-time early stopping rule was proposed, which is based on estimating the BER using the log-likelihood-ratios (LLRs) of the target VNs within a decoding window. When the estimated BER is less than a threshold BER or the maximum number of iterations reached, the decoding window moves forward. The BER is estimated by averaging the soft bit error indicators [17] of the target VNs, where the soft bit error indicator for a VN having LLR $I$ is $1/(1+\exp(I/|I|))$. However, the evaluation of the soft bit error indicator requires a lookup table, and computing the average requires $N_t$ (in our example, $N_t = 2Z_s$) real-valued additions after each iteration.

We propose to use a simple stopping rule based on checking the syndrome of only few of the CNs in the decoding window. Let us define a partial syndrome-check window of size $W_t$ as the first $W_t$ protograph sections in the decoding window. This is illustrated using the green rectangle in Fig. 3. The updated sliding window decoder is as follows:

- For all window positions $p_i = p_{i-1} + s$, where $s$ is the window step size, $i = 1, \ldots, L$, and $p_1 = 1$:
  1. For a maximum number of $I_w$ iterations:
     a. Update all VNs and CNs in the decoding window according to an iterative LDPC code decoding algorithm.
     b. Compute the syndrome for the CNs in the partial syndrome-check window. If all CNs are satisfied or maximum number of iterations reached, go to the next step. Otherwise, continue the next iteration.
  2. Make a decision on the value of the target VNs.

Fig. 4 shows the BER and frame error rate (FER) performance of this stopping rule using the sliding window decoder with $W = 15, s = 1$, and $I_w = 90$ with different partial syndrome-check window size $W_s = 5, 4, 3$. The results also compared to that using the full block decoder with 1000 iteration maximum. It is noticeable that the FER performance of the sliding window decoder starts to degrade for $W_t < 5$. The results for $W_t > 5$ seem to have similar performance to $W_t = 5$. Note, $W_t$ for this code should be greater than or equal 3, as the CNs in this window are directly connected to the target VNs.

![Fig. 4: FER (dashed) and BER (solid) performance of the proposed early stopping rule with sliding window decoder compared to block decoder with full syndrome check.](image)

IV. ZIGZAG-WINDOW DECODER

In the sliding-window decoder the window moves in one direction from the start-termination toward the end-termination. This allows the information to flow between decoding windows in one direction only, which results in some loss in performance compared to the full block decoder. Moreover, simulations using sliding-window decoder showed that when a decoding window fails to converge (i.e., the maximum number of iterations is reached and the CNs in the partial syndrome-check window are not satisfied), few successive decoding window will also diverge, but after that it will converge back to the correct decision. This is illustrated in Fig. 5a, where the circles represent the state of the decoding
windows as it moves along the protograph sections of the code. A green circle indicates that the decoding window converged to the correct decision, and the red circle indicates that the decoding window didn’t converge. Let us define a divergence as the set of consecutive protograph sections, which fail to converge, and define the divergence size as the number of protograph sections in it. Example, the divergence in Fig. 5a has size 4.

In the zigzag-window decoder, at the end of a divergence the decoding window reverses its direction trying to correct the divergence by pushing the information from the correct state at the end of the divergence back to the divergence. Fig. 5b illustrates the direction of the zigzag-window decoder. The decoding window has two directions: Forward direction, in which the decoding window moves toward the end-termination (right), and a backward direction, in which the decoding window moves toward the start-termination (left). The zigzag-window decoder process is as follows:

1) Set window position \( p = 1 \), window_direction = Forward, and previous_window_converged = True.

2) For a maximum number of \( I_w \) iterations:
   - Update all VNs and CNs in the decoding window according to an iterative LDPC code decoding algorithm.
   - Compute the syndrome for the CNs in the partial syndrome-check window.
     - If all CNs are satisfied, set window_converged = True, then go to Step 3.
     - Else if the maximum number of iterations reached, set window_converged = False, then go to Step 3.
   - Check the decoding window direction
     - If window_direction = Forward,
       - If window_converged = False, and previous_window_converged = True,
         - window_start_position = \( p \).
       - Else if window_converged = True,
         - previous_window_converged = False, and the number of Z-rounds is less than the maximum number of Z-rounds,\n           - window_start_position = \( p \).
       - Else
         - Make a decision on the value of the target VNs.
         - previous_window_converged = window_converged.
         - \( p = p + s \).
     - Else if window_direction = Backward,
       - If \( p > \) divergence_start_position
         - \( p = p - s \).
       - Else
         - \( p = p + s \).
         - window_direction = Forward.
         - previous_window_converged = True.

3) If \( p = L \), then Go to Step 2. Else, Stop.

In the proposed zigzag-window decoder, a Z-round is defined as the combination of a forward, a backward, and then a forward window direction on a divergence. If some of the protograph sections in the divergence have not converged while in the second forward window direction of the Z-round, then a second Z-round can be nested within the first Z-round. An illustration of this concept is shown in Fig. 6. Similarly multiple Z-rounds can be performed. However, to prevent unlimited bouncing within a set of VNs at the low SNR region a maximum number of Z-rounds should be specified.

Fig. 5: Illustration for the deriction of the decoding window in sliding-window and zigzag-window decoders.

Fig. 6: Example of one Z-round and two Z-rounds of the zigzag-window decoder.

Fig. 7 and Fig. 8 show the performance of the zigzag-window decoder for different maximum number of Z-rounds, \( Z_R \), and different maximum number of iterations, \( I_w \). The simulation is done using BPSK modulation over the AWGN channel, and the decoder used within the decoding window is the scaled min-sum with flooding scheduling [15]. The zigzag-window decoder with \( Z_R = 0 \) is exactly the sliding-window decoder. Note, the zigzag-window decoder allows the information to flow from the end of the divergence backward. And so, the BER/FER performance of the zigzag-window decoder is better than the sliding window decoder. Also, this decoder allows using a small value for the maximum number of iterations compared to the sliding-window decoder without degrading the BER performance. Note that the zigzag-window decoder with \( Z_R = 1 \) and \( I_w = 5 \) is 0.25 dB better than the sliding-window decoder with \( I_w = 90 \). Moreover, Fig. 8 shows that the zigzag-window decoder with \( Z_R = 1 \) and \( I_w = 5 \) requires less number of iteration per VN on average than the sliding-window decoder with \( I_w = 90 \). Finally, Fig. 7 and Fig. 8 show that increasing the number of Z-rounds beyond \( Z_R = 1 \) has a diminishing return.

The results in Fig. 7 and Fig. 8 assume no limitations on the divergence size. However, a realizable implementation of the zigzag-window decoder has to specify the maximum divergence size. Note that the memory needed to implement the zigzag-window decoder increases when increasing the supported maximum divergence size. The maximum divergence size is called the Z-window size, \( Z_W \). (The needed implementation memory is proportional to \( W + Z_W + m \)). Fig. 9 shows that the probability of correcting a divergence decreases...
with its size. Fig. 10 shows the performance of the zigzag-window decoder for different \( Z_W \) sizes (\( Z_W = 5, 10, 15, 20, \) and infinity). The results show that increasing the \( Z_W \) size improves the performance, but it requires more implementation memory. Future study is needed to optimize the choice of \( Z_W \) and \( I_w \). For example, we expect that it is possible to reduce \( Z_W \) and increase \( I_w \) to maintain the performance.

![Fig. 7: FER performance of the zigzag-window decoder for different maximum number of Z-rounds (\( Z_R = 0, 1, 2, \) and 3) and different maximum number of iterations in a decoding window (\( I_w = 5, 10, \) and 90).](image)

![Fig. 8: Number of iterations performance of the zigzag-window decoder for different maximum number of Z-rounds (\( Z_R = 0, 1, 2, \) and 3) and different maximum number of iterations in a decoding window (\( I_w = 5, 10, \) and 90).](image)

![Fig. 9: Probability of correcting a divergence of size \( x \) protograph sections with one Z-round of the zigzag-window decoder.](image)

![Fig. 10: FER (dashed) and BER (solid) performance of the zigzag-window decoder with a finite \( Z_W \) size.](image)

![Fig. 11: Average number of iterations performance of the zigzag-window decoder applying the early codeword-failure detection rule with different \( Z_W \) size.](image)

In a zigzag-window decoder with a finite \( Z_W \) size, \( Z_W \), observing a divergence of size greater than \( Z_W \) can be used as an early indication of block decoding failure. In this case, the decoder may stop decoding the rest of the SC-LDPC codeword, then the receiver can request for retransmission. This early codeword-failure detection rule reduces the average number of iterations especially at low SNR (See, Fig. 11), and this reduces the decoder’s power consumption. Also, the early codeword-failure detection rule allows the receiver to quickly request retransmission via automatic repeat request (ARQ), or hybrid ARQ (HARQ), which reduces the latency of the feedback error correction scheme.

V. CODE-FAMILY DESIGN: SHORTENING VERSUS PUNCTURING

In this section, we address the issue of designing a family of SC-LDPC codes to support multiple code rates. We consider the following two candidate approaches: 1) design an SC-LDPC ensemble with low code rate, and then puncture it to obtain codes with higher code rates. This approach is similar to the one used with convolutional codes, except here puncturing is done on the protograph sections level, rather than on the bit level (i.e., if a VN type in the SC-LDPC code is punctured, then all Z VNs of that type in the lifted SC-LDPC code will be punctured). 2) Design an SC-LDPC code...
ensemble with high code rate, and then shorten it into codes with lower code rates. This approach is similar to one used in designing the AR4JA LDPC code family in [17].

In this paper, we compare the two code family design approaches by designing and simulating two example-families of SC-LDPC codes (one per approach). And so the presented results are meant to give a sense of how these two approaches compare in terms of BER performance and convergence speed, and how it is translated into computational complexity. More general results can be optioned by comparing families of SC-LDPC code ensembles, which can be done in a future study. Also in this study, we only consider codes with regular degree distribution (regular except for the terminations). According to [7] regular SC-LDPC codes have good decoding threshold, which is comparable to optimized irregular designs.

In the first code family, the (3, 6, 720) SC-LDPC code designed in Section II has the lowest rate (~rate-1/2). This code is punctured to an “approximately” rate-2/3, rate-3/4, or rate-5/6 using the puncturing pattern [1, 0, 1, 1], [1, 0, 1, 1, 1, 0], or [1, 0, 1, 0, 1, 1, 0, 1, 0], respectively. An example on puncturing the (3, 6, 720) SC-LDPC code to a rate-3/4 code is given in Fig. 12, where the red circles indicate punctured VNs, and the green circles indicate transmitted VNs.

In the second code family, we designed a (3, 18, L) SC-LDPC code ensemble based on the protograph in Fig. 13. Then, we lifted this code using Z =112 cyclic permutation matrices into a time-variant quasi-cyclic SC-LDPC code with periodicity equal three. The parity check matrix for the designed code is given in Fig. 14. The permutation matrices were chosen in a hierarchal way: first, the ones lifting \( i_1 \) and \( p_1 \) (yellow columns in Fig. 14) were chosen to maximize the girth of the first nine protograph sections of the code, where all other permutations were set to the \( Z\times Z \) all-zeroes matrices. Then the permutation matrices lifting \( i_2 \) (green) were added to maximize the girth, and so on lifting \( i_3 \) (blue), \( i_4 \) (orange), and finally \( i_5 \) (red). To shorten this code to a rate-4/5 code, all columns in the red color are deleted (i.e., set all VNs in \( i_5 \) to ‘0’, then delete them before transmission). To shorten it to a rate-3/4 code, all columns in red and orange are deleted, and so on, the code can be further shortened to obtain rate-2/3 and rate-1/2 codes.

In the punctured code family, the number of information bits in a codeword is fixed and equal to 80416 bits. In the shortened code family, we adjusted the number of protograph copies \( L \) for each code rate to keep the number of information bit in a codeword 80416. Consequently, \( L = 720, 360, 240, \) and 144 for rate-1/2, 2/3, 3/4, and 5/6, respectively. Also, trying to keep a fair comparison, we varied the size of the decoding window such that the number of VNs in the decoding window is 30. That is \( W = 15 \) for all codes in the punctured code family, and \( W = 15, 10, 7, \) and 5 for the rate-1/2, 2/3, 3/4, and 5/6, respectively, of the shortened code family (For rate-3/4, \( W \) should be 7.5, but it was approximated to 7).

![Fig. 13: A rate-5/6 (3, 18) LDPC code protograph, which can be shortened to rate-4/5, 3/4, 2/3, or 1/2.](image)

The FER and average number of iterations results in Fig. 15 and Fig. 16 show that the rate-1/2 codes in the punctured and the shortened code families have similar performance. The rate-2/3 code in the shortened code family performs slightly better than that in the punctured code family. The rate-3/4 code in the shortened code family outperforms that in the punctured code family by 0.6 dB in term of FER performance, and provides a reduction of more than 12 iterations per VN on average at any given SNR point. The rate-5/6 code in the shortened code family outperforms that in the punctured code family by 0.9 dB in term of FER performance, and provides a reduction of more than 28 iterations per VN on average at any given SNR point. In general, the designed shortened code family performs better than the punctured code family.

Although the codes in the shortened code family require less average number of iterations per VN, this may not necessarily result in lower computation complexity. Note that the codes in this family have CNs of larger degree than that in the punctured code family, and this should be taken in consideration in the computation complexity comparison. For computation complexity, we assume the use of the scaled Min-Sum decoder, in which a CN with degree \( d_c \) (per iteration) requires 2 addition operations for scaling, \( 3 (d_c-2) \) compare operations for finding the minimum, and \( 2 d_c - 1 \) XOR operation for computing the sign. Moreover, a VN with degree \( d_v \) requires \( 2 d_v \) -addition operations per iteration. Let us assume that 1 addition operation is equivalent to one compare operation, or 8 XOR operations, then we can define the shortened/punctured complexity ratio as the number of XOR operations per decoding a VN in the shortened code family divided by the number of XOR operations per decoding a VN in the punctured code family. Moreover, the comparison can...
be made on the same SNR point or at the same FER performance. The complexity ratio is summarized in Table I.

![Graph](image1.png)

**Fig. 15:** FER performance comparison between the punctured code family and the shortened code family.

![Graph](image2.png)

**Fig. 16:** Number of iterations performance comparison between the punctured code family and the shortened code family.

### Table I. Summary of the shortened/punctured complexity ratios.

<table>
<thead>
<tr>
<th>Rate</th>
<th>Comparison Criterion</th>
<th>Avg. Num. of iterations (shortened, punctured)</th>
<th>Complexity ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>@ FER 1e-2</td>
<td>36, 40</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>@ SNR 1.625 dB</td>
<td>30, 32</td>
<td>0.94</td>
</tr>
<tr>
<td>2/3</td>
<td>@ FER 1e-2</td>
<td>39, 38</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>@ SNR 2.375 dB</td>
<td>33, 35</td>
<td>0.74</td>
</tr>
<tr>
<td>3/4</td>
<td>@ FER 1e-2</td>
<td>34, 27</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>@ SNR 3.375 dB</td>
<td>17, 28</td>
<td>0.44</td>
</tr>
<tr>
<td>5/6</td>
<td>@ FER 1e-2</td>
<td>32, 49</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td>@ SNR 4.25 dB</td>
<td>22, 49</td>
<td>0.3</td>
</tr>
</tbody>
</table>

### VI. CONCLUSION

In this paper, we proposed an early stopping rule for window decoding of SC-LDPC codes. The rule is based on partial syndrome-check, which performs only XOR operations. We also propose a zigzag-window decoder, which allows for pushing the information backward from the end of a divergence. The proposed zigzag-window decoder together with the proposed early stopping rule has a similar FER performance as the full-block decoder, and it reduces the average number of iterations per VN from 55 for the full-block decoder to 30 at FER = 1e-2.

We also compared two examples of code families supporting different rates via puncturing, or shortening. The results show that the shortened code family performs better than the punctured code family, and have lower computation complexity, especially for the high rate codes.

### REFERENCES


