Low-Latency LDPC Decoding Achieved by Code and Architecture Co-Design

Elsa Dupraz*, François Leduc-Primeau*†, and François Gagnon†
* IMT Atlantique, Lab-STICC, UBL, Brest, France
† École de Technologie Supérieure, Montréal, Canada

***Abstract*—A novel low-density parity-check decoder architecture is presented that can achieve a high data throughput while retaining the flexibility to decode a wide range of quasi-cyclic codes. The proposed architecture allows to combine multiple message-update schedules, providing an additional degree of freedom to jointly optimize the code and decoder architecture. Protograph-based code constructions are introduced that exploit this added degree of freedom in order to maximize data throughput, and that are also optimized to reduce the complexity of the required parallel data accesses. For some examples and under an ideal pipeline speedup assumption, the proposed architecture and code designs reduce decoding latency by a factor of $3.2\times$ compared to a decoder using a strict sequential schedule.**

I. Introduction

A desirable feature of low-density parity-check (LDPC) decoders is the ability to support a wide range of code characteristics, in order to allow code rate and length adaptation, or to handle multiple communication standards with a single decoder. In this paper, we are interested in designing highly parallel LDPC decoder architectures that retain the flexibility to decode any quasi-cyclic (QC) code that satisfies basic constraints (maximum node degrees, maximum lifting factor, etc.). Highly parallel architectures are interesting for applications that demand large data throughputs. They are also useful for low-power operation, since the latency reduction obtained from parallel execution can be traded off to tolerate an increase in propagation delays resulting from the low-voltage operation of the circuit.

Three key strategies are widely used to achieve high-throughput LDPC decoders. The first consists in generating the decoder messages by following a sequential (also known as *serial*) update schedule, which reduces the number of decoding iterations approximately by a factor of two [1]. The other two are standard circuit design strategies: implementing several processing units in parallel, and using circuit pipelining to split up each unit into several stages and thus increase the clock frequency. Unfortunately, it is not possible to use the three techniques simultaneously, because the sequential schedule in general prevents the overlap of computations belonging to different layers.

We propose a novel decoder architecture that can simultaneously use a large number of processing units together with pipelining while also taking advantage of an efficient message-passing schedule. This architecture uses a mechanism called "Δ-updates" to maintain the correctness of the computation irrespective of the message-update schedule. As a result, the schedule can be chosen on a node-by-node basis, providing an additional design parameter that can be optimized. We show that this allows to combine parallel processing with pipelining with only a small penalty in the average number of iterations, resulting in a decoder with a faster convergence time.

Because of the highly parallel nature of the proposed architecture, it becomes challenging to ensure that the required data is always accessible, while maintaining a low complexity for the memory management circuits. We discuss how to optimize the data management for general QC codes, and furthermore propose an optimized code construction that reduces the decoder complexity.

In our construction, the code degree distributions are described by protographs [2], which allows to obtain very efficient QC codes [3]. The standard approach for constructing QC codes from protographs consists of a two-step lifting [3] that aims to improve the minimum distance and girth properties of the code. The first lifting step produces a base matrix from a given protograph by means of a Progressive Edge-Growth (PEG) algorithm [4] that seeks to maximize the girth of the base matrix. The second step is realized with a circulant-PEG algorithm [5] and consists of replacing all the non-zero components of the base matrix by circulant matrices. We propose a modified PEG algorithm for constructing the base matrix at the first lifting step. As shown in our simulation results, the modified construction enables efficient data management at the price of a slight performance degradation.

The remainder of this paper is organized as follows. Section II reviews the standard protograph-based code construction approach. Section III briefly reviews some decoder architectures available in the literature and describes the proposed architecture. Then, Section IV presents our architecture-aware optimized code constructions. Finally, Section V evaluates the error-correction and throughput performance of the proposed codes and architecture.

II. Standard Code Construction

A. Parity-check matrix

The parity check matrix H of size $M \times N$ of an LDPC code can be represented by a bipartite Tanner graph. In this Tanner graph, the set of vertices is composed of N Variable Nodes (VNs) $\mathcal{V} = \{v_1, \cdots, v_N\}$ and M Check Nodes (CNs) $\mathcal{C} = \{c_1, \cdots, c_M\}$. There is an edge between a VN v_n and a CN c_m if $H_{m,n} = 1$. We denote by d_c the degree of a CN, and by $d_{c,\max}$ the largest CN degree in the Tanner graph. We also denote by $\mathcal{C}_v \subseteq \mathcal{C}$ the set of CNs that are connected to VN v, and by $\mathcal{V}_c \subseteq \mathcal{V}$ the set of VNs that are connected to CN c. We now describe the standard method for constructing a parity-check matrix H that ensures good decoding performance.

B. Protographs

A protograph [2] is a small Tanner graph that describes the connections between CNs and VNs in the full Tanner graph of the code. We denote by $M_S \times N_S$ the size of the protograph, and the matrix representation S of the protograph is given by

$$S = \begin{bmatrix} S_{1,1} & \cdots & S_{1,N_S} \\ S_{M_S,1} & \cdots & S_{M_S,N_S} \end{bmatrix}, \tag{1}$$

where the coefficients $S_{i,j}$ are positive integers. A protograph describes the connections between M_S types of CNs and N_S types of VNs. In any LDPC code constructed from the protograph S, any CN of type i will be connected to $S_{i,j}$ VNs of type j. The coefficients $S_{i,j}$ can be greater than 1, which will give parallel edges in the Tanner graph representation of S.

The final code performance highly depends on its underlying protograph S. For an AWGN channel, Density Evolution [6] evaluates the protograph threshold as the minimum SNR that can be tolerated by the decoder to reconstruct the original codeword without error, when the codeword length tends to infinity. For a given rate, the protograph can be optimized by Differential Evolution [7], which aims at finding the protograph with the smallest threshold.

From a given protograph, we can construct a QC parity check matrix H of the desired size by applying the two-steps lifting procedure of [3]. This two-steps lifting will not only allow us to improve the minimum distance and girth properties of the code, but also to address the constraints of the decoder implementation by proposing novel code constructions that only modify one of the two steps of the lifting.

C. Two-steps lifting

The first lifting step aims to construct a base matrix B of size $M_B \times N_B$ from the protograph, where $M_B = Z_1 M_S$, $N_B = Z_1 N_S$, and Z_1 is called the first lifting factor. A base matrix constructed from a given protograph S will contain Z_1 VNs of each of the N_S types and Z_1 CNs of each of the M_S types. In the following, the VNs (resp. CNs) of the base matrix are referred to as B-VN (resp. B-CN). The first lifting is realized by means of a copy-and-permute procedure that first consists of duplicating Z_1 times the protograph S. The edges of the obtained Tanner graph are then interleaved so that the protograph degrees $S_{i,j}$ are fulfilled, the Tanner graph of B is connected, and there is no remaining parallel edges. Edge interleaving is realized by using a PEG algorithm [4] that reduces the amount of short cycles in B, since short cycles could degrade the final code performance.

The second lifting aims to construct a QC parity-check matrix H of size $M \times N$ from the base matrix B, where $M = Z_2 M_B$, $N = Z_2 N_B$, and Z_2 is called the second lifting factor. The second lifting is done by replacing all the non-zero components of the matrix B by circulant matrices of size $Z_2 \times Z_2$. This replacement is realized by a circulant PEG algorithm [5] that again aims at reducing the amount of short cycles in the final parity-check matrix H.

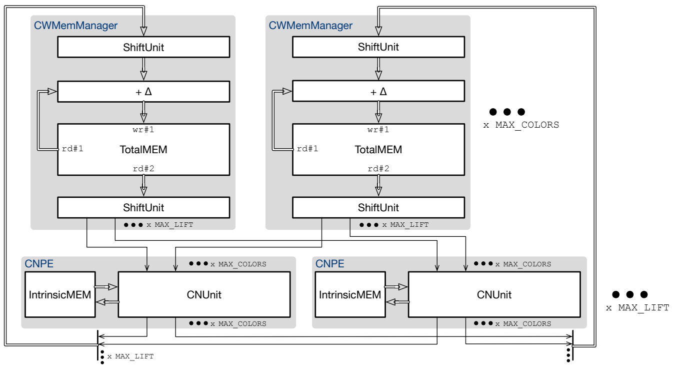

Fig. 1. High-level view of the decoder architecture.

III. Decoder Architecture

A. Review of state-of-the-art architectures

Most architectures in the literature targeted at QC codes perform the processing row-wise and implement the well-known Offset Min-Sum (OMS) algorithm. Two main approaches allow combining the use of a strict row-layered message schedule with parallel computations. The first consists in implementing one [8], [9] or two [10] small processing units that in each clock cycle accept as input the messages from one B-VN to one B-CN and output the messages from one B-CN to one B-VN. The processing of one row layer (i.e. all messages to/from one B-CN) then requires at least d_c/U clock cycles, where U is the number of processing units. Because of the relatively large number of cycles required per layer, it is possible to order the computations in such a way that a deep pipeline can be used while keeping the number of stall cycles at a minimum. A second approach consists in implementing large processing units that process one row layer per clock cycle. In general, such an architecture cannot use pipelining because the layered message schedule requires the processing of the current layer to be completed before the next layer can start. Exceptionally, if the parity-check matrix is designed to ensure that consecutive layers never share a variable node, then a two-stage pipeline can be used [11].

B. The Δ-update architecture

Our proposed architecture is similar to the second approach described above. However, unlike the solution of [11], our architecture admits the use of a moderately deep pipeline by introducing the possibility of ignoring some of the data dependencies of the layered schedule. The architecture, shown in Fig. 1, can be split in two parts. The top part is composed of memory management units that store the belief[1] sums associated with each variable node. The bottom part is composed of at least Z_2 processing units called CNPEs, each

[1]We call *belief* a log-likelihood ratio scaled by a constant.

one responsible for evaluating all the messages sent to and from a particular check node.

Typically, the processing units of a parallel row-layered architecture would take as input a vector $\mathbf{\Lambda}$ of VN belief sums, and output an updated vector $\mathbf{\Lambda'}$. The first novelty of the proposed architecture is that the processing units, rather than generating updated VN sums, compute the difference $\mathbf{\Delta} = \mathbf{\Lambda'} - \mathbf{\Lambda}$. Once the processing completes, this difference is used to update the VN sums. As a result, the architecture seamlessly supports any kind of message schedule. If a particular B-VN is involved in multiple concurrent check-node computations, its message-update schedule is simply altered, while other B-VNs can still benefit from sequential updates.

Compared to a standard row-layered architecture, this modified architecture has one minor drawback. Most state-of-the-art architectures only require one shifting unit per check-node input, by allowing the position of each VN in memory to change throughout the decoding operation. In this architecture, since a particular B-VN might be involved in multiple concurrent check-node computations, the position of VNs in memory must remain fixed, and a second write-side shifting unit is required, as shown in Fig. 1. Note that this shifting unit is smaller than the read-side one, since it routes $\mathbf{\Delta}$ vectors that require fewer bits per element than $\mathbf{\Lambda}$ vectors. Also, the additional delay introduced by this shifting unit is not a major concern since the Δ-update architecture enables the use of a deeper pipeline.

C. Memory access

At each cycle, the processing units must access the belief sums associated with d_c B-VNs, where d_c is the degree of the B-CN currently being processed. Since the architecture is intended to support any quasi-cyclic code, it must support parallel data access to a $\mathbf{\Lambda}$ vector corresponding to any B-VN subset of size $d_{c,\max}$. To avoid requiring the costly routing logic that would be necessary to select such arbitrary subsets, we propose to group all B-VNs into $K \geq d_{c,\max}$ memory banks such that no two B-VNs placed in the same bank need to be accessed simultaneously. We then design the CNPE units so they accommodate up to K inputs. Since the computation involves finding minimum values, unused inputs can easily be disabled by setting their value to the maximum representable value. With this strategy, the complexity of the architecture depends on K. In the following section, we propose a B-VN grouping method that minimizes K.

IV. Proposed Code Construction

A. Memory layout optimization

To optimize the proposed architecture, we would like to group into the same memory bank only B-VNs that do not share any B-CNs as neighbors. More formally, consider K memory banks and denote by $\mathcal{M}_k$, $k \in \{1, \cdots, K\}$, the set of B-VNs that are allocated to the k-th memory bank. For all $k \in \{1, \cdots, K\}$, the set $\mathcal{M}_k$ is constructed such that for all $v, v' \in \mathcal{M}_k$ such that $v \neq v'$,

$$\mathcal{C}_v \cap \mathcal{C}_{v'} = \emptyset. \qquad (2)$$

This condition ensures that the B-VNs allocated to the same memory bank cannot be updated in parallel, so that there is no conflict in memory access. In order to dimension the memory and to allocate each B-VN to a memory bank, we want to partition the set of B-VNs into K sets $\mathcal{M}_k$ that satisfy condition (2). This partitioning problem could be solved as a graph coloring problem applied on a VN-only graph. The VN-only graph contains all the B-VNs as vertices, and there is an edge between two B-VNs if they are connected to at least one common B-CN. A standard graph coloring algorithm [12] is then applied on the VN-only graph in order to construct the sets $\mathcal{M}_k$.

The graph coloring algorithm aims to partition the graph into the minimum possible number of colors. However, with the above approach, this minimum number is determined by the structure of the Tanner graph and some Tanner graphs may not allow for a small number of colors. This is why we would like to minimize the number of colors directly during the code construction. For this, we propose a modified PEG algorithm which we now describe.

B. Modified PEG algorithm

Our modified PEG algorithm replaces the standard PEG algorithm used for the first lifting in the code construction of Section II. This first lifting constructs the base matrix B from a given protograph S. In this section, for simplicity "VN" refers to "B-VN" and "CN" refers to "B-CN".

The proposed algorithm takes the maximum number of colors $K \geq d_{c,\max}$ as input, which gives a set of colors $\{1, 2, \cdots, K\}$. Each CN c maintains a list of colors $\mathcal{L}_c$ containing the colors of its VN neighbors. Each VN v also maintains a list $\mathcal{L}_v$ of the colors of all the VNs with which it shares a common CN. At the beginning of the algorithm, all the lists of colors $\mathcal{L}_c$ and $\mathcal{L}_v$ are initialized to $\emptyset$.

When our modified algorithm needs to add new edges to the Tanner graph, it first selects a VN v at random in $\mathcal{V}$, starting with VNs of highest degrees. Once a VN is selected, the algorithm chooses all its connections in succession instead of just one at random as in the standard PEG. This will allow the algorithm to assign a color to VN v once all its connections are established. When v is selected, its list of colors is given by $\mathcal{L}_v = \emptyset$, since it has no connection yet with any CN. For every edge it wants to assign, the algorithm computes all the distances $d(v, c)$ between this VN and all the CNs $c \in \mathcal{C}$, where $d(v, c)$ is the length of the shortest path between v and c. If there is no path between v and c, then $d(v, c) = +\infty$. In order to add one edge, the algorithm verifies the following saturation and colors condition.

1) *Saturation condition*: the algorithm first constitutes a set that contains for all $j \in \{1, \cdots, S_M\}$, all CNs of type j such that VN v has strictly less than $S_{i,j}$ connections with CNs of type j. From this set, it constitutes $\mathcal{S}$ by retaining for all $j \in \{1, \cdots, S_M\}$ only the CNs of type j that have strictly less than $S_{i,j}$ connections with VNs of type i.

2) *Color condition*: for the current VN v, the algorithm computes the union between its list of colors $\mathcal{L}_v$ and the list of colors of all the CNs $c \in \mathcal{S}$. The set $\mathcal{D}$ is then composed by the CNs that satisfy the color condition, *i.e.* for which the size of the union is strictly lower than the maximum number of colors.

At this step, if $\mathcal{D} = \emptyset$, then the algorithm is restarted. If after a given number of restarts, the algorithm is not able to construct the code, the maximum number of colors must be augmented. If $\mathcal{D} \neq \emptyset$, the algorithm selects at random a CN $\hat{c}$ that both belongs to $\mathcal{D}$ and that has maximum distance with VN v among $\mathcal{D}$. To finish, it adds an edge between v and $\hat{c}$, and it updates the list of colors $\mathcal{L}_v$ of v as $\mathcal{L}_v = \mathcal{L}_v \cup \mathcal{L}_{\hat{c}}$.

Once it added a new edge, the algorithm moves to the next one, until all the connections of VN v have been assigned. It then attributes a color f_v to VN v. This color is selected at random over the set $\{1, 2, \cdots, K\} \setminus \mathcal{L}_v$. The color condition guarantees that this set is not empty. The algorithm also updates the lists of colors $\mathcal{L}_c$ of all the CNs $c \in \mathcal{C}_v$ as $\mathcal{L}_c = \mathcal{L}_c \cup \{f_v\}$. The algorithm may also update all the lists of colors of all the VNs that are connected to CNs $c \in \mathcal{C}_v$, but this is not useful since the edges of these VNs have already been assigned by the algorithm.

When adding a new edge, our algorithm must verify the color condition, which is an additional condition compared to the standard PEG. In the simulation results section, we discuss the influence of this condition on the code performance.

C. Message schedule optimization

Since the decoder architecture is pipelined and processes one B-CN per cycle, T B-CNs are processed concurrently, where T is the number of pipeline stages (we assume that $T \leq M_B$). For a pair of B-CNs present at the same time in the pipeline, some data dependencies of the sequential message-update schedule will be ignored for any B-VN that is connected to both B-CNs. To speed up the convergence of the decoder, we wish to optimize the order in which the B-CNs are processed to minimize the number of such ignored dependencies.

Let us define a weight $w_{i,j}$ that represents the number of dependencies between B-CNs c_i and c_j, i.e., for $i \neq j$, $w_{i,j} = |\mathcal{V}_{c_i} \cap \mathcal{V}_{c_j}|$. We wish to find an ordering of the B-CNs that minimizes

$$\sum_{d=1}^{T-1} \sum_{i=1}^{M_B} w_{i,i\oplus d}\,, \tag{3}$$

where $i \oplus d = (i - 1 + d \mod M_B) + 1$.

Since the number of base-row permutations $M_B!$ is usually too large to be explored exhaustively, we rely on the following randomized greedy algorithm. This algorithm takes as input the set of B-CNs $\mathcal{C}$, and iteratively outputs an ordering $\sigma(t)$, $t \in \{1, 2, \cdots, M_B\}$. As the algorithm iterates, it keeps track of the content of the processing pipeline as a vector P, which contains up to $T - 1$ indices.

1) *Initialization:* The first element $\sigma(1)$ is chosen randomly from the set of B-CNs having the smallest total weight. Formally let $w_{c_i} = \sum_{j=1}^{M_B} w_{i,j}$. Then $\sigma(1)$ is chosen randomly from the set $S_{\text{init}} = \{i : w_{c_i} = \min_{c \in \mathcal{C}}(w_c)\}$ and added as the first element of P.
2) *Iteration $t > 1$:* Subsequent B-CNs are chosen to minimize their dependencies with other nodes in the pipeline. We define $w_{i,P} = \sum_{j \in P} w_{i,j}$. The next element $\sigma(t)$ is chosen randomly from the set $S = \{i : w_{i,P} = \min_{j \in U}(w_{j,P})\}$, where $U = \{1, \cdots, M_B\} \setminus \{\sigma(1), \cdots, \sigma(t-1)\}$ is the set of unassigned indices.
3) *Pipeline update:* After each iteration, the new element $\sigma(t)$ is added at the end of P. After this, if P contains more than $T - 1$ elements, $P(0)$ is discarded and all other elements are moved to the next lower index.

This randomized algorithm can be invoked multiple times to try to improve the global score given by (3).

V. Simulation Results

To evaluate the performance obtained using the proposed QC codes and decoder architecture, we consider a binary-input additive white Gaussian noise channel. The channel output is given by $y = x + w$, where $x \in \{-1, 1\}$ and w is a Gaussian random variable with mean 0 and variance σ^2.

All codes were constructed from the same protograph, which was optimized by differential evolution. In order to increase the sparsity of the base matrices obtained from this protograph, we set $M_S = 2$, $N_S = 4$, and we imposed a maximum value of 3 for the coefficients $S_{i,j}$. The optimization procedure yielded the protograph with $d_{c,\max} = 7$:

$$S = \begin{bmatrix} 0 & 2 & 3 & 1 \\ 2 & 0 & 3 & 2 \end{bmatrix}, \tag{4}$$

From this protograph, we applied the two-steps lifting introduced in Section II. We first used the modified PEG algorithm introduced in Section IV with lifting factor $Z_1 = 36$ and three different maximum number of colors $K = 7, 8, 9$. This provided three base matrices of size 72×144. We also constructed a fourth base matrix of size 72×144 by applying the standard PEG algorithm without any color restriction. In the following, the codes obtained from $K = 7, 8, 9$, are called C_7, C_8, C_9, respectively, and the code constructed without a color restriction is called C_{NR}.

We evaluated that the base matrix of C_7 has girth 4, while the three other base matrices have girth 6. This girth difference can be explained by the fact that for C_7, $K = d_{c,\max} = 7$, which places a difficult constraint on the code construction. We further observed that the base matrices of C_8, C_9, and C_{NR} have approximately the same number of length-6 cycles. Although the code performance does not only depend on cycle distribution, this means that there is a good chance that the final decoding performance of C_8, C_9, and C_{NR}, will be similar. For the second lifting step, we considered $Z_2 = 18$ and we applied the standard circulant PEG algorithm to the four base matrices in order to obtain QC matrices of size 1296×2592. It is worth noting that all four obtained QC-codes have girth 8.

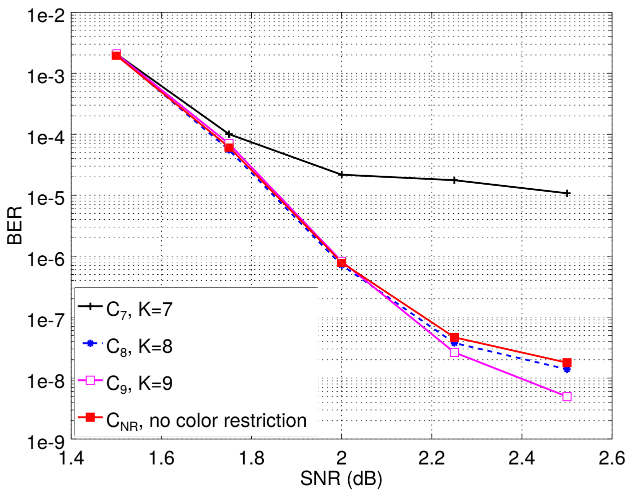

Fig. 2. Performance comparison of the four constructed QC-codes

The bit-error rate (BER) performance of the four codes is obtained with an OMS decoder implemented according to the architecture described in Section III. For each codeword bit, the decoder takes as input a belief value $\mu = \alpha y/\sigma^2$, where α is set to 4 and μ is quantized on 6 bits by rounding it to the nearest integer and saturating it within $[-31, 31]$. The maximum number of iterations is set to 25 and the OMS offset parameter is set to 1. The constructed codes have base matrices with a relatively low density (4.5% of non-zero elements). As a result, it is in fact possible to use the algorithm of Section IV-C to find a row ordering that is compatible with a strict row-layered message schedule (i.e. for which (3) is zero) up to a pipeline depth of $T = 5$. The BER results for this case are shown in Figure 2. Each BER point was obtained from 100 frames in error. We first observe that C_7 shows degraded performance compared to the three other codes. This result was expected since this code is the only one for which the base matrix has girth 4. On the other hand, we observe that C_8 and C_9 have similar performance. C_8 shows a slight performance degradation in the error floor compared to C_9, but it interestingly reduces the architecture memory requirements. Surprisingly, C_{NR} also shows a small performance degradation compared to C_9. The modified PEG algorithm constructs the edges in a different order than the standard PEG, which may explain the performance improvement.

The proposed architecture allows to increase the pipeline depth by ignoring some data dependencies of the row-layered message schedule. To illustrate the impact of this approach, let us assume that the pipeline is ideal, that is it permits a clock period of τ/T, where τ is the clock period without pipelining. We take code C_8 and consider increasing the pipeline depth to $T = 20$. After optimizing the row ordering using the algorithm of Section IV-C, we obtain the BER through Monte-Carlo simulation with an iteration limit of 25 iterations. We find that this BER is approximately equal to the BER obtained using a strict row-layered schedule with a limit of 20 iterations. Therefore, under the ideal pipelining assumption, the deeper pipeline combined with the use of a relaxed schedule decreases latency by a factor of $20/5 \cdot 20/25 = 3.2$.

The proposed approach can also be applied to existing codes. For instance, we consider the rate $\frac{1}{2}$ code defined in the IEEE 802.11n (WiFi) standard, which has a base matrix density of 30%, and cannot be pipelined under a strict row-layered schedule. We evaluated the BER performance of a pipelined decoder with $T = 4$ stages, optimized B-CN ordering, and a maximum of 25 iterations. We find that a decoder using a strict schedule requires 20 iterations to achieve approximately the same BER. Therefore, the proposed pipelined decoder also reduces latency by a factor of $4 \cdot 20/25 = 3.2$ on this code.

VI. Conclusion

This paper introduced a novel LDPC decoder architecture that greatly reduces the decoding latency by carefully combining parallel processing and pipelining. It also proposed new QC code constructions that further improve this throughput and lower the memory requirements of the architecture. Future work will be dedicated to the optimization of the code and decoder parameters for improved latency, decoding performance, and energy consumption.

Acknowledgements

The authors were supported by the grant ANR-17-CE40-0020 of the French National Research Agency ANR (project EF-FECtive).

References

[1] E. Sharon, S. Litsyn, and J. Goldberger, “Efficient serial message-passing schedules for LDPC decoding,” *IEEE Trans. on Information Theory*, vol. 53, no. 11, pp. 4076–4091, Nov 2007.

[2] J. Thorpe, “Low-density parity-check (LDPC) codes constructed from protographs,” *IPN progress report*, vol. 42, no. 154, pp. 42–154, 2003.

[3] D. G. Mitchell, R. Smarandache, and D. J. Costello, “Quasi-cyclic LDPC codes based on pre-lifted protographs,” *IEEE Transactions on Information Theory*, vol. 60, no. 10, pp. 5856–5874, 2014.

[4] X.-Y. Hu, E. Eleftheriou, and D.-M. Arnold, “Regular and irregular progressive edge-growth Tanner graphs,” *IEEE Transactions on Information Theory*, vol. 51, no. 1, pp. 386–398, 2005.

[5] J. Thorpe, K. Andrews, and S. Dolinar, “Methodologies for designing LDPC codes using protographs and circulants,” in *Intl. Symp. on Information Theory (ISIT)*, 2004, p. 238.

[6] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, “Design of capacity-approaching irregular low-density parity-check codes,” *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, 2001.

[7] R. Storn and K. Price, “Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces,” *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[8] C. Studer, N. Preyss, C. Roth, and A. Burg, “Configurable high-throughput decoder architecture for quasi-cyclic LDPC codes,” in *2008 42nd Asilomar Conference on Signals, Systems and Computers*, Oct 2008, pp. 1137–1142.

[9] C. Marchand, L. Conde-Canencia, and E. Boutillon, “Architecture and finite precision optimization for layered LDPC decoders,” in *2010 IEEE Workshop On Signal Processing Systems*, Oct 2010, pp. 350–355.

[10] A. Balatsoukas-Stimming, N. Preyss, A. Cevrero, A. Burg, and C. Roth, “A parallelized layered QC-LDPC decoder for IEEE 802.11ad,” in *11th Intl. New Circuits and Systems Conf. (NEWCAS)*, June 2013, pp. 1–4.

[11] T. T. Nguyen-Ly, V. Savin, K. Le, D. Declercq, F. Ghaffari, and O. Boncalo, “Analysis and design of cost-effective, high-throughput ldpc decoders,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 508–521, March 2018.

[12] F. T. Leighton, “A graph coloring algorithm for large scheduling problems,” *Journal of research of the national bureau of standards*, vol. 84, no. 6, pp. 489–506, 1979.