Design of Next-Generation Tbps Turbo Codes

Vinh Hoang Son Le

December 21, 2020

Contents

Acronyms

ACQ	Acquisition
ACS	Add-Compare-Select
ACSU	Add-Compare-Select Unit
ARP	Almost Regular Permutation
APP	A Posteriori Probability
AWGN	Additive White Gaussian Noise
BCH	Bose–Chaudhuri–Hocquenghem
BCJR	Bahl-Cocke-Jelinek-Raviv
BER	Bit Error Rate
BMU	Branch Metric Unit
BPSK	Binary Phase Shift Keying
BR	Battail Rule
CS	Compare-Select
Dual-LM	Dual-Log-MAP
Dual-MLM	Dual-Max-Log-MAP
DVB	Digital Video Broadcasting
DVB-RCS	Digital Video Broadcasting - Return Channel via Satellite
DVB-SH	Digital Video Broadcasting - Satellite Handheld
ESF	Extrinsic Scaling Factor
FB	Forward-Backward
FEC	Forward Error Correction
FER	Frame Error Rate
FPMAP	Fully-Parallel MAP
FSM	Finite State Machine
HI	Half-Iteration
HR	Hagenauer Rule
HSPA	High Speed Packet Access
KPI	Key Performance Indicators
LDPC	Low Density Parity Check

LLR	Log-Likelihood Ratio
LM	Log-MAP
LTE	Long Term Evolution
LUT	Lookup Table
MAP	Maximum A Posteriori
ML	Maximum Likelihood
MLM	Max-Log-MAP
NII	Next Iteration Initialization
PMAP	Parallel MAP
QPP	Quadratic Permutation Polynomial
RSC	Recursive Systematic Convolutional
SIHO	Soft-Input Hard-Output
SISO	Soft-Input Soft-Output
SM	Sign-Magnitude
SMA	Sign-Magnitude Addition
SMD	Sign-Magnitude Division
SMM	Sign-Magnitude Multiplication
SNR	Signal-to-Noise Ratio
SOU	Soft-Output Unit
SOVA	Soft-Output Viterbi Algorithm
UMTS	Universal Mobile Telecommunications System
UXMAP	Unrolled XMAP
VA	Viterbi Algorithm
WiMAX	Worldwide Interoperability for Microwave Access
XMAP	Pipelined MAP

Chapter 1

Introduction

Digital wireless communications can be considered as the most dynamic area in the communication field today. The last few decades have seen a surge of research and industrial activities in the area. A combination of several factors can explain this surge. The first and most obvious is the demand for wireless connectivity. Cordless devices are now able to communicate efficiently with each other and can access the information without geographical boundaries. Furthermore, the development of the semiconductor technology, which is governed by Moore's law [1], allows sophisticated signal processing algorithms to be integrated into the devices with small area and low power consumption. Last but not least, the development of information theory, pioneered by Claude Shannon in his original article [2], paved the way for efficient communication systems to be implemented with high reliability.

Subsequently, the success of the second generation (2G) of mobile communication standards was a concrete demonstration that good wireless digital communication systems can be achieved. However, the 2G system still concentrates mostly on voice communication, as it can only provide a bidirectional communication link with a maximum throughput of 9.6 kb/s. Then, the demand for data transmission with increasingly high data rates, including video telephony, drove the world of wireless communications. It is these challenges and also the related interests that have fascinated many researchers and attracted them to this field. In 1999, the UMTS technology [3] was introduced, which allows a data rate of 384 kb/s to be achieved. Since then, transmission with increased data throughput has been one of the most important aspects of wireless communications. As a result, a data rate of 84 Mb/s can be reached by the HSPA+ technology [4], released in 2007, and the LTE standard [5] can transmit with a maximum data rate of 300 Mb/s. With the most recent mobile standards, LTE Advanced Pro [6] and the 5G New Radio [7], throughputs from a few Gb/s to a few tens of Gb/s can be achieved. Following this trend, the throughput of mobile broadband communications will be in the order of hundreds of Gb/s and up to Tb/s in the near future, especially with the advent of THz communications.

As part of the communication system, *channel coding*, or *forward error correction* (FEC), plays a critical role in enabling the communication link. A channel code adds *redundant information* to the *original information* to be transmitted. Thanks to the redundancy, corrupted information due to noise and channel impairments can be corrected to recover the message at the receiver side. Sophisticated channel codes can drastically improve the reliability of the communication, thus allowing efficient data transmission between devices with limited transmit power. Many channel codes can be found in the

literature as well as in industrial communication standards. The two classic code families are block codes, such as Hamming, Golay, BCH, or Reed-Solomon codes, and convolutional codes [8]. Then, in the nineties came the turbo codes with iterative decoding [9], the rediscovery of low-density parity-check (LDPC) codes [10] and more recently the polar codes [11]. The invention of turbo codes by C. Berrou in 1991 is considered as one of the most important milestones of the channel coding research. They were the first practical codes able to operate within a fraction of a dB of the Shannon limit. Since then, turbo codes have been chosen as the channel code in many digital communication standards, such as the 3G/4G mobile standards and the digital video broadcasting DVB-RCS/RCS2 and DVB-SH standards [12]. Furthermore, the *iterative decoding* of these codes and the concept of *extrinsic information* have had influences on the decoding of LDPC codes but also on many other elements of the communication chain, like modulation, equalization, and multi-antenna techniques.

1.1 Motivation and Challenges

For the time being, wireless digital communications using the LTE Advanced Pro or the newly 5G standards can provide up to tens of Gb/s communication links. As a result, we can enjoy high-quality media material and streaming while on the move with our mobile phones. Nevertheless, as technologies develop, so do applications. Therefore, the demand for even higher data rates of hundreds of Gb/s or Tb/s can be foreseen in the future. Yet, new use cases of this "ultra-high-throughput" era have already been identified. Several examples can be cited, such as wireless intra-devices communication [13], mobile augmented/virtual reality communications [14], wireless backhaul/fronthaul [15], communications in data centers [16], hybrid fiber-wireless networks [17], and high-throughput satellites communications [18].

The increase of throughput up to Tb/s comes with the need to move to frequencies at the THz range (0.1 THz − 10 THz) as well as with having the appropriate corresponding baseband processing, especially for FEC. To this end, the European H2020 project EPIC (Enabling Practical Wireless Tb/s Communications with Next Generation Channel Coding, `https://epic-h2020.eu/`) lent itself to develop a set of new implementation-ready FEC technologies that meet the cost and performance requirements of a variety of future wireless Tb/s use cases.

In the past, steady progress in silicon technology – as governed by Moore's law – could be regarded as the enabler of large leaps in data rates without the need for major algorithmic innovations on the FEC design part. However, a key finding of EPIC is the prediction that the upgrade to Tb/s wireless data rates will not be smooth: the improvements carried by silicon technology progress in the next decade will significantly fall short of meeting the Tb/s FEC challenge [19]. Therefore, the EPIC project is based on the thesis that major algorithmic and architectural innovations will be required in the design and implementation of FEC algorithms to make Tb/s wireless communications feasible. As candidates for the FEC technologies, the EPIC project considered three code classes: turbo codes, LDPC codes, and polar codes, as their error correction performance come close to the Shannon limit.

This thesis focuses on the development of turbo codes in the framework of the EPIC project. Efficiently achieving ultra-high throughput (up to Tb/s) for turbo codes is very challenging, since turbo codes are inherently serial at the component decoder level. Be-

sides the main requirement in throughput, within the EPIC project framework, additional constraints such as chip area, area efficiency, power density, code flexibility, and bit error rate (BER) should also be met [20]. Table 1.1 shows the main FEC Key Performance Indicators (KPI) in the EPIC context under 28 nm technology node. Addressing all these constraints requires the exploration of new decoding algorithms and highly parallel architecture templates.

Applications	BER	Flexibility	Latency	Throughput (Gb/s)	Area eff. (Gb/s/mm^2)	Power dens. (W/mm^2)	Energy eff. (pJ/bit)
Virtual Reality	10^{-6}	high	0.5 ms	500	50	0.02	0.48
Intra-Device Com.	10^{-12}	low	100 ns	500	50	0.13	1.00
Fronthaul	10^{-13}	medium	25 ns	1000	100	0.17	0.60
Backhaul	10^{-8}	medium	100 ns	250	25	0.09	3.60
Data Center	$> 10^{-12}$	medium	100 ns	1000	100	0.20	0.75
Hybrid Fiber-Wireless	10^{-12}	medium	200 ns	1000	100	0.23	1.13
High Throughput Sat.	10^{-10}	medium	10 ms	100 - 1000	100	0.27	0.50

Table 1.1: Summary of FEC level KPI for different use cases under 28 nm technology node.

1.2 Objectives, Contribution and Outline

The objective of this thesis is to explore innovative turbo decoding techniques, allowing the decoder to achieve or approach very high-throughput transmission (Tb/s). The contribution of the thesis is shown as follows.

- The discovery of a new soft-input soft-output (SISO) decoding algorithm based on the manipulation of paths in the trellis diagram, namely the Local-SOVA. It exhibits a lower computational complexity than the conventional *maximum a posteriori* algorithm with Max-Log approximation (Max-Log-MAP) when employed for high-radix decoding in order to increase throughput, while having the same error correction performance even when used in a turbo decoding process. Furthermore, with some simplifications, it offers various trade-offs between error correction performance and computational complexity.

- The application of the Local-SOVA in an associated hardware architecture, the Unrolled-XMAP (UXMAP). This architecture combines iteration unrolled and spatial parallelism with fully pipelined component decoders. Since complete decoded frames are produced in each clock cycle, the UXMAP architecture guarantees a very high-throughput for the turbo decoder. The Local-SOVA helps lower the area cost of the UXMAP architecture in high-radix schemes, which can translate into a higher decoder throughput. The high-radix Local-SOVA also proposes several low latency solutions to the UXMAP architecture.

- The exploration of decoding algorithms based on the dual trellis of the convolutional codes. To this end, we generalized the method for constructing the dual trellis, given

a convolutional code with arbitrary code rate k/n. Furthermore, a new decoding algorithm using the dual trellis was proposed with a lower complexity than the state-of-the-art algorithms.

The remainder of this thesis is structured as follows:

- **Convolutional Codes and Turbo Codes (Chapter 2)**: This chapter first introduces the concept of channel coding in a point-to-point communication context. It further presents convolutional codes, the process of encoding as well as the decoding algorithms using the trellis diagram. Then, it gives an overview of turbo codes consisting of the encoding process, the role of interleavers, and the iterative decoding process.

- **High-Throughput SISO Decoders (Chapter 3)**: This chapter first provides several key performance indicators for the turbo decoders such as throughput, latency, complexity, error performance, and flexibility. Furthermore, the chapter describes well-known techniques employed in turbo decoders to increase the throughput and to lower the complexity. Then, several parallelism architectures applicable to turbo decoders are discussed. Notably, for high-throughput, the PMAP and XMAP architectures are considered, and for very-high-throughput, the FPMAP and the UXMAP architectures are discussed and compared to each other.

- **The Local-SOVA (Chapter 4)**: The Max-Log-MAP algorithm has been employed extensively for decoding the component convolutional codes in a turbo code. When consider high-radix schemes, the complexity of the Max-Log-MAP algorithm increases rapidly with the radix orders. Therefore, in this chapter, we propose a new decoding algorithm and name it the Local-SOVA. The Local-SOVA can operate in a similar manner to the Max-Log-MAP algorithm, and exhibits a lower computational complexity with the same error performance. Thus, the Local-SOVA can be employed in various turbo decoder architectures. For high-radix schemes such as radix-4 and radix-8, we investigate the saving in complexity that the Local-SOVA can provide compared to the Max-Log-MAP. Furthermore, the Local-SOVA enables even higher radix orders (radix-16, radix-32) to be implemented efficiently.

  The results of this work were published in the IEEE Transaction on Communications [21].

- **The Local-SOVA in Unrolled-XMAP Architecture (Chapter 5)**: Based on the computational complexity analysis of the Local-SOVA, this chapter focuses on the analysis of several Local-SOVA implementations in the UXMAP architecture. The chapter first gives an overview of the UXMAP architecture coupled with the Max-Log-MAP algorithm. Then, the implementation of the Local-SOVA computational units are carried out and compared with the Max-Log-MAP algorithm in terms of throughput and area complexity. The chapter also provides different Local-SOVA schemes with different radix orders to further exploit the possibility of using the Local-SOVA in the context of a high throughput, low latency UXMAP architecture.

- **SISO Decoding Algorithms with The Dual-Trellis (Chapter 6)**: For decoding a convolutional codes with high code rates, the decoder usually employs

the trellis of the mother code in case of puncturing at the encoder side. This approach has the advantages of flexibility but the throughput of the decoder is always bounded by the throughput of decoding the mother code. To this end, decoding a high-rate convolutional codes on the dual trellis provides a high-throughput solution. This chapter first introduces the concept of dual trellis and presents the decoding algorithms for the dual trellis in the literature. Then, we generalize the method of constructing the dual trellis for a given convolutional codes with arbitrary code rates. Secondly, a new decoding algorithm is proposed with a lower complexity than the state-of-the-art algorithms in exchange with a sub-optimal error performance.

The contributions discussed in this chapter were published and presented in the following conferences and workshop [22], [23], and [24].

- **Conclusion and Future Works (Chapter 7)**: This chapter concludes the thesis and gives an perspective to the future works that can be done with the Local-SOVA and the decoding algorithm using the dual trellis.

Chapter 2

Convolutional Codes and Turbo Codes

This chapter aims at giving the necessary background information about convolutional codes and turbo codes in the context of a point-to-point digital communication chain. Special attention will be given to the different decoding algorithms for these codes.

The rest of this chapter is organized as follows. First, Section 2.1 introduces the transmission system model considered throughout this thesis with a specific focus on channel coding. Then, Section 2.2 gives an overview of convolutional codes: the convolutional encoding process, the main techniques to achieve high coding rates, the trellis diagram representation of the encoder. Decoding algorithms for these codes are reviewed in Section 2.3 consisting of the Viterbi algorithm, the soft-output Viterbi algorithm, and the MAP-based algorithms. Next, Section 2.4 introduces turbo codes, their encoding process, and the role of the interleaver is explained. The principle of turbo decoding and the choice of the decoding algorithm are discussed in Section 2.5. Finally, Section 2.6 summarizes the chapter.

2.1 Short Introduction to Channel Coding

Figure 2.1 depicts the simplified diagram of a point-to-point digital communication chain used to convey information between the source and the destination (or the sink).

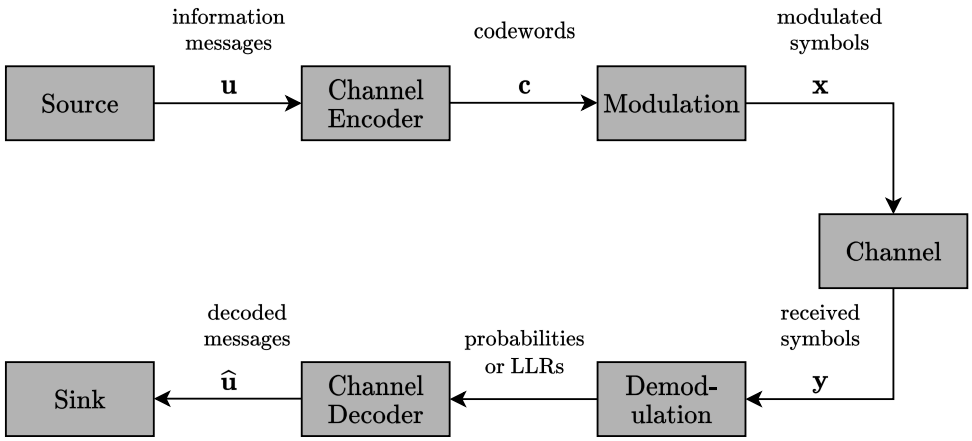


Figure 2.1: Simplified block diagram of a digital communication system with channel coding.

The source generates information in packets of length K represented by vector $\mathbf{u} = [u_0, \ldots, u_{K-1}]$. This information is provided in digital format that could come from a text, an image, a video, or could be the sample of an analog signal, possibly after a compression step. After that, channel encoding is used to map the input information onto a *codeword* $\mathbf{c} = [c_0, \ldots, c_{N-1}]$ of length N. Note that N should be greater than K since the role of the channel code is to add *redundant information* to the original message so that at the receiver, the redundant information can be used to recover the corrupted or distorted message information. The ratio $R = K/N$ denotes the *code rate* of the channel code. Then, the channel codeword is modulated to convert the digital sequence to an analog waveform compatible with the transmission channel. The channel here is modeled to mimic the physical channel. Noise, distortion, interference can be elements of a channel that cause degradation in the received signal.

At the receiver, reverse processing blocks are employed to recover the transmitted message. The received symbol sequence $\mathbf{y}$ is first demodulated to provide the conditional probabilities or the log-likelihood ratios to the channel decoder. The channel decoder then employs specific decoding algorithms to get the estimate of transmitted message information or the decoded message denoted by $\hat{\mathbf{u}} = [\hat{u}_0, \ldots, \hat{u}_{K-1}]$.

In this work, a simple modulation/demodulation scheme and channel model have been chosen to facilitate the assessment of our study. However, the work done and the achieved results are also valid for other modulations and channels. For modulation, binary phase shift keying (BPSK) is employed. The modulator maps a codeword bit $c_k \in \{0, 1\}$ to a modulated symbol $x_k \in \{-1, +1\}$. Moreover, the channel is assumed to be additive white Gaussian noise (AWGN) with zero mean and variance σ^2. Thus, the received noisy symbol is

$$y_k = x_k + w_k, \text{ for } k = 0, 1, \ldots, N-1, \tag{2.1}$$

where $w_k \sim \mathcal{N}(0,\, \sigma^2)$. The conditional probability of the received symbol y_k is

$$\Pr\{y_k|x_k\} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_k - x_k)^2}{2\sigma^2}\right). \tag{2.2}$$

Then, the likelihood ratio can be derived as

$$\frac{\Pr\{y_k|x_k = 1\}}{\Pr\{y_k|x_k = -1\}} = \exp\left(-\frac{(y_k - 1)^2}{2\sigma^2} + \frac{(y_k + 1)^2}{2\sigma^2}\right) = \exp\left(\frac{2y_k}{\sigma^2}\right). \tag{2.3}$$

Finally, the *log-likelihood ratio* (LLR) is obtained by taking the natural logarithm of (2.3)

$$\mathrm{L}\left\{y_k|x_k\right\} = \frac{2y_k}{\sigma^2}. \tag{2.4}$$

In order to assess the performance of a communication system, *bit-error rate* (BER) and *frame-error rate* (FER) measurements are usually employed. These metrics are usually obtained by Monte-Carlo simulations where the decoded message $\hat{\mathbf{u}}$ is compared with the original message $\mathbf{u}$. Furthermore, the performance of the system varies with the *signal-to-noise ratio* (SNR). In digital communication systems, the SNR is described by E_b/N_0, where E_b is the *bit energy* and N_0 represents the power spectrum density (PSD) of the white noise. The general expression for the SNR is

$$E_b/N_0 = \frac{E_s}{R{\times}M{\times}N_0}, \tag{2.5}$$

where E_s is the *symbol energy*, R is the code rate of the channel code, and M is the number of bit per modulated symbol. In the case of BPSK modulation, M is equal to one bit per symbol.

In simulations, for a given E_b/N_0, the symbol energy E_s is assumed to have a nominal value of 1 Joule, then the noise PSD is

$$N_0 = \frac{1}{R{\times}E_b/N_0} \text{ (Watt/Hz)}. \tag{2.6}$$

Consequently, the variance of the band-limited AWGN is

$$\sigma^2 = \frac{N_0}{2} = \frac{1}{2R{\times}E_b/N_0}, \tag{2.7}$$

and is used to generate the Gaussian noise added to the transmitted symbols.

Figure 2.2 shows the performance in BER of different communication settings with and without the use of channel coding. The channel code employed in the figure is a convolutional code with various code rates. The first observation is that compared to the *uncoded* scheme, using channel coding tends to have higher BER at low SNR region (from $0 \sim 1$ dB, as shown in the figure). However, as the SNR increases, the performances of the coded schemes are far superior to the uncoded one. It is also important to observe that different code rates yield different performance at a given SNR. For high code rates, although the bit energy E_b is higher, the amount of redundant information is less than the lower code rates. Thus, the performance is worse as the code rate increases. Nonetheless, the use of less redundant information means that more message information can be transmitted for a fixed bandwidth. Therefore, high code rates produce higher data rates at the expense of performance deterioration.

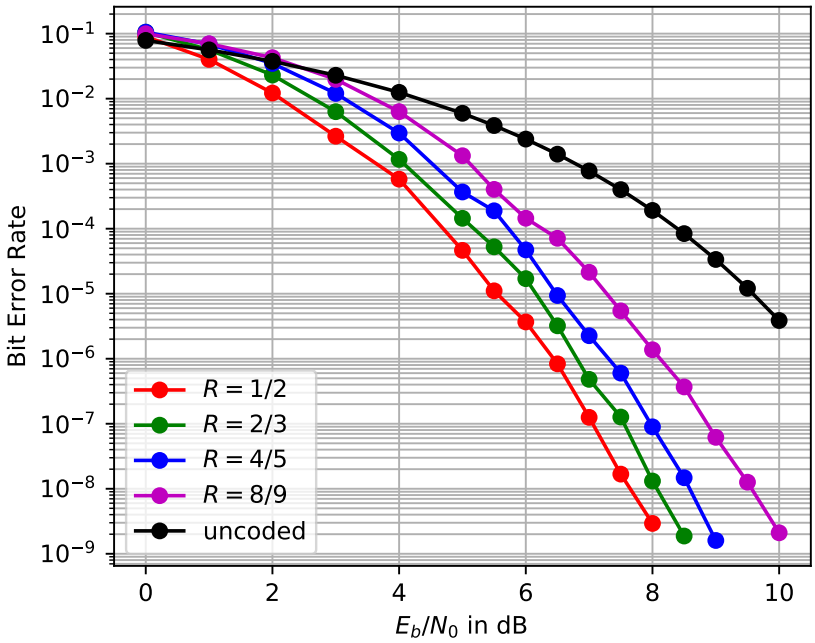


Figure 2.2: Performance amelioration due to convolutional codes.

2.2 Convolutional Codes

The field of channel coding is traditionally partitioned into two categories: block codes and convolutional codes. In block codes, each input message $\mathbf{u}$ of length K is mapped to a codeword $\mathbf{c}$ of length N by linear combinations. These linear combinations are often represented by a $K \times N$ generator matrix $\mathbf{G}$. Each input message is encoded independently; hence, the encoding process of block codes is memoryless.

In convolutional codes, the information message of length K is divided into M information chunks of k bits. Then, for each k input bits, the convolutional encoder produces n output bits generated by a linear combination of the k current information inputs and the previous information through ν steps of code memories, known as the *memory length* of the convolutional code. The value $(\nu + 1)$ is also usually referred as the *constraint length* of the convolutional code. Therefore, a convolutional code can be denoted as a 3-tuple (ν, k, n). Moreover, for each message of length K, the encoding process produces in total M chunks of n output bits, and the codeword length is $N = n \times M$. Thus, for convolutional codes, the code rate R can be described by $k/n = K/N$.

Convolutional codes were first proposed in 1955 by Elias [25]. But it was not until 1967 when Viterbi proposed a *maximum-likelihood* decoding algorithm with reasonable complexity [26], that convolutional codes started to be widely employed.

2.2.1 Convolutional Encoding

A (ν, k, n) convolutional code can be described by a *generator matrix* $\mathbf{G}$, or its D-transform notation $\mathbf{G}(D)$. While the former can be seen as a $K \times N$ generator matrix as in block codes, the latter is more convenient to represent convolutional codes.

The matrix $\mathbf{G}(D)$ is referred as the *polynomial generator matrix* consisting of $k \times n$

polynomials $g_{i,j}(D)$

$$\mathbf{G}(D) = \begin{pmatrix} g_{0,0}(D) & g_{0,1}(D) & \dots & g_{0,n-1}(D) \\ g_{1,0}(D) & g_{1,1}(D) & \dots & g_{1,n-1}(D) \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0}(D) & g_{k-1,1}(D) & \dots & g_{k-1,n-1}(D) \end{pmatrix}. \tag{2.8}$$

Consequently, a *convolutional encoder* can be viewed as a set of digital filters - linear, time invariant - characterized by the transfer functions defined by the entries of the polynomial generator matrix $\mathbf{G}(D)$. Figure 2.3a shows a $(2, 1, 2)$ convolutional encoder with polynomial generator matrix:

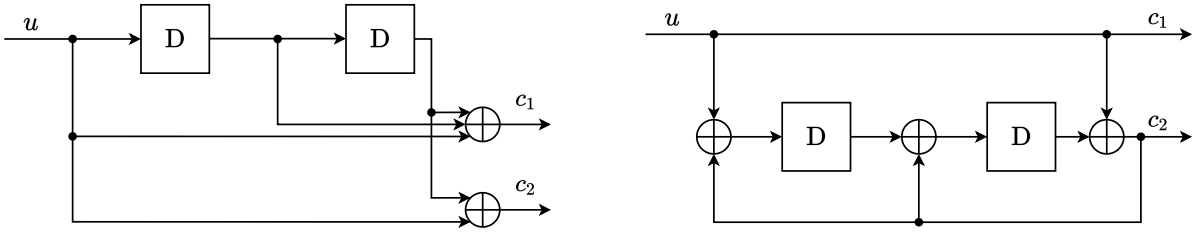
$$\mathbf{G}(D) = (1{+}D{+}D^2 \quad 1{+}D^2). \tag{2.9}$$

Furthermore, the polynomial entries can also be represented in octal numbers with the most significant bit (MSB) being the highest degree of the polynomial. For example, the polynomial generator matrix in (2.9) can also be represented as $(7,\ 5)_{\text{oct}}$.

Encoders for convolutional codes can be classified as *systematic* or *non-systematic*. For a rate $R = k/n$ systematic convolutional encoder, k out of n outputs from the encoder are replicas of the k inputs. Otherwise, the encoder is said to be non-systematic. Furthermore, convolutional encoders can also fall into two classes: *recursive* (or *feedback*) and *non-recursive* (or *feedforward*). A convolutional encoder is recursive if there exists an output entry that is fed back to the calculation of the shift registers. In other words, if there exists an entry of the polynomial generator matrix $\mathbf{G}(D)$ whose realization is an infinite impulse response filter, then the encoder is recursive. Otherwise, the encoder is non-recursive. The generator matrix in (2.9) and its encoder shown in Figure 2.3a are non-recursive and non-systematic. Meanwhile, the polynomial generator matrix

$$\mathbf{G}_{\mathbf{sys}}(D) = \begin{pmatrix} 1 & \dfrac{1{+}D^2}{1{+}D{+}D^2} \end{pmatrix}, \tag{2.10}$$

and its encoder shown in Figure 2.3b are recursive and systematic.



(a) Non-recursive non-systematic (b) Recursive systematic

Figure 2.3: Two typical types of convolutional encoder.

Note that the non-recursive non-systematic generator matrix in (2.9) and the recursive systematic one in (2.10) are equivalent. Both encoders will generate the same *codespace* (the set of all codewords). Therefore, depending on the applications, the convolutional encoder could be chosen to be non-recursive and non-systematic, or to be recursive and systematic. While the former could be observed in applications such as the control channel of mobile communication systems, the latter is extensively used in systems employing turbo codes.

2.2.2 The Trellis Diagram Representation of Convolutional Encoders

A convolutional encoder consists of a set of shift registers, and the output is calculated as a linear combination of the input and the contents of the shift registers. Therefore, the convolutional encoder can be viewed as a *finite state machine* (FSM). Then the *trellis diagram* is simply a cascade over time of the state diagram of the state machine. The trellis diagram is a useful representation of a convolutional code since it helps to visualize the encoding process as well as the decoding process of the code.

Given an encoder of a (ν, k, n) convolutional code, the process of constructing the trellis diagram is as follows. First, the FSM of the encoder is obtained: for each of the 2^{ν} states s of the FSM and for each input symbol, the next state s' and the output symbols are computed. Figure 2.4 depicts the FSM of the convolutional encoder shown in Figure 2.3b. Then, the trellis diagram of the convolutional code is deduced from the FSM, as shown in Figure 2.5 for the same code.

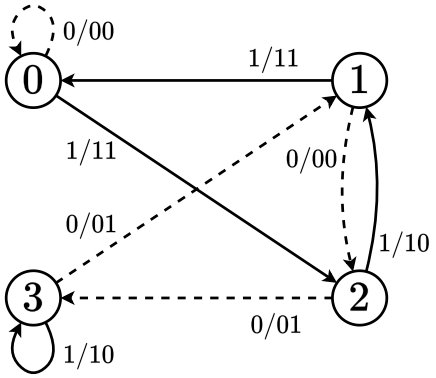


Figure 2.4: FSM of the convolutional encoder in Figure 2.3b.

The trellis diagram can be viewed as a two-dimensional plane. The horizontal axis consists of *time instants*, and the vertical axis consists of the values of the encoder states. In this work, we consider mostly binary convolutional codes (ν, k, n). Therefore, if an input message has K information symbols, the trellis diagram of the encoder consists of $(K/k + 1)$ time instants. At each time instant t, the state s_t can take a value in $[0, 2^{\nu-1}]$. The interval between time instants t and $(t+1)$ is called *trellis section* t. The states at instant t are connected to states at instant $(t+1)$ in trellis section t by *branches* in the trellis diagram. In the case of binary convolutional codes with $k = 1$ as shown in Figure 2.5, there are two branches coming out of any state s at instant t, each being connected to a state at instant $(t+1)$. Of the two branches, one is associated with input bit $u_t = 0$ (dashed line) and the other with input bit $u_t = 1$ (solid line). An aggregation of continuous successive branches forms a *state sequence* which defines a *path* in the trellis diagram. The concept of path is extensively useful for decoding convolutional codes using the well-known Viterbi algorithm.

In the case of packet transmission, the convolutional code can be made into a block code by the means of trellis termination. There are two main termination methods that are used: *force-to-zero* [27] and *tailbiting* [28]. Terminating with force-to-zero introduces additional dummy bits at the end of the message to force the trellis from state s at instant K to a known state (usually state 0) at instant $K+\nu$. On the other hand, with

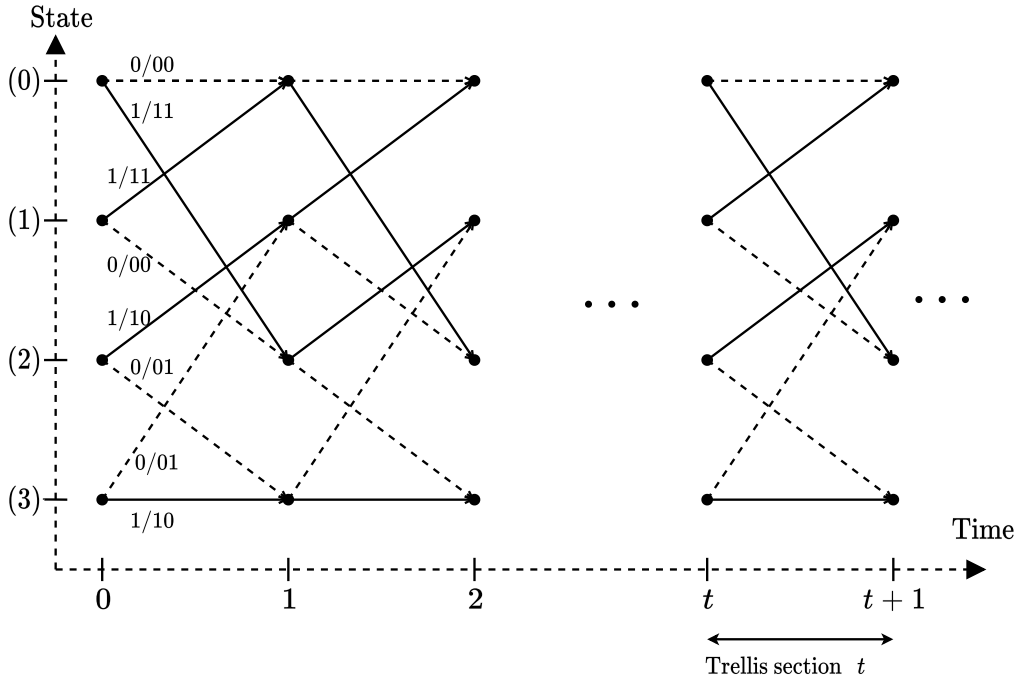


Figure 2.5: Trellis diagram of the convolutional encoder in Figure 2.3b.

the *tailbiting* technique, the initial state of the trellis diagram is found based on the input sequence. This initial state is obtained to ensure that the final state will be equal to the initial state, thus making the trellis diagram circular.

2.2.3 High-Rate Convolutional Codes

For high-throughput transmission, the redundancy part of the channel encoder is usually limited so that higher number of information can be transmitted with the same transmission resources. This is generally referred to as high coding rate schemes.

For convolutional codes, high coding rates can be achieved with two methods: with *true high-rate* encoders or by *puncturing* a mother encoder with lower coding rate.

2.2.3.1 True High-Rate Convolutional Encoding

A first method to achieve high coding rates is the direct use of true high-rate convolutional encoders. The convolutional encoder is characterized by a $k \times n$ polynomial generator matrix. Common coding rates values are of the form $R = k/(k+1)$, i.e. $n = k + 1$. For a systematic convolutional codes, this corresponds to the case where 1 output symbol is computed for very k-symbol chunk.

Good convolutional codes with high coding rates have been studied in the literature for both non-systematic [29, 30] and systematic codes [31]. The following example is taken from table IV in [31], where a $(3, 4, 5)$ recursive systematic convolutional code is described

by the following systematic generator matrix:

$$\mathbf{G}_{\text{sys}}(D) = \begin{pmatrix} 1 & 0 & 0 & 0 & \frac{1+D+D^2+D^3}{1+D+D^3} \\ 0 & 1 & 0 & 0 & \frac{1+D+D^2}{1+D+D^3} \\ 0 & 0 & 1 & 0 & \frac{1+D^2+D^3}{1+D+D^3} \\ 0 & 0 & 0 & 1 & \frac{1+D^3}{1+D+D^3} \end{pmatrix} \tag{2.11}$$

The corresponding encoder is shown in Figure 2.6. The recursive systematic encoder with code rate $k/(k+1)$ can also be represented by a vector of octal numbers, where the first k entries are the numerator polynomials and the last entry is the denominator polynomial of the last column of $\mathbf{G}_{\text{sys}}(D)$. For this example, the corresponding octal vector is $(17, 7, 15, 11, 13)_{\text{oct}}$.

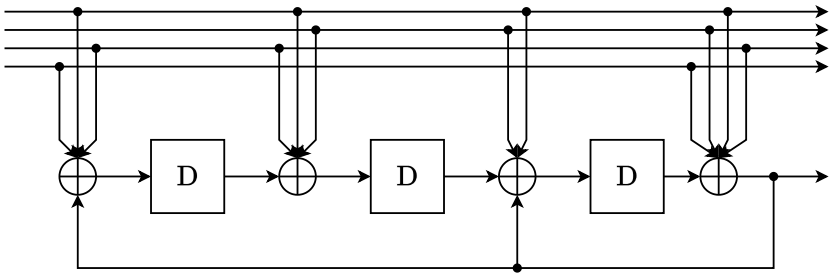


Figure 2.6: Recursive systematic encoder of $(17, 7, 15, 11, 13)_{\text{oct}}$ convolutional code.

2.2.3.2 High-Rate Codes Obtained by Puncturing

Besides the use of true high-rate convolutional encoders, puncturing a low-rate mother encoder is an alternative technique to obtain a high-rate convolutional code. The message is first encoded using a rate $R = 1/n$ convolutional code (the *mother code*), but the coded symbols are not all transmitted over the transmission channel. For ease of specification and implementation, the discarded symbols can be periodically removed (or punctured) following a periodic pattern, called the *puncturing pattern* represented by a $n \times p$ matrix $\mathbf{F}$, where p is the puncturing period. The successive rows of matrix $\mathbf{F}$ describe the pattern applied for the n outputs of the mother code. For example, a rate-4/5 code can be obtained from a rate-1/2 mother code using the following puncturing pattern:

$$\mathbf{F} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{2.12}$$

The search for good puncturing patterns has been intensively studied in [32, 33] for feedforward convolutional codes and in [31, 34] for recursive systematic convolutional (RSC) codes, employed in turbo codes.

The main advantage of the puncturing technique over the true high-rate encoder is the flexibility in changing the code rate. Indeed, a wide range of code rates can be easily achieved by changing the puncturing pattern. Meanwhile, changing code rates in the true high-rate method requires different encoders, each for a specific code rate. Furthermore, since punctured high-rate codes solely employ the trellis diagram of the mother code, one single decoder for the mother code can be used for various code rates. On the contrary, different decoders should be employed for each code rate with true high-rate codes. That

is why, in most transmission systems using convolutional codes, puncturing is the chosen solution for achieving high coding rates. In return, true high-rate codes show better error correction performance than punctured high-rate codes [31].

2.3 Decoding Algorithms for Convolutional Codes

In communication systems, a convolutional code can be employed as a stand-alone code, or it can be concatenated in serial or in parallel with an outer code. Therefore, depending on the case, two main types of decoders can be used: *soft-input hard-output* (SIHO) decoders and *soft-intput soft-output* (SISO) decoders. In this section, the Viterbi algorithm (VA) is introduced as a candidate for SIHO decoders. For SISO decoders, the soft-output Viterbi algorithm (SOVA) and the *maximum a posteriori* (MAP) based family algorithms will be presented. Note that these algorithms uses the code trellis diagram for decoding. For other decoding algorithms, readers can refer to [8] for a full reference. Furthermore, for the sake of simplicity, we restricted ourselves to binary convolutional codes with code rate $R = 1/n$.

2.3.1 The Viterbi Algorithm

The VA was first introduced in 1967 by Andrew Viterbi [26] as a decoding algorithm with reasonable complexity for convolutional codes. Then, in 1973, the VA was recognized as a maximum-likelihood decoder by Forney in [35]. Since then, the VA and its derivatives play a major role in decoding convolutional codes.

The VA decodes by exclusively employing the trellis diagram of the convolutional code. The trellis diagram is used to represent a finite-state discrete-time Markov process observed in white noise. Using the channel LLRs introduced in (2.4), the VA finds the state sequence $\mathbf{s}$ in the trellis diagram that maximizes the log-likelihood function $\ln \Pr\{\mathbf{y}|\mathbf{s}\}$. Then, from the estimated state sequence, the codeword as well as the message can be retrieved. The decoding algorithm consists of two processes: the recursive *path propagation* process and the *traceback* process.

2.3.1.1 The Path Propagation Process

Let us denote $M_t(s)$ the *path metric* at instant t for state $s \in [0;\, 2^\nu - 1]$ in the trellis diagram. To compute the path metrics at instant t, the VA recursively calculates the path metrics from the initial instant $t = 0$. In case the trellis diagram starts from the state zero, the 2^ν path metrics are initialized as

$$M_0(s) = \begin{cases} +\infty, & \text{if } s = 0, \\ 0, & \text{otherwise,} \end{cases} \tag{2.13}$$

and in case in case the initial state is not known in advance, they are initialized as

$$M_0(s) = 0, \quad \forall s \in [0;\, 2^\nu - 1]. \tag{2.14}$$

Assuming there are 2^ν paths at instant t, the VA then calculates the path metric for the next instant $t+1$. For binary convolutional codes, there are two branches coming from two different states s_t^0 and s_t^1 at instant t that *merge* to state s_{t+1} at instant $t+1$.

Therefore, there are two candidate paths for state s_{t+1}. The path metric of each candidate is computed as

$$M_t(s_t^i) + \Gamma_t(s_t^i, s_{t+1}), \quad i = 0,\ 1, \tag{2.15}$$

where $\Gamma_t(s_t^i, s_{t+1})$ is the metric of the branch connecting s_t^i and s_{t+1}. The branch metric is calculated as

$$\Gamma_t(s_t^i, s_{t+1}) = \sum_{j=nt}^{n(t+1)-1} c_j(s_t^i, s_{t+1}) \times \mathrm{L}\{y_j|x_j\}, \tag{2.16}$$

where $c_j(s_t^i, s_{t+1}) \in \{0, 1\}$ is the j-th output bit labelling branch (s_t^i, s_{t+1}), and $\mathrm{L}\{y_j|x_j\}$ is the j-th channel LLR. The path metric for state s_{k+1} is selected as the most probable path between the candidates as

$$M_{t+1}(s_{t+1}) = \max\left(M_t(s_t^0) + \Gamma_t(s_t^0, s_{t+1}), M_t(s_t^1) + \Gamma_t(s_t^1, s_{t+1})\right). \tag{2.17}$$

The selected path is usually referred to as the *surviving path* and the other path is considered as the *discarded path*. Based on its elementary operators, equation (2.17) is widely recognized as the *add-compare-select* (ACS) operation. Furthermore, the VA use the ACS operation for each state $s_{t+1} \in [0;\ 2^\nu - 1]$ at instant $t+1$, resulting in 2^ν new path metrics. The VA then stores these metrics to calculate the path metrics for the next instant $t+2$. As a result, the path metrics for every instant can be calculated recursively using the same computation unit, denoted as the add-compare-select unit (ACSU). The process of calculating recursively the path metrics is the *path propagation* process.

2.3.1.2 The Traceback Process

When the path propagation has reached the end of the trellis diagram, the maximum-likelihood (ML) path can be obtained. If the trellis diagram is terminated by forcing to zero, then the surviving path at state 0 is the ML path. Otherwise, if the ending state is not known in advance, the ML path should be the path with the highest metric. Then, from the ML path, the ML state sequence and the ML codeword can be retrieved. This can be done by allowing the ACSU, while selecting the surviving paths at instant t, to store also the corresponding output bits of the branches in the trellis section t. Consequently, the ML state and ML bits can be recursively obtained from the ending state of the ML path and the stored surviving bits. However, this procedure may require a large amount of memory to store the surviving bits for every trellis section and may also introduce a large additional latency to the decoder. To this end, the VA employs a low complexity traceback.

Assuming that the surviving paths have reached the time instant t, the path convergence property of convolutional codes [35] dictates that all these surviving paths at instant t originate from a state s at time instant $(t-D)$. With the stored surviving bits, this state s can be traced back from any surviving path, and it is considered as the ML state at instant $(t-D)$. The value of D should be chosen large enough to ensure that the convergence property is satisfied with sufficiently high probability [36]. As a result, memories of only size D are necessary for storing the surviving bits of D previous trellis sections to perform the traceback.

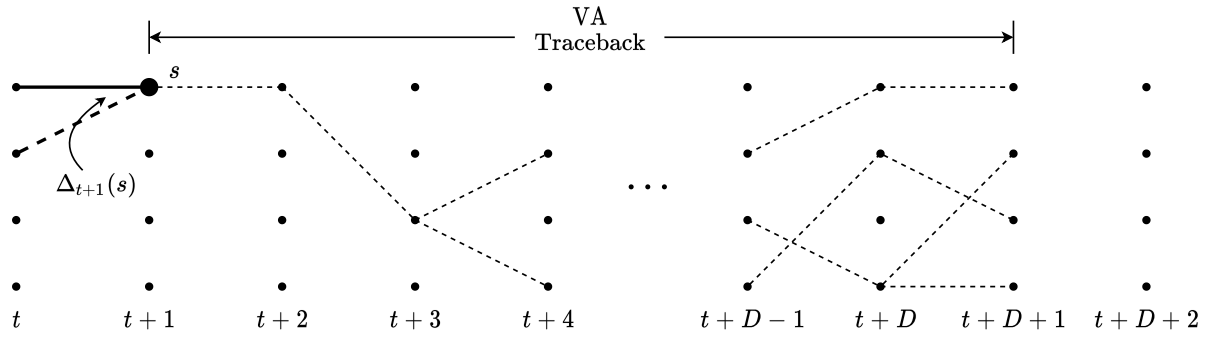
2.3.2 The Soft-Output Viterbi Algorithm

The Soft-Output Viterbi Algorithm (SOVA) was proposed by Hagenauer in 1989 in order to meet the need for a soft-output convolutional decoder [37]. In addition to the hard decisions provided by the VA, the SOVA also produces reliability values for these decisions. In general, a SOVA decoder is made of two parts: a conventional VA decoder and a reliability update unit to produce the reliability values.

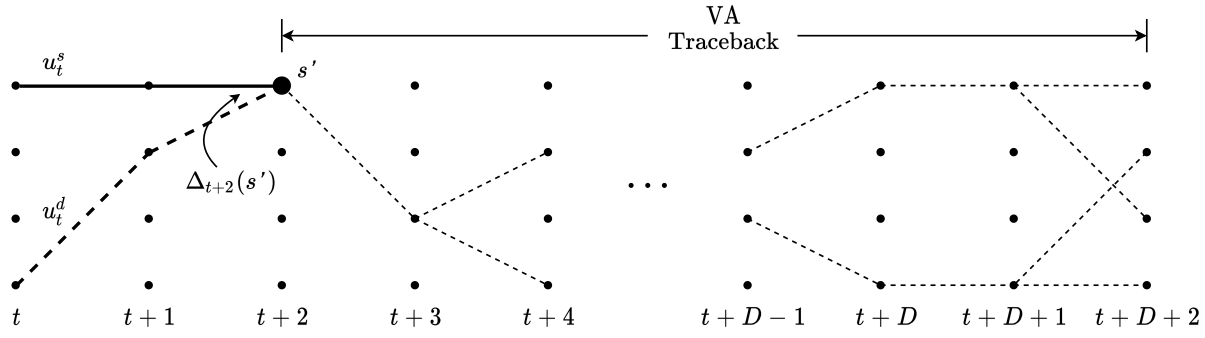
The reliability update process can be demonstrated from the perspective of the bit u_t in the trellis section t as shown in Figure 2.7. Assuming that the path propagation of the VA has reached instant (t+D+1) as depicted in Figure 2.7a, then the traceback process provides the ML state s at instant (t+1). Then, the metric difference $\Delta_{t+1}(s)$ between the surviving path and the discarded path at instant t for state s is the very first estimate on the reliability of the decision on bit u_t in section t

$$\mathrm{L}_t = \Delta_{t+1}(s). \tag{2.18}$$

Note that this metric difference can be obtained by two approaches. In an intuitive way, the values of $\Delta_{t+1}(s)$ can be stored in memory when the ACSU of the VA calculates the path metrics at instant (t+1). The second approach involves recomputing the metric difference by performing two traceback processes, one for the surviving path and one for the discarded path [38].



(a) First update where $\mathrm{L}_t = \Delta_{t+1}(s)$



(b) Second update of the reliability value of u_t using HR

Figure 2.7: SOVA process of finding the reliability value of bit u_t in trellis section t.

Next, the VA moves to instant (t+D+2) and produces the ML state s' for instant (t+2) as shown in Figure 2.7b. Similarly, the metric difference $\Delta_{t+2}(s')$ between the surviving path and the discarded path at instant (t+2) is obtained. Furthermore, denoting by u_t^s

and u_t^d the sought bit decision of the surviving path and the discarded path, respectively, the reliability of bit u_t can be updated using the following rule known as the *Hagenauer rule* (HR)

$$\mathrm{L}_t = \begin{cases} \min\left(\Delta_{t+2}(s'), \mathrm{L}_t\right), & \text{if } u_t^s \neq u_t^d, \\ \mathrm{L}_t, & \text{if } u_t^s = u_t^d. \end{cases} \tag{2.19}$$

Subsequently, as the VA continues tracking the ML state for subsequent instants, the reliability of bit u_t is be consecutively updated with the HR. Finally, the LLR of bit u_t is computed as

$$\mathrm{L}\{u_t\} = (2\hat{u}_t - 1) \times \mathrm{L}_t, \tag{2.20}$$

where $\hat{u}_t$ is the hard decision of the t-th bit provided by the VA. The quality of the reliability value of a bit in the SOVA depends on the number of updates, denoted by U. In [39], the authors proposed that for $\nu = 3$, with $U = 24$ and $D = 32$, there is no significant difference in performance compared to the global traceback and update case.

2.3.3 MAP-Based Algorithms

The decoding algorithms based on the MAP criterion are very popular alternatives for SISO decoding. Different from the Viterbi-based algorithms, these algorithms focus on finding the most likely transmitted symbol rather than finding the most likely sequence. The algorithm was proposed in 1974 to decode linear codes, and the name of the algorithm is given after the names of the authors, the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm [40].

The BCJR algorithm estimates the *a posteriori* probability (APP) for each data symbol, and then the hard decision is chosen that corresponds to the maximum APP. The implementation of the BCJR algorithm proceeds somewhat like the path propagation of the Viterbi Algorithm but in both directions over the trellis diagram. It involves two recursive processes called *forward recursion* for the propagation from left-to-right and *backward recursion* for the propagation from right-to-left in the trellis diagram, respectively. Once both recursions have been completed, the APP and the hard decision can be obtained for each data symbol. Furthermore, in practice, applying the BCJR algorithm in the logarithmic domain (Log-MAP algorithm) and possibly simplifying it using the max-log approximation (Max-Log-MAP or MLM algorithm) makes it more suitable for hardware implementation.

2.3.3.1 The BCJR Algorithm

Assuming a binary convolutional code with rate $R = 1/n$, and $\mathbf{y}$ being the received vector of length N, the APP of bit u_t in trellis section t is expressed as [40]

$$\Pr\{u_t; \mathbf{y}\} = \sum_{(s',s)|u_t} \Pr\left\{s_t = s',\, s_{t+1} = s;\, \mathbf{y}\right\}, \tag{2.21}$$

which is equivalent to the sum of probability of all the sequence having $u_t \in \{0,\, 1\}$.

The BCJR algorithm defines the following probability functions where $\mathbf{y}_i^j = [y_i,\, y_{i+1}, \ldots, y_j]$

is the sub-vector from index i to j ($0 \leq i < j \leq N-1$) of vector $\mathbf{y}$

$$\sigma_t(s',\, s) = \Pr\left\{s_t = s',\, s_{t+1} = s;\, \mathbf{y}\right\}, \tag{2.22}$$

$$\alpha_{t+1}(s) = \Pr\left\{s_{t+1} = s;\, \mathbf{y}_0^{n(t+1)-1}\right\}, \tag{2.23}$$

$$\beta_{t+1}(s) = \Pr\left\{\mathbf{y}_{n(t+1)}^{N-1} \,|\, s_{t+1} = s\right\}, \tag{2.24}$$

$$\gamma_t(s',\, s) = \Pr\left\{s_{t+1} = s;\, \mathbf{y}_{nt}^{n(t+1)-1} \,|\, s_t = s'\right\}. \tag{2.25}$$

Furthermore, the probability $\sigma_t(s',\, s)$ can be decomposed as

$$\begin{aligned}\sigma_t(s',\, s) &= \Pr\left\{s_t = s';\, \mathbf{y}_0^{nt-1}\right\} \Pr\left\{s_{t+1} = s;\, \mathbf{y}_{nt}^{N-1} \,|\, s_t = s'\right\} \\ &= \alpha_t(s') \Pr\left\{s_{t+1} = s;\, \mathbf{y}_{nt}^{n(t+1)-1} \,|\, s_t = s'\right\} \Pr\left\{\mathbf{y}_{n(t+1)}^{N-1} \,|\, s_{t+1} = s\right\} \\ &= \alpha_t(s')\gamma_t(s', s)\beta_{t+1}(s). \end{aligned} \tag{2.26}$$

As a result, from (2.21) and (2.26), the APP is

$$\Pr\{u_t; \mathbf{y}\} = \sum_{(s',s)|u_t} \alpha_t(s')\gamma_t(s', s)\beta_{t+1}(s), \tag{2.27}$$

where $\gamma_t(s', s)$, $\alpha_t(s')$, and $\beta_{t+1}(s)$ are the branch metric, the forward state metric, and the backward state metric, respectively. The branch metric $\gamma_t(s', s)$ can be derived from (2.25) as

$$\begin{aligned}\gamma_t(s', s) &= \Pr\left\{s_{t+1} = s \,|\, s_t = s'\right\} \Pr\left\{\mathbf{y}_{j=nt}^{n(t+1)-1} \,|\, s_t = s';\, s_{t+1} = s\right\} \\ &= \Pr\left\{s_{t+1} = s \,|\, s_t = s'\right\} \Pr\left\{\mathbf{y}_{j=nt}^{n(t+1)-1} \,|\, \mathbf{c}_{j=nt}^{n(t+1)-1}(s', s)\right\} \\ &= \Pr\left\{s_{t+1} = s \,|\, s_t = s'\right\} \prod_{j=nt}^{n(t+1)-1} \Pr\left\{y_i \,|\, c_j(s', s)\right\}, \end{aligned} \tag{2.28}$$

where $\Pr\left\{s_{t+1} = s \,|\, s_t = s'\right\}$ is the probability having $s_{t+1} = s$, given $s_t = s'$. Note that if no *a priori* information is given, this probability is equal for all branches (s', s) in the trellis and it can be omitted since it does not affect the final MAP decision. The branch metric in (2.28) can be calculated *locally* per trellis section from the conditional probability of the received symbol given in (2.2). Moreover, the forward state metric is computed as

$$\begin{aligned}\alpha_{t+1}(s) &= \sum_{s'} \Pr\left\{s_t = s';\, s_{t+1} = s;\, \mathbf{y}_0^{n(t+1)-1}\right\} \\ &= \sum_{s'} \Pr\left\{s_t = s';\, \mathbf{y}_0^{nt-1}\right\} \Pr\left\{s_{t+1} = s;\, \mathbf{y}_{nt}^{n(t+1)-1} \,|\, s_t = s'\right\} \\ &= \sum_{s'} \alpha_t(s')\gamma_t(s', s). \end{aligned} \tag{2.29}$$

Similarly, the backward state metric can be expanded as

$$\begin{aligned}\beta_t(s') &= \sum_s \Pr\left\{s_{t+1}=s;\, \mathbf{y}_{nt}^{N-1} \,|\, s_t = s'\right\} \\ &= \sum_s \Pr\left\{\mathbf{y}_{n(t+1)}^{N-1} \,|\, s_{t+1}=s\right\} \Pr\left\{s_{t+1}=s; \mathbf{y}_{nt}^{n(t+1)-1} \,|\, s_t = s'\right\} \\ &= \sum_s \beta_{t+1}(s')\gamma_t(s',s). \end{aligned} \tag{2.30}$$

It is important to note that the state metrics in (2.29) and (2.30) can be calculated recursively during the forward recursion and backward recursion, respectively. Once the forward and backward state metrics are available for each instant t, the final step of the BCJR algorithm is to find the MAP decision, which can be derived from (2.27) as

$$\hat{u}_t = \underset{u_t\in\{0,1\}}{\arg\max} \sum_{(s',s)|u_t} \alpha_t(s')\gamma_t(s',s)\beta_{t+1}(s). \tag{2.31}$$

Operating in the probability domain, the BCJR algorithm employs multiplications in (2.25), (2.29), (2.30) and (2.26) making it not attractive for implementation. To circumvent this shortcoming, the BCJR can be applied in logarithmic domain, resulting in the Log-MAP (LM) algorithm.

2.3.3.2 The Log-MAP Algorithm

The LM algorithm is the implementation of the BCJR algorithm in the logarithmic domain. Therefore, similarly to the BCJR algorithm, it consists of four steps: the branch metric calculation, the forward recursion, the backward recursion, and the soft-output calculation. Furthermore, instead of computing probability functions, the LM algorithm employs LLRs as the basis for every calculation, using the channel LLRs obtained from Eq. (2.4).

In the LM algorithm, the branch metric is denoted as $\Gamma_t(s',s) = \ln\gamma_t(s',s)$ and is calculated as

$$\Gamma_t(s',s) = \ln\Pr\left\{s_{t+1}=s \,|\, s_t=s'\right\} + \sum_{j=nt}^{n(t+1)-1} c_j(s',s)\mathrm{L}\{y_j|x_j\}. \tag{2.32}$$

Now, since the convolutional code is binary, each branch $(s',\, s)$ in trellis section t is attached to an information bit $u_t(s',s)$ having its value in $\{0,\, 1\}$. Assuming that the decoder has the following *a priori* information on bit u_t,

$$\mathrm{L}^a\{u_t\} = \ln\frac{\Pr\{u_t=1\}}{\Pr\{u_t=0\}}, \tag{2.33}$$

the logarithm probability of branch (s',s) is derived as

$$\ln\Pr\left\{s_{t+1}=s \,|\, s_t=s'\right\} = \ln\Pr\{u_t = u_t(s',s)\}, \tag{2.34}$$

and can be normalized as

$$\ln\Pr\left\{s_{t+1}=s \,|\, s_t=s'\right\} = \ln\frac{\Pr\{u_t=u_t(s',s)\}}{\Pr\{u_t=0\}} + \ln\Pr\{u_t=0\} \tag{2.35}$$

$$= \begin{cases} \mathrm{L}^a\{u_t\}, & \text{if } u_t(s',s)=1 \\ 0, & \text{if } u_t(s',s)=0. \end{cases} \tag{2.36}$$

Note that from (2.35) to (2.36), the term $\ln \Pr\{u_t = 0\}$ is omitted since it will be cancelled out in the subsequent calculations. As a result, equation (2.32) can be transformed into

$$\Gamma_t(s', s) = u_t(s', s) \times \mathrm{L}^a\{u_t\} + \sum_{j=nt}^{n(t+1)-1} c_j(s', s) \times \mathrm{L}\{y_j|x_j\}. \tag{2.37}$$

The forward state metric in the LM algorithm, denoted as $A_{t+1}(s) = \ln(\alpha_{t+1}(s))$, is computed by applying the logarithm function to Eq. (2.29)

$$A_{t+1}(s) = \ln\left(\sum_{s'} \alpha_t(s')\gamma_t(s', s)\right) = \ln\left(\sum_{s'} e^{A_t(s')+\Gamma_t(s',s)}\right). \tag{2.38}$$

Following [41], (2.38) can be reformulated using the *Jacobian logarithm* defined as

$$\ln(e^{\delta_1} + e^{\delta_2}) = \max^*(\delta_1,\, \delta_2) = \max(\delta_1,\, \delta_2) + \ln\left(1+e^{-|\delta_1-\delta_2|}\right), \tag{2.39}$$

where the logarithmic correction term $\ln\left(1+e^{-|\delta_1-\delta_2|}\right)$ can be easily pre-calculated and stored in a *lookup table* for implementation. Note that the Jacobian logarithm can be extended to any number of inputs greater than two by successively using (2.39). As a result, the forward state metric can be expressed as

$$A_{t+1}(s) = \max_{s'}{}^* \left(A_t(s') + \Gamma_t(s', s)\right). \tag{2.40}$$

Similarly, using the Jacobian logarithm in the backward recursion yields

$$B_t(s') = \max_{s}{}^* \left(B_{t+1}(s) + \Gamma_t(s', s)\right). \tag{2.41}$$

In terms of the soft output, the LM algorithm produces the LLR of the APP of bit u_t as

$$\mathrm{L}(u_t) = \ln \frac{\sum_{(s',s)|u_t=1} \alpha_t(s')\gamma_t(s', s)\beta_{t+1}(s)}{\sum_{(s',s)|u_t=0} \alpha_t(s')\gamma_t(s', s)\beta_{t+1}(s)}. \tag{2.42}$$

By using the max* operation, it turns into

$$\begin{aligned}\mathrm{L}(u_t) = &\max_{(s',s)|u_t=1}{}^* \left(A_t(s') + \Gamma_t(s', s) + B_{t+1}(s)\right) \\ &- \max_{(s',s)|u_t=0}{}^* \left(A_t(s') + \Gamma_t(s', s) + B_{t+1}(s)\right).\end{aligned} \tag{2.43}$$

Moreover, the hard decision of bit u_t can be derived as

$$\hat{u}_t = \begin{cases} 0, & \text{if } \mathrm{L}(u_t) < 0, \\ 1, & \text{if } \mathrm{L}(u_t) \geq 0. \end{cases} \tag{2.44}$$

2.3.3.3 The Max-Log-MAP Algorithm

The MLM algorithm is considered as a sub-optimal version of the LM algorithm but with lower complexity [41]. The algorithm uses the *max-log* approximation

$$\ln\left(e^{\delta_1} + e^{\delta_2} + \ldots + e^{\delta_k}\right) \approx \max\left(\delta_1,\, \delta_2,\, \ldots,\, \delta_k\right), \tag{2.45}$$

which discards the correction term in the max* operation and turns it into a max operation. As a result, the forward recursion, the backward recursion, and the soft-output calculation in the MLM algorithm are performed as follows

$$A_{t+1}(s) = \max_{s'} \left(A_t(s') + \Gamma_t(s', s) \right) \tag{2.46}$$

$$B_t(s') = \max_{s} \left(B_{t+1}(s) + \Gamma_t(s', s) \right) \tag{2.47}$$

$$\begin{aligned} \mathrm{L}(u_t) = & \max_{(s',s)|u_t=1} \left(A_t(s') + \Gamma_t(s', s) + B_{t+1}(s) \right) \\ & - \max_{(s',s)|u_t=0} \left(A_t(s') + \Gamma_t(s', s) + B_{t+1}(s) \right). \end{aligned} \tag{2.48}$$

In spite of a slight loss in performance compared to the LM algorithm due to the max-log approximation, the MLM algorithm gets rid of the use of lookup tables for the correction terms. Therefore, it is attractive due to its lower complexity. In practice, for turbo codes, the decoders employ mostly the MLM algorithm to decode the constituent convolutional codes.

2.4 Turbo Codes

Turbo codes were first proposed by Berrou *et al.* in 1993 [9] as convolutional parallel concatenated codes. For the past two decades, turbo codes have been adopted as channel codes in several wireless communication standards thanks to their outstanding error correction capabilities with a high degree of flexibility, in terms of block length and code rate. Notably, turbo codes were chosen for the third and fourth generations (3G and 4G) of wireless mobile telecommunications as well as WiMAX, but also in digital video broadcasting standards such as DVB-RCS/RCS2 and DVB-SH [12].

2.4.1 Turbo Encoding

A turbo encoder generally consists of two recursive systematic convolutional (RSC) encoders concatenated in parallel and separated by an *interleaver*. Figure 2.8 shows the diagram of a typical turbo encoder. Let RSC_1 and RSC_2 be the first and second RSC encoder. Then the information message $\mathbf{u}$ is encoded by RSC_1 producing the systematic part and the first parity part of the codeword, denoted by $\mathbf{c}^s$ and $\mathbf{c}^{p_1}$, respectively. On the other hand, the information message is also passed through the interleaver to produce the interleaved information $\mathbf{u}^\pi$. This sequence is then encoded by the second encoder RSC_2 to produce the second parity part of the codeword $\mathbf{c}^{p_2}$. Note that encoder RSC_2 does not produces any systematic part, since the systematic bits need to be transmitted only once. In the end, the systematic, the first parity and the second parity parts are grouped together to make the turbo codeword $\mathbf{c} = [\mathbf{c}^s, \mathbf{c}^{p_1}, \mathbf{c}^{p_2}]$. Furthermore, assuming that the code rate of RSC_1 and RSC_2 are R_1 and R_2, the code rate of the turbo code is

$$R = \frac{R_1 R_2}{R_1 + R_2 - R_1 R_2}. \tag{2.49}$$

If both RSC codes are identical with the same code rate R', then the code rate of the turbo code is $R = R'/(2 - R')$. Furthermore, if they are binary RSC codes with $R' = 1/2$, then the turbo rate is $R = 1/3$.

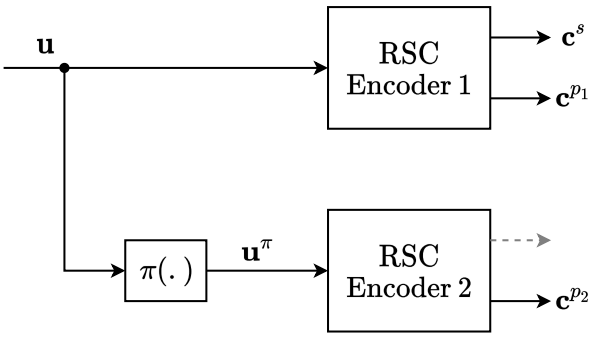


Figure 2.8: Turbo encoder as the parallel concatenation of two convolutional encoders.

Similarly to convolutional codes, high-rate turbo codes can be achieved by employing true high-rate RSC codes or by puncturing low-rate RSC mother codes. In most communication standards that employ turbo codes, the puncturing technique is chosen since it provides flexibility in changing the code rate and it allows the reuse of the same decoder.

2.4.2 The Interleaver

The interleaver is placed in-between the two RSC component encoders of the turbo code. It plays the basic role of scrambling the information message before passing it to the second RSC_2 encoder. Given the information sequence in natural order $\mathbf{u} = [u_0, u_1, \ldots, u_{K-1}]$, the interleaved sequence is $\mathbf{u}^\pi = [u_{\pi(0)}, u_{\pi(1)}, \ldots, u_{\pi(K-1)}]$, where $j = \pi(i)$ is the interleaver function. Reversely, as the interleaver function is bijective, the deinterleaver function can be defined as $i = \pi^{-1}(j)$.

The interleaver is a key component of turbo codes. Its first purpose is the time-spreading of errors that could be produced in bursts over the transmission channel. Secondly, the interleaver design has a great impact on the error correction performance of the turbo code and especially on its minimum Hamming distance.

In the literature, interleavers are generally classified into two categories: random-based interleavers and structured interleavers. The random-based interleaving function is generated using random or pseudo-random methods. Hence, their practical implementation involves the process of memorizing the addresses of the interleaving/deinterleaving functions. As a result, the required memory increases the power consumption and area cost of the turbo encoder and decoder, especially for large frame sizes.

On the contrary, for structured interleavers, the interleaving/deinterleaving functions are generally obtained by simple algebraic equations, only requiring the storage of a small set of parameters. Furthermore, their structure is more appropriate for hardware implementations of turbo decoders, and for parallel architectures if they have the contention-free property [42]. The most popular structured interleavers are *quadratic permutation polynomial* (QPP) interleavers [43] and *almost regular permutation* (ARP) interleavers [44]. The QPP interleavers have been adopted in the LTE standard and are defined as

$$\pi(i) = (f_1 i + f_2 i^2) \bmod K, \quad \forall i = 0, 1, \ldots, K-1, \tag{2.50}$$

where f_1 and f_2 are predefined interleaver coefficients for each information size. On the other hand, the ARP interleavers have been adopted in communication standards such as DVB-RSC/RSC2 and are defined as follows

$$\pi(i) = (Pi + S_{i \bmod Q}) \bmod K, \tag{2.51}$$

where P and Q are the interleaver period and the disorder degree, respectively. Also, $\mathbf{S} = [S_0, \ldots, S_{Q-1}]$ is a vector of length Q called the shift vector.

2.5 Turbo Decoding

2.5.1 Principle

The basic turbo decoder structure was proposed by Berrou in [9]. It consists of two SISO decoders connected through an interleaver and a deinterleaver, which exchange information through an iterative process. The turbo decoding process is depicted in Figure 2.9. Generally, it starts with the first SISO decoder, named SISO_1, whose inputs are the systematic part received from the channel, $\mathbf{L}^s$, the received first parity part $\mathbf{L}^{p_1}$ and the *a priori* information $\mathbf{L}^a$ provided by the other SISO decoder, if available. Then, SISO_1 computes the *a posteriori* LLRs, denoted by $\mathbf{L}$, and the *extrinsic information* $\mathbf{L}^e$ is extracted as follows:

$$L_i^e = L_i - (L_i^s + L_i^a), \forall i = 0, 1, \ldots, K-1. \tag{2.52}$$

The extrinsic information was originally proposed by Berrou as the additional but not repetitive information to be provided by a SISO decoder to the other in the iterative process. Therefore, the extrinsic information extracted from SISO_1 is interleaved and used as *a priori* information for the second decoder SISO_2.

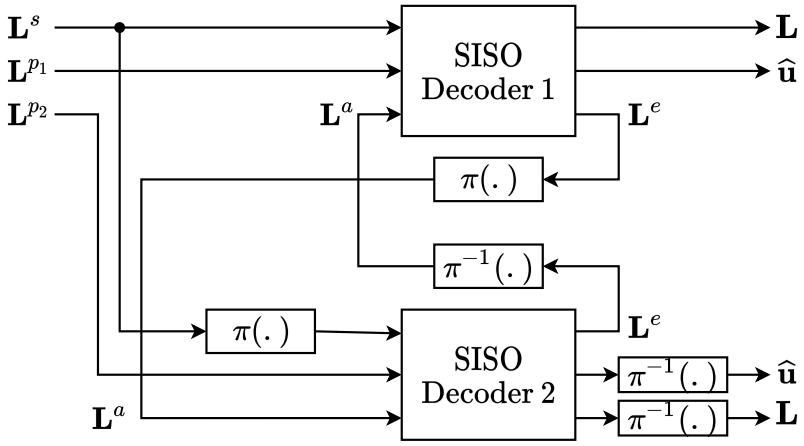


Figure 2.9: Turbo decoder consisting of two SISO decoders with the exchange of extrinsic information.

Then, SISO_2 performs a similar decoding process with inputs consisting of the interleaved systematic and the second parity parts received from the channel and of the extrinsic information provided by the first decoder and used as *a priori* information. SISO_2 decoder also computes *a posteriori* information as well as extrinsic information. Then, the extrinsic information is de-interleaved and used by the first decoder to start a second iteration.

Conventionally, the process of decoding one constituent RSC code is counted as one *half-iteration* (HI) and a complete two HIs is considered as a *full-iteration* or iteration for short. The iterative process of turbo decoding continues until a predefined number

of iterations is reached or until a stopping criterion is fulfilled. In mobile standards, the use of *cyclic redundancy check* (CRC) codes [6] to detect decoding errors can serve as the stopping criteria.

From the decoding principle, the turbo decoder is affected by the choice of the constituent SISO decoders. Different SISO decoding algorithms can be used and, thus, have different effects on the performance, complexity, throughput, etc. of the turbo decoder.

2.5.2 Comparison of Constituent Codes Decoding Algorithms

In this section, we compare the error correction performance of three candidates for the constituent SISO decoding algorithm of turbo decoders: the LM algorithm, the MLM algorithm, and the SOVA. These algorithms have been already detailed in section 2.3. Note that for turbo decoding with iterative process, the branch metric calculation is now including the *a priori* information as in equation (2.37).

In order to make the comparison, the following setting is carried out for the turbo code. First, the $(13, 15)_{\text{oct}}$ constituent RSC code of the LTE standard is chosen with generator matrix

$$\mathbf{G}_{\text{LTE}} = \begin{pmatrix} 1 & \frac{1+D+D^3}{1+D^2+D^3} \end{pmatrix} \tag{2.53}$$

and the encoder shown in Figure 2.10. We also use the QPP interleaver defined in the LTE standard. The information length is set to $K = 256$, and the code rate of the turbo code is $R = 1/3$, yielding codewords of length $N = 768$. The codewords are then transmitted using BPSK modulation through the AWGN channel. The number of iterations for turbo decoder is set to eight, and all the SISO decoding algorithms use a floating-point arithmetic representation of data.

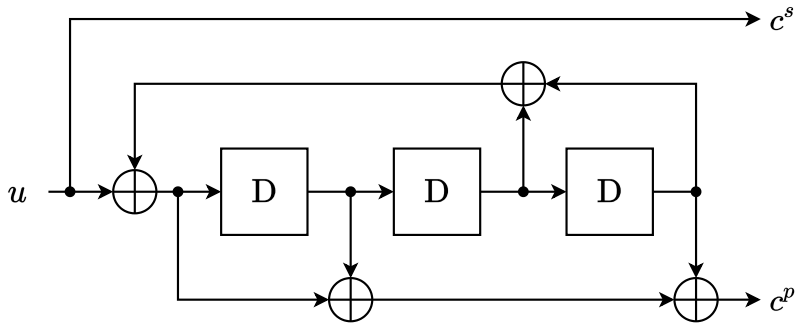


Figure 2.10: Constituent RSC encoder of the LTE standard.

Figure 2.11 shows the turbo decoder performance for the three decoding algorithms. We can observe that, at 10^{-6} BER, the LM and the MLM algorithms outperform the SOVA by a gap of 0.7 dB and 1 dB, respectively. Thus, although the SOVA is claimed to require half of the number of computational operators compared to LM and MLM [41], the interest of this algorithm for turbo decoding remains limited due to its lower performance. Furthermore, the serial processing of traceback and reliability update in the SOVA make it not inherently suitable for parallel processing.

Regarding the comparison between the LM and MLM algorithms, the LM outperforms the MLM algorithm by about 0.3 dB at the expense of higher complexity and latency due to the additional correction terms (2.39). However, MLM can be easily improved with

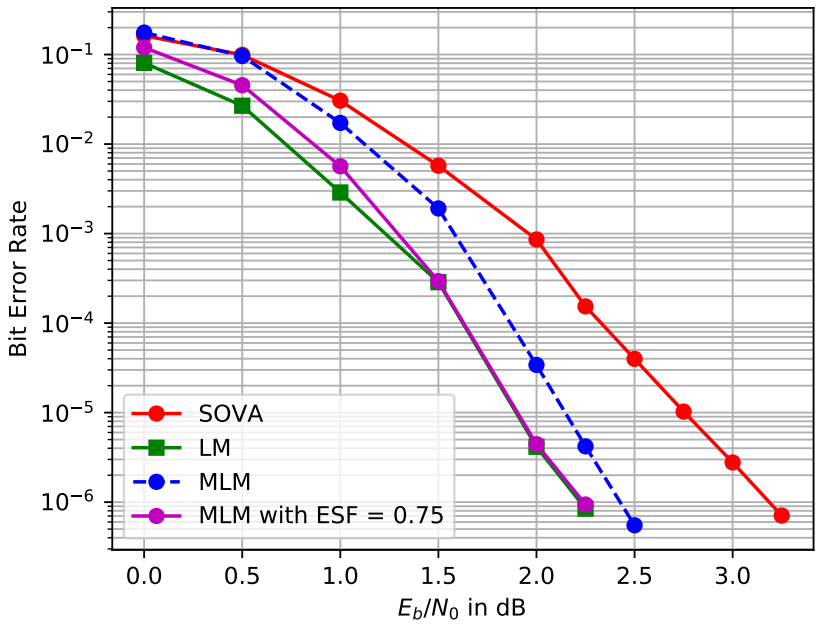


Figure 2.11: Performance comparison of decoding algorithms in turbo decoder

the introduction of an *extrinsic scaling factor* (ESF) to compensate for the overestimation of the extrinsic information caused by the max-log approximation [45]. Extrinsic scaling with the ESF allows the gap between the LM and the MLM to drop under 0.1 dB, as shown in Figure 2.11. Usually, the ESF is set to 0.75 for easy hardware implementation. Additionally, another advantage of the MLM over the LM algorithm is that the MLM is insensitive to a bad estimation of the SNR, making it more robust than the LM algorithm [46].

In conclusion, throughout this work, we will use the MLM algorithm with ESF = 0.75 as the baseline for the SISO decoders employed in turbo decoding processes.

2.6 Summary

In this chapter, we provided the necessary background information about convolutional codes and turbo codes in the context of a point-to-point digital communication chain. In particular, we described the main decoding algorithms for these codes and especially the SISO algorithms that are usually used for turbo decoding, namely the BCJR, the Log-MAP, and the Max-Log-MAP algorithms, as well as the SOVA that served as a basis for a part of our work. Furthermore, we made a brief review of turbo codes, a family of codes consisting of a parallel concatenation of two recursive systematic convolutional codes separated by an interleaver. The turbo decoding principle was explained, and we introduced the concept of extrinsic information. Finally, a performance comparison between the main SISO decoding algorithms was carried out in the context of turbo codes, which led to the choice of the Max-Log-MAP algorithm as the baseline decoding algorithm for the next chapters.

Chapter 3

High-Throughput Turbo Decoders

As presented in the previous chapter, SISO decoder plays a fundamental part of a turbo decoder architecture. However, inherently, the component SISO decoder is serial in nature due to the recursive calculation of the state metrics using the MLM algorithm. As a result, the SISO decoder requires a relatively large amount of memory for storing the state metrics, and its throughput is very limited. This, in turns, also limits the throughput of the turbo decoder. Therefore, this chapter presents several techniques and decoder architectures that have been proposed in the literature in order to design turbo decoders with a throughput up to 100 Gb/s.

The chapter starts by introducing in Section 3.1 the basic turbo decoding process with several basic techniques making a practical decoder feasible. We also define a set of useful metrics to analyze and evaluate the advantages and disadvantages of each technique or architecture. Then, in Section 3.2, techniques to increase the throughput of the decoder are presented, such as splitting the trellis into sub-trellises and processing with parallel or pipelined architectures. High-radix schemes are also considered because they allow several trellis sections to be aggregated, so as to be processed in a single clock cycle, thus further increasing the throughput. Next, ultra-high-throughput architectures that examine fully parallel or fully pipelined solutions are reviewed in Section 3.3. Finally, we summarize the contents of the chapter in Section 3.4.

3.1 Basic Turbo Decoders

The turbo decoding process is carried out through an iterative process where two component SISO decoders exchange extrinsic information. The two component decoders can process in serial (sequential decoding) or in parallel (shuffle decoding) as shown in Figure 3.1a and 3.1b, respectively. The former has been described in the previous chapter where the completion of one component decoding is referred to as one half-iteration (HI). In the latter, two SISO decoders process simultaneously, thus, each decoder acquires the *a priori* information more quickly than in the sequential decoding, which translates into a faster convergence of the decoding algorithm. The drawback of shuffle decoding is that it requires two SISO decoders in the implementation, while only one SISO decoder is employed in sequential decoding, since it can be reused for all HIs. Therefore, double the amount of hardware resources is needed for shuffle decoding. As a result, in this work, unless stated otherwise, we only consider the sequential decoding schedule.

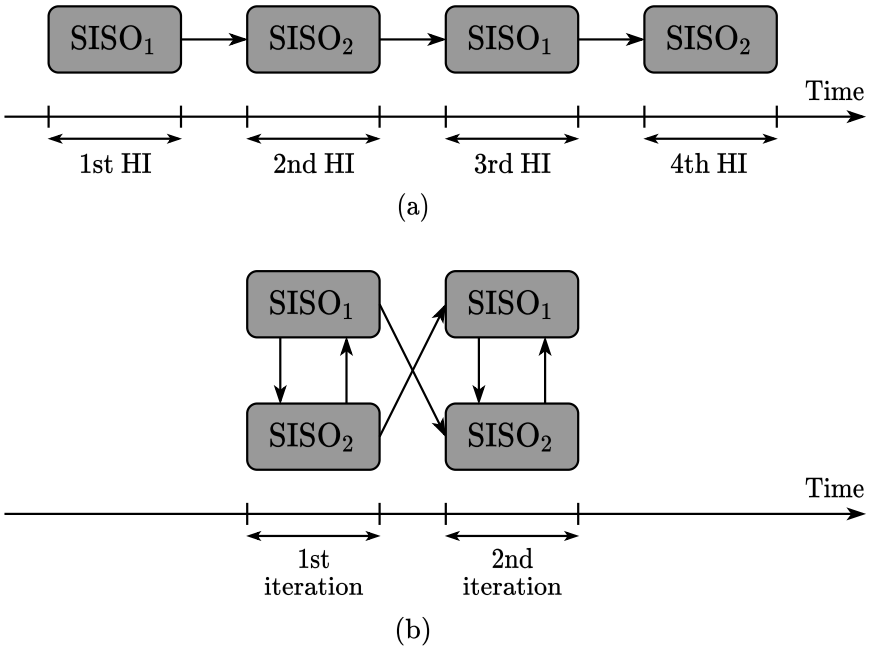


Figure 3.1: Decoding process of turbo codes with 2 iterations using (a) sequential decoding and (b) shuffle decoding

The main objective of this work is to design a very high throughput turbo decoder. Hence, the decoder throughput is considered as the main performance indicator that needs to be optimized. The decoder throughput is defined as the number of decoded bits per time unit. Since the turbo decoder consists of component SISO decoders, the throughput of the turbo decoder can easily be derived from that of a SISO decoder. Additionally, in the framework of the EPIC project, other criteria should be taken into account to validate an architectural solution such as:

- **Complexity**: it can be measured by the algorithmic computational complexity and/or architectural complexity. The former is very useful in the phase of algorithm development to provide an overview and estimate of the architectural complexity. It determines the number of elementary operations performed by the decoding algorithm. In the case of the MLM algorithm, additions and compare-select operations

are considered as the elementary operations. Also, the amount of required memory should also be counted in this phase. Then, the architectural complexity can be measured as the number of hardware resources required to implement the developed algorithm. If an application-specific integrated circuit (ASIC) is targeted, the area (in mm^2) is a direct measure of the complexity. If a field-programmable gate array (FPGA) is targeted, the complexity is measured as the number of functional units occupied by the algorithm (LUTs, RAM blocks, Flip-Flops).

- **Error correction performance**: it is measured in terms of BER and/or FER at a given SNR as explained in Section 2.1. In the context of high throughput decoding, some operations can be carried out in parallel and approximations can be introduced into the decoding algorithm, thus degrading the error performance of the decoder. If such a degradation is not tolerated, additional iterations can be required or counteracting techniques can be introduced, which translate into a higher complexity or a throughput decrease. Depending on the context, some applications accept the error performance degradation in exchange for a throughput gain.

- **Latency**: this is the elapsed time between the moment the decoder receives an input frame and the moment the decoding process has been completed. Among the use cases defined by the EPIC project [20], some applications have a very stringent latency requirement such as wireless connectivity in data centers (0.1 μs), hybrid fiber-wireless networks (0.2 μs), while others have more relaxed requirements like mobile virtual reality (500 μs) or communications through high-throughput satellites (10000 μs).

- **Flexibility**: it is characterized by the scalability of the architecture in terms of frame size and coding rate (rate compatibility). In other words, it is given by the number of code lengths and code rates that the decoder architecture can support without changing its structure.

- **Other metrics**: others metrics have also to be taken in account, such as the area efficiency (in bit/s/mm^2, throughput per unit area), the energy efficiency (in pJ/bit, energy required for decoding one information bit) or the power density (in W/mm^2, power dissipation per unit area), which must be admissible with respect to the thermal dissipation of the device.

3.1.1 Generic Components of SISO Decoders

When considering the Max-Log-MAP algorithm, the SISO decoding process involves branch metric calculations (Eq. (2.37)), a forward recursion (Eq. (2.46)), a backward recursion (Eq. (2.46)), and the soft output calculation (Eq. (2.48)). These calculations are carried out respectively by the following computation units: the branch metric unit (BMU), the forward and the backward ACSUs, and the soft-output unit (SOU). Figure 3.2a illustrates the decoding process mapped on the trellis diagram of an RSC code.

Intuitively, one can *schedule* the decoding process as shown in Figure 3.2b. First, the forward state metrics are recursively calculated by the BMU and the forward ACSU. The calculated state metrics are stored in the memory. Then, after reaching the end of the trellis diagram, the backward state metrics are calculated by the BMU and the backward ACSU. In the same time, the forward state metrics $A_k(s)$ are loaded from the memory, and

are added to the intermediate values of $(B_{k+1}(s')+\Gamma_k(s,s'))$. The sums are then fed to the SOU for calculating the *a posteriori* and extrinsic LLRs. The architecture implementing this scheduling is depicted in Figure 3.3, and the structures of the component computation units are shown in Figure 3.4. Note that the presence of the feedback loop of the ACSU for the state metrics recursion prevents this unit from being pipelined or parallelized [47]. Therefore, the ACSU contains the critical path of the decoder, which limits the maximum operating frequency.

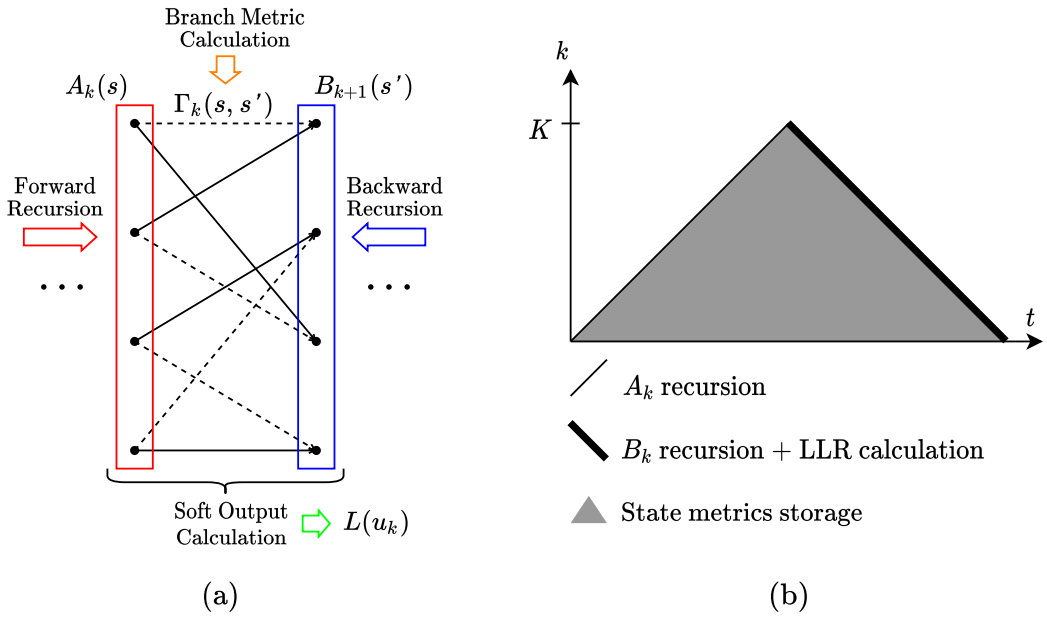


Figure 3.2: Metric calculations in the Max-Log-MAP algorithm.

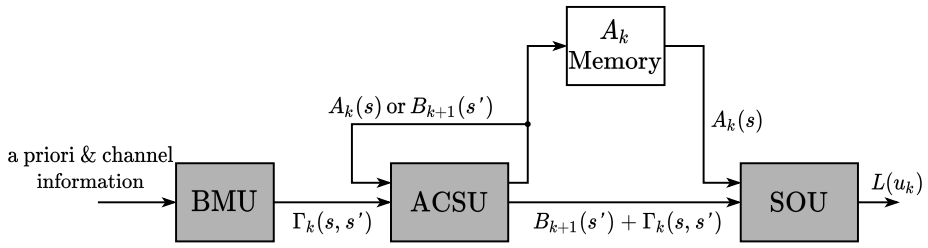
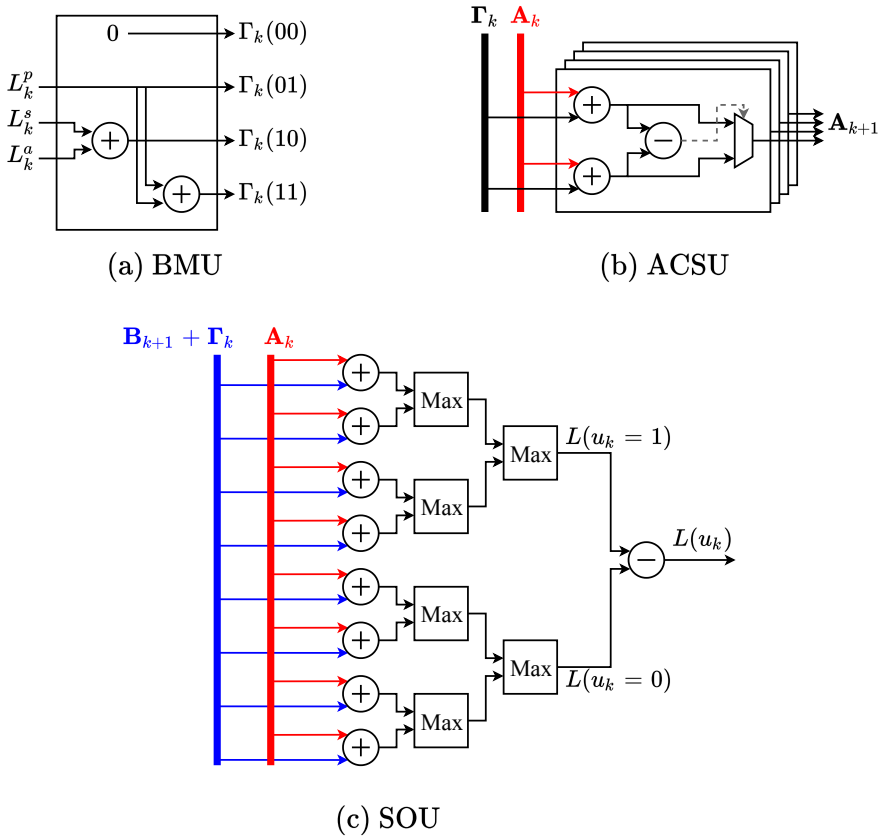


Figure 3.3: Basic SISO decoder architecture.

3.1.2 Scheduling SISO decoding

In the Max-Log-MAP algorithm, the order of execution of the forward recursion, backward recursion and soft ouput computation can vary, depending on the scheduling. There are two basic types of scheduling: *Forward-Backward* (FB) scheduling and *butterfly* scheduling [48, 49]. FB scheduling corresponds to the intuitive way of running MAP-based algorithms and has already been introduced in the previous section and illustrated by Figure 3.3(b): the forward state metrics are recursively calculated and stored in memories. Then, the backward ACSU calculates the backward state metrics, and the soft output can be computed on the fly. Another equivalent alternative is backward-forward scheduling, which reverses the order of forward and backward recursions.



(a) BMU (b) ACSU

(c) SOU

Figure 3.4: Generic computation units in a SISO decoder for a binary RSC with $\nu = 2$.

Butterfly scheluding is shown in Figure 3.5 where the forward and backward state metrics are calculated in parallel. At halfway, two SOUs are necessary for producing the soft output along with the state metric calculation in both directions.

The soft output produced by the two types of scheduling is the same; hence, they yield the same error correction performance. Furthermore, the computational complexity of both schedulings is similar and the amount of memory needed to store the state metrics is the same. However, in terms of hardware complexity, butterfly scheduling requires two ACSU in parallel and two SOUs to operate compared to only one ACSU and one SOU for FB scheduling, making it more complex. Nevertheless, butterfly scheduling inherently offers a higher throughput and a lower latency due to the fact that it can decode in K recursion steps compared to $2K$ recursion steps for FB scheduling. Therefore, the constraints and requirements of the target application dictate the type of scheduling, FB or butterfly, to be applied.

3.1.3 Sliding Window Decoding

We can observe from Figures 3.2b and 3.5 that the amount of memory required to store the state metrics of a SISO decoder increases linearly with the trellis length K, which can be a limiting factor for long frame sizes. Therefore, the *sliding window* technique was proposed [50] to overcome this drawback. The trellis of length K is split into *windows* of length W and the decoding schedule is applied at window level, thus reducing the memory requirements: the amount of storage required for the state metrics is linear with

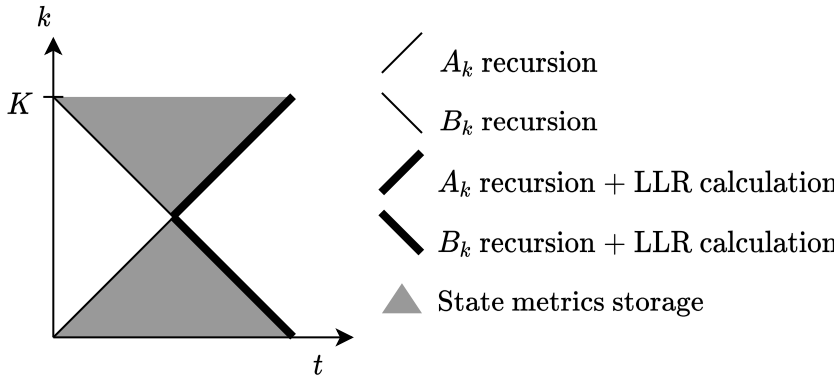


Figure 3.5: Butterfly scheduling.

W intead of K. Figure 3.6 shows the overall decoding schedule with sliding windows for both considered types of scheduling. Note that there is a continuity of forward recursions between consecutive windows, while the data dependency of the backward recursions is loosened due to the discontinuity at the window borders. This raises the question of the initialization of the backward metrics for each window.

The sliding window technique can be coupled with either FB scheduling or butterfly scheduling as depicted in Figure 3.6a and 3.6b, respectively. We can observe that, for both types of scheduling, the amount of storage is reduced by a factor of K/W. Furthermore, for FB scheduling, using sliding window reduces the decoding latency from $2K$ recursion steps down to $(K+W)$ steps. For butterfly schedule, the decoding latency with sliding window is still K recursion steps. From the memory requirement and latency points of view, a low window size W is better. However if the window is too small, the error correcting performance of the decoder can be degraded [51]. For applications where the value of K/W is relatively large, sliding window is usually employed with FB scheduling since it generates nearly the same throughput and latency while having lower complexity than butterfly scheduling.

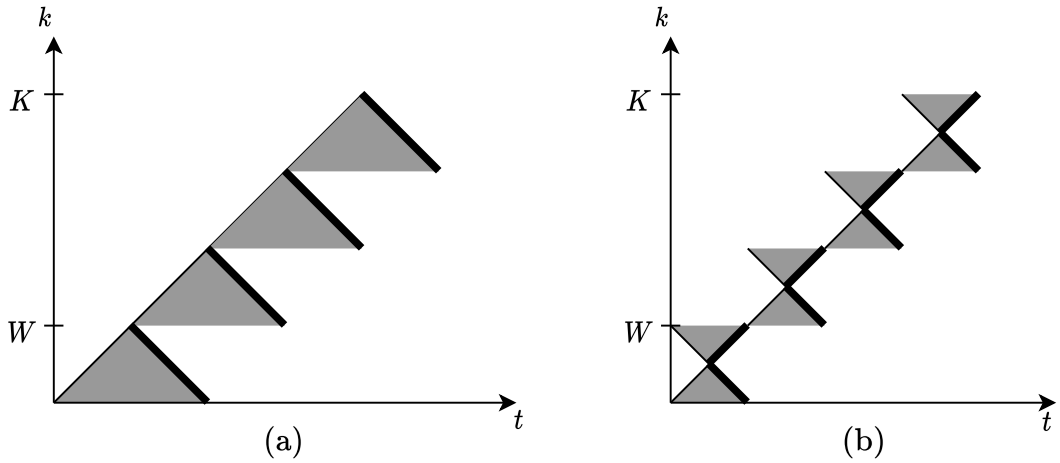


Figure 3.6: Decoding with sliding windows
(a) FB scheduling (b) butterfly scheduling.

3.1.4 State Metric Initialization

If the backward state metrics at the border of the sliding windows are initialized with equiprobable values, this can result in poor error correction performance of the decoder, especially if W is low. Therefore, initializing the state metrics with reliable values at the window border is necessary. Two main techniques can be found in the literature [48]: *Acquisition* (ACQ) and *next iteration initialization* (NII).

The use of ACQ initialization combined with FB scheduling and sliding window is shown in Figure 3.7a, in dashed line: L_{ACQ} recursion steps are performed prior to the regular resursion steps to initialize the state metric values at the beginning of each window. The ACQ step can be regarded as a tool to enrich the backward state metrics before the decoder starts producing the soft outputs. Using the ACQ stage prevents the error correction performance of the decoder to be degraded, at the expense of increased hardware complexity, due to an additional BMU and ACSU. This is all the more important for high coding rates R obtained with puncturing. This can be explained by the fact that the frequency of parity bits in the punctured codeword is low for high coding rates. Therefore, in this case, the ACQ length L_{ACQ} should cover at least several non-punctured parity bit positions so that the resulting state metric is reliable enough.

NII is another technique to obtain good estimates for the initial state metrics as shown in Figure 3.7b. The principle is based on storing the last calculated state metric at the end of a window and using it as the initial value for the next window at the next iteration. Therefore, at the cost of extra memory to store the final state metric values, the error correction performance can be improved without extra processing unlike in the case of ACQ. On the other hand, since the initial state metrics cannot benefit from NII at the first iteration, one additional iteration can be needed to obtain the same performance as with ACQ. Therefore, a combination of ACQ and NII can be employed and is particularly useful for punctured high rate codes to limit the number of iterations. Figure 3.7c depicts an example of combining ACQ and NII in a turbo decoding process.

3.1.5 Evaluation of Basic Key Performance Indicators for Turbo Decoders

This section shows the analytical evaluation of basic key performance indicators (KPI) for turbo decoders using the MLM SISO decoding algorithm. The basic turbo decoder uses sequential decoding as shown in Figure 3.1 with only one SISO decoder for different half-iterations. This means that the LLRs output from the channel and the extrinsic information from each half-iteration are stored in the memory. For each half-iteration, the SISO decoder reads those values from the memory and performs the MLM decoding algorithm. Furthermore, we assume that the SISO decoder performs FB scheduling with sliding window and that the state metrics are initialized using combined ACQ and NII as shown in Figure 3.7c.

First of all, in terms of throughput and latency, we assume the maximum operating frequency is f and it is limited by the critical path of the ACSU. Then, by neglecting the I/O latency, the latency of the decoder can be roughly estimated as

$$\text{Latency} = \frac{1}{f} \times (K + W) \times n_{\mathrm{HI}}, \tag{3.1}$$

where n_{HI} is the number of half iterations performed by the decoding process. The

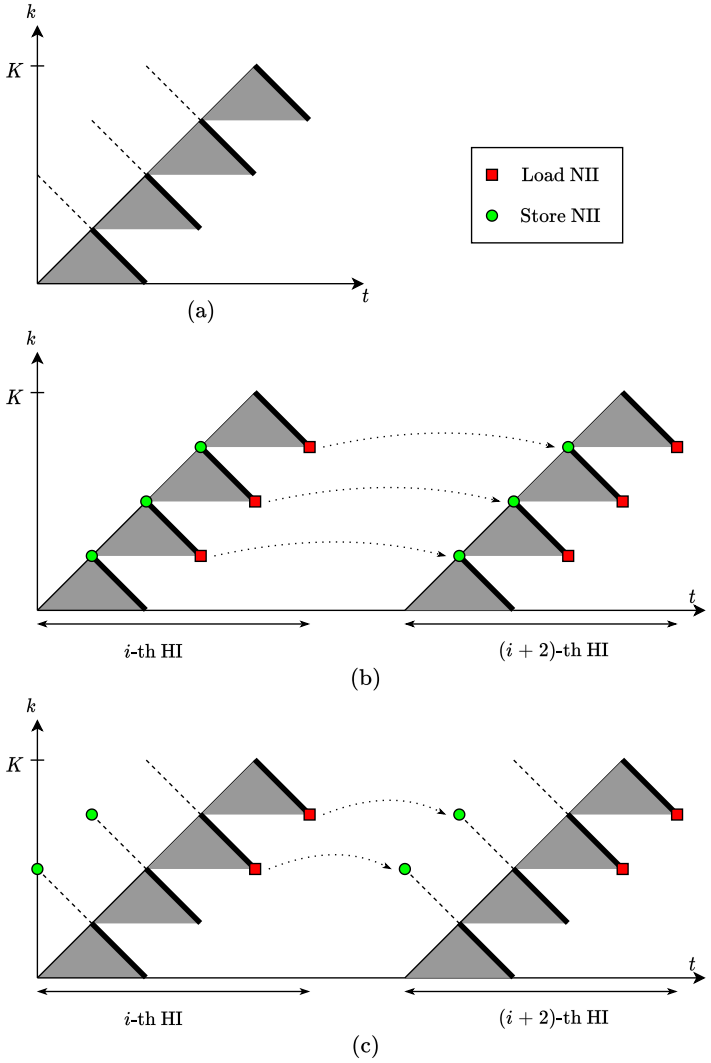


Figure 3.7: State metric initialization by (a) ACQ, (b) NII, and (c) combining both methods.

throughput of the decoder is then expressed as

$$\text{Throughput} = \frac{K}{(K+W) \times n_{\text{HI}}} \times f. \tag{3.2}$$

Note that one can increase the throughput by decreasing W or n_{HI} but it will yield a degradation in the error correction performance. Furthermore, we can see that the decoder throughput is a monotonic function increasing with the frame size K. Asymptotically, the throughput of the decoder as K goes to infinity is

$$\lim_{K \to \infty} \frac{1}{1 + W/K} \times \frac{f}{n_{\text{HI}}} = \frac{f}{n_{\text{HI}}}, \tag{3.3}$$

and is bounded by the operating frequency f. For a turbo decoder with 6 iterations ($n_{\text{HI}} = 12$), with the operating frequency f of 800 MHz, the asymptotic throughput of this decoder is about 67 Mb/s.

The algorithmic computational complexity of each decoded bit in the component SISO decoder in each half iteration is shown in Table 3.1 for a binary RSC code with rate 1/2

and ν memory elements. Note that for each decoded bit, the branch metric calculation and the ACS operation are performed three times (ACQ, forward and backward), and the soft output calculation is performed once. For implementation, this basic turbo decoder requires 3 BMU, 3 ACSU and 1 SOU with W memory units for the state metrics. Note that these computation and memory units are reused for each half iteration in a sequential decoding fashion, thus, resulting in a relatively low complexity decoder. Furthermore, this implementation can support any frame size that is a multiple of the window size W, and can support any code rate with puncturing. Intuitively, one can increase the throughput

	Addition	**Comparison-selection**	**Total**
Branch metrics calculation (forward & backward)	2	0	2
State metrics recursion (forward & backward)	$2\times2^{\nu}$	2^{ν}	$3\times2^{\nu}$
Soft-output calculation	$2\times2^{\nu}+1$	$2\times(2^{\nu-1}-1)$	$3\times2^{\nu}-1$
Computational complexity per decoded bit	$8\times2^{\nu}+7$	$4\times2^{\nu}-2$	$12\times2^{\nu}+5$

Table 3.1: Computational complexity per decoded bit of the MLM algorithm with FB scheduling, sliding window and ACQ according to Figure 3.7c. Convention for complexity: one addition equals one compare-select operation and is counted as one computational unit.

of the decoder by applying direct spatial parallelism, i.e. by using multiple instance of the basic decoder running in parallel. However, this method does not reduce the latency and the area efficiency (throughput/complexity) remains unchanged. Therefore, in the subsequent sections, we introduce alternative parallelism techniques to increase the throughput of the decoder, at the architectural and algorithmic levels.

3.2 High-Throughput Architectures

The basic turbo decoder architecture considered in the previous section can only achieve a throughput of tens of Mb/s [47, 52]. However, due to the increasing throughput demand over time, new techniques and architectures have to be devised. The literature on parallel architectures for turbo codes mainly considers splitting the trellis into sub-trellises and then processing it using a parallel MAP (PMAP) architecture architecture [53] or the so-called XMAP architecture [54]. On top of that, the algorithmic parallelism technique consisting in processing several trellis sections at the same time and called *high-radix* processing [55] is also considered to help further increase the decoding throughput.

3.2.1 The PMAP Architecture

A straightforward method to help increase the throughput is to divide the trellis into sub-trellis of length K_P, and having each sub-trellis decoded independently in parallel. This architecture is generally referred to as the PMAP architecture [53]. Denoting by $P = K/K_P$ the number of sub-trellises, by neglecting the I/O latency and the latency

due to metric initialization, the latency of the PMAP architecture can be estimated in terms of clock cycles as

$$\text{Latency}_{\text{PMAP}} = (K_P + W) \times n_{\text{HI}}, \tag{3.4}$$

and the throughput of the PMAP architecture is

$$\text{Throughput}_{\text{PMAP}} = \frac{K}{(K_P + W) \times n_{\text{HI}}} \times f. \tag{3.5}$$

With a fixed frame size K, the throughput increases inversely with K_P, i.e. it increases with the number of parallel sub-trellises P. As K goes to infinity, the asymptotic throughput of the PMAP architecture is

$$\lim_{K \to \infty} \frac{1}{(K_P/K + W/K)} \frac{f}{n_{\text{HI}}} = \frac{Pf}{n_{\text{HI}}}. \tag{3.6}$$

As an example, with $P = 64$ independent SISO cores, $f = 800$ MHz and $n_{\text{HI}} = 12$ (6 iterations), the asymptotic throughput of the PMAP architecture is about 4.2 Gb/s.

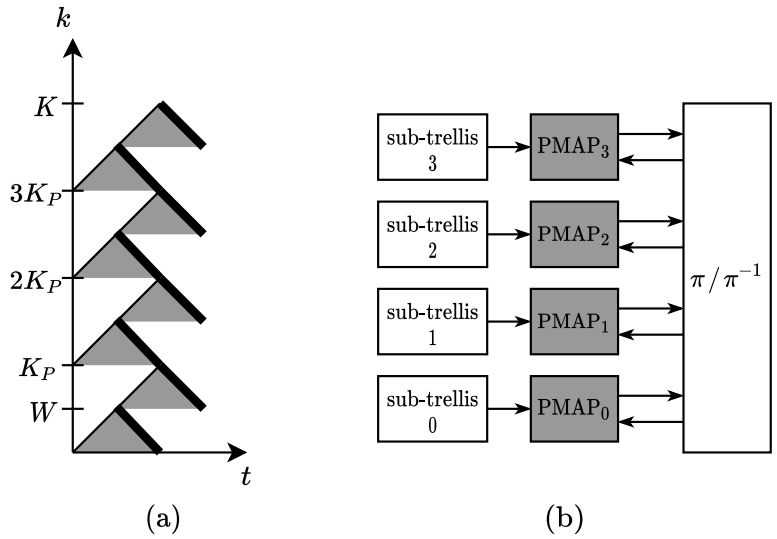


Figure 3.8: The PMAP architecture with $P = 4$.

On the other side, to implement the PMAP architecture, the amount of hardware resources increases by a factor P since each sub-trellis is processed by an independent SISO decoder. Figure 3.8 shows the scheduling and the architectural overview of the PMAP decoder. Note that Figure 3.8a shows the PMAP employing sliding window and FB scheduling, but the butterfly scheduling can also be used instead. Furthermore, NII and/or ACQ initialization can be used to compensate for the loss in data dependency due to sub-trellis division, thus preserving the inherent error correction performance of the turbo decoder.

3.2.2 The XMAP Architecture

Given P sub-trellises of length K_P, the XMAP architecture [54] processes a *single* sub-trellis at a time. Sub-trellises are time-multiplexed into a *pipeline* consisting of a chain of

computation units (BMU, ACSU and SOU) connected through pipeline registers. As a result, for each clock cycle, the decoder can produce K_P soft-output values. Note that the XMAP architecture only differs from the PMAP architecture in the way each individual sub-trellis is processed.

An example of the XMAP architecture with 4 sub-trellises is shown in Figure 3.9 with the scheduling on the left and the architectural overview on the right. Furthermore, the X-shaped architecture of the XMAP core is shown in Figure 3.10. By neglecting the I/O latency and the latency due to metric initialization, the latency in clock cycles of the turbo decoder with XMAP architecture is

$$\text{Latency}_{\text{XMAP}} = (K_P + P - 1) \times n_{\text{HI}}, \tag{3.7}$$

and the throughput is

$$\text{Throughput}_{\text{XMAP}} = \frac{K}{(K_P + P - 1) \times n_{\text{HI}}} \times f. \tag{3.8}$$

Since the XMAP core consists of a chain of computation units setting up in a pipeline fashion, its complexity increases linearly with the sub-trellis length K_P. Particularly, given the architecture shown in Figure 3.10, the XMAP core requires $2K_P$ BMUs, $2K_P$ ACSUs, K_P SOUs, as well as the pipeline registers for the state metrics, the channel and the *a priori* information. Furthermore, if ACQ initialization is employed, the pipeline registers and the number of BMUs and ACSUs will increase with the ACQ length.

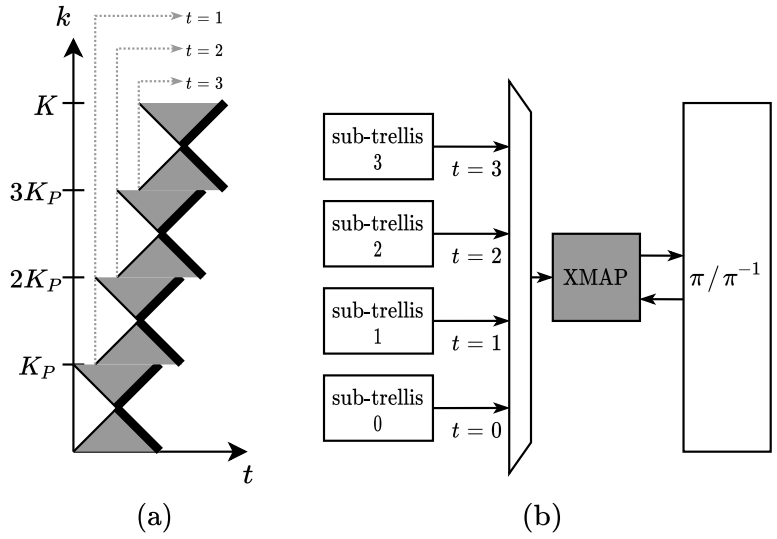


Figure 3.9: The XMAP decoder with $P = 4$ with (a) the scheduling with the delay of 1 clock cycle between sub-trellis processing and (b) the architectural overview of the corresponding XMAP decoder.

3.2.3 High-Radix Schemes

For SISO decoding algorithms based on the trellis diagram, like the SOVA and the MAP-based algorithms, the state metrics are calculated *recursively* by the ACSU. As a result, the decoder processes one trellis section per clock cycle. This is a *bottleneck* for a high

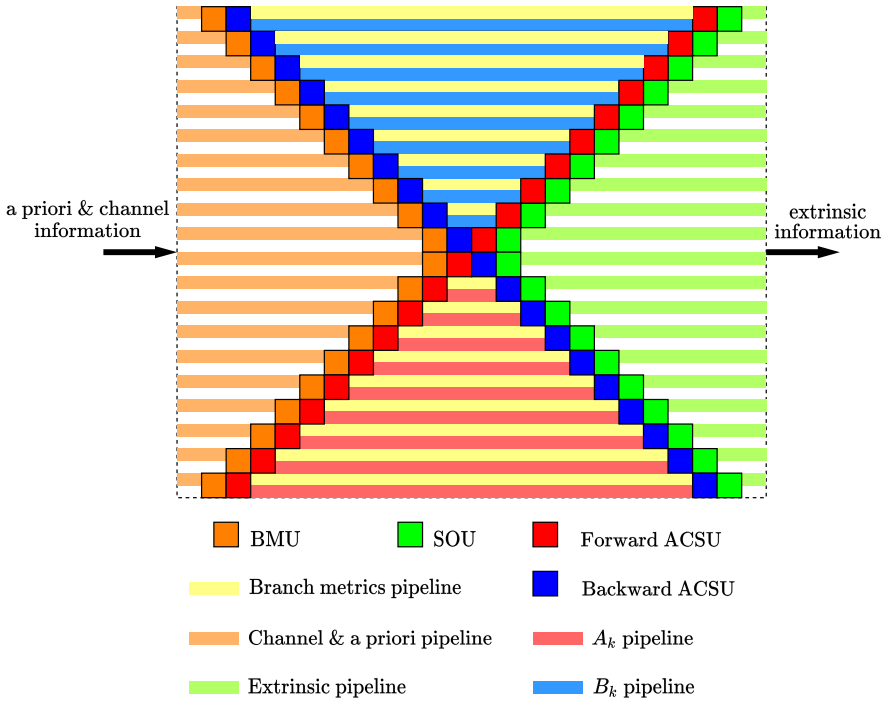


Figure 3.10: The XMAP core comprising chain of computation units and pipeline registers.

throughput implementation of SISO decoding algorithms. To overcome this problem, the authors in [55] proposed a high-throughput computation of the state metrics by aggregating successive trellis sections. This technique was first proposed for the Viterbi Algorithm and then was applied to the SOVA and the MAP-based algorithms.

For a binary RSC code with rate $1/n$, the original trellis is in *radix-2* form since there are two branches coming in and out of a state. Then, the aggregation of two successive radix-2 trellis sections results in a *radix-4* trellis section as there are four branches coming in and going out of a state. Similarly, the aggregation of three and four radix-2 trellis sections leads to a radix-8 and radix-16 trellis section, respectively. Figure 3.11 shows an example of a radix-4 trellis section of a binary RSC code with $\nu = 2$.

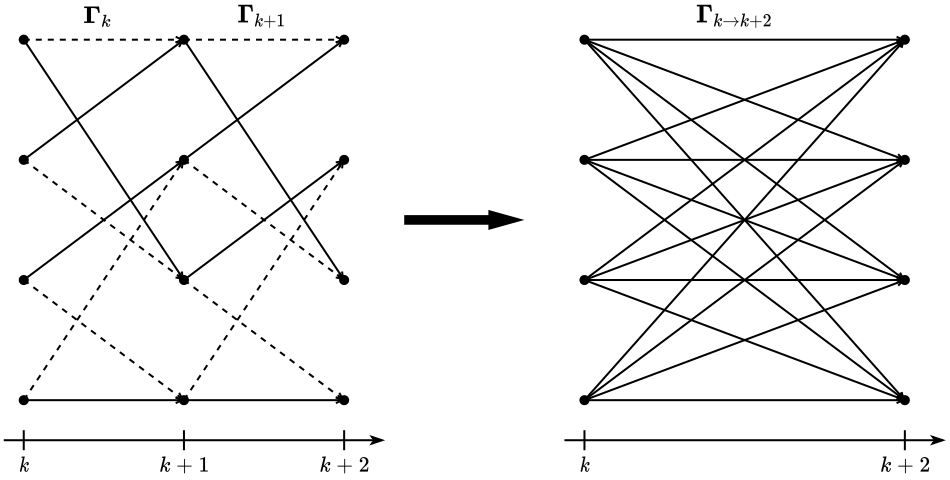


Figure 3.11: A radix-4 trellis section made of two consecutive radix-2 trellis sections.

Denoting by T the *radix order*, the forward state metrics calculation of the radix-2^T ($T > 1$) scheme for the Max-Log-MAP algorithm is obtained as follows

$$A_{k+T}(s) = \max_{s'} \big(A_k(s') + \Gamma_{k \to k+T}(s', s)\big), \tag{3.9}$$

where $\Gamma_{k \to k+T}(s', s)$ is the radix-2^T branch metric connecting state s' at instant k and state s at instant $k+T$, and is calculated as the accumulation of T component radix-2 branch metrics. As a result, a high-radix BMU has to compute all the combinations of the component radix-2 branch metrics to obtain the high-radix branch metrics provided to the ACSU. Furthermore, the *a posteriori* LLR of bit u_k in radix-2^T is calculated as

$$\begin{aligned} \mathrm{L}(u_k) = &\max_{(s',s)|u_k=1} \left(A_k(s') + \Gamma_{k \to k+T}(s', s) + B_{k+T}(s)\right) \\ &- \max_{(s',s)|u_t=0} \left(A_k(s') + \Gamma_{k \to k+T}(s', s) + B_{k+T}(s)\right), \end{aligned} \tag{3.10}$$

where the set of branches (s', s) for each max operator has $2^{\nu+T-1}$ elements, which increases exponentially with the radix order.

In terms of advantages, using radix-2^T schemes produces T soft-output in a single clock cycle. Hence, ideally, it helps increase the throughput of the decoder by a factor T while keeping the same error correction performance. However, radix-2^T schemes require an increase in complexity in terms of computation units (BMU, ACSU, SOU) as shown in (3.9) and (3.10). Furthermore, as shown in (3.9), the size of the set of state for the max operation in the ACSU increases with the radix order. This induces an increase in the critical path of the ACSU, which in turns lowers the maximum operating frequency, thus decreasing the throughput of the decoder.

For the MLM algorithm, the use of radix-4 has been demonstrated to provide a higher throughput than radix-2 schemes at the expense of a higher decoder complexity for the LTE turbo code [56]. However, radix orders greater than 2 (radix-8, radix-16 and above) were shown to be inefficient due to the increased complexity and longer critical path [57].

3.2.4 High-Throughput Turbo Decoders: State of the Art

Table 3.2 provides an overview of state-of-the-art turbo decoder implementations reported in the literature, which use the high-throughput techniques described in the previous sections. This table shows that, by employing the PMAP or XMAP architecture with or without the use of high-radix schemes, turbo decoders can achieve data rates in the order of several Gb/s.

3.3 Ultra-High-Throughput Architectures

Previous architectures and techniques can support applications with throughput requirements up to several Gb/s. However, in the context of the EPIC project, throughputs of tens of Gb/s up to hundreds of Gb/s are required in several use cases. As far as we know, there are two architectures that can meet such high-throughput constraints: the fully-parallel MAP [64] and the unrolled-XMAP architectures.

[1] Throughput measured at the maximum number of iterations

Reference	[58]	[59]	[60]	[60]	[61]	[62]	[63]
Code - Flexibility	LTE-A	LTE-A	LTE-A	LTE-A	LTE-A	$K = 4096$	LTE-A
Max block size	6144	6144	6144	6144	6144	4096	6144
Architecture	PMAP	PMAP	PMAP	PMAP	PMAP	PMAP	XMAP
Radix/P	4/16	4/32	2/64	2/64	2/64	16/32	4/192
Window size	32	192	96	96	64	32	32
No. iterations max	5.5	6	8	5.5	6	8	7
Technology (nm)	65	65	90	90	65	90	28
Throughput (Gb/s)[1]	**1.0**	**2.2**	**2.3**	**3.3**	**1.3**	**1.4**	**1.1**
Frequency (MHz)	410	450	625	625	400	175	625
Area (mm^2)	2.5	7.7	19.8	19.8	8.3	9.6	0.49
Area efficiency ($Gb/s/mm^2$)	0.41	0.3	0.1	0.2	0.2	0.1	2.3

Table 3.2: Comparison of high-throughput turbo decoder implementations.

3.3.1 The Fully-Parallel MAP (FPMAP) Architecture

The FPMAP architecture combines shuffle decoding with a splitting of the trellis into sub-trellises of size $K_P = 1$, which can be seen as an extreme case of the PMAP architecture with $P = K$. The approach was first presented in [64] and an implementation based on the LTE turbo codes was published in [65].

A schematic of the FPMAP architecture is shown in Figure 3.12. It employs $2K$ processing elements (PE). Each PE computes the branch metrics, the forward and backward state metrics and the extrinsic information for one trellis step in one clock cycle. The calculated state metrics at the border are then exchanged with neighbouring PEs, and the calculated extrinsic information is fed to the interleaved/deinterleaved PEs.

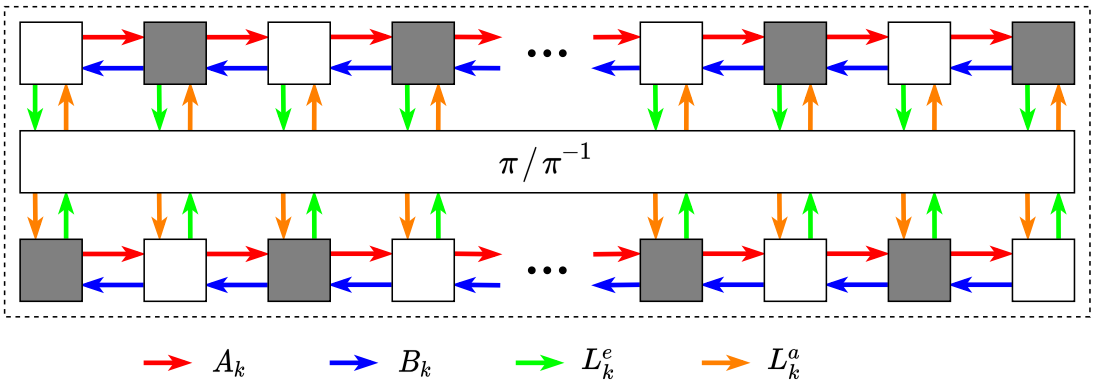


Figure 3.12: The FPMAP architecture.

For the LTE turbo code, the use of the odd-even QPP interleaver helps split the PEs into two groups, where each group can be processed independently. The first group

consists of $K/2$ PEs of the SISO_1 with even index and $K/2$ PEs of the SISO_2 with odd index (shown by grey blocks in Figure 3.12). The second group is made of $K/2$ PEs of the SISO_1 with odd index and $K/2$ PEs of of the SISO_2 with even index (shown by white blocks in Figure 3.12). Then, one complete iteration is the decoding process of a group, with the alternation of the first and the second group. This setting halves the implementation complexity [64] since one PE can be used to process two consecutive trellis sections. However, this setting also halves the throughput of the decoder as it now takes two clock cycles to finish an iteration. By neglecting the I/O latency, the decoder latency in clock cycles of the FPMAP is

$$\text{Latency}_{\text{FPMAP}} = 2 \times n_{\text{I}}, \tag{3.11}$$

where n_{I} is the number of iterations. The throughput of the FPMAP decoder is calculated as

$$\text{Throughput}_{\text{FPMAP}} = \frac{K \times f}{2 \times n_{\text{I}}}, \tag{3.12}$$

In [65], the authors showed that for $K = 6144$, the throughput of the FPMAP with 39 iterations and a clock frequency of 100 MHz is 15.8 Gb/s in 65 nm CMOS. Furthermore, with this implementation the area complexity of the FPMAP decoder is 109 mm^2, thus, the area efficiency is 0.145 Gb/s/mm^2. Therefore, although the FPMAP can achieve a high throughput, the price to pay is a decrease in area efficiency compared to the PMAP architectures [58, 59] shown in Table 3.2.

Moreover, other drawbacks of the FPMAP architecture should be the lack of flexibility with respect to code block sizes. Furthermore, the combination of sub-trellis size of 1 and shuffle decoding degrades the error correction performance of the decoder. At low code rates such as $R = 1/3$, the FPMAP requires 16 to 32 iterations to compensate for the performance loss. However, for high code rates such as 8/9 and 18/19, it is shown in [57] that the number of iterations increases drastically up to 80.

3.3.2 The Iteration Unrolled XMAP (UXMAP) Architecture

Starting from the XMAP core shown in Figure 3.10, using multiple cores in parallel and unrolling the iterations of the turbo decoding process leads to the fully-pipelined UXMAP architecture shown in Figure 3.13. Assuming a completely filled pipeline, by neglecting the I/O latency and the latency due to the metric initialization, the decoder produces a complete decoded frame of K bits per clock cycle, thus resulting in the following throughput expression:

$$\text{Throughput}_{\text{UXMAP}} = K \times f. \tag{3.13}$$

Note that this throughput is only limited by the frame size and the achievable clock frequency. In terms of flexiblity, the UXMAP architecture is rate-flexible with puncturing, and the frame-flexible aspect of the UXMAP was treated in [66].

Figure 3.14 shows the UXMAP architecture in more details, displaying the different computations units and the pipeline registers. For a radix-2 sub-trellis of size K_P, each XMAP component employs $2K_P$ BMUs, K_P ACSUs for the forward state metrics, K_P ACSUs for the backward state metrics, and K_P SOUs for the computation of the extrinsic information. Besides, pipeline registers are employed extensively to carry the channel

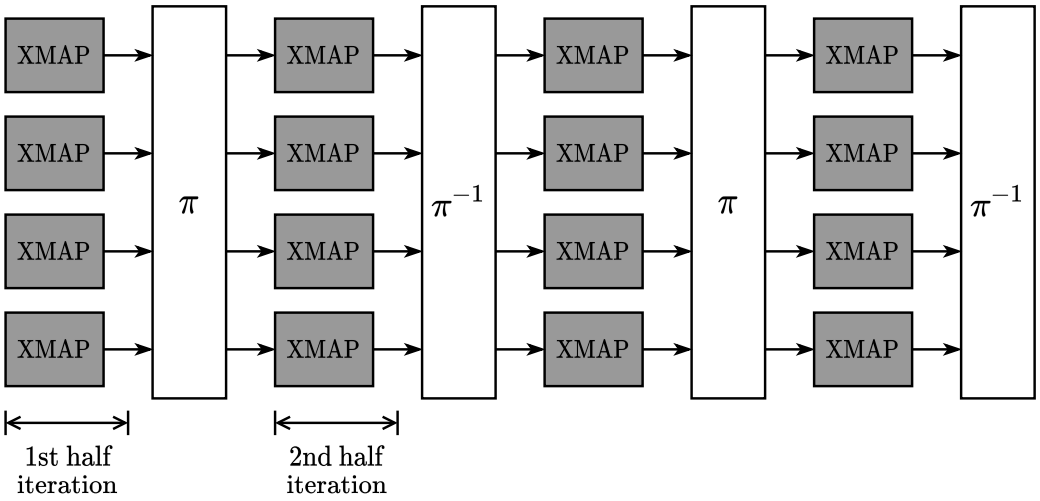


Figure 3.13: The UXMAP architecture with two iterations.

values, the calculated state metrics and the extrinsic information throughout the UXMAP decoder. The latency of the decoder can be derived in clock cycles from Figure 3.14 as

$$\text{Latency}_{\text{UXMAP}} = n_{\text{HI}} \times (T_{\text{BMU}} + K_P + T_{\text{SOU}}), \tag{3.14}$$

where T_{BMU} and T_{SOU} are the number of clock cycles required for processing the first BMU and the last SOU, respectively. Moreover, the ACSU takes one clock cycle to finish a trellis section, thus, it takes K_P clock cycles to finish a sub-trellis. Furthermore, the interleaving and the deinterleaving functions of the UXMAP architecture are hardwired between the XMAP components [57], so that it provides no processing latency during extrinsic exchange between HIs in the decoder.

Employing high radix schemes in the fully pipelined UXMAP architecture has a particular impact. Increasing the number of trellis sections processed in a clock cycle leads to a reduction in the number of pipeline stages for all the pipeline registers (state metrics, channel and extrinsic information). Hence, using radix-4 instead of radix-2 yields an area saving of about 50% and halves the overall pipeline latency. However, as we increase the radix order, the area overhead in the computation units also increases. For Max-Log-MAP decoders using radix-8 and radix-16 processing, this overhead supersedes the area savings in the pipelines, making the use of radix greater than 4 unattractive [57].

Table 3.3 shows the comparison between the FPMAP and the UXMAP architectures both implemented in 28 nm CMOS technology and reported in [57, 66]. For $K = 128$, the UXMAP can achieve a throughput of 102.4 Gb/s with an area of 23.61 mm^2 with only one XMAP core per HI [57], and with an area of 16.52 mm^2 with 4 XMAP cores in parallel per HI [66]. The FPMAP, on the other hand, can only achieve 39.86 Gb/s in throughput with similar area consumption (24.09 mm^2) for $K = 6144$. Beside, for $K = 128$, the FPMAP can only achieve 1.6 Gb/s with an area consumption of 1.04 mm^2. Therefore, in terms of area efficiency, the UXMAP architecture outperforms the FPMAP architecture by a factor of 2.6 – 4.0.

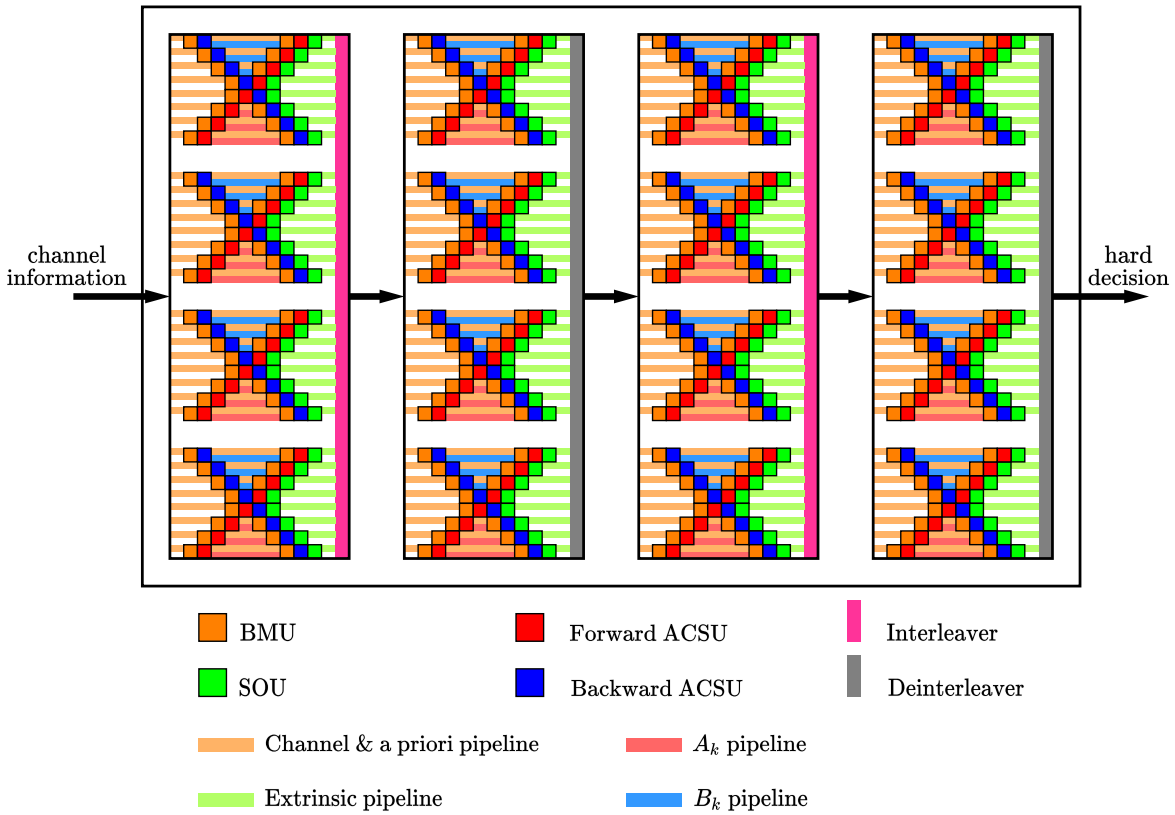


Figure 3.14: The detailed UXMAP architecture with 2 iterations (4 HIs) and with 4 XMAP cores per HI. The area complexity of the decoder can be reduced with the increasing number of XMAP cores employed in parallel [67].

3.4 Summary

This chapter presents an overview of the techniques required for the hardware implementation of turbo decoders, such as scheduling, sliding window, and state metric initialization. When high data rates are needed, the throughput of the turbo decoder can be increased by employing various parallel or pipelined architectures like PMAP, XMAP, FPMAP, and UXMAP, combined with a high-radix processing of the code trellis. As a result, depending on the chosen techniques and architecture, the throughput of a turbo decoder can range from hundreds of Mb/s up to 100 Gb/s.

Moreover, a set of metrics have been defined in order to evaluate these architectures and several turbo decoder implementations reported in the literature have been provided. Although we focus on the throughput metric in this work, one should also take account of the complexity and the latency of the decoder. To this end, it has been shown that the UXMAP architecture combined with a radix-4 processing of the trellis sections can achieve 102.4 Gb/s with a relatively high area efficiency of 4.32 Gb/s/mm^2. Therefore, the UXMAP was chosen as our main architecture template for our work towards the design of an ultra high-throughput turbo decoder.

Architecture	UXMAP [57]	UXMAP [66]	FPMAP [57]	FPMAP [65]
K	128	128	128	6144
Radix/P	4/1	4/4	2/128	2/6144
Frequency (MHz)	800	800	500	252
Throughput (Gb/s)	102.4	102.4	1.6	39.86
Area (mm^2)	23.61	16.52	1.04	24.09
Area Eff. ($Gb/s/mm^2$)	4.34	6.20	1.53	1.65

Table 3.3: Comparison of the implementation results of the UXMAP and FPMAP architectures in 28 nm CMOS technology.

Chapter 4

The Local-SOVA

This chapter is dedicated to the description of a completely new algorithm that I developed in my thesis, which I called the local Soft-Output Viterbi algorithm (local-SOVA). The algorithm, the corresponding hardware architecture, and its application to high-radix SISO decoding are presented thoroughly.

The rest of this chapter is organized as follows. Section 4.1 gives an overview of high-radix schemes and provides the motivation for the introduction of the local-SOVA. Section 4.2 recalls the MLM algorithm and analyzes it from the SOVA perspective. Section 4.3 describes the local-SOVA and a corresponding hardware architecture. It is also shown that the MLM algorithm can be seen as an instance of the proposed local-SOVA. High-radix turbo decoders using local SOVA are then considered with new simplifications in Section 4.4 and simulation results and a computational complexity analysis are provided to illustrate the advantages of the local-SOVA. Finally, Section 4.5 concludes the chapter.

4.1 Introduction

In this chapter, we mainly focus on how to enable high-order radix decoding schemes for turbo decoders. Previous works in the literature, such as [68, 69], mainly tried to increase the throughput of BCJR-based SISO decoders, without specifically considering the complexity reduction of the studied algorithm. Only in [70], a low-complexity radix-16 SISO decoder for the MLM algorithm was proposed, with the introduction of specific processing to limit the resulting error correction degradation at high signal-to-noise ratios.

On the other hand, the SOVA [37] has been recently reconsidered as an efficient SISO candidate for turbo decoding [71, 72]. However, the interest in this algorithm remains limited. This is because, first of all, the MLM algorithm outperforms the SOVA by about 0.75 dB when used for turbo decoding [41] and, second, the serial nature of the SOVA is even more pronounced when compared to the MLM algorithm due to the involved traceback and the soft output update procedures. Nevertheless, the SOVA provides an alternative way to perform the soft output computation for SISO decoding. In [73], the authors proposed the *modified SOVA* that combines two update rules, the Hagenauer rule [37] and the Battail rule [74, 75]. This modified algorithm performs better than the classical SOVA [37] where only the Hagenauer rule is employed. Nonetheless, the decoder in [73] is a SOVA-like decoder involving the soft output update procedure, which is a serial process in nature. The modified SOVA is able to achieve the same performance as the MLM algorithm but the decoder has then to perform the soft output update for each bit in each path every time it calculates the metric difference in the trellis. Consequently, it exhibits higher complexity and memory consumption than the classical SOVA.

In this section, we propose the *local-SOVA*, a combination between the MLM algorithm and the two above-mentioned update rules, the Hagenauer rule (HR) and the Battail rule (BR). More specifically, the proposed algorithm first performs the forward and backward state metric recursions as in the MLM algorithm. Then, for each trellis section, while the MLM generates the soft output as the difference between the maximum cumulated metric corresponding to the value 1 of the considered bit and the maximum cumulated metric corresponding to the value 0 of the considered bit, the proposed local-SOVA applies a different computational process using both update rules. Note that both methods result into the same soft output value. In fact, we will show that the MLM generation of soft output appears to be a particular case of the local-SOVA method. Moreover, the proposed algorithm does not involve either the traceback nor the soft output update of the classical SOVA and it applies computation steps closer to the MLM algorithm except for the soft output calculation. When applied to high-radix orders, the algorithm reveals to be quite different from the MLM and SOVA algorithms. Indeed, it performs high-radix decoding of convolutional codes in an efficient way, simplifying the high-throughput implementations of turbo decoders.

4.2 The Max-Log-MAP algorithm from the SOVA viewpoint

Referring to Section 2.3.3.3, the Max-Log-MAP (MLM) algorithm consists of the following steps: branch metric calculation, forward recursion, backward recursion and soft-output computation. For an RSC code with coding rate $r = 1/2$, these processes are expressed

as follows:

$$\Gamma_k(s,s') = c_k^s(s,s') \times \left(\mathrm{L}_k^a + \mathrm{L}_k^s\right) + c_k^p(s,s') \times \mathrm{L}_k^p, \tag{4.1}$$

$$A_{k+1}(s') = \max_s \left(A_k(s) + \Gamma_k(s,s')\right), \tag{4.2}$$

$$B_k(s) = \max_{s'} \left(B_{k+1}(s') + \Gamma_k(s,s')\right), \tag{4.3}$$

$$\begin{aligned}\mathrm{L}_{\mathrm{MLM}}(u_k) = &\max_{(s,s')|u_k=1} \left(A_k(s) + \Gamma_k(s',s) + B_{k+1}(s)\right) \\ &- \max_{(s,s')|u_k=0} \left(A_k(s') + \Gamma_k(s',s) + B_{k+1}(s)\right),\end{aligned} \tag{4.4}$$

where

- $c_k^s(s,s')$, $c_k^p(s,s')$ are the systematic and the parity symbols labeling the trellis branch (s,s');
- L_k^s, L_k^a, L_k^p are the systematic, the *a priori* and the parity LLRs for trellis section k;
- $\Gamma_k(s,s')$ is the metric of branch (s,s') for trellis section k;
- $A_k(s)$ is the forward state metric of state s at instant k, $B_{k+1}(s')$ is the backward state metric of state s' at instant $k+1$;
- $\mathrm{L}_{\mathrm{MLM}}(u_k)$ is the soft-output calculated by the MLM algorithm for the decoded bit u_k.

Let us take as an example the estimation of the soft output related to u_k at trellis section k in Fig. 4.1, using the MLM algorithm. The forward and backward propagations provide all the values for $A_k(s)$, $s = 0,\ldots,3$ and $B_{k+1}(s')$, $s' = 0,\ldots,3$. Then, assuming that $(s,s') = (0,0)$ is the most likely trellis branch for $u_k = 0$ (bold, dashed line) and that $(s,s') = (2,1)$ is the most likely trellis branch for $u_k = 1$ (bold, solid line), $L_{\mathrm{MLM}}(u_k)$ can be written as:

$$L_{\mathrm{MLM}}(u_k) = (A_k(2) + \Gamma_k(2,1) + B_{k+1}(1)) - (A_k(0,0) + \Gamma_k(0,0) + B_{k+1}(0)) \tag{4.5}$$

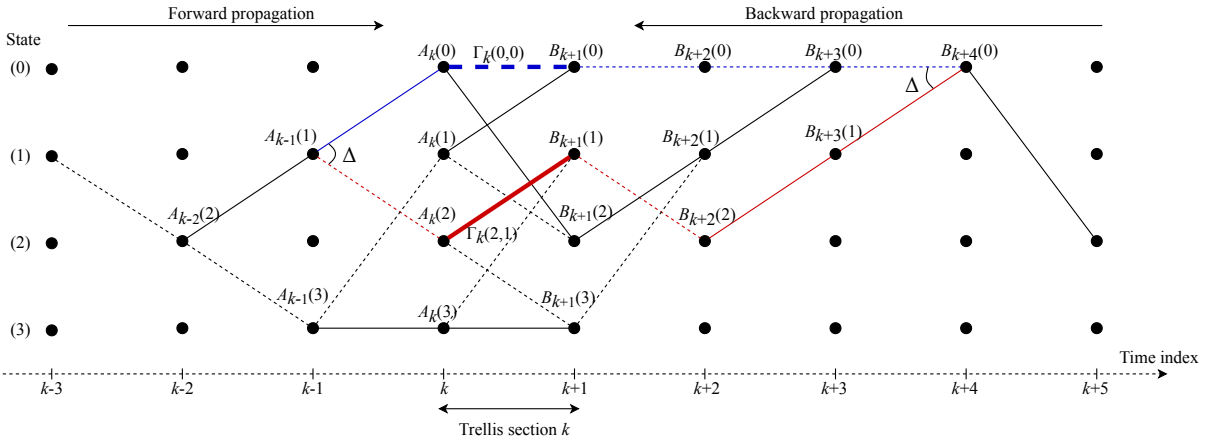


Figure 4.1: Trellis representation of a convolutional code with $\nu = 2$. Dashed branches correspond to data bits equal to 0 and solid branches to data bits equal to 1.

On the other hand, the LLRs provided by the MLM algorithm can also be seen in a different way, involving *paths* in the trellis diagram. From section 2.2.2, a path is a state sequence corresponding to a particular input bit sequence in the trellis. Furthermore, performing the forward state metrics recursion in the MLM from instant 0 to instant k is equivalent to performing the path propagation process of the VA (see Section 2.3.1.1) from instant 0 to k. As a result, at instant k, there are 2^ν values of A_k that can be considered as 2^ν surviving paths. Similarly, the backward state metric recursion can also be considered as a path propagation process but in the backward direction and one can identify another set of 2^ν surviving paths after calculating B_{k+1}.

Furthermore, thanks to the path convergence property of convolutional codes [35], the surviving paths can be truncated to some manageable length $\mathcal{L}$ to limit the amount of memory necessary for their storage, without any noticeable impact on the error correction performance. For a truncation length of $\mathcal{L}$ trellis sections, the VA needs to process the trellis paths until instant k to take the decision on the information bit at instant $(k-\mathcal{L})$. The value of $\mathcal{L}$ should be chosen large enough so that the 2^ν surviving paths from the forward propagation originate from a single state at instant $(k-\mathcal{L})$ with sufficiently high probability [36]. The same rule applies if a backward propagation is carried out: in this case, the VA needs to process the trellis paths until instant k' to take the decision on the information bit at instant $(k'+\mathcal{L})$.

Now, let us consider that both forward and backward propagations are carried out through the trellis. Then, for trellis section k, there are 2^ν surviving paths at instant k resulting from the forward propagation and 2^ν surviving paths at instant $k+1$ resulting from the backward propagation. Connecting these two sets of paths with the $2^{\nu+1}$ trellis branches yields a total number of $2^{\nu+1}$ surviving paths for trellis section k. The path going through state s at instant k and through state s' at instant $k+1$ has its path metric equal to $A_k(s)+\Gamma_k(s,s')+B_{k+1}(s')$. Moreover, because of the path convergence property, all $2^{\nu+1}$ paths originate from a single state at instant k' such that $\max(0,k-\mathcal{L}) \leq k' < k$ in the forward propagation and from a single state at instant k'' such that $k+1 < k'' \leq \min(k+\mathcal{L}+1, K)$ in the backward propagation. So, these $2^{\nu+1}$ paths merge together at times indexes k' and k''.

Now, with the concept of path merging exploited in the SOVA [37], for each pair of merging paths, we can obtain the metric difference between these two paths by subtracting the lower path metric from the higher one. Then, we can manipulate these resulting metric differences between paths to get the soft estimate of bit u_k for trellis section k.

Taking the example in Fig. 4.1, the forward propagation and the backward propagation provide us with four forward surviving paths at instant k and four backward surviving paths at instant $k+1$, respectively. So, eight paths have to be considered for trellis section k, and as shown in Fig. 4.1, these paths merge into state 2 at instant $k' = k-2$ and into state 0 at instant $k'' = k+4$. Furthermore, if we denote by P_0 the path going through the branch (0,0) and by P_1 the path going through the branch (2,1) in trellis section k, the LLR of bit u_k can be estimated as the metric difference Δ between these two paths where the path metric M_0 of P_0 is $M_0 = (A_k(0)+\Gamma_k(0,0)+B_{k+1}(0))$ and the path metric M_1 of P_1 is $M_1 = (A_k(2)+\Gamma_k(2,1)+B_{k+1}(1))$. The result is then equal to the LLR estimated by the MLM algorithm in (4.5).

Based on this equivalence, we can reformulate the MLM algorithm. In the trellis diagram, each trellis section k involves $2^{\nu+1}$ paths going through $2^{\nu+1}$ branches and merging together. If a path goes through the branch (s,s'), its path metric M_k is then expressed

as

$$M_k(s, s') = A_k(s) + \Gamma_k(s, s') + B_{k+1}(s'). \tag{4.6}$$

The soft output related to bit u_k is then equal to the metric difference between the path with the largest path metric carrying $u_k = 1$ and the path with the largest path metric carrying $u_k = 0$. In the next section, we define the path merge operation, present its properties and show how it can be used to reformulate the MLM algorithm.

4.3 The Local-SOVA

Conventionally, a path in a trellis diagram is defined as a sequence of states and is associated with an input bit sequence and a path metric. Starting from this section, we will adopt an alternative mathematical definition of a path, with a more local sense, that focuses on a particular trellis section k. In the following, all the derivations focus on trellis section k. Therefore, for sake of simplicity, we will omit k in the notations.

We define a path P as a 3-tuple consisting of a path metric denoted by M, a hard decision denoted by u and a reliability value related to u, denoted by L:

$$P = \{M, u, L\} \in \mathbb{R} \times \{0, 1\} \times \mathbb{R}^+, \tag{4.7}$$

where $\mathbb{R}$ is the set of real numbers and $\mathbb{R}^+$ is the set of positive real numbers.

As stated in the previous section, if path P goes through branch (s, s') at trellis section k, its path metric M is given by (4.6) where we omit (s, s') for the sake of simplicity. Moreover, the hard decision u is the data bit carried by the corresponding branch in the trellis diagram (0 for a dashed line or 1 for a solid line in Fig. 4.1). The reliability of the hard decision, L, is initialized to $+\infty$ or to the largest possible value achievable with the used quantization.

We further define the *merge operation*

$$\mathcal{M} \ : \ \{\mathbb{R} \times \{0, 1\} \times \mathbb{R}^+\}^2 \rightarrow \mathbb{R} \times \{0, 1\} \times \mathbb{R}^+, \tag{4.8}$$

taking two paths as arguments and producing one path as output. $P_a = \{M_a, u_a, L_a\}$ and $P_b = \{M_b, u_b, L_b\}$ being two paths, determining path P_c such that $P_c = \mathcal{M}(P_a, P_b)$ involves three procedures: finding M_c, u_c and L_c. The output path metric M_c and hard decision u_c can be obtained by comparing the path metrics M_a and M_b. Let $p = \arg\max_{a,b}(M_a, M_b)$, then $M_c = M_p$ and $u_c = u_p$. Through that mechanism, if several paths merge at a trellis stage, the resulting output path will be assigned the largest path metric and will be considered as the ML path. Then, the hard decision carried by the ML path is the decision with highest path metric, and this is also the hard decision provided by the decoder. Furthermore, in order to find L_c, we employ the two well-known reliability HR and BR update rules [37, 74, 75]. Both rules were proposed independently in the late 80's for SOVA decoders.

4.3.1 Reliability Update Rules

Let P_a and P_b be two paths to be merged and let us define p and p' as

$$p = \arg\max_{a,b}(M_a, M_b); \quad p' = \arg\min_{a,b}(M_a, M_b), \tag{4.9}$$

and the metric difference between P_a and P_b as $\Delta_{p,p'} = M_p - M_{p'}$. Note that the metric difference between two paths is always positive. Then, if $P_c = \mathcal{M}(P_a, P_b)$, L_c is calculated as follows:

1. If $u_a \neq u_b$, apply HR
$$L_c = \min\big(L_p, \Delta_{p,p'}\big). \tag{4.10}$$
2. If $u_a = u_b$, apply BR
$$L_c = \min\big(L_p, \Delta_{p,p'} + L_{p'}\big). \tag{4.11}$$

These two update rules can be summarized using the following update function ϕ:

$$\begin{aligned} L_c &= \phi\big(L_p, L_{p'}, \Delta_{p,p'}, u_p, u_{p'}\big) \\ &= \min\big(L_p, \Delta_{p,p'} + \delta(u_p, u_{p'})L_{p'}\big), \end{aligned} \tag{4.12}$$

where

$$\delta(u_p, u_{p'}) = \begin{cases} 1, & \text{if } u_p = u_{p'} \\ 0, & \text{otherwise.} \end{cases} \tag{4.13}$$

The combination of these two rules for SOVA decoding was already proposed in [73] and the authors proved the equivalence with the MLM algorithm. However, in [73], the authors only considered forward propagation. To estimate the reliability of the hardware decision at trellis section k, the algorithm carries out a forward propagation up to trellis stage $k + L$ and then performs a traceback procedure. Thus, a large number of paths should be considered, which translates into a massive use of function ϕ and also into large memory for storing the reliability values after each update.

We propose an alternative algorithm that uses both forward and backward propagations and hence limits the number of paths considered for trellis section k to $2^{\nu+1}$, thus reducing the use of function ϕ.

4.3.2 Commutative and Associative Properties of the Merge Operation

As already mentioned earlier, the merge path operation described above involves three procedures: 1) selecting the output path metric 2) selecting the related hard decision and 3) updating the related reliability value using function ϕ in (4.12). We show in this section that the merge operation has the commutative and associative properties.

Theorem 4.1. *The merge operation is commutative and associative.*

- *Commutative property: let P_a, P_b be two merging paths, then*
$$\mathcal{M}(P_a, P_b) = \mathcal{M}(P_b, P_a), \tag{4.14}$$
- *Associative property: let P_a, P_b, and P_c be three merging paths, then*
$$\mathcal{M}(\mathcal{M}(P_a, P_b), P_c) = \mathcal{M}(P_a, \mathcal{M}(P_b, P_c)). \tag{4.15}$$

The "=" operator between two paths is defined as the equality between all the elements in their tuples.

Proof. We will prove the commutative and associative properties of the three above-mentioned procedures of the merge operation.

For the commutative property, let us define p and p' as in (4.9). The path metric of the output path is then $M_{\mathcal{M}(P_a,P_b)} = M_p$, the hard decision is $u_{\mathcal{M}(P_a,P_b)} = u_p$, and according to (4.12), the reliability value is then updated as

$$L_{\mathcal{M}(P_a,P_b)} = \min(L_p, \delta(u_p, u_{p'})L_{p'} + \Delta_{p,p'}). \tag{4.16}$$

Since p and p' do not depend on the order of P_a and P_b, the merge operation is commutative.

For the associative property, we can easily show that selecting a path metric and providing a hard decision are associative procedures because they get the values of the path with the largest metric and the maximum function is associative since

$$\max(M_a, \max(M_b, M_c)) = \max(\max(M_a, M_b), M_c).$$

Concerning the reliability update procedure, without loss of generality, we assume that M_c is the largest path metric and we define p and p' as in (4.9). The updated reliability values of $\mathcal{M}(P_a, \mathcal{M}(P_b, P_c))$ and $\mathcal{M}(\mathcal{M}(P_a, P_b), P_c)$ in (4.15) are respectively derived as

$$\begin{aligned} L_{\mathcal{M}(P_a,\mathcal{M}(P_b,P_c))} &= \min\Big(\min\big(L_c, \delta(u_c, u_b)L_b + \Delta_{c,b}\big), \delta(u_c, u_a)L_a + \Delta_{c,a}\Big) \\ &= \min\big(L_c, \delta(u_c, u_b)L_b + \Delta_{c,b}, \delta(u_c, u_a)L_a + \Delta_{c,a}\big), \end{aligned} \tag{4.17}$$

$$\begin{aligned} L_{\mathcal{M}(\mathcal{M}(P_a,P_b),P_c)} &= \min\Big(L_c, \delta(u_c, u_p)\min\big(L_p, \delta(u_p, u_{p'})L_{p'} + \Delta_{p,p'}\big) + \Delta_{c,p}\Big) \\ &= \min(L_c, \delta(u_c, u_p)L_p + \Delta_{c,p}, \delta(u_c, u_p)\delta(u_p, u_{p'})L_{p'} + \delta(u_c, u_p)\Delta_{p,p'} + \Delta_{c,p}). \end{aligned} \tag{4.18}$$

Note that both $\delta(u_c, u_p)$ and $\delta(u_p, u_{p'})$ can take value in $\{0, 1\}$ as defined by (4.13), the proof is then divided into four cases:

1. $\delta(u_c, u_p) = 1$ and $\delta(u_p, u_{p'}) = 1$, then (4.18) becomes

   $$\min(L_c, L_p + \Delta_{c,p}, L_{p'} + \Delta_{p,p'} + \Delta_{c,p}) = \min(L_c, L_p + \Delta_{c,p}, L_{p'} + \Delta_{c,p'}), \tag{4.19}$$

   where $\Delta_{c,p} + \Delta_{p,p'} = \Delta_{c,p'}$. Moreover, (4.17) becomes

   $$\min(L_c, L_b + \Delta_{c,b}, L_a + \Delta_{c,a}). \tag{4.20}$$

   We can see that (4.19) is equivalent to (4.20).

2. $\delta(u_c, u_p) = 1$ and $\delta(u_p, u_{p'}) = 0$, then (4.18) becomes

   $$\min(L_c, L_p + \Delta_{c,p}, \Delta_{c,p} + \Delta_{p,p'}) = \min(L_c, L_p + \Delta_{c,p}, \Delta_{c,p'}). \tag{4.21}$$

   Without loss of generality, assuming that $p = b$ and $p' = a$, then $\delta(u_c, u_b) = 1$ and $\delta(u_c, u_a) = 0$, and (4.17) becomes

   $$\min(L_c, L_p + \Delta_{c,p}, \Delta_{c,p'}), \tag{4.22}$$

   which is the same as (4.21).

3. $\delta(u_c, u_p) = 0$ and $\delta(u_p, u_{p'}) = 1$, then (4.18) becomes

$$\min(L_c, \Delta_{c,p}). \tag{4.23}$$

Without loss of generality, assuming that $p = b$ and $p' = a$, then $\delta(u_c, u_b) = 0$ and $\delta(u_c, u_a) = 0$, and (4.17) is expressed as

$$\min(L_c, \Delta_{c,p}, \Delta_{c,p'}) = \min(L_c, \Delta_{c,p}, \Delta_{c,p} + \Delta_{p,p'}). \tag{4.24}$$

Note that $\Delta_{c,p} + \Delta_{p,p'} \geq \Delta_{c,p}$, hence, (4.24) is equivalent to (4.23).

4. $\delta(u_c, u_p) = 0$ and $\delta(u_p, u_{p'}) = 0$, then (4.18) becomes

$$\min(L_c, \Delta_{c,p}). \tag{4.25}$$

Again, without loss of generality, assuming that $p = b$ and $p' = a$, then $\delta(u_c, u_b) = 0$ and $\delta(u_c, u_a) = 1$, and (4.17) is expressed as

$$\min(L_c, \Delta_{c,p}, L_{p'} + \Delta_{c,p'}) = \min(L_c, \Delta_{c,p}, L_{p'} + \Delta_{c,p} + \Delta_{p,p'}). \tag{4.26}$$

Also note that $L_{p'} + \Delta_{c,p} + \Delta_{p,p'} \geq \Delta_{c,p}$, therefore, (4.26) is equivalent to (4.25).

Therefore, the associative property is proved for all four cases.

□

Remark. Based on the commutative and associative properties of the merge operation, two important statements can be inferred:

- We can extend the merge operation to more than two paths. For instance, for four paths P_a, P_b, P_c and P_d, we can write $\mathcal{M}(P_a, P_b, P_c, P_d)$ to refer to the output path obtained by merging the four paths.
- The merge operation can be processed in a *dichotomous* fashion:

$$\mathcal{M}(P_a, P_b, P_c, P_d) = \mathcal{M}\big(\mathcal{M}(P_a, P_b), \mathcal{M}(P_d, P_c)\big),$$

where $\mathcal{M}(P_a, P_b)$ and $\mathcal{M}(P_c, P_d)$ can be processed in parallel and then the resulting paths are merged to yield the output path.

4.3.3 The MLM Algorithm as an Instance of the Merge Operation

Referring back to the analyze in Section 4.2, let us consider the $2^{\nu+1}$ paths going through trellis section k and merging together. Among them, $n = 2^\nu$ paths, denoted by $\{P_{p_1^0}, \ldots, P_{p_n^0}\}$, carry hard decision $u = 0$ at trellis section k and the remaining n, denoted by $\{P_{p_1^1}, \ldots, P_{p_n^1}\}$, carry hard decision $u = 1$. The reliability value related to bit u provided by the MLM algorithm is

$$L_{\text{MLM}}(u) = \max_{i=1,\ldots,n} \{M_{p_i^1}\} - \max_{j=1,\ldots,n} \{M_{p_j^0}\}. \tag{4.27}$$

Now, we can take a look at the operation merging all the paths with hard decision $u = 1$: $\mathcal{M}(P_{p_1^1}, \ldots, P_{p_n^1})$. The resulting output hard decision is obviously 1 and the output path metric is $M_1 = \max_{i=1,\ldots,n}\{M_{p_i^1}\}$. For the output reliability, merging paths with the

same hard decision value requires the application of BR (4.11). Since the reliability values are all initialized at $+\infty$, applying (4.11) yields an output reliability also equal to $+\infty$. Similarly, merging together all the paths carrying hard decision $u = 0$ applying $\mathcal{M}(P_{p_1^0}, \ldots, P_{p_n^0})$ results in an output hard decision 0, an output path metric equal to $M_0 = \max_{i=1,\ldots,n}\{M_{p_i^0}\}$ and an output reliability at $+\infty$.

Then, if we merge the two resulting paths:

$$\mathcal{M}\big(\mathcal{M}(P_{p_1^1}, \ldots, P_{p_n^1}), \mathcal{M}(P_{p_1^0}, \ldots, P_{p_n^0})\big), \tag{4.28}$$

the computation of the output reliability amounts to the application of HR (4.10):

$$L_{\mathcal{M}} = \min\Big(+\infty, \big|M_1 - M_0\big|\Big) = \big|M_1 - M_0\big|, \tag{4.29}$$

which is the absolute value of the expression of L_{MLM} in (4.27). If we denote by $u_{\mathcal{M}}$ the output hard decision deriving from (4.28), then L_{MLM} is equal to

$$L_{\mathrm{MLM}}(u_k) = (2u_{\mathcal{M}} - 1) \times L_{\mathcal{M}}. \tag{4.30}$$

Therefore, the result of the MLM algorithm can be interpreted as the outcome of a merge operation applied to all the paths.

On another note, thanks to the commutative and associative properties of the merge operation stated in (4.14) and (4.15), the operation merging all the paths (4.28) can be performed in a different order for a better match with efficient software or hardware implementations. In particular, instead of first merging the paths with the same hard decision, one can start by merging pairs of paths with different hard decisions $(P_{p_i^1}, P_{p_i^0})$, $i = 1, \ldots, n$ as

$$\mathcal{M}\big(\mathcal{M}(P_{p_1^1}, P_{p_1^0}), \ldots, \mathcal{M}(P_{p_n^1}, P_{p_n^0})\big). \tag{4.31}$$

Moreover, if $P_{p_i^1}$ and $P_{p_i^0}$ are chosen in such a way that the corresponding trellis branches at trellis section k, $(s_{p_i^1}, s'_{p_i^1})$ and $(s_{p_i^0}, s'_{p_i^0})$ verify $s'_{p_i^1} = s'_{p_i^0} = s'$, the merge operation of the pair of paths yields the following path metric

$$\begin{aligned}
M_{p_i} &= \max(M_{p_i^1}, M_{p_i^0}) \\
&= \max\big(A_k(s_{p_i^1}) + \Gamma_k(s_{p_i^1}, s') + B_{k+1}(s'), A_k(s_{p_i^0}) + \Gamma_k(s_{p_i^0}, s') + B_{k+1}(s')\big) \\
&= \max\big(A_k(s_{p_i^1}) + \Gamma_k(s_{p_i^1}, s'), A_k(s_{p_i^0}) + \Gamma_k(s_{p_i^0}, s')\big) + B_{k+1}(s') \qquad (4.32) \\
&= A_{k+1}(s') + B_{k+1}(s'), \qquad (4.33)
\end{aligned}$$

and since $u_{p_i^1} \neq u_{p_i^0}$, the updated reliability with HR is

$$L_{p_i} = \min\big(+\infty, \Delta_{p_i^1, p_i^0}\big) = \Delta_{p_i^1, p_i^0} \tag{4.34}$$

where

$$\Delta_{p_i^1, p_i^0} = \Big|M_{p_i^1} - M_{p_i^0}\Big| = \Big|\big(A_k(s_{p_i^1}) + \Gamma_k(s_{p_i^1}, s')\big) - \big(A_k(s_{p_i^0}) + \Gamma_k(s_{p_i^0}, s')\big)\Big|. \tag{4.35}$$

The output hard decision u_{p_i} is provided by the path, $P_{p_i^0}$ or $P_{p_i^1}$, with the higher path metric.

We can see from (4.32) and (4.33) that, with the proposed merge ordering, the calculation of the output path metric for this scheduling proposal $\mathcal{M}(P_{p_i^1}, P_{p_i^0})$ includes the

derivation of the forward state metric $A_{k+1}(s')$ as in the forward recursion (4.2). Therefore, there is no need to perform a preliminary calculation of the forward state metrics. Only the backward state metrics need to be computed in advance. Similarly, one can show that if the paths $P_{p_i^1}$ and $P_{p_i^0}$ are chosen in such a way that the corresponding trellis branches at trellis section k are stemming from the same state ($s_{p_i^1} = s_{p_i^0} = s$), the calculation of the output path metric includes the same derivation of the backward state metric $B_k(s)$ as in the backward recursion (4.3). Then, there is no need to perform a preliminary calculation of the backward state metrics and only the forward state metrics need to be computed in advance.

For a convolutional code with memory length ν, the overall merge operation for the computation of the soft output for the decoder can be carried out in a dichotomous fashion: the merge operation then requires a binary tree of $2^{\nu+1} - 1$ elementary merge operators organized in $\nu + 1$ layers. Taking Fig. 4.1 as an example, $\nu = 2$ and the soft output related to bit u_k is obtained by a binary tree consisting of $\nu + 1 = 3$ layers of merge operators, as shown in Fig. 4.2.

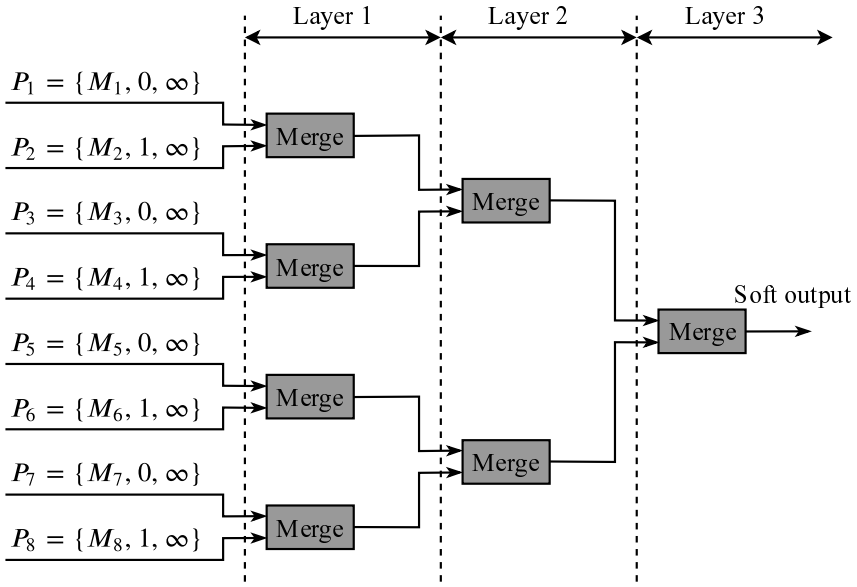


Figure 4.2: Binary tree architecture used for the soft output computation in the local-SOVA decoder for the code of Fig.4.1.

The next section describes the overall algorithm implementing this particular arrangement for the merge operations.

4.3.4 The Soft Output Computation Algorithm

The soft output calculation using the merge operations in a dichotomous fashion can be performed according to Algorithm 1. Note that this algorithm is a generic version that assumes that all the forward and backward state metrics are pre-computed and stored in a memory.

Specific simplifications can be made for the local-SOVA. First, at the initialization step, we can organize the set of transitions $\mathbf{T}$ so that two adjacent transitions (s_{2i-1}, s'_{2i-1}) and (s_{2i}, s'_{2i}) have the same state on the left, i.e. $s_{2i-1} = s_{2i}$, for $i = 1, \ldots, 2^{\nu}$. Then, in line 4 of the Algorithm 1, we can initialize path metrics $M_p(0)$ to be $\Gamma_k(s, s') + B_{k+1}(s')$ instead

Algorithm 1 The generic local-SOVA

Assumption: The values of $\Gamma_k(s, s')$, $A_k(s)$, $B_{k+1}(s')$ are available for all $k = 0, \ldots, K-1$, and for all pair of (s, s').
Initialization: $\mathbf{T} = [T(1) \ \ldots \ T(2^{\nu+1})]$ is the set of $2^{\nu+1}$ transitions of the trellis section defined by the pairs of states (s, s');

1: **for** each trellis section $k = 0, \ldots, K-1$ **do**
2: **for** each path $p = 1, \ldots, 2^{\nu+1}$ **do**
3: $(s_p, s'_p) = T(p)$
4: $M_p(0) = A_k(s_p) + \Gamma_k(s_p, s'_p) + B_{k+1}(s'_p)$;
5: $u_p(0) =$ data bit on transition (s_p, s'_p);
6: $L_p(0) = +\infty$;
7: **end for**

Initialize path P_p, $p = 1, \ldots, 8$, for layer $l = 0$

8: **for** each layer $l = 1, \ldots, \nu+1$ **do**
9: **for** each path $p = 1, \ldots, 2^{(\nu-l+1)}$ in layer l **do**
10: $a = \arg\max_{j \in \{2p-1, 2p\}} \{M_j(l-1)\}$;
11: $b = \arg\min_{j \in \{2p-1, 2p\}} \{M_j(l-1)\}$;
12: $\Delta_{a,b} = M_a(l-1) - M_b(l-1)$;
13: $M_p(l) = M_a(l-1)$;
14: $u_p(l) = u_a(l-1)$;
15: $L_p(l) = \phi\big(L_a(l-1), L_b(l-1), \Delta_{a,b}, u_a(l-1), u_b(l-1)\big)$;
16: **end for**
17: **end for**

Process the $(\nu+1)$-layer binary tree

18: $\hat{u}_k = u_1(\nu+1)$; ▷ Hard output
19: $\hat{L}(u_k) = (2\hat{u}_k - 1) \times L_1(\nu+1)$; ▷ Soft output
20: **end for**

of $A_k(s) + \Gamma_k(s, s') + B_{k+1}(s, s')$. The resulting path metric $M_a(1)$ at layer $l = 1$ is equal to $B_k(s)$, thus allowing the backward recursion to be incorporated into the soft output computation. To compensate for omitting $A_k(s)$ in the expression of $M_p(0)$, the output path metric $M_p(1)$ in line 13 should be taken equal to

$$M_p(1) = M_a(1) + A_k(s). \tag{4.36}$$

Furthermore, in line 15 of Algorithm 1, the output reliability $L_p(1)$ can be directly assigned $\Delta_{a,b}$ and the calculation of the reliability value at the first layer can be replaced by a simple assignment operation, making the initial assignment to infinity in line 6 unnecessary. After layer $l = 1$, the subsequent layers should be carried out following Algorithm 1 without any modification.

As mentioned in Chapter 3, for a practical application of SISO decoding algorithms, a scheduling (FB or Butterfly) has to be chosen and a sliding window technique can be applied to limit the memory consumption. To achieve high throughputs, parallel architectures have to be implemented. Therefore, like the MLM algorithm, the local-SOVA should be compatible with these techniques and architectures. In fact, this is not a problem since the only difference between the proposed local-SOVA and the MLM algorithm

is the way the soft output is calculated. The local-SOVA performs the branch metric calculation and the state metric recursion in the same way as the MLM algorithm does. Thus, not only these techniques are applicable to the local-SOVA, but the memory for the state metrics are the same for both decoding algorithms. As an illustration, a local-SOVA compatible with the FB scheduling is described in Algorithm 2. In the FB scheduling, the forward state metrics are first recursively calculated and stored in the memory. Then, the algorithm recursively calculates the backward state metrics and derives the soft output. The application of sliding window and other technique is straightforward.

To complete this algorithmic description, the next sections provide some details about possible hardware architectures for a local-SOVA decoder. To this end, we first focus on a radix-2 trellis and highlight the differences with a conventional MLM decoder.

4.3.5 Radix-2 Local-SOVA Decoder Architecture

Since the proposed algorithm only differs from the MLM algorithm in the soft output calculation, its global architecture is composed of the same blocks as the architecture of a MLM decoder. In case of FB scheduling, it consists of the BMU, the forward ACSU, the backward ACSU, and the SOU. Fig. 4.3 shows the corresponding basic architecture for the local-SOVA with FB scheduling. Note that the forward recursion is not shown in the figure since it is the same as for the MLM algorithm. Furthermore, we assume the forward state metrics have already been recursively calculated and stored in a state metric memory. If a symmetric forward-backward scheduling is applied, the roles of forward and backward units are just swapped.

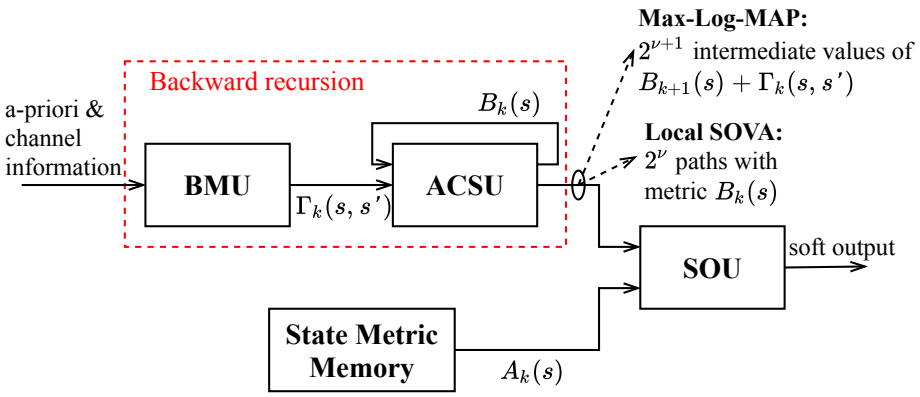


Figure 4.3: Basic architecture considered for the local-SOVA using the FB scheduling.

At trellis section k, the BMU calculates all possible values of $\Gamma_k(s, s')$ and forwards them to the ACSU. For the local-SOVA architecture, the first layer of Algorithm 2, depicted in Fig. 4.4, performs the backward state metric recursion as well as it produces the hard decision, the reliability value and the path metric for subsequent layers.

Besides the right-hand side adder, the structure shown in Fig. 4.4 for $l = 1$ is very close to the ACSU structure of a conventional MLM decoder. Therefore, in order to make the local-SOVA easy to compare with the MLM algorithm, we consider this sub-structure as the ACSU of the local-SOVA decoder and the final adder plus the units processing the subsequent layers as its SOU.

As already mentioned, the main difference between the local-SOVA and the MLM algorithm comes from the SOU. In the MLM architecture, $2^{\nu+1}$ intermediate values

Algorithm 2 The local-SOVA compatible with FB scheduling

Assumption: The values of $A_k(s)$, are available for all $k = 0, \ldots, K-1$ from the forward recursion;
Initialization: $\mathbf{T} = [T(1) \ \ldots \ T(2^{\nu+1})]$ is the set of $2^{\nu+1}$ transitions of the trellis section defined by the pairs of states (s, s') so that two adjacent transitions (s_{2i-1}, s'_{2i-1}) and (s_{2i}, s'_{2i}) have the same state on the left, i.e. $s_{2i-1} = s_{2i}$, for $i = 1, \ldots, 2^{\nu}$;

Initialize $B_K(s)$;
for each trellis section $k = K-1, \ldots, 0$ **do**
  Calculate $\Gamma_k(s, s')$ according to (4.1);
  for each path $p = 1, \ldots, 2^{\nu+1}$ **do**
    $(s_p, s'_p) = T(p)$;
    $M_p(0) = \Gamma_k(s_p, s'_p) + B_{k+1}(s'_p)$;
    $u_p(0) =$ data bit on transition (s_p, s'_p);
    if $(p \mod 2 = 0)$ **then**
      $a = \arg\max_{j \in \{p-1, p\}} \{M_j(0)\}$;
      $b = \arg\min_{j \in \{p-1, p\}} \{M_j(0)\}$;
      $q = p/2$;
      $M_q(1) = M_a(0) + A_k(s_q)$
      $u_q(1) = u_a(0)$;
      $L_q(1) = M_a(0) - M_b(0)$;
      $B_k(s_q) = M_a(0)$;
    end if
  end for

Initialize paths, process layer $l = 1$, recursion of $B_k(s)$

  for each layer $l = 2, \ldots, \nu+1$ **do**
    for each path $p = 1, \ldots, 2^{(\nu-l+1)}$ in layer l **do**
      $a = \arg\max_{j \in \{2p-1, 2p\}} \{M_j(l-1)\}$;
      $b = \arg\min_{j \in \{2p-1, 2p\}} \{M_j(l-1)\}$;
      $\Delta_{a,b} = M_a(l-1) - M_b(l-1)$;
      $M_p(l) = M_a(l-1)$;
      $u_p(l) = u_a(l-1)$;
      $L_p(l) = \phi\big(L_a(l-1), L_b(l-1), \Delta_{a,b}, u_a(l-1), u_b(l-1)\big)$;
    end for
  end for

Process the next ν layers

  $\hat{u}_k = u_1(\nu+1)$; ▷ Hard output
  $\hat{L}(u_k) = (2\hat{u}_k - 1) \times L_1(\nu+1)$; ▷ Soft output
end for

$B_{k+1}(s') + \Gamma_k(s, s')$ computed in the ACSU are added to $A_k(s)$. Then, the most reliable branch for bit $u_k = 1$ and for bit $u_k = 0$ are selected using two 2^{ν}-input maximum selection operators and the LLR is obtained by computing the difference between the terms $A_k(s) + \Gamma_k(s, s') + B_{k+1}(s')$ for these two branches as shown in Fig. 4.5a for $\nu = 2$. On the other hand, the local-SOVA SOU takes the 2^{ν} values of $B_k(s)$, adds them to 2^{ν} corresponding values of $A_k(s)$ to provide the path metrics. The hard decisions and their

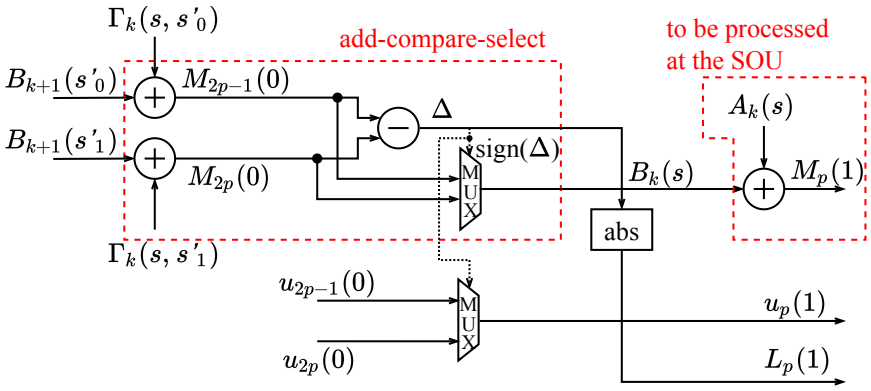


Figure 4.4: Hardware architecture of Algorithm 2 for layer $l = 1$. This is also considered as the ACSU of the local-SOVA.

reliability values computed by the ACSU are forwarded to the SOU. The SOU has then to process 2^ν paths using a binary tree of merge operators. The structure of merge operators used to process layers $l > 1$ is depicted in Fig. 4.6 and the overall structure of the tree is shown in Fig. 4.5b for $\nu = 2$ (including the adders actually part of layer $l = 1$).

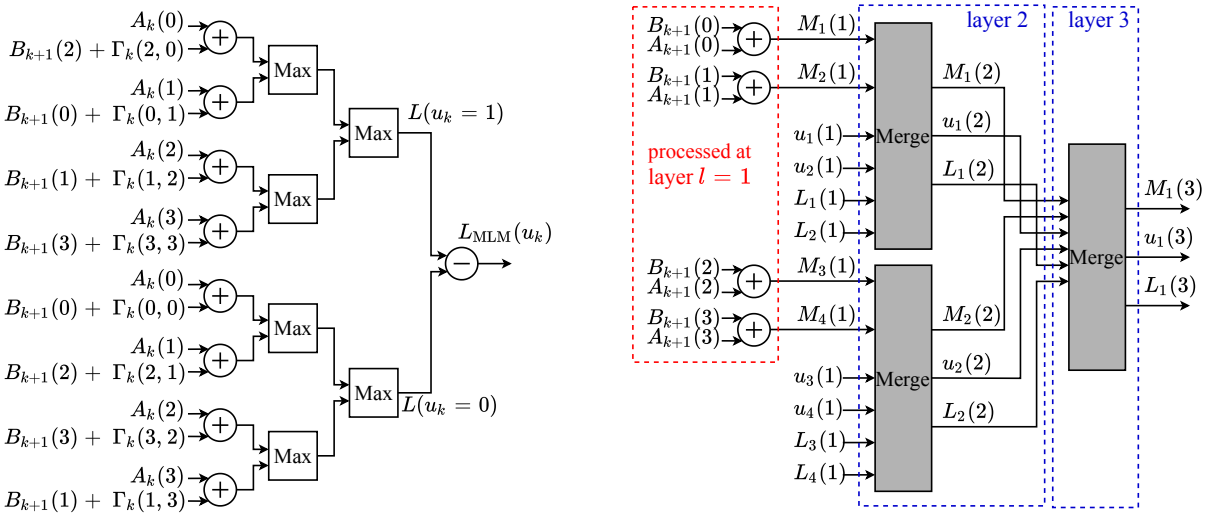


Figure 4.5: SOU architecture of (a) the MLM algorithm and (b) the local-SOVA with the merge operations.

In terms of computational complexity, with the convention that one adder is equivalent to one max or min operator and is accounted for as one computational unit and neglecting the multiplexers, the operator implementing function ϕ consists of one adder and one min operator and is therefore accounted for as two computational units. Consequently, the MLM SOU requires $(4 \times 2^\nu - 1)$ computational units while the local-SOVA SOU requires $(4 \times 2^\nu - 3)$ computational units. Since the ACSU and the BMU of both architectures are similar, both algorithms have roughly the same computational complexity when a conventional radix-2 architecture is implemented. However, the proposed algorithm is mainly of interest when higher radix orders are considered, as explained in the following

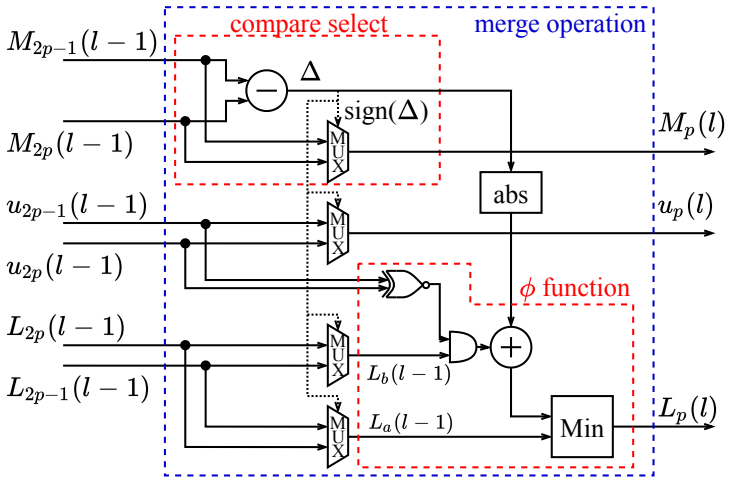


Figure 4.6: Generic hardware architecture of a merge operation $\mathcal{M}$ of two paths indexed $2p$ and $2p-1$ at layer $l > 1$.

section.

4.4 High-Radix Decoding Algorithms Using Local-SOVA

In the previous section, we have been considering the conventional radix-2 trellis diagram. In this section, we will concentrate on higher radix orders. Recall that in a radix-2^T trellis diagram, $T > 1$, there are 2^T branches coming in and out of a state s at instant k. This is obtained by aggregating T consecutive radix-2 trellis sections. Hence, a branch in a radix-2^T trellis stage is now labeled with T systematic bits and we have to reconsider the definition of a path and its corresponding merge operation.

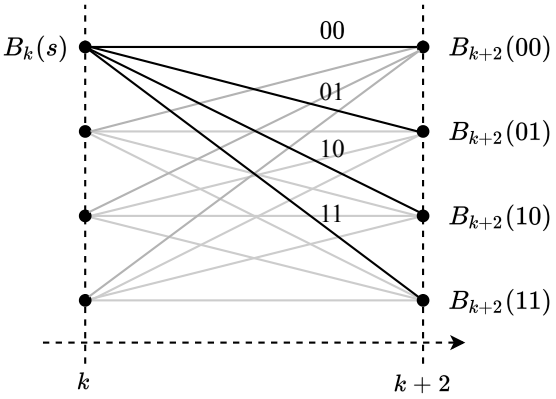


Figure 4.7: A radix-4 stage for an convolutional code with $\nu = 2$.

For a radix-2^T trellis diagram, we define a radix-2^T path P going through the branch (s, s') as

$$P = \{M, u^1, \ldots, u^T, L^1, \ldots, L^T\} \in \mathbb{R} \times \{0,1\}^T \times \{\mathbb{R}^+\}^T \tag{4.37}$$

where M is the path metric, $u^1, \ldots, u^T$ are the T hard decisions attached to branch (s, s'), and $L^1, \ldots, L^T$ are the reliability values for each hard decision and are initialized to $+\infty$.

We also define the radix-2^T merge operation $\mathcal{M}$ as in Section 4.3 with three procedures: path metric selection, hard decision selection and update of the reliability values. The selection of the path metric remains unchanged. The only difference is now that we have to select T hard decisions instead of one, and to update T reliability values using function ϕ, one for each hard decision. Note that the merge operation for high-radix paths is also commutative and associative, therefore, the order of the paths in the merge operation does not affect the output. To this end, if we arrange wisely the input paths, we can reduce complexity when compared to a straightforward implementation.

4.4.1 Radix-4 Local-SOVA Decoder with Minimum Complexity ACSU

A branch in a radix-4 trellis diagram is the aggregation of two consecutive branches in a radix-2 diagram, as illustrated in Fig. 4.7 for an convolutional code with $\nu = 2$. From instant $k+2$ to instant k, four branches are leaving and merging into each trellis state, corresponding to the transmission of two systematic bits with possible values 00, 10, 01 and 11. Therefore, at instant k, there are four radix-4 paths, denoted by $\{P_{00}, P_{01}, P_{10}, P_{11}\}$, merging into each state s. Since these four paths have the same $A_k(s)$ value, as in section 4.3.4 we can initialize the corresponding path metrics with $\Gamma_{k\to k+2}(s, s') + B_{k+2}(s')$, where $\Gamma_{k\to k+2}(s, s')$ is the sum of the two successive branch metrics at sections k and $k+1$ in the equivalent radix-2 trellis diagram. Then, we perform the radix-4 merge operation:

$$\mathcal{M}\big(P_{00}, P_{01}, P_{10}, P_{11}\big). \tag{4.38}$$

The path metric resulting from this layer-1 merge operation is also the backward state metric of state s at instant k, $B_k(s)$. Hence, we can consider the operator implementing (4.38) as the radix-4 ACSU of the local-SOVA decoder.

An important property of the radix-4 local-SOVA ACSU is that its complexity depends on the processing order of the paths in the merge operation (4.38). If we implement (4.38) as

$$\mathcal{M}\big(\mathcal{M}(P_{00}, P_{01}), \mathcal{M}(P_{10}, P_{11})\big) \tag{4.39}$$

with the hardware architecture shown in Fig. 4.8, only one ϕ operator has to be implemented. Actually, we do not need to resort to function ϕ in the first layer since the output reliability is $+\infty$ or the metric difference between the two paths, depending whether BR or HR is employed.

On the other hand, if we implement (4.38) as

$$\mathcal{M}\big(\mathcal{M}(P_{00}, P_{11}), \mathcal{M}(P_{01}, P_{10})\big), \tag{4.40}$$

we can use HR for both bits at the first layer but have to use a ϕ operation for each bit at the second layer since we do not know *a priori* which hard decision will be selected. Therefore, the implementation of (4.39) is less complex than the one of (4.40) and it has minimum complexity. Note that (4.39) is not unique, since other processing orders of the paths can also yield the same complexity, such as $\mathcal{M}\big(\mathcal{M}(P_{00}, P_{10}), \mathcal{M}(P_{01}, P_{11})\big)$.

At the output of the radix-4 ACSU, 2^ν radix-4 paths are forwarded to the radix-4 SOU. First of all, each path metric is added to the appropriate $A_k(s)$ since it has been omitted in the ACSU. Then, a ν-layer tree of radix-4 merge operators is employed to produce the

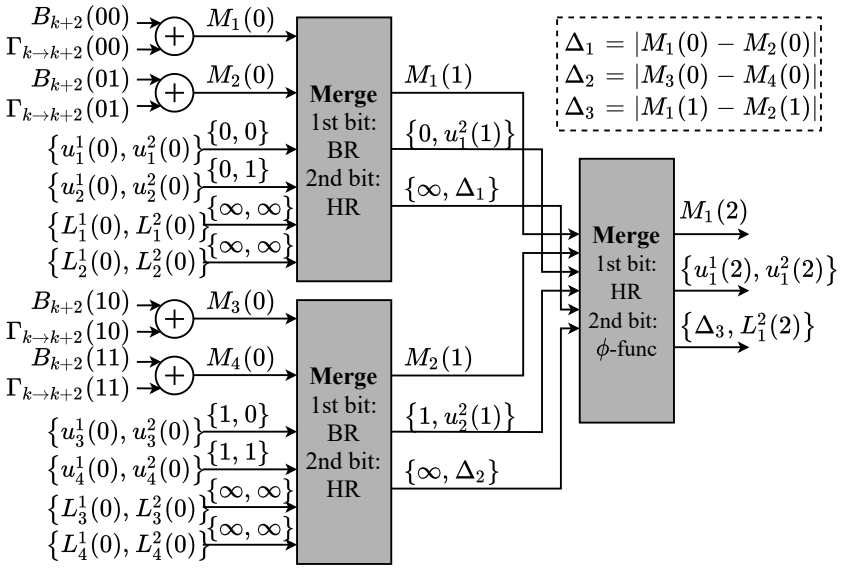


Figure 4.8: Radix-4 local-SOVA ACSU architecture implementing 2-bit merge operators according to (4.39).

final hard decision along with its reliability value. Note that the radix-4 merge operation requires two ϕ operators for updating the reliability values, one for each bit.

When radix orders higher than 4 are considered, techniques for further reducing complexity can be considered, as explained in the following section.

4.4.2 Radix-8 local-SOVA Decoder Using a Simplified Reliability Update Operator

In this section, we introduce a sub-optimal but less complex version of the function ϕ, called ω. Consequently, by combining the corresponding operator with the minimum complexity ACSU previously described in Section 4.4.1, we propose a number of local-SOVA architectures with different complexity-performance tradeoffs.

A radix-8 trellis section aggregates three consecutive radix-2 trellis sections. Each path is now composed of a path metric, three hard decisions and their three reliability values. From instant $k+3$ to instant k, eight radix-8 branches are leaving and merging into each trellis state. Therefore, at instant k, there are eight radix-8 paths merging into each state s, denoted by $\{P_{000}, P_{001}, \ldots, P_{111}\}$, where the indices represent the hard decisions associated with each path. Similarly to the previous cases, when applying the radix-8 merge operation to this set of paths

$$\mathcal{M}\big(P_{000}, P_{001}, \ldots, P_{111}\big), \tag{4.41}$$

the resulting path metric is also the backward state metric of state s at time index k, $B_k(s)$. This merging step can therefore be considered as the ACSU of the radix-8 local-SOVA decoder. Then the output path is processed by the SOU to produce the soft decision.

The minimum complexity radix-8 ACSU can be obtained by implementing (4.41) as

$$\mathcal{M}\Big(\mathcal{M}\big(\mathcal{M}(P_{000}, P_{001}), \mathcal{M}(P_{010}, P_{011})\big), \mathcal{M}\big(\mathcal{M}(P_{100}, P_{101}), \mathcal{M}(P_{110}, P_{111})\big)\Big). \tag{4.42}$$

We can see from (4.42) that the first bit in $\mathcal{M}\big(\mathcal{M}(P_{000}, P_{001}), \mathcal{M}(P_{010}, P_{011})\big)$ is always zero, hence, we can resort to a radix-4 merge operation for the last two bits using only one ϕ operator. Similarly, the first bit in $\mathcal{M}\big(\mathcal{M}(P_{100}, P_{101}), \mathcal{M}(P_{110}, P_{111})\big)$ is always one, then again, only one ϕ operator is necessary. In the second stage, the first bit is always different in the two input paths, thus only the second and the third bits require the implementation of a ϕ operator. In total, four ϕ operators have to be implemented in a minimum complexity radix-8 ACSU.

However, employing ϕ operators has two main drawbacks. First of all, they consist of one adder and one min operator, therefore prohibitively increasing the complexity of the decoder if used excessively. Second and more importantly, the adder and the min operators are connected serially. This is not desirable since the ACSU dictates the critical path of the decoder [47]. Therefore, we propose a lower complexity, lower latency, sub-optimal update operator, based on new update function, called function ω.

4.4.2.1 The Function ω

Motivated by the the possibility of using only HR as in [37], one can substitute a simplified function ω for function ϕ. Assuming that paths P_{2p-1} and P_{2p} are to be merged at layer $l-1$, the output reliability is then computed as:

$$\begin{aligned} L_p(l) &= \omega\big(L_a, \Delta_{a,b}, u_{2p-1}(l-1), u_{2p}(l-1)\big) \\ &= \begin{cases} \min(L_a, \Delta_{a,b}), & \text{if } u_{2p-1}(l-1) \neq u_{2p}(l-1) \\ L_p, & \text{if } u_{2p-1}(l-1) = u_{2p}(l-1) \end{cases} \end{aligned} \tag{4.43}$$

where a, b and $\Delta_{a,b}$ are defined at lines 10, 11 and 12, respectively, in Algorithm 1. Fig. 4.9 shows the architecture of the elementary path merging operator using function ω. The ω operator is less complex than the ϕ operator since it uses only a min operation and a multiplexer resulting in a computational complexity of about one unit. However, the price to pay is a degradation of error correction performance. Indeed, a performance degradation of 0.5 dB is observed between the conventional SOVA that uses only functions ϕ [73] and the one that uses functions ω [37].

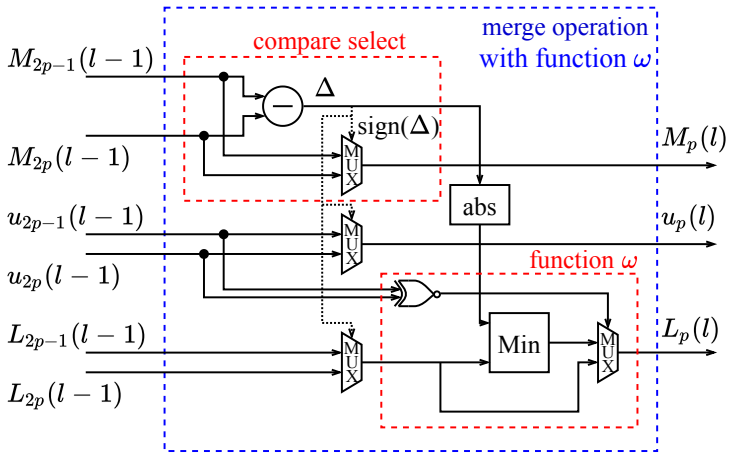


Figure 4.9: Generic hardware architecture of path merging operation using function ω.

Unlike the classical SOVA, the local-SOVA can mix both types of functions. Therefore, this provides the flexibility of several complexity/correction performance trade-offs.

However, care must be taken in making substitutions so that paths with high metrics are not eliminated from the selection process due to simplification. This is less likely to happen if the simplifications are made in the first layers of the tree, where the number of paths to be processed is high. Consequently, we observed that if we only substitute the ω operators for the ϕ operators in the first layers, we can significantly reduce complexity without degrading the performance of the decoder.

4.4.2.2 Radix-8 ACSU and SOU using ω operators

For a binary convolutional code with $\nu = 3$, the first 3 layers of the path merge binary tree are in the ACSU while the last 3 layers are processed by the SOU.

As already mentioned above, the radix-8 ACSU requires 4 ϕ operators to update the reliability values. Substituting the ω operators for the ϕ operators reduces the complexity by 4 computational units. For $\nu = 3$, the use of $2^{\nu} = 8$ ACSUs saves 32 computation units.

For the 3 layers of the binary tree implementing the radix-8 SOU, 7 radix-8 merge operations are required, resulting in the use of 21 ϕ operators (3 per merge operation). Replacing ϕ operators with ω operators reduces complexity but is expected to penalize the performance of the decoder.

4.4.3 Simulation Results

Simulations were carried out to assess the error performance of the radix-8 local-SOVA and its variants for the LTE turbo code, in order to compare them with the radix-8 MLM algorithm. For radix-2 and radix-4 local-SOVA, we proved in Section 4.3.3 that the LLR produced by the local-SOVA is the same as with the MLM algorithm. Therefore, their performance should be identical. Nevertheless, we first simulated the radix-8 local-SOVA with only ϕ operators to confirm this fact. Then, we gradually substituted ω operators for ϕ operators to observe the impact on error correction when simplifications are made to the radix-8 local-SOVA. We use the notations ACSU-(i, j) and SOU-(i, j) to represent the different configurations, where i and j are the number of layers where ω and ϕ operators are employed, respectively. For example, ACSU-(2,1) means that ω operators are implemented in the two first layers of the ACSU and that ϕ operators are used in the last layer of the ACSU. Similarly, if we use SOU-(1,2), it means that we use ω operators in the first layer of the SOU and use ϕ operators in the last two layers of the SOU.

We simulated the following seven configurations for the SISO decoding algorithms, where L-SOVA is an abbreviation for local-SOVA:

- DEC 1: MLM algorithm.
- DEC 2: L-SOVA with ACSU-(0,3) and SOU-(0,3).
- DEC 3: L-SOVA with ACSU-(3,0) and SOU-(0,3).
- DEC 4: L-SOVA with ACSU-(3,0) and SOU-(1,2).
- DEC 5: L-SOVA with ACSU-(3,0) and SOU-(2,1).
- DEC 6: L-SOVA with ACSU-(3,0) and SOU-(3,0).

- DEC 7: classical SOVA.

The simulations were carried out with information frames of $K = 1056$ bits, encoded with the non-punctured $r = 1/3$ LTE turbo code, modulated by BPSK and transmitted over the AWGN channel. A floating point representation of data is used in the decoder. The resulting bit error rate (BER) is measured after 5.5 decoding iterations. Fig. 4.10 shows that, as expected, the local-SOVA with only ϕ operators (DEC 2) has the same performance as the MLM algorithm. Moreover, by substituting ω operators for ϕ operators in the ACSUs, i. e. in the first three layers, (DEC-3) the simulated curves confirm that the error correction performance of the decoder is not degraded, thus providing a low complexity alternative to the original local-SOVA decoder. By gradually replacing the ϕ operators in the SOU, the performance is degraded by 0.05 dB at $BER = 10^{-6}$ when ω operators are used in the first two layers and by about 0.3 dB when only ω operators are implemented. Fig. 4.10 also shows that the classical SOVA (DEC 7) performs 0.1 dB worse than the local-SOVA with only ω operators (DEC 6). According to [73], using ϕ operators instead of ω operators in the both the classical and local-SOVAs would be equivalent to apply the MLM algorithm (DEC 1). However, contrary to the classical SOVA, the operations in the local-SOVA are arranged so as to minimize the number of used ϕ operators. Therefore the performance gap between DEC 6 and DEC 7 is explained by the fact that the number of sub-optimal ω operators needed in DEC 7 is greater than in DEC 6.

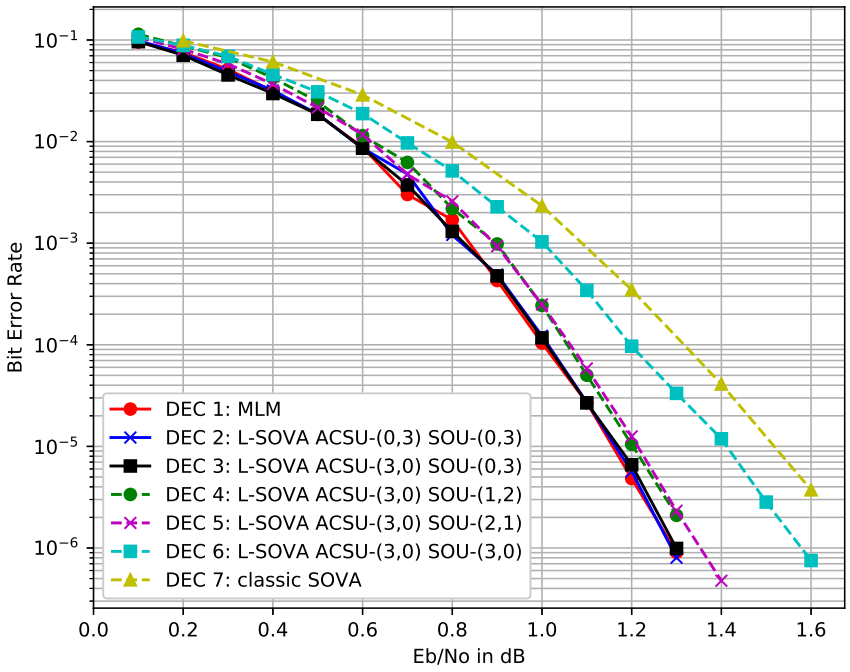


Figure 4.10: BER performance of a LTE decoder using radix-8 MLM algorithm, local-SOVA and its variant with $K = 1056$, $r = 1/3$ after 5.5 iterations. AWGN channel, BPSK modulation.

4.4.4 Computational Complexity Analysis

In this section, we perform the comparison between the local-SOVA and the MLM algorithm in terms of complexity for different radix orders. To quantify complexity, we calculate the number of additions and the number of compare select (CS) operations and consider again one adder or CS operator as one computational unit. In this comparison, we use the FB scheduling and we do not take the memory requirements into account and only consider computational complexity. As aforementioned, the difference between the local-SOVA decoder and the MLM decoder resides in the way it calculates the soft output. Therefore, the memory for the channel information, the state metrics and the extrinsic information is the same as in the MLM decoder. Moreover, sliding window techniques or different decoding schedules can also be applied to the local-SOVA, hence, the memory consumption of the decoder depends on the hardware architecture.

In this analysis, we use a convolutional code with 8 states ($\nu = 3$). The decoding complexity, denoted by $\mathcal{C}$, is reported for one trellis section consisting of eight forward ACSUs, eight backward ACSUs, and one SOU as

$$\mathcal{C} = (8 \times \mathcal{C}_F) + (8 \times \mathcal{C}_B) + \mathcal{C}_S, \tag{4.44}$$

where $\mathcal{C}_F$, $\mathcal{C}_B$ and $\mathcal{C}_S$ denote the computational complexity of the forward ACSU, the backward ACSU, and the SOU, respectively. Note that we excluded the BMU in the expression since the calculation of branch metrics is the same for the local-SOVA and the MLM algorithm. For the MLM algorithm, the forward and backward ACSUs have the same computational complexity. Since we are considering the FB schedule, the forward ACSU of the local-SOVA is the same as in the MLM algorithm. However, the backward ACSU of the local-SOVA not only calculates the state metrics but also take part in the soft output calculation with ϕ and ω operators. Therefore, there will be some difference compared to the backward ACSU of the MLM algorithm.

Firstly, we perform the complexity comparison between the local-SOVA employing only ϕ operators and the MLM algorithm to observe the change of the complexity difference versus the radix order. Then, for radix-8, we perform further complexity analysis for cases where ϕ operators are gradually replaced by ω operators.

4.4.4.1 Radix-2 local-SOVA and radix-2 MLM algorithm

As shown in Section 4.3.5, the forward ACSUs of the local-SOVA and the MLM algorithm are identical. Therefore, for both algorithms, the forward ACSU and the backward ACSU have the same complexity and consist of two adders and one CS operator. For the SOU, the structures are shown in Fig. 4.5a and Fig. 4.5b for the MLM algorithm and the local-SOVA, respectively. With $\nu = 3$, the MLM SOU consists of 17 adders and 14 CS operators, whereas the SOU of the local-SOVA requires 22 adders and 7 CS operators. In total, applying (4.44), the complexity of the MLM decoder is 79 computational units and the complexity of the local-SOVA decoder is 77 computational units.

4.4.4.2 Radix-4 local-SOVA and radix-4 MLM algorithm

For the MLM algorithm, the backward and the forward ACSUs have the same complexity and consist of 4 adders and 3 CS operators. Furthermore, the SOU of the MLM algorithm begins by adding 32 intermediate values of $\Gamma_{k\to k+1}(s, s') + B_{k+2}(s')$ to $A_k(s)$, using 32

adders. Next, two maximum selection trees for each systematic bit, each consisting of 15 CS operators, are needed. Finally, each LLR value is obtained by using one extra subtractor (adder). In total, MLM decoding requires 98 adders and 108 CS operators, i.e. 206 computational units.

As for the local-SOVA, the forward ACSU is the same as for the MLM algorithm. Apart from the calculation of the state metric, the backward ACSU of the local-SOVA requires one extra ϕ operator (one adder and one CS operator) to update the reliability values, as shown in Section 4.4.1. Therefore, the backward ACSU of the local-SOVA consists of 5 adders and 4 CS operators. For the SOU, the $A_k(s)$ values are added to 8 path metrics using 8 adders. Then, a 3-layer tree of radix-4 merge operators is employed, which consists of 7 CS operators and (2×7) ϕ operators (to decode 2 bits). Therefore, the total complexity of the SOU for the local-SOVA is 22 adders and 21 CS operators. Finally, from (4.44), the complexity of the radix-4 local-SOVA is 151 computational units.

4.4.4.3 Radix-8 local-SOVA and radix-8 MLM algorithm

For the MLM algorithm, the forward/backward ACSU requires 8 adders and 7 CS operators. The MLM SOU uses 64 adders to add $A_k(s)$ to the intermediate values provided by the backward ACSU. Then, two maximum selection trees, each consisting of 31 CS operators, are employed for each systematic bit. Finally, 3 extra subtractors are required to compute the LLR of the 3 bits. In total, the complexity of the radix-8 MLM decoder is 493 computational units.

On the other hand, for the local-SOVA, the backward ACSU requires 4 extra ϕ operators to update the reliability values. For the SOU, 8 adders are first required to add the values of $A_k(s)$ to the path metrics. Then, a 3-layer tree of radix-8 merge operators is employed, which consists of 7 CS operators and (3×7) ϕ operators (to decode 3 bits). Thus, in total, the complexity of the radix-8 local-SOVA decoder is 361 computational units.

Table 4.1 summarizes the results of the complexity analysis for radix-2, radix-4 and radix-8 schemes. $\mathcal{C}_{\text{MLM}}$ and $\mathcal{C}_{\text{LSOVA}}$ denote the computational complexity of the MLM algorithm and the local-SOVA, respectively. For radix-2, the MLM algorithm and the local-SOVA have roughly the same complexity, while for radix-4 and radix-8 schemes, the local-SOVA is less complex. For high-radix schemes (4 and 8), using the local-SOVA reduces complexity by 27%.

Table 4.1: Comparison of the computational complexity of the MLM algorithm and the local-SOVA for various radix schemes (Add: adder; CS: compare-select).

Radix schemes	**Max-Log-MAP**							**local-SOVA**							$\frac{\mathcal{C}_{\text{LSOVA}}}{\mathcal{C}_{\text{MLM}}}$
	$8\times\mathcal{C}_F$		$8\times\mathcal{C}_B$		$\mathcal{C}_S$		$\mathcal{C}_{\text{MLM}}$	$8\times\mathcal{C}_F$		$8\times\mathcal{C}_B$		$\mathcal{C}_S$		$\mathcal{C}_{\text{LSOVA}}$	
	Add	CS	Add	CS	Add	CS		Add	CS	Add	CS	Add	CS		
Radix-2	16	8	16	8	17	14	**79**	16	8	16	8	22	7	**77**	**0.975**
Radix-4	32	24	32	24	34	60	**206**	32	24	40	32	22	21	**151**	**0.733**
Radix-8	64	56	64	56	67	186	**493**	64	56	96	88	29	28	**361**	**0.732**

Furthermore, the complexity per decoded bit of the MLM algorithm increases from 79 to 206/2 = 103 computational units when moving from radix 2 to radix 4. In contrast, it decreases from 77 to 75.5 computational units for the local-SOVA. Therefore, using the

local-SOVA for radix-4 schemes can further increase the efficiency of the decoder. On the other hand, for radix-8 schemes, although the local-SOVA is less complex than the MLM algorithm, its complexity per decoded bit is raised to 120.3 computational units. It is higher than for radix 4 but enables a throughput increase. Moreover, it can be significantly reduced by replacing the ϕ operators by ω operators, as shown in the follows.

4.4.4.4 Radix-8 local-SOVA with ω operators

The complexity analysis of the local-SOVA for a radix-8 scheme was reiterated by gradually replacing ϕ operators by ω operators in the backward ACSUs and in the SOU. We considered the same configurations and notations as in Section 4.4.3, and the computational complexity is still calculated based on (4.44).

The complexity comparison is shown in Table 4.2, where the MLM decoding complexity is taken as a reference for complexity normalization and performance comparison. We can observe that if we use ω operators instead of ϕ operators in the backward ACSUs (DEC 3), the resulting complexity is reduced to 67% of the complexity of the reference decoder compared to 73% if only ϕ operators are used, with no noticeable impact on the correction performance. If a lower complexity is desired, the local-SOVA with ACSU-(3,0) and SOU-(2,1) (DEC 5) can be employed to reach 63% of the reference complexity, at the cost of a performance degradation of 0.05 dB. Table 4.2 also shows that using only ω operators (DEC 6) further reduces the complexity by 1% but the resulting 0.3 dB performance loss makes this configuration less attractive.

Table 4.2: Comparison of the computational complexity of various radix-8 algorithms (CS: compare-select).

Algorithm	$8 \times \mathcal{C}_B$		$8 \times \mathcal{C}_F$		$\mathcal{C}_S$		Computational complexity $\mathcal{C}$	Complexity normalization	Performance loss at BER 10^{-6} (dB)
	Add	CS	Add	CS	Add	CS			
MLM	64	56	64	56	67	186	493	1	–
DEC 2	64	56	96	88	29	28	361	0.73	0.0
DEC 3	64	56	64	88	29	28	329	0.67	< 0.01
DEC 4	64	56	64	88	17	28	317	0.64	0.05
DEC 5	64	56	64	88	11	28	311	0.63	0.05
DEC 6	64	56	64	88	8	28	308	0.62	0.3

4.4.5 Radix-16 Local-SOVA Decoder for Convolutional Codes with Memory Length $\nu = 3$

In this section, we extend the previous study by considering a particular implementation of the radix-16 local-SOVA for $\nu = 3$ (8-state) convolutional codes. A section of a radix-16 trellis diagram is the aggregation of four consecutive sections in a radix-2 trellis diagram or, equivalently, of two consecutive sections in a radix-4 trellis diagram. With the latter representation, a pair of states s and s' between two instants k and $k+4$ are connected by two *parallel* branches as illustrated in Fig. 4.11 between $s = 2$ and $s' = 4$.

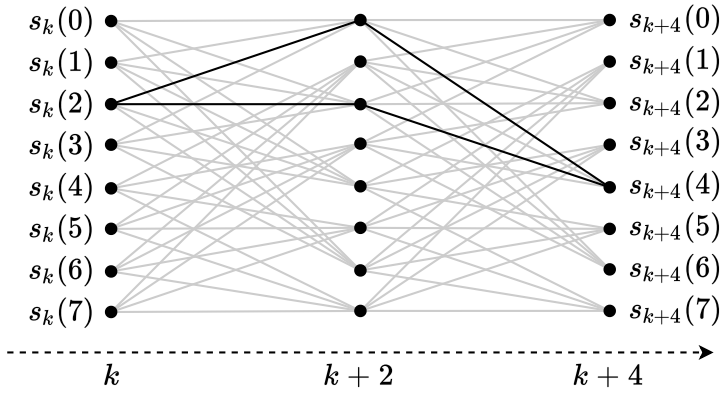


Figure 4.11: A radix-16 trellis section made of two radix-4 trellis sections.

Intuitively, one could decode this code with a radix-16 ACSU using 16-input max-select operations to produce the state metrics. However, increasing the number of inputs of the max-select operators results in a longer propagation path in the ACSU, thus lengthening the critical path of the decoder. To get around this problem, the authors of [70] suggested that we can select the branch with larger branch metric $\Gamma_{k\to k+4}(s, s')$ among the two parallel branches connecting two states, and discard the other one. Since this task could be done in the BMU to reduce by half the number of branches, we can then use radix-8 ACSUs for the calculation of the state metrics. However, the main drawback of this approach when applied to the MLM algorithm is that the branches selected after the BMU might carry either $u_j = 1$ or $u_j = 0$, for $j = k, \ldots, k+3$, which we cannot know in advance. This creates a ratio between the number of branches having $u_j = 1$ and the number of branches having $u_j = 0$ that varies over different trellis sections and different received frames. This causes a major problem for the MLM SOU since it naturally employs max-select operations with a constant number of inputs referring to hard decisions 1 and 0.

On the other hand, from the local-SOVA perspective, the two parallel branches processed by the BMU can be considered as the merge of two paths. Hence, we can use the path merging operation to produce one path carrying the selected hard decision and the updated reliability value. The BMU can then forward 64 paths to radix-8 ACSUs, which in turn produce 8 output paths for the SOU to finally compute the soft output. So, in the radix-16 local-SOVA decoder, the BMU, the ACSU and the SOU constitute a path merging binary tree with 7 layers: the first layer is in the BMU, and thus not in the critical path, the three next layers are in the ACSU, and the last three layers in the SOU that are also not in the critical path. In addition, since the order of the input paths does not affect the result of the merge operations, the local-SOVA is immune to the problem of the non-static ratio of hard decision value mentioned above for the MLM algorithm.

Extensive simulations with the radix-16 local-SOVA were also carried out. Similarly to the radix-8 case, we first performed a radix-16 local-SOVA with only ϕ operators. Then, we gradually substituted ω operators for ϕ operators in order to observe the behavior of the decoder. The results are shown in Fig. 4.12. We can observe that we can still replace ϕ operators by ω operators in the ACSU and in the first layer of the SOU with a negligible degradation in performance. However, further substitutions are not recommended since a penalty of 0.4 dB at 10^{-4} of BER can be observed if we use solely ω operators.

When considering higher radix orders such as 32 and 64 for the same convolutional

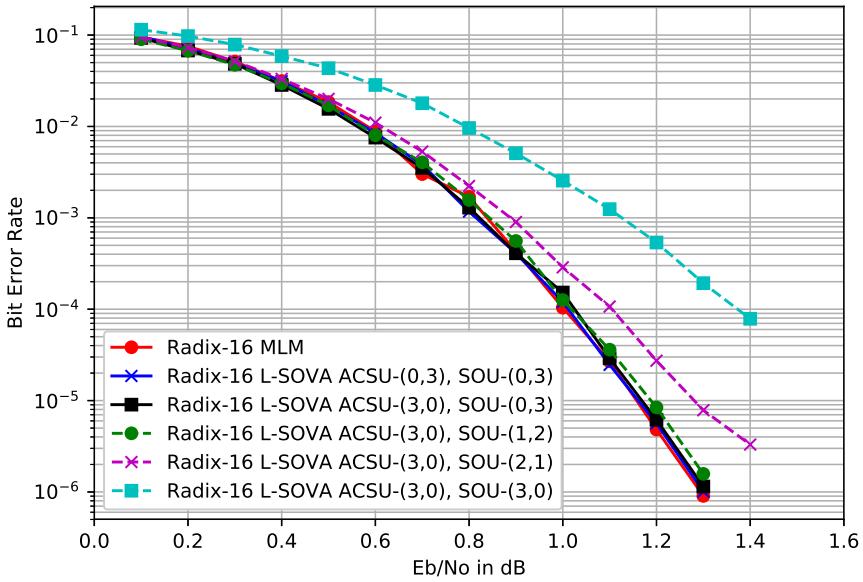


Figure 4.12: BER performance of a LTE decoder using radix-16 MLM algorithm, local-SOVA and its variants with $K = 1056$, $r = 1/3$ after 5.5 iterations. AWGN channel, BPSK modulation.

code, there are respectively 4 and 8 parallel branches connecting two states in each high-radix trellis section. In this case, the BMU selects the path with the largest path metric among the 4 or 8 paths. Moreover, since the BMU does not involve a recursive loop as in the ACSU, it can be pipelined to ensure that the critical path always resides in the ACSU. However, since the complexity of the decoder increases exponentially with the number of bits decoded simultaneously, it is then necessary to find the processing order of the paths with the best compromise between throughput and complexity. Nonetheless, using local-SOVA with high-radix orders provides an ultra high throughput solution for convolutional and turbo codes since the critical path of the decoder can always remains in the radix-8 ACSU while decoding an increasing number of systematic bits in a single clock cycle.

4.5 Conclusion

In this chapter, we have introduced a new SISO decoding algorithm for convolutional codes: the local-SOVA. The decoder architecture for the local-SOVA is shown to exhibit a more hierarchical structure and a lower computational complexity than the conventional Max-Log-MAP algorithm.

We observed that using local-SOVA in radix-8 LTE turbo decoders significantly reduces the computational complexity of the decoder compared to the respective radix-8 Max-Log-MAP architecture. For instance, employing the local-SOVA algorithm for radix-8 decoding of the LTE turbo code reduces the complexity by 33% without any performance degradation and by 36% with a slight penalty of only 0.05 dB. Moreover, the local-SOVA makes it possible to increase the radix-order without penalizing the error correction per-

formance or the critical path of the decoder, but at the cost of added complexity. These advantages make local-SOVA our first-choice algorithm for developing high-radix turbo decoders.

The study of the proposed algorithm and its results were published in the IEEE Transaction on Communications [21].

Chapter 5

The Local-SOVA in Unrolled-XMAP Architecture

The previous chapter showed that, when used with high radix orders, the local-SOVA decoder has a lower computational complexity than the MLM decoder. Therefore, this chapter deals with the implementation of the computational units of radix-4, radix-8 and radix-16 local-SOVA decoders in the UXMAP architecture introduced in Section 3.3.2 for high-throughput applications. For each radix order, hardware architectures are proposed and implemented to observe how the computational complexity savings translate into area savings.

The rest of the chapter is organized as follows. Section 5.1 briefly describes the UXMAP architecture and provides an overview of the benefits of employing the local-SOVA in the UXMAP architecture. Then, the state-of-the-art UXMAP architecture with the MLM algorithm is described in Section 5.2. Section 5.3 presents the implementation of the radix-4, radix-8 and radix-16 local-SOVA decoders. These high-radix schemes are also compared with the standard MLM decoder and with each other in terms of area complexity, error correction performance and latency/throughput. Finally, Section 5.4 concludes the chapter.

5.1 Introduction

It was recently shown that a turbo decoder using the UXMAP architecture and implemented in CMOS 28 nm technology could achieve a throughput of 100 Gp/s for $K = 128$ with the radix-4 MLM algorithm and 4 decoding iterations (8 HIs) [57, 67]. According to Eq. (3.13) in Section 3.3.2, the throughput of a turbo decoder with the UXMAP architecture increases linearly with K since it produces K decoded bits in a single clock cycle when the pipeline is full. Therefore, a throughput of 1 Tb/s can in theory be achieved with the UXMAP architecture when $K \approx 1280$. However, for a given number of decoding iterations, the area complexity of the UXMAP decoder also increases with the frame size K. Nonetheless, when K increased, the correlation between the two constituent RSC decoders is reduced provided the interleaver is well designed [34] and the error correction performance is improved. On that basis, the authors in [67] showed that the UXMAP architecture with $K = 256$ and $K = 512$ only needs 3 and 2.5 decoding iterations, respectively, to achieve the same correction performance as in the case of $K = 128$ with 4 iterations. As a result, for $K = 512$, the decoder throughput is increased by 4 times, while the area complexity is only increased by 2.5 times, thus, resulting in an area efficiency increased by 1.6 times.

Furthermore, the previous chapter showed that the local-SOVA has a lower computational complexity than the MLM algorithm, expecially for high radix orders. For the radix-4 case, the analysis in Chapter 4 led to a reduction of 27% in computational complexity. Assuming that this computational complexity saving can be translated into area saving, one can expect that embedding a radix-4 local-SOVA in the UXMAP architecture can reduce the decoder area by 25 to 30% compared to a radix-4 MLM UXMAP decoder. Equivalently, keeping the same area complexity as the MLM decoder, we can increase K by 1.4 times by using the local-SOVA decoder instead, and thus, have a throughput gain and an area efficiency gain of 1.4.

The promising results of Chapter 4 encouraged us to consider using even higher radix schemes (radix-8, radix-16) with the UXMAP architecture. There are then several aspects to be taken into account. Assuming that using higher radix schemes does not lower the operating frequency, then the latency of the decoder can be reduced, since more bits are processed in a clock cycle. This characteristic is particularly interesting for applications requiring low processing latency. Using higher radix schemes also means that the number of computational units (BMU, ACSU, SOU) and registers in the pipeline of the UXMAP can be decreased, leading to a reduction in the area complexity. But, on the other hand, the complexity of the computational units also increases with the radix order. Therefore, the actual impact of increasing the radix order can only be meaningfully assessed through implementing these high radix local-SOVA decoders in an UXMAP architecture.

In the next section, the fixed-point implementation of the UXMAP with the radix-4 MLM algorithm is discussed. This will serve as a basis for the design of our local-SOVA/UXMAP decoder and for comparisons. All the implementations and comparisons were carried out for the LTE turbo code using rate-1/2 RSC component codes.

5.2 Fixed-Point Implementation of the Radix-4 MLM/UXMAP Decoder

This section describes the implementation of turbo decoders using the UXMAP architecture with the radix-4 MLM decoding algorithm. As described in Section 3.3.2, the fully pipelined UXMAP architecture unrolls the decoding iterations and applies a spatial parallelization of the XMAP cores where each XMAP core processes a sub-trellis of length K_P within a half-iteration. The authors in [67] have shown that $K_P = 32$ is the optimal length for each XMAP core as smaller lengths entail a performance degradation, which in turns requires more iterations to be compensated for. It is also shown in [67] that, compared to higher sub-trellis lengths ($K_P = 64$ or $K_P = 128$), using $K_P = 32$ with a small ACQ length at the first iteration does not degrade the error correction performance and up to 40% of the area complexity can then be saved. Moreover, using $K_P = 32$ also enables frame size flexibility to support $K = 128, 64, 32$ [66].

The XMAP core is made of computational units and pipeline registers as described in Section 3.3.2. Three distinct computational units, BMU, ACSU and SOU, constitute the backbone of the XMAP core and hence of the UXMAP architecture. The next section deals with the implementation of these computational units.

5.2.1 Computational Units Implementation

5.2.1.1 Branch Metric Unit

The BMU calculates all the branch metrics of the current trellis section and send them to the ACSU for further process. The constituent RSC code has coding rate $R = 1/2$. Therefore, for a radix-2 trellis section k, there are one systematic bit and one parity bit, and we have 4 branch metrics values:

$$\begin{cases} \Gamma_k(00) = 0 \\ \Gamma_k(01) = \mathrm{L}_k^p \\ \Gamma_k(10) = \mathrm{L}_k^s + \mathrm{L}_k^a \\ \Gamma_k(11) = \mathrm{L}_k^s + \mathrm{L}_k^a + \mathrm{L}_k^p \end{cases} \tag{5.1}$$

This representation suppose that all the branch metrics are normalized by subtracting the metric of the *all-zeros* branch.

The radix-4 branch metrics can be directly calculated from the radix-2 branch metrics of sections k and k+1. The implementation of the radix-4 BMU is therefore a network of adders and can be straightforwardly implemented.

5.2.1.2 Add Compare Select Unit

The ACSU receives the branch metrics from the BMU and computes the forward and backward state metrics recursively. For instance, the forward state metric calculation in radix-4 scheme is expressed as

$$A_{k+2}(s) = \sum_{s'} \big(A_k(s') + \Gamma_{k \to k+2}(s', s) \big), \tag{5.2}$$

where the size of the set of state s' in the summation is 4. Equation (5.2) is implemented by an ACS operator. The radix-4 ACS operator can be implemented as a tree-like structure as shown in Figure 5.1a or can be carried out by six compare-select (CS) operators followed by a look-up table (LUT) [56, 76, 77] as shown in Figure 5.1b. The tree-like structure has lower complexity but it has higher critical path [56]. Moreover, the tree-like architecture could not be pipelined because it would have the same throughput and latency compared to the radix-2 case but with higher complexity. On the other hand, although the implementation using LUT in Figure 5.1b is more complex (6 CS operators) than to the tree-like implementation (3 CS operators), this architecture shortens the critical path of the ACSU by about 50% as shown in [56]. Therefore, it is the one that we chose to be implemented in the UXMAP architecture, since we are seeking high operating frequency for high throughput and low latency.

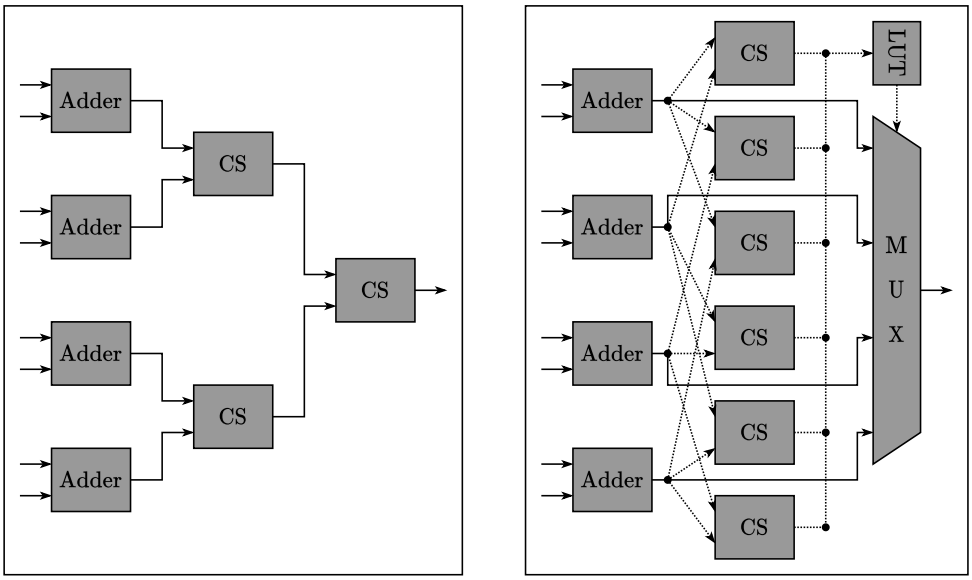


Figure 5.1: The schematic of the radix-4 Max-Log-MAP ACS operator using (a) a tree-like architecture, and (b) a LUT approach.

5.2.1.3 Soft Output Unit

The input of the SOU are metrics $A_k(s)$, $\Gamma_{k\to k+2}(s, s')$, and $B_{k+2}(s')$. It performs the soft output calculation for two bits u_k and u_{k+1} as follows:

$$\begin{aligned} \mathrm{L}(u_i) = &\max_{(s',s)|u_i=1} \left(A_k(s') + \Gamma_{k\to k+2}(s', s) + B_{k+2}(s)\right) \\ &- \max_{(s',s)|u_i=0} \left(A_k(s') + \Gamma_{k\to k+2}(s', s) + B_{k+2}(s)\right), \quad i = k, k{+}1. \end{aligned} \tag{5.3}$$

Then, the extrinsic information of the systematic bits can be derived from the soft output by subtracting the a priori LLR $\mathrm{L}^a(u_i)$ and the channel LLR $\mathrm{L}^c(u_i)$ from the soft output

$$\mathrm{L}^e(u_i) = \mathrm{L}(u_i) - \big(\mathrm{L}^a(u_i) + \mathrm{L}^c(u_i)\big). \tag{5.4}$$

Finally, an extrinsic scaling factor of 0.75 is applied to the extrinsic information.

An architecture for the radix-4 SOU was proposed in [59] and is shown in Figure 5.2 where CS-2 denotes a CS operator with two inputs and CS-4 denotes a CS operator with four inputs. Note that the SOU should be pipelined (through the use of the pipeline registers) to ensure that its critical path is smaller than that of the ACSU. In this case, the CS-4 stage of the SOU can be carried out by using the LUT approach as in the ACSU which takes only one clock cycle since its critical path is equal to the critical path of the ACSU. Alternatively, it can be implemented with the tree-like architecture where another extra pipeline stage should be added between two layers of the tree to ensure that its critical path is always smaller than that of the ACSU. As a result, depending on the choice of the CS-4 architecture, the SOU takes 4 or 5 clock cycles to produce two extrinsic LLRs for bit u_k and u_{k+1}.

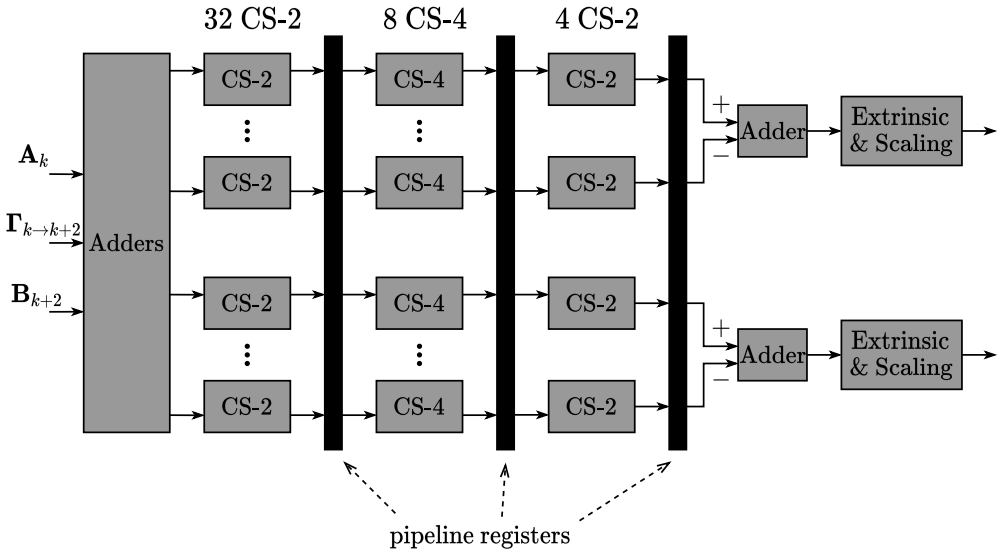


Figure 5.2: The architecture schematic of the radix-4 MLM SOU that produces two extrinsics.

5.2.2 Metrics Quantization

In this work, we only considered uniform quantization to represent the different metrics. It is defined by two parameters: the range of the metric and the number of bits used to represent that metric, also referred as to the metric bitwidth. For a metric in the decoder with a fixed range of values, the bitwidth should be chosen very carefully since the hardware complexity increases linearly with the defined bitwidth but, on the other hand, the error correction performance of the decoder worsens when the bitwidth decreases.

For the computational units in the UXMAP, there are 4 types of metrics that should be considered for quantization: the channel metrics representing the LLRs of the systematic and parity bits, the branch metrics, the state metrics, and the extrinsic information (which is also the input *a priori* information). To this end, we only needs to define the range of the channel metrics since other metrics are calculated based on the channel metrics. However, the bitwidth for each metric should be defined very carefully to avoid using more bits or less bits than necessary.

5.2.2.1 Quantization of Channel Values and Extrinsic Information

The turbo decoder receives the channel LLRs computed by the demodulator. Assuming a transmission over an AWGN channel using BPSK modulation, the received channel LLRs at the receiver are

$$L_i^c = \frac{2y_i}{\sigma^2}, \quad i = 0, \ldots, N-1 \tag{5.5}$$

where σ is the variance of the additive white Gaussian noise. Note that in order to calculate the channel LLRs in (5.5), receiver needs to estimate of the noise variance σ^2. This estimation is not required with the MLM algorithm [46] since a constant multiplier to the channel LLRs does not affect the decoder performance. The MLM decoder can therefore directly perform the quantization of the received values $y_i, i = 0, \ldots, N-1$.

In the fixed-point implementation of the UXMAP, the two's complement representation of numbers is used. The reason is that it facilitates the modulo normalization of the state metrics (see Section 5.2.2.3). Let A be the interval of the quantization, meaning that y_i is quantized between $[-A, A)$. With the bitwidth of w bits, the two's complement quantization function Q is defined as

$$y_i^Q = Q(y_i) = \text{sat}\left(\left\lfloor y_i \frac{2^w - 1}{A} + 0.5 \right\rfloor, 2^w\right), \tag{5.6}$$

where $\lfloor x \rfloor$ is the floor function returns the greatest integer less than or equal to x, and the saturation function $\text{sat}(a, b)$ is defined as

$$\text{sat}(a, b) = \begin{cases} a, & \text{if } a \in [-b, b-1] \\ b-1, & \text{if } a > b-1 \\ -b, & \text{if } a < -b \end{cases} \tag{5.7}$$

The value of A should be chosen so that it provides the optimal performance of the code. If A is too large, most of uncertain values around zero would be quantized to zero. On the contrary, if A is too small, the saturation function prevents the quantized value from taking high reliability values. In [78], the authors proposed that, for a rate 1/2 turbo code, the value of A could be taken around 1.2. Furthermore, as the code rate increases, the value of A can be decreased.

On the other hand, the choice of the channel bitwidth w depends on the target error performance and the decoder architecture. Typical values of w are between 3 to 6 bits [78]. For the UXMAP architecture, a quantization of 6 bits for the channel LLRs was chosen in [57, 66, 67]. From the chosen channel bitwidth w, the bitwidth for the extrinsic information (and also the *a priori* information) is then $w + 1 = 7$ bits [67, 78]. Therefore, the value of the extrinsic information is in the range of $[-2A, 2A)$, and if the SOU produces an extrinsic value out of this range, it should be saturated as given by (5.7) with $b = 2^{w+1}$ before sending to the other constituent decoder.

5.2.2.2 Quantization of Branch Metrics

The bitwidth of the branch metrics can be directly deduced from the branch metric calculation. For a radix-2 trellis section k, according to (5.1), if L_k^s and L_k^p are quantized with w bits and L_k^a is quantized with $(w + 1)$ bits, the bitwidth of the radix-2 branch metric should be $w + 2$.

The bitwidth of the high-radix branch metrics can then be obtained from the bitwidth of radix-2 branch metrics. For example, the bitwidth of the radix-4 branch metrics is $w + 3$, and the bitwidth of the radix-8 branch metrics is $w + 4$. Table 5.1 provides the bitwidth of the branch metrics for different radix orders with the channel bitwidth $w = 6$ bits.

Radix	Branch Metric Bitwidth
Radix-2	$w + 2 = 8$
Radix-4	$w + 3 = 9$
Radix-8	$w + 4 = 10$
Radix-16	$w + 5 = 11$

Table 5.1: Bitwidth of the branch metrics given the channel bitwidth $w = 6$ bits

Another important metric that we need to exploit is the maximum difference between the branch metrics, denoted by Δ_Γ. This metric will be used later to decide the bitwidth of the state metrics. Similarly to the process of finding the bitwidth of the branch metrics, this maximum difference can be deduced by setting the extreme case where all the channel values and *a priori* information reach their minimum values. In this case, the maximum difference is the difference between the branch with all bits equal to one and the branch with all bits equal to zero. The resulting value is $\Delta_\Gamma = 2 \times 2^w$ for the radix-2 scheme, 4×2^w for the radix-4 scheme, 6×2^w for the radix-8 scheme, and 8×2^w for the radix-16 scheme.

5.2.2.3 Quantization of State Metrics

In order to determine the number of bits to represent the state metrics, the authors in [79] showed that for the Viterbi algorithm, the difference between path metrics was bounded by a value at any time instant on the trellis diagram. Since the forward and backward state metric recursion share the same process as the path propagation in the Viterbi algorithm, the difference between forward state metrics and the difference between backward state metrics are also bounded at any time instant. We denote this bound value for the state metric as Δ_{SM}

Furthermore, since the state metrics are calculated recursively, if no normalization is used, the metrics would increase over time. Therefore, metric normalization should be employed in fixed-point implementation. A straightforward normalization method involves subtracting the minimum state metric from all the state metrics. Then, the minimum number of bits required to represent the state metric is $\lceil \log_2(\Delta_{\mathrm{SM}}) \rceil$ [78]. However, this method implies that additional hardware resources are needed to find the minimum metric and then to subtract it from all the metrics. This increases both the complexity and the critical path of the recursion unit.

Another method to normalize the state metrics consists in using directly two's complement arithmetic where modulo normalization corresponds to the overflow mechanism of adders [79]. The advantage of this method is that there is no additional hardware cost provided that the state metric range is sufficiently large. For a given set of metrics

$\{m_1(t), \ldots, m_n(t)\}$ that change over time, let us assume that the difference between these metrics is always bounded by $\Delta_{\max}$, i.e. at any time instant t,

$$|m_i(t) - m_j(t)| \leq \Delta_{\max}, \quad \forall i, j \in 1, \ldots, n. \tag{5.8}$$

Then, if we want to represent these metrics with W bits with two's complement representation ranging in $[-2^{W-1}, 2^{W-1} - 1)$, the following condition has to be met [79]:

$$\Delta_{\max} \leq 2^{W-1} - 1. \tag{5.9}$$

Figure 5.3 shows two examples of a valid and a non-valid bitwidth value W. Note that the set of possible values for the quantized metrics are equally spaced on the circumference of the circle. As the metrics increase over time, their two's complement quantized values rotate counter-clockwise over the circle. Condition (5.9) ensures that, at a given time instant, the difference between two metrics is always less than or equal to $\Delta_{\max}$ as shown in Figure 5.3a, and does not take value $2^{W-1} - \left(\Delta_{\max} \bmod 2^{W-1}\right)$ as shown in Figure 5.3b.

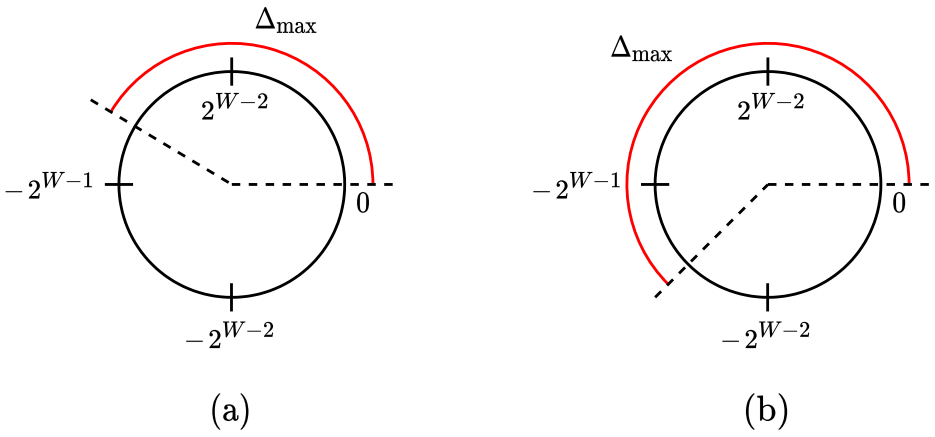


Figure 5.3: Example of (a) a valid value of bitwidth W, and (b) a non-valid value of bitwidth W which causes a wrong result.

For trellis section k, the MLM algorithm calculates:

$$A_{k+1}(s) = \max\left(A_k(s_0) + \Gamma_k(s0, s), A_k(s1) + \Gamma_k(s1, s)\right). \tag{5.10}$$

If we assume that, at time instant k, the difference between the state metrics $A_k(s_0)$ and $A_k(s_1)$ reaches the bound value Δ_{SM} and the difference between two branch metrics also reaches the maximum value of Δ_Γ mentioned in the previous section, then, the bitwidth for $A_{k+1}(s)$ has to be large enough to support a difference equal to $(\Delta_{\text{SM}} + \Delta_\Gamma)$.

Moreover, for the soft output calculation, we use two intermediate values $A_k(s_0) + \Gamma_k(s_0, s)$ and $A_k(s_1) + \Gamma_k(s_1, s')$, which are added up to $B_{k+1}(s)$ and $B_{k+1}(s')$ respectively. Now, if the differences between the two forward state metrics, between the two branch metrics and between the two backward state metrics reach their corresponding bound values, then, the bitwidth of the state metric has to be large enough to support the difference of $(2\Delta_{\text{SM}} + \Delta_\Gamma)$ since for the representation of the *a posteriori* LLR.

Consequently, for the MLM algorithm, the maximum difference $\Delta_{\max}$ that the state metrics have to support while fulfilling condition (5.9) is

$$\Delta_{\max} = 2\Delta_{\text{SM}} + \Delta_\Gamma, \tag{5.11}$$

and the bitwidth of the state metric, denoted as w_{SM}, can be obtained from (5.9) as

$$w_{\mathrm{SM}} = \lceil \log_2 (2\Delta_{\mathrm{SM}} + \Delta_\Gamma + 1) + 1 \rceil, \tag{5.12}$$

where $\lceil x \rceil$ is the ceiling function that returns the least integer greater than or equal to x.

As explained in 5.2.2.2, the value of Δ_Γ is $2T \times 2^w$, where T is the radix order and w is the bitwidth of the channel values. Regarding the value of Δ_{SM}, there are several bounds derived in the literature [79, 78, 49]. Nonetheless, to derive the exact bound of the state metric, the authors in [49] proposed a process using simulation. By starting from all-zero state metrics, we perform the state metric recursion with all the channel values and *a priori* information having their corresponding minimum values, until the system reaches a steady state. Then, the maximum difference between state metrics is Δ_{SM}. For the LTE constituent RSC code [5], the simulation method provided a value of Δ_{SM} equal to 4×2^w.

Table 5.2 shows the resulting bitwidths for the state metrics representation in the MLM algorithm for radix-2, radix-4, radix-8 and radix-16 LTE RSC code with the channel bitwidth $w = 6$. We can observe that the bitwidth for the state metrics remains 11 bits for radix-2, radix-4 and radix-8 schemes, despite the changes in the value of $\Delta_{\max}$, and only the radix-16 scheme requires a state metrics bitwidth of 12 bits.

Radix	Δ_Γ	Δ_{SM}	$\Delta_{\max}$	w_{SM}
Radix-2	128	256	640	11 bits
Radix-4	256	256	768	11 bits
Radix-8	384	256	896	11 bits
Radix-16	512	256	1024	12 bits

Table 5.2: Maximum values for Δ_Γ, Δ_{SM}, $\Delta_{\max}$ and minimum bitwidth of the state metrics for various radix orders for the MLM algorithm, given the channel bitwidth is $w = 6$ bits

5.2.3 Performance of the MLM/UXMAP Architecture

In this section, we take the architecture of the computation units as well as their corresponding quantization schemes into account, and perform simulation to assess the performance. It is important to note that the results from the simulation are bit-true. Therefore, an implementation of the solution in hardware should produce the same performance.

We simulated three settings of an MLM/UXMAP turbo decoder ($K_P = 32$) for different frame sizes $K = 128, 256, 512$ and 8, 6, 5 HIs respectively. These settings are chosen because they provide similar correcting performances while having different number of iterations. The coding rate 1/3 LTE turbo code with QPP interleaver is considered [5]. The modulation scheme is BPSK and the channel is AWGN. Furthermore, the channel LLRs are quantized using $w = 6$ bits. The corresponding BER curves are shown in Figure 5.4. We can clearly see that the three settings provide similar performance with a 0.25 dB coding gain for $K = 256$ with 6 HIs, and 0.25 dB performance degradation for $K = 512$ with 5 HIs, compared to the case of using $K = 128$ with 8 HIs.

Concerning area and throughput, the implementation of the decoder with $K = 128$ and 8 HIs was reported in [66] with an area complexity of 16.54 mm^2 and a throughput

of 102.4 Gb/s for a clock frequency equal to 800 MHz in CMOS 28 nm technology node. Therefore, the area and throughput of the two other decoders can be deduced from these values. The estimates of the throughput of the decoder for $K = 256$ and $K = 512$ are 204.8 Gb/s and 409.6 Gb/s, respectively. The estimated area for the case $K = 256$ and 6 HIs is 24.81 mm^2, and 41.35 mm^2 for the case $K = 512$ with 5 HIs. As a result, using the setting $K = 512$ and 5 HIs provides an area efficiency of 9.91 Gb/s/mm^2, which is greater than the area efficiency of the setting $K = 256$ and 6 HIs (8.25 Gb/s/mm^2) and almost twice the area efficiency of the setting $K = 128$ and 8 HIs (6.19 Gb/s/mm^2).

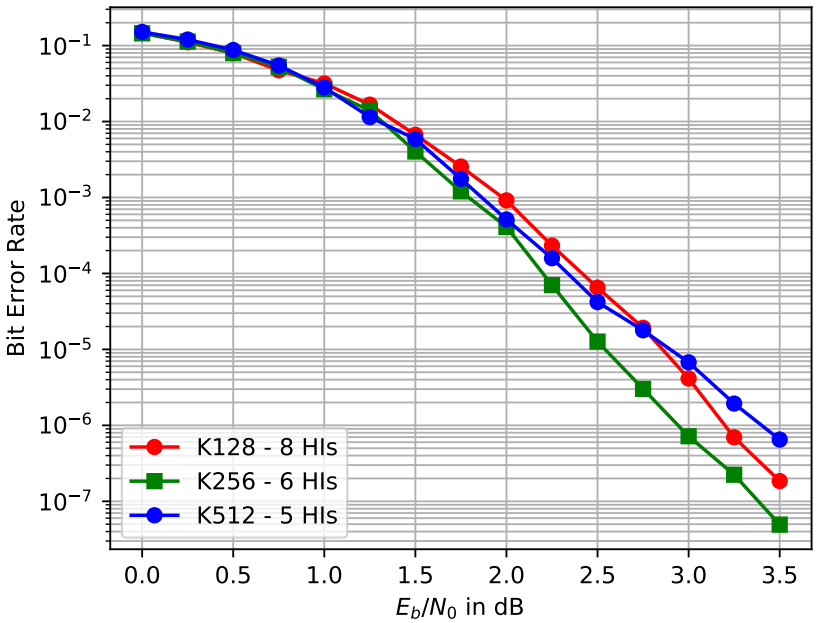


Figure 5.4: BER performance of UXMAP decoder with $K_P = 32$ for different settings.

5.3 High-Radix Local-SOVA in UXMAP Architecture

This section focuses on the application of the local-SOVA to the UXMAP architecture. As shown in Section 4.4.4, the computational complexities of the local-SOVA and MLM algorithm are similar for radix 2, but the local-SOVA is less complex for radices 4 and 8. Therefore, we skipped the radix-2 implementation and we focused directly on the radix-4, radix-8 and radix-16 implementations of the local-SOVA.

In this section, we first perform the analysis on the bitwidth of the state metric for the local-SOVA. Then, we implemented the three computational units (BMU, ACSU and SOU) for the local-SOVA and compared them with the MLM units in terms of area complexity. Moreover, in order to perform area complexity comparisons between radix-4, radix-8 and radix-16 local-SOVA implementations, we implemented the equivalent of 12 radix-2 trellis sections, which correspond to 6 radix-4 trellis sections, 4 radix-8 trellis sections and 3 radix-16 trellis sections, respectively.

5.3.1 Metric Quantization in the Local-SOVA

For the local-SOVA, since the decoding process is different from the MLM algorithm, the analysis related to the bitwidth of state metrics has to be carried out again. In the local-SOVA, the ACSU also performs the maximum selection for the state metric as in (5.10). Therefore, the bitwidth of the state metrics should also support the difference of $\Delta_{\mathrm{SM}} + \Delta_{\Gamma}$. However, for the soft output calculation, the local-SOVA employs the metrics $(A_{k+1}(s)+B_{k+1}(s))$ instead of $(A_k(s')+\Gamma_k(s',s)+B_{k+1}(s'))$ in the MLM algorithm. So, the maximum difference that the state metrics have to support in the SOU of the local-SOVA is only $2\Delta_{\mathrm{SM}}$. Consequently, the value of $\Delta_{\max}$ in the local-SOVA is

$$\Delta_{\max} = \max\left(\Delta_{\mathrm{SM}} + \Delta_{\Gamma}, 2\Delta_{\mathrm{SM}}\right) \tag{5.13}$$

Table 5.3 shows the resulting bitwidths for the state metrics representation in the local-SOVA algorithm for radix-2, radix-4, radix-8 and radix-16 LTE RSC code with the channel bitwidth $w = 6$. We can observe that the bitwidth for the state metrics remains 11 bits for all radix orders, despite the changes in the value of $\Delta_{\max}$.

Radix	Δ_{Γ}	Δ_{SM}	$\Delta_{\max}$	w_{SM}
Radix-2	128	256	512	11 bits
Radix-4	256	256	512	11 bits
Radix-8	384	256	640	11 bits
Radix-16	512	256	768	11 bits

Table 5.3: Maximum values for Δ_{Γ}, Δ_{SM}, $\Delta_{\max}$ and minimum bitwidth of the state metrics for various radix orders for the local-SOVA , given the channel bitwidth is $w = 6$ bits

5.3.2 Radix-4 Computational Units

For the radix-4 local-SOVA, the BMU operates in the same way as for the MLM algorithm. Therefore, we only focus on the implementation of the ACSU and the SOU for the radix-4 local-SOVA in this section.

5.3.2.1 The radix-4 ACSU

Section 4.4.1 mentioned the possibility of implementing the radix-4 ACSU through a tree of merge operations. However, for the same reason as for the radix-4 maximum selection in the MLM decoder, the tree-like architecture introduces long critical paths into the ACSU, thus drastically limiting the maximum operating frequency of the decoder. Therefore, one need to resort to the radix-4 ACSU using the LUT approach. The local-SOVA ACSU, apart from calculating the next state metrics (the surviving paths), also has to find the reliability values for each surviving path to be provided to the SOU. To this end, the authors in [76] proposed a method for calculating the reliability values that is compatible with the LUT approach. Different from the original method that updates the reliability values by the HR and BR using the metric differences, the proposed method acts on the

absolute value of the metrics and provides the reliability values represented by the path metrics of the contenders of the surviving path.

Let us denote by $\{P_0, P_1, P_2, P_3\}$ the set of 4 paths considered for the selection of the surviving path for state s' at instant $k+2$ in the radix-4 ACSU. Each path P_i is a 3-tuple consisting of the path metric M_i, the set of two hard decisions $\mathbf{u}_i$ and the set of two reliability values $\mathbf{l}_i$. For a forward ACSU, each path metric corresponds to the sum of the forward state metric $A_k(s)$ and the branch metric $\Gamma_{k\to k+2}(s, s')$. The hard decisions are the two systematic bits labeling the corresponding branch. The ACSU takes these 4 paths and produce a surviving path

$$P^{s'} = \{M^{s'}, \mathbf{u}^{s'}, \mathbf{l}^{s'}\}, \tag{5.14}$$

where $M^{s'} = A_{k+2}(s')$ and $\mathbf{u}^{s'} = \{u_0^{s'}, u_1^{s'}\}$ are the surviving path metric and the hard decisions. The values of $\mathbf{l}^{s'} = \{l_0^{s'}, l_1^{s'}\}$ are defined as the metrics of the contender paths, which are interpreted as the highest path metrics having different hard decisions with respect to $u_0^{s'}$ and $u_1^{s'}$, respectively. This new definition of the reliability values is different from the original definition $\mathbf{L}$ in Section 4.3, which is based on the difference between path metrics. Nonetheless, we will come back to the $\mathbf{L}$ values later in the next section dealing with the SOU.

With the LUT approach, the ACSU first computes the set of difference values between the path metrics as in the MLM algorithm. Then, from the sign of these metric differences, the index of the path with the highest metric and its corresponding hard decisions are retrieved. For the computation of the reliability values, the input paths can be arranged in a pre-determined order according to the hard decisions. Therefore, from the index of the maximum path and the signs of the path metric differences, the index of the contender paths can the deduced, and the reliability values computed. For example, we can arrange the input paths according to the hard decisions such that $\mathbf{u}_0 = \{0, 0\}$, $\mathbf{u}_1 = \{0, 1\}$, $\mathbf{u}_2 = \{1, 0\}$, and $\mathbf{u}_3 = \{1, 1\}$, and from the LUT we know that M_1 is the highest path metric. Then, metric of the contender path for the first bit is either M_2 or M_3 and can be decided by the sign of the difference between them. Similarly, the metric of contender path for the second bit is either M_0 or M_2 and is decided by the sign of $(M_0 - M_2)$. Note that the resulting reliability values produced by this approach are equivalent to the original approach shown in Figure 4.8 in Section 4.4.1.

Figure 5.5 shows the hardware architecture of the radix-4 ACSU producing one surving path. For a convolutional codes with 2^ν states, the ACSU should employ 2^ν replica of Figure 5.5. Compared to the radix-4 MLM architecture, the radix-4 ACSU of the local-SOVA has a complexity overhead for the LUTs for the contender paths and the multiplexers for the selection of these paths.

5.3.2.2 The radix-4 SOU

According to the original local-SOVA that we have studied in Chapter 4, for a code with 2^ν states, the SOU receives 2^ν surviving paths from the forward ACSU, and it also receives 2^ν values of $B_{k+2}(s')$ from the pipeline registers in the UXMAP. First, the SOU adds the values of $A_{k+2}(s')$ to $B_{k+2}(s')$ to compute the new path metrics for the 2^ν paths. Then, the merge operators are applied in a dichotomous fashion to produce the soft output. The update rules in the merge operators can be either only HR (ω operator) or both HR and BR (ϕ operator).

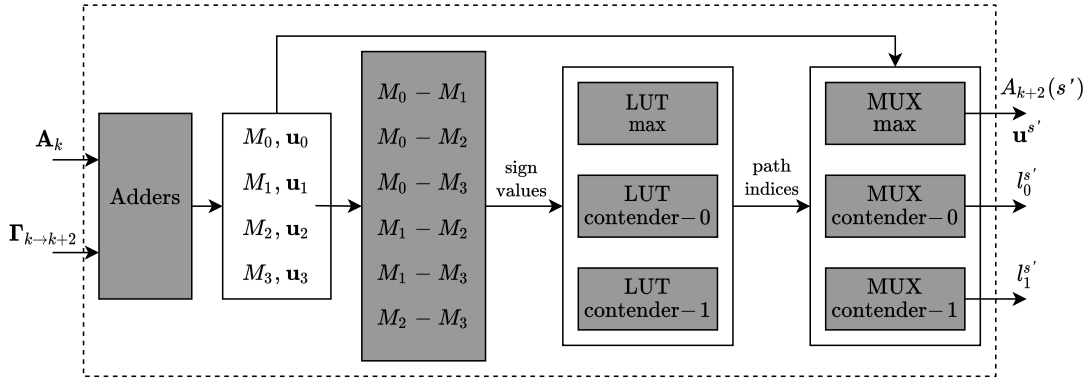


Figure 5.5: ACSU processing of one surviving path in the radix-4 local-SOVA.

However, the reliability values received from the ACSU being the path metrics $\mathbf{l}$ instead of the metric differences $\mathbf{L}$, the application of the update rules, especially the BR, can be obstructed since these rules should act on metric differences. As a first approach, one could convert the values of $\mathbf{l}$ into $\mathbf{L}$ by subtracting $\mathbf{l}$ from their corresponding forward state metric $A_{k+2}(s')$. However, this process requires a high number of subtractors and introduces a high complexity overhead. Therefore, the authors in [76] proposed a low-complexity solution for the SOU, described hereafter.

In Section 4.4.3, for the LTE RSC code with 8 states (3 layers of merge in the SOU), we have observed from the simulations that, in the first two layers of the SOU, one can employ only ω operators (i.e. only apply HR) and then use ϕ operators (both HR and BR) in the last layer with negligible performance degradation of the decoder. Therefore, we can split 8 paths in the SOU into two sets of 4 paths, the upper set and the lower set, and perform ω operators with each set. Note that using the ω operators for a set of paths means that we find the maximum path metric, its corresponding hard decisions and update the reliability values for the hard decision using HR. For example, the output reliability (metric difference) of the i-th hard decision in the upper set is expressed as

$$L_i^h = \min\left(L_i^h, \{\Delta_i^{h,j}\}\right), \tag{5.15}$$

where the subscript h is the index of the path with the highest metric in the upper set, L_i^h is the reliability value (metric difference) of the i-th decision of the h-th path, and $\{\Delta_i^{h,j}\}$ is the set of metric differences between the h-th path and the other paths having a different i-th hard decision.

Since the min operator in (5.15) is commutative and associative, we can rewrite it as

$$\begin{aligned} L_i^h &= \min\left(\min(\{\Delta_i^{h,j}\}), L_i^h\right) \\ &= \min\left(\Delta_{i,\min}^h, A_{k+2}(h) - l_i^h\right), \end{aligned} \tag{5.16}$$

where $\Delta_{i,\min}^h = \min(\{\Delta_i^{h,j}\})$. The process of implementing (5.16) can be as follows. First, we find the index h of the path with the highest metric, which can be obtained by calculating the metric differences between all pairs of paths followed by a LUT. Then, from the index h and the hard decision differences between paths, we can get the set of $\{\Delta_i^{h,j}\}$ and select the minimum metric difference between them as $\Delta_{i,\min}^h$. Furthermore, from the index h, we can also obtain $A_{k+2}(h)$ and l_i^h. Then, we can compute $L_i^h = A_{k+2}(h) - l_i^h$, and perform the min operation between L_i^h and $\Delta_{i,\min}^h$. The advantage of this method is

that we only have to convert the value of l_i^h into the value of L_i^h for the h-th path. The process can be generalized to other bit decisions and to the lower set of paths (denoted by subscript g).

After having selected the path metric and the hard decisions, the application of (5.16) to the set of upper paths and to the sets of lower paths gives two paths for the last layer of the SOU. We then can employ the ϕ operator to find the final hard decisions and the *a posteriori* LLRs, and from that, the scaled extrinsic information can be obtained.

The architecture of the SOU is depicted in Figures 5.6a and 5.6b. In order to reduce the critical path of the decoder, the implementation of Eq. (5.16) is divided into the first clock cycle cycle (Figure 5.6a) and the second clock cycle (Figure 5.6b). Note that Figure 5.6a shows the architecture for the set of upper paths. The architecture for the set of lower paths is identical but with different inputs. Overall, the SOU takes 4 clock cycles to produce the extrinsic information. Therefore, for the set of upper and lower paths, intermediate registers are required to hold the values of $A_{k+2}(h)$, $\mathbf{l}^h$, and the values of $\mathbf{\Delta}^h$ as the metric difference between the h-th path and other paths.

5.3.2.3 Error correction performance and implementation results

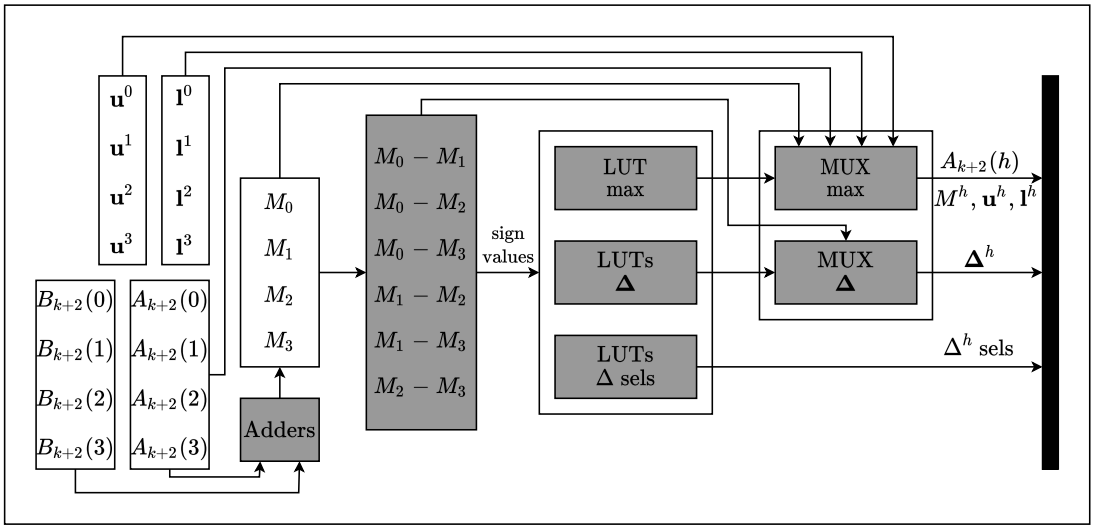
We have simulated the error correction performance of the fixed-point implementations of the LTE turbo decoder using a radix-4 local-SOVA decoder and the UXMAP architecture. The simulation settings are the same as in Section 5.2.3. Figure 5.7 compares the resulting performance curves with those obtained using the MLM algorithm. We can see that the local-SOVA with the use of ω operators in the first two layers of the SOU only entails a neligible performance degradation (less than 0.05 dB) compared to the MLM algorithm for all three settings.

The computational units (BMU, ACSU, and SOU) of the radix-4 local-SOVA were implemented in VHDL, and the designs were placed and routed with Synopsis IC Compiler for a CMOS 28 nm process under PVT (Process/Voltage/Temperature) worst case constraints and a target clock frequency of 800 MHz. For comparison, the respective BMU, ACSU and SOU of the radix-4 MLM algorithm were also implemented, placed and routed. The resulting area complexity comparison is shown in Table 5.4. We can see that, as expected, the BMU complexity is the same for both algorithms and that the area of the local-SOVA ACSU is a little higher due to the LUTs and multiplexers required for the contender paths. In return, the SOU of the local-SOVA decoder occupies only half of the area of the MLM SOU.

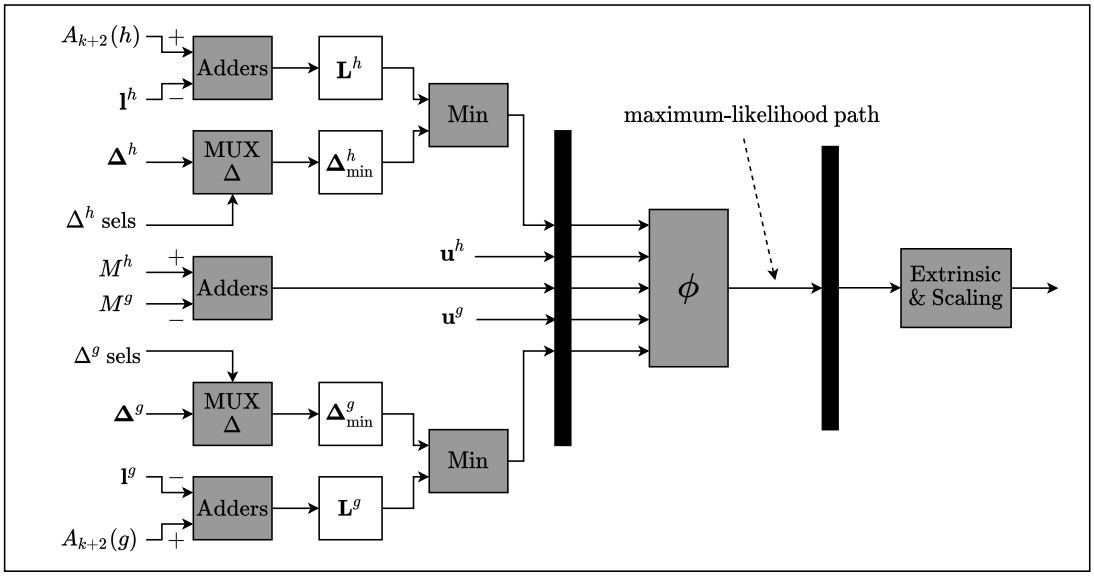
To decode 6 radix-4 trellis sections (equivalent to 12 radix-2 trellis sections), the radix-4 scheme needs 6 BMUs, 6 ACSUs and 6 SOUs. The overall complexity of the radix-4 MLM and the radix-4 local-SOVA decoders is shown in the last column of Table 5.4. From these results, we can state that using the local-SOVA yields a complexity reduction of 33% compared to the MLM, which is in line with the computational complexity analysis performed in Section 4.4.4.

5.3.3 Radix-8 Computational Units

The radix-8 XMAP core consists of the implementation of a radix-8 BMU, a radix-8 ACSU and a radix-8 SOU. For a convolutional code with 8 states, the radix-8 trellis is fully connected but there are no parallel branches. Then, the radix-8 BMU of the local-SOVA



(a)



(b)

Figure 5.6: The SOU architecture for the radix-4 local-SOVA with (a) the hardware processing the first part of (5.16) for the set of upper paths (the process for the lower paths is identical) and (b) the hardware processing the second part of (5.16) for both upper paths and lower paths, and then applying the ϕ operation and calculating the extrinsic information.

and the MLM decoder should be the same and their implementation is straightforward from the branch metric calculation.

The ACSU of the local-SOVA takes 8 input paths and produces one surviving path for each state s' at time instant $k+3$. The surviving path can be described similarly to (5.14), except that each path has now 3 hard decisions as well as 3 contender paths, one for each hard decision. The input paths can also be organized in a predetermined order according to the hard decisions. Then, the metric differences of path pairs are computed, and LUTs are employed to find the index of the surviving path as well as the indices of the contender paths. The architecture of the radix-8 ACSU is similar to the radix-4 ACSU

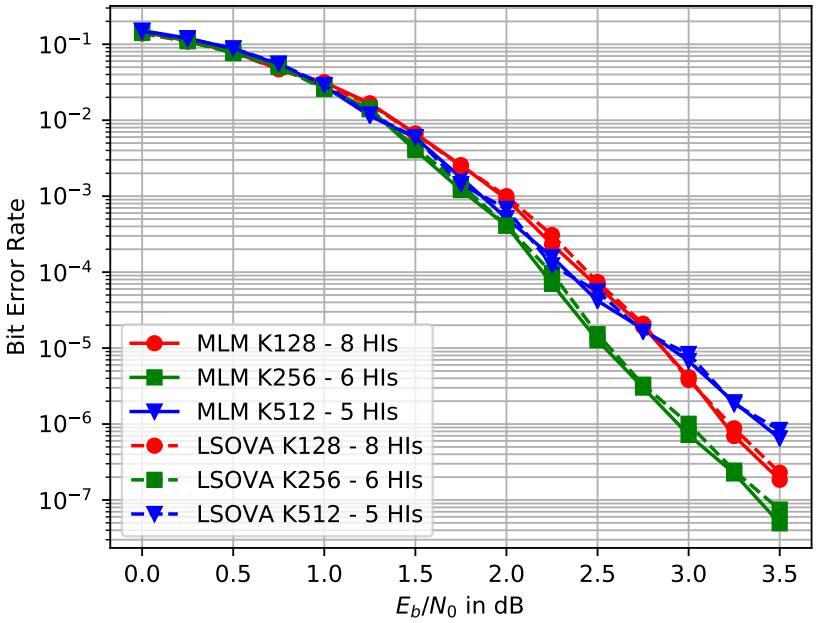


Figure 5.7: BER performance comparison between the radix-4 local-SOVA and the radix-4 MLM algorithm employed in the UXMAP decoder with $K_P = 32$ for frame sizes $K = 128$, $K = 256$ and $K = 512$.

Algorithm	BMU	ACSU	SOU	6 radix-4 trellis sections
Max-Log-MAP	1200	3885	10485	93420
Local-SOVA	1200	4076	5267	63258
$\frac{\text{Local-SOVA}}{\text{Max-Log-MAP}}$	1.0	1.05	0.5	0.677

Table 5.4: Post place & route area of computational units (in μm^2) in radix-4 MLM and radix-4 local-SOVA decoders.

architecture shown in Figure 5.5, except that 26 subtractors are required compared to 6 subtractors for the radix-4 ACSU, since there are 8 path metrics. Furthermore, the size of the LUTs and multiplexers required to find and select a path among 8 is also increased. Nonetheless, despite these changes, the critical path of the radix-8 ACSU does not limit the target frequency of 800 MHz.

Concerning the SOU, the radix-8 architecture is also similar to the radix-4 one shown in Figure 5.6. The only changes is that some operators now require 3 units instead of 2 in order to process the 3 bits of the radix-8 unit in parallel. Therefore, it is expected that the complexity of the radix-8 SOU does not exceed 1.5 times the complexity of the radix-4 SOU.

Overall, the radix-8 local-SOVA uses the same decoding structure as the radix-4 local-SOVA, where ϕ operators are employed in the ACSUs and in the last layer of the SOU and ω operators are used in the first two layers of the SOU. Therefore, the error correction performance of the UXMAP turbo decoder using radix-8 local-SOVA is not expected to differ from the radix-4 case, shown in Figure 5.7.

The area complexity resulting from the implementation of the radix-8 computational units is shown in Table 5.5 for the local-SOVA and MLM decoders. There is an overhead of 29% in the ACSU of the local-SOVA decoder due to the LUTs for the contenders. However, an area reduction of 73% is observed for the SOU of the local-SOVA decoder compared to the MLM decoder. Overall, to decode 4 radix-8 trellis sections (equivalent to 12 radix-2 trellis sections) using the local-SOVA decoder yields an area complexity reduction of 42%. On the other hand, when compared to the radix-4 local-SOVA decoder, the radix-8 local-SOVA decoder is approximately 1.5 times more complex. Note that this increase in complexity is traded for a decrease in latency and an increase in throughput since a radix-8 decoder decodes a frame 1.5 times faster than a radix-4 decoder, if we neglect the I/O latency.

Algorithm	BMU	ACSU	SOU	4 radix-8 trellis sections
Max-Log-MAP	5341	9022	26444	163228
Local-SOVA	5341	11673	6792	95224
$\frac{\text{Local-SOVA}}{\text{Max-Log-MAP}}$	1.0	1.29	0.26	0.58

Table 5.5: Post place & route area of computational units (in μm^2) in radix-8 MLM and radix-8 local-SOVA decoders.

5.3.4 Radix-16 Computational Units

This section describes the implementation of the radix-16 local-SOVA decoder for the same code as above.

5.3.4.1 Straightforward approach

Intuitively, one can implement the radix-16 computational units with the same structures as for the radix-4 and radix-8 cases. In the straightforward radix-16 implementation, the BMU calculates 128 branch metrics and pass them to the ACSU. The ACSU adds the state metrics to the branch metrics, calculates 128 metric differences and uses a LUT to find the maximum state metric. For comparison, in the radix-4 and radix-8 cases, 6 metric differences and 28 metric differences have to be considered, respectively. Therefore, the radix-16 SOU can be implemented with a structure similar to the radix-4 and radix-8 SOUs but with an increased complexity due to the higher number of bits to be processed in parallel (4 bits).

Table 5.6 shows the area complexity of the computational units for the local-SOVA and the MLM algorithm, as well as the area required to decode 3 radix-16 trellis sections (equivalent to 12 radix-2 trellis sections). Compared to the radix-4 local-SOVA, the complexity of the radix-16 local-SOVA decoder is 3.7 times higher while the latency is divided by only a factor of 2 (and consequently the throughput is doubled). Therefore, modifications should be introduced to the radix-16 local-SOVA architecture to make it more efficient.

According to Section 4.4.5, since there are two parallel branches connecting two states in a radix-16 trellis section, the BMU can actually select one branch and discard the

Algorithm	BMU	ACSU	SOU	3 radix-16 trellis sections
Max-Log-MAP	22457	26301	64574	339996
Local-SOVA	22457	45800	8949	231618
$\frac{\text{Local-SOVA}}{\text{Max-Log-MAP}}$	1.0	1.74	0.14	0.68

Table 5.6: Post place & route area of computational units (in μm^2) in radix-16 MLM and radix-16 local-SOVA decoders without specific optimization.

other based on the branch metric. This means that it only passes 64 branch metrics to the ACSU, and the ACSU can use the radix-8 structure to calculate the state metrics and calculate the reliability values. Note that this is already a saving in area since this method can be directly applied to the first half process of the XMAP core, where we only need to calculate the state metrics and do not need to calculate the reliability values. However, for the second half, care should be taken to the radix-16 BMU, ACSU and SOU for calculating the reliability values.

5.3.4.2 Optimized radix-16 BMU

For a radix-16 trellis section of the LTE RSC code, all pairs of parallel branches have hard decisions in the form $\mathbf{u} = \{u_0, u_1, u_2, u_3\}$ and $\mathbf{u}' = \{\overline{u_0}, u_1, \overline{u_2}, \overline{u_3}\}$, where $\overline{u_i}$ is the logical not of the bit $u_i \in 0, 1$. This means that the metric difference between two parallel branches, denoted by Δ_{BMU}, is given by the reliability values of bits at positions 0, 2, and 3 for the surviving branch. These reliability values could then be passed to the ACSU with the surviving branch metrics and be used to compute the output reliability values as described in Section 4.4.5. However, as the the computation of the reliability values in the ACSU is already a complex process, adding another layer to update the reliability values would result in an excessive complexity overhead and a longer critical path. Instead, the values of Δ_{BMU} can be pushed into pipelined registers, and only be used at the end of the SOU, where the maximum-likelihood path has already been determined. In this case, the SOU will select the Δ_{BMU} corresponding to the maximum-likelihood path to update the final reliability values (for the bits at positions 0, 2 and 3) before calculating the extrinsic information. The ACSU can then ignore these metric differences and process the 64 branches from the BMU as if there were no *a priori* reliability values.

The structure of the radix-16 BMU that selects one out of two parallel branches was proposed in [70] and is depicted in Figure 5.8. As mentioned above, the systematic bits are the same for both parallel branches at position 1 and they are different at other positions. The same can be observed for the parity bits, where the parity bits are the same for both parallel branches at position 2 and they are different at other positions. Therefore, out of 12 values (4 values of systematic LLRs, 4 values of parity LLRs, and 4 values of a priori LLRs), we only need 9 values to calculate the metric difference Δ_{BMU} and to select the surviving branch metric. Furthermore, although there are 64 pairs of parallel branches, there are only 16 values of metric difference Δ_{BMU} due to repetition. The adopted radix-16 BMU needs two clock cycles to produce the branch metrics since it has to select 16 surviving metrics before calculating 64 branch metrics.

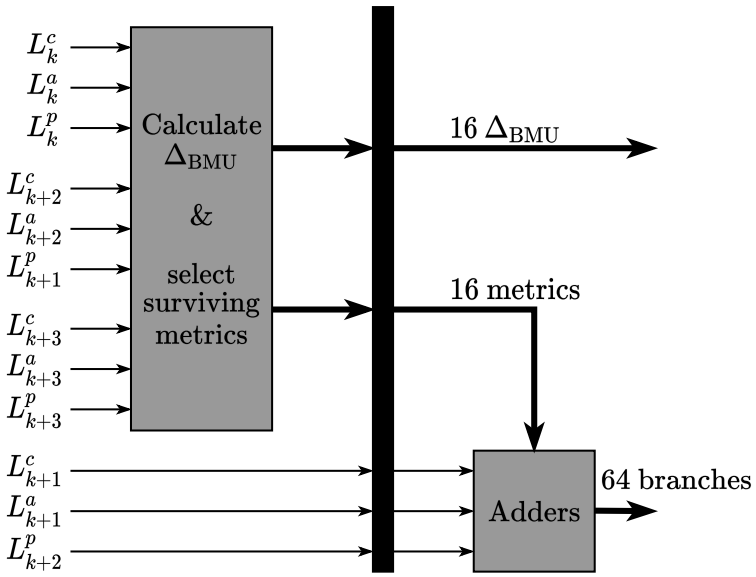


Figure 5.8: Selected BMU architecture for the radix-16 local SOVA decoder.

5.3.4.3 Optimized radix-16 ACSU

As mentioned earlier, the structure of the radix-8 ACSU can be used in order to implement the radix-16 ACSU for the local-SOVA decoder. However, in the radix-8 ACSU, the hard decisions of each path are known in advance, which facilitates the selection of the contender paths. In the radix-16 case, except for the position 1, the hard decisions of the branches coming from the BMU are not known in advance due to the merge in the BMU. The uncertainty about hard decisions makes the selection for the contender paths much more complex. Hence, this translates into more complex LUTs for contender paths than in the radix-8 case. We called the ACSU which implements this approach *type-I* ACSU.

Another alternative structure which can be considered for the radix-16 ACSU is the use of four radix-2 CS operators whose outputs are passed to a radix-4 CS operator implemented using the LUT approach, similarly to [70]. The corresponding architecture is shown in Figure 5.9. In order to avoid additional complexity in the LUTs for contender paths, the metric differences Δ_{ACSU} and decision differences $\mathbf{d}_{\text{ACSU}} = \{d_{\text{ACSU},0}, d_{\text{ACSU},1}, d_{\text{ACSU},2}, d_{\text{ACSU},3}\}$[1] obtained from the radix-2 CS operators are passed to the SOU to be processed later. Δ_{ACSU} and $\mathbf{d}_{\text{ACSU}}$ can be selected according to the maximum-likelihood path founded in the SOU. Then, the SOU can use them to update the final reliability values with the HR. Concerning the ACSU, the reliability values of the surviving paths are note updated during the radix-2 CS. Only the path metrics and and hard decisions are selected for the surviving paths. We called the corresponding architecture *type-II* ACSU.

5.3.4.4 Radix-16 SOU

The radix-16 SOU can reuse the architecture shown in Figure 5.6 for the radix-4 case. However, the SOU architecture in Figure 5.6b has to be slightly modified to update the reliability values after finding the maximum-likelihood path using the 16 metric differences Δ_{BMU} provided by the BMU. To prevent the critical path from being increased by the

[1] $d_{\text{ACSU},i} = 1$ means that the ith hard decisions between two paths are different.

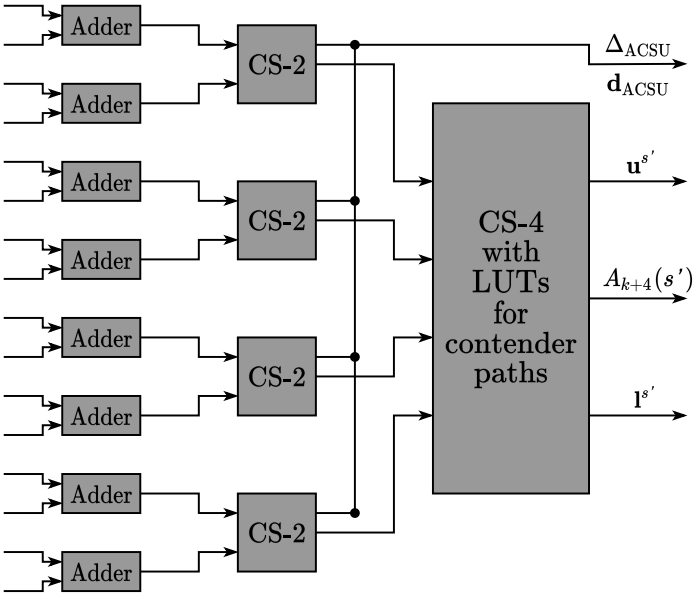


Figure 5.9: Type-II ACSU architecture for radix-16 local SOVA decoder.

SOU, we had to add another pipeline stage to perform the final update process right before the extrinsic calculation and scaling. Furthermore, when the type-II radix-16 ACSU is implemented, the SOU has to include the metric difference Δ_{ACSU} in the final update of the reliability values.

Let us denote by $\mathbf{L}^{\mathrm{ML}} = \{L_0^{\mathrm{ML}}, L_1^{\mathrm{ML}}, L_2^{\mathrm{ML}}, L_3^{\mathrm{ML}}\}$ the reliability values of the maximum-likelihood path before the final update taking account of Δ_{BMU}, Δ_{ACSU} and $\mathbf{d}_{\mathrm{ACSU}}$. At this stage, from the maximum-likelihood path, we can choose the corresponding metric difference $\Delta_{\mathrm{BMU}}^{\mathrm{ML}}$ from the BMU and the corresponding metric difference $\Delta_{\mathrm{ACSU}}^{\mathrm{ML}}$ and decision difference $\mathbf{d}_{\mathrm{ACSU}}^{\mathrm{ML}}$ from the ACSU (in the case of type-II ACSU). Then the reliability values of the maximum-likelihood path are updated using HR as follows:

$$L_i^{\mathrm{ML}}(\text{type-II}) = \begin{cases} \min\left(L_i^{\mathrm{ML}}, \Delta_{\mathrm{BMU}}^{\mathrm{ML}}, d_{\mathrm{ACSU},i}^{\mathrm{ML}} \times \Delta_{\mathrm{ACSU}}^{\mathrm{ML}}\right), & i = 0, 2, 3 \\ \min\left(L_i^{\mathrm{ML}}, d_{\mathrm{ACSU},i}^{\mathrm{ML}} \times \Delta_{\mathrm{ACSU}}^{\mathrm{ML}}\right), & i = 1 \end{cases} \tag{5.17}$$

The update process for L_1^{ML} does not include $\Delta_{\mathrm{BMU}}^{\mathrm{ML}}$ because in the merge of the BMU, the hard decisions at position 1 are the same. If the type-I ACSU is employed, the final reliability updates are

$$L_i^{\mathrm{ML}}(\text{type-I}) = \begin{cases} \min\left(L_i^{\mathrm{ML}}, \Delta_{\mathrm{BMU}}^{\mathrm{ML}}\right), & i = 0, 2, 3 \\ L_i^{\mathrm{ML}}, & i = 1 \end{cases} \tag{5.18}$$

5.3.4.5 Area complexity and error correction performance

We described the hardware architecture of radix-16 local-SOVA computational units in VHDL language and implemented them. The designs were placed and routed with Synopsis IC Compiler for a CMOS 28 nm process under worst case PVT constraints and a target clock frequency of 800 MHz. The resulting area complexity for type-I radix-16 and type-II radix-16 is shown in Table 5.7. The area of the BMU in both cases is the same. However, with the use of a simpler architecture consisting of 4 radix-2 CS and a

radix-4 CS, the type-II ACSU yields a lower complexity (55%) compared to the type-I ACSU. Although the type-II SOU is slightly more complex than the type-I SOU (4.5%), the overall area for 3 radix-16 trellis sections (equivalent to 12 radix-2 trellis sections) of type-II local-SOVA is only 70% of the area of the type-I solution. Furthermore, the type-II radix-16 local-SOVA decoder has the same area complexity as the radix-8 local-SOVA decoder, while providing a lower latency and higher throughput solution.

Architecture	BMU	ACSU	SOU	3 radix-16 trellis sections
Radix-16 type-I	5491	30860	8778	135387
Radix-16 type-II	5491	16996	9174	94983

Table 5.7: Post place & route area of computational units (in μm^2) in type-I and type-II radix-16 local-SOVA decoders.

The error correction performance of the type-I and type-II radix-16 local-SOVA decoder is shown in Figure 5.10, where it is also compared with the radix-4 MLM algorithm. For the type-I radix-16 local-SOVA, since we omit the initial reliability values in the BMU and only use them for an update at the end of the process, the performance is slightly degraded with less than 0.1 dB loss for all settings. Meanwhile, the type-II radix-16 local-SOVA further skips the reliability update in the first layer of the ACSU and also uses them for the final update, the performance is then further degraded resulting in an overall loss of about 0.1 dB.

Compared to the radix-4 local-SOVA decoder, the area complexity of the type-II radix-16 local-SOVA decoder is 1.5 times higher, but the latency of the decoder is divided by 2 times and the throughput doubled. Therefore, the UXMAP architecture with type-II radix-16 local-SOVA is recommended for applications requiring low latency and high throughput and tolerating a small performance loss (about 0.1 dB) as well as a higher complexity.

5.4 Conclusion

In this chapter, we have first reviewed the implementation of turbo decoders using UXMAP architecture with the MLM algorithm. The corresponding computational units (BMU, ACSU, SOU) are described in the case of the radix-4 MLM algorithm. The area complexity and the error correction performance of the UXMAP architecture with the radix-4 MLM algorithm were also shown, which served as the baseline to compare with our implementations of the local-SOVA in the UXMAP architecture.

The radix-4 local-SOVA decoder was first presented with the architecture of the computational units. Compared to the radix-4 MLM decoder, the radix-4 local-SOVA decoder employed in the UXMAP architecture provides a lower complexity solution with a saving of 33% in area complexity, at the cost of a slight performance loss (smaller than 0.05 dB). In light of this complexity reduction, the radix-8 and radix-16 local-SOVA were also investigated. Both schemes yield a higher area complexity than the radix-4 scheme, but have the advantage of a lower latency. The radix-8 scheme is 1.5 times more complex but provides a 1.5 gain in latency/throughput and no performance loss compared to the

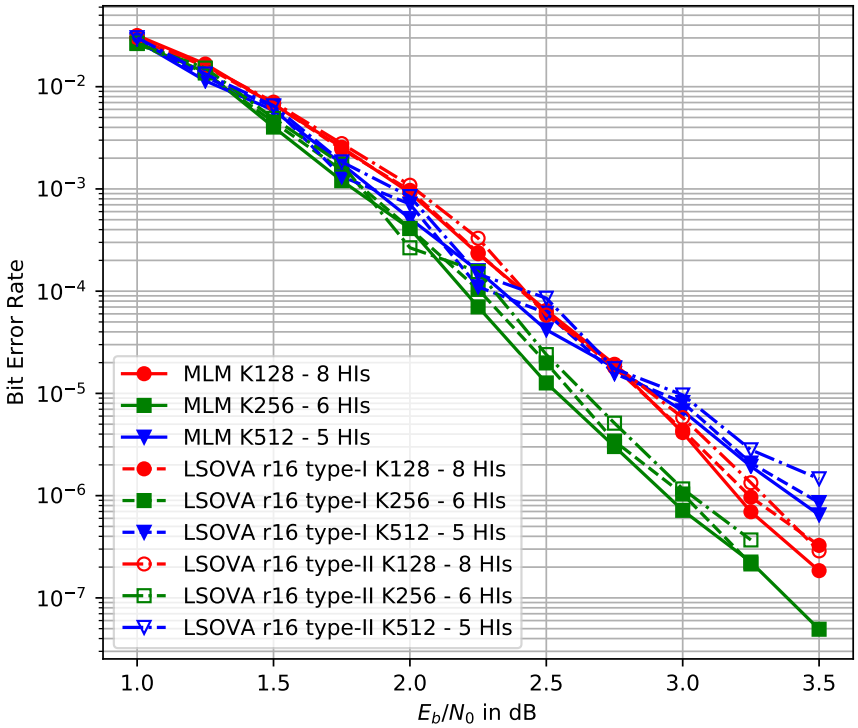


Figure 5.10: BER performance comparison between the MLM algorithm and the radix-16 local-SOVA employed in the UXMAP decoder with $K_P = 32$ for frame sizes $K = 128$, $K = 256$ and $K = 512$.

radix-4 scheme. Even better, the radix-16 scheme is also 1.5 times more complex but can still increase the latency/throughput gain to a factor of 2, at the cost of a very small degradation in performance (about 0.1 dB) compared to the radix-4 scheme. Therefore, depending on the requirements of the application, the radix-4, radix-8 or radix-16 schemes of the local-SOVA can be can be beneficially employed instead of the MLM decoder in the UXMAP architecture.

Chapter 6

SISO Decoding Algorithms Using the Dual Trellis

In search of decoding algorithms for high-throughput contexts, a part of my thesis work was devoted to the efficient decoding of high-rate convolutional codes. Therefore, this chapter focuses on decoding convolutional codes with high coding rates ($r > 1/2$) using the trellis of the corresponding dual codes, i. e., the *dual trellis*.

The authors in [80] have proposed a derivation of the BCJR algorithm based on the dual code which is perfectly equivalent to the BCJR algorithm applied to the original convolutional code. For a code with rate $r = k/n$, the dual code has rate $(n-k)/n$. If $k > n/2$, i.e. if $r > 1/2$, the dual code has a rate lower than the original one. Therefore, the dual trellis is less complex than the original trellis since it has fewer branches per trellis section. Thus, it is expected that if the BCJR algorithm is applied to the dual trellis, the process of finding the *a posteriori* probability estimate and then convert it into the original code is also less complex than the straightforward application of the BCJR to the original code.

The rest of the chapter is organized as follows. Section 6.1 introduces the benefits of using the dual trellis to decode a convolutional code. Then, Section 6.2 describes the state-of-the-art method for constructing the dual trellis, given a convolutional code with rate $k/(k+1)$. The application of the BCJR algorithm and its logarithmic version to the dual trellis are also presented and it is shown that they could provide a higher throughput compared to the MLM algorithm. Section 6.3 then provides a method that we have proposed to derive the dual trellis for a convolutional code with arbitrary rate k/n. The construction method can be applied to a punctured convolutional code so that decoding on the dual trellis can be applied. Then, a new sub-optimal low-complexity decoding algorithm based on the dual trellis is presented and analyzed in Section 6.4. It is shown that despite a minor loss of about 0.2 dB in performance, the new decoding algorithm significantly reduces the decoder complexity compared to the state-of-the-art algorithm. Finally, Section 6.5 concludes the chapter.

6.1 Introduction

In communication systems requiring high data throughput, channel coding using high rates is preferred as the number of transmitted redundancy bits is limited. For a (k, n, ν) convolutional code, high coding rates ($r = k/n > 1/2$) can be achieved by puncturing or by using true high-rate convolutional encoders.

With the former approach, the information message is encoded by a low-rate mother code (e.g. with code rate $1/n$). Then, the output from the mother code is punctured using a *puncturing pattern* $\mathbf{P}$ to shorten the code length and achieve the desired coding rate. At the receiving side, the decoder first sets the LLR of the punctured bits to 0. Then, the decoder can use the MLM algorithm, the SOVA, or the local-SOVA on the trellis of the low-rate mother code to compute the *a posteriori* LLRs.

The advantage of puncturing is that the decoder structure designed for the original mother code can be reused for the high-rate punctured codes. Therefore, the complexity for decoding high-rate punctured codes is about the same as for the original code. Moreover, the code enjoys the flexibility to change the code rate by applying different puncturing patterns at the encoder, without changing the decoder structure. However, this also implies that, for high rates, the decoder inherits the throughput and decoding latency of the mother code. In other words, regardless of the coding rate, the decoder architecture always employs the trellis of the mother code, and thus, its throughput and latency are defined by this trellis. Furthermore, as mentioned in Section 3.1.4, the decoders designed for high coding rates usually adopt the ACQ initialization technique with long acquisition lengths to maintain the performance. Hence, the decoding latency is further increased and the throughput is reduced. In [81], the authors proposed a solution to mitigate the long acquisition lengths, namely the trellis compression. However, the solution requires extra hardware resources for the ACQ steps [63]. As a consequence, for high-rate schemes, other code families such as LDPC codes may provide a better performance in terms of decoding complexity, throughput and latency compared to turbo codes employing puncturing.

As mentioned earlier, another alternative to achieve a high code rate k/n is to use an *true high-rate* encoder, characterized by a $k \times n$ generator matrix. However, the main drawback of this method is the complexity of the trellis diagram of the code. Consider a (k, n, ν) convolutional codes, then for a state $s \in \{0, \ldots, 2^\nu - 1\}$, there are 2^k branches coming in and out, and there are $2^{k+\nu}$ branches in total for each trellis section. Processing such a trellis section is equivalent to processing the radix-2^k scheme for a code with rate $1/n$. Since the complexity of the MLM algorithm increases exponentially with the radix order, which is equal to k in this case, for high coding rates (e.g. $k > 4$), decoding algorithms based on the trellis of the true high-rate encoder are not necessarily attractive.

To overcome the high complexity drawback in the *original* trellis, decoding a true high-rate convolutional code can be carried out with the *dual trellis* of the code. For a convolutional code $\mathcal{C}$ with rate k/n encoded by generator matrix $\mathbf{G}(D)$, there exists an associated *dual code* $\mathcal{C}^\perp$ with rate $(n-k)/n$ generated by matrix $\widetilde{\mathbf{H}}(D)$ such that $\mathbf{G}\widetilde{\mathbf{H}}^\top = 0$. In other words, any codeword generated by $\mathbf{G}(D)$ should be orthogonal with all codewords generated by $\widetilde{\mathbf{H}}(D)$. The generator matrix $\widetilde{\mathbf{H}}(D)$ is usually referred to as the *reciprocal parity-check matrix* since it is the reciprocal of the parity-check matrix $\mathbf{H}(D)$ of the code $\mathcal{C}$. Then, the *reciprocal dual trellis* is constructed according to $\widetilde{\mathbf{H}}(D)$, and we will refer it as the *dual trellis* for short. In contrast, we will refer to the trellis generated by

the generator matrix $\mathbf{G}(D)$ as the *original trellis*. A very interesting property of the dual trellis is that there are $2^{(n-k+\nu)}$ branches in a trellis section. As the code rate increases, the value of $(n-k)$ decreases, and for a convolutional code with rate $k/(k+1)$, there are $2^{\nu+1}$ branches, which corresponds to the conventional radix-2 scheme.

6.2 The Dual Trellis and the MAP-Based Algorithms on the Dual Trellis

As in Chapter 2, we begin with a presentation related to the dual trellis and the MAP-based algorithms performed on the dual trellis. In this section, we only consider the case of true high-rate convolutional codes with rate $k/(k+1)$ unless stated otherwise. The generalization to arbitrary code rates k/n and punctured high-rate codes will be dealt with in subsequent sections.

6.2.1 The Dual Trellis

As already stated in Section 2.2.1, a (k, n, ν) convolutional code $\mathcal{C}$ can be characterized by a polynomial generator matrix $\mathbf{G}(D)$ with rank k, consisting of $k \times n$ polynomials $g_{i,j}(D) \in F_2(D)$:

$$\mathbf{G}(D) = \begin{pmatrix} g_{0,0}(D) & g_{0,1}(D) & \dots & g_{0,n-1}(D) \\ g_{1,0}(D) & g_{1,1}(D) & \dots & g_{1,n-1}(D) \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0}(D) & g_{k-1,1}(D) & \dots & g_{k-1,n-1}(D) \end{pmatrix}, \tag{6.1}$$

where $F_2(D)$ is the field of formal Laurent series with binary coefficients. We further define the constraint length of the ith input of the generator matrix $\mathbf{G}(D)$ as

$$\lambda_j = \max_{0 \le j < n} \big\{ \deg(g_{i,j}(D)) \big\}, \tag{6.2}$$

where $\deg(g_{i,j}(D))$ denotes the degree of the polynomial $g_{i,j}(D)$. Then, we define the *overall constraint length* of the generator matrix as the sum of the constraint lengths for all inputs

$$\lambda = \sum_{i=0}^{k} \lambda_i \tag{6.3}$$

Let us denote by $\mathbf{u} = (\mathbf{u}_0, \mathbf{u}_1, \dots)$ and $\mathbf{c} = (\mathbf{c}_0, \mathbf{c}_1, \dots)$ the input sequence and output sequence of the encoder, respectively, where $\mathbf{u}_i = (u_i^0, u_i^1, \dots, u_i^{k-1})$ is the ith input vector consisting of k bits and $\mathbf{c}_i = (c_i^0, c_i^1, \dots, c_i^{n-1})$ is the ith output vector consisting of n bits.

The sequence $\mathbf{u}$ and $\mathbf{c}$ can be written as

$$\mathbf{u}(D) = \sum_{i=0}^{\infty} \mathbf{u}_i D^i \in F^k(D), \tag{6.4}$$

and

$$\mathbf{c}(D) = \sum_{i=0}^{\infty} \mathbf{c}_i D^i \in F^n(D), \tag{6.5}$$

where $F^m(D)$ is the m-dimensional vector space over $F(D)$. Then, the encoder process can be written as

$$\mathbf{c}(D) = \mathbf{u}(D)\mathbf{G}(D). \tag{6.6}$$

Given a convolutional code $\mathcal{C}$ encoded with an encoder characterized by the generator matrix $\mathbf{G}(D)$ of size $k \times (k+1)$, we are seeking to find the parity-check matrix $\mathbf{H}(D)$ of this generator matrix in order to derive the reciprocal parity-check matrix $\widetilde{\mathbf{H}}(D)$. To do so, we first try to find a *systematic* version of $\mathbf{G}(D)$, denoted by $\mathbf{G}_{\text{sys}}(D)$, and then, the parity check matrix can be directly derived from $\mathbf{G}_{\text{sys}}(D)$.

6.2.1.1 Generator matrix equivalence

Two convolutional codes are said to be *equivalent* if they generate the same codeword space $\mathcal{C}$. Two convolutional encoders are *equivalent* if their generator matrices are equivalent, i. e.represent equivalent convolutional codes.

Theorem 6.1. *Two rate k/n convolutional generator matrices $\mathbf{G}(D)$ and $\mathbf{G}'(D)$ are equivalent if and only if there exists a $k \times k$ non-singular matrix $\mathbf{T}(D)$ such that*

$$\mathbf{G}(D) = \mathbf{T}(D)\mathbf{G}'(D). \tag{6.7}$$

Proof. If (6.7) holds, then for any input $\mathbf{u}(D) \in F^k(D)$, we have

$$\begin{aligned} \mathbf{u}(D)\mathbf{G}(D) &= \mathbf{u}(D)\mathbf{T}(D)\mathbf{G}'(D) && (6.8)\\ &= \mathbf{u}'(D)\mathbf{G}'(D), && (6.9) \end{aligned}$$

where $\mathbf{u}'(D) = \mathbf{u}(D)\mathbf{T}(D)$. Therefore, $\mathbf{G}(D)$ and $\mathbf{G}'(D)$ are equivalent.

Conversely, suppose that $\mathbf{G}(D)$ and $\mathbf{G}'(D)$ are equivalent. Then, there exists an input $\mathbf{u}_i(D) \in F^k(D)$ such that

$$\mathbf{g}_i(D) = \mathbf{u}_i(D)\mathbf{G}'(D), \quad i = 0, \ldots, k-1 \tag{6.10}$$

where $\mathbf{g}_i(D) \in F^n(D)$ is the ith row of $\mathbf{G}(D)$. Then, by denoting

$$\mathbf{T}(D) = \begin{pmatrix} \mathbf{u}_0(D) \\ \mathbf{u}_1(D) \\ \vdots \\ \mathbf{u}_{k-1}(D) \end{pmatrix}, \tag{6.11}$$

we have

$$\mathbf{G}(D) = \mathbf{T}(D)\mathbf{G}'(D). \tag{6.12}$$

Since $\mathbf{G}(D)$ and $\mathbf{G}'(D)$ are generator matrix, they have rank k. Therefore, $\mathbf{T}(D)$ also has rank k and, hence, is non-singular. □

6.2.1.2 Equivalent systematic encoders

Let $\mathbf{T}(D)$ be a $k \times k$ non-singular sub-matrix of $\mathbf{G}(D)$, where $\mathbf{G}(D)$ is a non-systematic generator matrix of rate-$k/(k+1)$ convolutional code $\mathcal{C}$. Without loss of generality, we

assume that $\mathbf{T}(D)$ consists of the k left-most columns of $\mathbf{G}(D)$. Then, the systematic generator matrix can be obtained as

$$\mathbf{G}_{\text{sys}}(D) = \mathbf{T}^{-1}(D)\mathbf{G}(D) = \begin{pmatrix}\mathbf{I}_k & \mathbf{z}(D)\end{pmatrix}, \tag{6.13}$$

where $\mathbf{I}_k$ is the $k\times k$ identity matrix and $\mathbf{z}(\mathbf{D})$ is a $k\times 1$ column vector, which is the product of the matrix $\mathbf{T}^{-1}(D)$ with the last column of $\mathbf{G}(D)$. Since $\mathbf{T}^{-1}$ is also non-singular, according to the Theorem 6.1, $\mathbf{G}_{\text{sys}}(D)$ is equivalent to $\mathbf{G}(D)$. Furthermore, the elements of $\mathbf{z}(D)$ may be rational polynomials with the same denominator; hence, the resulting systematic encoder may become *recursive*.

Example 6.1. Consider a rate $r = 2/3$ non-systematic convolutional encoder with generator matrix

$$\mathbf{G}(D) = \begin{pmatrix}1{+}D^2 & 1 & 1{+}D{+}D^2 \\ D & 1{+}D & 1\end{pmatrix}. \tag{6.14}$$

Let $\mathbf{T}(D)$ be the matrix consisting of the first two columns of $\mathbf{G}(D)$ as

$$\mathbf{T}(D) = \begin{pmatrix}1{+}D^2 & 1 \\ D & 1{+}D\end{pmatrix}. \tag{6.15}$$

Since $\mathbf{T}(D)$ is full rank, the determinant of $\mathbf{T}(D)$ is non zero and $\det(\mathbf{T}(D)) = 1{+}D^2{+}D^3$. Therefore, the inverse of $\mathbf{T}(D)$ is

$$\mathbf{T}^{-1}(D) = \frac{1}{1{+}D^2{+}D^3}\begin{pmatrix}1{+}D & 1 \\ D & 1{+}D^2\end{pmatrix}. \tag{6.16}$$

By multiplying $\mathbf{G}(D)$ with $\mathbf{T}(D)$, we obtain the systematic encoding matrix $\mathbf{G}_{\text{sys}}(D)$ equivalent to $\mathbf{G}(D)$ as

$$\begin{aligned}\mathbf{G}_{\text{sys}}(D) &= \frac{1}{1{+}D^2{+}D^3}\begin{pmatrix}1{+}D & 1 \\ D & 1{+}D^2\end{pmatrix}\begin{pmatrix}1{+}D^2 & 1 & 1{+}D{+}D^2 \\ D & 1{+}D & 1\end{pmatrix} \\ &= \begin{pmatrix}1 & 0 & \dfrac{D^3}{1{+}D^2{+}D^3} \\ 0 & 1 & \dfrac{1{+}D{+}D^3}{1{+}D^2{+}D^3}\end{pmatrix}.\end{aligned} \tag{6.17}$$

In this case, the systematic encoder is also recursive.

6.2.1.3 Parity-check matrix and reciprocal parity-check matrix

A $(n-k)\times n$ polynomial matrix $\mathbf{H}(D)$ is the *parity-check* matrix corresponding to the $k\times n$ generator matrix $\mathbf{G}(D)$ if

$$\mathbf{G}(D)\mathbf{H}^{\top}(D) = \mathbf{0}, \tag{6.18}$$

where $^{\top}$ is the matrix transpose operator.

In the case of a $k\times(k{+}1)$ systematic generator matrix $\mathbf{G}_{\text{sys}}(D)$, we can rewrite it as

$$\mathbf{G}_{\text{sys}}(D) = \begin{pmatrix}\mathbf{I}_k & \mathbf{z}(D)\end{pmatrix} = \begin{pmatrix}\mathbf{I}_k & \dfrac{1}{q(D)}\mathbf{p}(D),\end{pmatrix} \tag{6.19}$$

where $\mathbf{p}(D)$ is the column vector consisting of the numerators of $\mathbf{z}(D)$, and $q(D)$ is the common denominator of the elements of $\mathbf{z}(D)$. Then, the $1\times(k+1)$ parity-check matrix $\mathbf{H}(D)$ can be obtained as

$$\mathbf{H}(D) = \begin{pmatrix} \mathbf{p}^\top(D) & q(D) \end{pmatrix}. \tag{6.20}$$

We can verify that

$$\mathbf{G}_{\text{sys}}(D)\mathbf{H}^\top(D) = \begin{pmatrix} \mathbf{I}_k & \dfrac{1}{q(D)}\mathbf{p}(D) \end{pmatrix} \begin{pmatrix} \mathbf{p}(D) \\ q(D) \end{pmatrix} = \begin{pmatrix} \mathbf{I_k p}(D) + \mathbf{p}(D) \end{pmatrix} = \mathbf{0}. \tag{6.21}$$

Given a convolutional code $\mathcal{C}$ generated by the encoder with the $k\times n$ generator matrix $\mathbf{G}(D)$, the *dual code* $\mathcal{C}^\perp$ of $\mathcal{C}$ is defined as the set of codewords $\mathbf{c}^\perp$ such that

$$\mathbf{c}(\mathbf{c}^\perp)^\top = 0, \quad \forall \mathbf{c} \in \mathcal{C}, \mathbf{c}^\perp \in \mathcal{C}^\perp. \tag{6.22}$$

It is important to note that the dual code $\mathcal{C}^\perp$ can be generated using the $(n-k)\times n$ generator matrix $\widetilde{\mathbf{H}}(D)$, where $\widetilde{\mathbf{H}}(D)$ is the *reciprocal* of the parity-check matrix $\mathbf{H}(D)$ [82]. The generator matrix $\widetilde{\mathbf{H}}(D)$ is often referred to as the *reciprocal parity-check* matrix. The procedure for obtaining the reciprocal parity-check matrix $\widetilde{\mathbf{H}}(D)$ from the parity-check matrix $\mathbf{H}(D)$ is as follows.

Let λ_i be the constraint length of the ith row $\mathbf{h}_i(D)$ of $\mathbf{H}(D)$, $i = 0, \ldots, (n-k)$. Then, the ith row of $\widetilde{\mathbf{H}}(D)$, denoted by $\widetilde{\mathbf{h}}_i(D)$ is

$$\widetilde{\mathbf{h}}_i(D) = D^{\lambda_i}\mathbf{h}_i(D^{-1}). \tag{6.23}$$

Example 6.2. Given the systematic generator matrix obtained from Example 6.1

$$\mathbf{G}_{\text{sys}}(D) = \begin{pmatrix} 1 & 0 & \dfrac{D^3}{1+D^2+D^3} \\ 0 & 1 & \dfrac{1+D+D^3}{1+D^2+D^3} \end{pmatrix}, \tag{6.24}$$

the parity-check matrix can be derived according to (6.20) as

$$\mathbf{H}(D) = \begin{pmatrix} D^3 & 1+D+D^3 & 1+D^2+D^3 \end{pmatrix}. \tag{6.25}$$

Then, by using (6.23), the reciprocal parity-check matrix is

$$\begin{aligned} \widetilde{\mathbf{H}}(D) &= D^3\mathbf{H}(D^{-1}) \\ &= \begin{pmatrix} 1 & 1+D^2+D^3 & 1+D+D^3 \end{pmatrix} \end{aligned} \tag{6.26}$$

Fig. 6.1 shows the original trellis constructed from $\mathbf{G_{sys}}(D)$ and the dual trellis constructed from $\widetilde{\mathbf{H}}(D)$. We can observe that a section of the original trellis has the form of radix-4, with 4 branches entering and leaving each state. Meanwhile, a section of the dual trellis has the form of radix-2, with only half the number of branches. This can be further generalized to $k > 2$: the original trellis of a rate $k/(k+1)$ code has the form of radix-2^k; hence, its complexity increases exponentially with k. The dual trellis, on the other hand, always has the form of radix-2 and is much less complex than the original one. Therefore, for values of k larger than 2, it seems attractive to resort to the dual trellis for decoding high-rate convolutional codes.

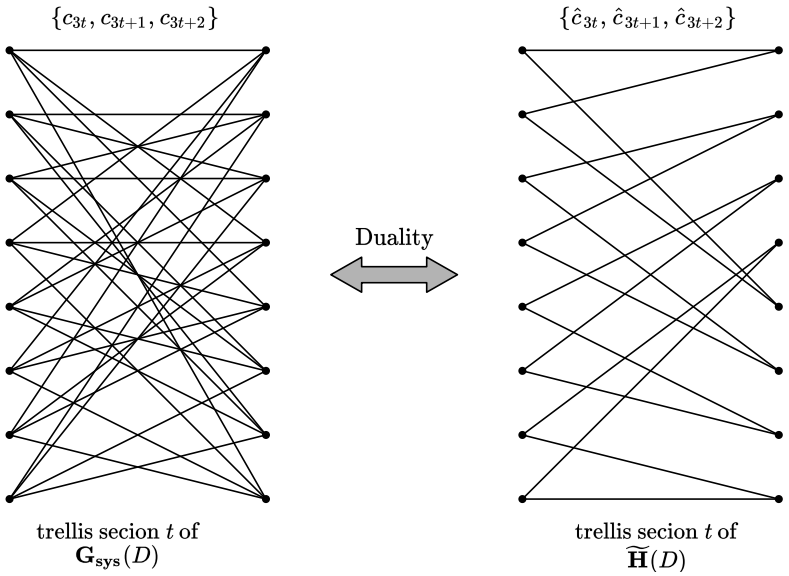


Figure 6.1: Comparison between a section of the original trellis generated from $\mathbf{G}_{\mathbf{sys}}(D)$ and a section of the dual trellis generated from $\widetilde{\mathbf{H}}(D)$ (Example 6.2).

6.2.1.4 Decoding with the dual trellis

In this section, we take a retrospective look at the decoding process of a convolutional code and we make the connection with the dual code.

Recalling from Section 2.3.3, a systematic convolutional encoder with rate k/n takes input $\mathbf{u} = (u_0, \ldots, u_{K-1})$ and produces codeword $\mathbf{c} = (c_0, \ldots, c_{N-1})$. Then, we assume that the codeword is transmitted over an AWGN channel using BPSK modulation. At the receiver side, the LLR is computed for each bit c_j, using the received vector $\mathbf{y} = (y_0, \ldots, y_{N-1})$:

$$\mathrm{L}^I(c_j; y_j) = \ln \frac{\Pr\{c_j = 1; y_j\}}{\Pr\{c_j = 0; y_j\}} \tag{6.27}$$

$$= \ln \frac{\Pr\{y_j | c_j = 1\}}{\Pr\{y_j | c_j = 0\}} + \ln \frac{\Pr\{c_j = 1\}}{\Pr\{c_j = 0\}} \tag{6.28}$$

$$= \mathrm{L}^c(y_j | c_j) + \mathrm{L}^a(c_j), \tag{6.29}$$

where $\mathrm{L}^c(y_j|c_j)$ is the channel LLR, obtained from the demodulator after the zero-mean AWGN channel with variance σ^2, and is calculated according to Eq. (2.4) as

$$\mathrm{L}^c(y_j | c_j) = \frac{2y_k}{\sigma^2}. \tag{6.30}$$

Furthermore, $\mathrm{L}^a(c_k)$ is the *a priori* LLR, which is zero with the assumption that $c_j \in \{0, 1\}$ is equiprobable. However, in the context of conventional turbo decoding, then the *a priori* LLR of the systematic bits might be non zero. Therefore, the previous LLR is derived as

$$\mathrm{L}^I(c_j; y_j) = \begin{cases} \mathrm{L}^c(y_j|c_j) + \mathrm{L}^a(c_j), & \text{if } c_j \text{ is an systematic bit} \\ \mathrm{L}^c(y_j|c_j), & \text{if } c_j \text{ is a parity bit} \end{cases} \tag{6.31}$$

The soft output of the decoder is the *a posteriori* LLR of the jth decoded bit $\hat{c}_j$, which can be estimated via the original code $\mathcal{C}$ by taking into account the codewords

$\mathbf{c} \in \mathcal{C}$ having $c_j = 1$ and the codewords $\mathbf{c} \in \mathcal{C}$ having $c_j = 0$, respectively. Therefore, for each decoded systematic bit $\hat{c}_j$, the output LLR is computed as

$$\begin{aligned}
\mathrm{L}(\hat{c}_j) &= \ln \frac{\sum\limits_{\mathbf{c}\in\mathcal{C}|c_j=1} \prod\limits_{l=0}^{N-1} \exp\left(\mathrm{L}^I(c_l;y_l)\times c_l\right)}{\sum\limits_{\mathbf{c}\in\mathcal{C}|c_j=0} \prod\limits_{l=0}^{N-1} \exp\left(\mathrm{L}^I(c_l;y_l)\times c_l\right)} \\
&= \ln \frac{\sum\limits_{\mathbf{c}\in\mathcal{C}|c_j=1} \exp\left(\mathrm{L}^I(c_j;y_j)\times c_j\right) \prod\limits_{l=0,l\neq j}^{N-1} \exp\left(\mathrm{L}^I(c_l;y_l)\times c_l\right)}{\sum\limits_{\mathbf{c}\in\mathcal{C}|c_j=0} \exp\left(\mathrm{L}^I(c_j;y_j)\times c_j\right) \prod\limits_{l=0,l\neq j}^{N-1} \exp\left(\mathrm{L}^I(c_l;y_l)\times c_l\right)} \\
&= \mathrm{L}^I(c_j;y_j) + \ln \frac{\sum\limits_{\mathbf{c}\in\mathcal{C}|c_j=1} \prod\limits_{l=0,l\neq j}^{N-1} \exp\left(\mathrm{L}^I(c_l;y_l)\times c_l\right)}{\sum\limits_{\mathbf{c}\in\mathcal{C}|c_j=0} \prod\limits_{l=0,l\neq j}^{N-1} \exp\left(\mathrm{L}^I(c_l;y_l)\times c_l\right)},
\end{aligned} \tag{6.32}$$

where the second term from (6.32) is the *extrinsic* LLR provided by the decoder, which has already been introduced in Chapter 2.

Alternatively, the value of $\mathrm{L}(\hat{u}_j)$ can also be calculated using the dual code $\mathcal{C}^\perp$ of $\mathcal{C}$ as [80]

$$\mathrm{L}(\hat{c}_j) = \mathrm{L}^I(c_j;y_j) + \ln \frac{\sum\limits_{\mathbf{c}^\perp\in\mathcal{C}^\perp} \left(-1\right)^{c_j^\perp} \prod\limits_{l=0,l\neq j}^{N-1} \tanh\left(\mathrm{L}^I(c_l;y_l)/2\right)^{c_l^\perp}}{\sum\limits_{\mathbf{c}^\perp\in\mathcal{C}^\perp} \prod\limits_{l=0,l\neq j}^{N-1} \tanh\left(\mathrm{L}^I(c_l;y_l)/2\right)^{c_l^\perp}}, \tag{6.33}$$

where $\mathbf{c}^\perp = (c_0^\perp, \ldots, c_{N-1}^\perp)$ denotes codewords of $\mathcal{C}^\perp$. By comparing (6.32) and (6.33), we can see that both equations perform a sum over all codewords in their respective set. However, the dimension of $\mathcal{C}$ is k, and the dimension of $\mathcal{C}^\perp$ is $(n-k)$. Therefore, for high code rates such that $n-k<k$, i.e. $r>1/2$, $\mathcal{C}^\perp$ has less codewords than $\mathcal{C}$.

Furthermore, we have seen in Chapter 2 that (6.32) can be obtained using the BCJR or the log-MAP algorithm, which operates on the trellis of the original encoding matrix $\mathbf{G}(D)$. Equivalently, (6.33) could also be implemented using the BCJR algorithm or its logarithmic version on the trellis of the reciprocal parity-check matrix $\widetilde{\mathbf{H}}(D)$, since $\widetilde{\mathbf{H}}(D)$ generates the codewords of the dual code $\mathcal{C}^\perp$. This trellis is referred to as the *dual trellis*, and the decoding algorithms operating on the dual trellis are referred to as the *dual-MAP* and *dual-Log-MAP*algorithms, corresponding to their counterparts on the original trellis.

6.2.2 The Dual-MAP and Dual-Log-MAP algorithms

6.2.2.1 The dual-MAP algorithm

In the original trellis, the BCJR algorithm consists of the following steps: branch metric calculation, forward recursion, backward recursion and soft-output calculation. Equivalently, the same steps can be applied into the dual trellis resulting in the dual-MAP algorithm.

Given the received vector $\mathbf{y}$ of length N as the modulated and noisy version of the codeword $\mathbf{c}$, encoded by a rate k/n convolutional code with input message of length K, the intrinsic LLRs $\{\mathrm{L}^I(c_j; y_j)\}_{0\leq j<N}$ are calculated as in (6.31). According to (6.33), the first step towards the decoding of the convolutional codes using the dual trellis is to convert the intrinsic LLRs into *bit metrics* $\{d_j\}_{0\leq j<N}$ as

$$d_j = \tanh\big(\mathrm{L}^I(c_j; y_j)/2\big). \tag{6.34}$$

With the dual trellis, the received frame of length N is decomposed into $T = N/n$ trellis sections. Then, the bit metrics related to a specific section t, $0 \leq t < T$, are $\{d_{nt+1}, \ldots, d_{nt+n}\}$. Let (s_t, s_{t+1}) denote a branch between state s_t at instant t and s_{t+1} at instant t+1, and let $\{c^{\perp}_{nt+1}(s_t, s_{t+1}), \ldots, c^{\perp}_{nt+n}(s_t, s_{t+1})\} \in \{0,1\}^n$ be the bit decisions carried by the branch (s_t, s_{t+1}).

In this section, we reuse the notation of the metrics in the original trellis for the dual trellis. Therefore, $\gamma_t(s_t, st{+}1)$ is the metric of branch (s_t, s_{t+1}), $\beta_t(s_t)$ is the backward state metric of state s_t at instant t, and $\alpha_{t+1}(s_{t+1})$ is the forward state metric of state s_{t+1} at instant t+1. The branch metric calculation related to trellis section t in the dual trellis yields

$$\gamma_t(s_t, s_{t+1}) = \prod_{j=nt+1}^{nt+n} \big(d_j\big)^{c_j^{\perp}(s_t, s_{t+1})}. \tag{6.35}$$

Then, the forward and backward state metrics can be recursively calculated as

$$\alpha_{t+1}(s_{t+1}) = \sum_{s_t} \alpha_t(s_t)\gamma_t(s_t, s_{t+1}), \tag{6.36}$$

$$\beta_t(s_t) = \sum_{s_{t+1}} \beta_{t+1}(s_{t_+1})\gamma_t(s_t, s_{t+1}). \tag{6.37}$$

Note that, as in the original trellis, the forward state metric recursion in the dual trellis also starts at instant $t = 0$ and moves up to $t = N{+}1$. Conversely, the backward recursion starts at instant $t = N{+}1$ and the backward state metrics are then calculated recursively down to $t = 0$. However, the initialization of the state metrics is different from the initialization in the original trellis.

Recall that, in the original trellis, there are two widely employed methods to terminate the trellis: force-to-zero and tailbiting. With the former termination method, the forward state metrics at the trellis start and the backward state metrics at the trellis end are initialized according to an *impulse* distribution $\delta(s, 0)$, where

$$\delta(s, i) = \begin{cases} 1, & \text{if the state } s = i, \\ 0, & \text{otherwise.} \end{cases} \tag{6.38}$$

For the tailbiting convolutional codes in the original trellis, the forward state metrics and the backward state metrics are initialized according to a *uniform* distribution with equal values for all states s.

The state metrics in the original trellis and the dual trellis exhibit a Fourier transform relation [83]. Therefore, an impulse (resp. uniform) state metric distribution in the original trellis corresponds to a uniform (resp. impulse) state metric distribution in the dual trellis. Thus, if force-to-zero is used in the original trellis, the initialization of the state metrics in the dual trellis is

$$\alpha_0(s) = 1, \forall s, \tag{6.39}$$

$$\beta_{N+1}(s) = 1, \forall s. \tag{6.40}$$

Conversely, if tailbiting is used in the original trellis, the initialization of the state metrics in the dual trellis is

$$\alpha_0(s) = \delta(s, 0), \tag{6.41}$$

$$\beta_{N+1}(s) = \delta(s, 0). \tag{6.42}$$

For trellis section t, after having computed $\gamma_t(s_t, s_{t+1})$, $\alpha_t(s_t)$, and $\beta_{t+1}(s_{t+1})$, the *a posteriori* likelihood ratio for each bit $c_j^\perp$ of the dual code, denoted by z_j, $j \in \{nt+1, \ldots, nt+n\}$ is computed as

$$z_j = \frac{q_j^1}{q_j^0} = \frac{\sum\limits_{(s_t, s_{t+1})|c_j^\perp(s_t, s_{t+1})=1} \alpha_t(s_t)\gamma_t(s_t, s_{t+1})\beta_{t+1}(s_{t+1})}{\sum\limits_{(s_t, s_{t+1})|c_j^\perp(s_t, s_{t+1})=0} \alpha_t(s_t)\gamma_t(s_t, s_{t+1})\beta_{t+1}(s_{t+1})}, \tag{6.43}$$

where q_j^1 and q_j^0 are the summations of all the metrics of codeword $\mathbf{c}^\perp$ in $\mathcal{C}^\perp$ having $c_j^\perp = 1$ and $c_j^\perp = 0$, respectively. By applying q_j^0 and q_j^1 into the sums in the numerator and denominator of (6.33), the *a prosteriori* LLR of the decoded bit $\hat{c}_j$ in the original trellis is obtained as

$$\begin{aligned} \mathrm{L}(\hat{c}_j) &= \mathrm{L}^I(c_j; y_j) + \ln \frac{q_j^0 - q_j^1/d_j}{q_j^0 + q_j^1/d_j} \\ &= \mathrm{L}^I(c_j; y_j) + \ln \frac{1 - z_j/d_j}{1 + z_j/d_j} \\ &= \mathrm{L}^I(c_j; y_j) + \ln \frac{1 - u_j}{1 + u_j}, \end{aligned} \tag{6.44}$$

where $u_j = z_j/d_j$ is the extrinsic information in the dual domain.

Equation (6.44) shows that one can completely derive the soft output related to decoded bit c_j using the dual trellis. However, similar to the BCJR algorithm, the dual-MAP algorithm requires a large number of multiplications and divisions and is therefore too complex, taken as it is, for practical hardware implementations. Consequently, to make the decoding on the dual trellis feasible, the application of the dual-MAP algorithm in the logarithmic domain was investigated in [84], which we call the dual-Log-MAP algorithm.

6.2.2.2 The dual-Log-MAP algorithm with sign-magnitude

The input bit metric of the dual-MAP decoder expressed in (6.34) can take positive or negative values. Therefore, one can resort to the *sign-magnitude* (SM) representation as in [84] to represent it in the logarithmic domain. Using the SM representation, a real number x is expressed as

$$x = (-1)^{X_s} \exp(X_m) \triangleq [X_s; X_m], \tag{6.45}$$

where $X_s \in \{0, 1\}$ and $X_m = \log(|x|)$ are the sign and the magnitude of x, respectively. Then, the arithmetic operations involved in the decoding algorithm can be expressed as follows, using the SM representation:

- Sign-magnitude multiplication (SMM):

$$xy \triangleq [X_s \oplus Y_s; X_m + Y_m]. \tag{6.46}$$

- Sign-magnitude division (SMD):

$$x/y \triangleq [X_s \oplus Y_s; X_m - Y_m]. \tag{6.47}$$

- Sign-magnitude addition (SMA):

$$x + y \triangleq \begin{cases} \left[X_s; X_m - \log\left(1 + (-1)^{X_s+Y_s} \exp(Y_m - X_m)\right)\right], & \text{if } X_m > Y_m, \\ \left[Y_s; Y_m - \log\left(1 + (-1)^{X_s+Y_s} \exp(X_m - Y_m)\right)\right], & \text{otherwise.} \end{cases} \tag{6.48}$$

With this representation, the multiplication and division operators used by the dual-MAP decoder are replaced by SMM and SMD operators.

Furthermore, from (6.44), the extrinsic information in the original trellis can be obtained as [85]

$$\ln \frac{1 - (-1)^{U_{j,s}} \exp(U_{j,m})}{1 + (-1)^{U_{j,s}} \exp(U_{j,m})} = \begin{cases} \ln \tanh\left(\left|\frac{U_{j,m}}{2}\right|\right), & \text{if } U_{j,s} = 0 \\ -\ln \tanh\left(\left|\frac{U_{j,m}}{2}\right|\right), & \text{otherwise} \end{cases}, \tag{6.49}$$

where $U_{j,s}$ and $U_{j,m}$ are the sign and the magnitude of the extrinsic information in the dual domain u_j, respectively.

When considering the decoding algorithm in the logarithmic domain, the dual-Log-MAP (dual-LM) decoder is more hardware friendly than the dual-MAP decoder: The multiplications and divisions are now replaced by adders and subtractors and the SMA can be implemented with a compare-select operator and two LUTs, while the extrinsic information conversion can be implemented with one LUT for the logarithm of hyperbolic tangent function.

In [85], two hardware implementations of a dual-LM decoder and a conventional radix-4 MLM decoder are reported and compared in terms of throughput and circuit area, for turbo codes using rate $k/(k+1)$ component convolutional codes with $k = 2, 4, 8, 16$. Figure 6.2 taken from [85], shows the comparison of the implementation results. Note that the turbo decoder used in [85] is a PMAP architecture with two parallel SISO decoders using forward-backward scheduling, sliding windows and ACQ initialization.

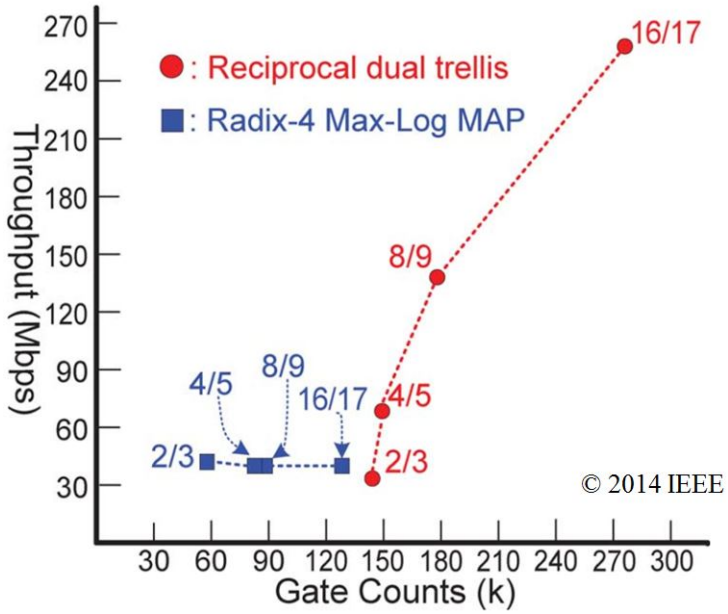


Figure 6.2: Implementation comparison between dual-LM and radix-4 MLM decoders [85].

From Figure 6.2, we can see that for a medium code rate such as $r = 2/3$, both algorithms yield the same throughput but the circuit area of the dual-MAP decoder is more than twice compared to the radix-4 MLM decoder. However, for higher code rates such as $r = 4/5$ or $r = 8/9$, the throughput of the MLM decoder remains unchanged while the circuit area increases with the code rate. This is due to the need of longer decoding windows and/or long acquisition sequences for state metric initialization to avoid any correction performance loss. On the contrary, the throughput of the dual-LM decoder is doubled from $r = 2/3$ to $r = 4/5$ and from $r = 4/5$ to $r = 8/9$, while its circuit area only increases from 140 kgates at $r = 2/3$ to 180 kgates at $r = 8/9$. This can be explained by the fact that, for a code rate $k/(k+1)$, the dual-LM decoder is able to process simultaneously k bits while the conventional radix-4 MLM decoder can only process 2 bits at each trellis stage. These are the main advantages of decoding using the dual trellis instead of the original trellis for high-rate convolutional codes.

6.3 Constructing the Dual Trellis for Punctured Convolutional Codes

In the previous section, we explained that the dual-LM algorithm employed on the dual trellis had a better area efficiency than the MLM in the original trellis, when considering true high-rate convolutional codes with rate $k/(k+1)$, for $k > 4$. With that motivation, this section proposes a fully generic procedure to construct the dual trellis for *high-rate punctured convolutional codes*, so that they can also be decoded using the dual-LM algorithm. This method is based on transformations proposed by Forney in [86].

The generic procedure is presented in Figure 6.3. The main idea of the procedure is that, given a mother code (e.g. $r = 1/2$) and a puncturing pattern to achieve code rate k/n, we first find the non-systematic generator matrix $\mathbf{G_{ns}}(D)$ of size $k \times n$ *equivalent* to the original punctured convolutional code of rate k/n. Then, parity-check matrix $\mathbf{H}(D)$

and its reciprocal $\widetilde{\mathbf{H}}(D)$ can be derived and the dual trellis is constructed directly from matrix $\widetilde{\mathbf{H}}(D)$. The dual-LM algorithm then decodes the received codeword based on the dual trellis yielding the soft estimate on the punctured codeword.

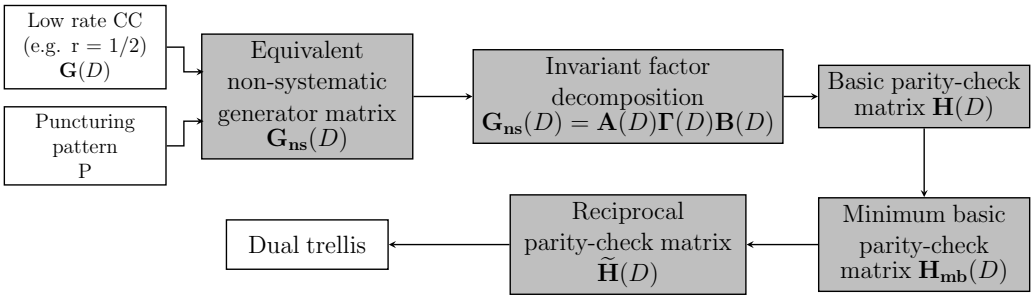


Figure 6.3: Procedure for obtaining the reciprocal dual trellis of a high-rate punctured convolutional code.

6.3.1 Equivalent Non-Systematic Generator Matrix of Punctured Convolutional Codes

The first step towards the reciprocal dual trellis involves converting the punctured convolutional code into a non-systematic encoding form, as described in [87]. This transformation has previously been used to assess the performance of convolutional codes, since all equivalent forms share the same Hamming weight spectrum.

In order to get a punctured convolutional code k/n, we start from the following general form of an generator matrix of a mother code with rate $1/m$ as

$$\mathbf{G}(D) = \begin{pmatrix} g_0(D) & g_1(D) & \dots & g_{m-1}(D) \end{pmatrix} \tag{6.50}$$

where $g_i(D)$, $i = 0, \dots, m-1$ are the polynomials of the generator matrix. This convolutional code can also be viewed as a rate-$k/(km)$ code, for any value of k. Therefore, we can derive a $k \times (km)$ generator matrix $\mathbf{G}'(D)$ that is equivalent to the generator matrix of the mother code. This $k \times (km)$ matrix can be defined by km polynomials obtained by splitting each polynomial $g_i(D)$, $i = 0, \dots, m-1$ into k sub-polynomials $f_{i,j}(D)$, $j = 0, \dots, k-1$ as follows

$$g_i(D) = \sum_{j=0}^{k-1} D^j f_{i,j}(D^k). \tag{6.51}$$

The element of the resulting $k \times (km)$ generator matrix $\mathbf{G}'(D)$ at row p and column q, $g'_{p,q}(D)$, is defined as:

$$\begin{cases} g'_{p,q}(D) = f_{q \bmod n, \lfloor q/m \rfloor - p}(D), & \text{if } p \times m \leq q \\ g'_{p,q}(D) = D \times f_{q \bmod n, \lfloor q/m \rfloor - p + k}(D), & \text{if } p \times m > q \end{cases} \tag{6.52}$$

where mod is the modulo operation. The generator matrix of the rate-k/n code is then obtained by selecting n out of (km) columns in $\mathbf{G}'(D)$ according to a selected puncturing pattern.

Example 6.3. Given the generator matrix of a rate 1/2 mother code as $\mathbf{G}(D) = (g_0(D), g_1(D))$, the procedure to obtain an equivalent 3×6 generator matrix is as follows. With $m = 2$ and $k = 3$, two polynomials $g_0(D)$ and $g_1(D)$ can be each decomposed into three sub-polynomials: $\{f_{0,j}(D)\}_{j=0,1,2}$ and $\{f_{1,j}(D)\}_{j=0,1,2}$, respectively. The generator matrix $\mathbf{G}'(D)$ of the equivalent rate-3/6 code is

$$\mathbf{G}'(D) = \begin{pmatrix} f_{0,0}(D) & f_{1,0}(D) & f_{0,1}(D) & f_{1,1}(D) & f_{0,2}(D) & f_{1,2}(D) \\ D\times f_{0,2}(D) & D\times f_{1,2}(D) & f_{0,0}(D) & f_{1,0}(D) & f_{0,1}(D) & f_{1,1}(D) \\ D\times f_{0,1}(D) & D\times g_{1,1}(D) & D\times f_{0,2}(D) & D\times f_{1,2}(D) & f_{0,0}(D) & f_{1,0}(D) \end{pmatrix}. \tag{6.53}$$

The non-systematic punctured generator matrix $\mathbf{G_{ns}}(D)$ of the rate-k/n code is then obtained by selecting n columns out of kN in $\mathbf{G}'(D)$, according to a selected puncturing pattern. For instance, if columns $1, 2, 3$ and 5 in $\mathbf{G}'(D)$ are selected, the following matrix generates a rate-3/4 code

$$\mathbf{G_{ns}}(D) = \begin{pmatrix} f_{0,0}(D) & f_{1,0}(D) & f_{0,1}(D) & f_{0,2}(D) \\ D\times f_{0,2}(D) & D\times f_{1,2}(D) & f_{0,0}(D) & f_{0,1}(D) \\ D\times f_{0,1}(D) & D\times f_{1,1}(D) & D\times f_{0,2}(D) & f_{0,0}(D) \end{pmatrix}. \tag{6.54}$$

6.3.2 Reciprocal Parity-Check Matrix of Punctured Convolutional Codes

6.3.2.1 Invariant-factor decomposition of the generator matrix

Let us start with the definition of a basic encoding matrix. An encoding matrix $\mathbf{G}(D)$ is called *basic* if it has a right inverse $\mathbf{G}^{-1}(D)$. The right inverse existence of an encoding matrix ensures that the encoder is a bijective function. Therefore, given any two different inputs, the basic encoder always yields two different codewords.

Based on the invariant-factor theorem, as in [86], or on the Smith form, as in [88], the invariant-factor decomposition of the generator matrix $\mathbf{G}(D)$ of a convolutional code with rate k/n gives

$$\mathbf{G}(D) = \mathbf{A}(D)\mathbf{\Gamma}(D)\mathbf{B}(D), \tag{6.55}$$

where $\mathbf{A}(D)$ is a $k \times k$ polynomial matrix with unit determinant; $\mathbf{B}(D)$ is a $n \times n$ polynomial matrix with unit determinant, thus having an polynomial matrix inverse $\mathbf{B}^{-1}(D)$; and $\mathbf{\Gamma}(D)$ is a $k \times n$ diagonal matrix, whose elements are called the invariant factors of $\mathbf{G}(D)$ and are unique. In fact, matrix $\mathbf{\Gamma}(D)$ is obtained by permuting and linearly combining the rows and the columns of matrix $\mathbf{G}(D)$. Since the permutation or linear combination of rows (or columns) can be represented by the pre- (or post-) multiplication of a square matrix with unit determinant by $\mathbf{G}(D)$, $\mathbf{A}(D)$ is the result of all row operations and matrix $\mathbf{B}(D)$ is the result of all column operations.

As pointed out in [86], matrix $\mathbf{U}(D)$ consisting of the first k rows of $\mathbf{B}(D)$ is a basic encoding matrix equivalent to $\mathbf{G}(D)$. Since the polynomial matrix $\mathbf{B}(D)$ has a polynomial inverse $\mathbf{B}^{-1}(D)$, the matrix consisting of the last $(n - k)$ columns of $\mathbf{B}^{-1}(D)$ is the transpose of the parity-check matrix, i.e., $\mathbf{H}^\top(D)$. This can be explained by the fact that $\mathbf{B}(D)\mathbf{B}^{-1}(D) = \mathbf{I}_n$ where $\mathbf{I}_n$ is the $n \times n$ identity matrix. Then, the inner product between row $i \in \{0, \ldots, k-1\}$ of $\mathbf{B}(D)$ and column $j \in \{n-k, \ldots, n-1\}$ of $\mathbf{B}^{-1}(D)$ is the element at row i and column j of the identity matrix $\mathbf{I}_n$. Since $(\mathbf{I}_n)_{i,j} = 0$ for $j > i$, $\mathbf{U}(D)\mathbf{H}^\top(D) = \mathbf{0}$ and consequently, $\mathbf{G}(D)\mathbf{H}^\top(D) = \mathbf{0}$. Note that matrix $\mathbf{H}(D)$ is basic

since it has a right inverse which is the transpose of the matrix consisting of the last $(n-k)$ rows of $\mathbf{B}(D)$.

6.3.2.2 Minimal basic parity-check matrix

According to Theorem 7 in [86], if $\mathbf{G}(D)$ is a basic encoder with overall constraint length λ, then there exists an associated basic parity-check matrix $\mathbf{H}(D)$ with overall constraint length λ. Recall from (6.3) that the overall constraint length of a polynomial matrix is the sum of the constraint lengths for all rows.

However, we observed that, in the previous decomposition, the parity-check matrix $\mathbf{H}(D)$ consisting of the last $(n-k)$ columns of $\mathbf{B}^{-1}(D)$ usually has an overall constraint length greater than λ, the overall constraint length of $\mathbf{G}(D)$. Therefore, we need to find the minimal basic parity-check matrix $\mathbf{H_{mb}}(D)$ with overall constraint length λ that is equivalent to $\mathbf{H}(D)$. To this end, the authors in [88] proposed an algorithm for finding the minimal basic form of an encoding matrix, called *Algorithm MB*. The main idea of this algorithm is to lower gradually the constraint length of the parity-check matrix by linearly combining multiple rows. Then, after a number of steps, its overall constraint length becomes equal to λ. Since the operation of linearly combining matrix rows can be represented by the pre-multiplication of a unit determinant matrix by $\mathbf{H}(D)$, the resulting minimal basic parity-check matrix $\mathbf{H_{mb}}(D)$ is equivalent to $\mathbf{H}(D)$.

Assuming that $\mathbf{G}(D)$ is a $k \times n$ encoding matrix, let $[\mathbf{G}(D)]_h$ be a (0,1)-matrix with 1 in the position (i,j) if $\deg\big(g_{i,j}(D)\big) = \lambda_i$ and 0 otherwise. The algorithm MB proceeds as follows:

- **Step 1:** If $[\mathbf{G}(D)]_h$ has full rank, then $\mathbf{G}(D)$ is a minimal basic encoding matrix and the algorithm stops; otherwise go to **Step 2**.
- **Step 2:** $[\mathbf{G}(D)]_h$ does not have full rank, meaning that there are at least two linearly dependent rows in $[\mathbf{G}(D)]_h$. Without loss of generality, we let $[\mathbf{r}_{i_1}], [\mathbf{r}_{i_2}], \ldots, [\mathbf{r}_{i_d}]$ denote the set of linearly dependent rows in $[\mathbf{G}(D)]_h$ such that $\lambda_{i_d} \geq \lambda_{i_j}$, $j = 0, \ldots, d-1$. Then, if we let $\mathbf{r}_{i_1}, \mathbf{r}_{i_2}, \ldots, \mathbf{r}_{i_d}$ be the corresponding set of rows in $\mathbf{G}(D)$, we can lower the constraint lengths by adding

  $$D^{\lambda_{i_d}-\lambda_{i_1}}\mathbf{r}_{i_1} + D^{\lambda_{i_d}-\lambda_{i_2}}\mathbf{r}_{i_2} + \ldots + D^{\lambda_{i_d}-\lambda_{i_{d-1}}}\mathbf{r}_{i_{d-1}}$$

  to the i_d-th row of $\mathbf{G}(D)$. Return to **Step 1**.

6.3.2.3 Reciprocal parity-check matrix

After having derived the minimal basic parity-check matrix $\mathbf{H_{mb}}(D)$, the reciprocal parity check matrix $\widetilde{\mathbf{H}}(D)$ can be directly obtained by

$$\widetilde{\mathbf{H}}_i(D) = D^{\lambda_i}\mathbf{H_{mb}}_i(D^{-1}),\ 0 \leq i < n-k \tag{6.56}$$

where $\widetilde{\mathbf{H}}_i(D)$ and $\mathbf{H_{mb}}_i(D)$ are respectively the ith row of $\widetilde{\mathbf{H}}(D)$ and $\mathbf{H_{mb}}(D)$.

Matrix $\widetilde{\mathbf{H}}(D)$ is the encoder generating the dual codeword $\widetilde{\mathbf{v}}$ that is orthogonal to any codeword $\mathbf{v}$ generated by the original high-rate punctured convolutional encoder. The dual-MAP algorithm can be run using the dual trellis generated by $\widetilde{\mathbf{H}}(D)$ to yield the soft estimates related to codeword $\mathbf{v}$.

To summarize, the procedure to obtain the dual trellis starting from the original high-rate punctured convolutional code can be described as follows:

- Find the equivalent high-rate non-systematic generator matrix $\mathbf{G_{ns}}(D)$;
- Perform the invariant-factor decomposition

$$\mathbf{G_{ns}}(D) = \mathbf{A}(D)\mathbf{\Gamma}(D)\mathbf{B}(D);$$

- Find the inverse matrix $\mathbf{B}^{-1}(D)$, then the parity-check matrix $\mathbf{H}(D)$ is the transpose of the matrix consisting of the last $(n-k)$ columns of $\mathbf{B}^{-1}(D)$;
- Apply Algorithm MB to derive the equivalent minimal basic parity-check matrix $\mathbf{H_{mb}}(D)$ from $\mathbf{H}(D)$;
- Deduce the reciprocal parity-check matrix $\widetilde{\mathbf{H}}(D)$ according to (6.56) and construct the dual trellis based on $\widetilde{\mathbf{H}}(D)$.

6.3.3 Example and Numerical Results

6.3.3.1 An example of dual trellis construction

In this section, we provide an example of derivation of the dual trellis from a high-rate punctured convolutional code. The considered convolutional code is the constituent RSC code of the LTE turbo code [5], punctured to achieve rate 5/7

$$\mathbf{G_{LTE}}(D) = \begin{pmatrix} 1 & \frac{1+D+D^3}{1+D^2+D^3} \end{pmatrix}; \qquad P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix},$$

thus yielding a rate-5/9 turbo code. In order to derive the dual trellis of the convolution code, we first convert the recursive systematic generator matrix into a non-recursive form

$$\begin{aligned} \mathbf{G}(D) &= (1+D^2+D^3)\begin{pmatrix} 1 & \frac{1+D+D^3}{1+D^2+D^3} \end{pmatrix} \\ &= \begin{pmatrix} 1+D^2+D^3 & 1+D+D^3 \end{pmatrix} \end{aligned}$$

and according to (6.7), $\mathbf{G_{LTE}}(D)$ and $\mathbf{G}(D)$ are equivalent.

6.3.3.2 Equivalent non-systematic generator matrix

First, we find the equivalent non-punctured rate-5/10 generator matrix. According to (6.51), we decompose $G_0(D) = 1 + D^2 + D^3$ and $G_1(D) = 1 + D + D^3$ into 5 sub-polynomials each

$$\begin{array}{lllll} g_{0,0}(D) = 1, & g_{0,1}(D) = 0, & g_{0,2}(D) = 1, & g_{0,3}(D) = 1, & g_{0,4}(D) = 0, \\ g_{1,0}(D) = 1, & g_{1,1}(D) = 1, & g_{1,2}(D) = 0, & g_{1,3}(D) = 1, & g_{1,4}(D) = 0. \end{array}$$

Then, the non-punctured rate-5/10 generator matrix $\mathbf{G}'(D)$ is

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ D & D & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ D & 0 & D & D & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & D & D & 0 & D & D & 0 & 0 & 1 & 1 \end{pmatrix}$$

The length of puncturing pattern P is 5. When applying it periodically to $\mathbf{G}(D)$, every second, third and fifth parity bits are punctured. Since these parity bits are equivalently generated by the even columns of $\mathbf{G}'(D)$, applying pattern P to $\mathbf{G}'(D)$ amounts to removing columns 4, 6 and 10 from $\mathbf{G}'(D)$. The resulting non-systematic rate-5/7 generator matrix $\mathbf{G_{ns}(D)}$ is then:

$$\mathbf{G_{ns}}(D) = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ D & D & 0 & 1 & 0 & 1 & 1 \\ D & 0 & D & 0 & 1 & 1 & 0 \\ 0 & D & D & D & 0 & 0 & 1 \end{pmatrix}$$

6.3.3.3 Reciprocal parity-check matrix

For the sake of simplicity, we do not show all the details of the invariant-factor decomposition here. Readers may refer to Example 2.4 in [88] for a detailed example. The invariant-factor decomposition of $\mathbf{G_{ns}}(D)$ results in $\mathbf{A}(D)\mathbf{\Gamma}(D)\mathbf{B}(D)$ as

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & 0 \\ D & D & D & 1 & 0 \\ 0 & D & 1{+}D & 0 & 1 \end{pmatrix}}_{\mathbf{A}(D)} \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}}_{\mathbf{\Gamma}(D)} \underbrace{\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1{+}D & D & 1{+}D & 1 \\ 0 & D & 0 & D^2 & 1{+}D^2 & 1{+}D^2 & 0 \\ 0 & D & 0 & 1{+}D{+}D^2 & D^2 & 1{+}D^2 & 0 \\ 0 & 1 & 0 & 0 & D & D & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}}_{\mathbf{B}(D)}$$

Then, the inverse matrix $\mathbf{B}^{-1}(D)$ is derived from $\mathbf{B}(D)$ by Gaussian elimination

$$\mathbf{B}^{-1}(D) = \begin{pmatrix} 1 & 0 & 0 & 1{+}D & 0 & 1{+}D{+}D^2 & 1{+}D^2{+}D^3 \\ 0 & 0 & 0 & D & 0 & 1{+}D^2 & D^3 \\ 0 & 1 & 1 & 1{+}D & 0 & D{+}D^2 & 1{+}D{+}D^2{+}D^3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1{+}D \\ 0 & 0 & 0 & 0 & 1 & D & 1{+}D{+}D^2 \\ 0 & 0 & 1 & D & 1 & D{+}D^2 & D^2{+}D^3 \end{pmatrix} \tag{6.57}$$

The code rate of the dual code is 2/7. Therefore, the parity-check matrix $\mathbf{H}(D)$ is obtained from the last 2 columns of $\mathbf{B}^{-1}(D)$ as

$$\mathbf{H}(D) = \begin{pmatrix} 1{+}D{+}D^2 & 1{+}D^2 & D{+}D^2 & 0 & 0 & D & D{+}D^2 \\ 1{+}D^2{+}D^3 & D^3 & 1{+}D{+}D^2{+}D^3 & 1 & 1{+}D & 1{+}D{+}D^2 & D^2{+}D^3 \end{pmatrix} \tag{6.58}$$

We can observe that the overall constraint length of $\mathbf{H}(D)$ in (6.58) is 5 while the overall constraint length of $\mathbf{G_{ns}}(D)$ in (6.3.3.2) is only 3. Therefore, one should apply algorithm MB to matrix $\mathbf{H}(D)$. We first derive the following matrix

$$[\mathbf{H}(D)]_h = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}. \tag{6.59}$$

Table 6.1: Parity puncturing pattern for $K = 400$

Turbo rate	Parity puncturing pattern
8/11	1100000000000010
4/5	0100000000000010
8/9	0100000000000000

Table 6.2: ARP interleaver parameters for $K = 400$

Q	P	$\Big(S(0), \ldots, S(Q-1)\Big)$
16	383	(8, 80, 311, 394, 58, 55, 250, 298, 56, 197, 280, 40, 229, 40, 136, 192)

Row 1 and row 2 of $[\mathbf{H}(D)]_h$ are linearly dependent. Therefore, applying **Step 2** of algorithm MB, row 2 is changed according to

$$[\text{Row 2}] \Longleftarrow [\text{Row 2}] + D \times [\text{Row 1}]$$

resulting in the minimal basic parity-check matrix whose overall constraint length is 3

$$\mathbf{H_{mb}}(D) = \begin{pmatrix} 1{+}D{+}D^2 & 1{+}D^2 & D{+}D^2 & 0 & 0 & D & D{+}D^2 \\ 1{+}D & D & 1{+}D & 1 & 1{+}D & 1{+}D & 0 \end{pmatrix} \tag{6.60}$$

Finally, applying (6.56) yields the reciprocal parity-check matrix $\widetilde{\mathbf{H}}(D)$ as

$$\widetilde{\mathbf{H}}(D) = \begin{pmatrix} 1{+}D{+}D^2 & 1{+}D^2 & 1{+}D & 0 & 0 & D & 1{+}D \\ 1{+}D & 1 & 1{+}D & D & 1{+}D & 1{+}D & 0 \end{pmatrix} \tag{6.61}$$

6.3.4 Simulation Results and Discussion

We present simulation results that compare the error performance of turbo codes implementing the conventional BCJR algorithm and the dual-LM algorithm based on the dual trellis, for three high coding rates using puncturing. The mother code is the rate-1/2 constituent RSC code of the LTE turbo code. Data are transmitted in AWGN channel, using BPSK modulation. The information frame length is $K = 400$ bits. The systematic bits are not punctured. The puncturing patterns of the parity bits (Table 6.1) and the internal ARP interleaver have been jointly optimized according to the method described in [34]. The interleaver is defined by

$$\pi(i) = \Big(Pi + S\big(i \bmod Q\big)\Big) \bmod K. \tag{6.62}$$

and the corresponding parameters are given in Table 6.2.

The simulations were carried out with floating point representation of data and the number of iterations is set to 8. Fig. 6.4 shows that both decoding approaches yield similar error correction performance for the turbo code. From simulation results, we can assure that for any given high-rate convolutional code, obtained by true-high-rate encoding matrix or puncturing, the corresponding dual-trellis can always be founded

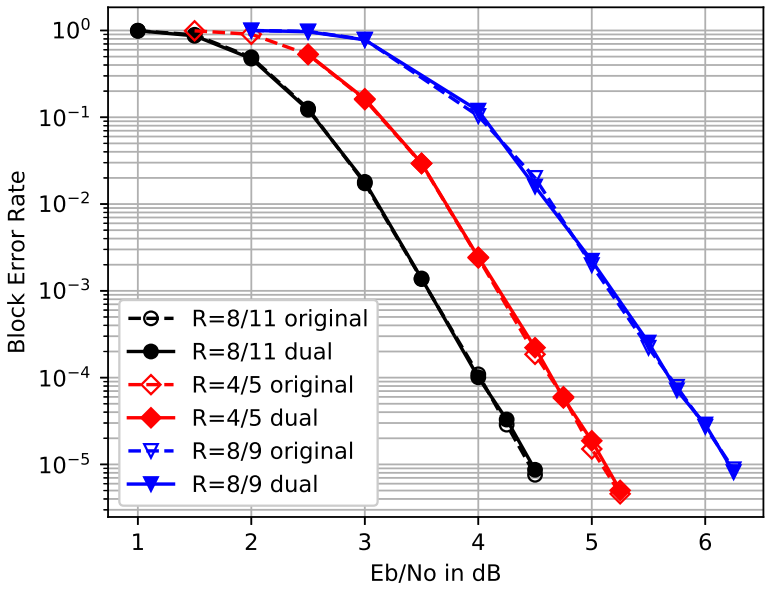


Figure 6.4: Performance comparison between MAP and dual-MAP algorithms for various high-rate schemes for $K = 400$. AWGN channel and BPSK modulation.

and the dual-LM algorithm produces the same performance as the BCJR (or log-MAP) algorithm.

Furthermore, the decoder throughput can be enhanced using the dual-LM algorithm, as explained in Section 6.2.2.2 at the expense of a higher hardware complexity. It turns out that this hardware complexity can be reduced by allowing some degradation in the error performance of the decoder. The motivation for such a solution is the same that led to the sub-optimal MLM algorithm with the original trellis. In the next section, we derive a sub-optimal but less complex version of the dual-LM algorithm, also using the dual trellis.

6.4 The Dual-Max-Log-MAP Algorithm

For convolutional codes with high coding rates k/n, since the dual trellis is shorter than the original one, the decoding throughput of the dual-LM decoder increases with the coding rate, while the decoding throughput of the MLM decoder remains unchanged. The price to pay for a higher throughput is a larger circuit area of the dual-LM decoder for all considered coding rates, since the dual-LM algorithm has to process a large number of systematic and extrinsic information values in parallel. This area increase is currently regarded as the main obstacle to the extensive implementation of the dual-LM algorithm.

In this section, a new decoding algorithm based on the dual trellis is proposed, called *dual-Max-Log-MAP* (dual-MLM). Compared to the dual-LM algorithm, it exhibits a lower complexity while keeping the high-throughput property. More specifically, the number of LUTs required in the SOU is significantly reduced, compared to the dual-LM algorithm. This reduction results in a decrease of the decoder circuit area at the cost of a minor penalty in performance.

6.4.1 Drawbacks of the Dual-Log-MAP Algorithm

Recalling from Section (6.2.2), if the SM representation of real numbers is adopted:

$$x = (-1)^{X_s} \exp(X_m) \triangleq [X_s; X_m], \tag{6.63}$$

the different processing steps of the dual-LM algorithm are the branch metric calculation (6.35), the forward metric recursion (6.36), the backward metric recursion (6.37), and soft output calculation (6.43).

Therefore, multiplications and divisions between two real numbers can be carried out by adding and subtracting their corresponding SM representations, respectively. However, in order to implement the addition of two real numbers, the SMA is performed as

$$x + y \triangleq \begin{cases} \left[X_s; X_m - \log\left(1 + (-1)^{X_s+Y_s} \exp(Y_m - X_m)\right)\right], & \text{if } X_m > Y_m, \\ \left[Y_s; Y_m - \log\left(1 + (-1)^{X_s+Y_s} \exp(X_m - Y_m)\right)\right], & \text{otherwise.} \end{cases} \tag{6.64}$$

This implementation requires the use of LUTs to perform the two following functions:

$$f(\Delta) = \log\left(1 + \exp(-\Delta)\right), \tag{6.65}$$

$$g(\Delta) = \log\left(1 - \exp(-\Delta)\right), \tag{6.66}$$

where Δ is a positive real number. Functions $f(\Delta)$ and $g(\Delta)$ are shown in Fig. 6.5 and the hardware architecture of the SMA is shown in Fig. 6.6. In the original trellis, only function $f(\Delta)$ is employed and it can be replaced in practice by a scaling factor in the MLM decoder [45]. Differently, the bit metrics in the dual-MAP algorithm can be negative, thus requiring the use of function $g(\Delta)$. Function $g(\Delta)$ is not bounded when Δ goes to zero, therefore, it can not be discarded or approximated as in the case of function $f(\Delta)$ in the MLM or LM decoder.

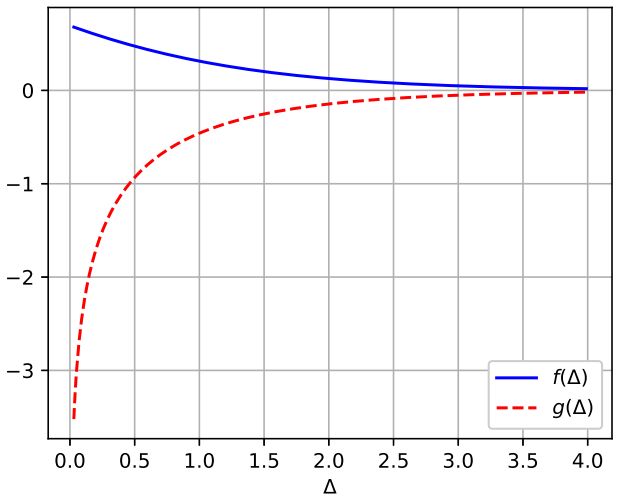


Figure 6.5: The function $f(\Delta)$ converges to $\sqrt{2}$ when Δ goes to zero, while the function $g(\Delta)$ does not converge and is $-\infty$ as $\Delta = 0$.

The required use of two LUTs to implement an SMA operator has a penalizing impact on the dual-LM decoder implementation. In order to achieve the highest possible throughput, the SOU in charge of computing the *a posteriori* likelihood ratios, has to perform

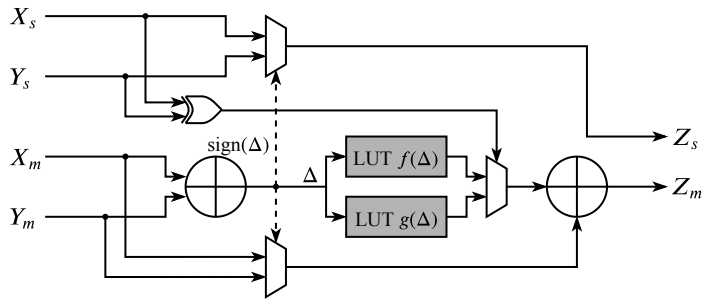


Figure 6.6: Hardware architecture of an SMA operator performing $z = x + y$.

simultaneously the computation of (6.43) for every information bit in a section of the dual trellis. The corresponding architecture is shown in Fig. 6.7 for a radix-2 dual trellis with 4 states. Furthermore, the number of required SMA operators employed increases with the number of states and with the rate of the original convolutional code: if the code has 2^ν states and rate $r = k/n$, the number of SMA operators required for the SOU of the dual-LM decoder is $2k \times (2^\nu - 1)$. Since each SMA operator requires two LUTs, the SOU consists of a total of $4k \times (2^\nu - 1)$ LUTs. Consequently, the implementation of the dual-LM decoder is only of interest for very high coding rates: it was reported in [85] that, for a rate-8/9 turbo code, the SOUs occupy more than 30 % the circuit area of the turbo decoder.

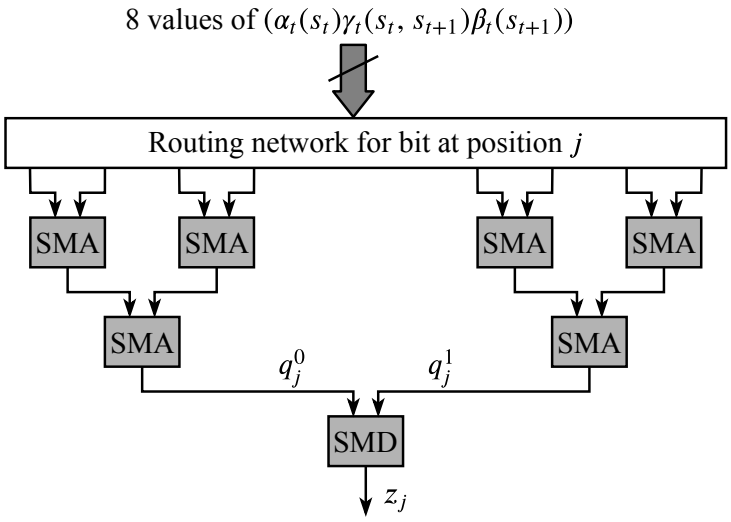


Figure 6.7: Hardware architecture for the calculation of (6.43) for bit at position j.

6.4.2 Max-Log Approximation for the Extrinsic Information Calculation

The complexity of the dual-LM decoder can be lowered by limiting the use of LUTs to compute (6.65) and (6.66). These LUTs are mainly used to derive the soft outputs according to (6.43) in the SOUs. Therefore, we consider approximating the calculation in the logarithmic domain of q_j^0 and q_j^1 for each bit at position j in the trellis section t,

where

$$q_j^0 = \sum_{(s_t,s_{t+1})|c_j^+(s_t,s_{t+1})=1} \alpha_t(s_t)\gamma_t(s_t,s_{t+1})\beta_{t+1}(s_{t+1}), \tag{6.67}$$

$$q_j^1 = \sum_{(s_t,s_{t+1})|c_j^+(s_t,s_{t+1})=0} \alpha_t(s_t)\gamma_t(s_t,s_{t+1})\beta_{t+1}(s_{t+1}) \tag{6.68}$$

For the calculation of q_j^0, if the trellis section has $\mathcal{N}$ branches having bit decision equal to zero at position j, we denote by $\mathcal{S}_j = \{1, \ldots, \mathcal{N}\}$ the index set of those branches and by w_s the metric of branch $s \in \mathcal{S}_j$ corresponding to state transition (s_t, s_{t+1}):

$$w_s = \alpha_t(s_t)\gamma_t(s_t,s_{t+1})\beta_{t+1}(s_{t+1}). \tag{6.69}$$

Then, (6.67) can be rewritten as:

$$q_j^0 = \sum_{s\in\mathcal{S}_j} w_s. \tag{6.70}$$

The value of w_s can be negative or positive and the corresponding sign is not known *a priori*. Therefore, with the SM representation, one cannot apply the max-log approximation for a set of positive numbers

$$\log\big(\sum_{i=1}^{N} e^{x_i}\big) \approx \max_{i=1,\ldots,N}\{x_i\},$$

to the summation in (6.70). However, if we denote by $\mathcal{S}_j^+$ and $\mathcal{S}_j^-$ the index sets of branches having positive metrics and negative metrics, respectively, (6.70) can be expressed as

$$q_j^0 = \Big(\sum_{r\in\mathcal{S}_j^+} w_r\Big) + \Big(\sum_{p\in\mathcal{S}_j^-} w_p\Big). \tag{6.71}$$

Since the elements in the two summations terms in (6.71) have the same sign value, these summations can be expressed in SM representation as

$$\sum_{r\in\mathcal{S}_j^+} w_r = \Big[0;\ \log\big(\sum_{r\in\mathcal{S}_j^+} e^{W_{r,m}}\big)\Big], \tag{6.72}$$

$$\sum_{p\in\mathcal{S}_j^-} w_p = \Big[1;\ \log\big(\sum_{r\in\mathcal{S}_j^-} e^{W_{p,m}}\big)\Big]. \tag{6.73}$$

where $W_{r,m}$ and $W_{p,m}$ are the magnitudes of w_r and w_p, respectively. Then, based on the max-log approximation, (6.72) and (6.73) can be approximated in the SM domain as

$$\sum_{r\in\mathcal{S}_j^+} w_r \approx \big[0;\ \max_{r\in\mathcal{S}_j^+}(W_{r,m})\big] \tag{6.74}$$

$$\sum_{p\in\mathcal{S}_j^-} w_p \approx \big[1;\ \max_{p\in\mathcal{S}_j^-}(W_{p,m})\big] \tag{6.75}$$

Finally, by substituting (6.74) and (6.75) into (6.71), the SM representation of q_j^0 after the SMA operation can be written as

$$Q_{j,s}^0 = W_{l_j,s}, \tag{6.76}$$

$$Q_{j,m}^0 \approx W_{l_j,m} - \log(1 - e^{-\Delta_j}), \tag{6.77}$$

where $l_j = \arg\min_{i \in \mathcal{S}_j}\{W_{i,m}\}$ and $\Delta_j = |\max_{r \in \mathcal{S}_j^+}\{W_{r,m}\} - \max_{p \in \mathcal{S}_j^-}\{W_{p,m}\}|$. We can observe that the calculation of (6.77) involves only one LUT for function $g(\Delta_j)$, regardless of the number of considered branches. Consequently, through branch rearrangement, we can significantly reduce the number of LUTs employed by the dual MAP decoder.

However, another problem arises when implementing (6.77). Intuitively, (6.77) can be computed by finding the minimum metric in $\mathcal{S}_j^+$ and the minimum metric in $\mathcal{S}_j^-$. Then, Δ_j and $W_{l_j,s}$ are the results of a subtraction and a sign detection, respectively. However, the number of branches in $\mathcal{S}_j^+$ and $\mathcal{S}_j^-$ is not constant. It depends on the considered dual trellis section and varies with each received frame. Therefore, a predefined hardware implementation for finding the minimum values in these two sets is not feasible in practice. This problem can be solved by resorting to the local-SOVA (see Section 4.3), to compute $W_{l_j,s}$, $W_{l_j,m}$ and Δ_j in (6.77).

6.4.3 Extrinsic Information Calculation using the Local-SOVA

The algorithm presented in this section is a variant of the original local-SOVA shown in Chapter 4. The local-SOVA performs operations on 3-tuple entities, called *paths*. A path P consists of a metric value denoted by M, a sign value S, and a reliability value related to S, denoted by R:

$$P = \{M, S, R\} \in \mathbb{R} \times \{0, 1\} \times \mathbb{R}^+, \tag{6.78}$$

where $\mathbb{R}$ is the set of real numbers and $\mathbb{R}^+$ is the set of positive real numbers.

Given a set of $\mathcal{N}$ paths, each has a sign value which is positive ($S = 0$) or negative ($S = 1$). Also, each path has a pre-computed path metric M and its initial reliability value R is set to $+\infty$ or to the largest possible value achievable with quantization. Then, the local-SOVA processes as shown in Algorithm 3. The outcome of the algorithm is the path with the minimum metric value among all paths. The corresponding sign and metric are provided as well as the associated reliability value, which is the minimum metric among the set of competing paths having a different sign value.

When using Algorithm 3 to calculate (6.76) and (6.77), the number of paths $\mathcal{N}$ to be considered is the cardinality of set $\mathcal{S}_j$. The metric of path $p \in \mathcal{S}_j$ is w_p, which is written as $[W_{p,s}, W_{p,m}]$ using the SM representation. The $\mathcal{N}$ paths are initialized as

$$P_p = \{M_p, S_p, R_p\} = \{W_{p,m}, W_{p,s}, +\infty\}, p = 1 \cdots \mathcal{N}. \tag{6.79}$$

After having processed the local-SOVA according to Algorithm 3, the output path is $\{M_1, S_1, R_1\}$. If $l_j = \arg\max_{p \in \mathcal{S}_j}\{W_{p,m}\}$, and if we assume that $l_j \in \mathcal{S}_j^+$, then we have:

$$M_1 = W_{l_j,m} = \max_{r \in \mathcal{S}_j^+}\{W_{r,m}\}, \tag{6.80}$$

$$S_1 = W_{l_j,s}, \tag{6.81}$$

$$R_1 = \max_{p \in \mathcal{S}_j^-}\{W_{p,m}\}. \tag{6.82}$$

Algorithm 3 The local-SOVA for extrinsic information approximation

Initialization: $\mathcal{N}$ paths $\{P_1, \ldots, P_{\mathcal{N}}\}$;
$P_i = \{M_i, S_i, R_i\}$, for $i = 1, \ldots, \mathcal{N}$;
$\mathcal{L} = \log_2(\mathcal{N})$ as the number of layers;
for each layer $l = 1, \ldots, \mathcal{L}$ **do**
    for each path $p = 1, \ldots, 2^{(\mathcal{L}-l)}$ **do**
        $a = \arg\max_{j \in \{2p-1, 2p\}} \{M_j\}$;
        $b = \arg\min_{j \in \{2p-1, 2p\}} \{M_j\}$;
        if $S_a = S_b$ **then**
            $R_p = \min(R_a, R_b)$;
        else
            $R_p = \min(R_a, M_b)$;
        end if
        $M_p = M_a$; $S_p = S_a$;
    end for
end for
Output:
M_1: maximum metric among all paths,
S_1: sign value of the path with maximum metric,
R_1: maximum metric among paths having sign value different from S_1.

If $l_j \in \mathcal{S}_j^-$, $\mathcal{S}_j^+$ and $\mathcal{S}_j^-$ have to be swapped in (6.80) and (6.82). The value of Δ_j in (6.77) is then

$$\Delta_j = |\max_{r \in \mathcal{S}_j^+} \{W_{r,m}\} - \max_{p \in \mathcal{S}_j^-} \{W_{p,m}\}| = |M_1 - R_1|. \tag{6.83}$$

To summarize, the computation of q_j^0 using the local-SOVA results in:

$$Q_{j,s}^0 = S_1 \tag{6.84}$$

$$Q_{j,m}^0 = M_1 - \log\left(1 - e^{-|M_1 - R_1|}\right) \tag{6.85}$$

In addition, the practical implementation of the local-SOVA can be carried out in a dichotomous fashion using elementary merge operations. In Algorithm 3, two paths P_{2p-1} and P_{2p} at layer $l-1$ are merged into a resulting path P_p at layer l. The overall extrinsic information calculation can therefore be implemented as a tree structure composed of elementary merge operators. For example, the overall structure using the local-SOVA to calculate (6.76) and (6.77) with $\mathcal{N} = 4$ paths is described in Fig. 6.8 in the form of a binary tree with $\mathcal{L} = \log_2(\mathcal{N}) = 2$ layers. A possible architecture for the elementary merge operator is described in Fig. 6.9.

6.4.4 The Dual-Max-Log-MAP Algorithm

Based on the above analysis, we propose the new low-complexity dual-MLM algorithm to decode high coding rates convolutional codes using their dual trellis. The decoding procedure is described as follows. Note that we skip the branch metric calculation since it is not different from the dual-LM algorithm.

As expressed in (6.36) and (6.37), the metric recursion involves summations of real values. The number of required additions depends on the dual trellis structure. If a

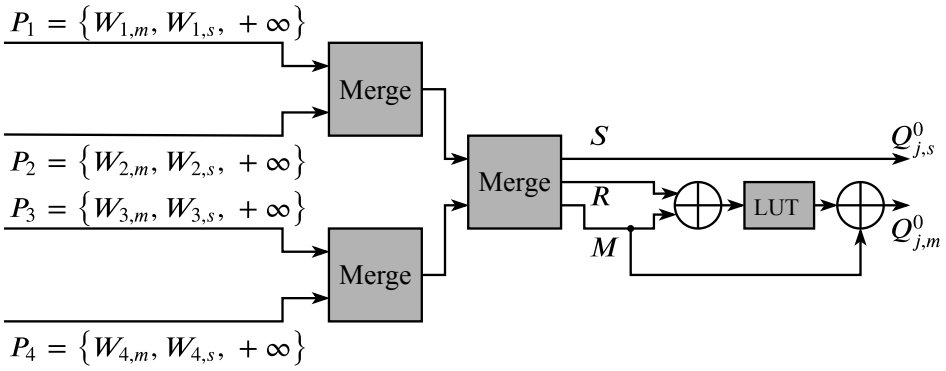


Figure 6.8: Proposed hardware architecture of a local-SOVA decoder used to compute (6.76) and (6.77) for $\mathcal{N} = 4$.

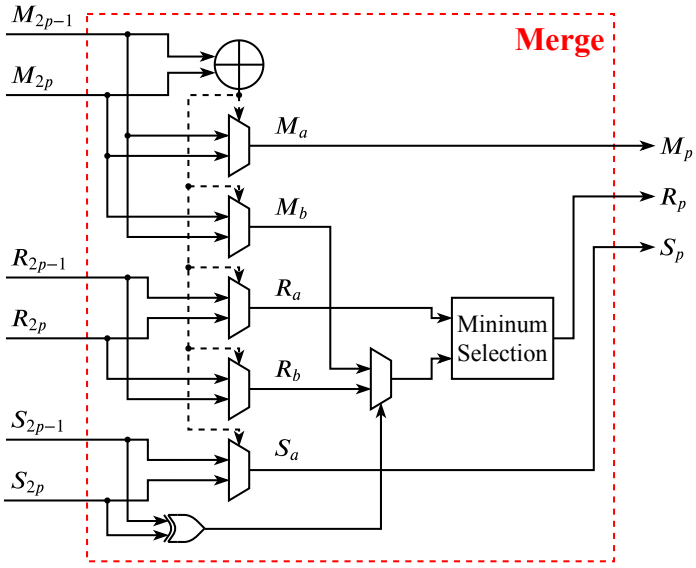


Figure 6.9: Hardware architecture of a merge operator that selects the minimum metric with its corresponding sign and updates the reliability value.

radix-2^T dual trellis (2^T state transitions enter and leave each state) is considered, the decoder needs $(2^T - 1)$ adders for each backward or forward metric recursion. With the SM representation, two LUTs are required for each SMA operator. Hence, a total number of $2\times(2^T - 1)$ LUTs are necessary for a straightforward implementation of each recursion.

However, thanks to the proposed SMA approximation and using the the local-SOVA for its implementation, the number of LUTs needed to implement each recursion using the SM representation is reduced to one.

The extrinsic information calculation for each bit in a dual trellis section is given by (6.43). It involves the calculation of q_j^0 and q_j^1 for each position j as expressed in (6.67) and (6.68), respectively. The corresponding calculation has already been dealt with the max-log approximation and the local-SOVA implementation solution. Then, the *a posteriori* information z_j can be easily derived using an SMD operator.

As a final step, the conversion of the extrinsic information from the dual trellis domain to the original trellis domain (6.49) is necessary. However, due to the above-mentioned approximations, the resulting extrinsic information values in the dual trellis domain are

found to be over-estimated, which in turn produce under-estimated values in the original trellis domain. Therefore, we employed two *scaling factors*, denoted by ϕ_1 and ϕ_2, to mitigate this over-estimation problem. Notice that the idea of using scaling factors is commonly used with sub-optimal SISO decoding algorithms such as the MLM [45] or SOVA [89] algorithms.

The expression used for the conversion of the extrinsic information into the original trellis domain is therefore:

$$\mathrm{L}^e(\hat{c}_j) = \begin{cases} \phi_1 \log\Big(\big|\tanh(\phi_2 \frac{U_{j,m}}{2})\big|\Big), & \text{if } U_{j,s} = 1, \\ -\phi_1 \log\Big(\big|\tanh(\phi_2 \frac{U_{j,m}}{2})\big|\Big), & \text{otherwise.} \end{cases} \tag{6.86}$$

In practice, the optimal values for the scaling factors were found empirically. We performed a computer search in an effort to find the values of ϕ_1 and ϕ_2 providing the best error correcting performance.

6.4.5 Complexity Analysis and Simulation Results

In order to illustrate the benefits of the proposed dual-MLM algorithm, we first perform an analysis of the complexity savings due to the simplifications and then present simulations results to assess its error correction performance.

First, a simplified complexity comparison, in terms of number of adders and LUTs required by the decoder, is carried out between the dual-LM and the proposed dual-MLM algorithms, for an 8-state (3 memory elements) convolutional code with various coding rates $k/(k+1)$. In this comparison, we exclude the calculation of the branch metrics since it is the same for both algorithms. As shown in Table 6.3, despite a minor increase in the number of adders, the dual-MLM algorithm uses significantly less LUTs than the dual-ML algorithm. More specifically, due to the max-log approximation, the use of function $f(\Delta)$ is no longer needed in (6.65) and the use of function $g(\Delta)$ is also limited, thanks to the local-SOVA. As the coding rate increases, the percentage of LUTs saved in the dual-MLM decoder also increases. For instance, with coding rate 4/5, the total number of LUTs used in the dual-LM decoder is 144, compared to 24 LUTs in the dual-MLM decoder. When raising the coding rate up to 8/9, dual-MLM decoding requires only 32 LUTs compared to 256 for dual-LM decoding.

Next, we conducted several numerical simulations to assess and compare the performance of the dual-LM and the dual-MLM algorithms. For validation purposes, we also included the performance of the MLM decoder on the original trellis as a reference.

In the simulations, information frames are encoded by the turbo code consisting of two identical LTE constituent RSC codes [5] with generator matrix

$$\mathbf{G_{LTE}}(D) = \begin{pmatrix} 1 & \frac{1+D+D^3}{1+D^2+D^3} \end{pmatrix}. \tag{6.87}$$

For the internal interleaver, we chose the ARP interleaver defined by

$$\pi(i) = \Big(Pi + S\big(i \bmod Q\big)\Big) \bmod K, \quad i = 1, \ldots, K. \tag{6.88}$$

The parameters of the ARP interleaver are given in Table 6.4. Furthermore, puncturing was employed to achieve different high coding rates, and the puncturing patterns of the

Table 6.3: Complexity comparison between the dual-LM and the dual-MLM decoders for various coding rates

Coding rate	**dual-Log-MAP**			**dual-Max-Log-MAP**		
	Adders	**LUTs**		**Adders**	**LUTs**	
		f(Δ)	**g(Δ)**		**f(Δ)**	**g(Δ)**
4/5	168	72	72	184	0	24
8/9	288	128	128	320	0	32
16/17	528	240	240	592	0	48

Table 6.4: ARP interleaver parameters

Q	P	$\Big(S(0), \ldots, S(Q-1)\Big)$
16	383	(8, 80, 311, 394, 58, 55, 250, 298, 56, 197, 280, 40, 229, 40, 136, 192)

Table 6.5: Parity puncturing patterns for various coding rates

Turbo rate	CC rate	Parity puncturing pattern
2/3	4/5	1000
4/5	8/9	01000000
8/9	16/17	0100000000000000

parity bits were jointly optimized with the ARP interleaver as described in [34]. Table 6.5 shows the puncturing patterns used in the simulations for the coding rates 4/5, 8/9 and 16/17 of the constituent convolutional code.

The encoded frames are modulated using a BPSK modulation and are sent through an AWGN channel. All three decoding algorithms were carried out with a fixed-point representation of data, the channel values being quantized on 6 bits. The number of iterations is set to 8 for all decoders. For the dual-MLM decoder, the values of the scaling factors (ϕ_1, ϕ_2) are $(1.3, 0.75)$ for $r = 2/3$, and $(1.15, 0.75)$ for $r = 4/5$ and $r = 8/9$. The simulation results are shown in Fig. 6.10 and Fig. 6.11 for information frame sizes $K = 400$ bits and $K = 992$ bits, respectively. We can see that the dual-LM and the MLM algorithms yield similar error rate, as expected. However, the dual-MLM algorithm entails a loss of about $0.2 - 0.3$ dB compared to the dual-LM algorithm at coding rate $r = 2/3$ but this loss reduces to $0.1 - 0.2$ dB at coding rates $r = 4/5$ and $r = 8/9$. Therefore, the dual-MLM algorithm can be regarded as a sub-optimal but low-complexity decoding algorithm compared to the dual-LM algorithm.

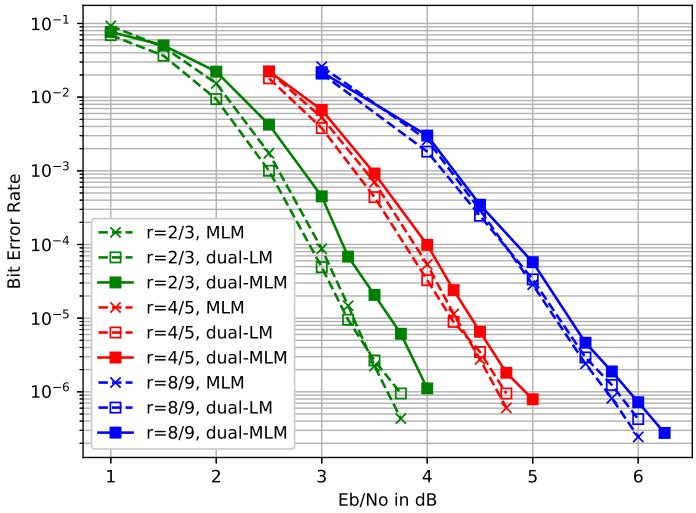


Figure 6.10: Performance comparison between the MLM, the dual-LM and the dual-MLM algorithms with $K = 400$ bits.

6.5 Conclusion

In this chapter, we have shown that a high code rate ($r > 1/2$) convolutional code can be decoded using the dual trellis, i. e. the trellis of the corresponding dual code. A high-rate convolutional code can be acquired by two means: using a true high-rate encoding matrix or puncturing. For both cases, we have shown a generic procedure to construct the dual trellis for a given high code rate convolutional code. Then, the BCJR algorithm and its logarithmic version can be used onto the dual trellis to decode the high-rate convolutional code and produce the extrinsic information. Since these algorithms are performed on the dual trellis, they are referred to as the dual-MAP and the dual-LM algorithms. Numerical results have shown that these algorithms using the dual trellis

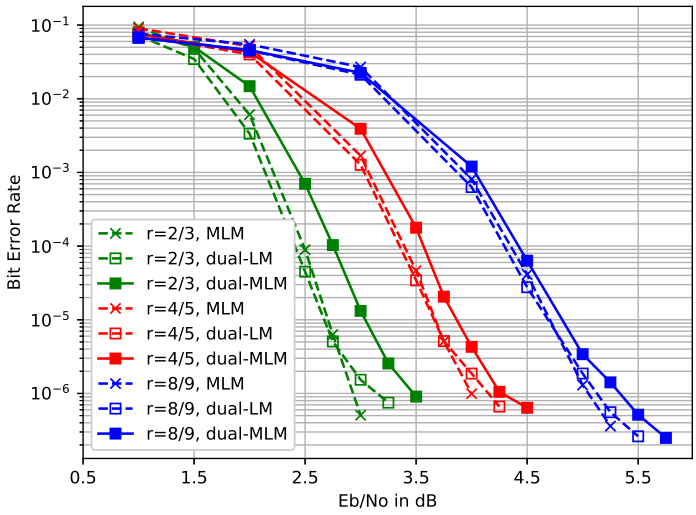


Figure 6.11: Performance comparison between the MLM, the dual-LM and the dual-MLM algorithms with $K = 992$ bits.

yield the same error correction performance as the BCJR and log-MAP algorithms, even in the turbo iterative decoding process.

Furthermore, as shown in [85], using the dual trellis offers the advantage of high throughput decoding for high-rate coding schemes and the ratio of throughput to chip area is largely increased compared to decoding using the original trellis of the mother code. Nevertheless, the circuit area of the decoder using the dual-LM algorithm is still higher than the MLM decoder since the extrinsics for all systematic bits in a dual trellis section have to be processed at the same time. Each extrinsic information calculation, in turns, involves a large number of LUTs to operate. Therefore, an abundant number of LUTs are required for the dual-LM algorithm. However, we have shown that the state metric recursions and the extrinsic information calculations can then be reformulated using the max-log approximation and can be implemented with the local-SOVA architecture. With these solutions, we came up with a new algorithm for the dual trellis, namely the dual-MLM algorithm. A complexity analysis was conducted, showing that the number of LUTs employed in the decoder can then be considerably reduced compared to the dual-LM algorithm. Also, based on numerical simulations, we observed that dual-MLM decoding yields only a minor loss of about 0.2 dB in performance at 10^{-6} of bit error rate compared to dual-LM decoding. Therefore, it can be considered as a viable and practical low-complexity sub-optimal decoding algorithm.

The generalization of the dual trellis construction method was published and presented at the 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2019) [22]. A presentation on this subject was also given in the GdR ISIS Workshop: Enabling Technologies for sub-TeraHertz and TeraHertz communications [23].

Furthermore, the proposed low-complexity dual trellis decoding algorithm was published and presented at the Wireless Communications and Networking Conference (WCNC 2020) [24].

Chapter 7

Conclusion and Future Works

7.1 Conclusion

In this thesis, we have studied and implemented a new decoding algorithm that can be employed in the turbo decoder with low complexity and provide several trade offs between complexity, latency, and error correction performance. Furthermore, the decoding process using the dual trellis was investigated, where a generalized dual trellis constructing method and a new low complexity decoding algorithm was addressed.

First, a review on turbo codes and decoding algorithms employed in turbo codes was given. We then focused further on techniques and architectures employed in turbo decoders to increase the throughput and lower the complexity of the decoder. For the use cases defined in the H2020 EPIC project, the requirement in throughput were in the order of 500 Gb/s up to 1 Tb/s. Therefore, the UXMAP architecture was chosen to achieve this goal in turbo decoding due to its ability to provide very-high-throughput with high area efficiency.

Originally, the UXMAP architecture was coupled with the Max-Log-MAP decoding algorithm and their combination has already led to a very high-throughput turbo decoder (up to 400 Gb/s). Nevertheless, we investigated its combination with a new decoding algorithm that we proposed and called Local-SOVA. Le local-SOVA can deliver the same error correction performance as the MLM algorithm but with a lower complexity. The analysis related to the local-SOVA was carried out in Chapter 4 revealing that the MLM algorithm is actually an instance of the Local-SOVA and that the Local-SOVA can have a lower computational complexity in high-radix schemes and can provide numerous trade-offs between complexity and error performance.

The implementation of the Local-SOVA in the UXMAP architecture in Chapter 5 consolidated the analysis provided in Chapter 4. The first implementation was the radix-4 Local-SOVA decoder. In comparison with the radix-4 MLM decoder, the Local-SOVA decoder brings a saving in area complexity of 33% with negligible performance loss (smaller than 0.05 dB). For the UXMAP architecture, this saving in area can be translated into an increase in throughput by using larger frame sizes. Furthermore, we also investigated the radix-8 and radix-16 local-SOVA decoders. Both schemes provide a lower latency solution to the UXMAP architecture at the expense of area complexity and error correction performance. The radix-16 Local-SOVA has a 2 times lower latency than the radix-4 Local-SOVA, and this is traded with an increase of 50% in area complexity and a error correction performance degradation of 0.1 dB.

On the other hand, the development of the dual trellis and of decoding algorithms using the dual trellis were studied in Chapter 6. The main advantage of decoding a convolutional code using its dual trellis is that the decoding throughput increases with the coding rate. On the contrary, when the decoding process uses the trellis of the mother code through puncturing, the throughput of the decoder is almost the same for every code rates. Therefore, we first generalized the method for constructing the dual trellis for high-rate punctured convolutional codes. The decoding algorithm can then be applied to the resulting dual trellis to produce the extrinsic information. One main drawback of the conventional decoding algorithm using the dual trellis is the high number of LUTs that are required for its hardware implementation, which leads to a high complexity decoder. To this end, we introduced a new variant, namely dual-MLM, that can decrease drastically the use of the LUTs. Although a small performance degradation of $0.1 - 0.2$ dB can be observed, the new decoding algorithm provides a lower complexity solution to the decoding process using the dual trellis for high-rate convolutional codes.

7.2 Future Works

The work presented in this thesis showed that the local-SOVA has a very high potential as a decoding algorithm for a SISO decoders of convolutional codes and turbo codes. From now on, when considering a decoding algorithm for convolutional and turbo codes, the local-SOVA shall be one candidate to compete with the MLM, LM or SOVA algorithms.

In particular, with the UXMAP architecture, it is obvious that the local-SOVA should now replace the MLM algorithm. Nevertheless, the current implementations of the local-SOVA are only a few realizations out of many possibilities that can be carried out, since numerous trade-offs between complexity and error correction performance can be done. The implemented radix-4 local-SOVA currently employs ϕ operators in the ACSU and a mix of ω and ϕ operators in the SOU. Further simplifications without foreseeable impact on the error performance, as shown in Chapter 4, should be investigated. However, each solution should come with an efficient hardware implementation to leverage its advantages, and this needs further investigation. For the radix-8 and radix-16 local-SOVA, further investigation on other implementations can be carried out to minimize the increase in complexity and reduce the performance loss. Furthermore, with the local-SOVA, one can also consider the implementation of the radix-32 or even radix-64 schemes for applications that require very low latency. As far as we know, these high-radix schemes are not feasible in practice for the MLM algorithm. For the local-SOVA, the target should be to limit the increase in complexity while providing an acceptable error correction performance.

For other turbo decoder architectures, the local-SOVA clearly deserves more attention and could be employed instead of the MLM algorithm. Moreover, for applications involving a trellis diagram with a high number of states, the local-SOVA can be considered as a candidate besides the SOVA [76]. These applications can be demodulation, equalization, and decoding in communication and storage systems. Compared to the SOVA, the local-SOVA can have a lower latency due to the absence of the traceback and update procedures. Furthermore, with a high number of states in the tellis diagram, the saving in complexity using the local-SOVA can be more pronounced.

On another note, for high-rate convolutional codes, a thorough investigation of the hardware implementation of the dual-MLM algorithm shall be considered in the future. The performed analysis has already led to a lower complexity algorithm but a hardware

implementation is required to complete the study of the dual-MLM algorithm. If the implementation results are in line with the preliminary analysis, then the dual-MLM algorithm can be considered as a first choice decoding algorithm for high-throughput high-rate convolutional and turbo decoders.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," 1965.

[2] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[3] *3rd Generation Partnership Project (3GPP);Technical Specification Group (TSG) Radio Access Network (RAN); Working Group 1 (WG1); Multiplexing and channel coding (FDD)*, 3GPP TS 25.212 V1.0.0, April 1999.

[4] *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Multiplexing and channel coding (FDD) (Release 7)*, 3GPP TS 25.212 V7.7.0, Nov. 2007.

[5] *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (Release 8)*, 3GPP TS 36.212 V8.5.0, Dec. 2008.

[6] *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (3GPP TS 36.213 version 13.1.0 Release 13)*, 3GPP TS 36.213 V13.1.0, Apr. 2016.

[7] *5G; NR; User Equipment (UE) radio access capabilities (3GPP TS 38.306 version 15.2.0 Release 15)*, 3GPP TS 38.306 V15.2.0, Sep. 2018.

[8] T. K. Moon, *Error correction coding: mathematical methods and algorithms*. John Wiley & Sons, 2005.

[9] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proc. IEEE Int. Conf. Commun.*, vol. 2, May 1993, pp. 1064–1070.

[10] R. Gallager, "Low-density parity-check codes," *IRE Trans. Info. Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[11] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Info. Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.

[12] C. Douillard and M. Jézéquel, "Chapter 1. Turbo codes: from first principles to recent standards," in *Channel Coding: Theory, Algorithms, and Applications*. Academic Press, July 2014, pp. 1–53.

[13] V. Petrov, A. Pyattaev, D. Moltchanov, and Y. Koucheryavy, "Terahertz band communications: Applications, research challenges, and standardization activities," in *Int. Congress Ultra Modern Telecom. Control Systems Workshops*, 2016, pp. 183–190.

[14] T. Braud, F. H. Bijarbooneh, D. Chatzopoulos, and P. Hui, "Future networking challenges: The case of mobile augmented reality," in *ICDCS*, 2017, pp. 1796–1807.

[15] T. S. Rappaport, Y. Xing, G. R. MacCartney, A. F. Molisch, E. Mellios, and J. Zhang, "Overview of millimeter wave communications for fifth-generation (5g) wireless networks—with a focus on propagation models," *IEEE Trans. Antennas Propagation*, vol. 65, no. 12, pp. 6213–6230, 2017.

[16] Y. Cui, H. Wang, X. Cheng, and B. Chen, "Wireless data center networking," *IEEE Wireless Comm.*, vol. 18, no. 6, pp. 46–53, 2011.

[17] ITU-T, "G.709: Interfaces for the optical transport network," http://https://www.itu.int/rec/T-REC-G.709/, 2016.

[18] H. Caleb, "Dankberg: Viasat-3 satellite will have more capacity than the rest of the world combined," *Satellite Today*, vol. 10, 2016.

[19] C. Kestel, M. Herrmann, and N. Wehn, "When channel coding hits the implementation wall," in *IEEE 10th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, Hong Kong, Dec. 2018, pp. 1–6.

[20] EPIC, "D1.2 B5G Wireless Tb/s FEC KPI Requirement and Technology Gap Analysis," https://epic-h2020.eu/downloads/EPIC-D1.2-B5G-Wireless-Tbs-FEC-KPI-Requirement-and-Technology-Gap-Analysis-PU-M07.pdf, 2018.

[21] V. H. S. Le, C. A. Nour, E. Boutillon, and C. Douillard, "Revisiting the Max-Log-Map algorithm with SOVA update rules: new simplifications for high-radix SISO decoders," *IEEE Trans. Comm.*, vol. 68, no. 4, pp. 1991–2004, 2020.

[22] ——, "Dual trellis construction for high-rate punctured convolutional codes," in *IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*, Istanbul, Turkey, Sept. 2019, pp. 1–7.

[23] V. H. S. Le, "Dual trellis construction for high-rate punctured convolutional codes (invited talk)," in *GdR ISIS Workshop: Enabling Technologies for sub-TeraHertz and TeraHertz communications*, Maisons-Alfort, France, Sept. 2019.

[24] V. H. S. Le, C. A. Nour, E. Boutillon, and C. Douillard, "A low-complexity dual trellis decoding algorithm for high-rate convolutional codes," in *IEEE Wireless Communications and Networking Conference (WCNC)*, Seoul, South Korea, May 2020.

[25] P. Elias, "Coding for noisy channels," *IRE Conv. Rec.*, vol. 3, pp. 37–46, 1955.

[26] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[27] D. Divsalar and F. Pollara, "Turbo codes for PCS applications," in *IEEE Int. Conf. Comm.*, vol. 1, 1995, pp. 54–59 vol.1.

[28] H. Ma and J. Wolf, "On tail biting convolutional codes," *IEEE Trans. Comm.*, vol. 34, no. 2, pp. 104–111, 1986.

[29] Hung-Hua Tang and Mao-Chao Lin, "On (n, n-1) convolutional codes with low trellis complexity," *IEEE Trans. Comm.*, vol. 50, no. 1, pp. 37–47, 2002.

[30] M. Tuchler and A. Dholakia, "New rate-(n-1)/n convolutional codes with optimal spectrum," in *IEEE Int. Symp. Info. Theory,*, 2002, pp. 396–.

[31] A. Graell i Amat, G. Montorsi, and S. Benedetto, "Design and decoding of optimal high-rate convolutional codes," *IEEE Trans. Inf. Theory*, vol. 50, no. 5, pp. 867–881, May 2004.

[32] J. Hagenauer, "Rate-compatible punctured convolutional codes (RCPC codes) and their applications," *IEEE Trans. Commun.*, vol. 36, no. 4, pp. 389–400, April 1988.

[33] G. Bégin, D. Haccoun, and C. Paquin, "Further results on high-rate punctured convolutional codes for Viterbi and sequential decoding," *IEEE Trans. Commun.*, vol. 38, no. 11, pp. 1922–1928, Nov 1990.

[34] R. Garzón-Bohórquez, C. Abdel Nour, and C. Douillard, "Protograph-based interleavers for punctured turbo codes," *IEEE Trans. Commun.*, vol. 66, no. 5, pp. 1833–1844, May 2018.

[35] G. D. Forney, "The Viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[36] F. Hemmati and D. Costello Jr, "Truncation error probability in Viterbi decoding," *IEEE Trans. Commun.*, vol. 25, pp. 530–532, 06 1977.

[37] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," in *Proc. IEEE GLOBECOM*, Nov 1989, pp. 1680–1686 vol.3.

[38] C. Berrou, P. Adde, E. Angui, and S. Faudeil, "A low complexity soft-output Viterbi decoder architecture," in *IEEE Int. Conf. Comm.*, vol. 2, 1993, pp. 737–740.

[39] Zhongfeng Wang and K. K. Parhi, "High performance, high throughput turbo/SOVA decoder design," *IEEE Trans. Comm.*, vol. 51, no. 4, pp. 570–579, 2003.

[40] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE Trans. Inf. Theory*, vol. 20, no. 2, pp. 284–287, March 1974.

[41] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. IEEE Int. Conf. Commun.*, vol. 2, June 1995, pp. 1009–1013 vol.2.

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

[43] J. Sun and O. Y. Takeshita, "Interleavers for turbo codes using permutation polynomials over integer rings," *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 101–119, 2005.

[44] C. Berrou, Y. Saouter, C. Douillard, S. Kerouedan, and M. Jezequel, "Designing good permutations for turbo codes: towards a single model," in *IEEE Int. Conf. Comm.*, vol. 1, 2004, pp. 341–345.

[45] J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder," *Electronics Letters*, vol. 36, no. 23, pp. 1937–1939, Nov 2000.

[46] A. Worm, P. Hoeher, and N. Wehn, "Turbo-decoding without SNR estimation," *IEEE Comm. Letters*, vol. 4, no. 6, pp. 193–195, 2000.

[47] C. Benkeser, A. Burg, T. Cupaiuolo, and Q. Huang, "Design and optimization of an HSDPA turbo decoder ASIC," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 98–106, Jan 2009.

[48] O. Muller, A. Baghdadi, and M. Jézéquel, "Exploring parallel processing levels for convolutional turbo decoding," in *Proc. 2nd ICTTA Conf.*, April 2006, pp. 2353–2358.

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

[50] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatenated codes," *TDA progress report*, vol. 42, no. 127, pp. 1–20, 1996.

[51] O. D. S. Gonzalez, "Towards higher speed decoding of convolutional turbocodes," Ph.D. dissertation, Télécom Bretagne, 2013.

[52] 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 Int. Solid-State Circuits Conference*, 2003, pp. 150–484 vol.1.

[53] M. J. Thul, F. Gilbert, T. Vogt, G. Kreiselmaier, and N. Wehn, "A scalable system architecture for high-throughput turbo-decoders," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 39, no. 1-2, pp. 63–77, 2005.

[54] A. Worm, H. Lamm, and N. Wehn, "A high-speed MAP architecture with optimized memory size and power consumption," in *IEEE Workshop Signal Proc. Systems*, 2000, pp. 265–274.

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

[56] C. Studer, C. Benkeser, S. Belfanti, and Q. Huang, "Design and implementation of a parallel turbo-decoder ASIC for 3GPP-LTE," *IEEE Journal Solid-State Circuits*, vol. 46, no. 1, pp. 8–17, 2011.

[57] S. Weithoffer, C. A. Nour, N. Wehn, C. Douillard, and C. Berrou, "25 Years of Turbo Codes: From Mb/s to beyond 100 Gb/s," in *IEEE 10th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, Hong Kong, Dec. 2018, pp. 1–6.

[58] A. Nimbalker, Y. Blankenship, B. Classon, and T. K. Blankenship, "ARP and QPP interleavers for LTE turbo coding," in *IEEE Wireless Communications Networking Conference*, 2008, pp. 1032–1037.

[59] T. Ilnseher, F. Kienle, C. Weis, and N. Wehn, "A 2.15GBit/s turbo code decoder for LTE advanced base station applications," in *IEEE 7th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, Gothenburg, Sweden, Aug. 2012, pp. 21–25.

[60] R. Shrestha and R. P. Paily, "High-throughput turbo decoder with parallel architecture for LTE wireless communication standards," *IEEE Trans. Circuits Systems I: Regular Papers*, vol. 61, no. 9, pp. 2699–2710, 2014.

[61] Y. Sun and J. R. Cavallaro, "Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder," *Integration VLSI Journal*, vol. 44, no. 4, pp. 305–315, 2011.

[62] C. Wong and H. Chang, "High-efficiency processing schedule for parallel turbo decoders using QPP interleaver," *IEEE Trans. Circuits Systems I: Regular Papers*, vol. 58, no. 6, pp. 1412–1420, 2011.

[63] S. Weithoffer, F. Pohl, and N. Wehn, "On the applicability of trellis compression to turbo-code decoder hardware architectures," in *IEEE 9th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, Brest, France, Sept. 2016, pp. 61–65.

[64] R. G. Maunder, "A fully-parallel turbo decoding algorithm," *IEEE Trans. Commun.*, vol. 63, no. 8, pp. 2762–2775, Aug 2015.

[65] A. Li, L. Xiang, T. Chen, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "VLSI implementation of fully parallel LTE turbo decoders," *IEEE Access*, vol. 4, pp. 323–346, 2016.

[66] S. Weithoffer, O. Griebel, R. Klaimi, C. A. Nour, and N. Wehn, "Advanced hardware architectures for turbo code decoding beyond 100 Gb/s," in *IEEE Wireless Communications and Networking Conference (WCNC)*, Seoul, South Korea, May 2020, pp. 1–6.

[67] S. Weithoffer, R. Klaimi, C. A. Nour, N. Wehn, and C. Douillard, "Fully pipelined iteration unrolled decoders the road to Tb/s turbo decoding," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Barcelona, Spain, May 2020, pp. 5115–5119.

[68] C. Tang, C. Wong, C. Chen, C. Lin, and H. Chang, "A 952ms/s Max-Log MAP decoder chip using radix-4× 4 ACS architecture," in *Proc. IEEE Asian Solid-State Circuits Conf.*, Nov 2006, pp. 79–82.

[69] K. Shr, Y. Chang, C. Lin, and Y. Huang, "A 6.6pj/bit/iter radix-16 modified log-MAP decoder using two-stage ACS architecture," in *Proc. IEEE Asian Solid-State Circuits Conf.*, Nov 2011, pp. 313–316.

[70] O. Sánchez, C. Jégo, M. Jézéquel, and Y. Saouter, "High speed low complexity radix-16 Max-Log-MAP SISO decoder," in *Proc. IEEE ICECS*, Dec 2012, pp. 400–403.

[71] F. J. Martin-Vega, F. Blanquez-Casado, F. J. López-Martínez, G. Gomez, and J. T. Entrambasaguas, "Further improvements in SOVA for high-throughput parallel turbo decoding," *IEEE Commun. Lett.*, vol. 19, no. 1, pp. 6–9, Jan 2015.

[72] Q. Huang, Q. Xiao, L. Quan, Z. Wang, and S. Wang, "Trimming soft-input soft-output Viterbi algorithms," *IEEE Trans. Commun.*, vol. 64, no. 7, pp. 2952–2960, July 2016.

[73] M. P. C. Fossorier, F. Burkert, and J. Hagenauer, "On the equivalence between SOVA and Max-Log-MAP decodings," *IEEE Commun. Lett.*, vol. 2, no. 5, pp. 137–139, May 1998.

[74] G. Battail, "Pondération des symboles décodés par l'algorithme de Viterbi," *Ann. Telecommun.*, vol. 42, no. 1-2, pp. 31–38, Jan 1987.

[75] L. Lin and R. S. Cheng, "Improvements in SOVA-based decoding for turbocodes," in *Proc. IEEE Int. Conf. on Commun.*, vol. 3, June 1997, pp. 137–139.

[76] S. Weithoffer, R. Klaimi, A. Charbel, N. Wehn, and C. Douillard, "Low-complexity computational units for the local-SOVA decoding algorithm," in *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, London, United Kingdom, Aug. 2020.

[77] T. Ilnseher, "Standard compliant ultra high throughput turbo code decoders," Ph.D. dissertation, Department of Electrical Engineering and Information Technology, University of Kaiserslautern, 2013.

[78] E. Boutillon, C. Douillard, and G. Montorsi, "Iterative decoding of concatenated convolutional codes: Implementation issues," *IEEE Proc.*, vol. 95, no. 6, pp. 1201–1227, 2007.

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

[80] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans.s Inf. Theory*, vol. 42, no. 2, pp. 429–445, 1996.

[81] E. Boutillon, J. Sánchez-Rojas, and C. Marchand, "Simplified compression of redundancy free trellis sections in turbo decoder," *IEEE Commun. Lett.*, vol. 18, no. 6, pp. 941–944, June 2014.

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

[83] S. Srinivasan and S. S. Pietrobon, "Decoding of high rate convolutional codes using the dual trellis," *IEEE Trans. Inf. Theory*, vol. 56, no. 1, pp. 273–295, Jan 2010.

[84] G. Montorsi and S. Benedetto, "An additive version of the SISO algorithm for the dual code," in *Proc. IEEE Int. Symp. Inf. Theory*, June 2001, p. 27.

[85] C. Lin, C. Wong, and H. Chang, "A 40 nm 535 Mbps multiple code-rate turbo decoder chip using reciprocal dual trellis," *IEEE J. Solid-State Circuits*, vol. 48, no. 11, pp. 2662–2670, Nov 2013.

[86] G. Forney, "Convolutional codes I: Algebraic structure," *IEEE Trans. Inf. Theory*, vol. 16, no. 6, pp. 720–738, November 1970.

[87] H. Sasano and S. Moriya, "A construction of high rate punctured convolutional codes," in *Int. Symp. Inf. Theory Applications*, Oct 2012, pp. 662–666.

[88] R. Johannesson and K. S. Zigangirov, *Fundamentals of convolutional coding*. Piscataway, NJ: IEEE Press, 1999.

[89] Chuan Xiu Huang and A. Ghrayeb, "A simple remedy for the exaggerated extrinsic information produced by the sova algorithm," *IEEE Trans. Wireless Comm.*, vol. 5, no. 5, pp. 996–1002, May 2006.