REDUCED COMPLEXITY TURBO DECODERS

YASSIR NAWAZ

Reduced Complexity Turbo Decoders

by

© Yassir Nawaz

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Engineering

Faculty of Engineering and Applied Science

Memorial University of Newfoundland

October, 2003

St John's Newfoundland

Your file Votre référence
ISBN: 0-612-99100-8
Our file Notre référence
ISBN: 0-612-99100-8

To mama, abu, saman, and khala

Abstract

Turbo codes are a class of forward error correction codes, which have outperformed all the previously known error coding schemes. The strength of this scheme lies in the parallel concatenation of component codes and their iterative decoding algorithm. Although turbo codes have found their way in a number of future wireless communications standards, their efficient implementation in hardware and software is still being actively researched. This study therefore focuses on the design of efficient turbo decoders. The dissertation begins with the description of encoding and decoding of turbo codes. Sliding window implementations of decoding algorithms, which are used to reduce the memory requirements in turbo decoders, are presented. The contribution of this work is the proposed modifications to the conventional sliding window implementations of SOVA, bi-directional SOVA and Max-Log-MAP based turbo decoders. The proposed modifications allow multiple bits to be released in a single decoding window thus reducing the computational complexity and increasing the decoding speed of turbo decoders. A performance and complexity comparison of these decoder implementations is also presented.

Acknowledgements

I would like to acknowledge the excellent supervision of Dr. Ramachandran Venkatesan and Dr. Paul Gillard during this work. This thesis could not have been completed without Dr. Gillard's ideas and Dr. Venkatesan's advice. I would also like to thank the School of Graduate Studies at Memorial University for supporting me financially during my M.Eng. program.

Graduate study at the Faculty of Engineering has been a pleasant and valuable experience for me. I always found CERL (Computer Engineering Research Lab) to be a friendly and inviting place to work. I am especially thankful to my fellow students Atiq, Reza, Sainath and Li Cheng for the interesting academic discussions we had during the course of this work.

Table of Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

3G	Third Generation
3GPP	Third Generation Partnership Project
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
CCSDS	Consultative Committee for Space Data Systems
CD	Compact Disc
DVB	Digital Video Broadcasting
DVD	Digital Versatile Disk
ESA	European Space Agency
FPGA	Field Programmable Gate Array
IMT-2000	International Mobile Telecommunications-2000
LLRs	Log Likelihood Ratios
MAP	Maximum a Posteriori
ML	Maximum Likelihood
NASA	National Aeronautics and Space Agency
RSC	Recursive Systematic Convolutional
SOVA	Soft Output Viterbi algorithm
Bi-SOVA	Bidirectional Soft Output Viterbi algorithm
A	forward path metric for Max-Log-MAP
B	backward path metric for Max-Log-MAP
D_{MAP}	decision depth of trellis for MAP
D_{SOVA}	decision depth of trellis for SOVA
d_{min}	minimum code distance
E_b/N_0	energy per bit to noise density ratio
E_s/N_0	energy per symbol to the noise density ratio
$\boldsymbol{e}$	memoryless noise
$\boldsymbol{k}$	number of inputs in a convolutional encoder
$L(i)$	soft output expressed as log likelihood ratio
L_c	channel reliability value
L_e	extrinsic value
M	SOVA path metric
m	number of memory elements in a convolutional encoder
N	number of bits released in a decoding window
n	number of outputs in a convolutional encoder
r	code rate
T_f	time to build one trellis stage in forward direction
T_t	time to complete SOVA traceback
T_{MAP}	number of trellis stages in the backward recursion
T_{SOVA}	traceback depth of trellis for SOVA
$\boldsymbol{u}$	binary message sequence
$\boldsymbol{v}$	code sequence

w	weight of a codeword
$\boldsymbol{x}$	modulated sequence
$\boldsymbol{y}$	received sequence
α	forward path metric for MAP
β	backward path metric for MAP
γ	branch metric for MAP
Γ	branch metric for Max-Log-MAP
Δ	metric difference
σ^2	variance of zero-mean Gaussian noise

Chapter 1

Introduction

Turbo Codes introduced by Berrou and Glavix in 1993 have revolutionized the field of error correction coding [1]. This powerful error correction technique is ideal for communications systems where significant power saving is required or the operating signal to noise ratio is very low. Wireless communications, with its rapid growth and ever increasing demand for transmission bandwidth, is its foremost candidate. Turbo codes therefore have already been selected for a number of wireless communications standards.

This chapter begins by providing a brief history of the error correcting codes. Section 1.2 discusses the error correcting codes used in wireless communications. Section 1.3 focuses on the implementation of these codes. The remaining sections present the purpose and overview of this dissertation.

1.1 A Brief History of Error Correcting Codes

Information transmitted in a communications system is always liable to errors due to channel impairments. To preserve the accuracy of the information during transmission error correcting codes are used. These codes are also called channel codes. Error correcting codes add structural redundancy to the source information prior to its transmission. This redundancy is then exploited at the receiver to detect and correct transmission errors in the received information. All modern error correcting techniques can be traced back to the ground-breaking work of Shannon [2], Hamming [3] and Golay [4]. While Shannon's work laid the theoretical basis of coding, Hamming and Golay developed the first practical error control schemes.

Shannon, in his pioneering paper in 1948, [2] introduced the concept of source entropy and channel capacity. He mathematically defined source entropy as the average amount of information in a source message and channel capacity as the maximum rate at which the information can be transmitted over this channel. He then showed that it was possible to achieve reliable communications over a noisy channel if the source entropy is lower than that channel's capacity. This remarkable result proved that it is not the accuracy with which the information can be transmitted that is limited, but the rate at which it can be transmitted error free. While Shannon established the channel capacity as the upper limit on transmission rate, he never explicitly stated how it can be practically reached. His channel coding theorem only guaranteed the existence of codes which can be used to achieve transmissions at channel capacity. The history of error correcting codes since 1948 can therefore be characterized as the quest for this Holy Grail: The Shannon Limit.

Hamming discovered the first error correcting code while he was working at the Bell Labs. His code called Hamming code was a great achievement; however it was inefficient and required three check bits to protect four data bits. The deficiencies in Hamming codes were addressed by Golay who discovered two more powerful and significant codes: Binary Golay codes and Tertiary Golay codes [5]. The work by Hamming and Golay laid the foundations of coding theory.

1.1.1 Block Codes

The codes discovered by both Hamming and Golay grouped information symbols into blocks of length k and then added n-k check symbols to each block to obtain n-symbol code words. These types of codes are referred to as block codes. Golay codes were soon replaced by more powerful Reed-Muller codes in 1954 [6]. While Hamming and Golay codes were specific in terms of n and k, Reed-Muller codes were a class of binary codes with flexible design parameters. National Aeronautics and Space Agency (NASA) extensively used Reed-Muller codes throughout 1960s and 1970s. Reed-Muller codes were followed by cyclic block codes that had the property that the cyclic shift of a codeword was also a code word [7]. The cyclic property of these codes enabled the design of encoders and decoders with reduced complexity. Bose Chaudhary and Hocquenghem discovered an important subclass of cyclic codes in 1960 [8]. These codes are known as BCH codes. BCH codes were binary codes; however, soon Reed and Solomon extended them to non binary codes [9]. Reed Solomon codes had superior burst error protection but the absence of an efficient decoder prevented their wide spread use in

practical applications. In 1967 Berlekamp introduced an efficient decoding algorithm for Reed Solomon codes [10]. Since then Reed Solomon codes have been extensively in a wide range of applications including Compact Disc (CD) and Digital Versatile Disk (DVD) players.

1.1.2 Convolutional Codes

Block codes were successfully used for error correction however they suffered from certain drawbacks. Block codes required the entire code word to be received before the decoding process can be completed. This resulted in decoding delays. Another major drawback of block codes was their typical hard decision decoders. A hard decision decoder operates on binary channel output whereas a soft decision decoder operates on a continuous valued channel output. Soft decision decoding is more powerful than hard decision decoding because it does not suffer from the sub optimality that results from the quantization of the channel output. Therefore block codes with their hard decision decoders, although suitable for benign channels, were not ideal for noisy channels.

To address these drawbacks of block codes, Elias introduced a new coding approach called convolutional coding in 1955 [11]. Instead of segmenting the data in blocks and adding redundancy to each block, convolutional codes add redundancy to a continuous stream of data using linear shift registers. The data at the decoder can therefore be decoded continuously with low latency. Another advantage of convolutional codes is that they can be decoded using soft decision decoders. Convolutional codes were more powerful than block codes however it was only after the discovery of a practical and

optimal decoding algorithm by Viterbi in 1967 that they began to see extensive application in communications systems [12]. Convolutional codes were used by NASA in deep space probes such as Voyager and Pioneer [5]. They are also used in second generation digital cellular standards such as GSM and have also been incorporated in future standards such as Third Generation (3G) wireless systems [13].

1.1.3 Concatenated Codes

Concatenated codes are formed by the concatenation of two codes separated by an interleaver. The role of the interleaver is to rearrange the information to provide protection against burst errors. There are two types of concatenated codes: serial concatenated codes and parallel concatenated codes. In serial concatenated codes the output of one encoder is interleaved and then encoded by the second encoder. This technique allows the use of different codes that complement each other. For example Reed Solomon codes with good performance at low noise can be combined with convolutional codes, which have a better performance at high noise. This coding scheme was proposed by Forney in 1966 and is still used by NASA and European Space Agency (ESA) in deep space communications [14].

1.1.4 Turbo Codes

Parallel concatenation of two or more codes is called turbo coding. Berrou and Glavix introduced turbo codes in 1993. They decoded the code using an iterative decoding algorithm and achieved performance very close to the theoretical Shannon limit. Turbo

codes have outperformed all the previously known coding schemes and therefore are rapidly finding applications in future communications standards. Their superior performance actually comes from iterative decoding, in which component decoders share information to improve their individual decoded estimates. This sharing leads to an improvement in decoding performance with each decoding iteration. Although Berrou and Glavix used parallel concatenation of convolutional codes in their turbo coding scheme, it was soon realized that the iterative decoding technique can also be used to decode concatenated block codes. This led to the iterative soft decision decoding of concatenated block codes and several new decoding schemes based on this technique have been proposed recently [15]. Turbo codes, based on both block and convolutional codes, have finally provided us with the opportunity of designing practical communication systems that can operate very close to the channel capacity.

1.2. Error Correcting Codes for Wireless Communications

The choice of an error correcting scheme in a communications system is determined by the nature of the source information (i.e. type of application) and the type of communications channel. Sources of errors in wireless communications among others include low signal strength, shadowing and multipath fading. The problem of low signal strength is inherent to all the wireless channels. The strength of the received signal decreases as the distance between the receiver and transmitters increase. This is of significant importance in mobile wireless systems. The error correcting schemes for wireless communications must therefore, have good performance at low signal to noise

ratio. Convolutional codes generally outperform block codes at low signal to noise ratio and therefore were preferred for wireless communications. NASA's deep space probes, second generation cellular wireless standards and major commercial satellites used convolutional codes. Another source of errors in wireless communications is shadowing which results in the transmitted signal being completely blocked for a period of time. This causes burst errors in the transmitted information sequence. Convolutional codes, although efficient at low signal to noise ratio, are susceptible to burst errors. To address this problem concatenated codes were employed. NASA and ESA used them in Galileo and Giotto missions respectively. The Second generation GSM standard also uses concatenated codes.

Turbo codes formed by the parallel concatenation of convolutional codes also have the properties desirable in a wireless channel code. Moreover iterative decoding of these codes gives near Shannon limit performance at low signal to noise ratio. Therefore convolutional turbo codes are emerging as the foremost choice of future deep space communications, mobile satellite/cellular communications and microwave links. Some examples are

- Inmarsat's new multimedia service is based on turbo codes that allow the user to communicate with existing Inmarsat 3 spot-beam satellites from a notebook-sized terminal at 64 kbit/s.
- The Third Generation Partnership Project (3GPP) proposal for International Mobile Telecommunications-2000 (IMT-2000) includes turbo codes in the multiplexing and

channel coding specification. The IMT-2000 represents the third generation mobile radio systems worldwide standard. The 3GPP objective is to harmonize similar standards proposals from Europe, Japan, Korea and the United States.

- NASA's next-generation deep-space transponder will support turbo codes and implementation of turbo decoders in the Deep Space Network.
- The new standard of the Consultative Committee for Space Data Systems (CCSDS) is based on turbo codes. The new standard outperforms by 1.5 to 2.8 dB the old CCSDS standard based on concatenated convolutional code and Reed-Solomon code.
- The new European Digital Video Broadcasting (DVB) standard has also adopted turbo codes for the return channel over satellite applications.

1.3. Implementation of Error Correcting Codes

The real success of an error correction scheme depends not only on its power to correct errors but also on its ability of being incorporated into a practical communications system. For example scientists and engineers knew that they can achieve transmission rates close to channel capacity by increasing the length of the block codes, however due to the exponential increase in the decoder complexity it was not feasible to design a practical communications system with long block codes. Similarly several very powerful coding schemes did not find practical applications until decoding algorithms with reasonable complexity were found. Reed Solomon codes were proposed in 1960 but it was only after the discovery of Berlekamp decoder in 1967 that they were adopted in various communications and storage systems. Convolutional codes introduced in 1955

also had to wait for the arrival of Viterbi decoder in 1967 before they gained any practical significance.

The introduction of turbo codes in 1993 revolutionized error correction coding because it was the first scheme that performed close to Shannon limit and was also practical. The real breakthrough therefore was not the parallel concatenation of codes but the powerful iterative decoding technique that was also practical. The iterative decoding of turbo codes, although practical, is still considerably more complex than many decoding schemes used in existing error correction schemes. To decode a block, the turbo decoder must go through several decoding iterations (anywhere from two to fifteen). Therefore each component decoder operates on the block several times making the turbo decoder so much slower. Hence designing turbo decoders which are fast, have reduced complexity, and consume less power is crucial to the success of future wireless networks.

1.4. Purpose of study

The purpose of this study is to investigate turbo coded systems for their implementation in wireless applications. The adoption of turbo codes as the standard channel codes for mobile satellite/cellular communications means their implementation in a range of wireless handheld devices. These devices must operate at fast link speeds, be small in size and weight, and consume little power. Since encoding in convolutional turbo codes is trivial, efficient implementations of turbo decoders are the key to the success of mobile networks. The nature of iterative decoding and the presence of interleavers prevent

parallel computation by component decoders. Increasing the speed of the individual component decoders however, can increase the overall decoding speed of the turbo decoder. This study therefore focuses on increasing the speed of component decoders, and then analyzing its resulting effects on the performance of iterative decoding. The benefits of the techniques studied in this thesis are not limited to mobile systems. They can be used for the efficient implementation of any turbo-coded system.

1.5. Organization of thesis

The rest of the thesis is organized as follows. Chapter 2 begins with an introduction to block codes followed by a detailed description of convolutional and turbo codes. Chapter 3 presents iterative decoding of turbo codes. This includes the description of component decoding algorithms and how they are used in an iterative decoding scheme. We restrict our discussion to convolutional turbo codes as they have better performance at low signal to noise ratio than block turbo codes. Chapter 4 introduces sliding window decoding of turbo codes. It is used to reduce the decoder memory requirements. In this chapter we also introduce multiple bit release sliding window implementations of various component decoders. The proposed multiple bit release implementations increase the speed of component decoders without significant performance degradation. Chapter 5 focuses on the performance and speed analysis of these implementations. Through extensive computer simulations we generate the performance curves and calculate the corresponding speed-ups for multiple bit release implementations of turbo codes. Chapter 6 provides conclusions and suggestions for the further extension of this work.

Chapter 2

An Overview of Error Correcting Codes

2.1 Introduction

In this chapter we provide a general overview of error correcting codes [16]. We shall divide these codes into three categories: block codes, convolutional codes and turbo codes. Block codes encode k bit information blocks into n bit coded blocks and are based rigorously on finite field arithmetic and abstract algebra. Some of the most commonly used block codes are Hamming codes, Golay codes, BCH codes and Reed Solomon codes. Convolutional codes, on the other hand, convert the entire data stream into a single codeword. The encoded bits depend not only on the current input bits but also on the previous input bits. They are widely used in real time communication. Finally we use the concepts and terminologies from the block and convolutional codes to explain turbo codes, which can be defined as a parallel concatenation of two component codes separated by a random interleaver.

2.2 Block Codes

A block code has n bit code words that contain k information bits and r parity bits, such that $n = k + r$. Such a code is referred to as an (n, k) block code where n and k are respectively the block length and information length of the code. The total number of code words in an (n, k) block code is 2^k and the rate of the code is $r = k/n$.

2.2.1 Encoding of Block Codes

The encoding process consists of breaking up the data into message blocks $\boldsymbol{m}$ of length k and then performing a one to one mapping of each message m_i to a block of length n called a code word x_i. For linear codes this process can be described by a matrix multiplication

$$x = m\,G\,, \tag{2.1}$$

where G has dimensions $k \times n$ and is called the generator matrix. Linear codes have the property that sum of two code words is also a codeword and thus all linear code words must contain the all-zero codeword. Following are a few important definitions regarding block codes.

Hamming distance: The Hamming distance $v(x_i, x_j)$ between the two code words x_i and x_j is the number of bit positions in which the two code words differ.

Minimum distance: The minimum distance d_{min} of a code is the minimum distance between any two code words.

$$d_{\min} = \min_{i \neq j} v(x_i, x_j)\,. \tag{2.2}$$

A code with minimum distance d_{min} is capable of correcting all code words with t or less errors, where

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor . \tag{2.3}$$

Hamming weight: Hamming weight $w(x)$ of a code word x is the Hamming distance between itself and the all-zero codeword

$$w(x) = v(x,0), \tag{2.4}$$

where 0 or x_0 represents the zero code word. The Hamming weight can be found by counting the number of ones in the codeword. For linear codes the minimum distance is the smallest Hamming weight of all code words except the all zero word.

$$d_{\min} = \min_{x \neq 0} w(x) . \tag{2.5}$$

2.2.2 Cyclic Codes

A code is cyclic if any cyclic shift of a code word produces another code word. A code C is cyclic if for every code word $x = (x_0, x_1, \ldots..x_{n-2}, x_{n-1}) \in C$ there is also a code word $x' = (x_{n-1}, x_0, x_1, \ldots..x_{n-2}) \in C$. The generator matrix of a cyclic code can be expressed in the form

$$G = \begin{bmatrix} g_0 & g_1 & \cdots & g_{n-k} & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & \cdots & g_{n-k} & \cdots & 0 \\ \vdots & & \cdots & \cdots & \cdots & \cdots & \vdots \\ 0 & \cdots & 0 & g_0 & g_1 & \cdots & g_{n-k} \end{bmatrix} . \tag{2.6}$$

A cyclic code can also be represented by a generator polynomial

$$g(D) = g_0 + g_1 D + g_2 D^2 + \cdots\cdots + g_{n-k} D^{n-k} . \tag{2.7}$$

Similarly the message m and the codeword x can also be represented by polynomials and the encoding process becomes the polynomial multiplication $x(D) = m(D)g(D)$. The polynomial multiplication can be implemented by a linear shift register network and thus the encoders for cyclic codes are extremely simple. Figure 2.1 shows a cyclic encoder for generator polynomial $g(D) = 1+D+D^2$.

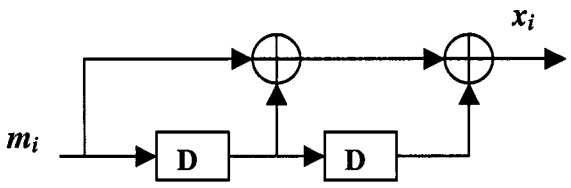


Figure 2.1. Cyclic encoder for $g(D) = 1+D+D^2$

2.3. Convolutional Codes

Convolutional codes are one of the most widely used channel codes in practical communication systems such as satellite communications, cellular mobile, digital video broadcasting etc. A convolutional encoder operates on a source data stream using a sliding window and generates a continuous stream of encoded symbols. Unlike an (n, k) block code where the n bit output of an encoder depends solely on k input bits, the n bit output of a convolutional encoder is constructed from the k bit input as well as $\boldsymbol{m}$ previous inputs. A convolutional code that generates n outputs from k inputs and $\boldsymbol{m}$ previous inputs is referred to as an (n, k, m) convolutional code.

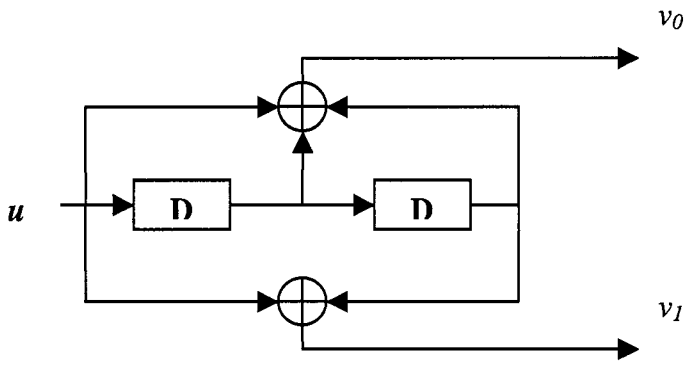


Figure 2.2. A (2, 1, 2) convolutional encoder

2.3.1 Convolutional Encoder Structure

A convolutional code introduces redundant bits in the data stream through the use of linear shift registers. The encoder of an *(n, k, m)* convolutional code consists of a bank of k linear shift registers. Depending on the number of shift registers a convolutional code can become very complicated and therefore we shall restrict our discussion to convolutional encoders with only one shift register: binary convolutional codes. Figure 2.2 shows a (2, 1, 2) convolutional encoder.

The code rate r of a convolutional code is defined as

$$r = \frac{k}{n}, \tag{2.8}$$

where k is the number of parallel input information bits and n is the number of parallel output encoded bits at any time interval. The constraint length K of a convolutional encoder is defined as

$$K = m + 1, \tag{2.9}$$

where m is the maximum number of stages in a shift register. The shift register stores the state of the convolutional encoder. The term constraint length refers to the number of previous bits on which the current output depends. The encoder shown in Figure 2.2 has a code rate of 1/2 and a constraint length of 3.

2.3.2 Systematic Convolutional Encoder

Systematic convolutional encoders encode in such a way that the unmodified input information stream is contained in the encoded output data sequence. This provides the systematic encoders with a significant advantage over nonsystematic encoders: the message is displayed in the encoded sequence and can be read directly from the received sequence thus eliminating the need for an inverter which is required if a nonsystematic encoder is used. Furthermore, the inverter for a nonsystematic code may not exist in which case a finite number of channel errors may cause an infinite number of decoding errors. Such codes are referred to as catastrophic codes. Systematic codes on the other hand do not require inverters and can never be catastrophic.

2.3.3 Recursive Systematic Convolutional (RSC) Encoder

A recursive systematic convolutional encoder can be obtained from a nonrecursive nonsystematic encoder by feeding back one of its encoded outputs to its input. The RSC encoder shown in Figure 2.3 is obtained from the nonrecursive nonsystematic encoder of Figure 2.2 by feeding back one of its outputs.

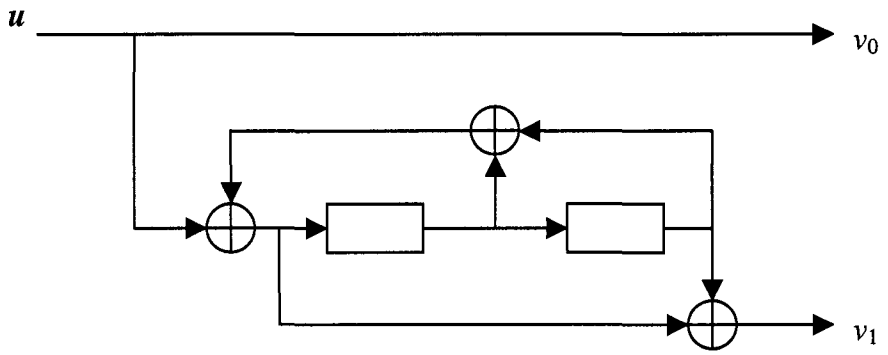


Figure 2.3. A (2, 1, 2) RSC convolutional encoder

A recursive convolutional encoder tends to produce codewords with increased weight relative to a nonrecursive encoder. This results in fewer codewords with lower weights which leads to better error performance. For example consider an input sequence u=(...,0,1,0,0,0,0,0,0,0,0,...) containing a single 1 to a nonrecursive convolutional encoder. The encoder will emerge and then go back to an all-zero state within a finite number of transitions. The encoder output will contain finite number of 1s corresponding to the minimum distance of the code. However this input when given to a recursive encoder will result in a 1 eternally cycling through the encoder shift register. This will repeatedly produce 1s in the encoder output stream resulting in a codeword of increased weight. The recursive and systematic nature of RSC encoders thus provide significant advantages which justify use of RSC encoders in many communication systems.

2.3.4 Convolutional Encoder Representations

A convolutional encoder can be represented in several different but equivalent ways:

1. Generator representation
2. State diagram representation
3. Trellis diagram representation.

2.3.4.1 Generator Representation

A Generator representation shows the hardware connection of the shift register taps to the modulo-2 adders. For an RSC encoder a feed-forward as well as a feedback polynomial is specified for each output. For example the RSC encoder of Figure 2.3 can be represented in generator form as

$$G(D)=\left[1,\frac{1+D^2}{1+D+D^2}\right], \tag{2.10}$$

where 1 corresponds to the systematic output v_0 and $1+D^2/1+D+D^2$ corresponds to the output v_1 with $1+D^2$ being its feed-forward and $1+D+D^2$ the feedback polynomial.

2.3.4.2 State Diagram Representation

The state diagram of a convolutional encoder is a graph that consists of nodes representing the encoder states, and directed lines representing the state transitions. Figure 2.4 shows the state diagram of the RSC encoder of Figure 2.3. The state of the encoder is defined as the contents of its shift register, and this is also referred to as encoder's memory contents. If m denotes the memory of the encoder then there are 2^m possible states. Each directed line is labeled with an input/output pair. Given the current state of the encoder the information sequence at the input determines the path

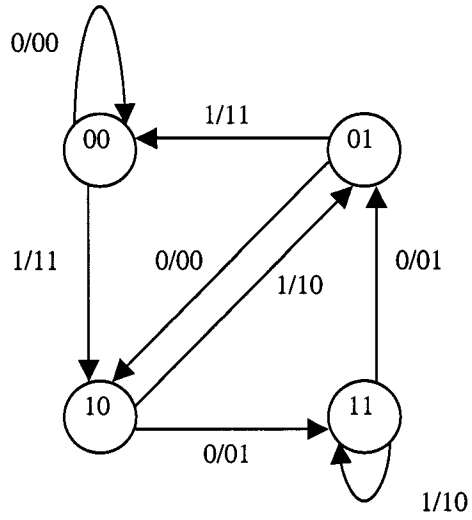


Figure 2.4. State diagram of a (2, 1, 2) RSC encoder

---

through the state diagram and the output sequence. It is customary to begin the convolutional encoding from the all-zero state.

2.3.4.3 Trellis Diagram Representation

A trellis diagram is derived from a state diagram by tracing all the possible input/output sequences and state transitions. The Trellis diagram of the RSC convolutional encoder of Figure 2.3 is shown in Figure 2.5. The black circles at each stage of the trellis represent the four possible states of the encoder and the directed lines represent the transition from one state to the next. The solid lines represent the transition caused by the input symbol 0 and dotted lines represent the transition caused by input symbol 1. We shall assume that the encoder will start from all-zero state and therefore some states in the first two stages

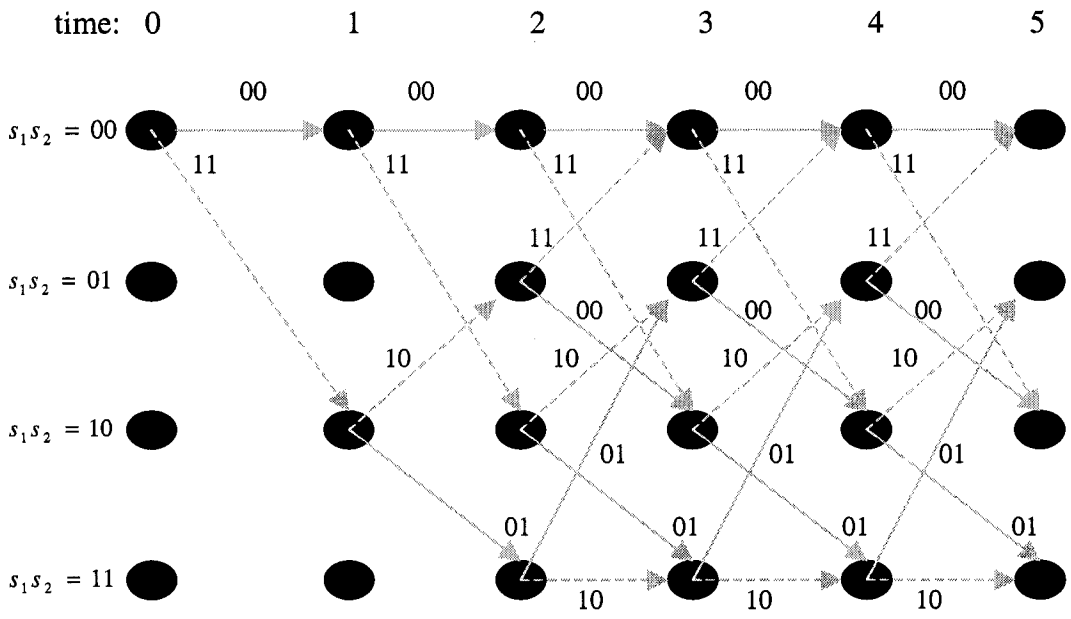


Figure 2.5. Trellis diagram of a (2, 1, 2) RSC encoder

---

of the trellis are inaccessible. That is why certain transitions in the initial stages of the trellis are omitted in the trellis diagram. The output of the encoder at each transition is also labeled in the diagram.

2.4 Concatenated Codes

A concatenated code is composed of two separate codes that are combined together to form a larger code. The primary reason for using a concatenated code is to achieve a low error rate with an overall decoder complexity which is less than that required for a single code of corresponding performance. There are two types of concatenated codes: serial concatenated codes and parallel concatenated codes.

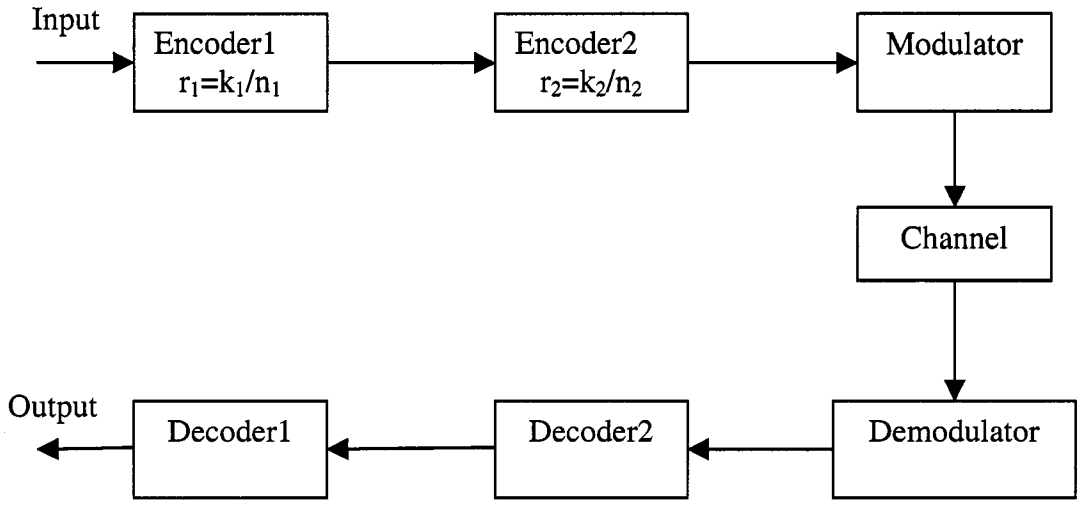


Figure 2.6. Serial concatenated code

2.4.1 Serial Concatenated Codes

The transmission scheme for serially concatenated codes is shown in Figure 2.6. The total code rate for this serial concatenation is

$$r_{tot} = \frac{k_1 k_2}{n_1 n_2}. \tag{2.11}$$

Serial concatenated codes have been used in space communication, with convolutional codes as the inner code and low redundancy Reed Solomon codes as the outer code. Another application of concatenation codes is the concatenation of two convolutional codes where the inner decoder uses a soft-input/soft-output decoding algorithm to produce soft decisions for the outer decoder.

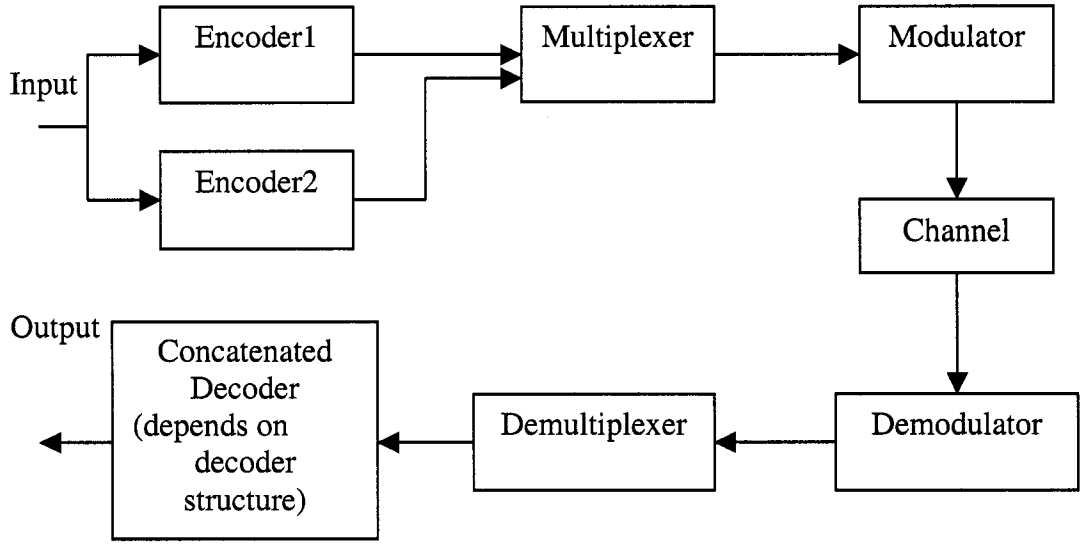


Figure 2.7. Parallel concatenated code

2.4.2 Parallel Concatenated Codes

The transmission scheme for parallel concatenated codes is shown in Figure 2.7. The total code rate for this parallel concatenation is

$$r_{tot} = \frac{k}{n_1 n_2}. \tag{2.12}$$

In both serial and parallel concatenation schemes an interleaver is incorporated between the two codes to decorrelate the received symbols thus increasing the burst error correction capability of the code.

2.5 Turbo Codes

Turbo codes are formed by connecting two identical systematic codes in parallel. These component codes may either be block or convolutional codes and the turbo code formed by their concatenation is referred to as block or convolutional turbo code accordingly. The original turbo codes, presented by Berrou and Glavieux in 1993, used recursive systematic convolutional (RSC) encoders as component encoders and it is customary to refer to these convolutional turbo codes as simply turbo codes. From here onwards we shall also use the term turbo codes to refer to convolutional turbo codes.

2.5.1 A Turbo Encoder

A turbo encoder is formed by the parallel concatenation of two RSC encoders separated by an interleaver. Figure 2.8 shows the diagram of a rate 1/3 turbo encoder obtained through the parallel concatenation of two identical rate 1/2 RSC encoders. The first encoder (RSC Encoder 1) operates on the input sequence u directly. The output of this encoder consists of two sequences v_0 and v_1. The sequence v_0 is identical to the input u since the encoder is systematic. The second sequence v_1 is the parity check sequence calculated by this encoder. The second encoder (RSC Encoder 2) receives an interleaved information sequence denoted by $\tilde{u}$. Only the parity check sequence from the second encoder, denoted by v_2, is transmitted. The information sequence v_0 and the parity check sequences v_1 and v_2 are multiplexed to generate the output of the turbo encoder. This results in an overall code rate of 1/3.

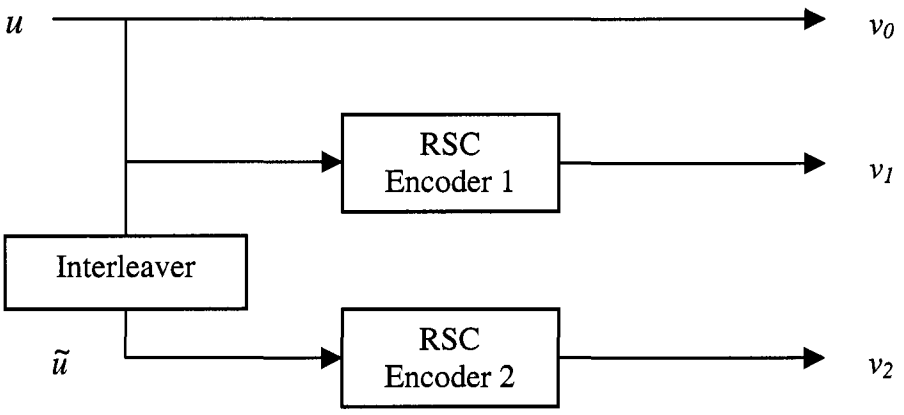


Figure 2.8. A rate 1/3 turbo encoder

2.5.2 Interleaving in Turbo Codes

The interleaver used in turbo codes is a permuter or a scrambler defined by a permutation of L elements with no repetition. The interleaver plays two important roles in turbo codes.

1. It is used to generate a long block code from small memory convolutional encoders. The code block length depends on the interleaver length L.
2. It decorrelates the inputs to the two encoders. This helps in the decoding process, where iterative algorithms based on information exchange between the two component decoders are used. The decorrelated input ensures that there is a high probability that after the correction of some errors in the first decoder some of the remaining errors will be corrected in the second decoder.

In addition to the above two roles, interleavers are also designed to achieve tasks such as increasing the minimum weight of the codewords, termination of both encoders in the all zero state, and puncturing. We will discuss the trellis termination and puncturing in the following subsections.

2.5.3 Trellis Termination in Turbo Codes

Trellis termination means driving the encoder to the all zero state. We drive the encoder to the all zero state at the end of a block to ensure that the initial encoder state for the next block is also the all zero state. For convolutional encoders we terminate the trellis by appending m zero bits, also known as tail bits, at the end of the information block, where m is the memory of the encoder. This strategy however does not work in RSC encoders due to the feedback. The tail bits required in this case depend on the state of the encoder after L information bits where L is the information block length. A simple solution to this problem is shown in Figure 2.9 [17]. After L information bits have been shifted in the encoder the switch is moved from position A to position B for m clock cycles. This drives the encoder to the all zero state. If a pseudorandom interleaver is used, it is highly unlikely that both component encoders in a turbo encoder will terminate in the all-zero state. Therefore only the first encoder is forced to return to the all-zero state and the second encoder is not forced to any particular state. The unknown state of the second encoder results in a performance degradation; however for large interleaver size this degradation is negligible. It is possible to drive both encoders to the all zero state by using a special interleaver such as Block Helical Simile interleaver [18].

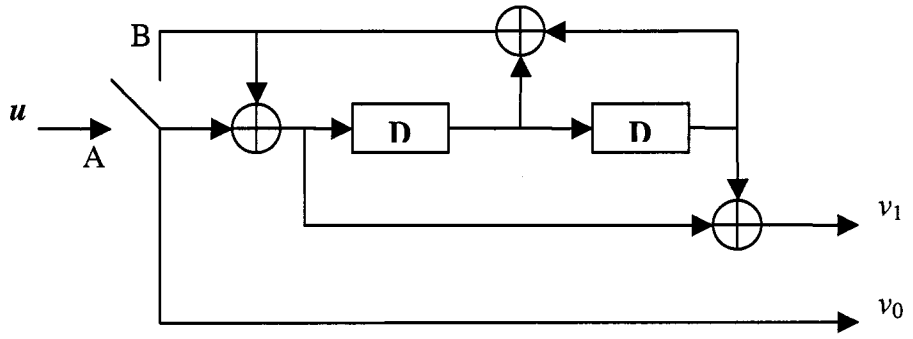


Figure 2.9. Trellis termination strategy for RSC encoder

2.5.4 Punctured Turbo Codes

Puncturing is used in turbo codes to increase the code rate. For example the output of the rate 1/3 turbo encoder shown in Figure 2.8 can be punctured to obtain higher code rates such as 1/2, 2/3, 3/4, 5/6 and so on. During puncturing some output bits of v_0, v_1, and v_2 are deleted according to a chosen pattern defined by a puncturing matrix P. A rate 1/2 turbo code can be obtained from a rate 1/3 code by using the following puncturing pattern

$$P = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{2.13}$$

where the puncturing period is two. The position of zeros indicates the bits from the encoder output that are punctured. In the first cycle v_2 is deleted by the zero in the third row of the first column and similarly in the second cycle v_1 is deleted by the zero in the second row of the second column followed by the deletion of v_2 in the next cycle and so on.

2.5.5 Summary

In this chapter we presented an overview of error correcting codes. We began by reviewing the most commonly used codes, i.e. block codes and convolutional codes. Block codes divide information bits into blocks and then map each information block to a unique code block. Convolutional codes encode the entire information stream into a single word through the use of shift registers. We further illustrated convolutional codes by introducing recursive systematic convolutional (RSC) encoders and various representations of convolutional encoders. This was followed by concatenated codes and finally, an introduction to turbo codes. We used the concepts and terminologies introduced in the preceding sections to define turbo codes and showed that a turbo encoder can be constructed from two RSC encoders separated by an interleaver.

Chapter 3

Iterative Decoding of Turbo Codes

3.1 Introduction

In this chapter we describe the iterative decoding of turbo codes. Turbo codes are formed by the parallel concatenation of two RSC codes. The optimal decoding of such coding schemes is extremely complex. It has also been found that optimal decoding schemes used for turbo codes perform only marginally better than the iterative decoding schemes [19]. Furthermore, several turbo coding schemes, based on iterative decoding, have been found that approach the Shannon limit thus providing an almost optimal performance [20]. Hence, iterative decoding schemes are almost exclusively used in the decoding of turbo codes. We begin this chapter with the description of a communication system model. We then present the general structure of an iterative decoder followed by a detailed description of the component decoders used within this iterative decoder.

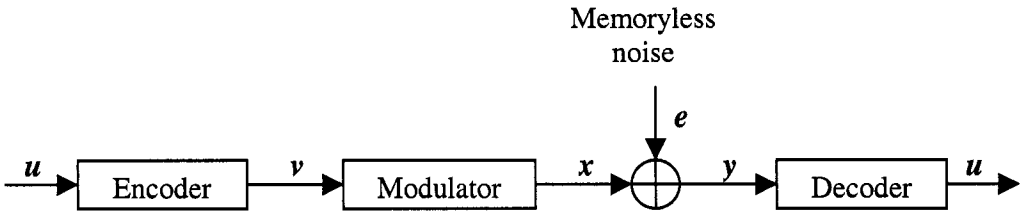


Figure 3.1. System model

3.2 System Model

We shall use the system model shown in Figure 3.1 to illustrate the decoding methods in this chapter. A binary message sequence, denoted by $\boldsymbol{u}$ is given by

$$\boldsymbol{u} = (u_1, u_2, u_3, \ldots\ldots, u_t, \ldots\ldots, u_L), \tag{3.1}$$

where u_t is the message symbol at time t and L is the sequence length. We will assume that all message symbols are generated independently and have equal a priori probabilities. The encoder encodes this binary message and produces a code sequence $\boldsymbol{v}$. The code sequence is then modulated to produce a modulated sequence $\boldsymbol{x}$. The code and modulated sequences are further explained as

$$\boldsymbol{v} = (\underline{v}_1, \underline{v}_2, \underline{v}_3, \ldots\ldots, \underline{v}_t, \ldots\ldots, \underline{v}_L), \tag{3.2}$$

where

$$\underline{v}_t = (v_{t,0}, v_{t,1}, \ldots, v_{t,n-1})$$

is the codeword of length n.

The modulated sequence is represented as

$$\boldsymbol{x} = (\underline{x}_1, \underline{x}_2, \underline{x}_3, \ldots\ldots, \underline{x}_t, \ldots\ldots, \underline{x}_L), \tag{3.3}$$

where

$$\underline{x}_t = (x_{t,0}, x_{t,1}, \ldots, x_{t,n-1})$$

is the modulated codeword of length n and

$$x_{t,i} = 2v_{t,i} - 1, \qquad i = 0,1,\ldots, n-1. \tag{3.4}$$

Equation (3.4) suggests a mapping of coded bit $v_{t,i} = 1$ to $x_{t,i} = +1$ and $v_{t,i} = 0$ to $x_{t,i} = -1$. The modulated sequence is corrupted by additive white Gaussian noise resulting in the received sequence

$$\boldsymbol{y} = (\underline{y}_1, \underline{y}_2, \underline{y}_3, \ldots\ldots, \underline{y}_t, \ldots\ldots, \underline{y}_L), \tag{3.5}$$

where

$$\underline{y}_t = (y_{t,0}, y_{t,1}, \ldots, y_{t,n-1})$$

and

$$y_{t,i} = x_{t,i} + e_{t,i}, \qquad i = 0,1,\ldots, n-1, \tag{3.6}$$

where $e_{t,i}$ is the zero-mean Gaussian noise random variable with zero mean variance σ^2. Each noise sample is independent from the others. The decoder receives the sequence $\boldsymbol{y}$ and after decoding provides an estimate of the input to the encoder. We shall represent the decoded bit at time unit k as u_k, in the description of the decoding methods.

3.3 Iterative Decoder Structure

The general structure of an iterative turbo decoder is shown in Figure 3.2. Two component decoders are linked by interleavers in a way similar to that of the turbo encoder. Each component decoder receives three inputs

1. The systematic channel output (information) bits,

2. The parity bits transmitted by its corresponding component encoder,
3. Information about the concerned bits from the other component decoder. This is also known as a priori information.

The component decoders use the inputs from the channel as well as the a priori information from the other decoder in the decoding process. They also provide the reliability information for each decoded bit. This reliability information, along with the bit estimate, is known as soft output of the component decoder. These soft outputs are typically represented as log likelihood ratios (LLRs). The magnitude of the ratio gives the reliability estimate and the sign, the bit estimate. For example LLR for the value of a decoded bit u_k is given by

$$L(u_k) = \ln\left(\frac{P(u_k = +1)}{P(u_k = -1)}\right) \tag{3.7}$$

where $P(u_k = +1)$ is the probability that bit $u_k = +1$, and $P(u_k = -1)$ is the probability that $u_k = -1$.

In the first iteration, the first component decoder takes channel output as its input and produces soft output. This soft output is the first decoder's estimate of the received data bits and is used as additional information by the second decoder. The second decoder uses this information along with the channel output to produce its own estimate of the data bits. This completes the first iteration. In the second iteration the first decoder again decodes the same channel output but this time it also uses the additional information provided by the output of the second decoder in the first iteration. This additional

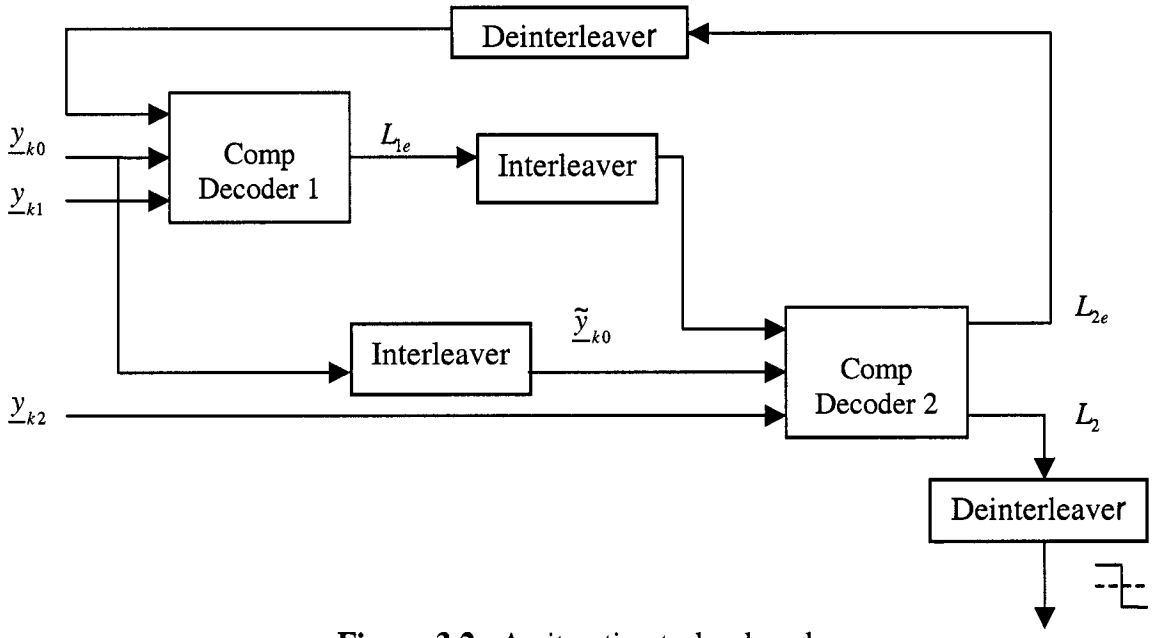


Figure 3.2. An iterative turbo decoder

information helps the first decoder improve its estimate and obtain more accurate soft outputs. These outputs are then used by the second decoder as a priori information and the cycle is repeated. After each iteration, the bit error rate (BER) of the decoded bits tends to fall, however the improvement in performance decreases with increasing iterations. For this reason the number of iterations is usually limited to a small number such as eight [23].

The iterative nature of decoding requires that the same information must not be reused more than once in each decoding step. Therefore, we must subtract any redundant information from the inputs of the component encoders. This leads us to the concept of extrinsic and intrinsic information which we shall consider later in this chapter.

3.4 Component Decoders

The component encoders used in iterative decoding of turbo codes must have the ability to use a priori information as well as provide reliability information for each decoded bit. Two decoders that satisfy the above criteria are MAP (maximum a posteriori algorithm) proposed by Bahl et al. [21], and SOVA (soft output Viterbi algorithm) proposed by Hagenauer and Hoeher [22]. Let us now consider these component decoders in detail.

3.4.1 The Maximum A Posteriori Algorithm (MAP)

The MAP algorithm was introduced by Bahl et al. in 1974 to estimate the a posteriori probabilities of the states and the transitions of a Markov source observed in a memoryless noisy channel. When used to decode convolutional codes, the algorithm is optimal in terms of minimizing decoded BER. It examines every possible path through the convolutional decoder trellis, and was considered infeasibly complex in most applications. For this reason it was largely ignored before the discovery of turbo codes. However due to the iterative decoding used in turbo codes, Berrou et al. employed MAP decoding in their seminal paper on turbo codes.

3.4.1.1 Introduction and Mathematical Preliminaries

Let us now examine the theory behind the MAP algorithm. We shall assume binary coding. For each decoded bit u_k, the MAP algorithm gives the probability that this bit was +1 or -1, given the received symbol $\underline{y}$. This is equivalent to finding the a posteriori LLR $L(u_k|\underline{y})$, where

$$L(u_k \mid \underline{y}) = \ln\left(\frac{P(u_k = +1 \mid \underline{y})}{P(u_k = -1 \mid \underline{y})}\right). \tag{3.8}$$

If $S_{k-1}=`s$ is the previous state and $S_k=s$ is the present state in a trellis, then we can use Bayes' rule and the fact that only one transition between S_{k-1} and S_k could have occurred at the encoder to rewrite the above equation as

$$L(u_k \mid \underline{y}) = \ln\left(\frac{\sum_{u_k=+1}^{(`s,s)\Rightarrow} P(S_{k-1} = `s \wedge S_k = s \wedge \underline{y})}{\sum_{u_k=-1}^{(`s,s)\Rightarrow} P(S_{k-1} = `s \wedge S_k = s \wedge \underline{y})}\right), \tag{3.9}$$

where $(`s, s) \Rightarrow u_k = +1$ is the set of transitions from previous state $`s$ to present state s that can occur if the input bit $u_k = +1$, and similarly for $(`s, s) \Rightarrow u_k = -1$. For brevity we shall write $P(S_{k-1} = `s \wedge S_k = s \wedge \underline{y})$ as $P(`s \wedge s \wedge \underline{y})$.

Let us now consider the individual probabilities in the above equation. We can split the received sequence $\underline{y}$ into three sections: the received codeword associated with the present transition $\underline{y}_k$, the received sequence prior to the present transition $\underline{y}_{j<k}$, and the received sequence after the present transition $\underline{y}_{j>k}$. We can thus write

$$P(`s \wedge s \wedge \underline{y}) = P(`s \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k \wedge \underline{y}_{j>k}). \tag{3.10}$$

Let's assume that the channel is memoryless, i.e. the future received sequence will depend only on the present state and not on the previous state or the present and previous received channel sequences. Now we can use the Bayes' rule $P(a \wedge b) = P(a|b)P(b)$ to rewrite the individual probabilities

$$\begin{aligned} P(`s \wedge s \wedge \underline{y}) &= P(`s \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k \wedge \underline{y}_{j>k}) \\ &= P(\underline{y}_{j>k} \mid s) \cdot P(`s \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k) \end{aligned}$$

$$= P(\underline{y}_{j>k} \mid s) \cdot P(\{\underline{y}_k \wedge s\} \mid `s) \cdot P(`s \wedge \underline{y}_{j<k})$$

$$= \beta_k(s) \cdot \gamma_k(`s, s) \cdot \alpha_{k-1}(`s), \tag{3.11}$$

where

- $\alpha_{k-1}(`s) = P(`s \wedge \underline{y}_{j<k})$ is the probability that trellis is in state $`s$ at time k-1 and the received sequence up to this point is $\underline{y}_{j<k}$.
- $\beta_k(s) = P(\underline{y}_{j>k} \mid s)$ is the probability that trellis is in state s at time k and the future received sequence will be $\underline{y}_{j>k}$.
- $\gamma_k(`s, s) = P(\{\underline{y}_k \wedge s\} \mid `s)$ is the probability that at time k-1 trellis was in state $`s$, it moves to state s at time k and the channel sequence for this transition is $\underline{y}_k$.

Figure 3.3 shows a section of a four state trellis with this split of the received sequence. From Equations (3.10) and (3.11) we can finally write the conditional LLR for u_k

$$L(u_k \mid \underline{y}) = \ln\left(\frac{\sum_{u_k=+1}^{(`s,s)\Rightarrow} \alpha_{k-1}(`s) \cdot \gamma_k(`s, s) \cdot \beta_k(s)}{\sum_{u_k=-1}^{(`s,s)\Rightarrow} \alpha_{k-1}(`s) \cdot \gamma_k(`s, s) \cdot \beta_k(s)}\right). \tag{3.12}$$

The MAP algorithm calculates the α_k(s) and $\beta_k(s)$ for all states throughout the trellis and uses Equation 3.12 to deliver the conditional LLRs. Let us see how the values of α_k(s), $\beta_k(s)$ and $\gamma_k(s)$ are calculated.

3.4.1.2 Forward Recursion and Calculation of $\alpha_k(s)$

From the definition of $\alpha_{k-1}(`s)$ in Equation 3.11, we can write

$$\alpha_k(s) = P(S_k = s \wedge \underline{y}_{j<k+1})$$

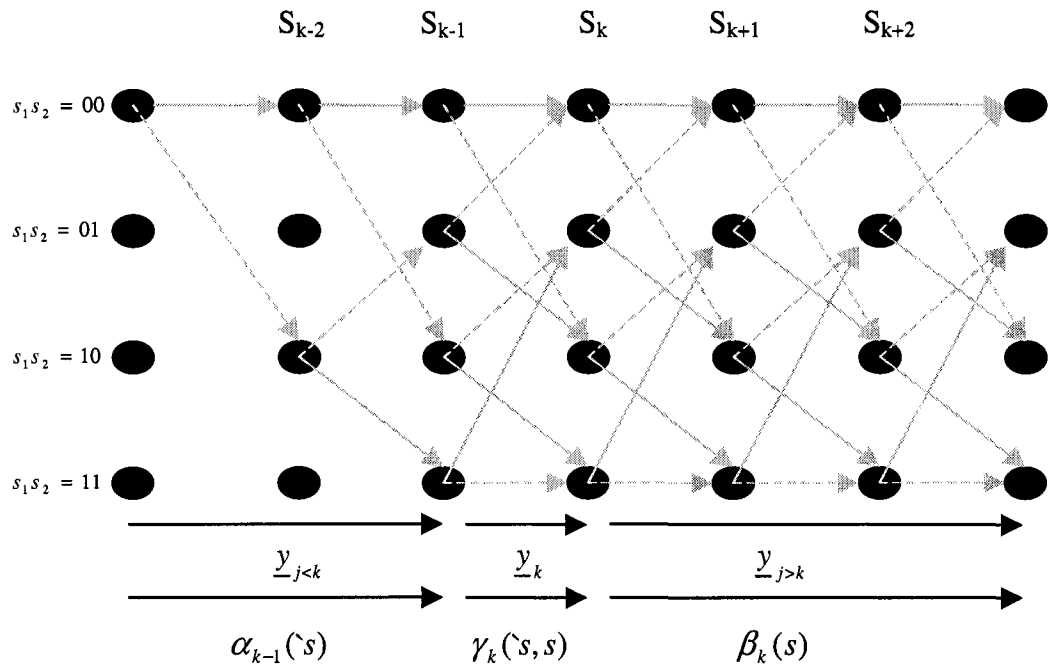


Figure 3.3. MAP decoder trellis for a 4 state RSC code

$$= P(s \wedge \underline{y}_{j<k} \wedge \underline{y}_k)$$

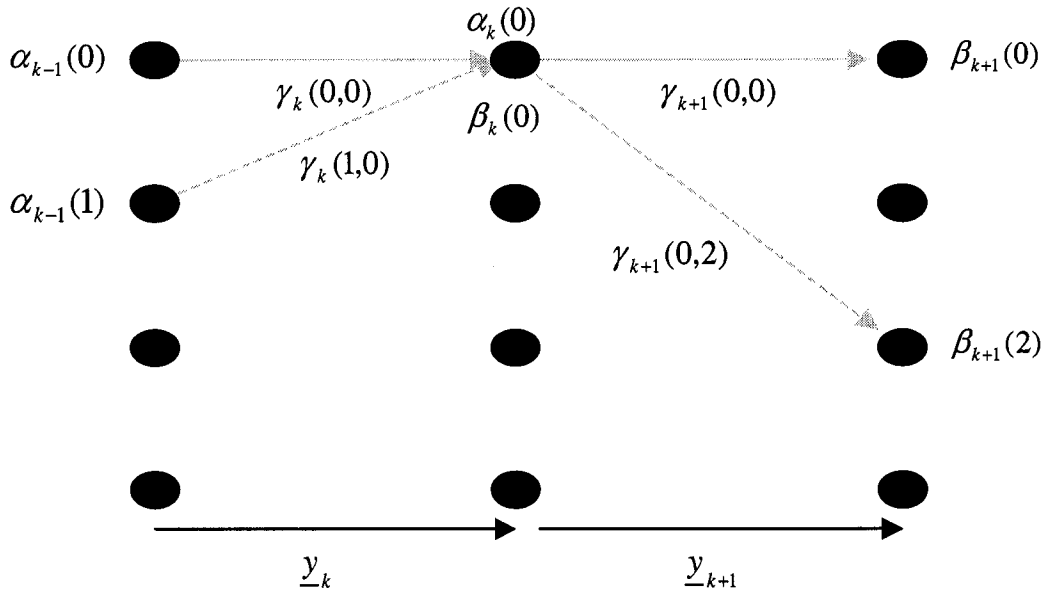
$$= \sum_{all `s} P(s \wedge `s \wedge \underline{y}_{j<k} \wedge \underline{y}_k), \tag{3.13}$$

where in the last step we split the probability $P(s \wedge \underline{y}_{j<k+1})$ into sum of joint probabilities $P(s \wedge `s \wedge \underline{y}_{j<k} \wedge \underline{y}_k)$ over all possible previous states. Again using Bayes' rule and assuming a memoryless channel we can write

$$\alpha_k(s) = \sum_{all `s} P(s \wedge `s \wedge \underline{y}_{j<k} \wedge \underline{y}_k)$$

$$= \sum_{all `s} P(\{s \wedge \underline{y}_k\} | \{`s \wedge \underline{y}_{j<k}\}) \cdot P(`s \wedge \underline{y}_{j<k})$$

$$= \sum_{all `s} P(\{s \wedge \underline{y}_k\} | `s) \cdot P(`s \wedge \underline{y}_{j<k})$$



$$\alpha_k(0) = \alpha_{k-1}(0) \cdot \gamma_k(0,0) + \alpha_{k-1}(1) \cdot \gamma_k(1,0)$$

$$\beta_k(0) = \beta_{k+1}(0) \cdot \gamma_{k+1}(0,0) + \beta_{k+1}(2) \cdot \gamma_{k+1}(0,2)$$

Figure 3.4. Recursive calculation of $\alpha_k(0)$ and $\beta_k(0)$

$$= \sum_{all`s} \alpha_{k-1}(`s) \cdot \gamma_k(`s, s). \tag{3.14}$$

Thus the values for $\alpha_k(s)$ can be calculated easily from the $\gamma_k(`s, s)$ values recursively. Figure 3.4 shows how $\alpha_k(s)$ can be calculated recursively using $\alpha_{k-1}(`s)$ and $\gamma_k(`s, s)$ for a four state RSC code. Notice that because of the binary trellis we only have two previous states that can transit into a current state.

3.4.1.3 Backward Recursion and Calculation of $\beta_k(s)$

The values of $\beta_k(s)$ can also be calculated recursively in a similar manner during a backward recursion. The backward recursion starts at the last stage of the trellis and moves in the reverse direction. Using a derivation similar to that of Equation 3.14 we can show

$$\beta_{k-1}(`s) = \sum_{alls} \beta_k(s) \cdot \gamma_k(`s, s). \tag{3.15}$$

Figure 3.4 again shows the calculation of one $\beta_{k-1}(`s)$ value from $\beta_k(s)$ and $\gamma_k(`s, s)$ recursively.

3.4.1.4 Calculation of $\gamma_k(`s, s)$

Let's consider how we can calculate the transition probability values $\gamma_k(`s, s)$ in Equation 3.11 from the received channel sequence and the a priori information. From Equation 3.11 and Bayes' rule we have

$$\begin{aligned}
\gamma_k(`s, s) &= P(\{\underline{y}_k \wedge s\} \mid `s) \\
&= P(\{\underline{y}_k \mid \{`s \wedge s\}) \cdot P(s \mid `s) \\
&= P(\{\underline{y}_k \mid \{`s \wedge s\}) \cdot P(u_k) \\
&= P(\underline{y}_k \mid \underline{x}_k) \cdot P(u_k),
\end{aligned} \tag{3.16}$$

where u_k is the bit required for the transition from state $`s$ to $s$ and $P(u_k)$ is the a priori probability of this bit. $\underline{x}_k$ is the codeword associated with this transition. Thus the transition probability $\gamma_k(`s, s)$ is given by the product of the a priori probability of the bit required for this transition and the probability that, given the codeword associated with

this transition $\underline{x}_k$ was transmitted, we received sequence $\underline{y}_k$. Assuming a memoryless Gaussian channel with BPSK (binary phase shift keying) modulation, $P(\underline{y}_k \mid \underline{x}_k)$ is given as [23]

$$P(\underline{y}_k \mid \underline{x}_k) = \prod_{l=1}^{n} P(y_{kl} \mid x_{kl}), \tag{3.17}$$

where x_{kl} and y_{kl} are individual bits within the transmitted and received codewords $\underline{y}_k$ and $\underline{x}_k$ respectively and n is the number of these bits in each codeword.

3.4.1.5 Iterative Decoding Using MAP Algorithm

As the first MAP decoder receives the channel values y_{kl}, it uses these values and the a priori LLRs $L(u_k)$ (which are provided by the other component decoder in iterative decoding) to calculate $\gamma_k(`s, s)$ according to Equations 3.16 and 3.17. The $\gamma_k(`s, s)$ are used to calculate $\alpha_k(s)$ recursively from Equation 3.14. This constitutes the forward recursion of the MAP algorithm. Once all the channel values have been received and all $\gamma_k(`s, s)$ have been calculated, backward recursion starts. $\beta_k(s)$ are calculated in the backward recursion according to Equation 3.15. Finally all the calculated values of $\alpha_k(s)$, $\gamma_k(`s, s)$, and $\beta_k(s)$ are used in Equation 3.12 to calculate the values of a posteriori LLRs $L(u_k \mid \underline{y})$.

In iterative decoding the output of the first MAP decoder provides the a priori probabilities for the second MAP decoder. However these probabilities should come from an independent source. Recall that we calculated the a posteriori LLRs $L(u_k \mid \underline{y})$ from a

priori LLRs $L(u_k)$ and the received sequence y_{kl}. $L(u_k)$ was provided by the other decoder and y_{kl} consisted of systematic bits y_{ks}, common to both MAP decoders. Therefore in order to provide the second decoder with independent a priori knowledge we must subtract the effect of the above two terms from the a posteriori LLRs of the first decoder. It can be shown [1] that, for a systematic code, output of the MAP decoder given by Equation 3.12 can be re-written as

$$L(u_k \mid \underline{y}) = \ln\left(\frac{\sum_{u_k=+1}^{(`s,s)\Rightarrow} \alpha_{k-1}(`s)\cdot\gamma_k(`s,s)\cdot\beta_k(s)}{\sum_{u_k=-1}^{(`s,s)\Rightarrow} \alpha_{k-1}(`s)\cdot\gamma_k(`s,s)\cdot\beta_k(s)}\right)$$

$$= L(u_k) + L_c(y_{ks}) + L_e(u_k), \tag{3.18}$$

where $L(u_k)$ is the a priori LLR given by(1), and L_c is the channel reliability value and y_{ks} is the received version of the transmitted systematic bit $x_{ks} = u_k$. L_c is further explained as

$$L_c = \frac{4a}{2\sigma^2}, \tag{3.19}$$

where σ^2 is the variance and a is the fading amplitude of the noise. For zero-mean Gaussian noise, a=1.

The final term $L_e(u_k)$ is derived from the a priori information sequence $L(u_n)$ and the received channel information sequence $\underline{y}$ excluding the received systematic bit y_{ks} and the a priori information $L(u_k)$ for the bit u_k. So it is called the extrinsic LLR of the bit u_k. It is this extrinsic LLR that is passed on to the second decoder as its a priori LLR.

Now we are ready to summarize the iterative decoding with MAP decoders. The first MAP decoder receives the channel sequence which consists of both the systematic and the parity bits. It uses this sequence and the a priori information available to calculate its

estimate of the conditional LLRs of data bits u_k, k=1,2..N. Note that in the first iteration of the first component decoder there is no a priori information available, and hence $P(u_k)$ in Equation 3.16 will be 0.5. The extrinsic information $L_e(u_k)$ is then calculated from Equation 3.18 which is then passed on to the second decoder. Next the second MAP decoder starts its operation. It receives the channel sequence containing the systematic bits and the interleaved parity bits from the second encoder. The second decoder uses this sequence and the a priori information (interleaved extrinsic information $L_e(u_k)$ from the first decoder) supplied by the first decoder to calculate its own estimate of the conditional LLRs. This completes one iteration of the turbo decoder. In the second iteration the first decoder again processes its received channel sequence but this time its a priori information is provided by the extrinsic value of the second decoder calculated in the first iteration. This results in improved estimates of a posteriori LLRs by the first decoder. The second iteration continues with second decoder using a priori values derived from the improved a posteriori LLRs of the first decoder to improve its own estimates. The iterative process proceeds and with each iteration, BER of the decoded bits falls. However the improvement in performance diminishes with increasing number of iterations. Therefore for time and computational complexity reasons, the number of iterations is usually limited to a small number such as eight. At the end of the iterative process the a posteriori LLRs are taken from the second decoder and a hard decision is made on the received sequence.

3.4.2. The Max-Log-MAP Algorithm

The MAP algorithm as described above is extremely complex due to the multiplications involved in the recursive calculation for $\alpha_k(s)$ in Equation 3.14 and $\beta_{k-1}(`s)$ in Equation 3.15. However its complexity can be dramatically reduced without affecting its performance. This is done by transferring the recursions in to the log domain and invoking an approximation to reduce its complexity. Max-Log-MAP was initially proposed by Koch and Baier [24]. It simplifies the MAP algorithm by transferring the calculation of $\alpha_k(s)$, $\beta_k(s)$, and $\gamma_k(`s, s)$ into the log arithmetic domain and then using the approximation

$$\ln\left(\sum_i e^{x_i}\right) \approx \max_i(x_i). \tag{3.20}$$

Let us define $A_k(s)$, $B_k(s)$ and $\Gamma_k(`s, s)$ as

$$A_k(s) \triangleq \ln(\alpha_k(s)) \tag{3.21}$$

$$B_k(s) \triangleq \ln(\beta_k(s)) \tag{3.22}$$

$$\Gamma_k(`s, s) \triangleq \ln(\gamma_k(`s, s)). \tag{3.23}$$

Now we can re-write Equation 3.14 as

$$\begin{aligned} A_k(s) &\triangleq \ln(\alpha_k(s)) \\ &= \ln\left(\sum_{all`s} \alpha_{k-1}(`s)\gamma_k(`s,s)\right) \\ &= \ln\left(\sum_{all`s} \exp[A_{k-1}(`s) + \Gamma_k(`s,s)]\right) \\ &\approx \max_{`s}(A_{k-1}(`s) + \Gamma_k(`s,s)). \end{aligned} \tag{3.24}$$

Equation 3.24 implies that for each path coming into a state $S_k=s$ at the present stage of the trellis, the algorithm adds a branch metric term $\Gamma_k(`s, s)$ to the previous $A_{k-1}(`s)$ to find the new $A_k(s)$ value for that path. Since there are two paths coming in to a present state the path with the higher branch metric is selected. This is known as the 'survivor' path and all the other paths are discarded. This value of $A_k(s)$ then gives the natural logarithm of the probability that trellis is in state $S_k=s$ at stage k given the received sequence up to this point is $\underline{y}_{j<k}$. Note that because of the approximation of Equation 3.20 we only considered one path, the Maximum Likelihood (ML) path, in the calculation of this probability. Therefore the algorithm gives us the probability of the most likely path through the trellis to the present state $S_k=s$ and not the probability of any path through the trellis to state $S_k=s$. Therefore Max-Log-MAP algorithm is a sub-optimal algorithm when compared to the original MAP algorithm.

Similar to the forward recursion of Equation 3.24, for backward recursion we can write

$$B_{k-1}(`s) \approx \max_{s}(B_k(s) + \Gamma_k(`s, s)), \tag{3.25}$$

and for the branch metrics $\Gamma_k(`s, s)$ we can write

$$\Gamma_k(`s, s) \triangleq \ln(\gamma_k(`s, s))$$

$$= C + \frac{1}{2}u_k L(u_k) + \frac{L_c}{2}\sum_{l=1}^{n} y_{kl}x_{kl}\,, \tag{3.26}$$

where C is a constant and can be omitted and the term $\sum_{l=1}^{n} y_{kl}x_{kl}$ is weighted by the channel reliability value L_c of Equation 3.19.

Finally we substitute the above approximations into our Equation 3.12 to obtain a posteriori LLRs that the Max-Log-MAP algorithm calculates.

$$L(u_k \mid \underline{y}) = \ln\left(\frac{\sum_{u_k=+1}^{(`s,s)\Rightarrow} \alpha_{k-1}(`s)\cdot\gamma_k(`s,s)\cdot\beta_k(s)}{\sum_{u_k=-1}^{(`s,s)\Rightarrow} \alpha_{k-1}(`s)\cdot\gamma_k(`s,s)\cdot\beta_k(s)}\right)$$

$$\approx \max_{\substack{(`s,s)\Rightarrow \\ u_k=+1}} (A_{k-1}(`s)+\Gamma_k(`s,s)+B_k(s)) - \max_{\substack{(`s,s)\Rightarrow \\ u_k=-1}} (A_{k-1}(`s)+\Gamma_k(`s,s)+B_k(s)). \tag{3.27}$$

Equation 3.27 means that in order to calculate the a posteriori LLR $L(u_k \mid \underline{y})$ of the bit u_k, the algorithm considers every transition from trellis stage S_{k-1} to S_k and then groups them into two categories: transitions which might have occurred if $u_k = +1$ and those which might have occurred if $u_k = -1$. It then selects the best transition in each group and the a posteriori LLRs are found based on these 'best' transitions.

3.4.3 The Log-MAP Algorithm

The Max-Log-MAP algorithm gives a slightly degraded performance due to the approximation of Equation 3.20 [25]. However this approximation can be made exact using the Jacobian logarithm

$$\begin{aligned}\ln(e^{x_1}+e^{x_2}) &= \max(x_1+x_2)+\ln(1+e^{-|x_1-x_2|}) \\ &= \max(x_1+x_2)+f_c(|x_1-x_2|) \\ &= g(x_1+x_2),\end{aligned} \tag{3.28}$$

where $f_c(x)$ can be thought of as a correction term. Therefore in the Log-Map algorithm the values of $A_k(s)$ and $B_k(s)$ are calculated using forward and backward recursions similar to the Max-Log-MAP, however the maximization in Equations 3.24 and 3.25 is complemented by the correction terms of Equation 3.28. These correction terms can be stored in a look up table. Consequently although it is only slightly more complex than the Max-Log-MAP algorithm, the Log-MAP algorithm gives exactly the same performance as the MAP algorithm.

3.4.4 The Soft Output Viterbi Algorithm (SOVA)

The Soft Output Viterbi Algorithm (SOVA) is a variation of the classical Viterbi algorithm [26]. The classical Viterbi algorithm is a hard decision algorithm and can not be employed in iterative decoding. Hagenauer et al. proposed two modifications, which enable a Viterbi decoder to accept a priori information as well as provide soft outputs for each decoded bit, thus fulfilling the requirements for iterative decoding [22]. Let us now consider the SOVA algorithm in detail.

3.4.4.1 Forward Recursion in SOVA

Given the received sequence $\underline{y}$, the forward recursion in both classical Viterbi and SOVA algorithms is meant to find the Maximum Likelihood (ML) path through the trellis. Let us define

- $\underline{s}_k^s$ to be a path through the trellis i.e. $\underline{s}_k^s$ is the state sequence which gives the states along this path at state $S_k=s$ and stage k in the trellis.

- $\underline{y}_{j\le k}$ to be the received channel sequence up to and including the stage k in the trellis.

The probability of a path $\underline{s}_k^s$ being correct can then be defined as

$$p(\underline{s}_k^s \mid \underline{y}_{j\le k}) = \frac{p(\underline{s}_k^s \wedge \underline{y}_{j\le k})}{p(\underline{y}_{j\le k})}, \tag{3.29}$$

where $p(\underline{y}_{j\le k})$ is constant for all the paths $\underline{s}_k$ through the trellis to stage k. The probability that path $\underline{s}_k^s$ is correct is then directly proportional to $p(\underline{s}_k^s \wedge \underline{y}_{j\le k})$. Therefore the path through the trellis for which this probability is the highest is our ML path.

Similar to the calculation of $\alpha_k(s)$, of Equation 3.14, in the forward recursion of MAP, we need a path metric for the forward recursion of SOVA. It is obvious that this metric must maximize $p(\underline{s}_k^s \wedge \underline{y}_{j\le k})$ and should be computable in a recursive manner. Let us see how this metric can be derived. If a path $\underline{s}_k^s$ at stage k of the trellis contains a path $\underline{s}_{k-1}^{`s}$ at stage $k$-1 of the trellis then we can use the definition of $\gamma_k(`s, s)$ in Equation 3.11 to write

$$\begin{aligned} p(\underline{s}_k^s \wedge \underline{y}_{j\le k}) &= p(\underline{s}_{k-1}^{`s} \wedge \underline{y}_{j\le k-1}) \cdot p(s \wedge \underline{y}_k \mid `s) \\ &= p(\underline{s}_{k-1}^{`s} \wedge \underline{y}_{j\le k-1}) \cdot \gamma_k(`s, s). \end{aligned} \tag{3.30}$$

Now we can define our metric $M(\underline{s}_k^s)$ as

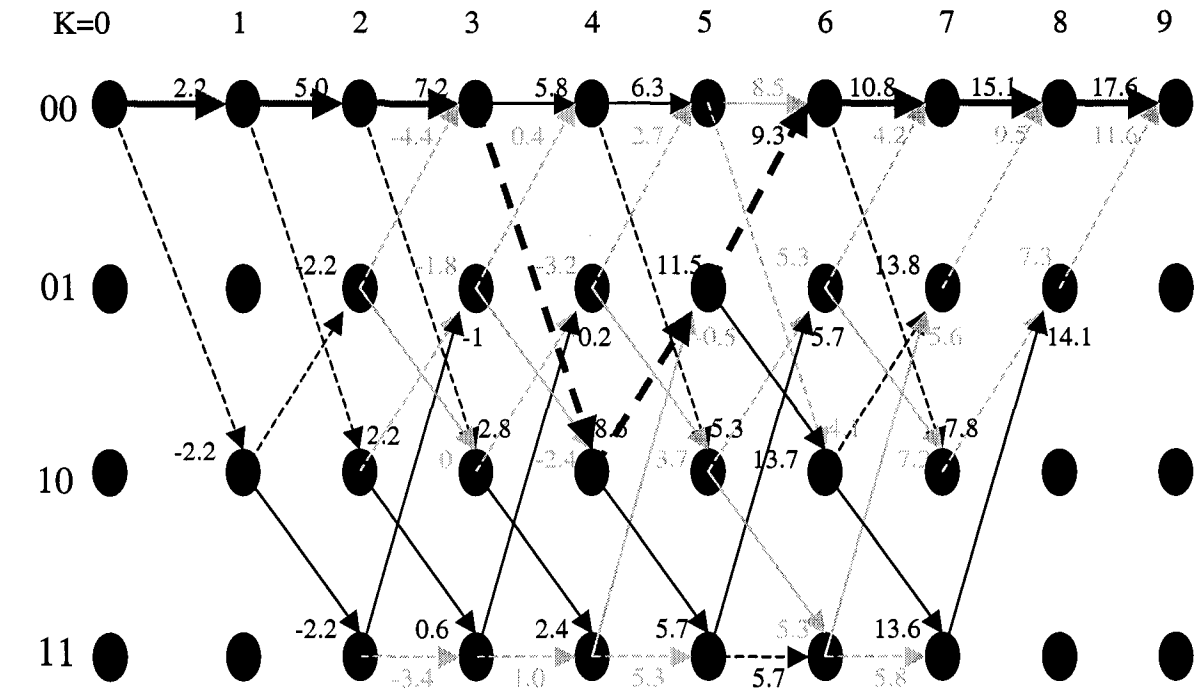
$$\begin{aligned} M(\underline{s}_k^s) &\triangleq \ln(p(\underline{s}_k^s \wedge \underline{y}_{j\le k})) \\ &= M(\underline{s}_{k-1}^{`s}) + \ln(\gamma_k(`s, s)). \end{aligned} \tag{3.31}$$

Using Equation 3.26 and omitting the constant term we can write

$$M(\underline{s}_k^s) = M(\underline{s}_{k-1}^{\hat{s}}) + \frac{1}{2}u_k L(u_k) + \frac{L_c}{2}\sum_{l=1}^{n} y_{kl}x_{kl} \,. \tag{3.32}$$

This is our SOVA path metric which can be calculated recursively and also includes the a priori term $u_k L(u_k)$. We should mention here that the path metric used in the classical Viterbi algorithm does not contain this a priori term. Notice that forward recursion in SOVA is identical to the forward recursion of Max-Log-MAP algorithm, i.e. calculation of $A_k(s)$. Therefore during the forward recursion, path metrics of all the paths coming into a state are computed. The path with the highest metric is then selected and the remaining paths are discarded. This path is known as the 'survivor' path. Once the entire trellis has been built, we take the survivor with the highest metric at the last stage of the trellis and designate it as the ML path. Figure 3.5 shows the forward recursion in SOVA algorithm.

As the channel sequence is received, path metrics are calculated using Equation 3.32. A solid arrow denotes a transition resulting from a -1 input bit, and a dashed arrow represents an input bit of +1.We show the survivor paths and their corresponding metrics in black, and discarded paths and their corresponding metrics in grey. The absence of certain branches in the last two stages of the trellis indicates that we forced the trellis to terminate in the all-zero state. The ML path is indicated with bold arrows. Once we find the ML path we can make hard decisions on the received sequence. The input bits required for the transitions along the ML path give the decoder estimate of the received information sequence. This constitutes the output of a classical Viterbi decoder.



Received: (-2.1,-0.1) (-1.4,-1.4) (-1.7,-0.5) (0.9,0.5) (1.2,-1.7) (-1.1,-1.1) (-0.7,-0.8) (-2.4, -1.9) (-1.6, -0.9)

Figure 3.5. Forward recursion in SOVA decoding

3.4.4.2 SOVA Traceback

The SOVA metric defined in Equation 3.32 takes into account the a priori information $u_k L(u_k)$, however to satisfy the requirements of iterative decoding, the algorithm must also provide soft outputs. For the binary trellis of Figure 3.5, forward recursion calculates metrics for both paths merging into every state and then rejects the path with the lower metric. If the two paths $\underline{s}_k^s$ and $\underline{\hat{s}}_k^s$ reaching a state S_k=s have metrics $M(\underline{s}_k^s)$ and $M(\underline{\hat{s}}_k^s)$ and $\underline{s}_k^s$ is the survivor path with the higher metric then we can define the metric difference as

$$\Delta_k^s = M(\underline{s}_k) - M(\hat{\underline{s}}_k) \geq 0. \tag{3.33}$$

The probability that we made the correct decision when we selected $\underline{s}_k^s$ and rejected $\hat{\underline{s}}_k^s$ is given by

$$P(\textit{correct decision at } S_k{=}s) = \frac{P(\underline{s}_k^s)}{P(\underline{s}_k^s) + P(\hat{\underline{s}}_k^s)}. \tag{3.34}$$

From our metric definition in Equation 3.31 we can write

$$P(\textit{correct decision at } S_k{=}s) = \frac{e^{M(\underline{s}_k^s)}}{e^{M(\underline{s}_k^s)} + e^{M(\hat{\underline{s}}_k^s)}} = \frac{e^{\Delta_k^s}}{1 + e^{\Delta_k^s}}, \tag{3.35}$$

and the Log Likelihood ratios, LLRs are simply given by Δ_k^s. Let us see how we can find the LLRs which give the reliability of the bit decisions along the ML path. Figure 3.6 shows a simplified version of the trellis shown in Figure 3.5. We show the ML path in black whereas the grey colored paths are the discarded paths that merge with the ML path. In order to determine the reliability of the bits given by ML path we consider the probability that the paths merging with the ML path were incorrectly discarded. This can be done by considering the metric difference $\Delta_k^{s_i}$ for all states s_i along the ML path. It is shown by Hagenauer in [27] that this LLR can be approximated by

$$L(u_k \mid \underline{y}) \approx u_k \min_{\substack{i=k..L \\ u_k \neq u_k^i}} \Delta_i^{s_i}, \tag{3.36}$$

where u_k is the bit estimate given by the ML path and L is the total number of stages in the trellis. Equation 3.36 means that in order to determine the soft value of the bit at stage k of the trellis, we first consider all discarded paths that merge with the ML path after stage k. The discarded paths that give the same estimate for the bit at stage k as the ML

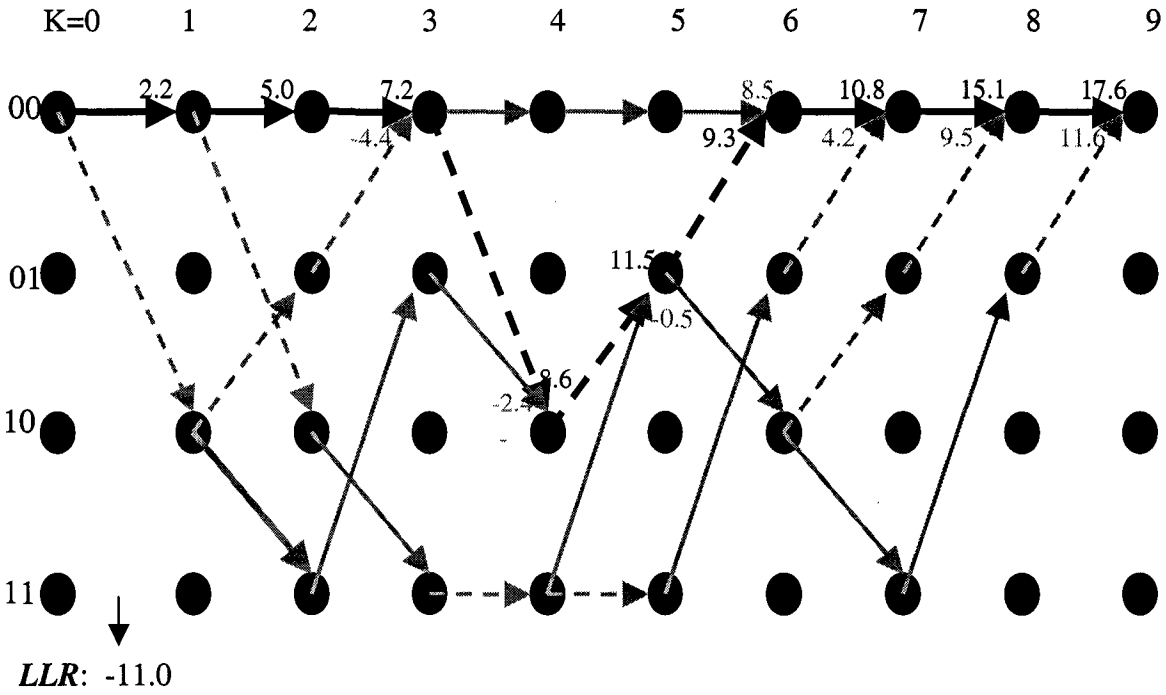


Figure 3.6. Simplified trellis during SOVA traceback

---

path, are ignored since they do not affect the reliability of the decision of u_k. For the remaining discarded paths the metric differences are computed. The minimum metric difference is then selected which gives the soft value of the bit at k^{th} stage of the trellis. For example in the calculation of the LLR of the bit at k=0, only the discarded paths which merge with the ML path at k=3 and k=4 have bit estimates different from the ML path. The metric differences at these mergers are 11.6 (7.2-(-4.4)) and 11 (8.6-(-2.4)) respectively and the estimate from ML path at k=0 is u_k=-1. The soft output at this stage is therefore -11.0. The LLRs of the remaining bits are calculated in a similar fashion.

3.4.4.3 Iterative Decoding using SOVA Algorithm

The iterative decoding with SOVA decoders is identical to the iterative decoding with MAP decoders explained in Section 3.4.1.5. As the channel sequence is received, the first SOVA decoder starts its decoding process. It uses the channel sequence and the a priori information to calculate the LLRs of the received bits. These LLRs provide the first decoder's estimate of the received bits. Extrinsic information is then calculated from these LLRs using Equation 3.18, which is interleaved and passed on to the second SOVA decoder as its a priori information. The second decoder which receives its own channel input (i.e. systematic bits and parity bits from the second encoder) uses this a priori information in its decoding process. The soft output of the second decoder is then used to compute its extrinsic information which is then passed on to the first decoder. In the second iteration the first decoder uses this information to improve its own estimates. New extrinsic information is calculated from these improved estimates and passed on to the second decoder. The iterative process continues until reliable estimates are available. As explained in Section 3.3, the number of iterations is usually limited to eight.

3.4.5 Bi-directional SOVA

In an effort to increase the performance of SOVA-based turbo decoding, Fossorier et al. introduced bi-directional SOVA [28]. They also showed that bi-directional SOVA can perform as well as Max-Log-MAP in turbo decoding. As the name suggests, bi-directional SOVA operates in forward as well as backward directions. Figure 3.7 shows a bi-directional SOVA based turbo decoder.

Note that there are four component decoders in this scheme. Two of the component decoders, i.e. SOVA 1 and SOVA 2, are identical to the SOVA decoder explained in Section 3.4.4, and the remaining two decoders i.e. B-SOVA 1 and B-SOVA 2 are what we call backward SOVA decoders. The backward SOVA decoder is similar to the regular SOVA decoder except the former operates in the reverse direction. Whereas the regular SOVA starts from the first stage of the trellis and moves in the forward direction, backward SOVA starts from the last stage of the trellis and moves in the reverse direction till it reaches the first stage of the trellis. Figure 3.8 shows the operation of forward and backward SOVA.

3.4.5.1 Rationale for Bi-directional SOVA

Let us first consider the forward SOVA. The SOVA calculates the metric differences $\Delta_k^{s_i}$ between the ML path and the discarded paths merging with the ML path at each state along the ML path (see Figure 3.5). It then selects the best path among the discarded paths, with a bit estimate opposite to the ML path to compute the reliability value of the bit. Although this path is the best among the discarded paths which merged with the ML path, the overall best path with the bit estimate opposite to that of the ML path may have been discarded before it could remerge with the ML path. Figure 3.9 shows an example scenario. Path 1 is the ML path and path 3 is the overall best path with bit estimate opposite to the ML path. However path 3 was discarded at stage i and path 2 survived instead, merging with the ML path. Therefore we can assume that in general reliability

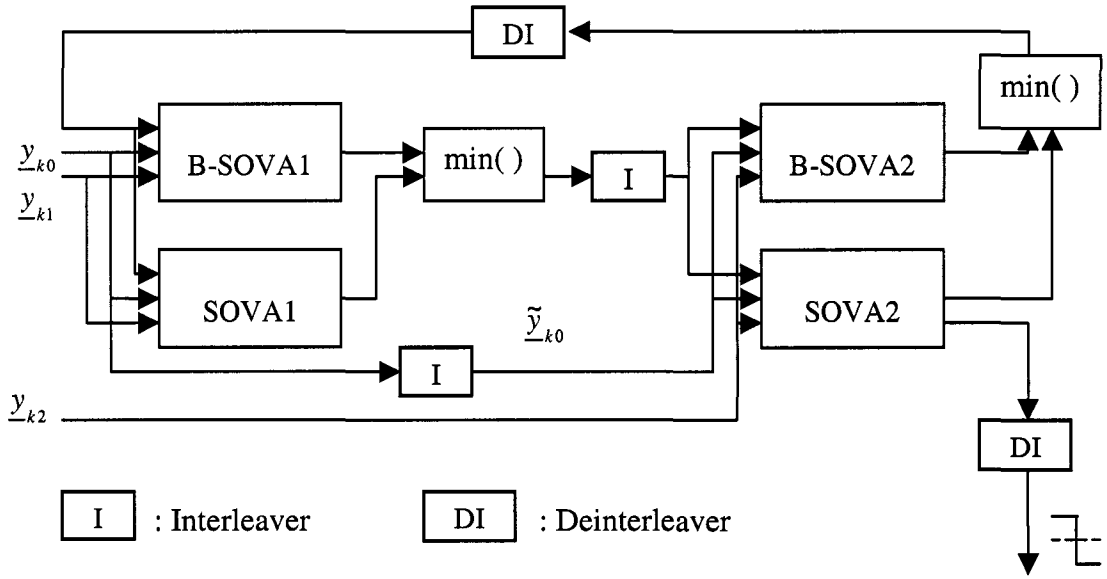


Figure 3.7. Bi-directional SOVA based turbo decoder

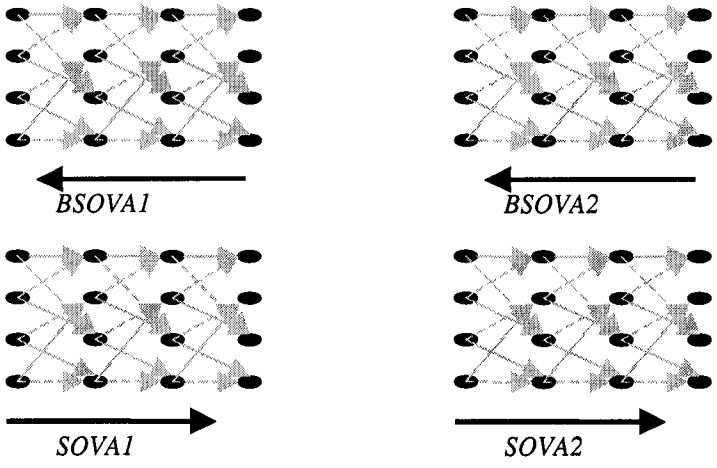


Figure 3.8. Trellis formation in bi-directional SOVA

values provided by SOVA are overestimated. Let us see how bi-directional SOVA can be used to improve these estimates.

The hard decisions of forward and backward SOVA, when operating on the same information, are identical. This is because they both choose the same ML path through the trellis. The 'quality' of the reliability values for both forward and backward SOVA is also the same. This means that if we replace forward SOVA with backward SOVA in a regular SOVA based turbo decoder there will be no improvement or degradation in the performance. Therefore backward SOVA has no advantage over forward SOVA as such. However the magnitudes of the reliability values delivered by forward and backward SOVA are different. The difference is due to the selection of different 'discarded' paths in forward and backward SOVA during the calculation of these reliability values. We can exploit these differences by using both forward and backward SOVA to improve the overall performance of a turbo decoder. Returning to our example in Figure 3.9, we note that although the best path (path 3) with bit estimate opposite to ML path (path 1) was discarded before it could remerge with the ML path, this path can survive and remerge with the ML path in backward SOVA. In general even if the best path does not survive, backward SOVA may still find a path better than that in forward SOVA. Therefore by comparing the soft values of the forward and backward SOVA and selecting the ones with lower magnitudes, we can avoid the overestimated reliability values calculated by conventional SOVA. We must note here that bi-directional SOVA does not consider all possible paths through the trellis and therefore is not optimal. Thus it does not have the same performance as the MAP algorithm.

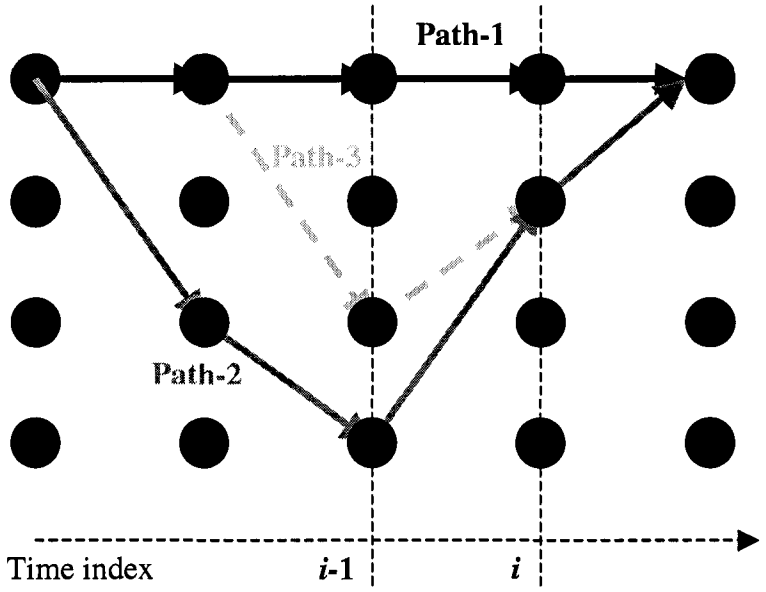


Figure 3.9. Path selection in SOVA decoding

3.4.5.2 Bi-directional SOVA Based Turbo Decoding

The bi-directional SOVA based turbo decoding is illustrated in Figure 3.7. As the channel sequence is received both component decoders (SOVA 1 and B-SOVA 1) in the first stage of the turbo decoder start their operation. Note that both component decoders receive the same channel sequence and a priori information. Since they both operate independent of each other, they can start their operation simultaneously. However the B-SOVA 1 cannot start its operation till the entire codeword has been received. This is because it decodes the codeword in the reverse direction. After both the decoders in the first stage calculate their soft outputs, we have two estimates for each decoded bit. We select the estimate that is smaller in magnitude and pass it on to the second stage of the turbo decoder. The second stage also has two decoders: SOVA 2 and B-SOVA 2. The

extrinsic information received from the first stage of the turbo decoder is passed on to both decoders in this stage as their a priori information. The decoding process in this stage is identical to the process in the first stage. The extrinsic information from this stage is passed on to the first stage and the iterative decoding process proceeds.

3.5 Summary

This chapter presents an overview of iterative decoding of turbo codes. We began our discussion by defining a channel model followed by the introduction to the principles of iterative decoding. We explained the importance of soft-input soft-output component decoders and the concept of log likelihood ratios (LLRs). Next we presented in detail, the description of the algorithms used in component decoders. The MAP algorithm calculates the a posteriori LLRs of individual bits by examining every possible path through the trellis. This results in optimum performance but makes the resulting decoder computationally complex. Max-Log-MAP and Log-MAP, presented in the subsequent sections transform the calculations in MAP, to the log domain, thereby making them considerably less complex. The second algorithm we presented was the SOVA algorithm, which is a modification of the classical Viterbi algorithm. It finds the ML path through the encoder trellis that corresponds to the ML transmitted sequence. The algorithm then considers the discarded paths together with the ML path to compute the LLRs of the individual bits in the ML sequence. Bi-directional SOVA consists of a forward and a backward SOVA. Backward SOVA is identical to forward SOVA except that it operates on the received sequence in the reverse direction. Backward SOVA can often find 'better'

discarded paths through the trellis which could have been missed in the forward SOVA, thus leading to an overall improvement in performance.

Chapter 4

Sliding Window Decoding of Turbo Codes

4.1 Introduction

The turbo coding schemes presented in previous chapters perform close to the Shannon limit only for very long frame lengths [1]. Their performance deteriorates with a decrease in the frame length. For example a 10,000-bit code outperforms a 1000-bit code by 0.7dB, and a 169-bit code by 1.6 dB at BER of 10^{-4} [23]. A long frame length, however, means a long decoding trellis for which the memory requirements as well as the decoder complexity are excessive from an implementation view point. In order to reduce the decoder complexity without affecting its performance significantly, we use longer frame lengths (e.g. 1000-bit or more) but decode the frames with sliding window component decoders [17]. The sliding window implementations of component decoders, i.e. MAP, Max-Log-MAP, SOVA and bi-directional SOVA reduce the decision depth of the trellis to around five times the encoder constraint length which eliminates the need to store the

trellis for the entire frame in memory. We begin this chapter with a review of conventional sliding window implementations of these algorithms. We then present new multiple bit release techniques which further reduce the complexity of the decoders significantly without any performance degradations.

4.2 Sliding Window Component Decoders

The component decoders (Max-Log-MAP, SOVA and bi-directional SOVA) explained in the previous chapter are all trellis based decoders with identical forward recursion. The number of trellis stages formed in the forward recursion is equal to the frame length of the code. Since the trellis has to be stored in the memory, for longer frame lengths, the decoder memory requirements are huge. However, it is possible to make reliable decisions after a relatively small number of trellis stages. This number is referred to as the decision depth D of the component decoder. The minimum decision depth is usually five times the encoder constraint length [17]. The reason we can make reliable decisions after the decision depth is that after this depth all the survivor paths at a given stage of the trellis tend to originate from the same initial state and have same first edge. The decoding decision corresponding to this edge will therefore not be affected by the subsequent trellis stages. This implies that we only need to compute and store the decision depth of the trellis to decode a single bit. This depth constitutes our decoding window. Once the bit in the decoded window is released, the next stage of the trellis is built and the window slides forward. A generalized sliding window decoding process is shown in Figure 4.1.

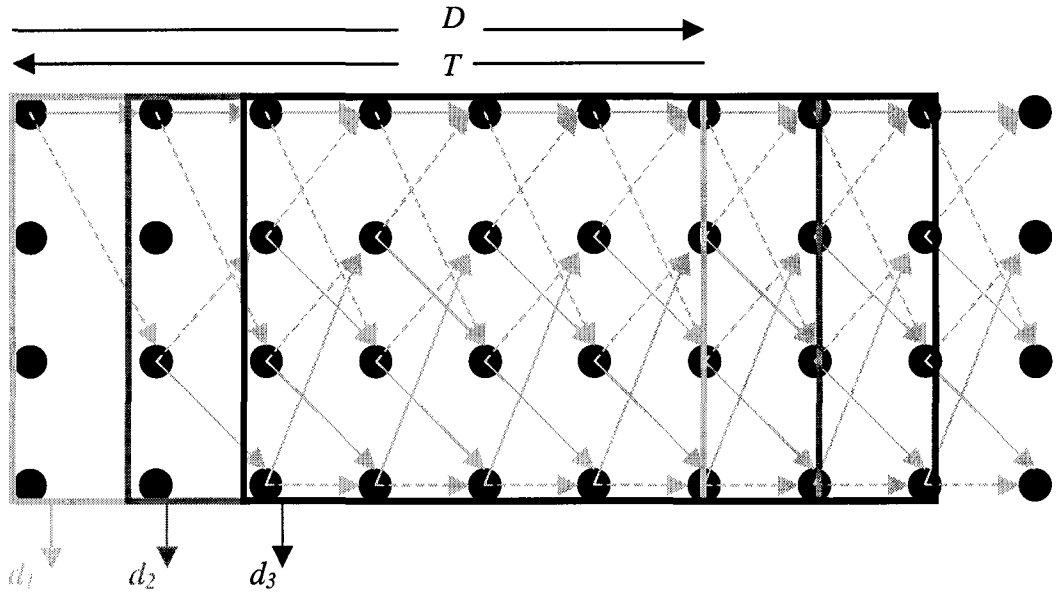


Figure 4.1. One bit release sliding window decoding

Let us now examine the sliding window implementations of Max-Log-MAP, SOVA and bi-directional SOVA component decoders in detail.

4.2.1 SOVA and Bi-directional SOVA

To explain the sliding window SOVA algorithm we define the following terms.

D_{SOVA} Decision depth of trellis for SOVA.

T_{SOVA} Traceback depth of trellis for SOVA.

T_{SOVA} is the total number of trellis stages, where the discarded path merging with the ML path is considered to find the reliability value of decoded bit. For single bit release SOVA, forward recursion starts by building the first D_{SOVA} stages of the trellis. This is followed by SOVA traceback at each stage of the trellis in the current window. T_{SOVA} in this case equals D_{SOVA}. The decoded bit at the first stage of the trellis is released and the

decoding window slides forward by one trellis stage. The decoded bit at the second trellis stage is released in this window followed by another slide of the window and so on. Decoding of bi-directional SOVA is the same as simple SOVA except that sliding window in backward SOVA starts from the last stage in the trellis and moves in the opposite direction, thus releasing the bits in reverse order.

4.2.2 MAP and Max-Log-MAP

The parameters for MAP and Max-Log-MAP algorithms are defined as follows.

D_{MAP} Same as D_{SOVA}

T_{MAP} Number of trellis stages in the backward recursion which is the same as D_{MAP} or D_{SOVA}.

Forward recursion in the MAP algorithm is similar to the forward recursion in SOVA, the only difference being the actual calculation of path metrics. Max-Log-MAP however has forward recursion equivalent to that of SOVA. The forward recursion in both MAP and Max-Log-MAP is followed by a backward recursion instead of a traceback as was the case in SOVA. This backward recursion is identical to the forward recursion but proceeds from the last stage in the decoding window to the first stage. T_{MAP} therefore is always the same as D_{MAP} in the sliding window decoding of MAP and Max-Log-MAP. After the release of the decoded bit, the window slides forward in the same manner as explained previously for SOVA.

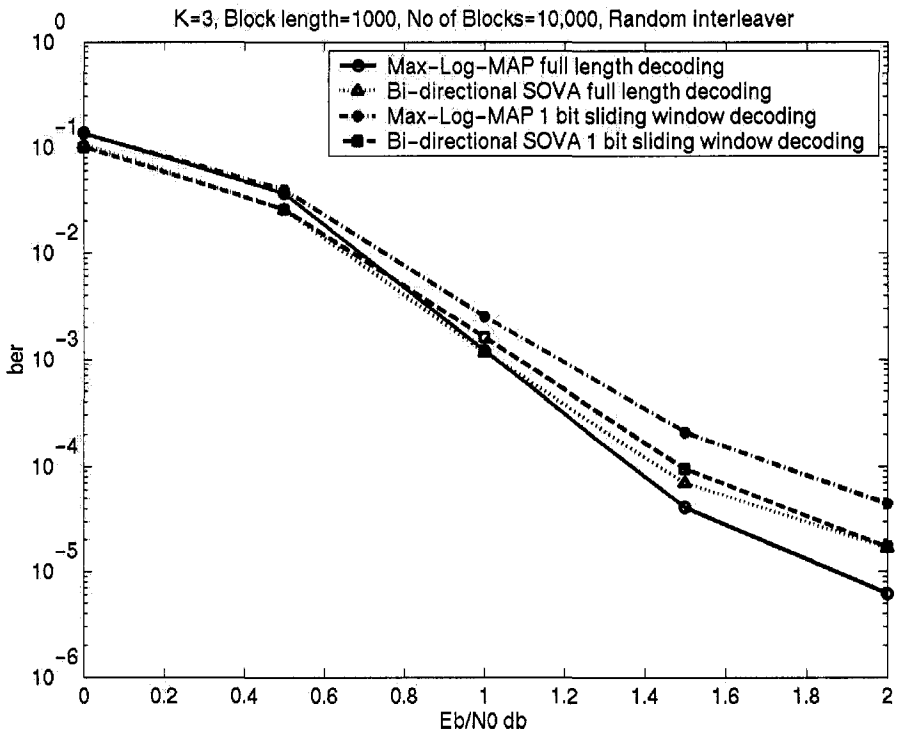


Figure 4.2. BER performance of one bit release sliding window decoding

4.2.3 Comparison of SOVA and MAP

The bit error rate (BER) performance comparison of full frame length decoding and sliding window single-bit release decoding for bi-directional SOVA and Max-Log-MAP is shown in Figure 4.2. A turbo encoder with pseudorandom interleaver and RSC component encoders of constraint length 3 have been used. The value of D_{SOVA}/D_{MAP} and T_{SOVA}/T_{SOVA} for the sliding window component decoders is 15. It is evident from the simulation results that sliding window implementations perform reasonably close to full frame length implementations. Moreover they are significantly less complex and therefore more suitable for practical implementations.

4.3 Multiple Bit Release Sliding Window Decoding

We have seen in the previous section that sliding window implementations significantly reduce the memory requirements of component decoders. However, to ensure high performance, the component decoders and hence the turbo decoder must also be able to decode at very high speeds. One way to increase the speed of the component decoders is to release more than one bit in a single decoding window. This will result in the fewer slides of the window and hence faster decoding of the encoded frame. Vucetic and Yuan suggested a Max-Log-MAP based decoder in which multiple bits are released by doubling the size of sliding window [17]. In this thesis however, we will study the effect of progressively releasing multiple bits in the Max-Log-MAP decoder. Additionally we will also analyze SOVA and bi-directional SOVA for multiple bit release implementations. One bit release sliding window decoding will be our reference for performance and complexity analysis. We begin by considering a number of modifications to one bit release sliding window implementations, which allow multiple bits to be released in one decoding window. We then examine the rationale behind these modifications and their probable influence on decoder complexity and performance.

4.3.1 SOVA and Bi-directional SOVA

In order to facilitate the release of N bits in one decoding window we consider the following modifications.

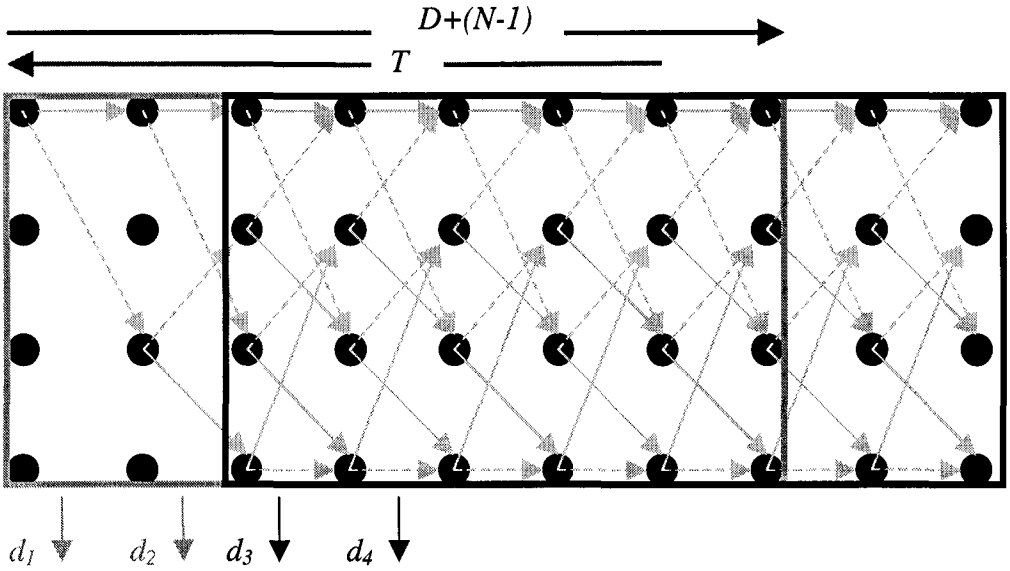


Figure 4.3. Multiple bit release sliding window decoding

---

1. Increase the decision depth of trellis by N-1

   $\boldsymbol{D}_{mult\_SOVA} = \boldsymbol{D}_{SOVA} + (N\text{-}1)$ where $1 \leq N \leq \boldsymbol{D}_{SOVA}$

2. Keep $\boldsymbol{T}_{mult\_SOVA}$ same as $\boldsymbol{T}_{SOVA}$ and use the same ML and discarded paths in the decoding of all the N bits in a decoding window.
3. After N bits in a window have been decoded, slide the window forward by N trellis stages.

Figure 4.3 shows the N bit release sliding window decoding graphically. The decoding process begins with a forward recursion which builds $\boldsymbol{D}_{mult\_SOVA}$ trellis stages and finds the ML path. This is followed by the SOVA traceback in which $\boldsymbol{T}_{mult\_SOVA}$ discarded paths, one at each stage in the SOVA traceback depth, are considered. The ML path and the discarded path at each trellis stage in the SOVA traceback are not only used to

calculate the soft value of the decoded bit at the first stage of the trellis but also for the decoded bits at trellis stages $k=n$, where $2 \leq n \leq N$. Once the SOVA traceback is complete and N bits are released, the window slides forward by N trellis stages.

4.3.2 The Effect of Modifications on Decoder Complexity and Performance

Let us first consider how these modifications will reduce the computational complexity of the component decoders. The number of computations in the forward recursion of SOVA and MAP remain unaffected by the modifications. The reason is that even though we are building the trellis in steps (i.e. N stages after each slide of the window), the number of stages required to decode the entire block is still the same. The number of SOVA tracebacks, however, is reduced by a factor of N which implies fewer computations and an overall increase in the decoder speed. This increase comes at the expense of additional hardware i.e. memory to store N-1 additional stages of the elongated trellis and logic to enable the release of N bits simultaneously. The amount of additional hardware required is proportional to N.

In order to determine the effect of proposed modifications on the decoder performance, we analyze how they will affect the reliability of individual bits released in a decoding window. An increase in the trellis decision depth implies more reliable ML paths for the first N-1 bits in the window. The ML path for the N^{th} bit has the same length as in one bit release implementation and therefore its reliability is unaffected. On the other hand, keeping SOVA traceback length unchanged means that all but the first bit in

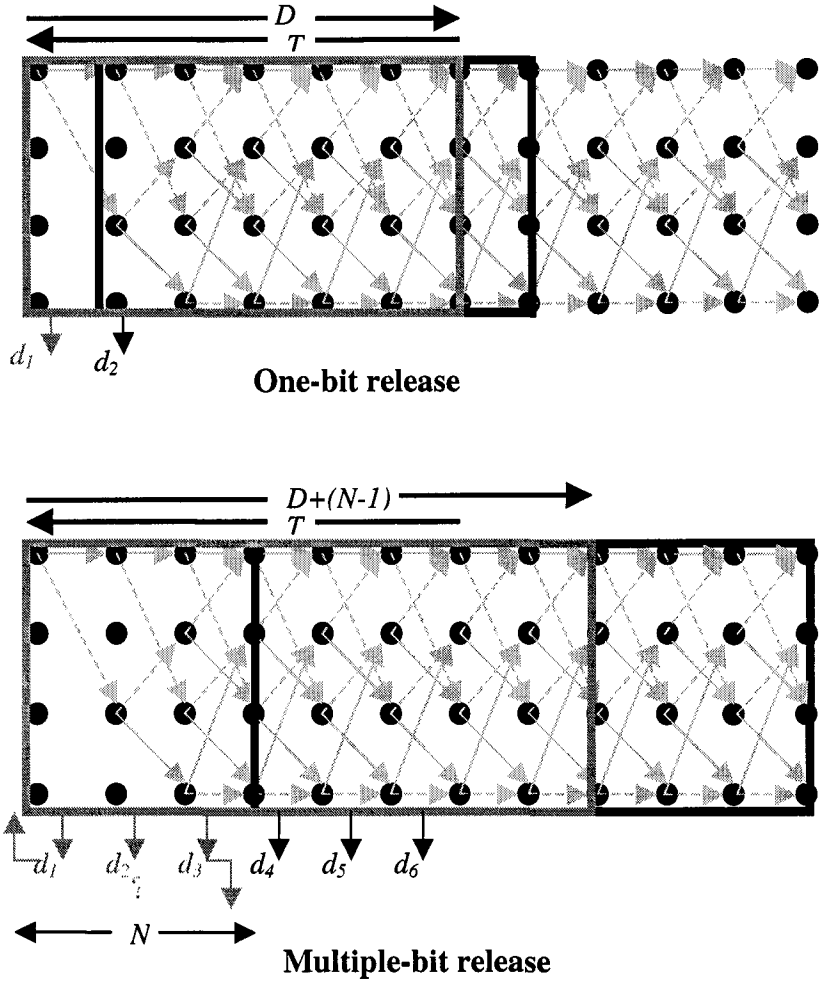


Figure 4.4. Performance analysis of multiple bit release sliding window decoding

---

the decoding window now have a reduced traceback. A reduced traceback implies fewer discarded or alternate paths available in the decoding process. It is clear from the above discussion that while the first modification tends to increase the reliability of individual bits, the later has the opposite effect. Consider the example shown in Figure 4.4. We compare a 3 bit release implementation (N=3) to a single bit release implementation. As discussed above the reliability of the first bit d_1 will increase since the length of the ML

path D, used in its decoding has increased by 2 while its traceback length T is unchanged. The reliability of the last bit d_3 will decrease since the length of its ML path is the same as it was in one bit release implementation while its traceback length has decreased by 2. The intermediate bit d_2 has the length of its ML path increased by1, however at the same time its traceback length has also decreased by 1. Therefore its reliability may increase, decrease or remain unaffected depending on these individual effects. In general we can expect the effect of two modifications to balance each other and therefore multiple bit release implementations to have performances similar to single bit release implementations. Moreover if the two effects are not uniform and decoded bits at different positions are affected differently, we can exploit these differences to even increase the overall performance of a turbo decoder. This is the motivation behind the multiple bit release implementation presented above.

4.3.3 MAP and Max-Log-MAP

A multiple bit release sliding window MAP and Max-Log-MAP can be implemented in a fashion similar to that of SOVA explained above. The key difference is the length of backward recursion which in the case of MAP and Max-Log-MAP is the same as forward recursion i.e. $D_{mult\_MAP} = T_{mult\_MAP}$. The performance and complexity analyses are also similar to those of SOVA. The number of computations in the forward recursion of MAP and Max-Log-MAP remains unaffected by the modifications. The number of computations in the backward recursion increases in each window; however, the number

of total windows is reduced by a factor of N, leading to an overall reduction in computations.

4.4 Summary

In this chapter we have discussed sliding window decoding of turbo codes. The sliding window approach allows us to use large block lengths but at the same time design decoders with reasonable complexity. All the component decoding algorithms presented in Chapter 3 (and the turbo decoders based on them) are suitable for sliding window implementations.

We described a generalized sliding window implementation in which one bit was released after each slide of the window and we also showed that the performance of sliding window decoding is comparable to that of full block length decoding. Next we examined the possibility of increasing decoding speed by releasing multiple bits in each decoding window. The proposed modifications to single bit release SOVA, bi-directional SOVA and Max-Log MAP enabled the release of multiple bits with a very little increase in hardware complexity. We also analyzed that multiple bit release implementations should be comparable in performance to the single bit release implementations. We shall investigate this claim more thoroughly by examining the BER (bit error rate) simulation results of these implementations in Chapter 5.

Chapter 5

Performance of Multiple Bit Release Turbo Decoders

5.1 Introduction

Multiple bit release implementations of component decoders, presented in Chapter 4, increase the decoding speed by enabling the release of multiple bits in a decoding window of trellis based decoding algorithms such as SOVA, bi-directional SOVA and Max-Log-Map. In Chapter 4 we also conjectured that multiple bit release implementations can be comparable to single bit release implementations in terms of BER performance. In this chapter we shall verify this claim by simulating the performance of a turbo coded system with turbo decoders based on SOVA, bi-directional SOVA and Max-Log-MAP component decoders. We shall also estimate the possible speedups that can be obtained from multiple bit release implementations relative to single

bit release implementations. This will provide us with speed versus performance trade-offs for various multiple bit release implementations. Finally we shall simulate the performance of multiple bit release punctured turbo codes to confirm that results obtained for non-punctured turbo codes also hold for punctured turbo codes. We must note here that multiple bit release implementations and their corresponding speedup estimates provided in this chapter are for component decoders. Since these component decoders operate in an iterative fashion in a turbo decoder, a faster component decoder translates into a faster turbo decoder. The BER performance results presented in this chapter however are for the turbo coded system which employs these multiple bit release component decoders.

5.2 Simulation Setup

In order to simulate the encoding process, random binary sequence u of length L is generated. This sequence is then encoded by a turbo encoder that consists of two identical RSC encoders of Figure 2.3, separated by a pseudo-random interleaver. The encoded sequence v is then mapped to signal levels using an antipodal baseband signaling scheme characterized by

$$x = 2v\text{-}1. \tag{5.1}$$

The channel symbols are then corrupted by additive white Gaussian noise resulting in the received sequence

$$y = x + e, \tag{5.2}$$

Table 5.1. Standard simulation parameters

Channel	Additive White Gaussian Noise (AWGN)
Component Encoders	2 identical RSC codes (SOVA, bi-directional SOVA & Max-Log-MAP)
RSC parameters	Constraint Length $K = 3$, forward polynomial = $1+D^2$, feedback polynomial = $1+D+D^2$ (Figure 2.3)
Interleaver	1000 bit random interleaver
Decoding iterations	8
Decoding window size for1 bit release SOVA: D_{SOVA}	$5\times K$(constraint length)
D_{MAP}	$5\times K$(constraint length)

where e is the zero-mean Gaussian noise random variable with variance σ^2. The variance σ^2 is calculated according to the desired energy per bit to noise density ratio, E_b/N_0, using the relation

$$\sigma = \sqrt{1/(2 \cdot (E_s / N_0))}\,, \tag{5.3}$$

where E_s/N_0 is the energy per symbol to the noise density ratio. For coded channels E_s/N_0 is related to E_b/N_0 by

$$E_s/N_0 = E_b/N_0 + 10log_{10}(r), \tag{5.4}$$

where r is the code rate. For the non-punctured turbo coding scheme of Figure 2.8, used in our simulations, r is 1/3.

The received sequence is decoded by a turbo decoder which consists of two component soft-in soft-out decoders operating in parallel as shown in Figure 3.2. The number of decoding iterations is limited to eight and the component decoders used in our simulations are SOVA, bi-directional SOVA and Max-Log-MAP. A summary of the standard parameters used in the simulations is given in Table 5.1.

5.3 Single Bit Release Component Decoders

Figure 5.1 shows the bit error rate (BER) performance comparison of one bit release SOVA, bi-directional SOVA and Max-Log-MAP. While both bi-directional SOVA and Max-Log-MAP are better than simple SOVA it is interesting to note that bi-directional SOVA is consistently better than Max-Log-MAP. Similar results were also reported for normal or full length decoding of bi-directional SOVA and Max-Log-MAP in [28]. We shall use these single-bit release curves as our reference, and compare the performance of multiple bit release implementations against these curves.

5.4 Performance of Multiple Bit Release SOVA

The BER performance results for multiple bit release SOVA are shown in Figure 5.2. As we increase N, i.e. the number of bits released in a single window, there is little effect on the performance of the turbo decoder. For example, for N=4 and N=8 the performance is almost identical to the single bit release implementation i.e. N=1. As we explained in

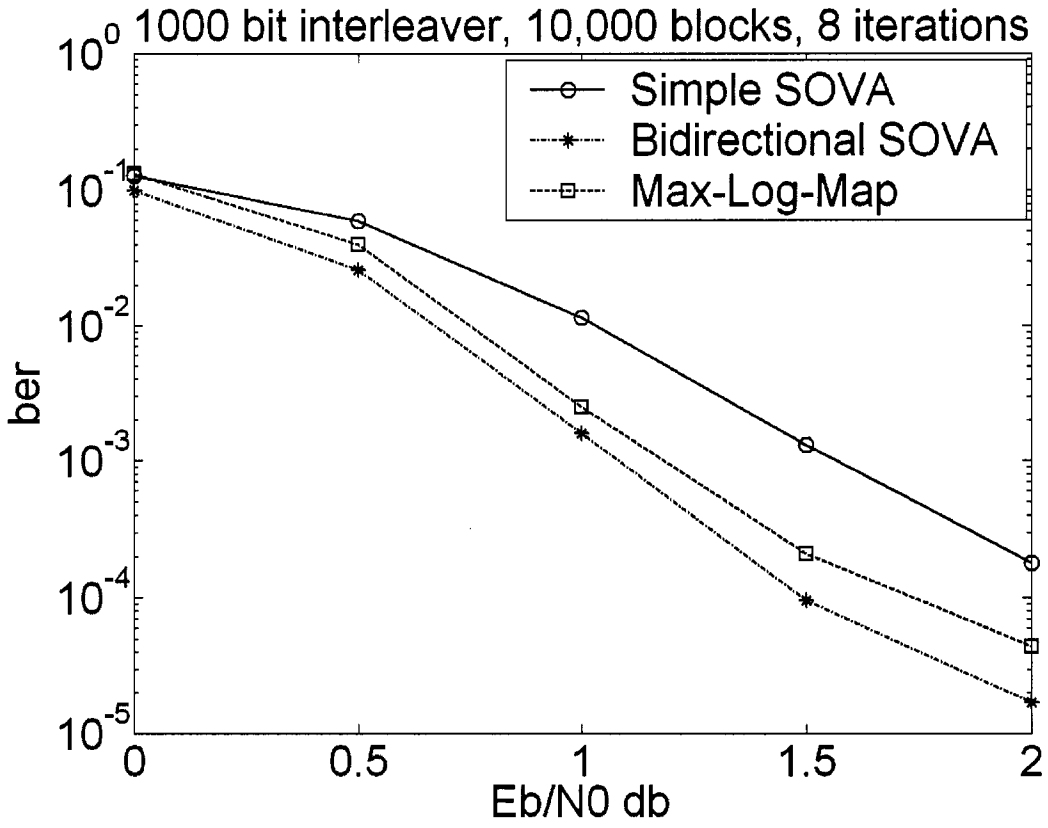


Figure 5.1. BER performance comparison of one bit release simple SOVA, bi-directional SOVA and Max-Log-MAP

Section 4.3.2, this implies that the decrease in reliability due to the reduced traceback is balanced by the increase in reliability due to the increased length of the forward recursion. However as we increase N beyond 8, the performance starts to degrade and for N=15 (original length of the window), the performance degrades significantly. At this point the effect of reduced traceback dominates the effect of increased window length.

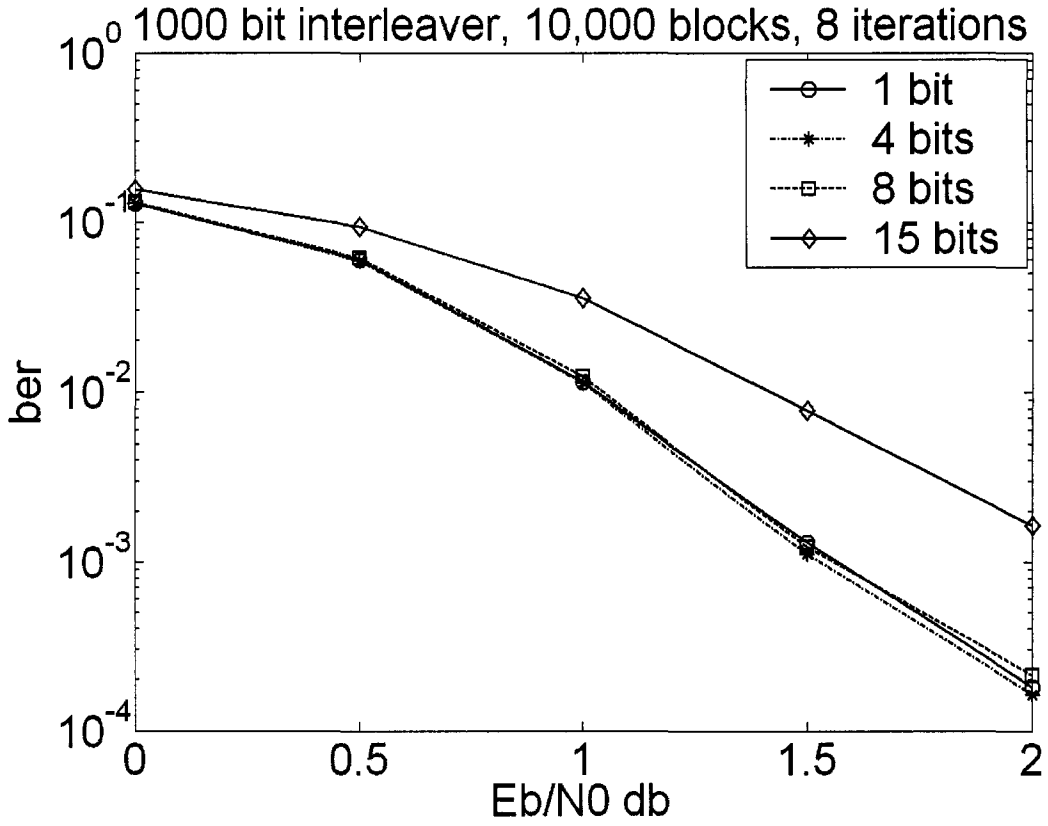


Figure 5.2. BER performance comparison of multiple bit release sliding window simple SOVA

5.5 Speedup from Multiple Bit Release SOVA

It is evident from Figure 5.2 that with the proposed modifications we can release up to 8 bits in a decoding window without any performance degradation. This means that in the decoding of a block by a component SOVA decoder, we can reduce the number of traceback windows by a factor of 8. This reduction leads to an increase in the overall decoding speed. The magnitude of this speedup depends on the implementation details and can vary significantly from implementation to implementation depending on the

speed versus hardware tradeoffs employed. In order to compare the efficiency of multiple bit release implementations of SOVA against single bit release SOVA we consider a simple and efficient implementation. We will assume that all the path metrics at a given stage of the trellis in the forward recursion can be calculated simultaneously. Therefore the time to build a single trellis stage in the forward recursion is constant and we will refer to it as T_f. For a block of length L, we need to build as many forward trellis stages. Therefore, irrespective of the window size, the time to complete the forward recursion for the entire block is given by $L \times T_f$.

The SOVA traceback simply involves the comparison of the differences between the ML path and the discarded paths at each trellis stage in the traceback window. Since all the metrics have been calculated and stored in the forward recursion, these differences can be calculated in parallel followed by a comparison. We will refer to the time that it takes to compute the metric differences and their comparison followed by the selection of the best metric difference as T_t. Recall that the length of traceback window in multiple bit release implementations does not change and therefore we can safely say that T_t will be the same irrespective of the value of N. The number of traceback windows in the decoding of a block is determined by the value of N. For N=1 the window slides by one trellis stage after the release of a bit estimate and therefore there are L traceback windows where L is the size of the block. For N=2 however, 2 bits are released in each window and the window slides by 2 trellis stages. Therefore the number of traceback windows is reduced by a factor of 2. Now we can calculate the approximate time it takes to decode a block by an N bit release component SOVA decoder as

Table 5.2. Speedup from an 8 bit release implementation of SOVA

T_t/T_f	**Speedup**
$T_t = 0.5\ T_f$	1.41
$T_t = T_f$	1.77
$T_t = 1.5\ T_f$	2.11
$T_t = 2\ T_f$	2.40
$T_t = 4\ T_f$	3.33

Time to decode 1 block = time for forward recursion + time for tracebacks

$$= L \times T_f + \frac{L}{N} \times T_t . \tag{5.5}$$

Equation 5.5 gives us an idea of the speed up that we can achieve with multiple bit release implementations. The magnitude of the speedup depends on the ratio of T_f and T_t and the value of N. For example if T_f and T_t are equal, an 8 bit release (N=8) implementation will translate in to a speed up of 1.77 over the single bit release implementation. Table 5.2 shows the speedups that can be achieved from an 8 bit release implementation for some possible values of T_t/T_f. It is evident from the table that reducing T_f with respect to T_t leads to higher speedups. A higher value of N also improves the speed further however as we have seen from the BER simulation results, the performance of SOVA based turbo decoder begins to deteriorate when N is increased beyond 8 when compared against single bit release implementations.

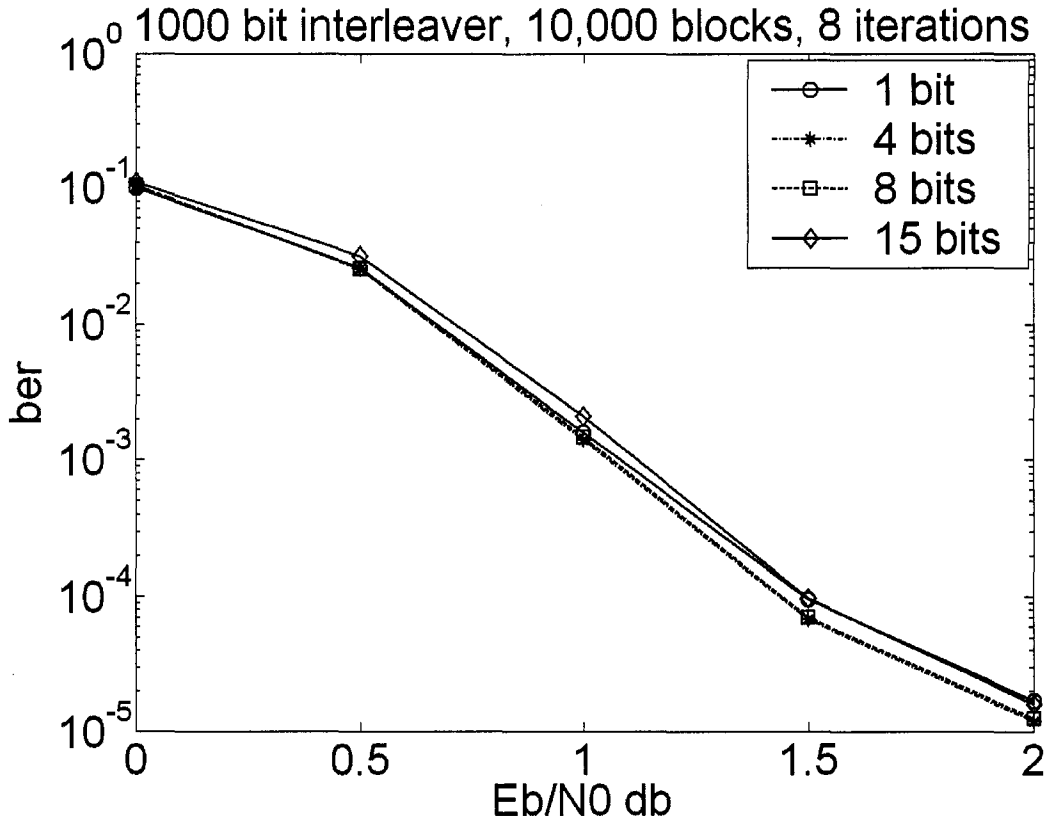


Figure 5.3. BER performance comparison of multiple bit release sliding window bi-directional SOVA

5.6 Performance of Multiple Bit Release Bi-directional SOVA

The BER performance results for multiple bit release bi-directional SOVA are shown in Figure 5.3. As we increase *N*, i.e. the number of bits released in a single window, the performance of the turbo decoder improves slightly. An 8 bit release bi-directional SOVA implementation is consistently better than single bit release bi-directional SOVA after 0.5 db. However if we release 15 bits, which is the size of the original window in one bit release implementation, there is no significant deterioration in performance. This

Table 5.3. Speedup from a 15 bit release implementation of bidirectional SOVA

T_t/T_f	**Speedup**
$T_t = 0.5\ T_f$	1.45
$T_t = T_f$	1.88
$T_t = 1.5\ T_f$	2.27
$T_t = 2\ T_f$	2.65
$Tt = 4\ Tf$	3.95

implies that for smaller values of N the increase in reliability due to the increased length of forward recursion surpasses the decrease in reliability due to the reduced traceback, thus leading to an overall increase in the reliability of decoded bit estimates. For larger values of N, however, the two effects more or less balance each other. Bi-directional SOVA therefore is much more resilient to a reduced traceback than SOVA.

5.7 Speedup from Multiple Bit Release Bi-directional SOVA

The speedup analysis of bidirectional SOVA is similar to the one presented for SOVA in Section 5.5. Since we can release 15 bits in bi-directional SOVA without any performance degradation, a speedup of 1.875 can be achieved when T_f and T_t are equal. The possible speedups from a 15 bit release implementation for different values of T_t/T_f are shown in Table 5.3.

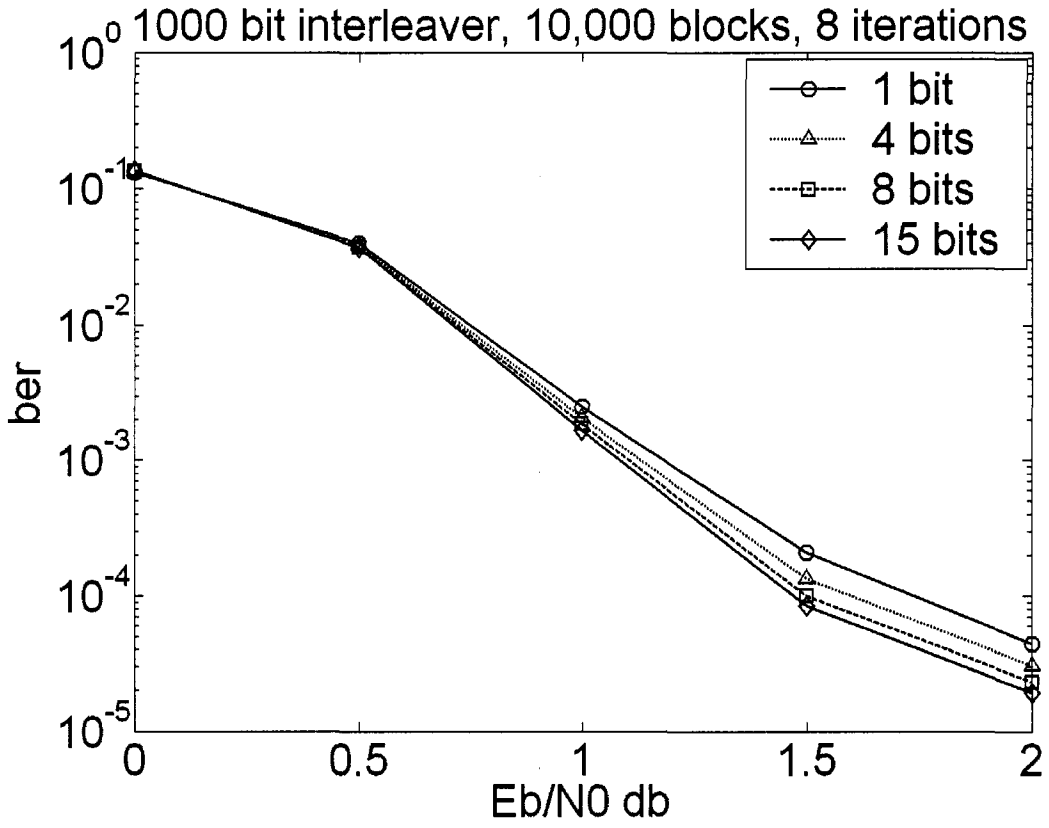


Figure 5.4. BER performance comparison of multiple bit release sliding window Max-Log-MAP

5.8 Performance of Multiple Bit Release Max-Log-MAP

The BER performance results for multiple bit release Max-Log-MAP are shown in Figure 5.4. The performance of the turbo decoder improves consistently with an increase in *N*. As we explained in Section 4.3.3, a multiple bit release implementation of Max-Log-MAP means an increase in the length of forward as well as backward recursion. This obviously leads to an improved performance, as more reliable estimates are available in the forward as well as backward directions.

5.9 Speedup from Multiple Bit Release Max-Log-MAP

The forward recursion of Max-Log-MAP is identical to that of SOVA; however, it is followed by a backward recursion instead of a traceback. Since this backward recursion is identical to the forward recursion we can safely assume that the time it takes to build a trellis stage in the forward direction is equal to the time it takes to build a stage in the backward direction. Now we can estimate the time required to decode a single block using a Max-Log-MAP component decoder as:

$$\text{Time to decode 1 block} = L \times T_f + \frac{L}{N} \times T_f \times D_{mult\_MAP}, \tag{5.6}$$

where T_f is the time to build one trellis stage and D_{mult_MAP} is the size of the decoding window in multiple bit release Max-Log-MAP or MAP decoding. It must be noted that towards the end of the block the size of the decoding window gets smaller. The above equation does not take this into account and therefore is not exact. However if D_{mult_MAP} is significantly smaller than L, this approximation is acceptable. Moreover we will use the above equation for the comparison of single and multiple bit release decoders maintaining the same assumptions across all implementations. Let us now calculate the speedup from multiple bit release implementations. We can rewrite Equation 5.6 as

$$\text{Time to decode 1 block} = L \times T_f \left(1 + \frac{D_{MAP} + N - 1}{N}\right), \tag{5.7}$$

where D_{MAP} is the length of the original window(N=1) in one-bit release implementation and $D_{mult\_MAP} = D_{MAP} + N - 1$. We use Equation 5.1 to calculate the speedups for different values of N for D_{MAP} =15. The results are listed in Table 5.4.

Table 5.4. Speedup from multiple bit release Max-Log-MAP

N	**Speedup**
2	1.78
4	2.90
8	4.26
15	5.46

5.10 Comparison of Bi-directional SOVA and Max-Log-MAP

Figures 5.5 and 5.6 show the BER performance comparison of bi-directional SOVA and Max-Log-MAP for N=8 and N=15 respectively. For N=8 bi-directional SOVA is consistently better than Max-Log-MAP whereas both decoders have a similar performance for N=15. In order to compare the decoding speed, we consider equations 5.5 and 5.6. The first term in both equations i.e. $L \times T_f$ is same since it represents the time of forward recursion which is identical in SOVA and Max-Log-MAP. The difference in the speed of the decoders therefore depends on the terms T_t in Equation 5.5 and $T_f \times D_{mult\_MAP}$ in Equation 5.6. For a 4-state decoder D_{mult_MAP} must be greater than 15 (five times the encoder's constraint length), which is the minimum window size required to make reliable decisions. Furthermore, for the same window size T_t/T_f can range from 0.5 to 2 for the implementation described in Section 5.5. Therefore we can safely say that the decoding speed of bi-directional SOVA is higher than that of Max-Log-MAP.

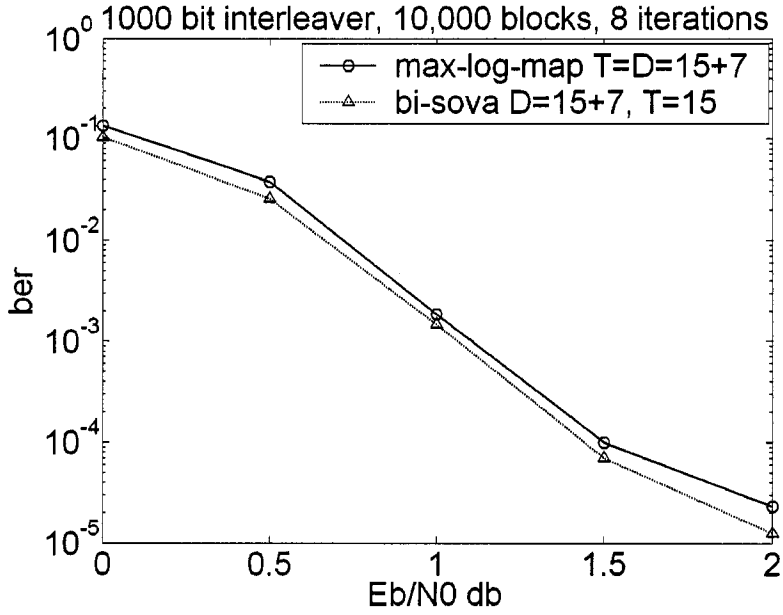


Figure 5.5. BER performance comparison of eight bit release sliding window bi-directional SOVA and Max-Log-MAP

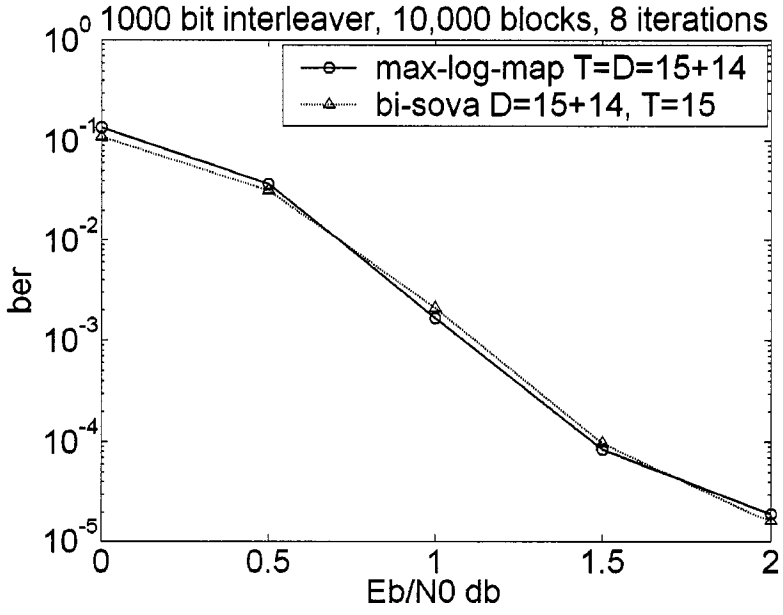


Figure 5.6. BER performance comparison of fifteen bit release sliding window bi-directional SOVA and Max-Log-MAP

The above analysis assumes a 4-state encoder. However let us also examine how the two decoders compare for 8 or 16 state encoders. An increase in the number of states implies an increase in the minimum decoding window length required for making reliable decisions. For example, D_{mult_MAP} must be greater than 20 or 25 for an 8 or 16-state encoder respectively. An increase in the window size will have no effect on T_f and a very little effect on T_t. An increased window implies a longer ML path and therefore more discarded paths in the SOVA traceback. Since the metric differences in SOVA traceback can be computed in parallel this only translates into a time penalty of using a larger comparator for comparing and selecting the best metric difference. A higher state encoder will therefore only enhance the speed difference between bi-directional SOVA and Max-Log-MAP.

5.11. Multiple Bit Release Punctured Turbo Codes

Puncturing is used to increase the rate of a turbo code and many practical channel coding standards employ punctured turbo codes. Therefore it is worthwhile to analyze the performance of multiple bit release component decoders in a turbo coded system which employs puncturing. The simulation of a punctured coded system is very similar to the one described in Section 5.2. The encoding operation is followed by an additional puncturing block which punctures the output of the encoder according to a fixed puncturing pattern to obtain the desired code rate. The turbo decoder used for the non-punctured codes can also be used to decode punctured codes; however, the punctured bits must be inserted in the received sequence at the decoder input. Since punctured bits are

not transmitted, we insert a neutral value in the stream where coded bits were punctured. For the antipodal baseband signaling scheme of Section 5.2, where bits 0 and 1 are mapped to signal levels -1 and 1 respectively, 0 would be the neutral value.

Figures 5.7 and 5.8 show the BER performance results for punctured turbo codes with multiple bit release bi-directional SOVA and Max-Log-MAP respectively. We have obtained a rate 1/2 code from a non punctured rate 1/3 code by puncturing the output of the two encoders alternately. In the figures the puncturing pattern is represented by a puncturing vector where 0 denotes the bit positions that were punctured. The results show that for punctured turbo codes the variation in performance due to the variation in N follow a pattern similar to the one we observed for non-punctured codes in Figures 5.3 and 5.4. This indicates that the behavior of multiple-bit release decoders does not change in a punctured turbo coded system.

When a code is punctured to increase the code rate, the performance of the coded system deteriorates. For a turbo coded system with conventional component decoders the performance drops approximately 0.6 dB, when the code rate is increased from 1/3 to 1/2 [23]. The comparison of non-punctured and punctured fifteen bit release SOVA and Max-Log-MAP in Figure 5.9 and 5.10 also shows a similar deterioration in performance. Therefore, multiple bit release component decoders behave very much like conventional component decoders in a punctured turbo coded system.

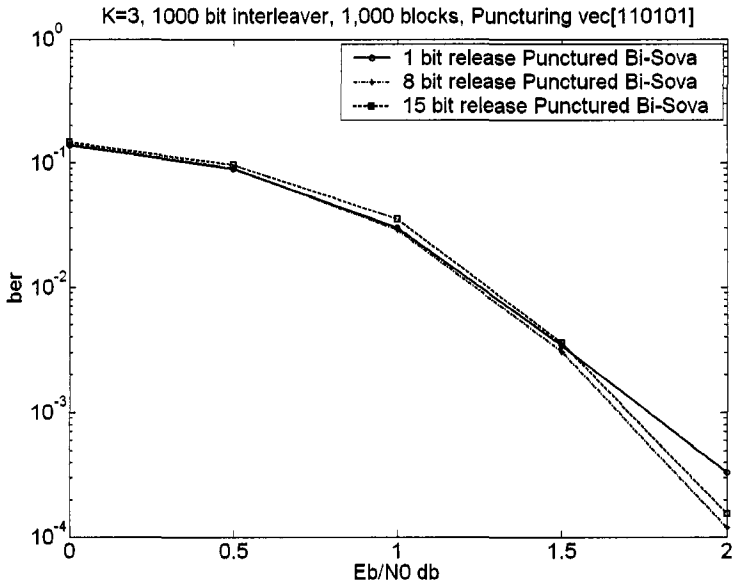


Figure 5.7. BER performance comparison of multiple bit release punctured bi-directional SOVA

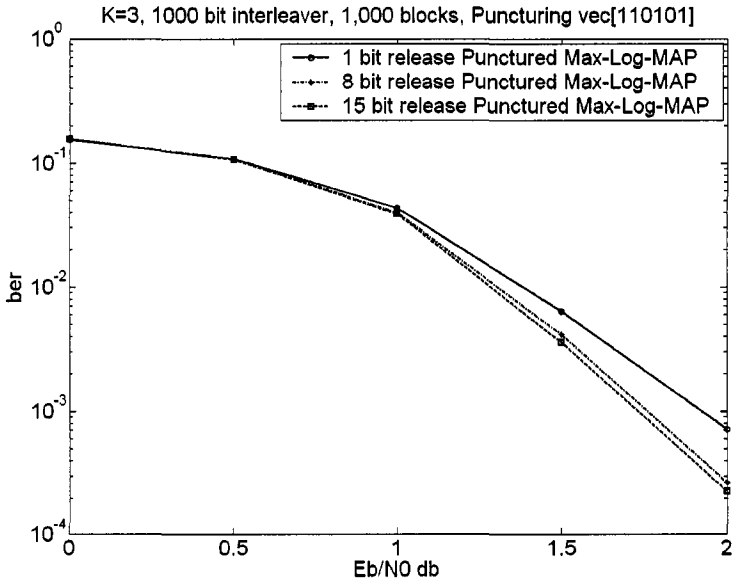


Figure 5.8. BER performance comparison of multiple bit release punctured Max-Log-MAP

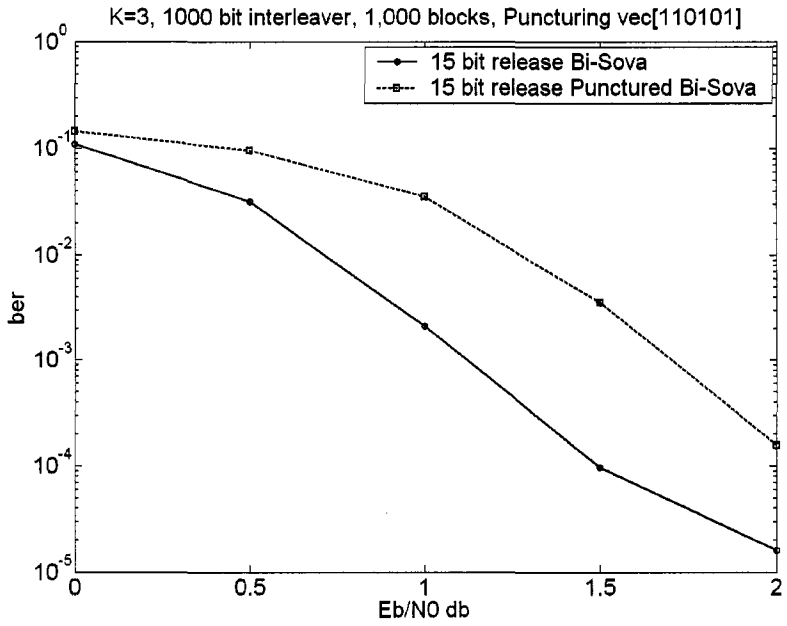


Figure 5.9. Performance degradation in multiple bit release punctured bi-directional SOVA

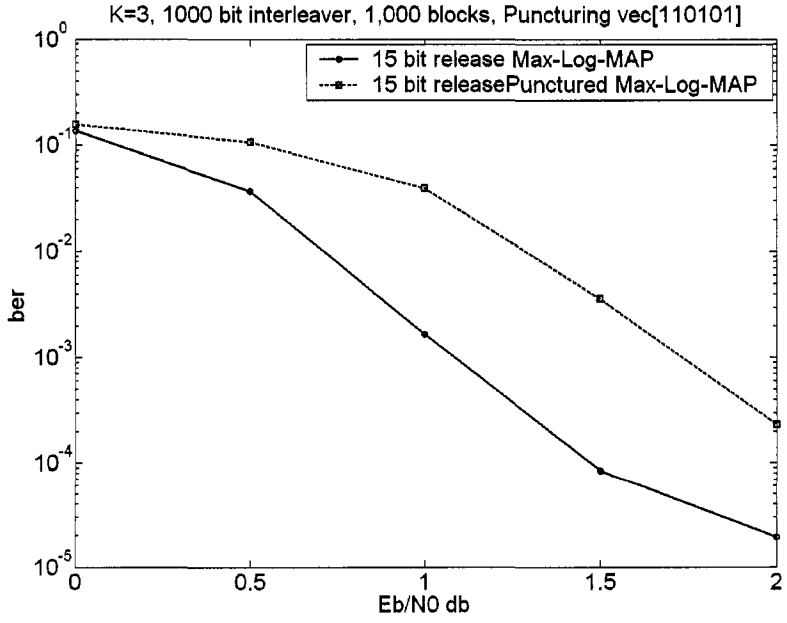


Figure 5.10. Performance degradation in multiple bit release punctured Max-Log-MAP

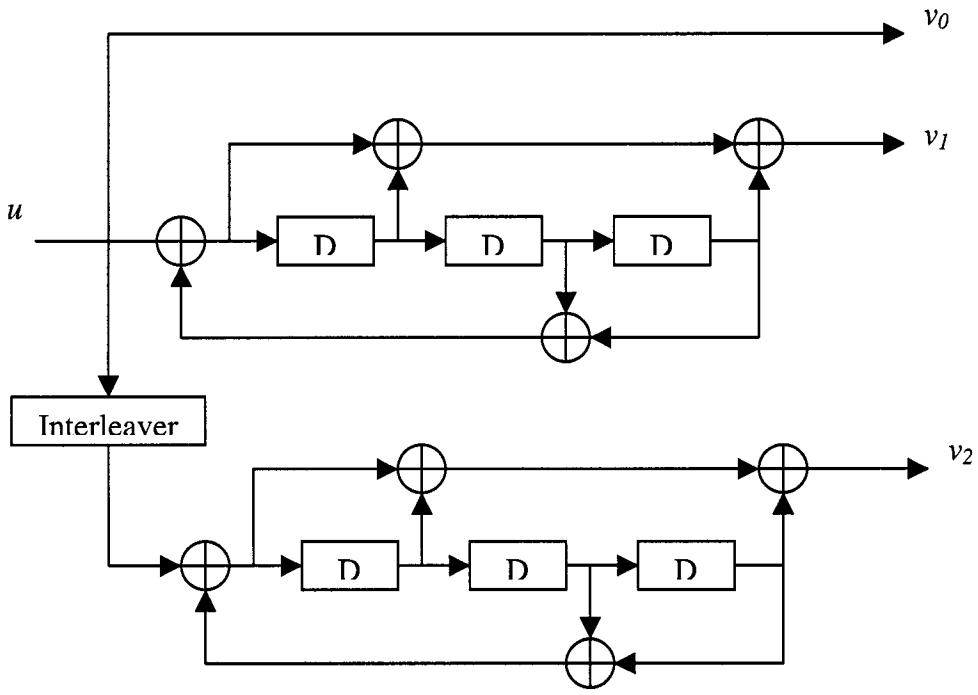


Figure 5.11. 3G turbo encoder

5.12 Turbo Codes with Higher State Encoders

The BER simulation results presented so far in this chapter are for turbo coded systems with four state component encoders. We have already established in Section 5.10 that speedup estimates derived for turbo codes with four state component encoders also hold for codes with higher state component encoders. Let us now confirm that the same is also true for the performance of the turbo coded system which employs component encoders with more than four states. Figure 5.11 shows the standard turbo encoder for 3G wireless communication systems [30]. It consists of two eight state RSC component encoders. Figures 5.12 and 5.13 show the BER simulations results for turbo coded systems that use the eight state encoder of Figure 5.11. We must mention here that 3G wireless standard

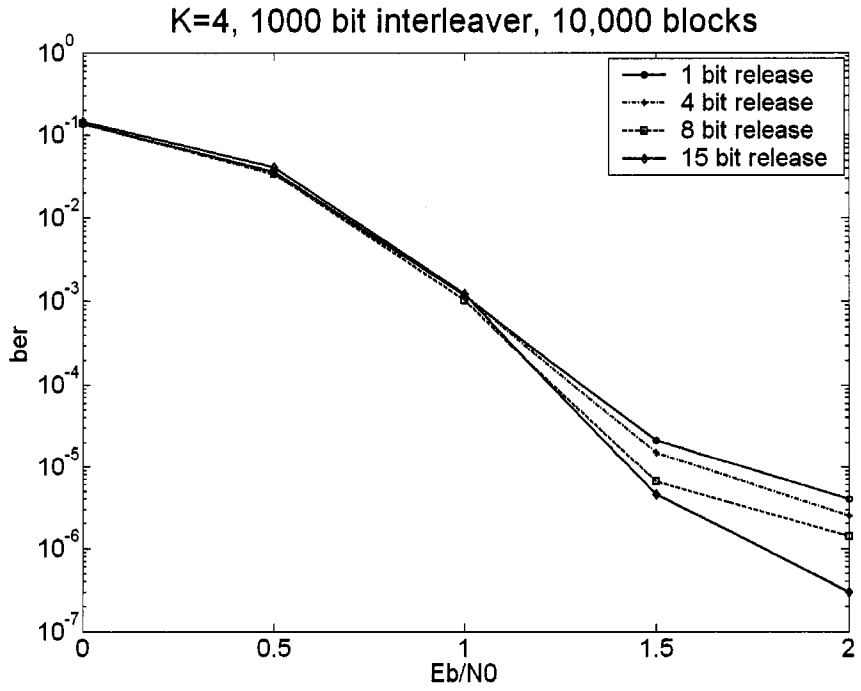


Figure 5.12. BER performance comparison of 8-state bi-directional SOVA

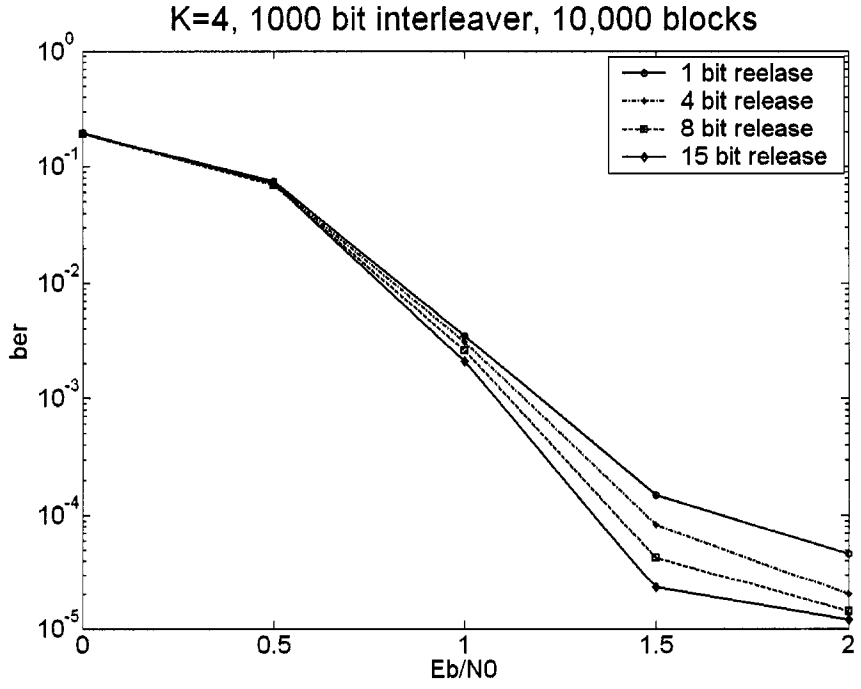


Figure 5.13. BER performance comparison of 8-state Max-Log-MAP

allows multiple frame lengths (between 40 and 5114 bytes), and the exact permutation of the interleaver is determined, based on the frame length according to predefined rules. However we have used a fixed frame size of 1000 bits and a random interleaver in our simulations. The results demonstrate a similar trend as was observed for four state encoders and even better performance than in the case of four state encoders.

5.13 Overall Speedup of the Turbo Decoder

The speedup estimates presented in this chapter are for the component decoders used in a turbo decoder. These component decoders operate in an iterative fashion in a turbo decoder. However the output of a component decoder has to be interleaved, or deinterleaved, before it can be given to the next decoder. The time required for interleaving is relatively small compared to the decoding time of component decoders for long frames (i.e. 1000 bits). If we ignore the time used by the interleavers / deinterleavers then the speedup estimates derived for component decoders also hold for the Turbo decoder. For example if a 15 bit release implementation of bi-directional SOVA provides a speedup of 2.65 over single bit release bi-directional SOVA, then a turbo decoder based on 15 bit release implementation will be approximately 2.65 times faster than the turbo decoder using single bit release bi-directional SOVA.

5.14 Summary

In this chapter we have presented the performance comparison of turbo coded systems with multiple bit release component decoders. We simulated the performance of these

systems and demonstrated that multiple bit release implementations can be used to increase the decoding speed without any degradation in performance. In case of Max-Log-MAP and bi-directional SOVA the performance actually improved slightly by releasing multiple bits. The comparison of turbo coded systems with different component decoders also established the superiority of multiple bit release bi-directional SOVA over multiple bit release Max-Log-MAP in speed as well as performance. Finally we extended our simulations to punctured turbo codes and turbo codes with higher state encoders. The speed and performance advantages obtained by using multiple bit release component encoders were confirmed in these systems as well.

Chapter 6

Conclusions

In this thesis design issues related to the implementation of high speed, low complexity turbo decoders have been investigated. Chapters 1 and 2 provided the background on error correcting codes. Chapter 3 explained the iterative decoding of turbo codes along with the description of several decoding algorithms. Sliding window decoding, which is used to reduce the decoder memory requirements, was presented in chapter 4. To increase the decoding speed, multiple bit release sliding window turbo decoders were also proposed in this chapter. Chapter 5 examined the BER and speed performance of the proposed multiple bit release decoders. The speedups obtained from these implementations and their effects on the decoder's performance were also discussed. Two publications resulted from this work: [29] and [31].

The results obtained in chapter 4 (Figure 4.2) demonstrate that sliding window decoding can be used to reduce decoder memory requirements without significant

performance degradation. Building on these results, multiple bit release sliding window implementations for SOVA, bi-directional SOVA and Max-Log-MAP based turbo decoders have been considered. These implementations sought to increase the decoder speed without affecting its performance. The BER simulation results in chapter 5 proved that it is possible to increase the speed and reduce the computational complexity of SOVA, bi-directional SOVA and Max-Log-MAP based turbo decoders through the proposed modifications. A comparative analysis of these results indicated that while considerable speedups can be achieved in SOVA based turbo decoder, bi-directional SOVA and Max-Log-MAP based turbo decoders are more suitable for such implementations. Bi-directional SOVA, due to its higher decoding speed and slightly better performance than Max-Log-MAP, proved to be the most suitable algorithm for multiple bit release sliding window implementations of turbo decoders.

The above results were also confirmed with punctured turbo codes and turbo codes with different constraint lengths (or encoder states). The increase in speed and performance was obtained at the expense of certain modifications which require extra hardware i.e. memory to store extra trellis stages and logic to release multiple bits. However this increase was modest and was greatly outweighed by the gain in the decoding speeds. The results obtained in this thesis argue strongly in favor of multiple bit release sliding window implementations of turbo decoders due to their reduced computational complexity, improved performance and faster decoding speeds.

6.1 Future Work

The purpose of this study was to research the design of fast and low complexity turbo decoders for future wireless networks. Therefore, implementation of multiple bit release sliding window turbo decoders, analyzed in chapter 5, in silicon (ASIC/FPGA) would be a natural progression of this work. Recent advances in turbo coding have led to the emergence of several new techniques. Multiple bit release sliding window decoders can also be used to the benefit of these techniques. Some suggestions are:

- Multiple turbo codes presented in [32] have recently been shown to outperform conventional turbo codes. A multiple turbo encoder consists of 3 or more simple component encoders and the turbo decoder consists of the same number of component decoders. Since the component decoders in multiple turbo codes are the same as in conventional turbo codes, the component decoders explained in chapter 5 can be used in a multiple turbo decoder to increase its speed. However the performance of the multiple turbo decoder which employs these multiple bit release sliding window component decoders will have to be verified through BER simulations.
- To increase the speed of decoding, a method has been proposed in [33] to divide the received frame in smaller slices and then decoding them in parallel. If the size of a single slice is large enough i.e. multiple decoding windows, then multiple bit release decoding can be employed in each slice independently leading to an overall increase in decoding speed.

References

[1] C. Berrou, A. Glavieux, and P. Thitimasjshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes,"in *Proc., IEEE Int. Conf. on Commun.*, (Geneva, Switzerland), pp. 1064-1070, May 1993.

[2] C.E. Shannon, "A mathematical theory of communication," *Bell Sys. Tech. J.,* vol. 27, pp. 379-423 and 623-656, 1948.

[3] R. W. Hamminf, "Error detecting and correcting codes," Bell *Sys. Tech. J.,* vol. 29, pp. 147-160, 1950.

[4] M.J.E Golay, "Notes on digital coding," *Proc. IEEE,* vol. 37, p. 657, 1949.

[5] S. Wicker, *Error Control Systems for Digital Communications and Storage.* Englewood Cliffs, NJ: Prentice Hall, Inc., 1995.

[6] D. E. Muller, "Application of boolean algebra to switching circuit design," *IEEE trans. on Computers,* vol. 3, pp. 6-12, Sept. 1954.

[7] E. Prange, "Cyclic error-correcting codes in two symbols," Tech. Rep. TN-57-103, Air Force Cambridge Research Center, Cambridge, MA, Sept. 1957.

[8] R.C. Bose and D.K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control,* vol. 3, pp. 68-79, Mar. 1960.

[9] I.S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *SIAM Journal on Applied Mathematics,* vol.8, pp. 300-304, 1960.

[10] E.R. Berlekamp, R.E. Peile, and S.P. Pope, "The application of error control to communications," *IEEE Commun. Magazine,* vol. 25, pp. 44-57, Apr. 1987.

[11] P. Elias, "Coding for noisy channels," *IRE Conv. Record,* vol. 4, pp. 37-47, 1955.

[12] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inform. Theory,* vol. 13, pp. 260-269, Apr. 1967.

[13] 3rd Generation Partnership Project, "Technical specification group radio access network: Multiplexing and channel coding (FDD)." 3GPP TS 25.212 V3.1.0, 1999.

[14] G. D. Forney, *Concatenated Codes*. Cambridge, MA: MIT Press, 1996.

[15] O. Aitsab and R. Pyndiah, "Performance of Reed-Solomon block turbo codes," in *Proc., IEEE GLOBECOM,* (London, UK), pp. 121-125, Nov. 1996.

[16] S. Gravano, *Introduction to Error Control Codes,* New York: Oxford University Press, 2001.

[17] Branka Vucetic and Jinhong Yuan, *Turbo Codes: Principles and Applications,* Norwell, Massachusetts: Kluwer Academinc Publishers Group, 2000.

[18] A.S.Barbulescu and S.S. Pietrobon, "Terminating the trellis of turbo-codes in the same state," *Electron. Lett.,* vol. 31, no. 1, pp. 22-23, Jan. 1995.

[19] M. Breiling and L. Hanzo, "The super-trellis structure of turbo codes," *IEEE Trans. Inform. Theory,* vol. ?, Sep. 2000.

[20] C. Berrou, "Some critical aspects of turbo codes," in *Proc. Int. Symp. Turbo Codes and Related Topics,* (Brest, France), pp. 26-31, Sep. 1997.

[21] L.R.Bahl, J.Cocke, F.Jelinek, and J.Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE trans. Inform. Theory,* vol. ?, pp.284-287, Mar.1974.

[22] J.Hagenauer and P.Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," *IEEE Globecom,* pp.1680-1686, 1989.

[23] J. P. Woodard and Lajos Hanzo, "Comparative study of turbo decoding techniques: An overview," IEEE trans. Vehicular Technology, vol. 49, 2208-2232, Nov. 2000.

[24] J.A.Erfanian, S.Pasupathy, and G.Gulak, "Reduced complexity symbol detectors with parallel structures for ISI channels," *IEEE trans. Commun.,* vol.42, pp. 1661-1671, 1994.

[25] P.Robertson, E.Villebrun, and P.Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. Int. Conf. Communications,* pp. 1009-1013, June. 1995.

[26] G. D. Forney, "The Viterbi algorithm," Proc. IEEE, vol. 61, pp. 268-278, Mar. 1973.

[27] J. Hagenauer, "Source-controlled channel decoding," IEEE Trans. Commun., vol. 43, pp. 2449-2457, Sep. 1995.

[28] J.Chen, M.Fossorier, S.Lin and C.Xu, "Bi-directional SOVA decoding for Turbo-codes," *IEEE Commun. Letters,* vol. CL-4, pp.405-407, Dec. 2000.

[29] Yassir Nawaz, R. Venkatesan and Paul Gillard, "Multiple bit release sliding window turbo decoding," in *Proc. 3rd Int. Symp. Turbo Codes and Related Topics,* (Brest, France), pp. 26-31, Sep. 2003.

[30] IEEE 802.16 Broadband Wireless Access Working Group, "Methods for using concatenated convolutional turbo codes in IEEE 802.16a." IEEE 802.16a-02/80, 2002.

[31] Yassir Nawaz, R. Venkatesan and Paul Gillard, "Sliding Window Implementation of 3G Turbo Decoder," in *Proc. IEEE NECEC,* (St Johns, Canada), pp. ? , Nov. 2003.

[32] P. C. Massey and D. J. Costello Jr., "New low-complexity turbo like codes," in *Proc. IEEE Information Theory Workshop*, (Cairns, Australia), pp. 70-72, Sept. 2001.

[33] D. Gnaedig, E. Boutillon, M. Jezequel, V. C. Gaudet and P. G. Gulak, "On Multiple Slice Turbo Codes," in *Proc. 3rd Int. Symp. Turbo Codes and Related Topics,* (Brest, France), pp. 343-346, Sep. 2003.