Iterative Decoding of Concatenated Convolutional Codes: Implementation Issues

Emmanuel Boutillon[1], Catherine Douillard[2], and Guido Montorsi[3]

Abstract

This tutorial paper gives an overview of the implementation aspects related to turbo decoders, where the term turbo generally refers to iterative decoders intended for Parallel Concatenated Convolutional Codes as well as for Serial Concatenated Convolutional Codes. We start by considering the general structure of iterative decoders, and the main features of the SISO algorithm that forms the heart of iterative decoders. Then, we show that very efficient parallel architectures are available for all types of turbo decoders allowing high speed implementations. Other implementation aspects like quantization issues and stopping rules used in conjunction with buffering for increasing throughput are considered. Finally, we perform an evaluation of the complexities of the turbo decoders as a function of the main parameters of the code.

[1] LESTER, Université de Bretagne Sud, Centre de recherche, BP 92116, 56321 Lorient Cedex, France, Phone: +33-297-874566, Fax: +33-297-874527, e-mail: emmanuel.boutillon@univ-ubs.fr

[2] Département Electronique, ENST Bretagne, Technopôle Brest-Iroise, CS 83818, 29238 Brest Cedex 3, France, Phone: +33-229-001283, Fax: +33-229-001184, e-mail: catherine.douillard@enst-bretagne.fr

[3] Dipartimento di Elettronica, Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Torino, Italy, Phone: +39-011-5644144, Fax: +39-011-5644099, e-mail: guido.montorsi@polito.it

I. INTRODUCTION

In 1993, at a moment when there were not many people to believe in the practicability of capacity approaching codes, the presentation of *turbo codes* [1] was a revival for the channel coding research community. Furthermore, the performance claimed in this seminal paper was soon confirmed with a practical hardware implementation [2].

Historical turbo codes, also sometimes called Parallel Concatenated Convolutional Codes (PCCC), are based on a parallel concatenation of two recursive systematic convolutional codes separated by an interleaver. They are called *turbo* in reference to the analogy of their decoding principle with the turbo principle of a turbo-compressed engine, which re-uses the exhaust gas in order to improve efficiency. The turbo decoding principle calls for an iterative algorithm involving two component decoders exchanging information in order to improve the error correction performance with the decoding iterations.

This iterative decoding principle was soon applied to other concatenations of codes separated by interleavers, such as Serial Concatenated Convolutional Codes (SCCC) [3][4], sometimes called *serial turbo codes*, or concatenation of block codes, also named *block turbo codes* [5][6].

The near-capacity performance of turbo codes and their suitability for practical implementation explain their adoption in various communication standards as early as the late 1990s: firstly, they were chosen in the telemetry coding standard by the CCSDS (Consultative Committee for Space Data Systems) [7], and for the medium to high data rate transmissions in the third generation mobile communication 3GPP/UMTS standard [8]. They have further been adopted as part of the Digital Video Broadcast - Return Channel Satellite and Terrestrial (DVB-RCS and DVB-RCT) links [9][10], thus enabling broadband interactive satellite and terrestrial services. More recently, they were also selected for the next generation of 3GPP2/cdma2000 wireless communication systems [11] as well as for the IEEE 802.16 standard (WiMAX) [12] intended for broadband connections over long distances. Turbo coding FEC are used in several Inmarsat's communication systems, too, such as in the new Broadband Global Area Network (BGAN [13]) that entered service in 2006. A serial turbo code was also adopted in 2003 by the European Space Agency for the implementation of a very high speed (1Gbps) adaptive coded modulation modem for satellite applications [14].

This paper deals with the implementation issues of iterative decoders for concatenated convolutional codes. Both parallel and serial concatenated convolutional codes are addressed and corresponding iterative decoders are equally referfed as turbo decoders. Due to the trend of the most recent applications towards an increasing demand for high-throughput data, special attention is paid to the design of high speed hardware decoder architectures, since it represents an important issue for industry. The remainder of the paper is divided into six parts. A survey of the general structure of turbo decoders is presented in Section II. Section III reviews the Soft-Input Soft-Output (SISO) algorithms that are used as fundamental building blocks of iterative decoders. Section IV deals with the issue of architectures dedicated to high throughput services. Particular stress is laid on the increase of the parallelism in the decoder architecture. Section V presents a review of the different stopping rules that can be applied in order to decrease the average number of iterations, thus also increasing the average decoding speed of the decoder. The

fixed-point implementation of decoders, which is desirable for hardware implementations is dealt with in Section VI. Finally, complexity issues, related to implementation choices, are discussed in Section VII.

II. Concatenation of convolutional codes and iterative decoding

A. Short history of concatenated coding and decoding

Code concatenation is a multilevel coding method allowing codes with good asymptotic as well as practical properties to be constructed. The idea dates back to Elias's product code construction in the mid 1950s [15]. A major advance was performed by Forney in his thesis work on concatenated codes ten years later [16]. As stated by Forney, concatenation is a method of building long codes out of shorter ones in order to resolve the problem of decoding complexity by breaking the required computation into manageable segments according to the divide and conquer strategy. The principle of concatenation is applicable to any type of codes, convolutional or block codes. This first version of code concatenation is now called *serial concatenation* (SC) of codes. Before the invention of turbo codes, the most famous example of concatenated codes was the concatenation of an outer algebraic code, such as a Reed-Solomon code, with an inner convolutional code, which has been used in numerous applications, ranging from space communications to digital broadcasting of television.

As far as the SC of convolutional codes is concerned, the straightforward way of decoding involves the use of two conventional Viterbi Algorithm (VA) decoders in a concatenated way. The first drawback that prevents such a receiver from performing efficiently is that the inner VA produces bursts of errors at its output, that the outer VA has difficulty correcting. This can be circumvented by inserting an interleaver between the inner and outer VAs, as illustrated in Figure 1.

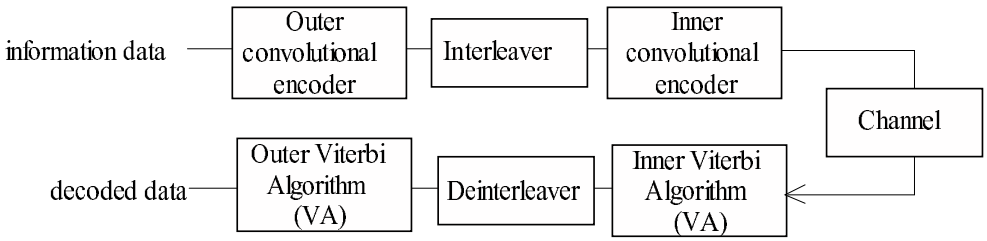


Fig. 1. Transmission scheme with the insertion of an interleaver into a serial concatenation of convolutional codes.

The second main drawback is that the inner VA provides hard decisions, thus preventing the outer VA from using its ability to accept soft samples at its input. In order to work properly, the inner decoder has to provide the outer decoder with soft information. Amongst the different attempts to achieve this goal, the Soft-Output Viterbi Algorithm (SOVA) approach calls for a modification of the VA in order to deliver a reliability value for each decoded bit [17][18]. Like the VA, the SOVA provides a maximum likelihood trellis[1] decoding of the convolutional code which minimizes the probability of a sequence error. However, it is suboptimal with respect to bit or symbol error probability. The minimal bit or symbol error probability can be achieved with symbol-by-symbol *Maximum A Posteriori* (MAP) decoding using the BCJR algorithm [19].

[1]The trellis diagram is a temporal representation of the code. It represents all the possible transitions between the states of the encoder as a function of time. The length of the trellis is equal to the number of time instants required to encode the whole information sequence.

The ultimate progress in decoding concatenated codes occurred when it was observed that a Soft-Input Soft-Output (SISO) decoder could be regarded as a signal-to-noise ratio amplifier, thus allowing common concepts in amplifiers such as the feedback principle to be implemented. This observation gave birth to the so-called *turbo* decoding principle of concatenated codes [20] in the early 1990s. Thanks to this decoding concept, the 1.5 dB gap still remaining between what theory promised and what the state of the art in error control coding was able to offer at the time[2] was almost removed. In Figure 1, the receiver clearly exploits the received samples in a sub-optimal way, even in the case where information passed from the inner decoder to the outer decoder is soft information. The overall decoder works in an asymmetrical way: both decoders work towards decoding the same data. The outer decoder takes advantage of the inner decoder work but the contrary is not true. The basic idea of turbo decoding involves a symmetric information exchange between both SISO decoders, so that they can converge to the same probabilistic decision, as a global decoder would. The issue of stability, which is crucial in feedback systems, was solved by introducing the notion of *extrinsic information*, which prevents the decoder from being a positive feedback amplifier. In the case where the component decoders compute the Logarithm of Likelihood Ratios (LLR) related to information data, the extrinsic information can be obtained with a simple subtraction between the output and the input of the decoder as described in Figure 2.

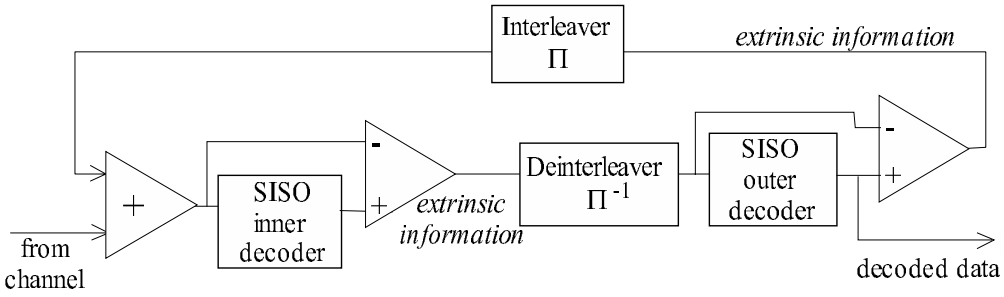


Fig. 2. General principle of turbo decoding in the logarithmic domain: extrinsic information symmetrically exchanged between the inner and outer SISO decoders can be obtained with a simple subtraction between the ouput and the input of the decoders.

B. Parallel and serial concatenation of convolutional codes

1) Parallel versus serial concatenation of convolutional codes: The introduction of turbo codes is also the origin of the concept of *parallel concatenation* (PC). Figure 3(a) shows the PC of two convolutional codes, as commonly used in classical turbo codes. With the PC of codes, the message to be transmitted is encoded twice in a separate fashion: the first encoder processes the data message in the order it is delivered by the source while the second one encodes the same sequence in a different order obtained by way of an interleaver.

[2]In the early 1990s, the state of the art in error control coding was the concatenation of a Reed-Solomon code and a memory 14 convolutional code proposed for the Galileo space probe [21]. This scheme was decoded using a four stage iterative scheme where hard decisions were fedback from the Reed-Solomon decoder to the Viterbi decoder.

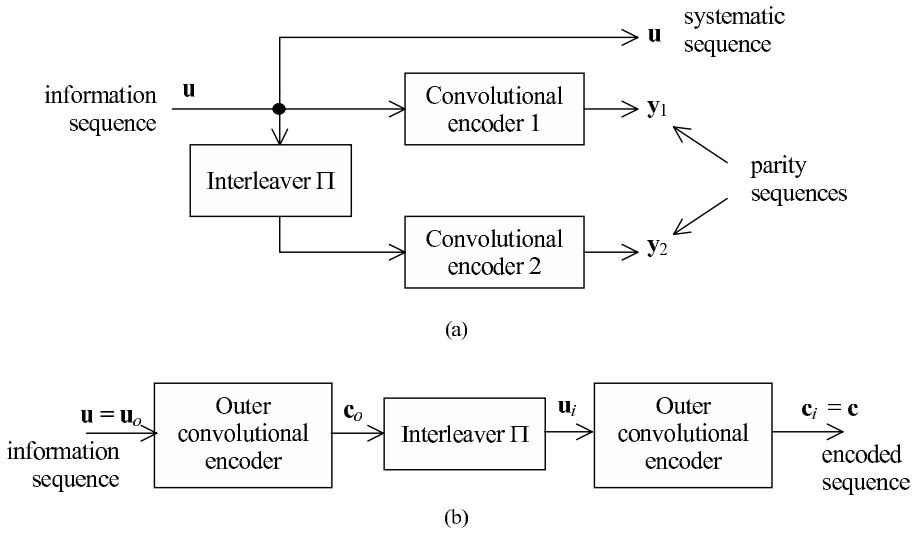


Fig. 3. General structures of a PC (a) and an SC (b) of two convolutional encoders. In (a), the encoded sequence $\mathbf{c}$ is obtained through the concatenation of the information or systematic sequence and the redundancy sequences provided by the two constituent encoders, $\mathbf{c} = (\mathbf{u}, \mathbf{y_1}, \mathbf{y_2})$.

The overall coding rates for SC and PC, R_s and R_p, are equal to

$$R_s = R_i R_o$$

and

$$R_p = \frac{R_1 R_2}{R_1 + R_2 - R_1 R_2} = \frac{R_1 R_2}{1 - (1 - R_1)(1 - R_2)}$$

where R_i and R_o refer to the coding rates of the inner and outer constituent codes in the SC scheme and R_1 and R_2 refer to the rates of code 1 and code 2 in the PC scheme.

Historical turbo codes [1]-[22] are based on the PC of two Recursive Systematic Convolutional (RSC) codes. Firstly, the choice of RSC codes instead of classical Non Recursive Convolutional (NRC) codes has to be found in the comparison of their respective error correction performance. An example of such memory length $\nu = 3$ codes is shown in Figure 4. Since they have the same minimum Hamming distance[3], the observed error correcting performance is very similar for both codes at medium and low error rates. However, the RSC code performs significantly better at low signal to noise ratios [22].

However, the reason why using RSC codes is essential in the construction of concatenated convolutional codes was explained through bounding techniques. It was shown in [23][24] that, under the assumption of *uniform*

[3]The minimum Hamming distance d_{min} of a code is the smallest Hamming distance between two any different encoded sequences. The correcting capability of the code is directly related to the value of d_{min}.

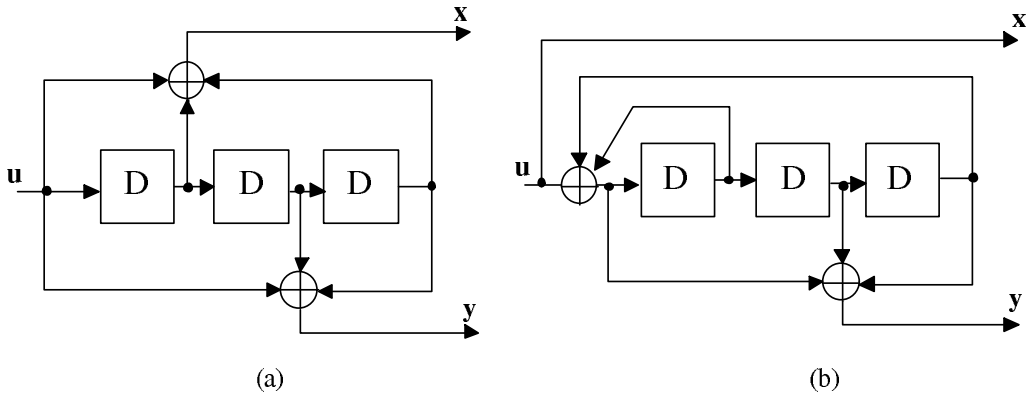


Fig. 4. (a) Classical non recursive non systematic onvolutional code with $\nu = 3$ memory units (8-state code). (b) Equivalent recursive systematic version of (a) code.

interleaving[4], the bit error probability in the low error region, P_b, varies asymptotically with the *interleaver gain*:

$$P_b \propto N^{\alpha_{max}} \tag{1}$$

where N is the interleaver length and α_{max} is the maximum exponent of N in the asymptotic union bound approximation. In the PC case, the maximum exponent is equal to $\alpha_{max} = 1 - w_{min}$, where w_{min} is the minimum input Hamming weight of finite error events. This result shows that there is an interleaving gain only if $w_{min} > 1$, which is true for RSC codes ($w_{min} = 2$ and $\alpha_{max} = -1$) but not for NRC codes ($w_{min} = 1$ and $\alpha_{max} = 0$). As for SC schemes, if both constituent codes are NRC, the same problem appears. It was shown that, for SCCC, at least the inner code has to be recursive [25] in order to ensure an interleaver gain. Provided this condition is satisfied, the maximum exponent is equal to $\alpha_{max} = -\left\lfloor \frac{d_{min,o}+1}{2} \right\rfloor$, where $d_{min,o}$ is the minimum Hamming distance of the outer code.

This analytical analysis also allows us to compare the SC and PC performance at low error rates: serial turbo codes perform better than parallel turbo codes in this region because their interleaver gain is larger. As a counterpart, the opposite behavior can be observed at high error rates, for the same overall coding rate. This can be explained through the analysis of extrinsic information transfer characteristics based on mutual information, the so-called *EXIT charts* [26].

2) Block coding with convolutional codes: Convolutional codes are not *a priori* well suited to encode information transmitted in block form. Nevertheless, most practical applications require the transmission of data in block fashion, the size of the transmitted blocks being sometimes reduced to less than a hundred bits (see *e.g.* the 3GPP turbo code [8] with a 40-bit minimum block length). In order to properly decode data blocks at the receiver side, the decoder needs some information about the encoder state at the beginning and the end of the encoding process.

[4]A uniform interleaver is a probabilistic device that maps a given sequence of length N and Hamming weight w into all distinct permutations of length N and Hamming weight w with equal probability. The uniform interleaver is representative of the average performance over all possible deterministic interleavers.

The knowledge of the initial state of the encoder is not a problem, since the "all zero" state is, in general, forced. However, the decoder has no special available information regarding the final state of the encoder and its trellis. Several methods can solve this problem, for example:

- **do nothing**, that is, no information concerning the final states of the trellises is provided to the decoder. The trellis is truncated at the end of each block. The decoding process is less effective for the last encoded data and the asymptotic coding gain may be reduced. This degradation is a function of the block length and may be low enough to be accepted for a given application.
- **force the encoder state at the end of the encoding phase** for one or all constituent codes. This solution was adopted by the CCSDS and UMTS turbo codes [7][8]. The trellis termination of a constituent code involves encoding ν extra bits, called *tail bits*, in order to make the encoder return to the all-zero state. These tail bits are then sent to the decoder. This method presents two drawbacks. Firstly, the spectral efficiency of the transmission is slighly decreased. Nevertheless, this reduction is negligible except for very short blocks. Next, for parallel turbo codes, tail bits are not identical for the termination of both constituent codes, or in other words, they are not turbo encoded. Consequently, symbols placed at the block end have a weaker protection again. As for serial turbo codes, the tail bits used for the termination of the inner coder are not taken into account in the turbo decoding process, thus leading to a similar problem. However, the resulting loss in performance is very small and can be acceptable in most applications.
- **adopt tail-biting** [27]. This technique allows any state of the encoder as the initial state. The encoding task is performed so that the final state of the encoder is equal to its initial state. The code trellis can then be viewed as a circle, without any state discontinuity. Tail-biting presents two main advantages in comparison with trellis termination using tail bits to drive the encoder to the all-zero state. Firstly, no extra bits have to be added and transmitted. Next, with tail-biting RSC codes, only codewords with minimum input weight 2 have to be considered. In other words, tail-biting encoding avoids any side effects, unlike classical termination. This is particularly attractive for highly parallel hardware implementations of the decoder, since the block sides do not require any specific processing. In practice, the straightforward circular encoding of a data block consists of a two-step process: at the first step, the information sequence is encoded from the all-zero state and the final state is stored. During this first step, the output bits are ignored. The second step is the actual encoding, whose initial state is a function of the final state previously stored. The double encoding operation represents the main drawback of this method, but in most cases it can be performed at a frequency much higher than the data rate. An increased amount of memory is also required to store the state information related to the start and the end of the block between iterations.

C. Iterative decoders for concatenated convolutional codes

The decoding principle of PCCC and SCCC is shown in Figures 5 and 6. The SISO decoders are assumed to process LLRs at their inputs $\lambda(\ ;I)$ and outputs $\lambda(\ ;0)$ (the notations used in the figure are those adopted in Section III).

In the PC scheme of Figure 5, each SISO decoder computes the extrinsic LLRs related to information symbols $\lambda(\mathbf{u}_1; O)$ and $\lambda(\mathbf{u}_2; O)$, using the observation of the associated systematic and parity symbols coming from the transmission channel $\lambda(\mathbf{c}_1; I)$ and $\lambda(\mathbf{c}_2; I)$, and the *a priori* LLRs $\lambda(\mathbf{u}_2; I)$ and $\lambda(\mathbf{u}_1; I)$. Since no *a priori* LLRs are avalaible from the decoding process at the beginning of the iterations, they are then set to zero. For the subsequent iterations, the extrinsic LLRs coming from the other decoder are used as *a priori* LLRs for the current SISO decoder. The decisions can be computed from any of the decoders. In the PC case, the turbo decoder structure is symmetrical with respect to both constituent decoders. However, in practice the SISO processes are executed in a sequential fashion: the decoding process starts arbitrarily with either one decoder, SISO1 for example. After SISO1 processing is completed, SISO2 starts processing and so on. In the SC scheme of Figure 6, the decoding diagram is no longer symmetrical. On the one hand, the inner SISO decoder computes extrinsic LLRs $\lambda(\mathbf{u}_i; O)$ related to the inner code information symbol, using the observation of the associated coded symbols coming from the transmission channel $\lambda(\mathbf{c}_i; I)$ and the extrinsic LLRs coming from the other SISO decoder $\lambda(\mathbf{u}_i; I)$. On the other hand, the outer SISO decoder computes the extrinsic LLRs $\lambda(\mathbf{c}_o; O)$ related to the outer code symbols using the extrinsic LLRs provided by the inner decoder. The decisions are computed as *a posteriori* LLRs $\lambda(\mathbf{u}_o; O)$ related to information symbols by the outer SISO decoder. Although the overall decoding principle depends on the type of concatenation, both turbo decoders can be constructed from the same basic building SISO blocks, as described in Section III.

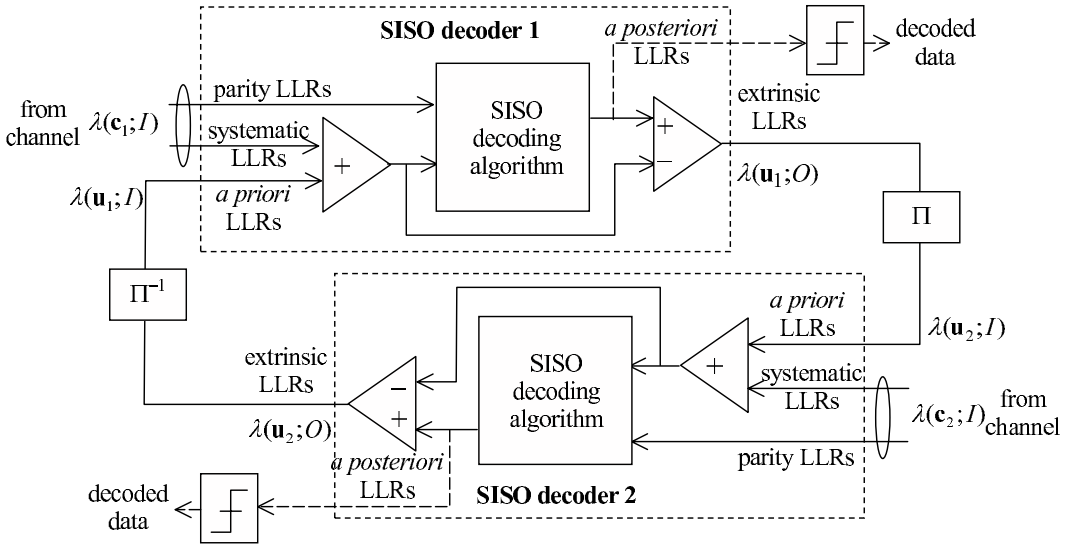


Fig. 5. The turbo decoding principle in the case of parallel turbo codes. Notations are taken from Figure 3(a). $\lambda(\ ; I)$ and $\lambda(\ ; 0)$ refer to the LLRs at the input and output of the SISO decoders.

For digital implementations of turbo decoders, the different processing stages present a non-zero internal delay, with the result that turbo decoding can only be implemented through an iterative process. Each SISO decoder processes its own data and passes it to the other SISO decoder. One iteration corresponds to one pass through each of all the SISO decoders. One pass through a single SISO decoder is sometimes referred to as half an iteration of

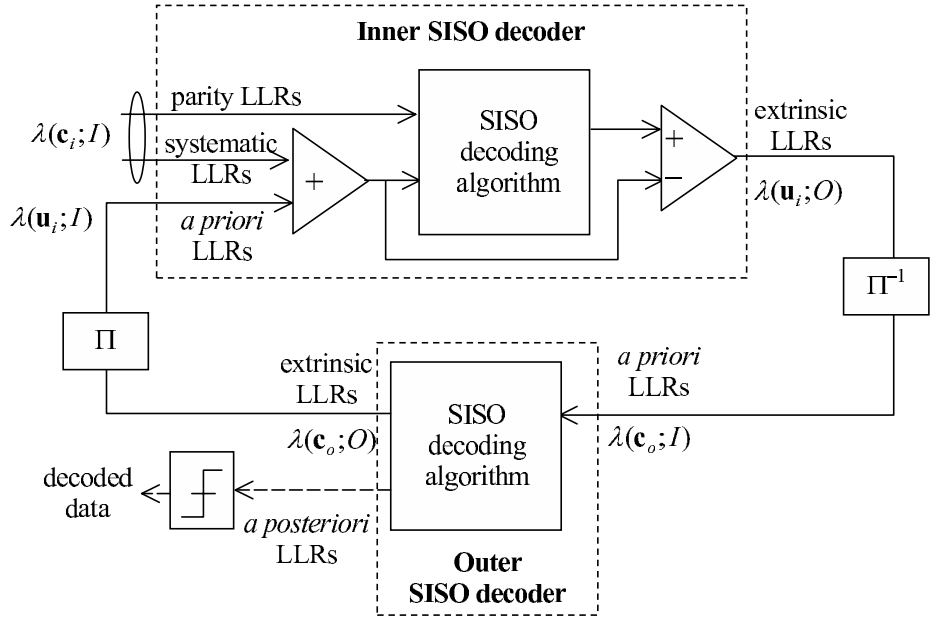


Fig. 6. The turbo decoding principle in the case of serial codes. Notations are taken from Figure 3(b). $\lambda(\ ;I)$ and $\lambda(\ ;0)$ refer to the LLRs at the input and output of the SISO decoders.

decoding.

The concatenated encoders and decoders can work continuously or block-wise. Since convolutional codes are naturally better suited to encode information in a continuous fashion, the very first turbo encoders and decoders [2][28] were stream-oriented. In this case, there is no constraint about the termination of the constituent encoders, and the best interleavers turned out to be *periodic* or *convolutional* interleavers [29][30]. The corresponding decoders call for a modular pipelined structure, as illustrated in Figure 7 in the case of parallel turbo codes. The basic decoding structure has to be replicated as many times as the number of iterations. In order to ensure the correct timing of the decoding process, delay lines have to be inserted into the decoder. The decision computation, not shown in the figure, is performed at the output of SISO decoder 2, at the last iteration stage.

However, as far as block decoding is concerned, the simplest decoding architecture is based on the use of a single SISO decoder, which alternately processes both constituent codes. This standard architecture requires three storage units: a memory for the received data at the channel and decoder outputs (LLR_{in} memory), a memory for extrinsic information at the SISO output (EXT memory), and a memory for the decoded data (LLR_{out} memory). The instantiations of this architecture in the PC and SC cases are illustrated in Figures 8 and 9. In the parallel scheme, the decoding architecture is the same for both component codes, since they play the same role in the overall decoding process. The SISO decoder decodes code 1 or 2 using the corresponding channel data from the LLR_{in} memory and the *a priori* information stored in the EXT memory at the previous half-iteration. The resulting extrinsic information is stored in the EXT memory and then used as *a priori* information at the SISO input when the other code is processed, at the next half-iteration. The decoded data are written into the LLR_{out} memory at the last

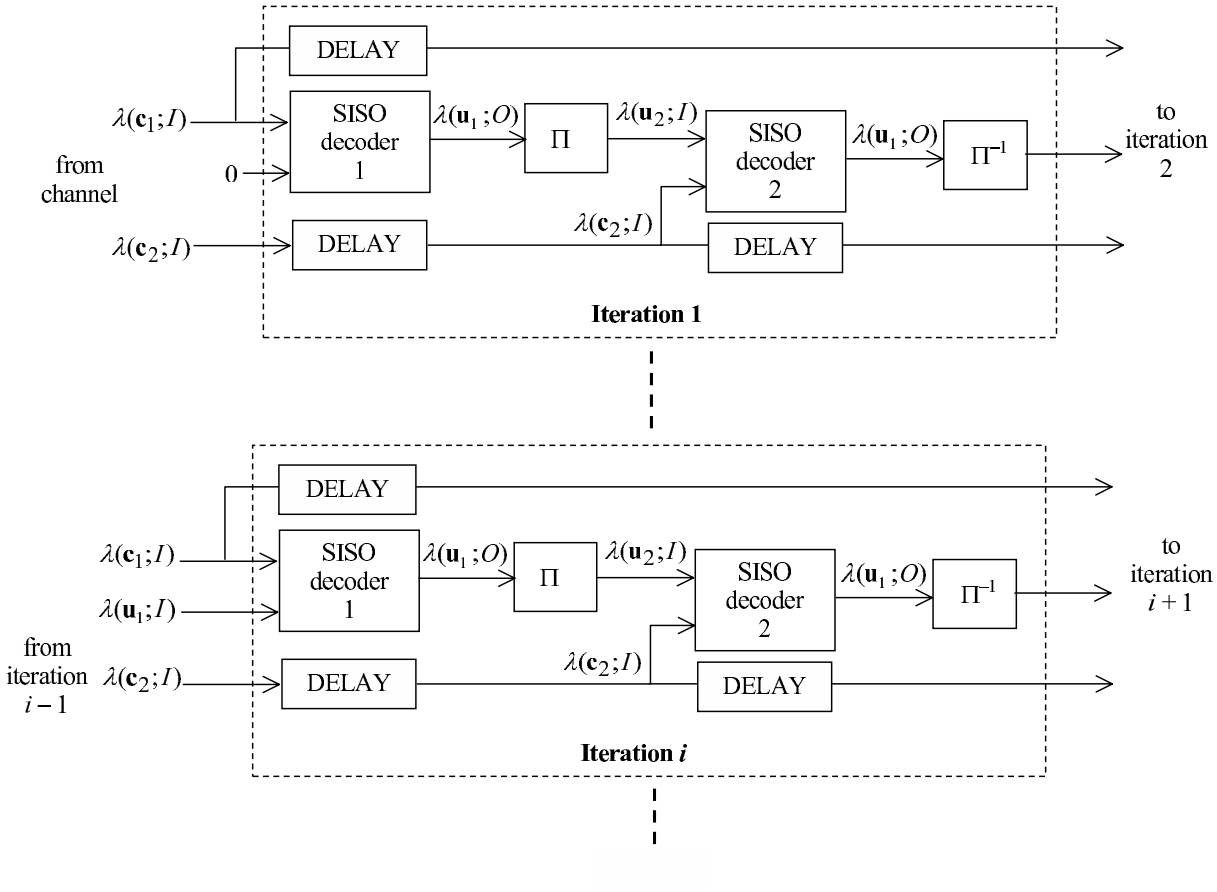


Fig. 7. Pipelined architecture of the turbo decoder of Figure 5, in the case of continuous decoding of PCCC. The *a priori* LLRs at the input of the first iteration stage are set to 0. The decision computation, performed at the last iteration stage, is not shown.

half-iteration of the decoding process. On the contrary, in the SC scheme, the architecture elements are not used in the same fashion for inner and outer code processing. The inner decoding process, shown in Figure 9(a), is similar to the elementary decoding process in the PC case, whereas the outer decoding process, shown in Figure 9(b), does not use the LLR_{in} memory contents or the *a priori* information input $\lambda(\mathbf{u}_o;I)$. The decoded data are written into the LLR_{out} memory after the last processing round of the outer decoder.

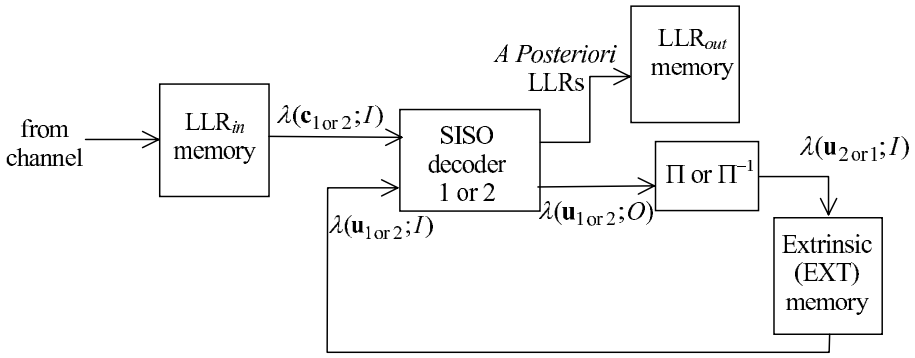
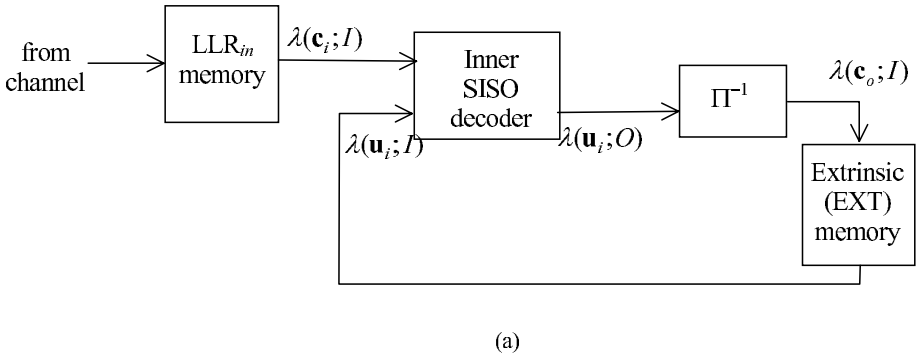
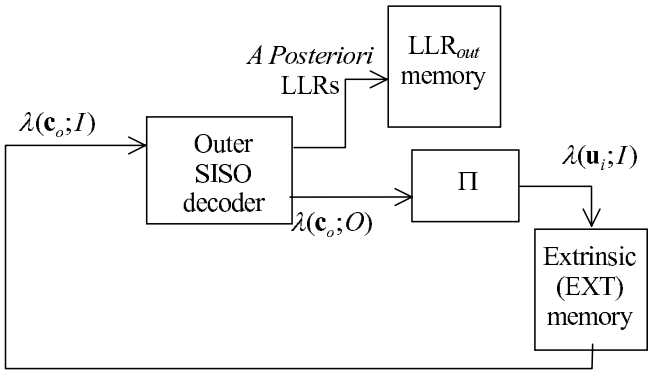


Fig. 8. Standard architecture for block-wise decoding of parallel turbo codes.



(a)



(b)

Fig. 9. Standard architecture for block-wise decoding of serial turbo codes: (a) refers to the inner decoder, (b) refers to the outer decoder.

III. Soft Input Soft Output Algorithms

Soft Input Soft Output (SISO) algorithms are the fundamental building blocks of iterative decoders. A SISO is in general a block that accepts *messages* (to be defined later) about both the input and output symbols of an *encoder* and provides *extrinsic* messages about the same symbols. The extrinsic message is generated by considering the *a priori* constraints that exist between the input and output sequences of the encoder. In this section we will give more precise definitions of the "message" and on the input output relationships of a SISO block. Moreover we will show efficient algorithms to perform SISO for some special types of encoders.

A. Definition of the input and output metrics

A SISO module generally works associated to a known mapping $\mathbf{f}$ (encoding) between input and output alphabets.

$$\mathbf{c} = \mathbf{f}(\mathbf{u}) = (f_1(\mathbf{u}), \ldots, f_n(\mathbf{u})) \quad \mathbf{u} \in \mathcal{U} \quad \mathbf{c} \in \mathcal{C} \tag{2}$$

where $\mathbf{u} = (u_1, \ldots, u_k)$ and $\mathbf{c} = (c_1, \ldots, c_n)$ are the input and output sequences of the encoder, respectively. The alphabets $\mathcal{U}$ and $\mathcal{C}$ are generic finite alphabets and the mapping is not necessarily invertible.

A SISO module is a four port device that accepts some messages of the input and output symbols of the encoder and provides homologous output *extrinsic* messages.

We will consider the following two types of normalized messages:

Likelihood ratio (LR)

$$L(x) \triangleq \frac{\mathrm{P}(X = x)}{\mathrm{P}(X = 0)},$$

represents the ratio between the likelihood of the symbol being x and the likelihood of being 0.

Log-Likelihood ratio (LLR)

$$\lambda(x) = \log \frac{\mathrm{P}(X = x)}{\mathrm{P}(X = 0)} \rightarrow e^{\lambda(x)} = L(x)$$

is its logarithmic version.

Normalized messages are usually preferred to not normalized ones as they allow the decoder to save one value in the representation. In fact, by definition $L(0) = 1$ and $\lambda(0) = 0$. In particular, when the variable x is binary ($x \in \{0, 1\}$) LRs and LLRs can be represented with a scalar $L_x = L(x = 1)$[5] and $\lambda_x = \lambda(x = 1)$ so that one can write

$$\begin{aligned} L(x) &= (L_x)^x \\ \lambda(x) &= x\lambda_x \end{aligned} \tag{3}$$

The sequence of LRs is always assumed to be *independent* at the input of SISO[6] so that the likelihood ratio of the sequences $\mathbf{u}$ and $\mathbf{c}$ is the product of the likelihoods of their constituent symbols

$$L(\mathbf{u}) = \prod_{i=1}^{k} L(u_i), \quad L(\mathbf{c}) = \prod_{j=1}^{n} L(c_j)$$

[5]We have kept the index x with the binary LLR to remind the reader the name of underlying symbol.

[6]This is actually the basic assumption of iterative decoding, which otherwise would be optimum.

and equivalently its LLR:

$$\lambda(\mathbf{u}) = \sum_{i=1}^{k} \lambda(u_i), \quad \lambda(\mathbf{c}) = \sum_{j=1}^{n} \lambda(c_j)$$

Furthermore, assuming that the sequence of LR of input and output symbols are mutually independent, the LR of a pair $(\mathbf{u}, \mathbf{c})$, with the constraint of being a valid correspondence can be obtained as

$$L(\mathbf{u}, \mathbf{c}) = \begin{cases} L(\mathbf{u})L(\mathbf{f}(\mathbf{u})), & \textbf{c=f(u);} \\ 0, & \text{otherwise.} \end{cases}$$

B. General SISO relationships

The formal SISO input output relationships are obtained with the independence assumption constraining the set of input/output sequences to be in the set of possible mapping correspondences. Using LR messages the relationships are

$$L(u_i;O)L(u_i;I) = \frac{\sum_{\mathbf{u}':u'_i=u_i} L(\mathbf{u}';I)L(\mathbf{f}(\mathbf{u}');I)}{\sum_{\mathbf{u}':u'_i=0} L(\mathbf{u}';I)L(\mathbf{f}(\mathbf{u}');I)} \tag{4}$$

$$L(c_j;O)L(c_j;I) = \frac{\sum_{\mathbf{u}':f_j(\mathbf{u}')=c_j} L(\mathbf{u}';I)L(\mathbf{f}(\mathbf{u}');I)}{\sum_{\mathbf{u}':f_j(\mathbf{u}')=0} L(\mathbf{u}';I)L(\mathbf{f}(\mathbf{u}');I)} \tag{5}$$

where we have introduced the letters "I" and "O" to distinguish between input and output messages.

In the logarithmic domain (LLRs), products translate to sums and sums are mapped to the operator

$$\max^*(\lambda_1, \lambda_2) \triangleq \log(e^{\lambda_1} + e^{\lambda_2}) = \max(\lambda_1, \lambda_2) + \log(1 + e^{-|\lambda_1 - \lambda_2|}) \tag{6}$$

and (4) and (5) (LLRs) become

$$\lambda(u_i;O) + \lambda(u_i;I) = \max_{\mathbf{u}':u'_i=u_i}{}^* [\lambda(\mathbf{u}';I) + \lambda(\mathbf{f}(\mathbf{u}');I)] - \max_{\mathbf{u}':u'_i=0}{}^* [\lambda(\mathbf{u}';I) + \lambda(\mathbf{f}(\mathbf{u}');I)] \tag{7}$$

$$\lambda(c_j;O) + \lambda(c_j;I) = \max_{\mathbf{u}':f_j(\mathbf{u}')=c_j}{}^* [\lambda(\mathbf{u}';I) + \lambda(\mathbf{f}(\mathbf{u}');I)] - \max_{\mathbf{u}':f_j(\mathbf{u}')=0}{}^* [\lambda(\mathbf{u}';I) + \lambda(\mathbf{f}(\mathbf{u}');I)] \tag{8}$$

The algorithm obtained from the first type of messages (LR) is called *multiplicative* (sum-prod) while the second (LLRs) is called *additive* (max*-sum), or log-MAP.

The $\max^*$ operator requires in general two sums and a look-up table. The look-up table size depends on the required accuracy and the whole correction term can be avoided, giving rise to a simpler and suboptimal version of the SISO (max-sum or max-log-MAP). To compensate the effect of neglecting the look-up table several strategies are possible, like scaling or offsetting the messages. These techniques are described in [31], [32]

Independently from the metric used, the input-output relationships (4),(5) or (7),(8) have a complexity that grows with the size of the code. This complexity can be affordable for very simple mappings but becomes impractical for most of the mappings used as encoders.

In the following sections we will describe some particular cases where this computation can be simplified.

C. Binary mappings

As we have seen, for binary variables LR and LLR have the appealing feature of being able to be represented as single values, in particular, using relationships (3) we have:

$$\begin{aligned}\lambda_{u_l}(O)+\lambda_{u_l}(I) &= \max_{\mathbf{u}':u'_l=1}{}^{*}\left(\sum_{i=1}^{k}u'_i\lambda_{u_i}(I)+\sum_{j=1}^{n}f_j(\mathbf{u}')\lambda_{c_j}(I)\right)\\ &- \max_{\mathbf{u}':u'_l=0}{}^{*}\left(\sum_{i=1}^{k}u'_i\lambda_{u_i}(I)+\sum_{j=1}^{n}f_j(\mathbf{u}')\lambda_{c_j}(I)\right) \qquad (9)\\ \lambda_{c_l}(O)+\lambda_{c_l}(I) &= \max_{\mathbf{u}':p_l(\mathbf{u}')=1}{}^{*}\left(\sum_{i=1}^{k}u'_i\lambda_{u_i}(I)+\sum_{j=1}^{n}f_j(\mathbf{u}')\lambda_{c_j}(I)\right)\\ &- \max_{\mathbf{u}':p_l(\mathbf{u}')=0}{}^{*}\left(\sum_{i=1}^{k}u'_i\lambda_{u_i}(I)+\sum_{j=1}^{n}f_j(\mathbf{u}')\lambda_{c_j}(I)\right) \qquad (10)\end{aligned}$$

D. SISO relationships on trellises

In this section we will explain how to simplify the computation of (4) and (5), or their logarithmic counterparts (7) and (8) when the mapping is represented over a *trellis*. As the correspondence between the multiplicative and additive domain requires only the use of different operators we will describe the algorithm in the multiplicative domain.

A trellis is an object characterized by the concatenation of L *trellis sections* $\mathcal{T}_l$. Each trellis section (see Figure 10) consists of a set of starting states $s_l \in \mathcal{S}_l$, an input set $x_l \in \mathcal{X}_l$, the set of *edges* defined as the pair $e_l = (s_l, x_l) \in \mathcal{E}_l = \mathcal{S}_l \times \mathcal{X}_l$ and two functions $p_l(e_l)$ and $y_l(e_l)$ that assign to each edge a final state in $\mathcal{S}_{l+1}$ and an output symbol in $\mathcal{Y}_l$.

A trellis is associated to a mapping (2) by making a correspondence of the L trellis section's input and output alphabets with the k input and n output alphabets with the mapping:

$$\begin{aligned}\bigotimes_{l=1}^{L}\mathcal{X}_l &= \bigotimes_{i=1}^{k}\mathcal{U}_i\\ \bigotimes_{l=1}^{L}\mathcal{Y}_l &= \bigotimes_{j=1}^{n}\mathcal{C}_j,\end{aligned}$$

where $\bigotimes$ denotes the Cartesian product of sets.

The alphabets of the trellis sections must then be the Cartesian product of a set of sub-alphabets of the original mapping:

$$\begin{aligned}\mathcal{X}_l &= \bigotimes_{i\in I_l}\mathcal{U}_i\\ \mathcal{Y}_l &= \bigotimes_{j\in J_l}\mathcal{C}_j\end{aligned}$$

where $\{I_l\}$ and $\{J_l\}$ are a partition of the sets of indexes of the input and output alphabets. Furthermore, the cardinality of $\mathcal{S}_1$ and $\mathcal{S}_{L+1}$ is always one.

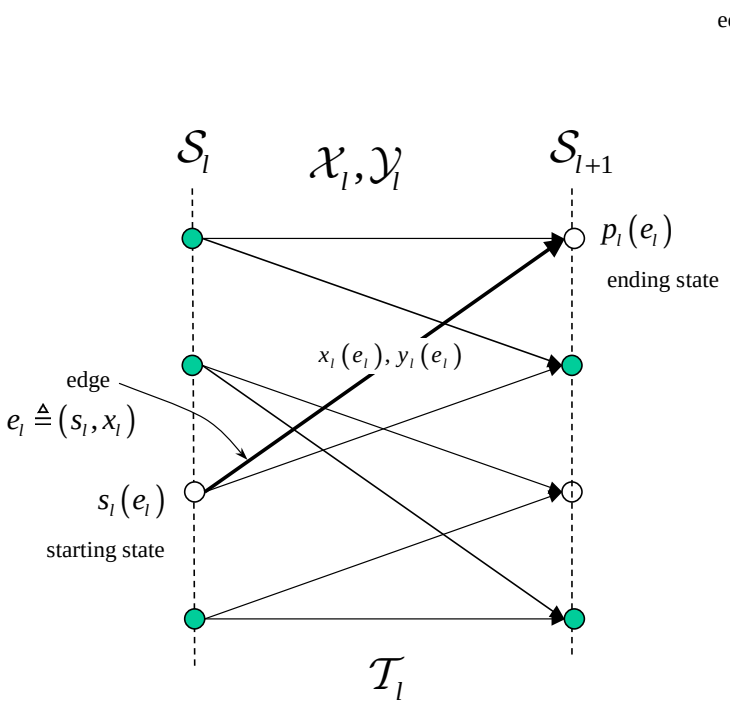


Fig. 10. A trellis section and related functions. The starting state, the ending state, the input symbol and the output symbol. Note that edge is *defined* as the pair starting state, input symbol.

Any mapping (2) in general admits several *time varying trellis* representations, in particular, the number of trellis sections L and the alphabets defining each trellis section can be arbitrarily fixed. Finding a trellis with minimal complexity (i.e., minimal number of edges as explained in [33]) is a rather complex task and is out of the scope of this tutorial paper.

A trellis representation for a mapping allows us to interpret the mapping itself as *paths* in the representing trellis. This correspondence allows us to build efficient encoders based on time varying finite state machines. More importantly, this same structure can be exploited for the efficient evaluation of expressions like those in (4) and (5) or (7) and (8) that involve associative and distributive operators.

In fact due to the distributive and associative properties of the operators appearing in (4), (5), it is possible to compute the required output extrinsic LRs with the following algorithm ([19])

1) From the likelihoods of the alphabets of the mapping $L(u_i; I)$ and $L(c_j; I)$ compute the likelihoods of the alphabets of the trellis sections and from them the likelihoods of the edges (branch metrics)

$$C_l(x_l) = \prod_{i \in I_l} L(u_i; I) \quad \forall x_l \in \mathcal{X}_l \tag{11}$$

$$C_l(y_l) = \prod_{j \in J_l} L(c_j; I) \quad \forall y_l \in \mathcal{Y}_l \tag{12}$$

$$\rightarrow \quad G_l(e_l) = C_l(x_l) C_l(y_l(e_l))$$

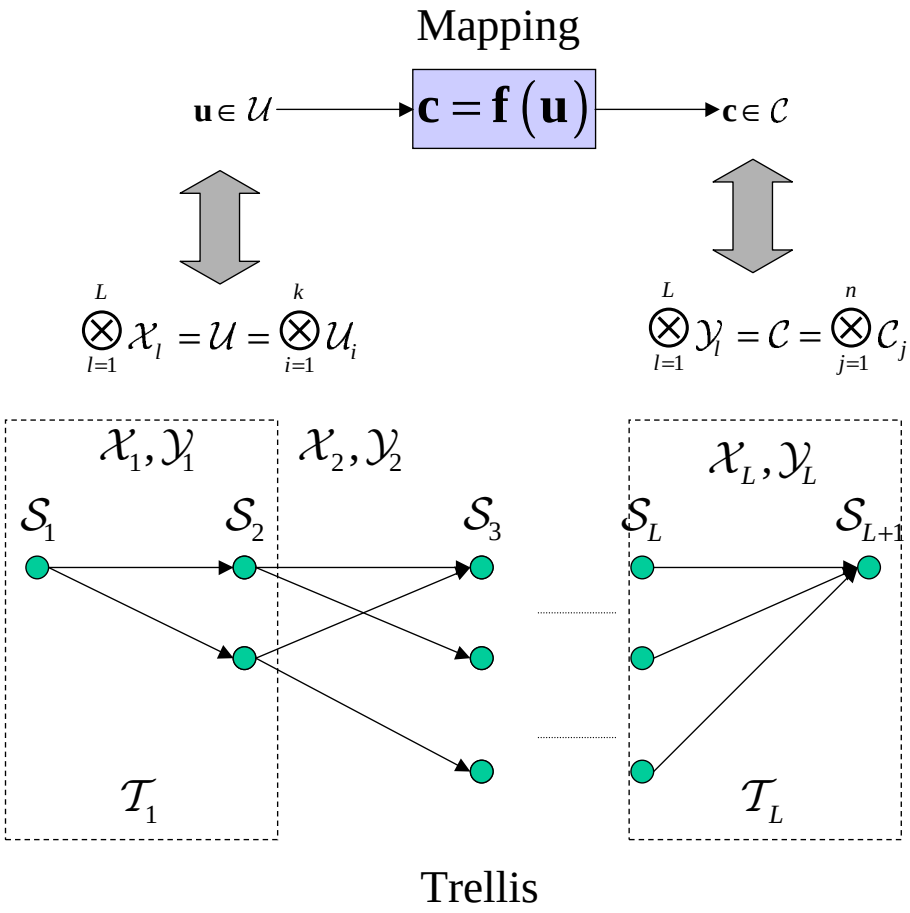


Fig. 11. Correspondence between a mapping and a time varying trellis.

2) Compute the *forward* and *backward* recursions according to

$$A_{l+1}(s) = \sum_{e_l : p_l = s} A_l(s_l) \cdot G_l(e_l) \quad l = 1, \ldots, L-1 \tag{13}$$

$$B_l(s) = \sum_{e_l : s_l = s} B_{l+1}(p_l) \cdot G_l(e_l) \quad l = L, \ldots, 2 \tag{14}$$

with initializations

$$A_1(0) = B_{L+1}(0) = 1$$

Periodically, normalization at this point may be introduced to avoid overflows. The normalization can be performed by dividing all state metrics by a reference one. Note however that in the additive version normalization can be avoided by using complement-two representation of state metrics using a technique that can be found in the literature relative to Viterbi decoders.

3) Compute the *a posteriori* likelihood of edges,

$$D_l(e_l) = A_l(s_l) \cdot G_l(e_l) \cdot B_{l+1}(p_l) \quad \forall e_l \quad l = 1, \ldots, L$$

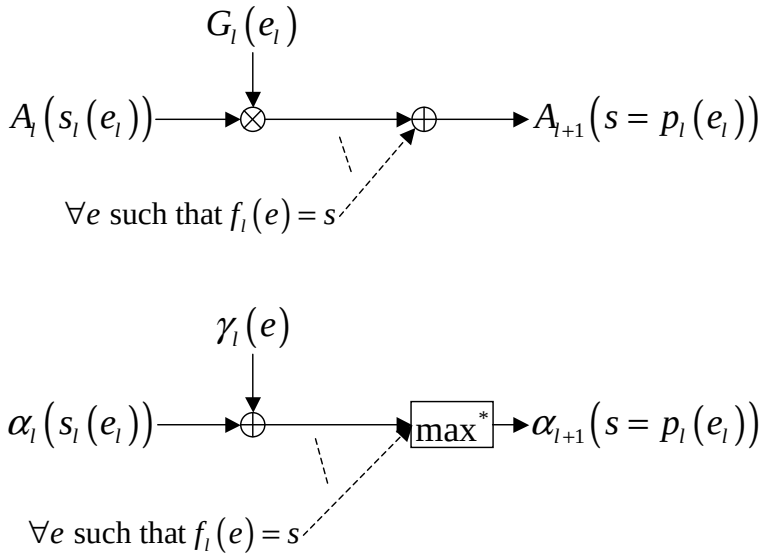


Fig. 12. Implementation of forward recursion in the additive and the multiplicative version.

4) Finally compute the desired extrinsic output LRs as

$$L(u_i;O) = \frac{1}{L(u_i;I)} \frac{\sum_{e_l:u_i(e_l)=u_i} D(e_l)}{\sum_{e_l:u_i(e)=0} D(e_l)} \quad (15)$$

$$L(c_j;O) = \frac{1}{L(c_j;I)} \frac{\sum_{e_l:c_j(e_l)=c_j} D(e_l)}{\sum_{e_l:c_j(e)=0} D(e_l)} \quad (16)$$

where l in (15) (resp. (16)) is the index of the trellis section associated to the symbol u_i (resp. c_j).

If no information is available for some of the symbols involved in the mapping, their corresponding LR should be set to the vector $\mathbf{1}$ and their LLR to the vector $\mathbf{0}$. If no extrinsic information is required for some of the symbols involved in the mapping, the corresponding final relationships (15) or (16) can be avoided.

Due to the finite memory of the trellis, forward and backward recursions (13) and (14) generally are such that $A_{l+W}(s)$ and $B_{l-W}(s)$ are independent from $A_l(s)$ (resp. $B_l(s)$) for sufficiently large W. This fact also allows the algorithm to work when messages are available only for a contiguous subset of trellis sections. See section III-F for more details.

In Figure 12 we show as an example the block diagram of the forward recursion in its multiplicative and additive forms. Note that in the additive form we denote with lower case Greek letters the logarithmic counterparts of the variables defined for the multiplicative version.

The described SISO algorithm is completely general and can be used for any mapping and trellis. In the following we will consider some special cases of mappings where the algorithm can be further simplified.

- **Convolutional encoders:** The trellis of a convolutional encoder has trellis sections that do not depend on the time index. A convolutional encoder has a constant set of states, a constant input and output alphabet and constant functions $p(e)$ and $y(e)$. Convolutional encoders define a mapping between semi-infinite sequences

when the starting state is fixed.

$$\begin{aligned} \mathcal{X} &= \bigotimes_{i=0}^{k_0} \mathcal{U}_i \\ \mathcal{Y} &= \bigotimes_{j=0}^{n_0} \mathcal{C}_j \end{aligned}$$

- **Binary Convolutional encoders:** have the additional constraint of having $\mathcal{U}_i = \mathcal{C}_j = \mathbb{Z}_2$. For them each trellis section is characterized with a set of k_0 input bits and n_0 output bits. LR and LLR are single quantities as shown in (3), so that the branch metrics can be computed simply as

$$\begin{aligned} C_l(x) &= \sum_{i=0}^{k_0-1} u_i \cdot \lambda_{u_i}(I) \\ C_l(y) &= \sum_{j=0}^{n_0-1} c_j \cdot \lambda_{c_j}(I) \end{aligned}$$

- **Linear Binary Convolutional encoders:** For linear encoders we have the additional linear property

$$\mathbf{f}(\mathbf{u_1} \oplus \mathbf{u_2}) = \mathbf{f}(\mathbf{u_1}) \oplus \mathbf{f}(\mathbf{u_2})$$

  The linearity of an encoder does not simplify the SISO algorithm. However linear encoders admit dual encoders and the SISO algorithm can be performed with modified metrics (as we will see in the next section) on the trellis of the *dual* code. This fact may lead to considerable savings especially for high rate codes that have simpler dual trellis.

- **Systematic encoders:** Systematic encoders are encoders for which the output symbol y_l is obtained by concatenating the input symbol x_l with a redundancy symbol r_l:

$$y_l = (x_l, r_l)$$

  For them the computation of metrics (11) and (12) is simplified as the metric on x_l can be incorporated in those of y_l:

$$C_l(y_l) = \prod_{i \in I_l} L(u_i; I) L(c_i; I) \prod_{j \in J_l / I_l} L(c_j; I),$$

  where the first product refers to the systematic part of the label y_l and the second part to the redundancy r_l.

E. SISO algorithm for dual codes

In the previous section we saw that the linear property of the code does not simplify the SISO algorithm. However, when the encoder is linear and binary[7] we recall a fundamental result first derived in [34] and restated here.

[7]The result can be easily extended more generally to finite fields.

Defining the new binary messages called *reflection coefficients* R_x from the LR L_x as

$$R_x \triangleq \frac{1 - L_x}{1 + L_x}$$

and the correspondent sequence reflection coefficients

$$R(\mathbf{c}) \triangleq \prod_{j=1}^{n} (R_{c_j})^{c_j}$$

The following relationship holds true

$$R(c_j; O)R(c_j; I) \quad = \quad \frac{\sum_{\mathbf{u}': f_j^{\perp}(\mathbf{u}')=c_j} R(\mathbf{f}^{\perp}(\mathbf{u}'); I)}{\sum_{\mathbf{u}': f_j^{\perp}(\mathbf{u}')=0} R(\mathbf{f}^{\perp}(\mathbf{u}'); I)} \tag{17}$$

where $f^{\perp}$ is the mapping that defines the dual encoder, i.e. the set of sequences orthogonal to the code. This relationship, formally identical to (5), can be used to perform SISO decoding of high rate linear binary encoders with a complexity that is the one associated to their duals ([35], [36], [37]). A very important case where this property is exploited is in the LDPC decoding for the efficient SISO computation at the check nodes side, as the dual code of the $(n, n-1)$ parity check code is the simple two-word $(n, 1)$ repetition code.

Although elegant and simple, this approach may sometimes lead to numerical problems in fixed point implementations and in these cases the approach based on puncturing a low rate mother code is still preferred.

F. Initialization of forward and backward recursions and windowing

For SISO on convolutional codes the initialization of forward and backward recursion, as well as the order in which these recursions are performed, leads to different solutions. In Figure 13 we show, pictorially, the most relevant approaches.

In the upper left part of the figure, we represent a possible solution when a single SISO processes a block of data. The data is divided into adjacent equally sized ”windows” and the SISO processes sequentially the set of windows in natural order. It first performs the forward recursion, storing the result in a temporary buffer and then performs a backward recursion and computation of the outputs at the same time. Since the SISO processes the windows sequentially and in natural order, the forward recursion results can be propagated to the next window for correct initialization. Backward recursion on the other hand needs to be properly initialized on each window and an additional unit must be deployed to perform this task (dashed lines).

Other scheduling possibilities, like performing first the backward and then the forward recursion are examined in [38] and give rise to similar overheads.

When parallel processor are used (top, right), initialization of the forward recursions are needed and an additional unit to perform this task must be deployed.

A more elegant and efficient solution that eliminates the overheads of initializations is reported in the bottom part of the figure. In this case we exploit the fact that SISO processors are employed in an *iterative* decoder. The results of the forward and backward recursion at a given iteration are propagated to the adjacent (previous or next) window in the next iteration. This approach leads to negligible performance losses provided that the window size is large enough.

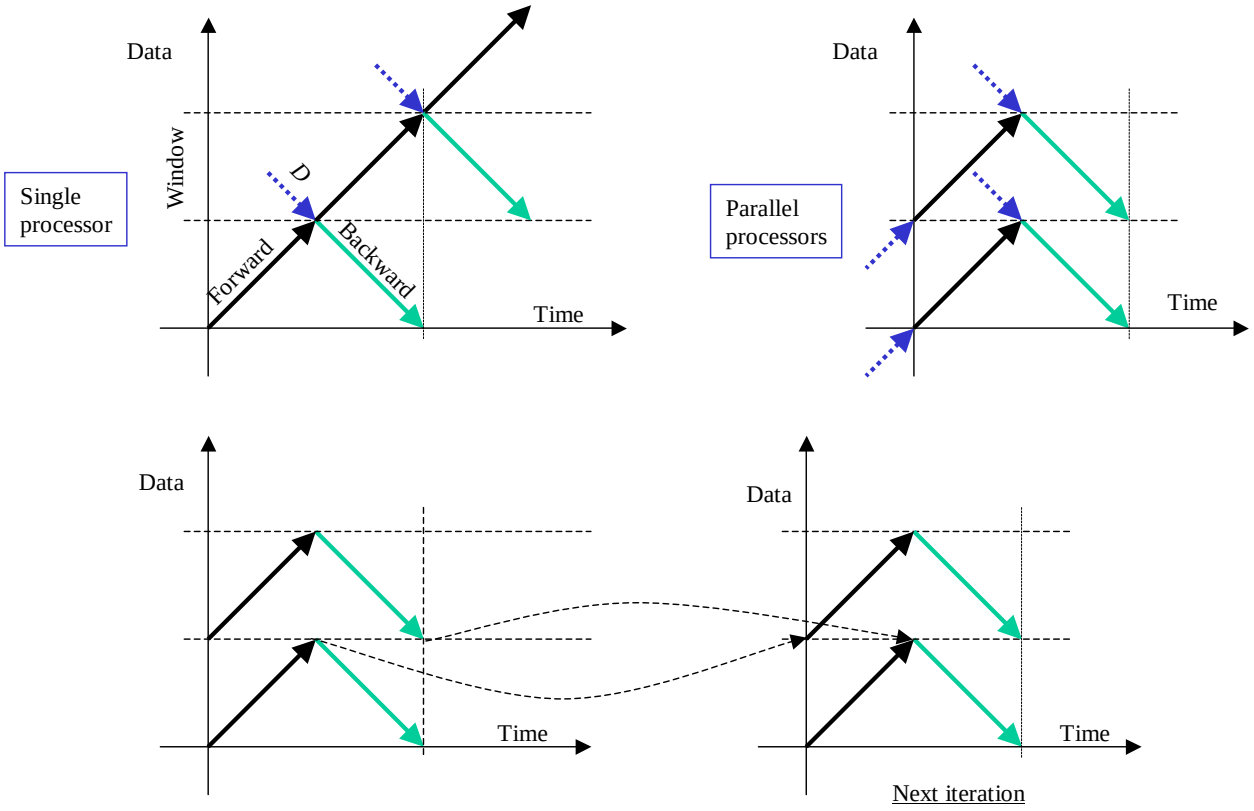


Fig. 13. Possible solutions for initialization of forward and backward recursions.

IV. Parallel architectures for turbo decoders

In this section, we consider the basic SISO decoder architecture with four processing units (branch metric, forward recursion unit, backward recursion unit and output computation unit, see Figure 21). This SISO is able to perform one trellis step during one clock cycle. The maximum throughput achievable by the turbo decoder using this SISO is $f_{\text{clk}}/\theta \cdot n_{it}$, where n_{it} is the number of iterations of the decoding process and f_{clk} is the clock frequency of the architecture. The factor θ indicates the minimum number of trellis stages per information bit ($\theta = 2$ for a PCCC turbo decoder, $\theta = 2 + \frac{1}{R_i}$ for a SCCC turbo decoder)[8].

There are three solutions to increase the throughput of the decoder: increasing the parallelism of the decoder, increasing the clock frequency and finally, decreasing the number of iterations. This last solution will be considered separately in V.

Theoretically, the increase of parallelism can be obtained at all the levels of the hierarchy of the turbo decoding algorithm, i.e., first at the turbo decoder level, second at the SISO level (duplication of the processing unit performing an iteration); third at the half iteration level (duplication of the hardware to speed up a SISO processing); and finally, at the trellis stage level.

A. Codeword pipe-line and parallelisation

The first method proposed in the literature to increase the throughput of a turbo decoder [2] is the simplest one: it dedicates a processor for each half iteration. Thus, the $2n_{it}$ processors work in a linear systolic way. While the first one processes the first half iteration of the newest received codeword of index k, the second one processes the second half iteration of the previous received codeword (index $k-1$), and so on, up to the $2n_{it}^{\text{th}}$ processor that performs the last iteration of the codeword of index $k - 2n_{it}$. Once the processing of a half iteration is finished, all codewords are shifted in the linear processor array. This method is efficient in the sense that the increase of throughput is proportional to the increase of the hardware: $2n_{it}$ processors working in parallel increase the throughput by a factor of $2n_{it}$. Another efficient alternative involves instanciating several turbo decoders working in parallel on independent codewords using demultiplexing.

Nevertheless, both methods imply a large latency of decoding and also require the duplication of the memories for extrinsic and intrinsic information. To avoid the memory duplication, it is far more efficient to use parallelism at the SISO level in order to speed up the execution time of an iteration.

B. Parallel SISO architecture

The idea is to use a parallelism to perform a SISO decoder, *i.e.*, use several independent SISO decoders working on the same codeword. To do so, the frame of size N ($N = k/R_o$ in the case of the inner code of the SCCC scheme, $N = k$ otherwise) is sliced into P slices of size $M = N/P$ and each slice is processed in parallel by P

[8]This maximum decoding rate is obtained when there is no idle cycle (to empty and/or initialized pipe-line) between two half iterations. In [39], a technique to obtain this condition is presented.

independent SISO decoders (in the following, we assume that the size N of the frame is a multiple of P). This technique implies two types of problems which should be solved: first, the problem of data dependency within an iteration; and second, the problem of the memory collisions in the parallel memory accesses.

Problem of data dependency: Since forward processing is performed sequentially from the first symbol u_0 to the last symbol u_{N-1}, the question arises of how to start simultaneously P independent forward recursions starting respectively from the symbols $u_0, u_{M-1}, u_{2M-1}, \ldots, u_{N-M-1}$? The same question also arises for the backward recursion in a symmetrical way. An elegant and simple solution to this problem was proposed independently by Blankenship *et al.* [40] and Giulietti *et al.* [41] in 2002. This solution is derived from the sliding window technique defined in Section III-D. The idea is to relax the constraint of performing the entire forward (respectively, backward) processing within a single iteration (see Figure 13). Thus, the P final state metrics of the forward processing of the P slices obtained during the j_{it}^{th} iteration are used as initial states, after a circular right shift, of the forward processing of iteration $j+1$ (j varying from 1 to n_{it}). The same principle also stands for the backward recursion.

Memory organization: In the natural order, the organization of the memory can be very simple: the first M data (u_0 to u_{M-1}) are stored in a first Memory Block (MB), then the next M data ($u_M, \ldots, u_{2M-1}$) are stored in a second MB and so on. This mapping is called direct mapping. With direct mapping, the processing of the first dimension is straightforward: each SISO unit has a direct access to its own memory. For the interleaved dimension, the problem is more complex. In fact, the first SISO needs to process sequentially the symbol $u_{\pi(1)}, u_{\pi(2)}, \ldots, u_{\pi(M-1)}$. Since at a given time l, $\pi(l)$ can be stored in any of the P memories, a network has to be created in order to connect the first SISO unit to the P memory banks. Similarly, a network should also be created to connect the P memory banks to the P SISO units. However, a conflict in the memory access can appear if, at a given time, two SISO units need to access data stored in the same memory bank.

The problem of memory conflict has been widely studied in the literature and so far, three kinds of solutions have been proposed: solution at the execution stage, solution at the compilation stage and solution at the design stage. These three solutions are described below.

Formulation of the problem: Since each stage of a trellis needs inputs (intrinsic information, *a priori* information and the associated redundancy) and generates an output (extrinsic and/or LLR information), the execution of P trellis stages at each clock cycle requires a memory bandwidth of P read/write accesses per clock cycle. Assuming that a memory bank can be read and written at each clock cFycle, at least P memory banks of size M are required to provide both the memory capacity and the memory bandwidth. Let us describe the sequence of read/write in the memory bank in the natural and interleaved order. At time l, the P symbols accessed by the P SISO processors are in the natural order $V_l^1 = \{l, l+M, \ldots, l+(P-1)M\}$ and in the interleaved order $V_l^2 = \{\pi(l), \pi(l+M), \ldots, \pi(l+(P-1)M)\}$. The memory organization should allow, for all $l = 1..M$, a parallel read/write access for both set V_l^1 and V_l^2. In the remainder of the paper, we assume that the memory address i corresponds to the bank $\lfloor \frac{i}{m} \rfloor$ at address ($i \mod m$) where $\lfloor . \rfloor$ denotes the integral part function.

The generic parallel architecture is shown in Figure 14. An iteration on this architecture works on two steps. When decoding the first encoder, the p^{th} memory bank is accessed thanks to its associated address generator (AG).

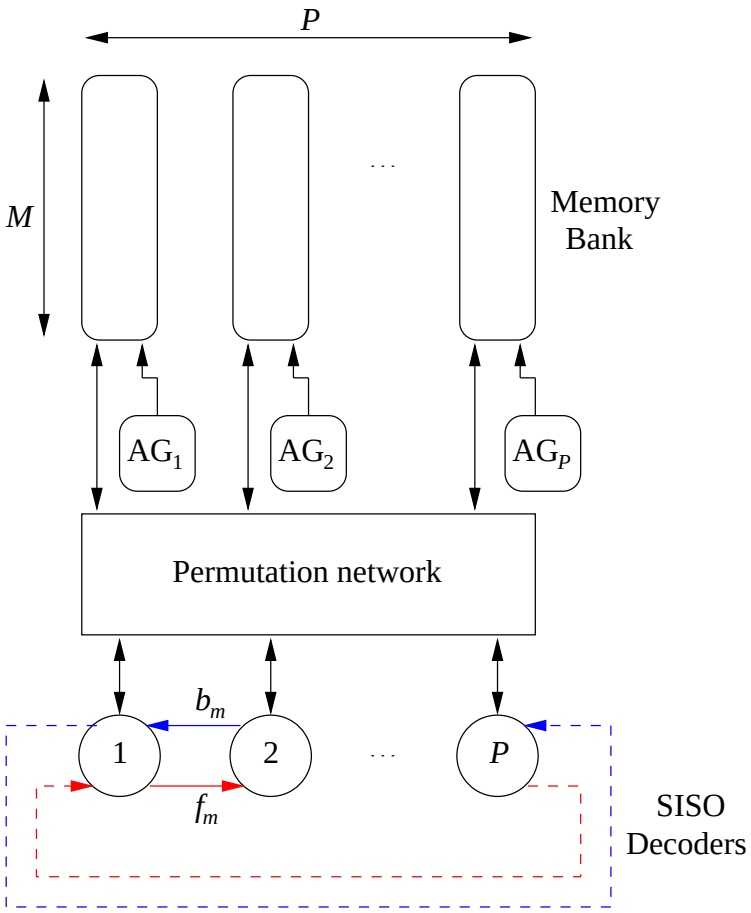


Fig. 14. Generic architecture of a parallel turbo decoder. At the end of each iteration, the final states f_m of forward processing (respectively backward processing b_m) are exchanged to the right (respectively, left) SISO decoder in a circular way. Parameter P represents the number of parallel SISO decoder. Parameter M represents the memory size of each slice.

The address generator delivers a sequence of addresses $\delta_p^1(l)_{l=1..M}$. The data coming from the memory banks are then sent to the SISO units through the permutation network. The permutation network shuffles the data according to a permutation defined at each cycle l by $\rho^1(l)$. The outputs of the SISO units are stored in the memory banks in a symmetrical way, thanks to the permutation network[9]. At the end of an iteration, the final forward recursion metrics f_m (respectively backward b_m) are sent to their right (respectively left) neighbor. Those metrics are stored temporarily during the processing of the second encoder. They are used as the initial state of the trellis at the beginning of the next iteration when the first encoder is processed again. Note that when tail-biting code is used, the left-most and right-most SISO also exchange their f_m and b_m values. The decoding of the second encoder is similar: the $\delta_p^1(l)_{l=1..M}$ are replaced by $\delta_p^2(l)_{l=1..M}$ and $\rho^1(l)$ by $\rho^2(l)$.

[9]The permutation network is composed in fact of two shuffle networks, one to shuffle the data between the SISOs and memory banks during a write access, the other one to shuffle the data during a read access between memory bank and SISO unit.

1) Solution at the execution stage: In this family of solutions, we encompass all solutions using direct mapping with some extra hardware or features to tackle the problem of memory conflict during the interleaved dimension processing. Two kinds of solutions have been proposed: the first one is technological, it uses memories with a bandwidth higher than necessary in order to avoid memory conflict. For example, if two read/write accesses in a memory cannot be performed in a single SISO clock cycle, then all double access memory conflicts are solved. Note that this solution is not efficient in terms of area and power dissipation. A more efficient solution was proposed by Thul *et al.* [42]. It relies on a "smart" permutation network that contains small FIFO modules to smooth the memory access. These FIFO modules allow a fraction of the write access to be delayed and a fraction of the read access to be anticipated so that, at each cycle, the whole memory bandwidth is used. In order to limit the size of the FIFO modules, Thul et al. also proposed to "freeze" the SISO module in order to solve the remaining memory conflict. This solution is generic and efficient but requires some additional hardware and extra latency in the decoding process [43].

2) Solution at the compilation stage: This solution was proposed by Tarable *et al.* [44]. The authors show that, regardless of the interleaver and the number P of SISO units, there is always a memory mapping free of read/write conflict access. A memory mapping is a permutation φ on the set $\{0..N-1\}$ that associates to the index l the memory location $i = \varphi(l)$. This non trivial mapping ($\varphi \neq Id$) implies non-trivial spatial permutations in both natural ($\rho^1(l) \neq Id, l = 0..M$) and interleaved order ($\rho^2(l) \neq Id, l = 0..M$). This method is general but has two main drawbacks: the complexity of the network and the amount of memory to store the different configurations of the network during the decoding process. In fact, the network should perform all possible permutations. It can be implemented in one stage by a crossbar at a cost of a very high area, or in several stages by an *ad hoc* network (a Benes network, for example [45]). In this later case, both complexity and latency of the network is multiplied by a factor 2 compared to the simple barrel shifter (see section IV-B.3). Moreover, memories are required to store the address generator sequences and the $2M$ permutations $\rho^1(l)_{l=1..M}$ and $\rho^2(l)_{l=1..M}$. Assuming a multiple frame, multiple rate decoder, the size of the memory to store each interleaver configuration may be prohibitive.

In conclusion to this sub-section, [44] proposes to deal with the problem of a turbo decoder of different size by defining a single interleaver of maximum size. Codes of shorter length are then generated by simply pruning the original interleaver. This type of interleaver is named *prunable collision-free interleaver*.

3) Solution at the design stage: The idea is to define jointly the interleaver and the architecture in order to solve *a priori* the memory conflict while keeping a simple architecture. Figure 14 presents the interleaver structure used to construct a parallel interleaver constrained by decoding parallelism.

With this kind of technique, memory mapping is the natural one, i.e., $\varphi = \text{Id}$. Thus, in the natural order, the spatial permutation is identity, $\rho_l^1 = \text{Id}$, and the temporal permutations are also identity, $(\delta_p^1 = \text{Id})_{p=1..M}$. In the interleaved order, spatial permutation at time l is simply a rotation of index $r(l)$, i.e., $\rho_l^2(p) = (p + r(l)) \mod P$. All the temporal permutations δ_p^2, for $p = 0..P-1$, are equal to a unique temporal permutation δ^2 (the same address is used for all memory blocks). Moreover, the expression of δ^2 can be computed on the flight as the sum of a linear congruence expression and a periodic offset: $\delta^2(l) = (a \cdot l + \Omega(l \mod \omega)) \mod M$, where a is the

linear factor prime with M, ω is the periodicity of the offset (generally, $\omega = 4$) and Ω is an array of size ω that contains the offset values.

At first glance, one could think that such constraints on the interleaver would lead to poor performance. On the contrary, the interleavers of this type are among the best known ones. Since the number of parameters to define an interleaver is low, an exhaustive search among all parameters configuration can be done. Note that the impulse method defined by Berrou [46] and its derivative [47] are efficient tools to optimize the construction of such interleavers.

One should note that the *almost regular permutation* [48] defined for the DVB-RCS standard (Digital Video Broadcast Return Channel Satellite), the *dithered relatively prime interleaver* [49] and the *slice turbo code* [50] belong to the family of "design stage interleaver".

Problem of activity of parallel architecture: However, another issue arises when dealing with parallel architectures: the efficient usage of computational resources. The idea is that it is inefficient to increase the number of SISOs if the computational power of theses SISOs is used only partially. This issue is particularly critical in the context of sequential decoding of turbo codes where half-iterations are processed sequentially. Due to data dependencies and pipeline stages in hardware implementation, the SW algorithm leads to an idle time at the beginning and end of each sub-block processing. This idle time can waste a significant part of the total processing power when the sub-block size is short, i.e., with short codeword length and/or high degree of parallelism. This problem is quite complex and not easy to solve in the general case. However it can be tackled by the use of a joint interleaver/decoder design (design stage solution) as proposed in [39], or going back to the pipeline solution of section IV-A, as proposed recently in [51] for very high speed turbo decoders.

C. Parallel trellis stage

In a given recursion (forward or backward), each trellis stage implies the computation, for each node of the trellis, of the recursive equations (13) and (14). It is easy to associate a processing unit to each node of the trellis so that a trellis stage can be processed at each clock cycle. The question is now: is it possible to increase the parallelism? Since the forward (or backward) recursion contains a loop, it is not possible to increase the parallelism directly. Some authors propose a technique, called trellis compacting. This technique is based on restructuring the conventional trellis by grouping two consecutive trellis stages in a single one. In other words, instead of serially computing the forward metric α_{l+1} from the α_l metrics and the branch metrics γ_l, and then computing α_{l+2} from α_{l+1} and γ_{l+1}, α_{l+2} is directly computed from α_l, γ_{l+1} and γ_l.

This technique was proposed initially in the context of the Viterbi decoder and speed improvement of a factor 1.7 was reported [52]. It can be directly applied when the max-log-map algorithm is implemented. In fact, in this case, forward and backward recursion are equivalent to the Viterbi recursion (the so-called Add Compare Select Unit). Trellis compaction can also be adapted, thanks to few approximations, when the log-MAP algorithm is implemented,

as shown in [53] and [54][10].

It is worth mentioning that trellis compaction leads to an equivalent trellis of the trellis of the double binary code [56]. This is one explanation, together with its good performance for medium and low rate code, of the success of double binary codes in several standards.

D. Increase of clock frequency

A direct way to increase the decoding throughput of a turbo decoder is to increase the clock frequency. To do that, the critical path of the turbo decoder should be reduced. A simple analysis shows that the BM units as well as the OCU units can be pipelined as needed. The critical path is in the forward or backward recursion loop. There are not so many solutions to reduce this path directly: use of a fast adder (at a cost of an increase of area), reduce the number of bits to code the forward and backward metrics (at a cost of a decreasing of the performance) and, if the log-map algorithm is implemented, delay of one cycle the addition of the correcting offset in order to reduce the critical path (see [38] for more details).

Another architecturally efficient solution for reducing this critical path consists of adding a pipe-line register in a recursion unit and then to interleave two (or more) independent SISOs on the same hardware. With the pipe-line register, the critical path can almost be reduced by two; thus, the hardware can operate at a double frequency compared to the direct solution. The over cost is low since only a single additional pipe line stage is introduced in the forward (or backward) metrics recursion loop after the first adder stage of Figure 17.

[10]One can note that treillis compaction is a special case of the general method presented in [55] to break the ACS bottleneck.

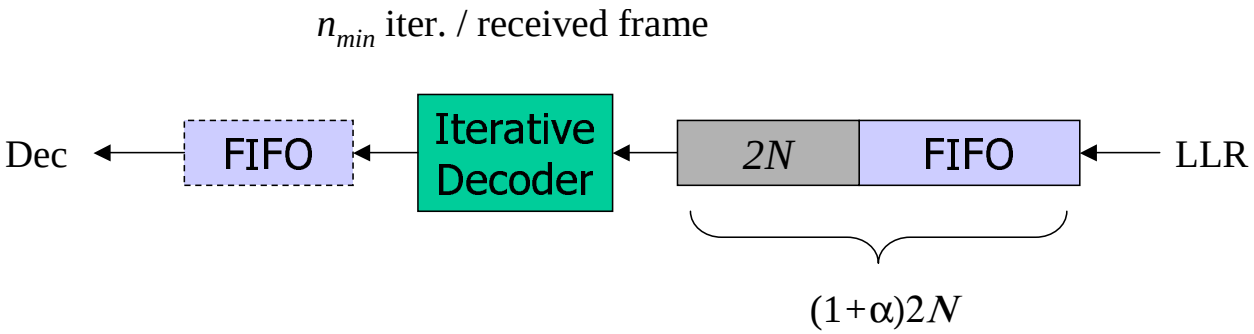


Fig. 15. Block diagram of an an iterative decoder with input and output FIFO to improve the throughput.

V. Stopping rules and buffering

An efficient way to increase the throughput of the decoder is to exploit the randomness of the number of required iterations when the decoder is embedded with some stopping criterion.

Assume as in Figure 15 that we have a decoder that is capable of performing $n_{\min} \geq 1$ iterations while receiving a frame. We add in front of it a FIFO buffer that has size $(1+\alpha)2N$, where N is the codeword size and $\alpha \geq 0$ is a constant that measures the memory overhead. If $\alpha = 0$ the decoder has no possibility of changing the number of iterations as the FIFO memory only stores the following frame while decoding the current one. If instead $\alpha > 0$ the time available for decoding ranges from $n_{\min}$ to $n_{\max} = (1+2\alpha)n_{\min}$ depending on the status of the FIFO.

In order to stop iterative decoding the decoder is embedded with a stopping rule so that the number of iterations for decoding is described through a (memoryless) random variable x with distribution $f(x)$. We represent the status of the FIFO as an integer ranging between $n_{\min}$ and $n_{\max}$ which is the number of *available iterations* at any given time. The transition probability matrix $\mathbf{P}$ of the underlying Markov chain has the following elements

$$p_{i\to j} = \begin{cases} 0, & i - j \leq -n_{\min}; \\ f^{+}(i-j+n_{\min}), & j = n_{\min}, i \geq n_{\min} \\ f^{-}(i-j+n_{\min}) & j = n_{\max}, i > n_{\max} - n_{\min}; \\ f(i-j+n_{\min}), & \text{otherwise.} \end{cases} \quad \forall i,j = n_{\min},\ldots,n_{\max} \tag{18}$$

where we have defined

$$f^{+}(x) \triangleq \sum_{k=x}^{\infty} f(k) \quad \text{and} \quad f^{-}(x) \triangleq \sum_{k=0}^{x} f(k)$$

This Markov chain is irreducible and aperiodic. Consequently, a steady-state probability vector, defined as

$$p_S \triangleq \lim_{n\to\infty} \mathbf{S}\mathbf{P}^n \ \ \forall \mathbf{S} \tag{19}$$

exists.

From the vector $p_S(x)$, which represents the probability of having x available iterations, the frame error probability is then computed as

$$P_F = \sum_{x=n_{\min}}^{n_{\max}} p_S(x) P_F(x) \tag{20}$$

where $P_F(x)$ is the frame error probability when x iterations are available. The steady state distribution p_S typically shows a rather sharp transition. As a consequence, when the average number of iterations $\bar{n}$ is below the minimum number of available iterations $n_{\min}$ the FIFO is typically empty and

$$P_F \approx P_F(n_{\max}).$$

On the other side if $\bar{n} > n_{\min}$ the FIFO is typically full and

$$P_F \approx P_F(n_{\min}).$$

The procedure to design a FIFO buffer is then the following

1) Fix a stopping criterion choosing from one of those listed in the following Section V-A.
2) Run a single simulation with an large (infinite) number of iterations. In this simulation carry on, for each desired E_b/N_0, the statistics of the number of iterations required $f(x)$ and frame error probability having x available iterations $P_F(x)$.
3) For all desired pairs $(n_{\min}, n_{\max})$ compute the matrix $\mathbf{P}$ through (18) and from $\mathbf{P}$ the steady state distribution of the FIFO $p_S(x)$ using (19). From the steady state distribution compute the frame error probability as in (20).
4) Generally, the decoder speed must have $n_{\min}$ slightly larger than the average required for the chosen stopping rule at the target FER. The maximum number of iterations $n_{\max}$ must be set according to the desired target number of iterations. The memory overhead α is then obtained as

$$\alpha = \frac{1}{2}\left(\frac{n_{\max}}{n_{\min}} - 1\right)$$

A. List of stopping rules

Several stopping rules have been proposed in the literature, see for example [57], [58], [59], [60], [61], [62], and references therein. In the following we list the most efficient and simple rules

1) **Hard rule 1:** The signs of the LLRs at the input and at the output of a constituent SISO module are compared and the iterative decoder is stopped if all signs agree. Note that the output of a SISO is always an *extrinsic* LLR.
2) **Hard rule 2:** To improve the reliability of the stop rule, the previous check has to be passed for two successive iterations
3) **Soft rule 1:** The minimum absolute value of all the extrinsic LLR at the output of a SISO is compared against a threshold. Increasing the threshold increases the reliability of the rule and also increases the average number of iterations.
4) **Soft rule 2:** The minimum absolute value of all the *total LLR* is compared against a threshold. Note that the total LLR is the sum of the input and output LLR of a SISO module.

The choice of the stopping rule and possibly of the corresponding threshold for soft rules, is mainly dictated by the complexity of its evaluation and by the probability of false stops that induce an error floor in the performance, for more details see [57].

In Figure 16 we report the FER performance of a rate 0.35 SCCC with 4 state constituent encoders and interleaver size of 8,640 with fixed number of iterations and those obtained with the structure of Figure 15 with the first hard stopping rule.

Solid lines refer to the decoder with a fixed number of iterations (5,6,7,10), while dashed curves are the performance obtained using stopping rules and finite buffering. For these curves n_{max} has been kept fixed to 10 and $n_{\min}$ takes the label values 5,6,7 and 10. As anticipated, it can be seen that all the dashed curves start from the performance of the correspondent $n_{\min}$ and at a given point they start to converge to the performance relative to $n_{\max}$ iterations. The threshold point corresponds to the situation for which $\bar{n} \approx n_{\min}$. Note that since the average number of iterations decreases with the signal to noise ratio the maximum gap between the curves is obtained at low values of signal-to-noise ratio.

The pair $(10, 6)$, corresponding to a memory overhead of α=33% and a speed-up of 66%, shows a maximum penalty of 0.15 dB at FER=10^{-1} that becomes 0.01 dB at 10^{-5}. The pair $(10, 5)$, corresponding to a memory overhead of 50% and a speed-up of 100%, shows instead a maximum penalty of 0.3 dB at FER=10^{-1} that becomes 0.1 dB at 10^{-5}.

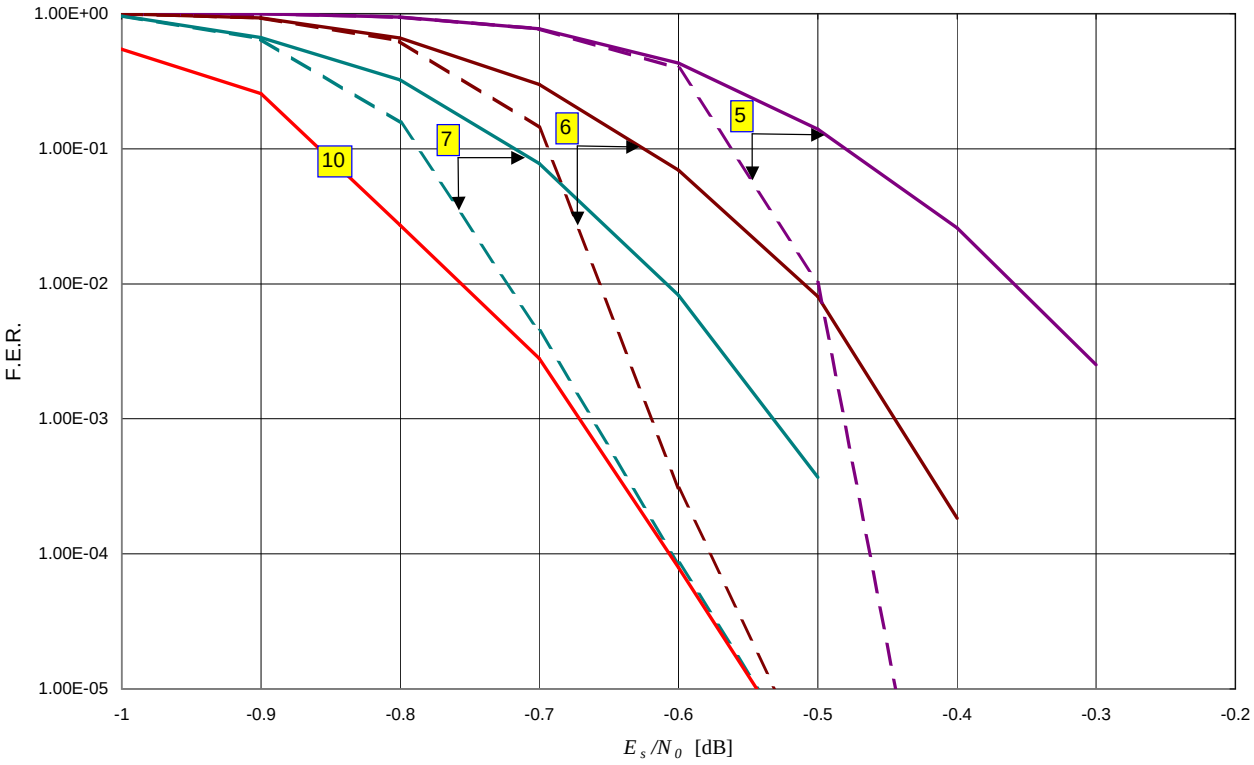


Fig. 16. Comparison of FER performance of an SCCC with 4 state constituent encoders and interleaver size of 8,640 with fixed number of iterations with those obtained using the structure of Figure 15 and the first hard stopping rule.

VI. Quantization issues in turbo decoders

The problem of fixed-point implementation is an important one since hardware complexity increases linearly with the internal bit width representation of the data. The trade-off can be formulated as follows: what is the minimum bit width internal representation that leads to an acceptable degradation of performance. It is interesting to note that the very function of the turbo-decoding process is the suppression of channel noise. This robustness against channel noise implies also, as a benecifial side effect, a robustness against the internal quantification noise. Thus, compared to a classical DSP application, the internal precision of a turbo-decoder can be very low without significant degradation of the performance.

In this section, we will give a brief survey of the problem of fixed-point implementation in the case of a binary turbo-decoder. First, we will discuss the problem of optimal quantization of the input signal and the resulting internal precision. Then, the problem of the scaling of the extrinsic message will be discussed. Finally, we will conclude this chapter by giving a not yet published pragmatic method to optimize the complexity versus performance trade-off of a turbo-decoder.

A. Internal precision of a turbo decoder

Let us consider a binary turbo code associated with a BPSK modulation. The received symbol at time l is thus equal to $\bar{x}_l = x_l + w_l$, where x_l equals to -1 if $c_l = 0$, and equals 1 otherwise, and w is white gaussien noise of variance σ. The LLR $\lambda(c_l; I)$ is then equal to

$$\lambda(c_l; I) = 2\bar{x}/\sigma^2. \tag{21}$$

The quantization of $\lambda(c_l; I)$ on b_{LLR} bits is a key issue that impacts both the performance and the complexity of the design. In the following, we will assume that the quantized value $\lambda(c; I)_Q$ of $\lambda(c; I)$ on b_{LLR} bit is given by $\lambda(c; I)_Q = Q(\lambda(c; I))$, where the quantification function Q is defined as:

$$x_Q = Q(x) = sat(\left\lfloor x \cdot \frac{2^{b_{LLR}-1} - 1}{A} + 0.5 \right\rfloor, 2^{b_{LLR}-1} - 1) \tag{22}$$

where $sat(a, b) = a$ if a belongs to $[-b, b]$, and $sat(a, b) = sign(a) \times b$ otherwise and A is the interval dynamic of the quantization (data are quantized between $[-A, A]$). Symmetrical quantization is needed to avoid giving a systematic advantage to one bit value against an other. In fact, such a situation would decrease the performance of the turbo decoder.

One can note that if A is very large, most of the input will be quantized by a 0 value, *i.e.*, an erasure. In that case, the decoding process will fail. On the other hand, a too small value of A would lead to a saturation most of the time and the soft quantization would thus be equivalent to a hard decision. Clearly, for a given code rate and a given SNR, there is an optimal value of A. As a rule of thumb, for a 1/2 rate turbo code, the optimal value of A is around 1.2. Equations (21) and (22) show that the input value $\lambda(c; I)_Q$ of the turbo decoder depends on the channel observation, the value A, as mentioned above, but also on the variance of the SNR of the signal, *i.e.*, the variance

of the noise σ^2. The SNR is not available at the receiver level and it should be estimated using the statistic of the input signal. In practical cases, the estimation of the SNR is not mandatory. When the max-log-map algorithm is used, i.e., the $\max^*$ operation is approximated by the simple $\max$ operator, its estimation is not necessary. In fact, the term $\frac{2}{\sigma^2}$ is just, in the logarithm domain, an offset factor that impacts both input and output of the max unit. When the log-MAP algorithm is used, a pragmatic solution is to replace the real standard deviation σ of (21) by the maximum value σ_o leading to a BER (or a FER) acceptable for the application. Note that when the effective noise variance σ is below σ_o, the decoding process becomes sub-optimal due to a sub-estimation of the $\lambda(c;I)$. Nevertheless, the BER (or the FER) would still decrease and thus remains in the functionality domain of the application.

The number of bits b_{ext} to code the extrinsic message can be deduced from b_{LLR}. In many reported publications, $b_{ext} = b_{LLR} + 1$, i.e., extrinsic messages are quantified in the interval $[-2A, 2A]$. Note that if the OCU delivers a binary value out of the range $[-2^{b_{ext}} + 1, 2^{b_{ext}} - 1]$, a saturation needs to be performed.

Once b_{LLR} and b_{ext} are chosen, the number of bits b_{fm} to code the forward recursion metrics α and the backward recursion nodes β can be derived automatically. In the following, we will only consider the case of the forward recursion metrics α. The same results stand also for the backward recursion unit.

According to (11) and (12), in the logarithm domain, the branch metric γ_l is a finite sum of bounded values. Let us assume that Γ is a bound of the absolute value of the branch metric γ_l (Γ is a function of the code, b_{LLR} and b_{ext}). Then, it is shown in [63] that at any time l[11]:

$$\max_s(\alpha_l(s)) - \min_s(\alpha_l(s)) < \nu \cdot \Gamma \tag{23}$$

where the parameter ν is the memory depth of the convolutional encoder.

Assuming that $\min_s(\alpha_l(s))$ is maintained equal to zero by dedicated hardware, then (23) shows that $b_{fm} = \lceil log_2(\nu \times \Gamma) \rceil$ bits are sufficient to code the α metrics. In practice, this solution is not efficient. In fact, maintaining $\min_s(\alpha_l(s))$ equal to zero implies additional hardware in the recursion loop to perform the determination of $\min_s(\alpha_l(s))$ and to subtract it from all the metrics. This hardware increases both the complexity and the critical path of the FR unit.

A more efficient solution is to replace those complex systematic operations by the subtraction of a fixed value when needed. Let us defined Δ as $\Delta = \lceil log_2((\nu + 1) \cdot \Gamma) \rceil$. Then, coding the forward metric on $b_{fm} = 1 + \Delta$ leads to a very simple scheme. In fact, since Γ is the maximum dynamic of the branch metric, (23) gives: $\max_s(\alpha_{l+1}(s)) - \min_{s'}(\alpha_l(s')) \leq (\nu + 1) \cdot \Gamma$.

This equation proves that, if at time l, $\min_s(\alpha_l(s))$ is below 2^Δ, then at time $l+1$, $\max_s(\alpha_{l+1}(s))$ also remains below $2^{1+\Delta} - 1 = 2^{b_{fm}} - 1$. If at time $l+1$, $\min_s(\alpha_l(s))$ becomes above 2^Δ, then all the forward metric will range between 2^Δ and $2^{\Delta+1}$. In that case, a rescaling operation is performed by subtracting 2^Δ from all the forward metrics. This situation is detected thanks to an AND gate connected to the Most Significant Bit (MSB) of the α

[11]the exact bound is derived in [38]

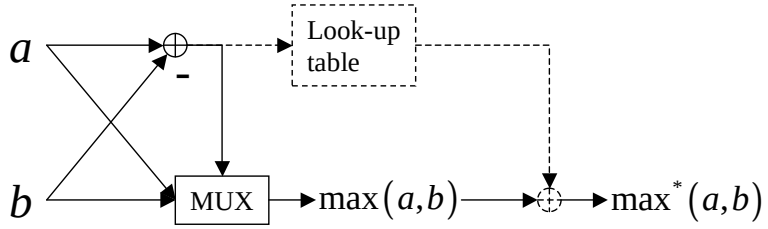


Fig. 17. Block diagram for the $\max^*$ operator.

metrics. The subtraction is simply realized by setting all the MSB to 0. Thanks to the rescaling process, the dynamic to represent the increasing function of the forward metric is very limited. Note that other efficient methods have been proposed in the literature, like more elaborated rescaling techniques or simply avoiding the rescaling operation using modulus arithmetic [64].

Typical values of b_{LLR} are between 3 to 6 bits. For an 8 state turbo code, the corresponding b_{fm} are around 7 to 10 bits. An example of the impact of b_{LLR} on performance is given in Figure 19. The reader can find in [65] more deeper analysis of the bit width precision of the turbo-decoder.

B. Practical implementation of the max^* algorithm

In figure 17, we report a block diagram of the implementation of the fundamental associative operator $\max^*$ according to its definition (6). The look-up table performs the computation of the correcting factor given by (24):

$$f(x) = ln(1 + e^{-|x|})) \tag{24}$$

Figure 18 shows the plot of the function $f(x)$. The maximum value of this function is $ln(2)$ and the function decreases rapidly toward 0. In hardware, all real values are replaced by integer values thanks to (22). Thus, the maximum quantized value of f is given by $Q(ln(2))$. Moreover, from (22), it is possible to compute the maximum integer m_Q so that $x_Q > m_Q$ implies $f_Q(x_Q) = 0$. As an example, for $M = 1.2$ and $b_{LRR} = 4$, the maximum quantized value of f_Q is 4 (thus f_Q takes its values between 0 and 4 and requires 3 bits to be coded) and $m_Q = 14$ (see Figure 18). The hardware realization of the computation of the offset factor is thus simple. A first test is performed to determine if $|x_Q|$ is below 15. If the test is positive, the 5 Least Significant Bits (LSB) of x_Q are used as input of a 3-bit output look-up table that contained the precomputed values of $f_Q(x_Q)$. Otherwise, the offset is set to 0. Note that it is also possible to compute the absolute value of x_Q before the access to the LUT in order to reduce its size by half.

There are many other different implementations of the offset function in the literature. For example, [66] proposes a linear approximation of the $f(x)$ function while [67] proposes a very coarse, but efficient, approximation. When the offset factor is omitted, the hardware is simplified at a cost of a performance degradation. This point is discussed in the next section. Note that, when the offset factor is omitted, the algorithm is named in the literature the min-sum or the max-log-MAP algorithm.

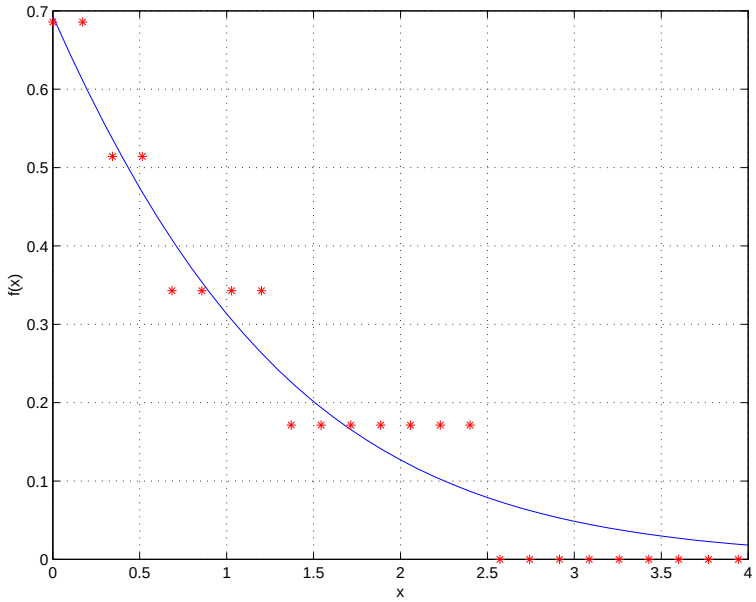


Fig. 18. Plot of the function $f(x)$. The quantized version of this function for $M = 1.2$ and $b_{LLR} = 4$ is also given (red dot points).

C. Rescaling of the extrinsic message

The use of the max-log-MAP algorithm leads to an overestimation of the extrinsic message and thus, decreases the performances of the turbo decoder by approximatively 0.5 dB. This penalty can be significantly reduced if the over-estimation of the extrinsic message is compensated, on average, by a systematic scaling down of the extrinsic message between two consecutive half iterations. This scaling is performed thanks to a multiplication by a scaling factor a_i, where the value of a_i depends on the number of the half iteration. Typically, during the first iteration, the value of a_i is low (around 0.5), and increases up to 1 for the two last half iterations. This technique, for a classical turbo decoder, reduces the degradation from 0.5 dB to 0.2 dB. For a double binary code, it reduces the degradation from 0.5 dB to 0.05 dB. Note that the scaling factor is also favorable with respect to the decorrelation of the extrinsic messages. Even if we use the log-MAP algorithm, the scaling factors help the decoder to converge.

D. Method of optimization

As seen above, many parameters impact both the performance and the complexity of the turbo decoder: number of bits of quantization, maximum number of iterations, scaling factors of the extrinsic message, value of the quantization dynamic M and also, the targeted BER and FER. For example, Figure 19 shows the bit error rate (BER) obtained for a turbo code of rate 1/2 and size $N = 512$ decoded for different numbers of iteration and different values of b_{LLR}.

All those parameters interact in a non linear way (see the effect of n_{it} and b_{LLR} in Figure 19 for example). The problem of finding a good performance-complexity trade-off is thus a complex task. Moreover, the evaluation

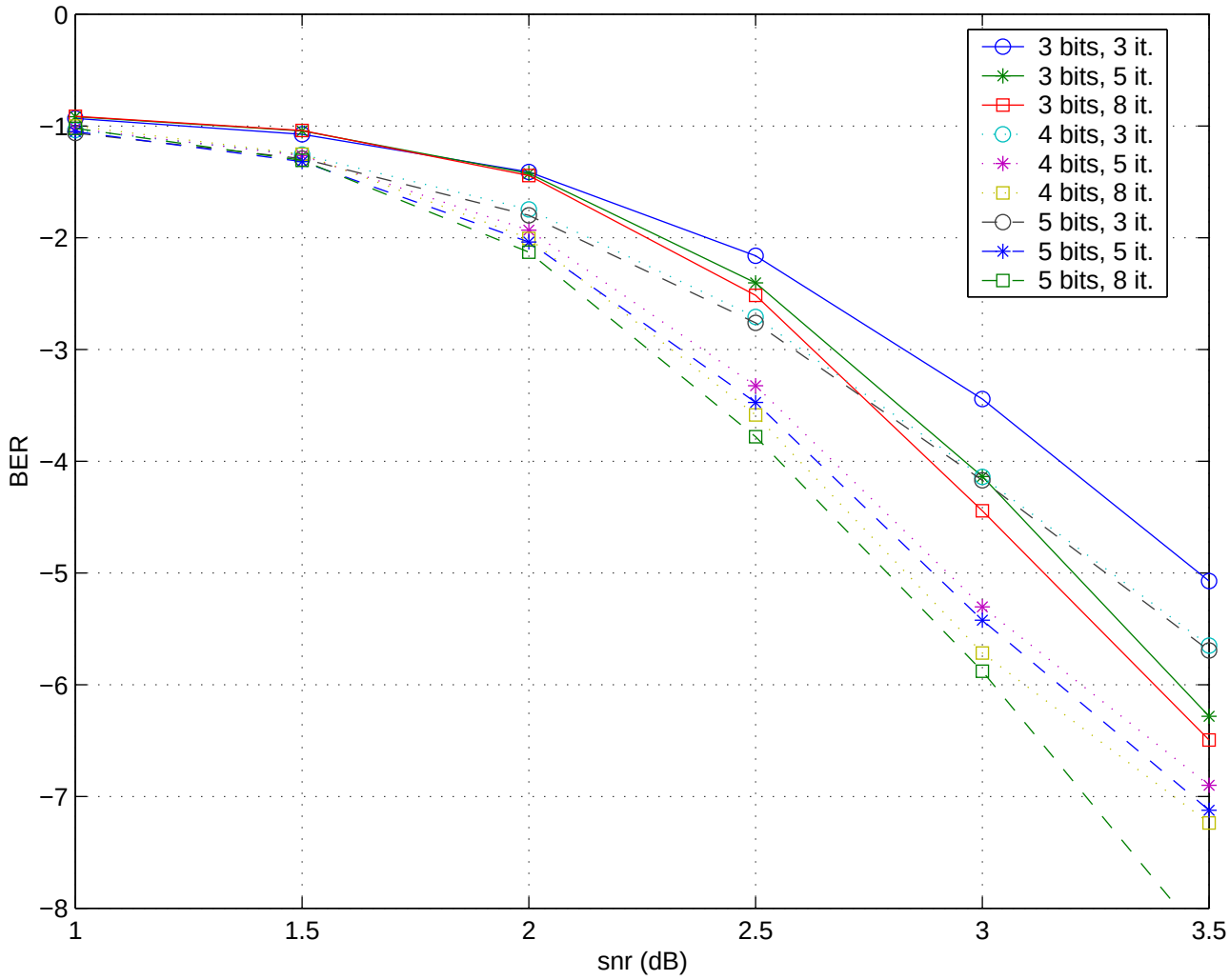


Fig. 19. Curve BER = f(SNR) for a 2/3 rate turbo code of length $N = 512$ for different b_{LLR} values and number of decoding iterations. The value of A is equal to 1.39.

of each configuration generally requires a CPU extensive Monte-Carlo simulation in order to have an accurate estimation of the BER. In order to avoid such simulation, we propose an efficient pragmatic method:

- **step 1**: Define the space of the search by defining the range of search for each parameter of the decoder. Define a model of complexity for each parameter[12]. Define also the maximum allowable complexity of the design.
- **step 2**: Define the "worst case" configuration by individually setting the parameters to the value that degrades performance most.
- **step 3**: Using this configuration, perform a Monte Carlo simulation at the SNR of interest. Each time a received

[12]See [31] and VII for an example of modelization of the hardware complexity of a turbo decoder

codeword fails to be decoded, store the codeword (or information to reconstruct it, i.e., the seeds of the pseudo random generators) in a set S. Stop the process when the cardinality of the set S is high enough (typically around 1000). Note that this operation can be very CPU consuming but it has to be done only once.

- **step 4**: Perform an optimization in order to find the set of parameters that minimize the BER (or the FER) over the set S with the constraint that the overall complexity of the decoder remains below a given value.
- **step 5**: Perform *a normal* Monte Carlo simulation in order to verify *a posteriori* the real performance of the selected parameters. Go to step 1 with a different scenario of optimization if needed.

This method is efficient since the test of one configuration can be a few orders faster than the direct method. For example, for a FER of 10^{-4}, the test of a configuration with a classical Monte-Carlo simulation requires, on average, the simulation of 10^6 codewords. In contrast, with the proposed method, testing a new configuration requires only the decoding of 10^3 codewords. An improvement of simulation speed by a factor of 10^3 is then obtained.

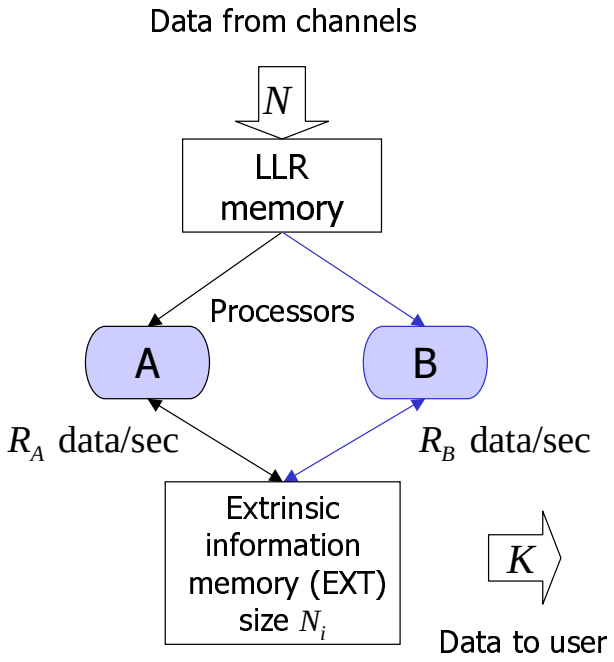


Fig. 20. General architecture of an iterative decoder. Shaded blocks are processors, white blocks are memories.

VII. Evaluation of the complexity of iterative decoders

In order to provide a high level evaluation of the complexity of the iterative decoders we will use as reference the architecture reported in Figure 20. Constituent encoders are assumed to be binary and messages are stored in the form of LLRs.

The iterative decoders are generally built around two sets of processors and two memories. The messages coming from the channel are stored in a buffer as they will be accessed several times during the iterative process. The extrinsic messages are instead stored in a temporary memory denoted by EXT in the figure.

The high level algorithm of the decoder can be summarized with the following steps:

1) initialize the inner memory to null messages,
2) apply the first set of constraints A using the EXT and LLR, write the updated messages in EXT,
3) apply the second set of constraints B using the EXT and LLR, write the updated messages in EXT,
4) iterate until some stopping criterion is satisfied or the maximum number of iterations is reached.

The application of constraints A and B here are executed serially. Different scheduling can be applied, especially for LDPC, but this is the common approach.

As we have seen in section IV PCCC, as well as SCCC and LDPC, admits efficient highly parallel structures so that the throughput can be arbitrarily increased by increasing the number of parallel processors without requiring additional memory. The trade-off between area and throughput of the decoder are then fully under designer control. In this section we will focus on C, defined as the *number of elementary operations required for decoding one information bit per iteration* as a function of the main design parameters of the code.

The throughput T of the implemented decoder can be well approximated by

$$T = f\frac{C_{\text{dep}}}{N_{\text{it}}C}$$

where C_{dep} is the number of deployed operators running at frequency f, and N_{it} is the number of required iterations.

For iterative decoders with messages in the form of LLR, the evaluation of computational complexity can be expressed in terms of number of sums and $\max^*$ operations. Note that $\max$ will substitute $\max^*$ when the max-sum version of SISO processors is used instead of the $\max^*$-sum version.

A. LDPC

For LDPC, each variable node processor with degree d_v requires $2d_v$ sums to compute the updated message. Thus, summing up over all possible nodes, the variable node processing requires 2 sums per edge.

For check nodes, the number of operations needed for updating the EXT messages depends linearly on the check degree d_c with a factor that depends on the used approximation. In [68] a comprehensive summary of the available optimal and approximated techniques for check node updates is presented, together with their correspondent complexity and performance trade-offs. Here we will assume the optimal and numerical stable algorithm (6) and (7) in [68] that requires $6(d_c - 2)$ $\max^*$ operators for a check node of degree d_c. The reader is warned however that the factor 6 can be reduced by using other sub-optimal techniques.

Summing up over the set of variable and check nodes we get the following complexity

$$C = \begin{cases} \frac{2\eta}{R}, & \text{sums;} \\ 6\left(\frac{\eta-2}{R}+2\right), & \max^* \end{cases} \tag{25}$$

per decoded bit and per iteration. In (25) we introduced the fundamental parameter η that is the average variable degree which measures the density of the LDPC parity check matrix. R is the rate of the code.

In Table I we show the normalized complexity C required for some values of the two relevant parameters η and R.

Note that LDPC decoders have a complexity that is inversely proportional to the rate of the code and proportional to the parameter η. The parameter η has also an impact on the performance of the code and takes typical values in the range $3 - 5$.

B. PCCC and SCCC

The complexity of PCCC and SCCC is strictly related to the complexity of their constituent SISO decoders. Here we consider rate k/n binary constituent encoders. In Figure 21 we report a block diagram of the SISO module showing its basic architecture according to the algorithm described in Section III. In the figure we have reported the sections of the architecture corresponding to the four operations reported in the algorithm described in Section III-D. Light blocks are computation blocks, while dark block refers to memory, which can be organized as LIFO or FIFO RAM. In this SISO structure, we have assumed that initialization of forward and backward state metrics are propagated across iterations so that no overhead is required for this operation.

R	0.10		0.20		0.30		0.40		0.50		0.60		0.70		0.80		0.90	
η	sum	max*	sum	max*	sum	max*	sum	max*	sum	max*	sum	max*	sum	max*	sum	max*	sum	max*
2	40	12	20	12	13	12	10	12	8	12	7	12	6	12	5	12	4	12
2.2	44	24	22	18	15	16	11	15	9	14	7	14	6	14	6	14	5	13
2.4	48	36	24	24	16	20	12	18	10	17	8	16	7	15	6	15	5	15
2.6	52	48	26	30	17	24	13	21	10	19	9	18	7	17	7	17	6	16
2.8	56	60	28	36	19	28	14	24	11	22	9	20	8	19	7	18	6	17
3	60	72	30	42	20	32	15	27	12	24	10	22	9	21	8	20	7	19
3.2	64	84	32	48	21	36	16	30	13	26	11	24	9	22	8	21	7	20
3.4	68	96	34	54	23	40	17	33	14	29	11	26	10	24	9	23	8	21
3.6	72	108	36	60	24	44	18	36	14	31	12	28	10	26	9	24	8	23
3.8	76	120	38	66	25	48	19	39	15	34	13	30	11	27	10	26	8	24
4	80	132	40	72	27	52	20	42	16	36	13	32	11	29	10	27	9	25
4.2	84	144	42	78	28	56	21	45	17	38	14	34	12	31	11	29	9	27
4.4	88	156	44	84	29	60	22	48	18	41	15	36	13	33	11	30	10	28
4.6	92	168	46	90	31	64	23	51	18	43	15	38	13	34	12	32	10	29
4.8	96	180	48	96	32	68	24	54	19	46	16	40	14	36	12	33	11	31
5	100	192	50	102	33	72	25	57	20	48	17	42	14	38	13	35	11	32
5.2	104	204	52	108	35	76	26	60	21	50	17	44	15	39	13	36	12	33
5.4	108	216	54	114	36	80	27	63	22	53	18	46	15	41	14	38	12	35
5.6	112	228	56	120	37	84	28	66	22	55	19	48	16	43	14	39	12	36
5.8	116	240	58	126	39	88	29	69	23	58	19	50	17	45	15	41	13	37
6	120	252	60	132	40	92	30	72	24	60	20	52	17	46	15	42	13	39
6.2	124	264	62	138	41	96	31	75	25	62	21	54	18	48	16	44	14	40
6.4	128	276	64	144	43	100	32	78	26	65	21	56	18	50	16	45	14	41
6.6	132	288	66	150	44	104	33	81	26	67	22	58	19	51	17	47	15	43
6.8	136	300	68	156	45	108	34	84	27	70	23	60	19	53	17	48	15	44
7	140	312	70	162	47	112	35	87	28	72	23	62	20	55	18	50	16	45

TABLE I

COMPLEXITY OF LDPC DECODER IN NUMBER OF ELEMENTARY OPERATIONS PER INFORMATION BIT AND PER ITERATION,

We will consider two versions of SISO. The first (inner SISO), which is used in PCCC and as the inner SISO for SCCC, gets messages on information and coded bits and provides messages on input bits. The second (outer SISO), which is used as outer SISO in SCCC, gets messages only on coded bits and provides updated messages on both information (user's data) and coded bits. In Figure 21 we can identify the following units:

- The Branch Metric Computer (**BMC**) is responsible for computing the LR or LLR to be associated with each trellis edge section according to (11) and (12). The number of sums required is $2^n - 1 - n$ for the outer SISO and $2^n - n - 1 + k$ for inner SISO
- The Forward Recursion computer (**FR**) is responsible for computing the forward recursion metrics A according to (13). Each state metric update requires $2^k - 1$ $\max^*$ between the incoming edges. The metrics of the $2^k N_s$ edges are obtained summing the previously computed branch metrics with each the path metrics ($2^k N_s$ sums). $\max^*$
- The Backward Recursion computer (**BR**), identical to forward recursion, is responsible for computing the backward recursion metrics B according to (14). As the recursion proceeds in the backward direction the input

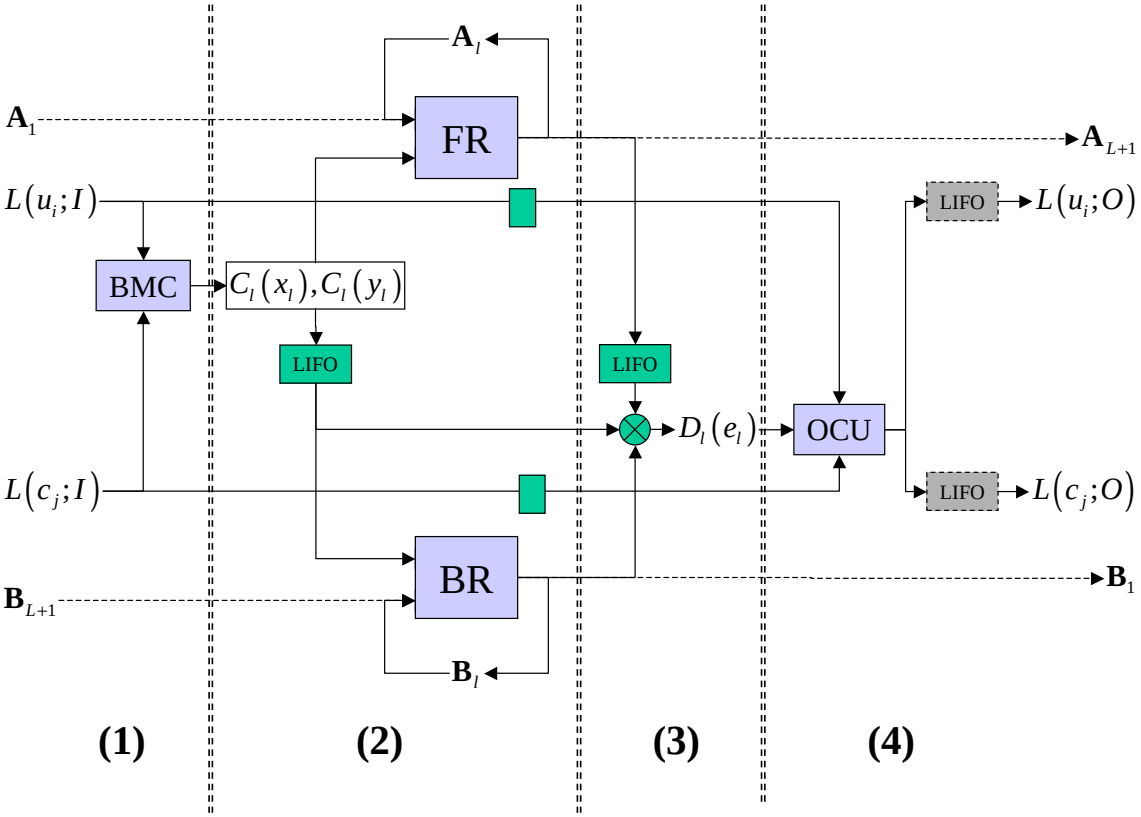


Fig. 21. General architecture of SISO module. The module uses 4 logic units (FR, BR, OCU and BMC). The sections of the block diagram refer to the 4 main steps of the algorithm described in Section III.

branch metrics must be reversed in time and this is the reason of the LIFO in front of it.

- The LR edges computed according to (3) require both the forward and backward metrics and the branch metrics. As backward metrics are provided in reversed order, a LIFO may also be inserted on the line coming from the FR. Note that the edge LF are then produced in reversed time order. For both FR and BR as well as inner and outer SISO the complexity is $2^k N_s$ sums and $(2^k - 1)N_s$ $\max^*$.
- The Output Computation Unit (**OCU**) computes the *a posteriori* LR applying (15) and/or (16) depending on the needs. The input LR are used to compute *extrinsic information* An efficient algorithm to compute the updated messages requires $2^k N_s + 2n + k$ sums and $2^k N_s + 2^{n+1} - 2n - 4$ $\max^*$. As the edges LR are provided in reversed time order, also the computed LR are reversed in order so that a LIFO may be necessary to provide the updated LR in the same order of the inputs. The correct ordering of the messages however can also be solved implicitly when storing them in the RAM.

In Table II we give the summary of the operations required for the inner and outer SISO while in Table III we report the numerical values for some typical values of k, n and N_s.

Having determined the complexity per information bit for the inner and outer SISO decoders C_I and C_O, the

	Outer SISO		Inner SISO	
Unit	Sum	$\max^*$	Sum	$\max^*$
Branch metric computation	$2^n - 1 - n$	-	$2^n - 1 - n + k$	-
Forward recursion	$2^k N_s$	$(2^k - 1)N_s$	$2^k N_s$	$(2^k - 1)N_s$
Backward recursion	$2^k N_s$	$(2^k - 1)N_s$	$2^k N_s$	$(2^k - 1)N_s$
Output computation	$2^k N_s + 2n + k$	$2^k N_s + 2^{n+1} - 2n - 4$	$2^k N_s + 2k$	$2^k N_s + 2^{k+1} - 2k - 4$

TABLE II

SUMMARY OF COMPLEXITY PER TRELLIS STEP FOR THE SISO ALGORITHM ON BINARY LLR.

			Outer							Inner						
			Sum				max*			Sum				max*		
k	*n*	*Ns*	BMC	FR+BR	OCU	Total	FR+BR	OCU	Total	BMC	FR+BR	OCU	Total	FR+BR	OCU	Total
1	2	2	1	8	9	**18.0**	4	4	**8.0**	2	8	6	**16.0**	4	2	**6.0**
1	2	4	1	16	13	**30.0**	8	8	**16.0**	2	16	10	**28.0**	8	6	**14.0**
1	2	8	1	32	21	**54.0**	16	16	**32.0**	2	32	18	**52.0**	16	14	**30.0**
1	2	16	1	64	37	**102.0**	32	32	**64.0**	2	64	34	**100.0**	32	30	**62.0**
1	2	32	1	128	69	**198.0**	64	64	**128.0**	2	128	66	**196.0**	64	62	**126.0**
1	3	2	4	8	11	**23.0**	4	10	**14.0**	5	8	6	**19.0**	4	2	**6.0**
1	3	4	4	16	15	**35.0**	8	14	**22.0**	5	16	10	**31.0**	8	6	**14.0**
1	3	8	4	32	23	**59.0**	16	22	**38.0**	5	32	18	**55.0**	16	14	**30.0**
1	3	16	4	64	39	**107.0**	32	38	**70.0**	5	64	34	**103.0**	32	30	**62.0**
1	3	32	4	128	71	**203.0**	64	70	**134.0**	5	128	66	**199.0**	64	62	**126.0**
2	3	2	4	16	16	**18.0**	12	14	**13.0**	6	16	12	**17.0**	12	8	**10.0**
2	3	4	4	32	24	**30.0**	24	22	**23.0**	6	32	20	**29.0**	24	16	**20.0**
2	3	8	4	64	40	**54.0**	48	38	**43.0**	6	64	36	**53.0**	48	32	**40.0**
2	3	16	4	128	72	**102.0**	96	70	**83.0**	6	128	68	**101.0**	96	64	**80.0**
2	3	32	4	256	136	**198.0**	192	134	**163.0**	6	256	132	**197.0**	192	128	**160.0**
2	4	2	11	16	18	**22.5**	12	28	**20.0**	13	16	12	**20.5**	12	8	**10.0**
2	4	4	11	32	26	**34.5**	24	36	**30.0**	13	32	20	**32.5**	24	16	**20.0**
2	4	8	11	64	42	**58.5**	48	52	**50.0**	13	64	36	**56.5**	48	32	**40.0**
2	4	16	11	128	74	**106.5**	96	84	**90.0**	13	128	68	**104.5**	96	64	**80.0**
2	4	32	11	256	138	**202.5**	192	148	**170.0**	13	256	132	**200.5**	192	128	**160.0**
3	4	2	11	32	27	**23.3**	28	36	**21.3**	14	32	22	**22.7**	28	22	**16.7**
3	4	4	11	64	43	**39.3**	56	52	**36.0**	14	64	38	**38.7**	56	38	**31.3**
3	4	8	11	128	75	**71.3**	112	84	**65.3**	14	128	70	**70.7**	112	70	**60.7**
3	4	16	11	256	139	**135.3**	224	148	**124.0**	14	256	134	**134.7**	224	134	**119.3**
3	4	32	11	512	267	**263.3**	448	276	**241.3**	14	512	262	**262.7**	448	262	**236.7**
4	5	2	26	64	46	**34.0**	60	82	**35.5**	30	64	40	**33.5**	60	52	**28.0**
4	5	4	26	128	78	**58.0**	120	114	**58.5**	30	128	72	**57.5**	120	84	**51.0**
4	5	8	26	256	142	**106.0**	240	178	**104.5**	30	256	136	**105.5**	240	148	**97.0**
4	5	16	26	512	270	**202.0**	480	306	**196.5**	30	512	264	**201.5**	480	276	**189.0**
4	5	32	26	1024	526	**394.0**	960	562	**380.5**	30	1024	520	**393.5**	960	532	**373.0**

TABLE III

COMPLEXITY PER INFORMATION BIT OF SISO ALGORITHM FOR SOME TYPICAL VALUES OF k, n AND N_s.

complexity of the PCCC and SCCC can be evaluated as [13]

$$\begin{aligned} C_{SCCC} &= \left(C_O + \frac{1}{r_o} C_I\right) \\ C_{PCCC} &= 2C_I, \end{aligned}$$

where r_o is the rate of the outer encoder for SCCC.

C. Memory requirements

The memory requirements of an iterative decoder is the sum of memory required for the storage of channel messages, which is N for all types of decoders and the memory for the storage of the extrinsic messages, which depends on the encoding scheme.

For LDPC the number of extrinsic messages is given by ηN, for SCCC by $\frac{K}{r_o}$n and for PCCC by K.

Memory requirements are usually not negligible and in some important cases, e.g., for low throughput implementation and/or high block sizes, can dominate the computational requirements considered in the next sections.

D. Non binary decoders

Slightly different conclusions are obtained using non binary constituent decoders. The main consequence of this approach is that messages are no longer scalars but vectors of dimension equal to the cardinality of the used alphabet minus one. The dimension of message memory must then be increased accordingly.

The operators BMC and OCU are slightly changed while FR and BR remain the same.

Non binary decoders also yield different performance. An important application of non-binary decoder is the DVB-RCS/RCT and WiMAX 8-state double-binary turbo code and its extension to 16 states [69].

[13] We assume here that the two constituent encoders are equal, generalization to different encoders is straightforward.

VIII. Conclusion

We have presented an overview of implementation issues for the design of iterative decoders in the context of the concatenation of convolutional codes. Hardware architectures were described and their complexity was assessed. Low throughput (below 10 Mbit/s) turbo decoders have already been widely implemented, either in software or hardware, and are commercially available. In this paper, we have laid stress on the different methods allowing the throughput of a turbo decoder to be increased. We have particularly investigated parallel architectures and stopping criteria. As for architecture optimization, a joint design of the concatenated code and the architecture was favoured, especially concerning interleaver design. Such an approach has already been introduced in standards such as DVB-RCS, DVB-RCT and WiMAX. Amongst the main challenges in the years to come, low energy consumption receiver design will represent a crucial one. Significant progress in this field will probably require a real technological breakthrough. Some answers to this problem are currently emerging, such as the analog decoding concept [70]-[71], which allows the iterative process to be removed, the SISO decoders being directly wired together in order to implement the feedback connections.

REFERENCES

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: turbo-codes," in *Proc. ICC'93*. Geneva, Switzerland, May 1993, pp. 1064–1070.

[2] Comatlas, *CAS5093: turbo encoder/decoder*, Nov. 1993, datasheet.

[3] S. Benedetto and G. Montorsi, "Iterative decoding of serially concatenated convolutional codes," *Electron. Lett.*, vol. 32, no. 13, pp. 1186–1187, June 1996.

[4] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Serial concatenation of interleaved codes: performance analysis, design, and iterative decoding," *IEEE Trans. Inform. Theory*, vol. 44, pp. 909–926, May 1998.

[5] S. Benedetto and G. Montorsi, "Serial concatenation of block and convolutional codes," *Elect. Lett.*, vol. 32, no. 10, pp. 887–888, May 1996.

[6] R. M. Pyndiah, "Near-optimum decoding of product codes: block turbo codes," *IEEE Trans. Commun.*, vol. 46, no. 8, pp. 1003–1010, Aug. 1998.

[7] CCSDS, *Recommendation for space data system standards. TM synchronization and channel coding*, Sept. 2003, 131.0-B-1, Blue Book.

[8] Third Generation Partnership Project (3GPP) Technical Specification Group, *Multiplexing and channel coding (FDD)*, June 1999, TS 25.212, v2.0.0.

[9] DVB, *Interaction channel for satellite distribution systems*, 2000, ETSI EN 301 790, v. 1.2.2.

[10] ——, *Interaction channel for digital terrestrial television*, 2001, ETSI EN 301 958, v. 1.1.1.

[11] Third Generation Partnership Project 2 (3GPP2), *Physical layer standard for cdma2000 spread spectrum systems, Release D*, Feb. 2004, 3GPP2 C.S0002-D, Version 1.0.

[12] IEEE, *IEEE standard for local and metriopolitan area networks. Part 16: air interface for fixed broadband wireless access systems*, Nov. 2004, IEEE 802.16-2004.

[13] A. Franchi and J. Sengupta, "Technology trends and market drivers for broadband mobile via satellite: Inmarsat BGAN," in *DSP 2001,Seventh International Workshop on Digital Signal Processing Techniques for Space Communications*. Sesimbra, Portugal, Oct. 2001.

[14] S. Benedetto, R. Garello, G. Montorsi, C. Berrou, C. Douillard, A. Ginesi, L. Giugno, and M. Luise, "MHOMS: high-speed ACM for satellite applications," *IEEE Trans. Wireless Commun.*, vol. 12, no. 2, pp. 66–77, April 2005.

[15] P. Elias, "Error-free coding," *IRE Trans. Inform. Theory*, vol. 4, pp. 29–37, Sept. 1954.

[16] G. D. Forney and Jr., *Concatenated codes*. Cambridge, MA: MIT Press, 1966.

[17] G. Battail, "Weighting the symbols decoded by the Viterbi algorithm," *(in French), Annals of Telecommunications*, vol. 42, pp. 31–38, Jan.-Feb. 1987.

[18] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," in *Proc. IEEE GLOBECOM'89*. Dallas, Texas, Nov. 1989, pp. 47.1.1–47.1.7.

[19] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, pp. 284–287, March 1974.

[20] C. Berrou and A. Glavieux, "Reflections on the prize paper: "near optimum error-correcting coding and decoding: turbo codes"," *IEEE IT Society Newsletter*, vol. 48, no. 2, June 1998.

[21] S. Dolinar and M. Belongie, "Enhanced decoding for the galileo low-gain antenna mission: Viterbi redecoding with four decoding stages," *JPL TDA Progress Report*, vol. 42-121, pp. 96–109, May 1995.

[22] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: turbo-codes," *IEEE Trans. Commun.*, vol. 44, pp. 1261–1271, Oct. 1996.

[23] S. Benedetto and G. Montorsi, "Unveiling turbo codes: some results on parallel concatenated coding schemes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 409–428, March 1996.

[24] ——, "Design of parallel concatenated convolutional codes," *IEEE Trans. Commun.*, vol. 42, pp. 591–600, May 1996.

[25] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Serial concatenation of interleaved codes: performance analysis, design, and iterative decoding," *IEEE Trans. Inform. Theory*, vol. 44, pp. 909–926, May 1998.

[26] S. ten Brink, "Convergence behavior of iteratively decoded parallel concatenated codes," *IEEE Trans. Commun.*, vol. 49, pp. 1727–1737, Oct. 2001.

[27] C. Weiss, C. Bettstetter, and S. Riedel, "Code construction and decoding of parallel concatenated tail-biting codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 366–386, Jan. 2001.

[28] M. Jézéquel, C. Berrou, C. Douillard, and P. Pénard, "Characteristics of a sixteen-state turbo-encoder /decoder (turbo4)," in *Proc. Int. Symp. Turbo Codes*. Brest, France, Sept. 1997, pp. 280–283.

[29] S. Benedetto and G. Montorsi, "Performance of continuous and blockwise decoded turbo codes," *IEEE Commun. Lett.*, vol. 1, no. 3, pp. 77–79, May 1997.

[30] E. K. Hall and S. G. Wilson, "Stream-oriented turbo codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 1813–1831, July 2001.

[31] G. Masera, G. Piccinini, M. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. VLSI Syst.*, vol. 7, pp. 369–379, Sept. 1999.

[32] C.-M. Wu, M.-D. Shieh, C.-H. Wu, Y.-T. Hwang, and J.-H. Chen, "VLSI architectural design tradeoffs for sliding-window log-MAP decoders," *IEEE Trans. VLSI Syst.*, vol. 13, pp. 439–447, April 2005.

[33] R. McEliece, "On the bcjr trellis for linear block codes," *IEEE Trans. Inform. Theory*, vol. 42, no. 4, pp. 1072–1092, july 1996.

[34] C. Hartmann and L. Rudolph, "An optimum symbol-by-symbol decoding rule for linear codes," *IEEE Trans. Inform. Theory*, vol. 22, pp. 514–517, Sept. 1976.

[35] S. Riedel, "MAP decoding of convolutional codes using reciprocal dual codes," *IEEE Trans. Inform. Theory*, vol. 44, no. 3, pp. 1176–1187, 1998.

[36] ——, "Symbol-by-symbol MAP decoding algorithm for high-rate convolutional codes that use reciprocal dual codes," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 175–185, 1998.

[37] G. Montorsi and S. Benedetto, "An additive version of the SISO algorithm for the dual code," in *IEEE International Symposium on Information Theory*, 2001, p. 27.

[38] E. Boutillon and P. G. Gross, W. J. and Gulak, "VLSI architectures for the MAP algorithm," *IEEE Trans. Commun.*, vol. 51, no. 2, pp. 175–185, Feb. 2003.

[39] D. Gnaedig, E. Boutillon, J. Tousch, and M. Jézéquel, "Towards an optimal parallel decoding of turbo codes," in *Proceeding of the 4th International Symposium on Turbo Codes & Related Topics*. Munich, Germany, April 2006.

[40] T. Blankenship, B. Classon, and V. Deai, "High-throughput turbo decoding techniques for 4G," in *Proc. Int. Conf. 3G Wireless and Beyond*. San Francisco, CA, June 2005, pp. 137–142.

[41] A. Giulietti, L. van der Perre, and A. Strum, "Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements," *Elec. Lett.* , vol. 38, no. 5, pp. 232–234, february 2002.

[42] M. J. Thul, N. When, and L. P. Rao, "Enabling high speed turbo-decoding though concurrent interleaving," in *Proc. Int. Symp. on Circuits and Systems (ISCAS'02)*, Phoenix, USA, May 2002, pp. 897–900.

[43] M. J. Thul, F. Gilbert, and N. When, "Concurrent Interleaving architecture for high-throughput channel coding," in *Proc. ICASSP'03*, vol. 2, Apr. 2003, pp. 613–616.

[44] A. Tarable, S. Benedetto, and G. Montorsi, "Mapping interleaving Laws to Parallel Turbo and LDPC decoder Architectures," *IEEE Trans. Inform. Theory*, vol. 50, no. 9, Sep. 2004.

[45] V. Benes, "Optimal rearrangeable multistage connecting networks," *Bell System technical Journal*, vol. 42, pp. 1641–1656, 1964.

[46] C. Berrou, S. Vaton, M. Jzquel, and C. Douillard, "Computing the minimum distances of linear codes by the error impulse method," in *Proc. IEEE GLOBECOM'02*. Taipei, Taiwan, Nov. 2002, pp. 1017–1020.

[47] S. Crozier, P. Guinand, and A.Hunt, "Estimating the minimum distance of turbo-codes using double and triple impulse methods," *IEEE Commun. Lett.*, vol. 9, no. 7, pp. 631–633, June 2005.

[48] C. Berrou, Y. Saouter, C. Douillard, S. Kérouedan, and M. M. Jézéquel, "Designing good permutations for turbo codes: towards a single model," in *Proc. ICC'04*. Paris, France, June 2004, pp. 341–345.

[49] S. Crozier and P. Guinand, "High-performance low-memory interleaver banks for turbo-codes," in *Proc. VTC2001*. Rhodes Greece, october 2001, pp. 2394–2398.

[50] D. Gnaedig, E. Boutillon, M. Gaudet, V. C. and Jézéquel, and P. G. Gulak, "On Multiple Slice Turbo Codes," in *Proc. 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, Sep. 2003, pp. 153–157.

[51] O. Muller, A. Baghdadi, and M. Jézéquel, "Exploring parallel processing levels for convolutional turbo decoding," in *Proc. of the 2nd ICTTA conf.* Damascus, Syria, April 2006.

[52] P. Black and T. Meng, "A 140-mb/s, 32-state, radix-4 viterbi decoder," *IEEE Commun. Lett.*, vol. 27, no. 12, pp. 1877–1885, December 1992.

[53] T. Miyauchi, K. Yamamoto, T. Yokokawa, M. Kan, Y. Mizutani, and M. Hattori, "High-performance programmable SISO decoder VLSI implementation for decoding turbo codes," in *Proc. of the IEEE Global Telecommunications Conference, GLOBECOM '01*. San Antonio, Texas, november 2001, pp. 305–309.

[54] M. Bickerstaff, L. Davis, C. Thomas, D. Garrett, and C. Nicol, "A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless," in *IEEE Solid-State Circuits Conference 2003. Digest of Technical Papers*. San Francisco, California, February 2003.

[55] G. F. andH. Meyr, "Parallel viterbi algorithm implementation: breaking the ACS-bottleneck," *IEEE Trans. Commun.*, vol. 37, no. 8, pp. 785–790, Aug. 1989.

[56] C. Berrou and M. Jezequel, "Non-binary convolutional codes for turbo coding," *Elec. Lett.* , vol. 35, no. 1, pp. 39–40, january 1999.

[57] A. Matache, S. Dolinar, and F. Pollara, "Stopping rules for turbo decoders," *JPL TMO Progress Report*, vol. 42-142, pp. 1–22, Aug. 2000.

[58] S. Shao, L. Shu, and M. Fossorier, "Two simple stopping criteria for turbo decoding," *IEEE Trans. Commun.*, vol. 47, pp. 1117–1999, 1999.

[59] A. Shibutani, H. Suda, and F. Adachi, "Reducing average number of turbo decoding iterations," *Elec. Lett.* , vol. 35, pp. 701–702, 1999.

[60] ——, "Complexity reduction of turbo decoding," in *Proc. of Vehicular Technology Conference, VTC'1999,*, vol. 3. Ottawa, Ontario, 1999, pp. 1570–1574.

[61] K. Gracie, S. Crozier, and A. Hunt, "Performance of a low-complexity turbo decoder with a simple early stopping criterion implemented on a SHARC processor," in *Proc. of the Sixth International Mobile Satellite Conference, IMSC 99*. Ottawa, Ontario, 1999, pp. 281–286.

[62] B. Kim and H. S. Lee, "Reduction of the number of iterations in turbo decoding using extrinsic information," in *Proc. of IEEE TENCON 99*. Inchon, South Korea, 1999, pp. 494–497.

[63] J. Cain, "CMOS VLSI implementation of $r = 1/2$, $k = 7$ decoder," in *IEEE Nat. Aerospace Electron. Conf., NAECOM'84*, vol. 1, 1984, pp. 20–27.

[64] A. Hekstra, "An alternative to metric rescaling in Viterbi decoders," *IEEE Trans. Commun.*, vol. 37, no. 11, pp. 1220–1222, Nov. 1989.

[65] S. Pietrobon, "Implementation and performance of a turbo/MAP decoder," *Int. J. of Satellite Commun.*, vol. 16, pp. 23–46, Jan.-Feb. 1998.

[66] T. Jung-Fu, C. and Ottosson, "Linearly approximated log-MAP algorithm for turbo decoding," in *Vehicular Technology Conference Proceedings, VTC'2000*, vol. 3, Tokyo, 2000, pp. 2252–2256.

[67] P. G. Gross, W. J. and Gulak, "Simplified MAP algorithm suitable for implementation of turbo decoder," *Elect. Lett.*, vol. 34, no. 16, pp. 1577–1578, Aug. 1998.

[68] C. J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and H. Xiao-Yu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Commun.*, vol. 53, no. 8, pp. 1288–1299, Aug. 2005.

[69] C. Douillard and C. Berrou, "Turbo codes with rate-$m/(m+1)$ constituent convolutional codes," *IEEE Trans. Commun.*, vol. 53, no. 10, pp. 1630–1638, October 2005.

[70] H.-A. Loeliger, F. Tarkoy, F. Lustenberger, and M. Helfenstein, "Decoding in analog VLSI," *IEEE Commun. Mag.*, vol. 37, pp. 99–101, Apr. 1999.

[71] J. Hagenauer, "Decoding of binary codes with analog networks," in *Proc. 1998 Information Theory Workshop*. San Diego, CA, USA, Feb. 1998, pp. 13–14.