

Generic description and synthesis of LDPC decoders

Frédéric Guilloud, *member, IEEE*, Emmanuel Boutillon, *member, IEEE*,
 Jacky Tousch, *member, IEEE*, and Jean-Luc Danger, *member, IEEE*

Abstract—Through a rapid survey of the architecture of Low-Density Parity-Check (LDPC) decoders, this paper proposes a general framework to describe and compare LDPC decoder architectures. A set of parameters makes it possible to classify the scheduling of iterative decoders, memory organization and type of check node processors and variable node processors. Using the proposed framework, an efficient generic architecture for non-flooding schedules is also given.

I. INTRODUCTION

Low-density parity-check (LDPC) codes [1] perform very close to the Shannon limit. Therefore they are being proposed more and more frequently to standardization committees. The first one to choose LDPC codes as a standard was the update of the digital video satellite broadcasting named DVB-S.2. After some years' theoretical studies, LDPC codes are now entering the industrial range of interest.

The performance of LDPC codes can be obtained thanks to a probabilistic and iterative decoding of the received codewords. Many different LDPC decoders have been described in the literature, and many LDPC codes have been designed for a given LDPC decoder architecture, such as in [2], [3]. A brief review can be found in [4]. However, there is no classification of these different architectures. It is quite difficult to compare them and thus to design a specific decoder which fits certain given specifications. Moreover, to our knowledge, all the published decoders use the flooding scheduling for the belief propagation algorithm (BP), except for the decoder of Mansour [5] and Hocevar [6]. In fact, no architecture has been proposed so far for shuffle BP scheduling [7], [8]. Finally, the complexity of LDPC code decoders is quite difficult to evaluate. Only the complexity of check node processors has been studied so far [9], [10], but the complexity of the check node is only a part of the answer. We suggest also considering the complexity of the LDPC code itself.

The first part of the next section presents the notations that will be used throughout this paper. It also recalls briefly the bipartite graph representation which is very convenient when dealing with LDPC codes. In the second part of section II, the decoding algorithm for LDPC codes is briefly recalled, independently of scheduling.

In section III, three classical different schedules are presented: the flooding schedule and the two shuffle schedules (the horizontal one and the vertical one from [7], [8]). In section IV, an evaluation of the complexity for decoding LDPC codes is proposed. The purpose is to compare different LDPC codes and help the designer to evaluate the message-passing structure that is to be proposed to suit the required specifications.

In section V, a generic architecture for LDPC code decoders is proposed. It is specified by several parameters related either to the datapath or to the processing modes of variable and check nodes. We show that the combination of these parameters enables us to describe most of the published LDPC decoders, and hence to compare them. Based on this generic architecture, the synthesis of a new architecture for LDPC code decoders is proposed, implementing a fast converging decoding algorithm. Section VII summarizes the results and concludes this paper.

II. DECODING ALGORITHMS

The most popular LDPC decoding algorithm is the belief propagation (BP) algorithm, which is optimal if the graph of the code does not contain any cycles. Although the graph of LDPC codes does contain cycles, this algorithm is still used and is considered as a reference. Before describing this algorithm, we introduce the notations that will be used hereafter.

A. Notations, bipartite graphs

An LDPC code or a repeat-accumulate (RA) code of size N and rate R can be represented as a bipartite graph where the N bits are represented by N variable nodes v_n . Each variable node is connected to some of the M parity-check nodes, $M \geq (1 - R) \times N$. We denote by $\mathcal{M}(n)$ (resp. $\mathcal{N}(m)$) the set of all the parity check indices (resp. variable indices) that are connected to the variable v_n (resp. parity check c_m). We denote also by $\mathcal{M}(n) \setminus m$ (resp. $\mathcal{N}(m) \setminus n$) the set of the parity check (resp. variable) indices that are connected to the variable v_n (resp. parity check c_m) without the parity check c_m (resp. variable v_n). A cycle on the graph is defined as a closed path. Finally, we denote by $|\mathcal{A}|$ the cardinal of the set \mathcal{A} . Thus, the parity check c_m is connected to $|\mathcal{N}(m)|$ variables and the variable node v_n is connected to $|\mathcal{M}(n)|$ check nodes. The degree of a node is the number of edges connected to it; so the variable node v_n has degree $|\mathcal{M}(n)|$ and the check node c_m has degree $|\mathcal{N}(m)|$. An LDPC code is said to be regular if the degree d_v of its variable nodes and the degree d_c of its check nodes are constants.

This work was supported by the E.U., under project number SPRING IST 1999-12342

F. Guilloud is associate professor at GET/ENST Bretagne, France.
 E. Boutillon is professor at the Université de Bretagne Sud, France.
 J. Tousch is co-founder and CTO of TurboConcept SAS, France.
 J.-L. Danger is professor at GET/ENST Paris, France.

B. Decoding with the BP-algorithm

Let $E_{m,n}$ denote the message from check c_m to variable v_n . Similarly, let $T_{n,m}$ denote the message from variable v_n to check c_m . Each node of the graph (check node or variable node) is replaced by a processor whose input-output ports are the connections of the graph. The BP algorithm describes the behavior of each type of processor:

- a variable-processor V_n has to compute the output messages $T_{n,m}$ using the input messages $E_{m,n}$ according to:

$$T_{n,m} = I_n + E_n - E_{m,n} \quad (1)$$

where $E_n = \sum_{m \in \mathcal{M}(n)} E_{m,n}$ is the extrinsic information

of the variable v_n . The variable I_n associated with each variable v_n is called intrinsic information. $I_n = 2y_n/\sigma^2$ in the case of a BPSK modulation over an additive white Gaussian noise (AWGN) channel of variance σ^2 , where y_n is the observation of the n -th received symbol of the codeword.

- a check-processor C_m has to compute the output messages $E_{m,n}$ using the input messages $T_{n,m}$ according to the function F : $E_{m,n} = \underset{n' \in \mathcal{N}(m) \setminus n}{F}(T_{n',m})$ defined by [1]:

$$\begin{cases} |E_{m,n}| = f^{-1} \left(\sum_{n' \in \mathcal{N}(m) \setminus n} f(T_{n',m}) \right) \\ \text{sign}(E_{m,n}) = \prod_{n' \in \mathcal{N}(m) \setminus n} \text{sign}(T_{n',m}) \end{cases} \quad (2)$$

where $f(x) = -\ln(\tanh(|x|/2))$.

The processors can process independently: they sample their input messages and process their output messages. For a graph without cycles, the algorithm converges toward a unique solution whatever the sample times are.

III. DECODING SCHEDULES

When the graph of the code contains cycles, the order of the sample times between the processor will have an influence on the results of the message passing algorithm. The schedule denotes the given order of the sampling times between all the node processors. Not all the schedules yield the same results: the unavoidable existence of cycles in an LDPC code of finite length generates a correlation between outgoing and incoming messages and yields some self-information behavior that decreases the performance of the iterative decoding process [11], [12] and introduces *pseudo-codewords* [13]. The loss in performance increases as the length of the cycles becomes shorter. The smallest cycle length in the graph is called the girth. A probabilistic schedule that achieves good performance was first presented by Mao and Banihashemi [14]. It will not be addressed however in this paper, since we only consider schedules updating all the edges in one iteration.

A. Flooding schedules

The most popular schedule associated with the BP algorithm is the flooding schedule (FS). All the variable-processors sample their input at the same time and then all the check-processors sample their input at the same time. Once all the messages have been processed, one decoding iteration has been completed. Iterations are repeated as long as required. The flooding schedule is summed up in algorithm 1. It can be noted that in this scheduling, the computation of the extrinsic information $E_n^{(i)}$ of a given variable v_n is performed in one step of the algorithm (line number 7).

Algorithm 1 Flooding schedule (FS)

- 1: Initialization:
 - 2: $i = 0$, $E_m^{(0)} = 0$, $\forall m$, $\forall n \in \mathcal{N}(m)$, $I_n = 2y_n/\sigma^2$, $\forall n$
 - 3: **repeat**
 - 4: $i = i + 1$
 - 5: {Variable Node Update}
 - 6: **for all** indices of variable nodes v_n , $n \in \{1, \dots, N\}$ **do**
 - 7: $E_n^{(i)} = \sum_{m \in \mathcal{M}(n)} E_{m,n}^{(i-1)}$ {Extrinsic Information Computation}
 - 8: $T_n^{(i)} = I_n + E_n^{(i)}$ {Total Information Computation}
 - 9: **for all** indices of parity-check node c_m connected to variable node v_n ($m \in \mathcal{M}(n)$) **do**
 - 10: $T_{n,m}^{(i)} = T_n^{(i)} - E_{m,n}^{(i-1)}$
 - 11: **end for**
 - 12: **end for**
 - 13: {Check Node Update}
 - 14: **for all** indices of parity-check node c_m , $m \in \{1, \dots, M\}$ **do**
 - 15: **for all** indices of variable node v_n implied in parity-check c_m ($n \in \mathcal{N}(m)$) **do**
 - 16: $E_{m,n}^{(i)} = \underset{n' \in \mathcal{N}(m) \setminus n}{F}(T_{n',m}^{(i)})$
 - 17: **end for**
 - 18: **end for**
 - 19: **until** $i \leq i_{\max}$ or convergence to a codeword
 - 20: The decoded bits are estimated through $\text{sign}(T_n^{(i)})$
-

It is possible to have a different and equivalent representation of the FS schedule, closer to some hardware implementations, combining the operations processed in the check nodes and in the variable nodes. For example, the variable-node processing can be distributed during the whole iteration while the check-nodes are processed sequentially (singly or in groups of P). More precisely, the computation of the extrinsic information is not processed in a single step as indicated in algorithm 2 on line 10. In this case, memories are required in the variable-node processor to save the accumulation of the check-to-variable messages of both the current and the previous iteration ($E_n^{(i-1)}$ and $E_n^{(i)}$ respectively). This schedule will be denoted the parity-check flooding schedule (FS-P). It is also possible to process the variables sequentially, the

Algorithm 2 Flooding schedule over the parity checks (FS-P)

```

1: Initialization:
2:  $i = 0, E_{m,n}^{(0)} = 0, E_n^{(0)} = 0, I_n = 2y_n/\sigma^2, \forall (m, n \in \mathcal{N}(m))$ 
3: repeat
4:    $i = i + 1, E_n^{(i)} = 0 \forall n \in \{1, \dots, N\}$ 
5:   for all indices of check-nodes  $c_m, m \in \{1, \dots, M\}$ 
   do
6:     for all indices of variable-node  $v_n$  implied in
     check-nodes  $c_m$  {Partial Variable Node Update}
     do
7:        $T_n^{(i)} = I_n + E_n^{(i-1)}$  {Total Information}
8:        $T_{n,m}^{(i)} = T_n^{(i)} - E_{m,n}^{(i-1)}$ 
9:     end for
10:    for all indices of variable-node  $v_n$  implied in
    check-nodes  $c_m$  {Check Node Update} do
11:       $E_{m,n}^{(i)} = \mathop{\text{F}}_{n' \in \mathcal{N}(m) \setminus n} (T_{n',m}^{(i)})$ 
12:       $E_n^{(i)} = E_n^{(i)} + E_{m,n}^{(i)}$  {Extrinsic Information
      Accumulation}
13:    end for
14:  end for
15: until  $i \leq i_{\max}$  or convergence to a codeword
16: The sent bits can be estimated through  $\text{sign}(T_n^{(i)})$ 

```

processing of the check-processors being distributed. It is then denoted by the variable flooding schedule (FS-V).

B. Fast converging schedules

Two other schedules are used which will be called the horizontal and vertical shuffle schedules (HSS and VSS) [4], [7], [8].

The VSS was proposed independently by Kfir and Kanner [15], and Zhang and Fossorier [7], [8]. One of the main advantages of the VSS is that it enables the decoding convergence to speed up. In flooding-like schedules, we observe that the processing of the check-to-variable messages $E_{m,n}^{(i)}$ at iteration (i) is based upon the variable-to-check messages $T_{n,m}^{(i-1)}$ at the previous iteration $(i-1)$. However, certain values of $T_{n,m}^{(i)}$ could already be computed based on partial computation of $E_{m,n}^{(i)}$ and then be used instead of $T_{n,m}^{(i-1)}$ to compute the remaining messages $E_{m,n}^{(i)}$ [7], [8], hence the shuffling of the check node update and the variable node update. The VSS is described in algorithm 3.

The HSS is the converse of the VSS: the roles of check-nodes and variable-nodes are swapped. It is a turbo-decoding like schedule, where the component codes are the rows or groups of rows of the parity-check matrix. A complete historical view of this schedule class can be found in [6]. An LDPC HSS decoder was generalized by Boutillon *et. al.* [16]. This schedule also enables the decoding convergence to speed up.

Note that these schedules force the node processing

Algorithm 3 Vertical Shuffle Schedule (VSS)

```

1: Initialization:
2:  $i = 0, E_{m,n}^{(0)} = 0, T_{n,m}^{(0)} = I_n = 2y_n/\sigma^2, \forall (m, n)$ 
3: repeat
4:    $i = i + 1$ 
5:   for all indices of variable-nodes  $v_n, n \in \{1, \dots, N\}$ 
   do
6:     for all indices of check-nodes  $c_m$  connected to
     variable-node  $v_n$  do
7:       {Check Node Update}
8:        $E_{m,n}^{(i)} = \mathop{\text{F}}_{\substack{n_1 \in \mathcal{N}(m) \setminus n, n_1 < n \\ n_2 \in \mathcal{N}(m) \setminus n, n_2 > n}} (T_{n_1,m}^{(i)}, T_{n_2,m}^{(i-1)})$ 
9:        $E_n = \sum_{m \in \mathcal{M}(n)} E_{m,n}^{(i)}$ 
10:      {Variable Node Update}
11:       $T_n^{(i)} = I_n + E_n$ 
12:       $T_{n,m}^{(i)} = T_n^{(i)} - E_{m,n}^{(i)}$ 
13:    end for
14:  end for
15: until  $i \leq i_{\max}$  or convergence to a codeword
16: The sent bits can be estimated through  $\text{sign}(T_n^{(i)})$ 

```

to be serial. The use of parallelism ($P > 1$) leads to implementing the group shuffled approach [7].

IV. ANALYSIS OF COMPLEXITY

The decoding complexity of an LDPC code is directly linked to the number of messages to be processed per iteration, *i.e.* to the number of edges within the bipartite graph of the code, or equivalently, to the number of non-zero entries of the parity-check matrix (two messages per edge), whatever the scheduling is : the scheduling is in fact only a partitionning of the different edges to be processed.

A. Processing power

One decoding iteration involves processing all the $T_{n,m}$ and $E_{m,n}$ messages, related to the edge between the variable v_n and the parity-check c_m . Let \mathcal{E} denote the total number of edges inside the bipartite graph of the code. For a regular (d_v, d_c) LDPC code of length N , for example, the total number of edges is given by:

$$\mathcal{E} = d_v M = d_c N \quad (3)$$

Let P_c denote the required processing power to decode LDPC codes, defined as the number of edges to be processed per clock cycle (we assume the decoder is implemented on synchronous-logic hardware, using a single clock). We can derive P_c from the following parameters:

- the number K of information bits to be transmitted per codeword;
- the rate R of the code;
- the information throughput D required;

- the maximum number¹ of iterations i_{\max} ;
- the clock frequency f_{clk} .

We will assume hereafter that the matrix is full rank, that is $M = (1 - R)N$. The number of variables ([var]) to be processed at each clock cycle is the number of variables per second ($\frac{D}{R}$ [var/s]) multiplied by the duration of a cycle, which yields to: $\frac{D}{f_{\text{clk}}R}$ [var/cycle]. Moreover, there are $\mathcal{E} \times i_{\max}$ edges to be processed to decode the N bit length codeword, *i.e.*: $\frac{\mathcal{E}i_{\max}}{N}$ [edges/var]. So the number of edges to be processed per clock cycle is equal to:

$$P_c = \frac{D}{f_{\text{clk}}R} \text{ [var/cycle]} \times \frac{\mathcal{E}i_{\max}}{N} \text{ [edges/var]} \quad (4)$$

$$= \frac{\mathcal{E}i_{\max}D}{Kf_{\text{clk}}} \text{ [edges/cycle]} \quad (5)$$

P_c can also be expressed using the average variable node degree $\bar{d}_v = \mathcal{E}/N$ and the rate $R = K/N$ of the code:

$$P_c = \frac{\bar{d}_v i_{\max} D}{f_{\text{clk}} R} \quad (6)$$

B. Example

To illustrate the results of the theoretical study, let us consider a regular LDPC code of length N and rate $R = 1/2$, with parameters $(d_v, d_c) = (3, 6)$. Assume that this code is decoded with a binary throughput $D = 10$ Mbits/s by means of a decoder with a clock frequency of 100 MHz. What is the minimum number of edges to be processed by the architecture per clock cycle to achieve the throughput if a maximum of $i_{\max} = 20$ iterations is specified?

The code being regular, the number \mathcal{E} of edges is $\mathcal{E} = 3N$. Considering that $K = N/2$ (code rate is $1/2$), the numerical application of (5) yields:

$$P_c = \frac{\mathcal{E}i_{\max}D}{Kf_{\text{clk}}} = \frac{3N \times 20 \times 10.10^6}{100.10^6 \times N/2} = 12 \text{ [edges/cycle]}$$

Thus, at each clock cycle, an average of $12/3 = 4$ variable nodes and $12/6 = 2$ parity-check nodes have to be processed. The architecture of the decoder has to use a parallelism of at least 2 check-processors and 4 variable-processors to achieve the specifications.

V. GENERIC ARCHITECTURE OF LDPC CODE DECODERS

We propose now to define a generic architecture for an LDPC code decoder, associated with various parameters. The aim of this section is to define a parametric and generic architecture of LDPC decoders that embeds most of the existing published architectures. This architecture is based on node processor architectures, the position and type of memory, the algorithm schedules and the level of parallelism.

¹Our analysis assumes a fixed maximum number of iterations. The results can be easily generalized to a variable number of iterations (e.g. using the well known syndrome-based convergence test), replacing i_{\max} by the average number of iterations actually performed. See also [17].

A. Generic Node processors

1) *Data-paths*: The generic node processor is made up of d input/output ports (e_j, s_j) , $j \in \{1, \dots, d\}$. Internal memory banks allow input or output messages to be saved inside the node processor. The processing of an output port s_j is performed according to: $s_j = \bigotimes_{i \neq j} \{e_i\}$ where \bigotimes

is a generic associative operator. Many architectures can be implemented for this processor: for example, the trellis architecture [2], [18] on figure 1-(b) or the “total sum” architecture [19], [20] in figure 1-(a) with both parallel and serial implementation. The total-sum implementation involves computing first the “total sum” which is defined as: $s = \bigotimes_i \{e_i\}$. Then the j -th input is inverted so as

to compute s_j from s . This implementation is interesting when the degree of the processor is high, since a lot of common computations are grouped. But it is possible only when the operator can be reversed. These are detailed in [4]. Note that these different architectures can have either a parallel implementation or a serial implementation. Registers can also be added to pipe-line the processing and thus decrease the critical path length. The generic operator may be either the sum (\sum), or the star (\star). The star operator between two log-likelihood ratios is defined as the function F applied on them [21]:

$$e_i \star e_j = \frac{1 + \exp(e_i \times e_j)}{\exp(e_i) + \exp(e_j)}$$

2) *Update modes*: There are three steps to be handled inside a generic node processor: input message reading (or sampling), computation of the output messages and output of the outgoing messages. There are mainly two types of approaches that can be defined to manage these three steps differently for the d input/output ports of the processor: grouped update and spread update.

The grouped update involves computing the output messages if and only if all the input messages have been sampled. So a typical scheme is first to wait for all the inputs to be updated and to save them. Then all the output messages are to be computed, and finally to be output. Then a new cycle can start again by waiting for all the inputs to be updated in the memory, erasing the previous ones.

The spread update is a kind of *on-demand* control. It means that, for example, the node processor can be asked for a given output message, and then it can be asked to take into account a new input message. Two spread update modes can be defined, depending on the memory inside the node processor: the *straight* one and the *delayed* one. When a new input message on a given edge has to be taken into account, either the previous message related to this edge is erased and replaced by this new one (*straight update*), or there are two memories (*delayed update*) denoted input-memory and compute-memory: the new message is saved in the input-memory while the previous one has been saved in the compute-memory. When the input-memory is full then the roles of the two memories are swapped. In the

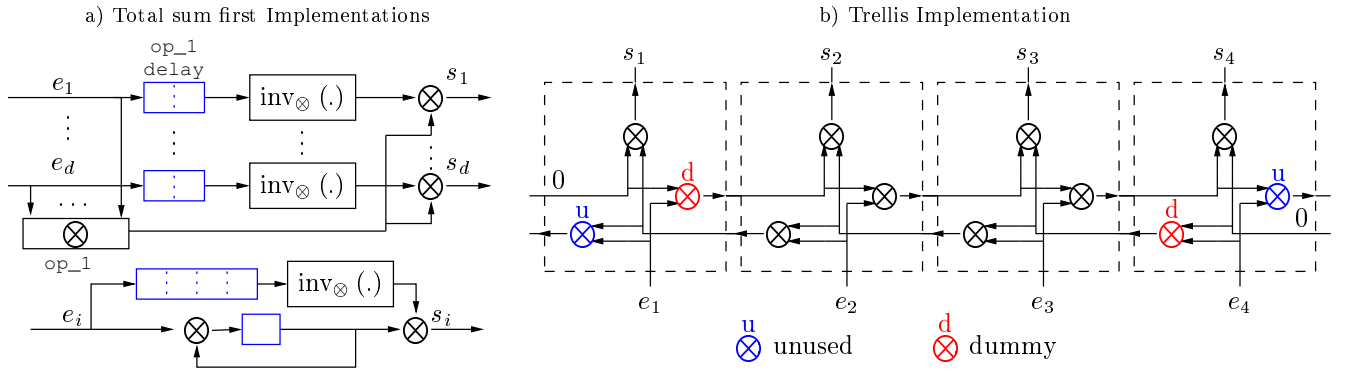


Fig. 1. Generic Node Processor: implementation for “total sum” and “trellis” schemes

TABLE I

THE SCHEDULES ASSOCIATED WITH THE DIFFERENT COMBINATIONS OF UPDATES FOR VARIABLE AND CHECK NODE PROCESSORS.

		Variable		
		Grouped	spread	
			delayed	straight
Parity-check	Grouped	FS	FS-V	VSS
	spread	delayed	FS-P	edge controlled
	straight	HSS		

straight update case, the same memory is used to save the new incoming inputs and to compute the output messages. Hence, the last output message will be processed with the most recent input messages. The *straight spread* update is implemented to speed up the propagation of the messages as in shuffle schedules.

To summarize, there are three ways of handling the processing steps in a generic node processor: the straight spread update, the delayed spread update and the grouped update. When combining these three handling on the check and the variable node processors, it is possible to span all the known schedules of LDPC decoding. These are summed up in table I.

B. Message Passing Architecture

The generic processors are instantiated so as to create the global architecture of the decoder, as illustrated in figure 2. The variable-processors V_i and the check-processors C_i are instances of the generic node processor with different values of parameters. P check-processors with d'_c input/output ports are instantiated. So each of them is able to process a parity-check of degree d_c within $\alpha = d_c/d'_c$ clock cycles.

The interconnection network represented in the bipartite graph of an LDPC code is materialized through a shuffle or a routing network π and its inverse π^{-1} . The complexity of the interconnection network depends on the structure of the parity-check matrix. From an implementation point of view, it seems desirable to have simple

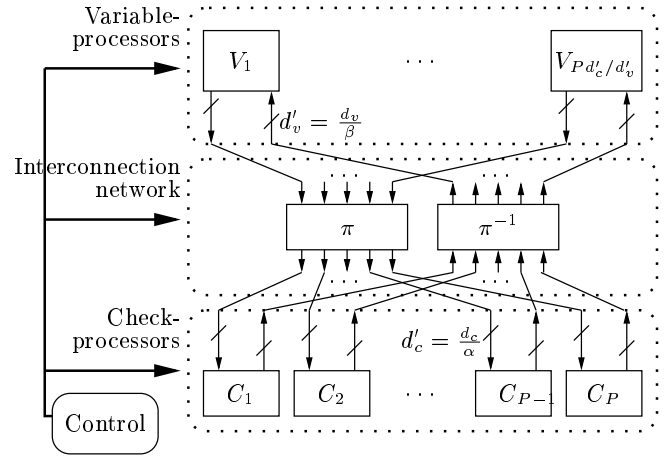


Fig. 2. Generic Message Passing Architecture

interconnections such as a barrel shifter, like in [6], [22]. Depending on specific hardware constraints, this network can be implemented on different alternative locations, as illustrated in figure 3: depending on the check node generic operator (star or sum operator), up to four different locations are presented (dashed lines) separating the variable node processors from the check node processors. With the use of the star operator (the input and the output of the parity-check node processors have the dimension of LLR's), there is only one possible location. But with the sum operator, there are four different locations, whether the inputs and the outputs of the node processors are in the Fourier domain or not.

The optimal number of variable-processors having d'_v input/output ports is given by the fact that the interconnection network has a determinate number of inputs and outputs. Each of the variable-processors is able to process a parity-check of degree d_v within $\beta = d_v/d'_v$ clock cycles.

Such a generic message passing architecture has a complexity P_c as defined by equation (5) which is equal to:

$$P_c = P \times d'_c = P \times d_c/\alpha \quad (7)$$

Thus, if we take the example of section IV-B where $P_c = 12$, a solution with $P = 2$ check-processors is possible

if the check-processors are able to process at least $d'_c = P_c/P = 6$ inputs/outputs per clock cycle ($\alpha = 1$), *i.e.* if $P = 2$ parallel generic processors are used. If serial generic processors are to be used ($\alpha = d_c = 6$), then $P = 12$ check-processors would have to be instantiated to achieve the specifications.

VI. ARCHITECTURE ANALYSIS AND SYNTHESIS FOR LDPC CODE DECODERS

A generic architecture for LDPC decoders was proposed in section V. In this section, some examples from the literature are taken to illustrate the versatility of the generic architecture. Then, we show that shuffling the parameters of the generic architecture can even yield a new architecture for the VSS schedule.

A. Architecture Analysis

The parameters specifying the architecture of LDPC code decoders have been defined in the previous sections. They are listed below:

- Node processors:
 - 3 possible architectures (direct, trellis, total sum)
 - 4 possible locations for the interconnection network
 - 3 possible update modes (grouped, straight or delayed spread)
- Message passing architecture:
 - 3 parameters for the parallelism specification (P , α , β)

Some combinations of values for these parameters have already been used or implemented in LDPC decoders. Some other combinations are new and yield interesting new implementations.

Due to the limited number of pages, we cannot present a classification of all the published decoder architectures, so we present only three of them hereafter: they fairly illustrate the diversity of the published architectures. Note

that the parameters we propose for the description of the following architectures are summed up in the first three columns of table II.

The first one is from Zhang and Parhi [23]. It is a decoder implemented on an FPGA Xilinx Virtex 2600 having a throughput of 54 Mbit/s, for a regular code of length $N = 9216$ bits and of rate $R = 0.5$. We can note that the node processors are completely parallel. The scheduling is the flooding scheduling (FS), and the locations of the interconnection network is such that look up tables are required in both the check nodes and the variable nodes (location number 2).

The second example is an architecture proposed by Chen and Hocevar [24] implemented on an ASIC 0.11μ and having a throughput of 376 Mbit/s. The length of the code is $N = 8088$, the rate is also $R = 0.5$, but the code is irregular. In this implementation, the check node processors are parallel whereas the variable node processors are serial. The location of the interconnection network is classical (the look up tables are in the check node processors). The scheduling is the flooding one over the variables (FS-V).

The third example is an architecture proposed by Mansour [25] implemented on an ASIC 0.18μ and having a throughput of 192 Mbit/s. The length of the code is $N = 2304$, the rate is $R = 2/3$, and the code is also irregular. In this implementation, the check node processors and the variable node processors are both serial. The location of the interconnection network is classical (the look up tables are in the check node processors). The scheduling is the shuffled one over the parity checks (HSS).

B. Architecture synthesis

1) *Example of a new architecture:* The architecture family generated by our framework also encompasses new efficient architectures. An example of an application of this formalism is given in the last column of table II. The algorithm performed by this architecture is exactly the Vertical Shuffle Schedule (VSS), as specified in table I. To our best knowledge, it is the first published architecture implementing this algorithm. It is depicted in figure 4, where only the magnitude processing is illustrated. Algorithm 4 is applied: it is a hardware-oriented description of algorithm 3 for the magnitude part only. The messages have changed: the variable-to-check message magnitude are now denoted $Q_{n,m} = f(T_{n,m})$, and the check-to-variable message magnitudes are denoted $|E_{m,n}| = f^{-1}(R_{m,n})$. They are function of the previous $E_{m,n}$ and $T_{n,m}$ messages through the f function. The $Q_{n,m}$ messages are saved in the memory either on the variable side (dotted lines in the figure) or on the parity-check side. They are read from this memory and are subtracted from the R_m value to obtain the $R_{m,n}$ message. The straight spread update is performed on the check-processor side: the values R_m relating to the parity-checks c_m are saved in memory. If we assume that $\mathcal{N}(m) = \{n_1, \dots, n_{d_c}\}$, then $R_m^{(i)}$ is updated d_c times during a decoding iteration. If we denote

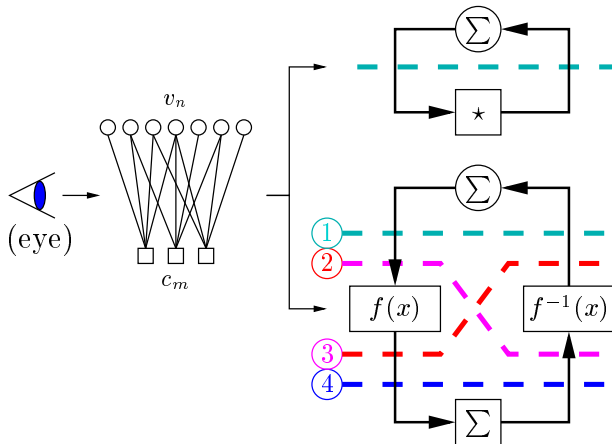


Fig. 3. The possible locations of the interconnection network. The datapath for the sign (\cdot) processing of the messages is omitted in this figure.

TABLE II
EXAMPLES OF THE PARAMETER VALUES FOR SEVERAL PUBLISHED IMPLEMENTATIONS OF LDPC DECODERS AND FOR A NEW ONE.

Reference		Zhang and Parhi [23]	Chen and Hocevar [24]	Mansour and Shanbhag [25]	New Implementation
Message passing architecture	α	1	1	1	d_c
	β	1	d_v	1	d_v
	P	18	24	64	P_c
Interconnection Network Location		2	1	1	4
Variable-Processor architecture	update	compact	delayed spread	straight spread	grouped
	architecture	trellis	total sum	total sum	total sum
Parity-check processor architecture	update	grouped	grouped	grouped	straight delayed
	architecture	trellis	total sum	trellis	total sum

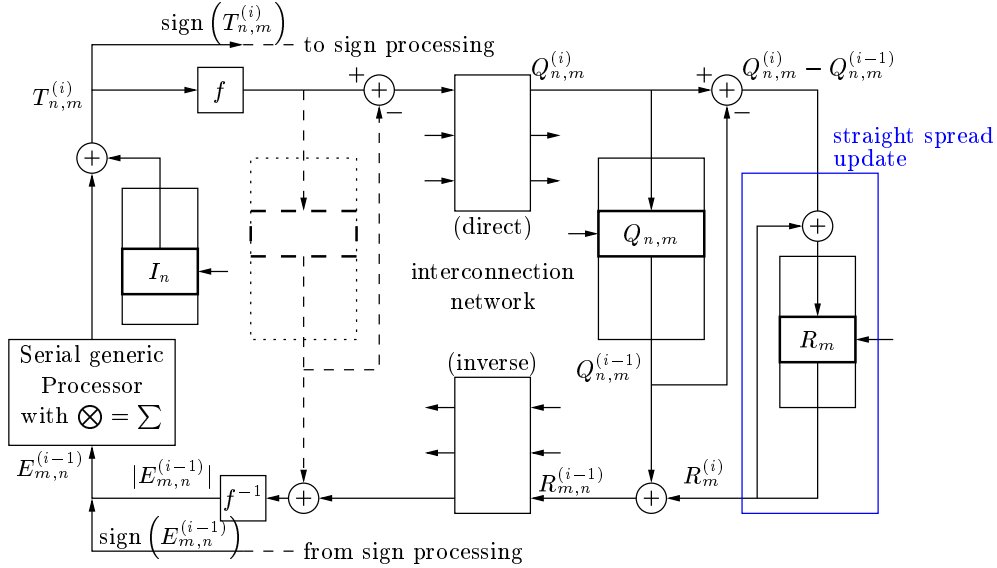


Fig. 4. New architecture proposal associated with the Vertical Shuffle Schedule of the BP algorithm (serial implementation for both check and variable node processors, magnitude processing on the check side).

the J^{th} update of $R_m^{(i)}$ by $R_m^{(i,J)}$ then we have:

$$R_m^{(i,J)} = \sum_{j=1}^J Q_{n_j,m}^{(i)} + \sum_{j=J+1}^{d_c} Q_{n_j,m}^{(i-1)} \quad (8)$$

$$= \sum_{j=1}^J f(T_{n_j,m}^{(i)}) + \sum_{j=J+1}^{d_c} f(T_{n_j,m}^{(i-1)}) \quad (9)$$

where J is a positive integer varying from 1 to d_c during the iteration. The $R_{m,n}$ messages are fed to the variable node processor where the $T_{m,n}$ values are processed and transformed into $Q_{n,m}$ messages. These new $Q_{n,m}$ messages replace in memory the message of the previous iteration. Also, the R_m value is updated by subtracting the old $Q_{n,m}$ and adding the new one (see algorithm 4, line 13). This architecture requires saving $(N + M + \mathcal{E})$ values for the I_n , R_m and $Q_{n,m}$ data respectively. Note that the architecture associated with the HSS requires only saving $(N + \mathcal{E})$ values for the $I_n + E_n$ and $E_{m,n}$ data.

2) *Comparison of memory requirements for the different schedules (table III):* We assume that an LDPC code of length N has an average variable degree of $d_{v\text{average}} = 3$. Then $\mathcal{E} = d_{v\text{average}}N = 3N$. We also assume that all the messages are coded using the same fixed-point format on w bits. Finally, we omit the input/output buffers which

TABLE III
MEMORY REQUIREMENTS FOR THE 3 DIFFERENT SCHEDULES

Schedule	Memory size	Numerical Application		
		$R = 1/3$	$R = 1/2$	$R = 9/10$
FS-P	$\mathcal{E}w + 3Nw$	$6Nw$		
HSS	$\mathcal{E}w + Nw$	$4Nw$		
VSS	$\mathcal{E}w + (2 - R)Nw$	$4.67Nw$	$4.5Nw$	$4.1Nw$

would add two memories of Nw bits each. The HSS has the lowest required memory size. The VSS memory size is a function of the code rate: for high code rates, it is possible to use almost the same amount of memory as for the HSS. It is to be noted that the FS-P has a higher memory requirement than the two shuffle schedules HSS and VSS. However, the main part of the memory is used to save the edge messages. This issue can be addressed using sub-optimal algorithms such as the (scaled or offset) BP-Based algorithm [10], [26], the λ -min algorithm [21] or the A -min* algorithm [27].

VII. CONCLUSION

A lot of LDPC code decoders have been proposed in the literature. However, it is difficult to compare them.

Algorithm 4 Vertical Shuffle Schedule (VSS): hardware-oriented description (Magnitude part only)

```

1: Initialization:
2:  $i = 0$ ,  $Q_{n,m}^{(0)} = f(I_n) = f(2y_n/\sigma^2)$ ,  $R_m^{(0)} = \sum_{n \in \mathcal{N}(m)} Q_{n,m}^{(0)}$ ,  $\forall (m, n)$ 
3: repeat
4:    $i = i + 1$ 
5:    $R_m^{(i)} = R_m^{(i-1)}$ 
6:   for all indices of variable-nodes  $v_n$ ,  $n \in \{1, \dots, N\}$  do
7:     for all indices of check-nodes  $c_m$  connected to variable-node  $v_n$  do
8:        $R_{m,n}^{(i-1)} = R_m^{(i-1)} - Q_{n,m}^{(i-1)}$ 
9:        $|E_{m,n}^{(i-1)}| = f^{-1} \left( R_{m,n}^{(i-1)} \right)$ 
10:       $T_n^{(i)} = I_n + \sum_{m \in \mathcal{M}(n)} E_{m,n}^{(i-1)}$ 
11:       $T_{n,m}^{(i)} = T_n^{(i)} - E_{m,n}^{(i-1)}$ 
12:       $Q_{n,m}^{(i)} = f \left( T_{n,m}^{(i)} \right)$ 
13:       $R_m^{(i)} = R_m^{(i-1)} - Q_{n,m}^{(i-1)} + Q_{n,m}^{(i)}$ 
14:    end for
15:  end for
16: until  $i \leq i_{\max}$  or convergence to a codeword
17: The sent bits can be estimated through  $\text{sign} \left( T_n^{(i)} \right)$ 

```

We have proposed a global framework for the description, analysis and synthesis of low-density parity-check (LDPC) code decoders. It is based on a generic model of a decoder described by parameters, related to a generic node processor and to a generic message passing architecture. A quantification of the complexity required by the decoding of an LDPC code has also been proposed. This framework makes it possible to describe several published LDPC decoder implementations using the same model and parameters. It also makes it possible to ensure a good match between algorithm and scheduling on the one hand, and decoder architectures on the other hand, as illustrated by the new efficient architecture proposed in this paper.

ACKNOWLEDGMENT

The authors would like to thank Prof. David Declercq for interesting comments on an earlier draft of this paper, and the anonymous reviewers for their interesting comments. Many thanks also to Janet Ormrod for her contributions.

REFERENCES

- [1] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, pp. 21–28, Jan. 1962.
- [2] E. Boutillon, J. Castura, and F. Kschischang, "Decoder-first code design," in *Proc. 2nd Int. Symp. on Turbo Codes & Related Topics*, Brest, France, Sept. 2000, pp. 459–462.
- [3] F. Verdier and D. Declercq, "A LDPC parity check matrix construction for parallel hardware decoding," in *Proc. 3rd Int. Symp. on Turbo Codes & related topics*, Sept. 1-5, 2003.
- [4] F. Guilloud, "Generic architectures for LDPC codes decoding," Ph.D. dissertation, Telecom Paris, Jul. 2004.

- [5] M. Mansour and N. Shanbhag, "Turbo decoder architectures for low-density parity-check codes," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Nov 17 - 21, 2002.
- [6] D. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *IEEE Workshop on Signal Processing Systems, SIPS 2004*, 13-15 Oct. 2004, pp. 107–112.
- [7] J. Zhang and M. Fossorier, "Shuffled belief propagation decoding," in *Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems and Computers, 2002.*, 3-6 Nov. 2002.
- [8] —, "Shuffled iterative decoding," *IEEE Trans. Commun.*, vol. 53, pp. 209–213, Feb. 2005.
- [9] T. Theodoridis, G. Link, E. Swankoski, N. Vijaykrishnan, M. Irwin, and H. Schmit, "Evaluating alternative implementations for LDPC decoder check node function," in *IEEE Computer Society Annual Symposium on VLSI*, 19-20 Feb. 2004, pp. 77–82.
- [10] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.-Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Commun.*, vol. 53, no. 8, pp. 1288–1298, Aug. 2005.
- [11] D. MacKay and R. Neal, "Good codes based on very sparse matrices," in *5th IMA Conference on Cryptography and Coding*. Berlin, Germany: Springer, 1995.
- [12] M. C. Davey and D. J. C. MacKay, "Low density parity check codes over GF(q)," *IEEE Commun. Lett.*, vol. 2, 1998.
- [13] N. Wiberg, "Codes and decoding on general graphs," Ph.D. dissertation, Linköping University, Sweden, 1996.
- [14] Y. Mao and A. Banihashemi, "Decoding low-density parity-check codes with probabilistic scheduling," *IEEE Commun. Lett.*, vol. 5, pp. 414–416, October 2001.
- [15] H. Kfir and I. Kanter, "Parallel versus sequential updating for belief propagation decoding," *Physica A*, vol. 330, pp. 259–270, Dec. 2003.
- [16] E. Boutillon, J. Tusch, and F. Guilloud, *LDPC decoder, corresponding method, system and computer program*, Dec. 2003, US Patent pending.
- [17] A. Martinez and M. Rovini, "Iterative decoders based on statistical multiplexing," in *Proc. 3rd Int. Symp. on Turbo Codes & related topics*, Sept. 1-5, 2003.
- [18] M. Mansour and N. Shanbhag, "Low-power VLSI decoder architectures for LDPC codes," in *Int. Symp. Low Power Electronic Design*, Aug. 12-14, 2002.
- [19] A. Blanksby and C. Howland, "A 690-mw 1-gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *Journal of Solid-State Circuits*, vol. 37, pp. 404–412, Mar. 2002.
- [20] E. Yeo, B. Nikolić, and V. Anantharam, "High throughput low-density parity-check decoder architectures," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Nov. 25-29, 2001.
- [21] F. Guilloud, E. Boutillon, and J.-L. Danger, "λ-min decoding algorithm of regular and irregular LDPC codes," in *Proc. 3rd Int. Symp. on Turbo Codes & related topics*, Sept. 1-5, 2003.
- [22] E. T. S. I. (ETSI), "Digital video broadcasting (dvb); second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satellite applications," Mar. 2005.
- [23] T. Zhang and K. K. Parhi, "An FPGA implementation of (3,6)-regular low-density parity-check code decoder," *EURASIP Journal on Applied Signal Processing, special issue on Rapid Prototyping of DSP Systems*, vol. 2003, no. 6, pp. 530–542, May 2003.
- [24] Y. Chen and D. Hocevar, "An FPGA and ASIC implementation of rate 1/2, 8088-b irregular low density parity check decoder," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, 1-5 Dec. 2003.
- [25] M. Mansour and N. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. VLSI Syst.*, vol. 11, pp. 976 – 996, Dec. 2003.
- [26] M. Fossorier, M. Mihaljević, and I. Imai, "Reduced complexity iterative decoding of low-density parity-check codes based on belief propagation," *IEEE Trans. Commun.*, vol. 47, pp. 673–680, May 1999.
- [27] C. Jones, E. Vallés, M. Smith, and J. Villasenor, "Approximate-min* constraint node updating for LDPC code decoding," in *Proc. IEEE Military Commun. Conf. (MILCOM)*, 13-16 Oct. 2003.