Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 723465, 15 pages
doi:10.1155/2009/723465

Research Article

Techniques and Architectures for Hazard-Free Semi-Parallel Decoding of LDPC Codes

Massimo Rovini, Giuseppe Gentile, Francesco Rossi, and Luca Fanucci

Department of Information Engineering, University of Pisa, Via G. Caruso 16, 56122 Pisa, Italy

Correspondence should be addressed to Massimo Rovini, massimo.rovini@gmail.com

Received 4 March 2009; Revised 18 May 2009; Accepted 27 July 2009

Recommended by Markus Rupp

The layered decoding algorithm has recently been proposed as an efficient means for the decoding of low-density parity-check (LDPC) codes, thanks to the remarkable improvement in the convergence speed (2x) of the decoding process. However, pipelined semi-parallel decoders suffer from violations or "hazards" between consecutive updates, which not only violate the layered principle but also enforce the loops in the code, thus spoiling the error correction performance. This paper describes three different techniques to properly reschedule the decoding updates, based on the careful insertion of "idle" cycles, to prevent the hazards of the pipeline mechanism. Also, different semi-parallel architectures of a layered LDPC decoder suitable for use with such techniques are analyzed. Then, taking the LDPC codes for the wireless local area network (IEEE 802.11n) as a case study, a detailed analysis of the performance attained with the proposed techniques and architectures is reported, and results of the logic synthesis on a 65 nm low-power CMOS technology are shown.

1. Introduction

Improving the reliability of data transmission over noisy channels is the key issue of modern communication systems and particularly of wireless systems, whose spatial coverage and data rate are increasing steadily.

In this context, low-density parity-check (LDPC) codes have gained the momentum of the scientific community and they have recently been adopted as forward error correction (FEC) codes by several communication standards, such as the second generation digital video broadcasting (DVB-S2, [1]), the wireless metropolitan area networks (WMANs, IEEE 802.16e, [2]), the wireless local area networks (WLANs, IEEE 802.11n, [3]), and the 10 Gbit Ethernet (10Gbase-T, IEEE 802.2ae).

LDPC codes were first discovered by Gallager in the far 1960s [4] but have long been put aside until MacKay and Neal, sustained by the advances in the very high large-scale of integration (VLSI) technology, rediscovered them in the early 1990s [5]. The renewed interest and the success of LDPC codes is due to (i) the remarkable error-correction performance, even at low signal-to-noise ratios (SNRs) and for small block-lengths, (ii) the flexibility in the design of the code parameters, (iii) the decoding algorithm, very suitable for hardware parallelization, and last but not least (iv) the advent of structured or architecture-*aware* (AA) codes [6]. AA-LDPC codes reduce the decoder area and power consumption and improve the scalability of its architecture and so allow the full exploitation of the complexity/throughput design trade-offs. Furthermore, AA-codes perform so close to random codes [6], that they are the common choice of all latest LDPC-based standards.

Nowadays, data services and user applications impose severe low-complexity and low-power constraints and demand very high throughput to the design of practical decoders. The adoption of a fully parallel decoder architecture leads to impressive throughput but unfortunately is also so complex in terms of both area and routing [7] that a semi-parallel implementation is usually preferred (see [6, 8]).

So, to counteract the reduced throughput, designers can act at two levels: at the algorithmic level, by efficiently rescheduling the message-passing algorithm to improve its convergence rate, and at the architectural level, with the pipeline of the decoding process, to shorten the iteration time. The first matter can be solved with the *turbo-decoding message-passing* (TDMP) [6] or the *layered* decoding

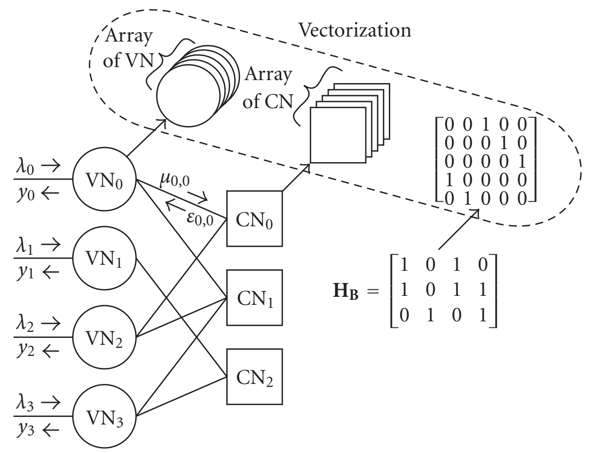


Figure 1: Tanner graph of a simple 3×4 base-matrix and principle of vectorization.

algorithm [9], while pipelined architectures are mandatory especially when the decoder employs serial processing units.

However, the pipeline mechanism may dramatically corrupt the error-correction performance of a layered decoder by letting the processing units not always work on the most updated messages. This issue, known as pipeline "hazard", arises when the dependence between the elaborations is violated. The idea is then to reschedule the sequence of updates and to delay with "idle" cycles the decoding process until newer data are available.

As an improvement to similar state-of-the-art works [10–13], this paper proposes three systematic techniques to optimally reschedule the decoding process in a way to minimize the number of idle cycles and achieve the maximum throughput. Also, this paper discusses different semi-parallel architectures, based on serial processing units and all supporting the reordering strategies, so as to attain the best trade-off between complexity and throughput for every LDPC code.

Semi-parallel architectures of LDPC decoder have recently been addressed in several papers, although none of them formally solves the issue of pipeline hazards and decoding idling. Gunnam et al. describe in [10] a pipelined semi-parallel decoder for WLAN LDPC codes, but the authors do not mention the issue of the pipeline hazards; only, the need of properly scrambling the sequence of data in order to clear some memory conflicts is described.

Boutillon et al. consider in [13] methods and architectures for layered decoding; the authors mention the problem of pipeline hazards (cut-edge conflict) and of using an output order different from the natural one in the processing units; nonetheless, the issue is not investigated further, and they simply modify the decoding algorithm to compute partial updates as in [14]. Although this approach allows the decoder to operate in full pipeline with no idle cycles, it is actually suboptimal in terms of both performance and complexity.

Similarly, Bhatt et al. propose in [11] a pipelined block-serial decoder architecture based on partial updates, but again, they do not investigate the dependence between elaborations.

In [12], Fewer et al. implement a semi-parallel TDMP decoder, but the authors boost the throughput by decoding two codewords in parallel and not by means of pipeline.

This paper is organised as follows. Section 2 recalls the basics of LDPC and of AA-LDPC codes and Section 3 summarizes the layered decoding algorithm. Section 4 introduces three different techniques to reduce the dependence between consecutive updates and analytically derives the related number of idle cycles. After this, Section 5 describes the VLSI architectures of a pipelined block-serial LDPC-layered decoder. Section 6 briefly reviews the WLAN codes used as a case study, while the performances of the related decoder are analysed in Section 7. Then, the results of the logic synthesis on a 65 nm low-power CMOS technology are discussed in Section 8, along with the comparison with similar state-of-the-art implementations. Finally, conclusions are drawn in Section 9.

2. Architecture-Aware Block-LDPC Codes

LDPC codes are linear block-codes described by a parity-check matrix $\mathbf{H}$ establishing a certain number of (even) parity constraints on the bits of a codeword. Figure 1 shows the parity-check matrix of a very simple LDPC code with length $N = 4$ bits and with $M = 3$ parity constraints. LDPC codes are also effectively described in a graphical way through a Tanner graph [15], where each bit in the codeword is represented with a circle, known as variable-node (VN), and each parity-check constraint with a square, known as check-node (CN).

Recently, the joint design of code and decoder has blossomed in many works (see [8, 16]), and several principles have been established for the design of implementation-oriented AA-codes [6]. These can be summarized into (i) the arrangement of the parity-check matrix in squared subblocks, and (ii) the use of deterministic patterns within the subblocks. Accordingly, AA-LDPC codes are also referred to as *block*-LDPC codes [8].

The pattern used within blocks is the vital facet for a low-cost implementation of the interconnection network of the decoder and can be based either on permutations, as in [6] and for the class of π-rotation codes [17], or on *circulants* or cyclic shifts of the identity matrix, as in [8] and in every recent standards [1–3].

AA-LDPC codes are defined by the number of *block*-columns n_c, the number of *block*-rows n_r, and the block-size B, which is the size of the component submatrices. Their parity-check matrix $\mathbf{H}$ can be conveniently viewed as $\mathbf{H} = P^{\mathbf{H_B}}$, that is, as the expansion of a base-matrix $\mathbf{H_B}$ with size $n_r \times n_c$. The expansion is accomplished by replacing the 1's in $\mathbf{H_B}$ with permutations or circulants, and the 0's with null subblocks. Thus, the block-size B is also referred to as *expansion*-factor, for a codeword length of the resulting LDPC code equal to $N = B \cdot n_c$ and code rate $r = 1 - n_r/n_c$.

A simple example of expansion or *vectorization* of a base-matrix is shown in Figure 1. The size, number, and location of the nonnull blocks in the code are the key parameters to get good error-correction performance and low-complexity of the related decoder.

3. Decoding of LDPC Codes

LDPC codes are decoded with the *belief propagation* (BP) or *message-passing* (MP) algorithm, that belong to the broader class of maximum *a posteriori* (MAP) algorithms. The BP algorithm has been proved to be optimal if the graph of the code does not contain cycles, but it can still be used and considered as a reference for practical codes with cycles. In the latter case, the sequence of the elaborations, also referred to as *schedule*, considerably affects the achievable performance.

The most common schedule for BP is the so-called two-phase or flooding schedule (FS) [18], where all parity-check nodes first, followed by all variable nodes then, are updated in sequence.

A different approach, taking the distribution of closed paths and girths in the code into account, has been described by Xiao and Banihashemi in [19]. Although *probabilistic* schedules are shown to outperform *deterministic* schedules, the random activation strategy of the processing nodes is not very suitable to HW implementation and adds significant complexity overheads.

The most attractive schedule is the *shuffled* or *layered* decoding [6, 9, 18, 20]. Compared to the FS, the layered schedule almost doubles the decoding convergence speed, both for codes with cycles and cycle-free [20]. This is achieved by looking at the code as a connection of smaller supercodes [6] or *layers* [9], exchanging intermediate reliability messages. Specifically, *a posteriori* messages are made available to the next layers immediately after computation and not at next iteration as in a conventional flooding schedule.

Layers can be any set of either CNs or VNs, and, accordingly, CN-*centric* (or horizontal) or VN-*centric* (or vertical) algorithms have been analyzed in [18, 20]. However, CN-*centric* solutions are preferable since they can exploit serial, flexible, and low-complexity CN processors.

The horizontal layered decoding (HLD) is summarized in Algorithm 1 and consists in the exchange of probabilistic reliability messages around the edges of the Tanner graph (see Figure 1) in the form of logarithms of likelihood ratios (LLRs); given the random variable x, its LLR is defined as

$$\mathrm{LLR}(x) = \log \frac{\Pr(x=1)}{\Pr(x=0)}. \quad (1)$$

In Algorithm 1, λ_n is the nth *a priori* LLR of the received bits, with $n = 0, 1, \ldots, N-1$ and N the length of the codeword, M is the overall number of parity-check constraints, and N_{it} the number of decoding iterations. Also, $\mathcal{N}(m)$ is the set of VNs connected to the mth CN, $\epsilon_{m,n}^{(q)}$ represents the check-to-variable (*c2v*) reliability message sent from CN m to VN n at iteration q, and y_n is the total information or *soft-output* (SO) of the nth bit in the codeword (see Figure 1).

For the sake of an easier notation, it is assumed here that a layer corresponds to a single row of the parity-check matrix. Before being used by the next CN or layer, SOs are refined with the involved *c2v* message, as shown in line 13, and thanks to this mechanism, faster convergence is achieved.

```
input: a-priori LLR $\lambda_n$, $n = 0, 1, \ldots, N-1$
output: a-Posteriori hard-decisions $\hat{y}_n = \mathrm{sign}(y_n)$
(1) // Messages initialization
(2) $q = 0$, $y_n = \lambda_n$, $\epsilon_{m,n}^{(0)} = 0$, $\forall n = 0, \ldots, N-1$,
     $\forall m = 0, \ldots, M-1$;
(3) while ($q < N_{\mathrm{it}}$ & !Convergence) do
(4)     // Loop on all layers
(5)     for $m \leftarrow 0$ to $M-1$ do
(6)         // Check-node update
(7)         forall $n \in \mathcal{N}(m)$ do
(8)             // Sign update
(9)             $-\mathrm{sign}(\epsilon_{m,n}^{(q+1)}) = \prod_{j \in \mathcal{N}(m)\setminus n} -\mathrm{sign}(y_j - \epsilon_{m,j}^{(q)})$;
(10)            // Magnitude update
(11)            $|\epsilon_{m,n}^{(q+1)}| = \mathrm{M-min}^*_{j \in \mathcal{N}(m)\setminus n}(|y_j - \epsilon_{m,j}^{(q)}|)$;
(12)            // Soft-output update
(13)            $y_n = y_n - \epsilon_{m,n}^{(q)} + \epsilon_{m,n}^{(q+1)}$
(14)        end
(15)    end
(16)    $q++$;
(17) end
```

Algorithm 1: Horizontal layered decoding.

Magnitudes are updated with the M-min* binary operator [21] defined as $\mathrm{M\text{-}min}^*(a,b) \doteq \min(a,b) + \log(e^{|a-b|}/(1+e^{|a-b|}))$ for $a, b \geq 0$. Following an approach similar to Jones et al. [22], the updating rule of magnitudes is further simplified with the method described in [23], which proved to yield very good performance. Here, only two values are computed and propagated for the magnitude of *c2v* messages; specifically, if we define

$$j_{\min} = \arg\left\{ \min_{j \in \mathcal{N}(m)} \left(\left| y_j - \epsilon_{m,j}^{(q)} \right| \right) \right\} \quad (2)$$

the index of the smallest variable-to-check (*v2c*) message entering CN m, then a dedicated *c2v* message is computed in response to VN $j_{\min}$:

$$\left| \epsilon_{m,j_{\min}}^{(q+1)} \right| = \mathrm{M\text{-}} \min^*_{j \in \mathcal{N}(m), j \neq j_{\min}} \left(\left| y_j - \epsilon_{m,j}^{(q)} \right| \right) \doteq \alpha_m \quad (3)$$

while all the remaining VNs receive one common, *non-marginalized* value for magnitude given by

$$\left| \epsilon_{m,n}^{(q+1)} \right|_{n \neq j_{\min}} = \mathrm{M\text{-}min}^* \left(\alpha_m, \left| y_{j_{\min}} - \epsilon_{m,j_{\min}}^{(q)} \right| \right) \doteq \beta_m. \quad (4)$$

4. Decoding Pipelining and Idling

The data-flow of a pipelined decoder with serial processing units is sketched in Figure 2. A centralized memory unit keeps the updated soft-outputs, computed by the node processors (NPs) according to Algorithm 1. If we denote with d_k the number of nonnull blocks in layer k, that is, the degree of layer k, then the processor takes d_k clock cycles to serially load its inputs. Then, refined values are written back in memory (after scrambling or permutation) with the

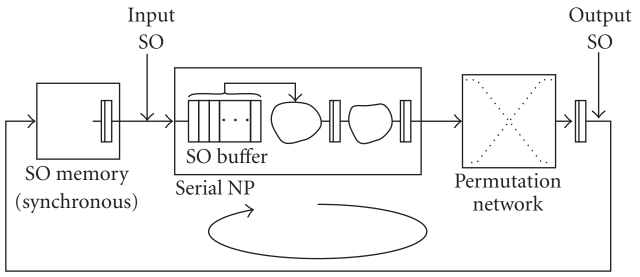


FIGURE 2: Outline of the flow of soft-outputs in an LDPC-layered decoder with serial processing units.

latency of L_{SO} clock cycles, and this operation takes again d_k clock cycles. Overall, the processing time of layer k is then $2d_k + L_{SO}$ clock cycles, as shown in Figure 3(a).

If the decoder works in pipeline, time is saved by overlapping the phases of elaboration, writing-out and reading, so that data are continuously read from and written into memory, and a new layer is processed every d_k clock cycles (see Figure 3(b)).

Although highly desirable, the pipeline mechanism is particularly challenging in a layered LDPC decoder, since the soft-outputs retrieved from memory and used for the current elaboration could not be always up-to-date, but newer values could be still in the pipeline. This issue, known as pipeline hazard, prevents the use and so the propagation of always up-to-date messages and spoils the error-correction performance of the decoding algorithm.

The solution investigated in this paper is to insert *null* or *idle* cycles between consecutive updates, so that a node processor is suspended to wait for newer data. The number of idle cycles must be kept as small as possible since it affects the iteration time and so the decoding throughput. Its value depends on the actual sequence of layers updated by the decoder as well as on the order followed to update messages within a layer.

Three different strategies are described in this section, to reduce the dependence between consecutive updates in the HLD algorithm and, accordingly, the number of idle cycles. These differ in the order followed for acquisition and writing-out of the decoding messages and constitute a powerful tool for the design of "layered", hazard-free, LDPC codes.

4.1. System Notation. Without any lack of generality, let us identify a layer with one single parity-check node and focusing on the set $\mathcal{S}_k$ of soft-outputs participating to layer k, let us define the following subsets:

(i) $\mathcal{A}_k = \mathcal{S}_k \cap \mathcal{S}_{k-1}$, the set of SOs in common with layer $k-1$;

(ii) $\mathcal{B}_k = \{\mathcal{S}_k \cap \mathcal{S}_{k+1}\} \setminus \mathcal{S}_{k-1}$, the set of SOs in common with layer $k+1$ and not in $\mathcal{A}_k$;

(iii) $\mathcal{C}_k = \mathcal{S}_{k-1} \cap \mathcal{S}_k \cap \mathcal{S}_{k+1}$, the set of SOs in common with both layers $k-1$ and $k+1$;

(iv) $\mathcal{E}_k = \{\mathcal{S}_k \cap \mathcal{S}_{k-2}\} \setminus \{\mathcal{S}_{k-1} \cup \mathcal{S}_{k+1}\}$, the set of SOs in common with layer $k-2$ and not in $\mathcal{A}_k$ or $\mathcal{B}_k$;

(v) $\mathcal{F}_k = \{\mathcal{S}_k \cap \mathcal{S}_{k+2}\} \setminus \{\mathcal{S}_{k-2} \cup \mathcal{S}_{k-1} \cup \mathcal{S}_{k+1}\}$, the set of SOs in common with layer $k+2$ but not in $\mathcal{E}_k$, $\mathcal{A}_k$, $\mathcal{B}_k$;

(vi) $\mathcal{G}_k = \{\mathcal{S}_{k-2} \cap \mathcal{S}_k \cap \mathcal{S}_{k+2}\} \setminus \{\mathcal{S}_{k-1} \cup \mathcal{S}_{k+1}\}$, the set of SOs in common with both layers $k-2$ and $k+2$, but not in $\mathcal{A}_k$ or $\mathcal{B}_k$;

(vii) $\mathcal{R}_k$, the set of remaining SOs.

In the definitions above the notation $A \setminus B$ means the relative complement of B in A or the set-theoretic difference of A and B. Let us also define the following cardinalities: $d_k = |\mathcal{S}_k|$ (degree of layer k), $\alpha_k = |\mathcal{A}_k|$, $\beta_k = |\mathcal{B}_k|$, $\chi_k = |\mathcal{C}_k|$, $\epsilon_k = |\mathcal{E}_k|$, $\phi_k = |\mathcal{F}_k|$, $\gamma_k = |\mathcal{G}_k|$, $\rho_k = |\mathcal{R}_k|$.

4.2. Equal Output Processing. First, let us consider a very straightforward and implementation friendly architecture of the node processor that updates (and so delivers) the soft-output messages with the same order used to take them in.

In such a case it would be desirable to (i) postpone the acquisition of messages updated by the previous layer, that is, messages in $\mathcal{A}_k$, and (ii) output the messages in $\mathcal{B}_k$ as soon as possible to let the next layer start earlier. Actually, the last constraint only holds when $\mathcal{A}_k$ does not include any message common to layer $k+1$, that is, when $\mathcal{C}_k = \emptyset$; otherwise, the set $\mathcal{B}_k$ could be acquired at any time before $\mathcal{A}_k$.

Figure 4 shows the I/O data streams of an equal output processing (EOP) unit. Here, L_{SO} is the latency of the SO data-path, including the elaboration in the NP, the scrambling, and the two memory accesses (reading and writing). Focusing on layer $k+1$, the set $\mathcal{C}_{k+1}$ cannot be assigned to any specific position within $\mathcal{A}_{k+1}$, since the whole $\mathcal{A}_{k+1}$ is acquired according to the same order used by layer k to output (and so also acquire) the sets $\mathcal{B}_k$ and $\mathcal{C}_k$. For this reason, the situation plotted in Figure 4 is only for the sake of a clearer drawing.

With reference to Figure 4, pipeline hazards are cleared if $\mathcal{I}_k$ idle cycles are spent between layer k and $k+1$ so that

$$\mathcal{I}_k + |\mathcal{S}_{k+1} \setminus \mathcal{A}_{k+1}| \geq L_{SO} + |\mathcal{S}_k \setminus (\mathcal{A}_k \cup \mathcal{B}_k)| \cdot u(|\mathcal{C}_k|) \quad (5)$$

with $u(x) = 1$ for $x > 0$ and $u(x) = 0$ otherwise. This means that if $\mathcal{C}_k$ is empty, then the messages in $\mathcal{S}_k \setminus (\mathcal{A}_k \cup \mathcal{B}_k)$ do not need to be waited for. The solution to (5) with minimum latency is

$$\mathcal{I}_k = L_{SO} - (d_{k+1} - \alpha_{k+1}) + (d_k - \alpha_k - \beta_k) \cdot u\left(\chi_k\right). \quad (6)$$

Note that (5) and (6) only hold under the hypothesis of $\mathcal{C}_k$ leading within $\mathcal{A}_k$. If this is not the case, up to $|\mathcal{A}_k \setminus \mathcal{C}_k|$ extra idle cycles could be added if $\mathcal{C}_k$ is output last within $\mathcal{A}_k$.

So far, we have only focused on the interaction between two consecutive layers; however, violations could also arise between layer k and $k+2$. Despite this possibility, this issue is not treated here, as it is typically mitigated by the same idle cycles already inserted between layers k and $k+1$ and between layers $k+1$ and $k+2$.

4.3. Reversed Output Processing. Depending on the particular structure of the parity-check matrix $\mathbf{H}$, it may occur that the

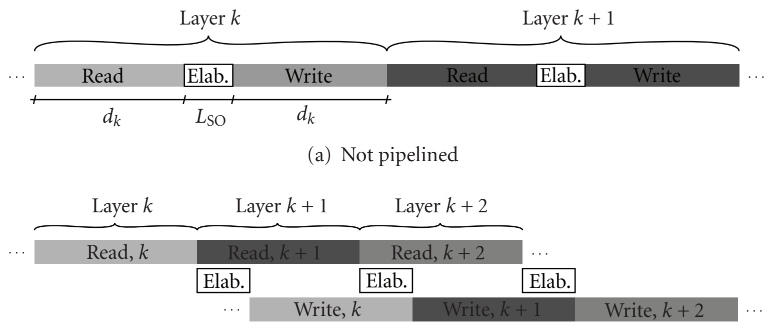


Figure 3: Pipelined and not pipelined data-flow.

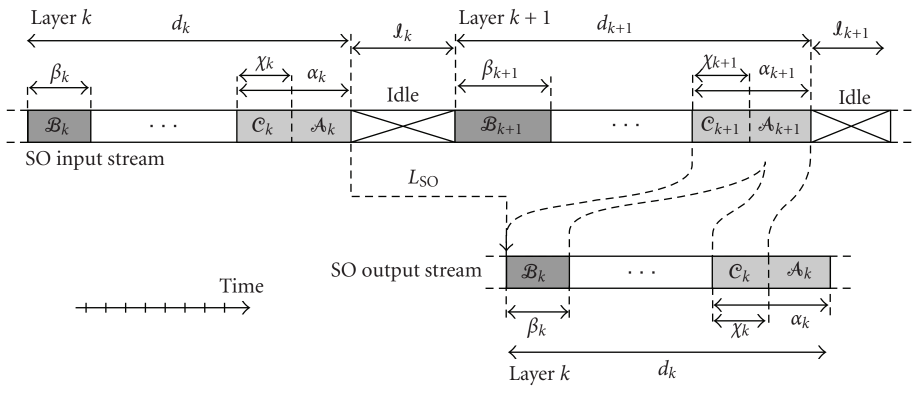


Figure 4: Input and output data streams in an NP with EOP.

most of the messages of layer k in common with layer $k-1$ are also shared with layer $k+1$, that is, $\mathcal{A}_k \simeq \mathcal{C}_k$ and $\mathcal{B}_k \simeq \emptyset$. If this condition holds, as for the WLAN LDPC codes (see Figure 11), it can be worth reversing the output order of SOs so that the messages in $\mathcal{A}_k$ can be both acquired last and output first.

Figure 5(a) shows the I/O streams of a reversed output processing (ROP) unit. Exploiting the reversal mechanism, the set $\mathcal{B}_k$ is acquired second-last, just before $\mathcal{A}_k$, so that it is available earlier for layer $k+1$.

Following a reasoning similar to EOP, the situation sketched in Figure 5(a) where $\mathcal{C}_k$ is delivered first within $\mathcal{A}_k$ is just for an easier representation, and the condition for hazard-free layered decoding is now

$$\mathcal{I}_k^{2l} + |\mathcal{S}_{k+1} \setminus \mathcal{A}_{k+1}| \geq L_{SO} + |\mathcal{A}_k \setminus \mathcal{C}_k| \cdot u(|\mathcal{B}_k|). \quad (7)$$

Indeed, when $\mathcal{B}_k = \emptyset$, one could output $\mathcal{C}_k$ first in $\mathcal{A}_k$, and so get rid of the term $|\mathcal{A}_k \setminus \mathcal{C}_k|$. However, since $\mathcal{C}_k$ is actually left floating within $\mathcal{A}_k$, (7) represents again a best-case scenario, and up to $|\mathcal{A}_k \setminus \mathcal{C}_k|$ extra idle cycles could be required. From (7), the minimum latency solution is

$$\mathcal{I}_k^{2l} = L_{SO} - (d_{k+1} - \alpha_{k+1}) + \left(\alpha_k - \chi_k\right) \cdot u(\beta_k). \quad (8)$$

Similarly to EOP, the ROP strategy also suffers from pipeline hazards between three consecutive layers, and because of the reversed output order, the issue is more relevant now. This situation is sketched in Figure 5(b), where the sets $\mathcal{E}_k$, $\mathcal{F}_k$, and $\mathcal{G}_k$ are managed similarly to $\mathcal{A}_k$, $\mathcal{B}_k$, and $\mathcal{C}_k$. The ROP strategy is then instructed to acquire the set $\mathcal{E}_k$ later and to output $\mathcal{F}_k$ earlier. However, the situation is complicated by the fact that the set $\mathcal{F}_{k-1} \cup \mathcal{G}_{k-1}$ may not entirely coincide with $\mathcal{E}_{k+1}$; rather it is $\mathcal{E}_{k+1} \subseteq (\mathcal{F}_{k-1} \cup \mathcal{G}_{k-1})$, since some of the messages in $\mathcal{F}_{k-1} \cup \mathcal{G}_{k-1}$ can be found in $\mathcal{B}_{k+1}$. This is highlighted in Figure 5(b), where those messages of $\mathcal{F}_{k-1}$ and $\mathcal{G}_{k-1}$ not delivered to $\mathcal{E}_{k+1}$ are shown in dark grey.

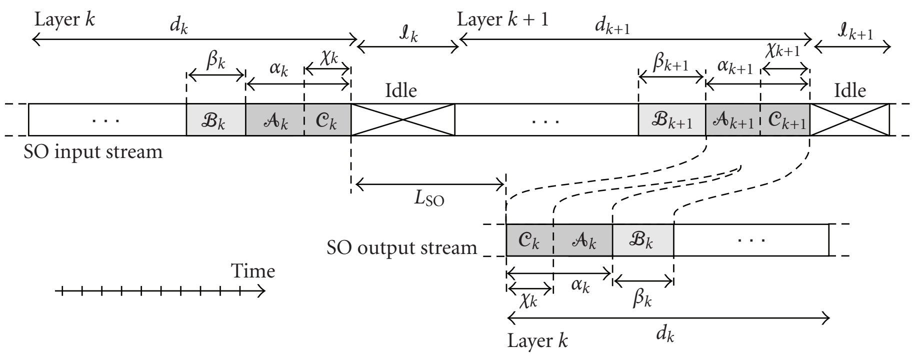
To clear the hazards between three layers, additional idle cycles are added in the number of

$$\mathcal{I}_k^{3l} = \max\{\text{ACQ}_{k+1} - \text{WR}_{k-1}, 0\}, \quad (9)$$

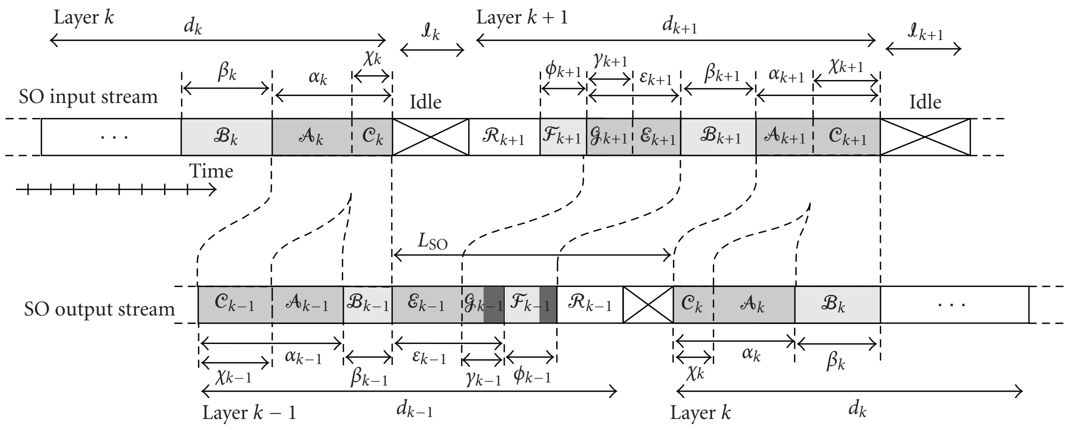
where ACQ_{k+1} is the acquisition margin on layer $k+1$, and WR_{k-1} is the writing-out margin on layer $k-1$. These can be computed under the assumption of no hazard between layer $k-1$ and k (i.e., $\mathcal{C}_k \cup \mathcal{A}_k$ is aligned with $\mathcal{C}_{k-1} \cup \mathcal{B}_{k-1}$ thanks to $\mathcal{I}_k^{2l}$ as shown in Figure 5(b)) and are given by

$$\begin{aligned} \text{ACQ}_{k+1} &= \mathcal{I}_k^{2l} + d_{k+1} - (\alpha_{k+1} + \beta_{k+1} + \epsilon_{k+1}), \\ \text{WR}_{k-1} &= (\epsilon_{k-1} - \gamma_{k-1} + |\mathcal{G}_{k-1} \setminus \mathcal{E}_{k+1}|) \cdot u(\phi_{k-1}). \end{aligned} \quad (10)$$

The margin WR_{k-1} is actually nonnull only if $\mathcal{F}_{k-1} \neq \emptyset$; otherwise, $\text{WR}_{k-1} = 0$ under the hypothesis that (i) the set $\mathcal{G}_{k-1}$ is output first within $\mathcal{E}_{k-1}$, and (ii) within $\mathcal{G}_{k-1}$, the messages not in $\mathcal{E}_{k+1}$ are output last.



(a) Pipeline hazards in the update of two consecutive layers



(b) Pipeline hazards in the update of three consecutive layers. Messages of $\mathcal{G}_{k-1}$ and $\mathcal{F}_{k-1}$ not in $\mathcal{E}_{k+1}$ are shown in dark grey

FIGURE 5: Organization of the input and output data stream in an NP with ROP.

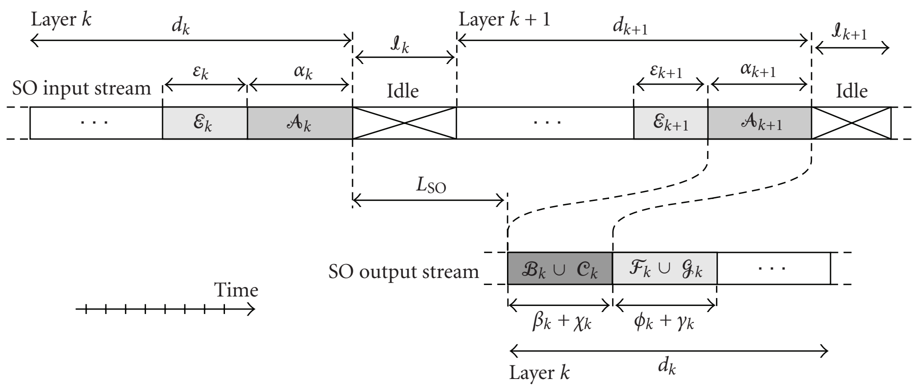


FIGURE 6: Input and output data streams in an NP with UOP.

Overall, the number of idle cycles of ROP is given by

$$\mathcal{I}_k = \mathcal{I}_k^{2l} + \mathcal{I}_k^{3l}. \quad (11)$$

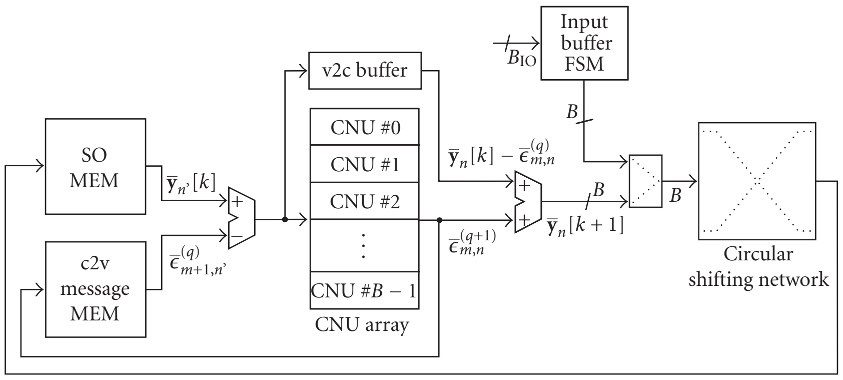
4.4. Unconstrained Output Processing. Fewer idle cycles are expected if the orders used for input and output are not constrained to each other. This implies that layer k can still delay the acquisition of the messages updated by layer $k-1$ (i.e., messages in $\mathcal{A}_k$) as usual, but at the same time the messages common to layer $k+1$ (i.e., in $\mathcal{B}_k \cup \mathcal{C}_k$) can also be delivered earlier.

The input and output data streams of an unconstrained output processing (UOP) unit are shown in Figure 6. Now, hazard-free layered decoding is achieved when

$$\mathcal{I}_k + |\mathcal{S}_{k+1} \setminus \mathcal{A}_{k+1}| \geq L_{SO}, \quad (12)$$

which yields

$$\mathcal{I}_k = L_{SO} - (d_{k+1} - \alpha_{k+1}). \quad (13)$$



Input buffer FSM
B_{IO}
v2c buffer
SO MEM
c2v message MEM
$\overline{y}_{n'}[k]$
$\overline{\epsilon}^{(q)}_{m+1,n'}$
CNU #0
CNU #1
CNU #2
CNU #$B-1$
CNU array
$\overline{y}_n[k] - \overline{\epsilon}^{(q)}_{m,n}$
$\overline{\epsilon}^{(q+1)}_{m,n}$
$\overline{y}_n[k+1]$
B
Circular shifting network

FIGURE 7: Layered decoder architecture with variable-to-check buffer.

Regarding the interaction between three consecutive layers, if the messages common to layer $k+2$ (i.e., in $\mathcal{F}_k \cup \mathcal{G}_k$) are output just after $\mathcal{B}_k \cup \mathcal{C}_k$, and if on layer $k+2$, the set $\mathcal{E}_{k+2}$ is taken just before $\mathcal{A}_{k+2}$, then there is no risk of pipeline hazard between layer k and $k+2$.

4.5. Decoding of Irregular Codes. A serial processor cannot process consecutive layers with decreasing degrees, $d_{k+1} < d_k$, as the pipeline of the internal elaborations would be corrupted and the output messages of the two layers would overlap in time. This is not but another kind of pipeline hazard, and again, it can be solved by delaying the update of the second layer with $\Delta d_k = d_k - d_{k+1}$ idle cycles.

Since this type of hazard is independent of that seen above, the same idle cycles may help to solve both issues. For this reason, the overall number of idle cycles becomes

$$\mathcal{I}'_k = \max\{\mathcal{I}_k, \Delta d_k, 0\} \tag{14}$$

with $\mathcal{I}_k$ being computed according to (6), (11), or (13).

4.6. Optimal Sequence of Layers. For a given reordering strategy, the overall number of idle cycles per decoding iteration is a function of the actual sequence of layers used for the decoding. For a code with Λ layers, the optimal sequence of layer $\hat{p}$ minimizing the time spent in idle is given by

$$\hat{p} = \arg\min_{p\in\mathcal{P}} \left\{ \sum_{k=0}^{\Lambda-1} \mathcal{I}'_k(p) \right\}, \tag{15}$$

where $\mathcal{I}'_k(p)$ is the number of idle cycles between layer k and $k+1$ for the generic permutation p and is given by (14), and $\mathcal{P}$ is the set of the possible permutations of layers.

The minimization problem in (15) can be solved by means of a brute-force computer search and results in the definition of a permuted parity-check matrix $\hat{\mathbf{H}}$, whose layers are scrambled according to the optimal permutation $\hat{p}$. Then, within each layer of $\hat{\mathbf{H}}$, the order to update the nonnull subblocks is given by the strategy in use among EOP, ROP, and UOP.

4.7. Summary and Results. The three methods proposed in this section are differently effective to minimize the overall time spent in idle. Although UOP is expected to yield the smallest latency, the results strongly depend on the considered LDPC code, and ROP and EOP can be very close to UOP. As a case-example, results will be shown in Section 7 for the WLAN LDPC codes.

However, the effectiveness of the individual methods must be weighed up in view of the requirements of the underlying decoder architecture and the costs of its hardware implementation, which is the objective of Section 5. Thus, UOP generally requires bigger complexity in hardware, and EOP or ROP can be preferred for particular codes.

5. Decoder Architectures

Low complexity and high throughput are key features demanded to every competitive LDPC decoder, and to this extent, semi-parallel architectures are widely recognised as the best design choice.

As shown in [6, 8, 12] to mention just a few, a semi-parallel architecture includes an array of processing elements with size usually equal to the expansion factor B of the base-matrix $\mathbf{H_B}$. Therefore, the HLD algorithm described in Section 3 must be intended in a vectorized form as well, and in order to exploit the code structure, a layer counts B consecutive parity-check nodes. Layers (in the number of $n_r = M/B$) are updated in sequence by the B check-node units (CNUs), and an array of B SOs ($\overline{y}_n$) and of $Bc2v$ messages ($\overline{\epsilon}^{(q)}_{m,n}$) are concurrently updated at every clock cycle. Since the parity-check equations in a layer are independent by construction, that is, they do not share SOs, the analysis of Section 4 still holds in a vectorized form.

The CNUs are designed to serially update the *c2v* magnitudes according to (3) and (4), and any arbitrary order of the *c2v* messages (and so of SOs, see line 13 of Algorithm 1) can be easily achieved by properly multiplexing between the two values as also shown in [23]. It must be pointed out that the 2-output approximation described in Section 3 is pivotal to a low-complexity implementation of EOP, ROP, or UOP in the CNU. However, the same strategies could also be used with a different (or even no) approximation in the CNU, although the cost of the related implementation would probably be higher.

Three VLSI architectures of a layered decoder will be described, that differ in the management of the memory

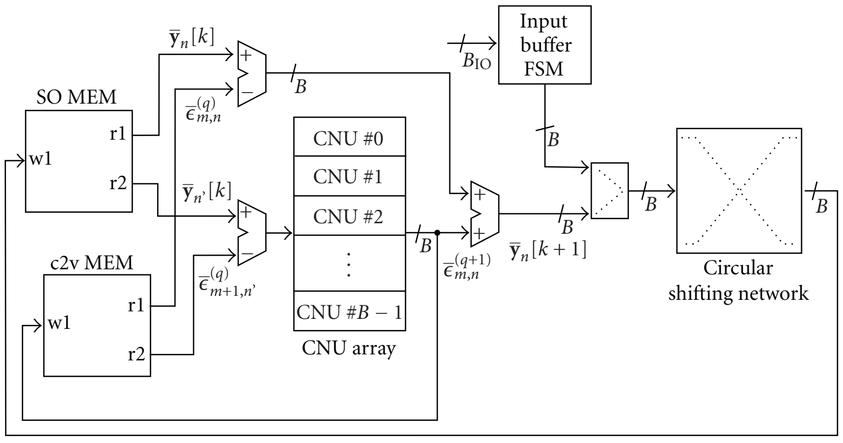

Input buffer FSM
B_{IO}
SO MEM
w1
r1
r2
c2v MEM
w1
r1
r2
$\overline{\mathbf{y}}_n[k]$
$\overline{\epsilon}^{(q)}_{m,n}$
$\overline{\mathbf{y}}_{n'}[k]$
$\overline{\epsilon}^{(q)}_{m+1,n'}$
CNU #0
CNU #1
CNU #2
CNU #$B-1$
CNU array
$\overline{\epsilon}^{(q+1)}_{m,n}$
$\overline{\mathbf{y}}_n[k+1]$
B
Circular shifting network

FIGURE 8: Layered decoder with *three*-port SO and *c2v* memories.

units of both SO and *c2v*, and so result in different implementation costs in terms of memory (RAM and ROM) and logic.

5.1. Local Variable-to-Check Buffer. The most straightforward architecture of a vectorized layered decoder is shown in Figure 7. Here, the arrays of *v2c* messages $\overline{\mu}^{(q)}_{m,n}$ entering the CNUs during the update of layer $m = 0, 1, \ldots, n_r - 1$, are computed on-the-fly as $\overline{\mu}^{(q)}_{m,n} = \overline{\mathbf{y}}_n - \overline{\epsilon}^{(q)}_{m,n}$ with $n \in \mathcal{N}(m)$, and both the arrays of *c2v* and SO messages are retrieved from memory.

Then, the updated *c2v* messages are used to refine every array of SOs belonging to layer m: according to line 13 of Algorithm 1, this is done by adding the new *c2v* array $\overline{\epsilon}^{(q+1)}_{m,n}$ to the input *v2c* array $\overline{\mu}^{(q)}_{m,n}$. Since the CNUs work in pipeline, while the update of layer m is still progress, the array of the *v2c* messages belonging to layer $m + 1$ is already being computed as $\overline{\mu}^{(q)}_{m+1,n'} = \overline{\mathbf{y}}_{n'} - \overline{\epsilon}^{(q)}_{m+1,n'}$, with $n' \in \mathcal{N}(m+1)$. For this reason, $\overline{\mu}^{(q)}_{m,n}$ needs to be temporarily stored in a local buffer as shown in Figure 7. The buffer is vectorized as well and stores $B \times d_{c,\max}$ messages, with $d_{c,\max}$ the maximum CN degree in the code.

Before being stored back in memory, the array $\overline{\mathbf{y}}_n$ is circularly shifted and made ready for its next use, by applying *compound* or *incremental* rotations [12]; this operation is carried out by the *circular* shifting network of Figure 7, and more details about its architecture are available in [24].

The *v2c* buffer is the key element that allows the architecture to work in pipeline. This has to sustain one reading and one writing access concurrently and can be efficiently implemented with shift-register based architectures for EOP (first-in, first-out, FIFO buffer) and ROP (last-in, first-out, LIFO buffer). On the contrary, UOP needs to map the buffer onto a *dual*-port memory bank, whose (reading) address is provided by and extra configuration memory (ROM).

5.2. Double Memory Access. The buffer of Arch. V-A can be removed if the *v2c* messages are computed twice on-the-fly, as shown in Figure 8: the first time to feed the array of CNUs, and then to update the SOs. To this aim, a further reading is required to get the arrays $\overline{\mathbf{y}}_n$ and $\overline{\epsilon}^{(q)}_{m,n}$ from memory, and so recompute the array $\overline{\mu}^{(q)}_{m,n}$ on the CNUs output.

It follows that *three*-port memories are needed for both SO and *c2v* messages since three concurrent accesses have to be supported: two readings (see ports $r1$ and $r2$ in Figure 8) and one writing. This memory can be implemented by distributing data on several banks of customary *dual*-port memory, in such a way that two readings always involve different banks. Actually, in a layered decoder a same memory location needs to be accessed several times per iteration and concurrently to several other data, so that resorting to only two memory banks would be unfeasible. On the other hand, the management of a higher number of banks would add a significant overhead to the complexity of the whole design.

The proposed solution is sketched in Figure 9 and is based on only two banks (A and B) but, to clear access conflicts, some data are redundantly stored in both the banks (see elements C1 and C2 in the example of Figure 9).

The most trivial and expensive solution is achieved when both banks are a full copy or a *mirror* of the original memory as in [11], which corresponds to 100% redundancy. Conversely to this route, data can be selectively assigned to the two banks through computer search aiming at a minimum redundancy.

Roughly speaking, if we denote by σ_i the cardinality of the set of data (SO or *c2v* messages) read concurrently to the ith data for $i = 0, 1, \ldots, N-1$, then the higher $\sum_{\forall i} \sigma_i$ is (for a given N), the higher is the expected redundancy. So, a small redundancy ρ_{c2v} is experienced by the *c2v* memory, since each *c2v* message can collide with at most two other data (i.e., $\max_i\{\sigma_i\} = 2$), while a higher redundancy ρ_{SO} is associated to the SO memory, since every SO can face up to $2d_{\text{VN},n}$ conflicts, with $d_{\text{VN},n}$ being the degree of the nth variable node, typically greater than 1 (especially for low-rate codes).

Indeed, the issue of memory partitioning and the reordering techniques described in Section 4 are linked to each other: whenever the CNUs are in idle, only one reading is performed. Therefore, an overall system optimization aiming at minimizing the iteration latency and the amount

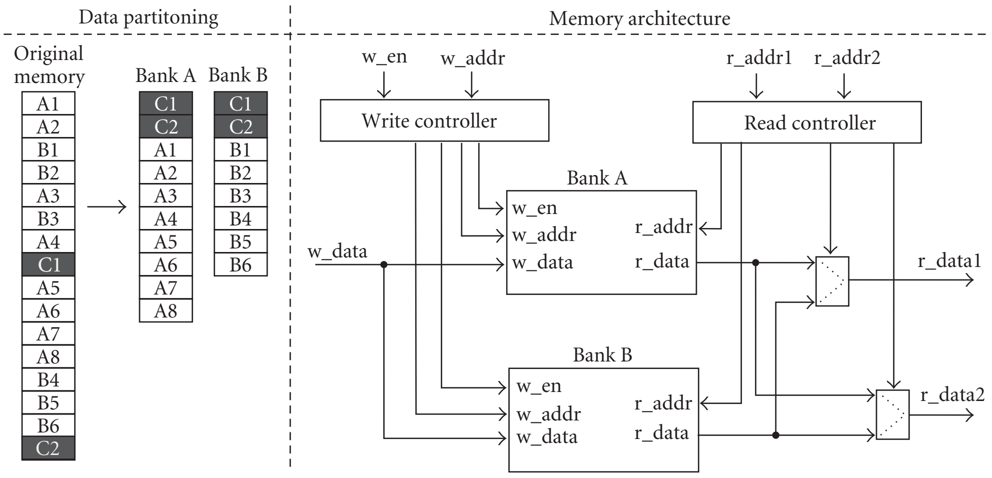


FIGURE 9: *Three*-port memory: data partitioning and architecture.

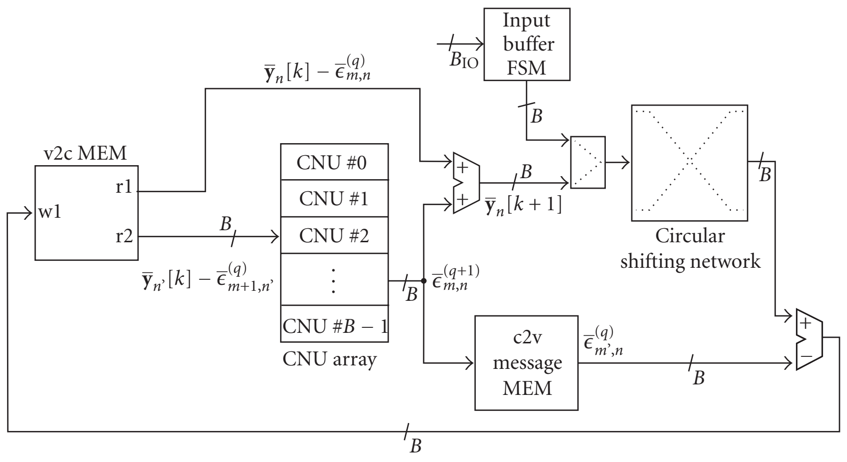

Input buffer FSM
B_{IO}
$\overline{\mathbf{y}}_n[k] - \overline{\epsilon}^{(q)}_{m,n}$
v2c MEM
w1
r1
r2
CNU #0
CNU #1
CNU #2
CNU #B − 1
CNU array
$\overline{\mathbf{y}}_{n'}[k] - \overline{\epsilon}^{(q)}_{m+1,n'}$
$\overline{\epsilon}^{(q+1)}_{m,n}$
$\overline{\mathbf{y}}_n[k+1]$
Circular shifting network
c2v message MEM
$\overline{\epsilon}^{(q)}_{m',n}$
B

FIGURE 10: Layered decoder with *v2c three*-port memory.

of memory redundancy at the same time could be pursued; however, due to the huge optimization space, this task is almost unfeasible and is not considered in this work.

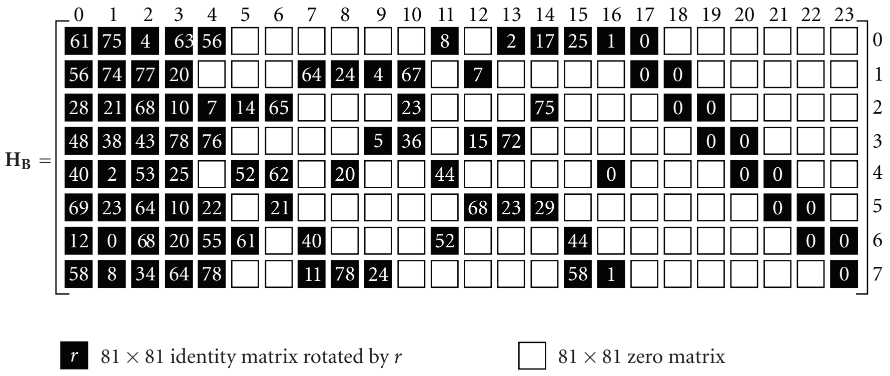
5.3. Storage of Variable-to-Check Messages. During the elaboration of a generic layer, a certain *v2c* message is needed twice, and a local buffer or multiple memory reading operations were implemented in Arch. V-A and Arch. V-B, respectively.

A third way of solving the problem is computing the array of *v2c* messages only once per iteration, like in Arch. V-A, but instead of using a local buffer, the *v2c* messages are precomputed and stored in the SO memory ready for the next use, as sketched in Figure 10. A similar architecture is used in [10, 16] but the issue of decoding pipeline is not clearly stated there.

In this way, the SO memory turns into a *v2c* memory with the following meaning: the array $\overline{\mathbf{y}}_n$ updated by layer m is stored in memory after marginalization with the *c2v* message $\overline{\epsilon}_{m',n}$, with m' being the index of the next layer reusing the same array of SOs, $\overline{\mathbf{y}}_n$. In other words, the array of *v2c* messages involved in the next update of the same block-column n is precomputed. Therefore, the data stored in the *v2c* memory are used twice, first to feed the array of CNUs, and then for the SOs update.

Similarly to Arch. V-B, a *three*-port memory would be required because of the decoding pipeline; the same considerations of Section 5.2 still hold, and an optimum partitioning of the *v2c* memory onto two banks with some redundancy can be found. Note that, as opposed to Arch. V-B, a customary dual-port memory is enough for *c2v* messages.

As far as the complexity is concerned, at first glance this solution seems to be preferable to Arch. V-B since it needs only two stages of parallel adders while the *c2v* memory is not split. However, the management of the reading ports of the *v2c* memory introduces significant overheads, since after the update of the soft outputs $\overline{\mathbf{y}}_n$ by layer m, the memory controller must be aware of what is the next layer m' using the same soft outputs $\overline{\mathbf{y}}_n$. This information needs to be stored in a dedicated configuration memory, whose size and area can be significant, especially in a multilength, multirate decoder.



$$\mathbf{H_B} = \begin{array}{|cccccccccccccccccccccccc|c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & \\
61 & 75 & 4 & 63 & 56 & & & & & & & 8 & & 2 & 17 & 25 & 1 & 0 & & & & & & & 0 \\
56 & 74 & 77 & 20 & & & & 64 & 24 & 4 & 67 & & 7 & & & & & 0 & 0 & & & & & & 1 \\
28 & 21 & 68 & 10 & 7 & 14 & 65 & & & & 23 & & & & 75 & & & & 0 & 0 & & & & & 2 \\
48 & 38 & 43 & 78 & 76 & & & & & 5 & 36 & & 15 & 72 & & & & & & 0 & 0 & & & & 3 \\
40 & 2 & 53 & 25 & & 52 & 62 & & 20 & & & 44 & & & & & 0 & & & & 0 & 0 & & & 4 \\
69 & 23 & 64 & 10 & 22 & & 21 & & & & & & 68 & 23 & 29 & & & & & & & 0 & 0 & & 5 \\
12 & 0 & 68 & 20 & 55 & 61 & & 40 & & & & 52 & & & & 44 & & & & & & & 0 & 0 & 6 \\
58 & 8 & 34 & 64 & 78 & & & 11 & 78 & 24 & & & & & & 58 & 1 & & & & & & & 0 & 7
\end{array}$$

■ r 81×81 identity matrix rotated by r □ 81×81 zero matrix

Figure 11: Parity-check base-matrix of the *block*-LDPC code for IEEE 802.11n with codeword size $N_2 = 1944$ and rate $r = 2/3$. Black squares correspond to cyclic shifts s of the identity matrix ($0 \leq s \leq B - 1$), also indicated in the square, while empty squares correspond to all-zero submatrices.

6. A Case Study: The IEEE 802.11n LDPC Codes

6.1. LDPC Code Construction. The WLAN standard [3] defines AA-LDPC codes based on *circulants* of the identity matrix. Three different codeword lengths are supported, $N_0 = 648$, $N_1 = 1296$, and $N_2 = 1944$, each coming with four code rates, 1/2, 2/3, 3/4, and 5/6, for a total of 12 different codes. As a distinguishing feature, a different block-size is used for each codeword length, that is, $B_0 = 27$, $B_1 = 54$, and $B_2 = 81$, respectively; accordingly, every code counts $n_c = N_i/B_i = 24$ *block*-columns, while the *block*-rows (layers) are in the number of $n_r = (1-r)n_c = 12, 8, 6, 4$ for code rates 1/2, 2/3, 3/4, and 5/6, respectively.

An example of the base-matrix $\mathbf{H_B}$ for the code with length $N_2 = 1944$ and rate $r = 2/3$ is shown in Figure 11.

6.2. Multiframe Decoder Architecture. In order to attain an adequate throughput for every WLAN codes, the decoder must include a number of CNUs at least equal to $\max\{B_i\} = 81$. This means that two thirds of the processors would remain unused with the shortest codes.

In the latter case, the throughput can be increased thanks to a *multiframe* approach, where $F_i = \lfloor \max\{B_i\}/B_i \rfloor$ frames of the code with block-size B_i are decoded in parallel. A similar solution is described in [12], but in that case two different frames are decoded in time-division multiplexing by exploiting the 2 nonoverlapped phases of the flooding algorithm. Here, F_i frames are decoded concurrently, and more specifically, three different frames of the shortest code can be assigned to a cluster of 27 CNUs each.

Note that to work properly, the circular shifting network must support concurrent subrotations as described in [24].

7. Decoder Performance

As to give a practical example of the reordering strategies described in Section 4, Figure 12 shows the data flow related to the update of layer 0 for the WLAN code of Figure 11. While 6 idle cycles are required following the original, natural order of updates (see Figure 12(a)), EOP needs 5 cycles (see Figure 12(b)), ROP reduces them to 1 (see Figure 12(c)), while no idle cycle is used by UOP (see Figure 12(d)). The subsets defined in Section 4.1 are also shown in Figure 5, along with the optimal sequence of layers followed for decoding.

7.1. Latency and Throughput. The latency of a pipelined LDPC decoder can be expressed as

$$T_{\text{dec}} = t_{\text{clk}} \cdot \left\{ N_{\text{it}} \cdot (N_{\text{B}} + \mathcal{I}_{\text{it}}) + L_{\text{pipe}} + 2_{\text{IO}} \right\} \qquad (16)$$

with $t_{\text{clk}} = 1/f_{\text{clk}}$ being the clock period, N_{it} being the number of iterations, N_{B} being the number of nonnull blocks in the code, $\mathcal{I}_{\text{it}} = \sum_{k=0}^{n_r-1} \mathcal{I}'_k$ being the number of idle cycles per iteration, L_{pipe} being the cycles to empty the decoder pipelin and finally, L_{IO} being the cycles for the input/output interface. Among the parameters above, N_{it} is set for good error-correction performance, N_{B} is a code-dependent parameter, and L_{IO} is fixed by the I/O management; thus, for a minimum latency, the designer can only act on $\mathcal{I}_{\text{it}}$, whose value can be optimised with the techniques of Section 4.

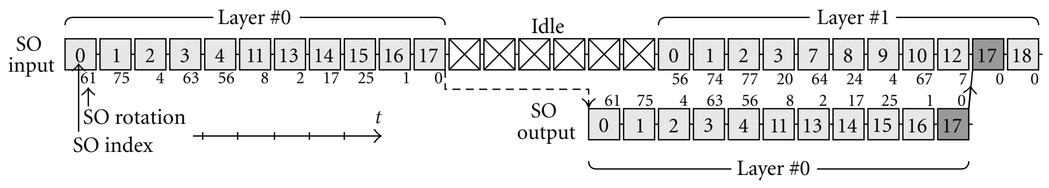
Focusing on the IEEE 802.11n codes, Table 1 shows the overall number of cycles for 12 iterations ($L_{\text{dec}} = T_{\text{dec}}/t_{\text{clk}}$), the number of idle cycles per iteration ($\mathcal{I}_{\text{it}}$), the percentage of idle cycles with respect to the total (idling %), and the throughput at the clock frequency of 240 MHz.

The latter is expressed in information bits decoded per time unit and is also referred to as net throughput:

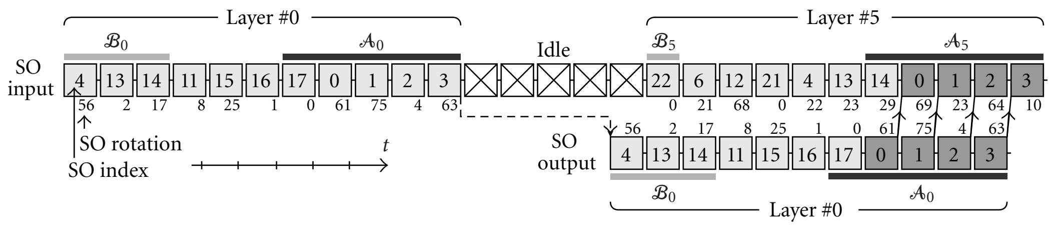
$$\Gamma_n = F_i \cdot \frac{r \cdot N_i}{T_{\text{dec}}}, \qquad (17)$$

where F_i is the number of frames decoded in parallel. For this reason, the figures of Table 1 for the short codes are very similar to those for the long codes ($N_0 F_0 = N_2 F_2$); on the contrary, the middle codes do not benefit from the same mechanism (i.e., $F_1 = 1$) and their throughput is scaled down by a factor 2/3.

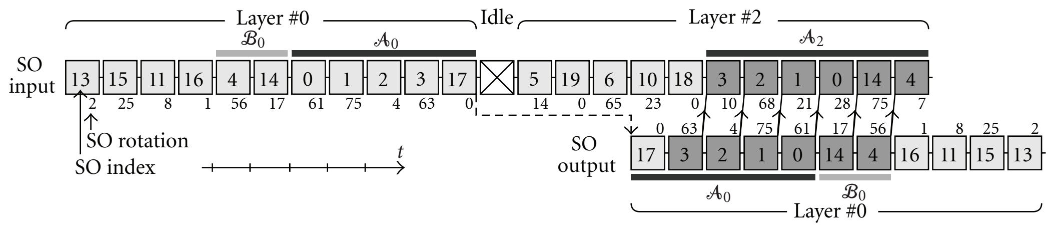
The results of Table 1 are for every technique of Section 4 as well as for the original codes before optimization. Although EOP clearly outperforms the original codes, better results are achieved with ROP and UOP for the WLAN case



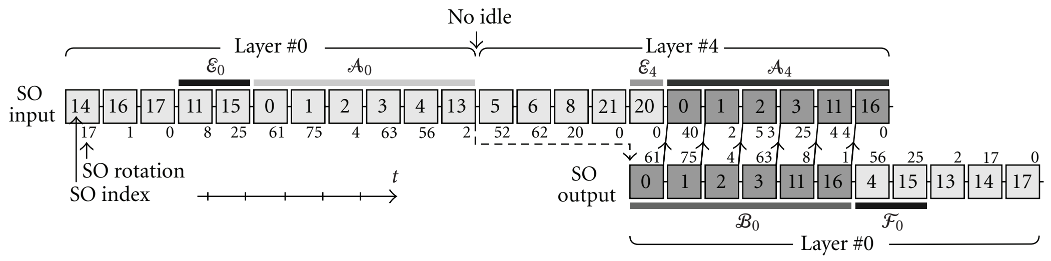
(a) Original base-matrix (sequence of layers: 0,1,2,3,4,5,6,7)



(b) EOP (optimised sequence of layers: 0,5,6,7,4,2,3,1)



(c) ROP (optimised sequence of layers: 0,2,7,5,6,3,4,1)



(d) UOP (optimised sequence of layers: 0,4,7,6,1,2,6,3)

Figure 12: An example of optimization of the base-matrix of the LDPC code IEEE 802.11n with $N_2 = 1944$ and $r = 2/3$ with EOP, ROP and UOP. Critical propagations are highlighted in dark gray.

example, where at most 14% and 11% of the decoding time are spent in idle, respectively. On average, the decoding time decreases from 7.6 to 6.7 ns with EOP and even to 5.3 ns with ROP and 5.1 ns with UOP. This behaviour can be explained by considering that for the WLAN codes the term $(d_k - \alpha_k - \beta_k) \cdot u(\chi_k)$ found in (6) for EOP is significantly nonnull, while comparing (8) to (13), ROP and UOP basically differ for the term $(\alpha_k - \chi_k) \cdot u(\beta_k)$, which is negligible for the WLAN codes.

7.2. Error-Correction Performance. Figure 13 compares the floating point frame error rate (FER) after 12 decoding iterations of a pipelined decoder using EOP, ROP, and UOP with a reference curve obtained by simulating the original parity-check matrix before optimization, in a nonpipelined decoder. Two simulations were run for each strategy, one with the proper number of idle cycles (curves with full markers), and the other without idle cycles and referred to as *full pipeline* mode (curves with empty markers).

As expected, the three strategies reach the reference curve of the HLD algorithm when properly idled. Then, in case of full pipeline ($\mathcal{I}_k = 0$, $\forall k$), the performance of EOP are spoiled, while ROP and UOP only pay about 0.6 and 0.3 dB, respectively. This means that the reordering has significantly reduced the dependence between layers and only few hazards arise without idle cycles.

Similarly to EOP, no received codeword is successfully decoded even at high SNRs (i.e., FER = 1) if the original code descriptors are simulated in full pipeline. This confirms

Table 1: Performance of an LDPC decoder for IEEE 802.11n with 12 iterations: $L_{SO} = 5$ and $f_{clk} = 240$ MHz.

Code lenght		$N_0 = 648$				$N_1 = 1296$				$N_2 = 1944$			
Code rate		1/2	2/3	3/4	5/6	1/2	2/3	3/4	5/6	1/2	2/3	3/4	5/6
	N_B	88	88	88	88	86	88	88	85	86	88	85	79
	L_{IO}	72	72	72	72	48	48	48	48	72	72	72	72
Original	L_{dec}	2299	1763	1779	1486	2106	1715	1886	1653	2107	1775	1752	1603
	$\mathcal{I}_{it}$	91	46	47	22	81	46	60	43	77	47	48	41
	idling %	47%	31%	31%	17%	46%	32%	38%	31%	44%	31%	32%	30%
	Γ_n (Mbps)	101	176	197	262	74	121	124	157	111	175	200	243
EOP	L_{dec}	1927	1691	1575	1462	1819	1643	1527	1377	1855	1691	1538	1352
	$\mathcal{I}_{it}$	60	40	30	20	57	40	30	20	56	40	30	20
	idling %	37%	28%	23%	16%	37%	29%	23%	17%	36%	28%	23%	17%
	Γ_n (Mbps)	121	184	222	266	85	126	153	188	126	184	228	288
ROP	L_{dec}	1308	1216	1290	1403	1223	1168	1239	1330	1283	1228	1243	1305
	$\mathcal{I}_{it}$	8	0	6	15	7	0	6	16	8	1	5	16
	idling %	7.3%	0%	5.5%	13%	6.8%	0%	5.5%	14%	7.4%	1%	4.8%	14%
	Γ_n (Mbps)	178	256	271	277	127	178	188	195	182	253	282	298
UOP	L_{dec}	1308	1216	1243	1380	1187	1168	1195	1260	1259	1216	1195	1164
	$\mathcal{I}_{it}$	8	0	2	13	4	0	2	10	6	0	1	4
	idling %	7.3%	0%	1.9%	11%	4%	0%	2%	9.3%	5.6%	0%	0.9%	4%
	Γ_n (Mbps)	178	256	282	282	131	178	195	206	185	256	293	334

once more the importance of idle cycles in a pipelined HLD decoding decoder and motivates the need of an optimization technique.

Considering the same scenario of Figure 13, Figure 14 shows the convergence speed, measured in average number of iterations, of the layered decoding algorithm. The curves confirm that HLD needs one half of the number of iterations of the flooding schedule, on average, and show that the full pipeline mode is also penalized in terms of speed.

8. Implementation Results

The complexity of an LDPC decoder for IEEE 802.11n codes was derived through logical synthesis on a low-power 65 nm CMOS technology targeting $f_{clk} = 240$ MHz. Every architecture of Section 5 was considered for implementation, each one supporting the three reordering strategies, for a total of 9 combinations. For good error correction performance, input LLRs and *c2v* messages were represented on 5 bits, while internal SO and *v2c* messages on 7 bits.

Table 2 summarizes the complexity of the different designs in terms of logic, measured in equivalent Kgates and number of RAM and ROM bits. Equivalent gates are counted by referring to the low-drive, 2-input NAND cell, whose area is 2.08 μm^2 for the target technology library. Arch. V-A needs the highest number of memory bits due to the local variable-to-check buffer, but its logic is smaller since it requires no additional hardware resources (adders) and less configuration bits.

Because of the partitioning of both the SO and the *c2v* memories, Arch. V-B needs more logic resources and more memory bits than Arch. V-C (both for data and configuration). The redundancy ratios ρ_{SO} and ρ_{c2v} of the SO and *c2v* memory in Arch. V-B, respectively, and ρ_{v2c} of the *v2c* memory in Arch. V-C, are also reported in Table 2.

As a matter of fact, the three architectures are very similar in complexity and performance, and, for a given set of LDPC codes, the designer can select the most suitable solution by trading-off decoding latency and throughput at the system level, with the requirements of logic and memory in terms of area, speed, and power consumption at the technology level.

Table 3 compares the design of a decoder for IEEE 802.11n based on Arch. V-C with UOP with similar state-of-the-art implementations: a parallel decoder by Blanskby and Howland [7], a 2048-bit rate 1/2 TDMP decoder by Mansour and Shanbhag [25], a similar design for WLAN by Gunnam et al. [10], and a decoder for WiMAX by Brack et al. [26]. Here, for a fair comparison, the throughput is expressed in channel bits decoded per time unit; that is, it is the *channel* throughput $\Gamma_c = N_i/T_{dec} = \Gamma_n/r$.

For the comparison, we focused on the architectural efficiency η_A defined as

$$\eta_A = \frac{T_{dec} \cdot f_{clk}}{N_{it} \cdot N_B} = \frac{N \cdot f_{clk}}{\Gamma_c \cdot N_{it} \cdot N_B}, \tag{18}$$

which represents the average number of clock cycles to update one block of $\mathbf{H}$. In decoders based on serial functional units it is $\eta_A \geq 1$, and the higher η_A is, the less efficient is the architecture. Actually, η_A can reach 1 only when the dependence between consecutive layers is solved at the code design level. This is the case of two WiMAX codes

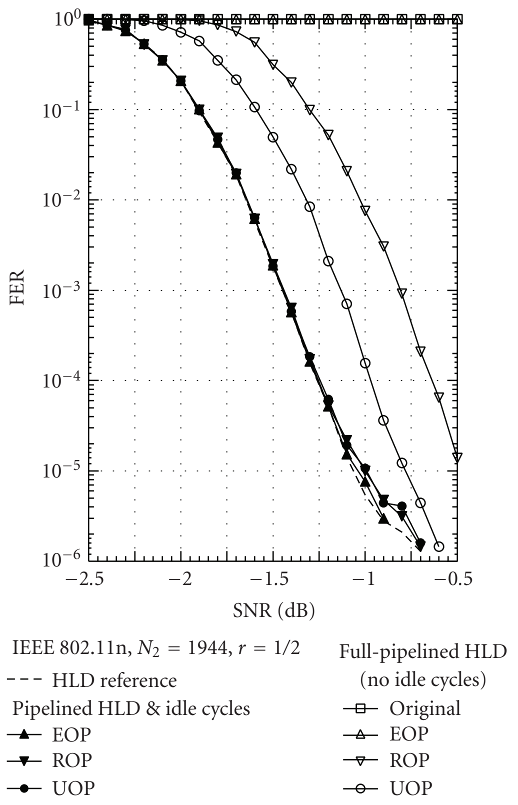


Figure 13: Error-correction performance of the IEEE 802.11n, $N_2 = 1944$, rate-1/2 LDPC code after 12 decoding iterations.

Table 2: IEEE 802.11n LDPC decoder complexity analysis.

		EOP	ROP	UOP
Arch. V-A	logic (Kgates)	71.29	71.62	74.65
	RAM bits	61,722	61,722	61,722
	ROM bits	23,159	23,159	40,788
Arch. V-B	logic (Kgates)	75.45	75.75	77.99
	RAM bits	53,622	54,837	57,024
	ρ_{SO}	29.2%	29.2%	33.3%
	ρ_{c2v}	1.1%	4.6%	9.1%
	ROM bits	36,582	36,582	51,849
Arch. V-C	logic (Kgates)	71.83	72.14	74.60
	RAM bits	53,217	53,217	53,784
	ρ_{v2c}	29.2%	29.2%	33.3%
	ROM bits	34,508	34,508	43,553

(specifically, class 1/2 and class 2/3B codes) which are hazard-free (or layered) "by construction", thus explaining the very low value of η_A achieved by [26]. However, [26] is as efficient as our design ($\eta_A \approx 1.3$) on the remaining nonlayered WiMAX codes, but the authors do not perform layered decoding on such codes.

For decoders with parallel processing units (see [7, 25]) the architectural efficiency becomes a measure of the parallelization used in the processing units and it can be expressed as $\eta_A \simeq 1/\overline{d}$ with $\overline{d}$ being the average check

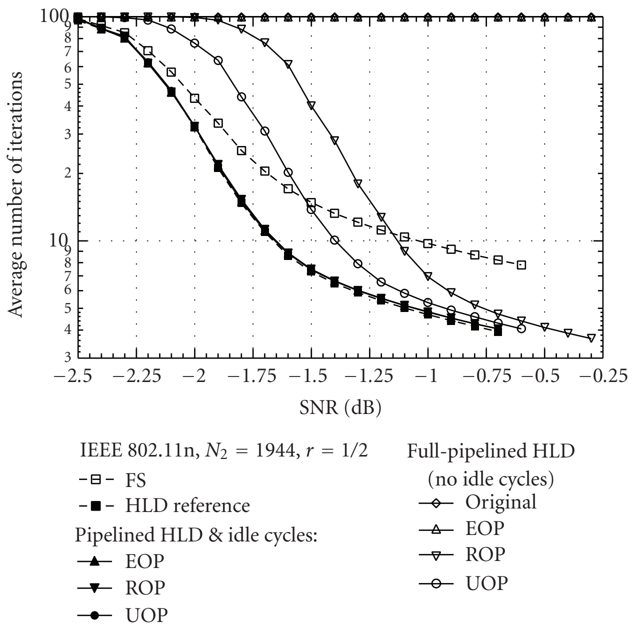


Figure 14: IEEE 802.11n, $N_2 = 1944$, rate-1/2 LDPC code: average decoding speed for a maximum of 100 iterations.

node degree. Indeed, in a two-phase decoder, the number of blocks can be equivalently defined as the overall number of exchanged messages, divided by the number of functional units. If E is the number of edges in the code, then $N_B = 2E/(N + rN)$, which is an index of the parallelization used in the processors.

The different designs were also compared in terms of energy efficiency, defined as the energy spent per coded bit and per decoding iteration. This is computed as

$$\eta_E = \frac{E_{\text{dec}}}{N \cdot N_{\text{it}}} = \frac{P}{\Gamma_c \cdot N_{\text{it}}} \tag{19}$$

with $E_{\text{dec}} = P \cdot T_{\text{dec}}$ being the decoding energy and P being the power consumption. The latter was estimated with Synopsys Power Compiler and was averaged out over three different SNRs (corresponding to different convergence speeds) and includes the power dissipated in the memory units (about 70% of the total). In terms of energy, our design is more efficient than [25] and gets close to the parallel decoder in [7].

Since the design in [10] is for the same WLAN LDPC codes and implements a similar layered decoding algorithm with the same number of processing units, a closer inspection is compulsory. Thanks to the idle optimization, our solution is more efficient in terms of throughput, the saving in efficiency ranging from 16% to 23%. Then, although our design saves about 70 mW in power consumption with respect to [10], the related energy efficiency has not been included in Table 2 since the reference scenario used to estimate the power consumption (238 mW) was not clearly defined. Finally, although curves for error correction performance are not available in [10], penalties are expected in view of the smaller accuracy used to represent $v2c$ (5 bits) and SOs (6 bits) messages.

Table 3: State-of-the-art LDPC decoder implementations.

		[this]	[7]	[10]	[25]	[26]
Technology		65 nm CMOS	0.16 μm CMOS 5-LM	0.13 μm TSMC CMOS	0.18 μm 1.8 V TSMC CMOS	0.13 μm CMOS
Algorithm		layered	flooding	layered	TDMP	flooding/layered
CPU arch.		serial	parallel	serial	parallel	serial
Nb. of CPUs		81	1536	81	64	96
Msg. width ($c2v$ + SO)		5 + 7	4 + 4	5 + 6	4 + 5	6
Clock fr (MHz)		240	64	500	125	333
Rates		1/2, 2/3, 3/4, 5/6	1/2	1/2, 2/3, 3/4, 5/6	1/2 : 1/16 : 7/8	1/2, 2/3, 3/4, 5/6
Codeword length, N		648, 1296, 1944	1024	648, 1296, 1944	2048	576 : 96 : 2304
Codeword size, B		27, 54, 81	1	27, 54, 81	64	24 : 4 : 96
Nb. of blocks, N_B		79–88	4,33	79–88	96	76–88
Speed	Iterations N_{it}	12	64	5	10	16
	Γ_c (Mbps)	262–401	1,024	541–1,618	640	177–999
Area	Kgates (mm^2)	100.7 (0.207)	1750 (52.5)	99.9 (1.85)	220 (14.3)	489.9 (2.964)
	RAM bits	56,376	—	55,344	51,680	NA
Power consumption (W)		0.162	0.69	0.238	0.787	NA
η_A (cycle/bit/iter)		1.103–1.306	0.231	1.361–1.521	0.417	1.01–1.31
η_E (pjoule/bit/iter)		33.7–51.5	10.5	—	123	—

9. Conclusions

An effective method to counteract the pipeline hazards typical of block-serial layered decoders of LDPC codes has been presented in this paper. This method is based on the rearrangement of the decoding elaborations in order to minimize the number of idle cycles inserted between updates and resulted in three different strategies named equal, reversed, and unconstrained output (EOP, ROP, and UOP) processing.

Then, different semi-parallel VLSI architectures of a layered decoder for architecture-aware LDPC codes supporting the methods above have been described and applied to the design of a decoder for IEEE 802.11n LDPC codes.

The synthesis of the proposed decoder on a 65 nm low-power CMOS technology reached the clock frequency of 240 MHz, which corresponds to a net throughput ranging from 131 to 334 Mbps with UOP and 12 decoding iterations, outperforming similar designs.

This work has proved that the layered decoding algorithm can be extended with no modifications nor approximations to every LDPC code, despite the interconnections on its parity-check matrix, provided that idle cycles are used to maintain the dependencies between the updates in the algorithm.

Also, the paradigm of code-decoder codesign has been reinforced in this work, since not only the described techniques have shown to be very effective to counteract the pipeline hazards but also they provide at the same time useful guidelines for the design of good, hazard-free, LDPC codes. To this extent, it is then overcome the assumption that consecutive layers do not have to share soft-outputs, like the WiMAX class 1/2 and 2/3B codes do, thus leaving more room to the optimization of the code performance at the level of the code design.

References

[1] "Satellite digital video broadcasting of second generation (DVB-S2)," ETSI Standard EN302307, February 2005.

[2] IEEE Computer Society, "Air Interface for Fixed and Mobile Broadband Wirelss Access Systems," IEEE Std 802.16e^{TM}-2005, February 2006.

[3] "IEEE P802.11n^{TM}/D1.06," Draft amendment to Standard for high throughput, 802.11 Working Group, November 2006.

[4] R. Gallager, *Low-density parity-check codes*, Ph.D. dissertation, Massachusetts Institutes of Technology, 1960.

[5] D. MacKay and R. Neal, "Good codes based on very sparse matrices," in *Proceedings of the 5th IMA Conference on Cryptography and Coding*, 1995.

[6] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 976–996, 2003.

[7] A. Blanksby and C. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 lowdensity parity-check code decoder," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, 2002.

[8] H. Zhong and T. Zhang, "Block-LDPC: a practical LDPC coding system design approach," *IEEE Transactions on Circuits and Systems I*, vol. 52, no. 4, pp. 766–775, 2005.

[9] D. E. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *Proceedings of the IEEE*

Workshop on Signal Processing Systems (SISP '04), pp. 107–112, 2004.

[10] K. Gunnam, G. Choi, W. Wang, and M. Yeary, "Multi-rate layered decoder architecture for block LDPC codes of the IEEE 802.11n wireless standard," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 1645–1648, May 2007.

[11] T. Bhatt, V. Sundaramurthy, V. Stolpman, and D. McCain, "Pipelined block-serial decoder architecture for structured LDPC codes," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '06)*, vol. 4, pp. 225–228, April 2006.

[12] C. P. Fewer, M. F. Flanagan, and A. D. Fagan, "A versatile variable rate LDPC codec architecture," *IEEE Transactions on Circuits and Systems I*, vol. 54, no. 10, pp. 2240–2251, 2007.

[13] E. Boutillon, J. Tousch, and F. Guilloud, "LDPC decoder, corresponding method, system and computer program," US patent no. 7,174,495 B2, February 2007.

[14] M. Rovini, F. Rossi, P. Ciao, N. L'Insalata, and L. Fanucci, "Layered decoding of non-layered LDPC codes," in *Proceedings of the 9th Euromicro Conference on Digital System Design (DSD '06)*, August-September 2006.

[15] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, 1981.

[16] H. Zhang, J. Zhu, H. Shi, and D. Wang, "Layered approx-regular LDPC: code construction and encoder/decoder design," *IEEE Transactions on Circuits and Systems I*, vol. 55, no. 2, pp. 572–585, 2008.

[17] R. Echard and S.-C. Chang, "The π-rotation low-density parity check codes," in *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM '01)*, pp. 980–984, November 2001.

[18] F. Guilloud, E. Boutillon, J. Tousch, and J.-L. Danger, "Generic description and synthesis of LDPC decoders," *IEEE Transactions on Communications*, vol. 55, no. 11, pp. 2084–2091, 2006.

[19] H. Xiao and A. H. Banihashemi, "Graph-based message-passing schedules for decoding LDPC codes," *IEEE Transactions on Communications*, vol. 52, no. 12, pp. 2098–2105, 2004.

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

[21] F. Zarkeshvari and A. Banihashemi, "On implementation of min-sum algorithm for decoding low-density parity-check (LDPC) codes," in *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM '02)*, vol. 2, pp. 1349–1353, November 2002.

[22] C. Jones, E. Valles, M. Smith, and J. Villasenor, "Approximate-MIN constraint node updating for LDPC code decoding," in *Proceedings of the IEEE Military Communications Conference (MILCOM '03)*, vol. 1, pp. 157–162, October 2003.

[23] M. Rovini, F. Rossi, N. L'Insalata, and L. Fanucci, "High-precision LDPC codes decoding at the lowest complexity," in *Proceedings of the 14th European Signal Processing Conference (EUSIPCO '06)*, September 2006.

[24] M. Rovini, G. Gentile, and L. Fanucci, "Multi-size circular shifting networks for decoders of structured LDPC codes," *Electronics Letters*, vol. 43, no. 17, pp. 938–940, 2007.

[25] M. M. Mansour and N. R. Shanbhag, "A 640-Mb/s 2048-bit programmable LDPC decoder chip," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 684–698, 2006.

[26] T. Brack, M. Alles, F. Kienle, and N. Wehn, "A synthesizable IP core for WiMax 802.16E LDPC code decoding," in *Proceedings of the 17th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '06)*, pp. 1–5, September 2006.