# Performance Optimization and Parallelization of Turbo Decoding for Software-Defined Radio

by

Jonathan Leonard Roth

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
September 2009

# Abstract

Research indicates that multiprocessor-based architectures will provide a flexible alternative to hard-wired application-specific integrated circuits (ASICs) suitable to implement the multitude of wireless standards required by mobile devices, while meeting their strict area and power requirements. This shift in design philosophy has led to the software-defined radio (SDR) paradigm, where a significant portion of a wireless standard's physical layer is implemented in software, allowing multiple standards to share a common architecture.

Turbo codes offer excellent error-correcting performance, however, turbo decoders are one of the most computationally complex baseband tasks of a wireless receiver. Next generation wireless standards such as Worldwide Interoperability for Microwave Access (WiMAX), support enhanced double-binary turbo codes, which offer even better performance than the original binary turbo codes, at the expense of additional complexity. Hence, the design of efficient double-binary turbo decoder software is required to support wireless standards in a SDR environment.

This thesis describes the optimization, parallelization, and simulated performance of a software double-binary turbo decoder implementation supporting the WiMAX

standard suitable for SDR. An adapted turbo decoder is implemented in the C language, and numerous software optimizations are applied to reduce its overall computationally complexity. Evaluation of the software optimizations demonstrated a combined improvement of at least 270% for serial execution, while maintaining good bit-error rate (BER) performance. Using a customized multiprocessor simulator, special instruction support is implemented to speed up commonly performed turbo decoder operations, and is shown to improve decoder performance by 29% to 40%.

The development of a flexible parallel decoding algorithm is detailed, with multiprocessor simulations demonstrating a speedup of 10.8 using twelve processors, while maintaining good parallel efficiency (above 89%). A linear-log-MAP decoder implementation using four iterations was shown to have 90% greater throughput than a max-log-MAP decoder implementation using eight iterations, with comparable BER performance. Simulation also shows that multiprocessor cache effects do not have a significant impact on parallel execution times. An initial investigation into the use of vector processing to further enhance performance of the parallel decoder software reveals promising results.

# Acknowledgements

I would like to thank my supervisors, Dr. Naraig Manjikian and Dr. Subramania Sudharsanan, for their combined guidance and expertise throughout this research effort. The wealth of knowledge and direction they have provided me with has made this thesis possible. Over the past two years, their continued support, patience, and dedication has inspired me both personally and professionally, and I'm sure it will continue to do so in my future endeavours.

This research has been supported by an R. S. McLaughlin Fellowship from Queens University. Additional support for this research has been provided by the Natural Sciences and Engineering Research Council of Canada, Communications and Information Technology Ontario, CMC Microsystems, and Sun Microsystems. The enhancements for more accurate modelling in the existing SimpleScalar-based multiprocessor cache simulator used in this research were implemented separately by Ronny Pau.

Finally, I would like to thank my family for their encouragement and support throughout the duration of the Masters program. Many thanks are owed to my friends, for keeping me distracted and entertained when I needed a break from research. I thank my girlfriend, Cara, for her support throughout this journey.

# Table of Contents

**Abstract** i

**Acknowledgements** iii

**Table of Contents** iv

**List of Tables** vii

**List of Figures** ix

**Chapter 1: Introduction** . . . . . 1

1.1 Contributions . . . . . 6

1.2 Organization . . . . . 8

**Chapter 2: Background** . . . . . 9

2.1 Software-Defined Radio . . . . . 9

2.2 Forward Error-Correcting Codes . . . . . 13

2.3 Decoding Turbo Codes . . . . . 17

2.4 Computer Architecture Design Considerations for SDR . . . . . 23

2.5 Chapter Summary . . . . . 28

**Chapter 3: Double-Binary Turbo Codes** . . . . . 29

3.1 The WiMAX Turbo Code . . . . . 30
3.2 Decoding Double-Binary Turbo Codes . . . . . 33
3.3 The log-MAP Algorithm . . . . . 35
3.4 The max* Operator . . . . . 37
3.5 Tailbiting Turbo Codes . . . . . 39
3.6 The Sliding Window Decoding Technique . . . . . 40
3.7 Chapter Summary . . . . . 42

**Chapter 4: Decoder Adaptation and Optimizations . . . . . 43**

4.1 Decoder Adaptation . . . . . 44
4.2 Reduction of Execution Overhead . . . . . 46
4.3 Integer Implementation . . . . . 53
4.4 Simulated Decoder BER Performance . . . . . 54
4.5 Simulated Execution Performance . . . . . 59
4.6 Chapter Summary . . . . . 70

**Chapter 5: Enhanced Turbo Decoding . . . . . 71**

5.1 Special Instruction for Trellis Look-up . . . . . 72
5.2 Accelerating the max* Calculation . . . . . 73
5.3 Parallel MAP Decoding . . . . . 75
5.4 Multithreaded Turbo Decoder Design . . . . . 78
5.5 Optimization of the Parallel Decoder . . . . . 80
5.6 More Accurate Modelling of Execution . . . . . 81
5.7 Vectorizing the Turbo Decoder . . . . . 83
5.8 Chapter Summary . . . . . 86

**Chapter 6: Enhanced Turbo Decoding Results . . . . . 88**

6.1 Custom Multiprocessor Simulator . . . . . 89

6.2 Baseline Decoder Performance . . . . . 91

6.3 Speedup from Special Instructions . . . . . 92

6.4 Parallel Decoder Performance . . . . . 95

6.5 Modelling Multiprocessor Caches and Memory . . . . . 100

6.6 Vectorized Turbo Decoder Initial Results . . . . . 105

6.7 Chapter Summary . . . . . 113

**Chapter 7: Conclusion . . . . . 115**

7.1 Future Work . . . . . 118

**Bibliography . . . . . 121**

**Appendix A: WiMAX Turbo Code Standard . . . . . 126**

**Appendix B: Additional Execution Performance Results . . . . . 128**

**Appendix C: Additional Parallel BER Performance Results . . . . . 130**

# List of Tables


2.1 Mobile device standards. . . . . 11

4.1 Execution times (s) and speedups associated with optimizations for decoding QPSK (6,12) frames. . . . . 65

4.2 Execution times (s) and speedups associated with optimizations for decoding QPSK (36,72) frames. . . . . 65

6.1 Execution cycles and cycles/bit (both in thousands) for the max-log-MAP and linear-log-MAP baseline decoders. . . . . 92

6.2 Simulation statistics of the parallel max-log-MAP turbo decoder for decoding 18-byte frames (cycles in thousands). . . . . 98

6.3 Simulation statistics of the optimized parallel max-log-MAP turbo decoder for decoding 18-byte frames (cycles in thousands). . . . . 98

6.4 Ideal, best-case and worst-case time (in thousands of cycles) when decoding QPSK (18,36) frames. . . . . 104

6.5 Estimated and actual execution cycles and speedup for vectorized $\Gamma$ calculations. . . . . 108

6.6 Vector instructions needed for vectorized $\Gamma$ calculations. . . . . 109

6.7 Vector speedup and efficiency using 4 threads. . . . . 113

A.1 CTC channel coding block sizes and modulation schemes. . . . . 127

B.1 Execution times (s) and speedups associated with optimizations for decoding QPSK (18,36) frames. . . . . 129

B.2 Execution times (s) and speedups associated with optimizations for decoding QPSK (60,120) frames. . . . . 129

# List of Figures


2.1 An example of a convolutional encoder. . . . . 15

2.2 Basic structure of a turbo encoder. . . . . 16

2.3 Turbo decoder structure. . . . . 18

3.1 WiMAX turbo encoder. . . . . 31

3.2 WiMAX turbo code trellis diagram. . . . . 32

3.3 WiMAX turbo decoder. . . . . 34

3.4 The sliding window technique using four sub-blocks, with the details of a single sub-block shown. . . . . 41

4.1 Optimized turbo decoder software structure. . . . . 52

4.2 BER performance of the original, optimized, and integer decoders. . . 56

4.3 BER performance comparison of rate-1/2 QPSK frames, and rate-3/4 QPSK and 16-QAM frames. . . . . 57

4.4 BER comparison of max* operators for decoding a QPSK (36,72) frame using eight iterations. . . . . 58

4.5 BER performance comparison of the max-log-MAP and linear-log-MAP decoders for decoding a QPSK (36,72) frame. . . . . 60

4.6 Decoding time/bit for QPSK (6,12) frame. . . . . 62

4.7 Decoding time/bit for QPSK (36,72) frame. . . . . 63

4.8 Function contribution to overall execution time. . . . . 69

5.1 The parallel double-window decoding scheme. . . . . 77

5.2 Parallel turbo decoder design following optimizations. . . . . 81

6.1 BER performance of the parallel turbo decoder. . . . . 96

6.2 Execution cycles comparison between max-log-MAP and linear-log-MAP decoders. . . . . 99

6.3 Multiprocessor cache miss rate for different cache sizes. . . . . 102

6.4 Multithreaded execution performance with caches. . . . . 105

6.5 C code for calcGamma. . . . . 106

6.6 Scalar assembly code for calcGamma. . . . . 107

6.7 Vector assembly code for calcGamma. . . . . 111

6.8 Preliminary execution results for vector decoding. . . . . 112

C.1 Parallel BER performance of the linear-log-MAP decoder. . . . . 131

# Chapter 1

# Introduction

Mobile devices are increasing in complexity due to the demand to integrate a wide range of features in a compact and power-efficient design. Multiple wireless standards are required to implement the functionality desired in mobile devices, particularly in the high-end market segment, such as handheld computers and smartphones. Wireless standards employed by mobile devices are used for communication with many different types of devices and networks [Ram07], including:

- wireless peripherals, e.g. Bluetooth,
- personal computers (PC), e.g. wireless USB (WUSB), ultra-wideband (UWB),
- wireless local area network (WLAN), e.g. 802.11 a/b/g/n,
- mobile Internet, e.g. Worldwide Interoperability for Microwave Access (WiMAX),
- mobile data/voice, e.g. 3G wideband code division multiple-access (W-CDMA), 4G long-term evolution (LTE), and

- mobile television, e.g. Digital Video Broadcasting for Handhelds (DVB-H).

Until recently, application-specific integrated circuits (ASICs) were the standard architecture to implement wireless transmitters and receivers in mobile devices. Implementing multiple standards in a device involved using a separate ASIC for each wireless standard, with minimal co-operative design. With the multitude of standards in use today, this design strategy can result in increased hardware complexity, cost, chip area, and power consumption.

The multitude of current and evolving wireless standards used by mobile devices has led to a shift away from using fixed ASICs to using more flexible architectures. Flexible architectures offer the benefit of being able to support multiple wireless standards, while allowing them to share hardware that implements similar functionality. This may require additional planning to identify common features implemented by different standards when designing a flexible hardware architecture. However, the potential benefits of increased flexibility, lower power consumption, and reduced chip area would be a worthwhile trade-off. This is especially true as only a few simultaneous standards are required at any given time in a mobile device. As a result, various reconfigurable architectures have emerged that can consist of reconfigurable data processing units and/or field-programmable gate arrays (FPGAs), depending on the chosen function granularity [Ram07, Nik05]. Reconfigurable architectures implement the elementary algorithms common among standards, such as forward-error correction or fast Fourier transform (FFT). Such an architecture offers some flexibility, but the diverse range of standards and algorithms present may still result in significant chip area and power consumption, limiting their benefit over ASICs. As transistors shrink in size, programmable multiprocessor architectures are becoming increasingly

attractive solutions, as software would implement the desired functionality, providing a high degree of flexibility.

The use of flexible, software-based wireless architectures has led to the emergence of the software-defined radio (SDR) paradigm, in which a significant portion of the physical-layer functionality of a wireless standard is implemented in software [Bur00]. A more general-purpose multiprocessor architecture offers the benefit of a simpler programming model than partially reconfigurable designs and a high-degree of hardware reuse for multiple protocols, hence SDR is well-suited for multiprocessors [Lin06b, Baa07]. SDR would support a protocol's continuing evolution, including prototyping and bug fixes, which would be easily implemented through software without requiring redesigned hardware. These benefits would shorten protocol implementation's time to market, and because hardware and software development could proceed in parallel. SDR requires efficient and flexible hardware and software co-design to ensure performance requirements are met, to maximize the flexibility of the system as a whole, and to proactively anticipate the need for future change.

The functionality of a wireless transceiver (combined transmitter and receiver) can be grouped into two broad categories: baseband functionality, which performs operations in the digital domain using digital hardware, and radio frequency functionality, which typically performs processing in the frequency domain using analog hardware. The baseband functionality of a wireless protocol can be grouped into four major components: filtering, modulation, channel estimation, and error correction [Erg09]. Filtering suppresses interference from frequencies transmitted outside the allowed frequency band. Modulation maps source bits to complex-valued symbols associated with the chosen modulation scheme and signal waveform for transmission,

while the receiver demodulates the signal to retrieve the source information. Channel estimation is performed at the receiver side to estimate channel conditions and synchronize transmission with the transmitter. Error correction is used to obtain reliable communication in the presence of noise. The transmitter encodes the original data sequence using a particular coding scheme. Encoding inserts redundant information into the data sequence, while the receiver uses this redundant information to more reliably detect the original data.

Mature convolutional codes are used as forward error-correction codes in the majority of wireless standards due to their simple implementation and good performance. More recently, however, turbo codes have gained popularity because they offer superior bit-error rate (BER) performance in comparison to other channel codes for forward error-correction [Ber93]. Turbo codes utilize convolutional codes as component codes in a parallel concatenation scheme, with two convolutional encoders/decoders used to cooperatively encode/decode a frame. Decoding proceeds in an iterative nature, and component decoders exchange information to increase BER performance with each additional iteration. A simplified variant of the maximum *a posterior* (MAP) algorithm is utilized in each component decoder [Rob95]

WiMAX, an emerging wireless radio standard [IEE04], relies on enhanced double-binary turbo codes, which offer better BER performance compared to the original binary turbo codes [Ber99]. Turbo decoding is one of the most computationally-intensive functions in a wireless communications system, and has been shown to contribute upwards of 40% to the total computational complexity of the physical layer of a wireless standard [Lin06b]. Thus, to support next-generation wireless standards such as WiMAX in a SDR environment, an efficient software implementation of a

double-binary turbo decoder is desirable, particularly as microelectronics technology enables the integration of multiple processors in a single chip to support parallel execution.

Various turbo decoder implementations exist that use architectures with varying degrees of flexibility, some of which are described in the following chapter. There have been many research efforts pursuing ASIC turbo decoder implementations [Kim08, Lin08], however these designs are dedicated to a single standard and offer no flexibility. Reconfigurable turbo decoding architectures offer some advantages over ASICs [Lin06a, MC07, Vog08], but typically support only a few standards at most. Using a general multiprocessor SDR architecture offers the greatest amount of flexibility for a transceiver functionality [Baa07, Lin07]. However, to the best of our knowledge, there exists limited research into the design and enhancement of turbo decoding software for execution on such an architecture. Some turbo decoder architectures use a parallel approach, but their degree of parallelism is fixed.

This thesis describes the efficient implementation of double-binary turbo decoder software suitable for SDR. Turbo decoding comprises a significant portion of a wireless standard's computational requirements, particularly for enhanced double-binary turbo codes. The target platform is a general-purpose multiprocessor architecture, which offers the most flexibility of the wireless radio hardware architectures described above. This effort was initiated by conducting a thorough analysis of the double-binary turbo decoding algorithm, and the insight obtained was applied towards an efficient software implementation that supports the WiMAX standard. Open-source turbo decoding software suitable for simulation purposes was selected as the basis for

this implementation. The software was adapted for stand-alone execution on a multiprocessor system, and various software and hardware-oriented optimizations were applied to enhance decoding performance. Throughout the evolution of the turbo decoder, the execution performance was quantified, and the bit-error rate (BER) performance was measured through simulation to ensure that acceptable performance was maintained.

## 1.1 Contributions

The contributions of this thesis can be broken into three broad categories. First, a thorough description of the turbo decoding algorithm is presented, for use in the WiMAX standard. Second, various enhancements to the turbo decoding software implemented for this research effort are presented in detail, and their effects on both execution and BER performance are discussed. Finally, a multiprocessor simulator is used to evaluate various hardware-oriented enhancements to support turbo decoding, including special instructions, parallel execution, and an initial investigation into vectorization. The work presented in this thesis can be applied to any turbo decoding standard to increase execution performance of an SDR implementation, including various architectural enhancements. The following paragraphs provide more details for each contribution.

Although there are numerous publications describing the turbo decoding algorithm in a variety of ways, the details of the algorithm can be hard to grasp and understand, particularly without prior knowledge of turbo decoding. As a result, an entire chapter in this thesis is dedicated to providing a clear and concise explanation of the turbo decoding algorithm as applied to the double-binary WiMAX turbo code

standard, and various design details that must be considered for implementation. The double-binary turbo code presentation was distilled from a variety of sources, and a uniform approach for mathematical notation was developed. A thorough understanding of the turbo decoding algorithm was certainly required when pursuing the various software and hardware enhancements presented in this thesis.

An open-source turbo decoder implementation in MATLAB with C-language extensions was adapted for a stand-alone C implementation, used throughout this work. Numerous software enhancements are presented, and their resulting performance improvements are quantified. The computational efficiency of the turbo decoder was significantly increased through various optimizations, while maintaining the same BER performance. An integer decoder was also implemented in software, which typically offers better execution performance on processor architectures, while only experiencing slight BER degradation. Two decoder implementations using different variants of the maximum *a posteriori* (MAP) decoding algorithm were compared, demonstrating that a more complex variant can achieve better execution performance using fewer iterations while maintaining the same BER performance.

Following software optimization of the turbo decoder, various hardware-oriented enhancements are presented for increased execution performance. Special instruction support was implemented for commonly-performed decoder operations, demonstrating further increases in execution performance. A parallel version of the turbo decoder was designed and implemented in software with a flexible degree of parallelism, demonstrating good parallel efficiency, while exhibiting only slight BER degradation. Finally, an initial investigation into the vectorization of the decoder software was pursued, showing promising results of a completely vectorized version of the software

running on a vector processor architecture.

## 1.2 Organization

This thesis is organized as follows. Chapter 2 presents background information related to turbo decoding in a SDR environment, including the motivation behind SDR and supporting architectures, and design details of some dedicated hardware and SDR turbo decoder implementations. Next, Chapter 3 presents a detailed discussion of the WiMAX turbo code standard, including the encoding process, the decoding of double-binary turbo codes using different variants of the MAP algorithm, and a popular implementation scheme. Various optimizations applied to the turbo decoding software, and their resulting effect on execution and BER performance, are discussed in Chapter 4. Hardware-oriented enhancements to improve decoding performance on a general multiprocessor architecture are discussed in Chapter 5, including special instruction support, a flexible parallel decoder design, and the potential benefit of vectorization. Chapter 6 presents implementation details and the resulting improvement in execution performance of the hardware-oriented enhancements as measured with simulation. Finally, conclusions and future work are discussed in Chapter 7.

# Chapter 2

# Background

This chapter provides an overview of important literature related to this research effort. Important aspects related to software-defined radio (SDR) and the motivation for its emergence, as well as important design details of two SDR architectures, are presented in Section 2.1. Next, an overview of forward error-correcting codes, and the details behind convolutional and turbo codes, are presented in Section 2.2. A brief explanation of decoding turbo codes, as well as various dedicated and SDR hardware implementations, are presented in Section 2.3. Finally, some background information related to computer architecture pursuant to the SDR system presented in this thesis is discussed in Section 2.4.

## 2.1 Software-Defined Radio

Software-defined radio (SDR) implements a significant portion of a wireless standard's physical layer in software, as opposed to dedicated hardware [Bur00]. SDR offers greater flexibility than application-specific integrated circuit (ASIC) designs,

which, until recently, have dominated implementation efforts of wireless devices. The demand for flexibility has been increasing with the advent of a greater number of wireless standards in use. There are a multitude of networking standards for mobile devices in use today, and supporting multiple standards has become commonplace in the mid- and high-end mobile device market. Standards currently employed by mobile devices include Global System for Mobile Communications (GSM), Wideband Code Division Multiple Access (W-CDMA), and High-Speed Downlink Packet Access (HSDPA), for dedicated mobile access, wireless Internet standards 802.11a/b/g/n, mobile TV standards including Digital Video Broadcasting-Handheld (DVB-H), as well as Bluetooth technology supporting local connections to headsets and other equipment. A number of evolving next-generation standards include Worldwide Interoperability for Microwave Access (WiMAX), Long-Term Evolution (LTE), and Wi-Fi ultra-wideband (UWB). WiMAX is a telecommunications technology that provides wireless transmission of data using a variety of transmission modes for mobile devices. Table 2.1 lists a number of current and evolving mobile device standards (adapted from [Ram07]).

Making use of programmable, more general-purpose hardware allows different standards to share a common architecture, reducing power consumption, device cost, etc. SDR aims to utilize a common hardware platform that can support multiple wireless protocols implemented and controlled by software. This strategy reduces the hardware needs of using a different ASIC design for each implemented wireless standard. Because only a few simultaneous standards would be utilized in mobile devices, reduced chip area, power savings, and flexibility resulting from using flexible

Table 2.1: Mobile device standards.

| Specification | Application | Range | Rate |
|---|---|---|---|
| **Wi-Fi ultra-wideband** | High-speed local interconnect, wireless USB | 10 m | 480 Mbps |
| **802.11a/b/g/n** | Wireless LAN | 80 m (a/b/g), 50-150 m (n) | 11 Mbps (b), 54 Mbps (a/g), 100-600 Mbps (n) |
| **W-CDMA** | Mobile wireless access | 1-5 km | 21 Mbps |
| **Mobile WiMAX** | Mobile wireless access | 1-5 km | 63 Mbps |
| **3G LTE** | Mobile data/voice | 1+ km | 100 Mbps |
| **DVB-H** | Mobile TV | Broadcast | 384 Kbps |
| **DVB-T** | Mobile TV | Broadcast | 7 Mbps |

architectures would minimize any increases in power and area due to reconfigurability. As suggested above, there is a demand for multi-standard mobile device support and alternative approaches to ASIC-centred and DSP-assisted architectures supporting SDR [Ram07]. More general architectures present different degrees of flexibility. Processor-centred designs present the highest degree of flexibility, the highest power consumption, but a simpler programming model. Reconfigurable architectures offer reduced area and reduced power, but offer limited flexibility and a complex programming model. ASIC-centred devices offer the lowest power consumption, but no flexibility. As transistors shrink in size, multiprocessor-based SDR-enabled mobile devices present an increasingly feasible architecture in terms of performance, chip area, and power consumption. The first commercial SDR programmable baseband multiprocessor, X-GOLD SDR 20 from Infineon Technologies, became available in the Spring of 2009, with support for GSM, GPRS, EDGE, W-CDMA, HSDPA, LTE,

802.11 a/b/g/n, DVB-T/H [Inf09].

There have been some recent notable efforts aimed at software-defined radio implementations. The Signal-Processing On-Demand (SODA) architecture is a fully programmable digital signal processor (DSP) architecture that supports software-defined radio [Lin06b, Lin07]. The system consists of one controller and four ultra wide single-instruction-multiple-data (SIMD) processing elements, with data communication done through explicit direct memory access (DMA) instructions. This type of system was the first proposed with energy characteristics suitable for mobile computing. The advantages of SDR were illustrated through the implementation of two widely differing protocols: W-CDMA and 802.11a. These two protocols were implemented in C, and were able to operate within the the strict power constraints of a mobile terminal on the SODA architecture.

The Asynchronous Array of simple Processors (AsAP) is a chip multiprocessor composed of simple processors with small memories suitable for SDR [Baa07]. The system operates with globally asynchronous, locally synchronous (GALS) clocking, and a nearest neighbour mesh interconnect. It was shown to achieve greater performance per area and better energy-efficiency than current DSPs, while being many times smaller. GALS clocking allows processor clock speeds to be precisely adjusted to match the workload of individual processors, contributing to the power efficiency of the system. AsAP is architecturally different than SODA, as its aims were to provide task-level parallelism as opposed to data- and instruction-level parallelism. The system's performance was illustrated through implementing an IEEE 802.11a/g wireless LAN transmitter using 22 processors, which was shown to offer greater than five times the performance and more than 35 times lower energy dissipation for the

same transmitter running on a commercially available DSP. The performance of both the SODA and AsAP general-purpose SDR architectures show promising potential for the future of SDR-enabled mobile devices.

## 2.2 Forward Error-Correcting Codes

When bits are transmitted over a noisy wireless channel, errors are likely to occur. To enable errors to be detected and corrected, wireless transmissions use forward error-correction (FEC) codes. FEC codes introduce redundant information in the binary information sequence before transmission, a task performed by an encoder, and can be classified as either convolutional or block codes. Block codes work with fixed-sized blocks of bits, or frames, of a predetermined size, while convolutional codes work on frames of an arbitrary size. When errors occur during transmission, the decoder at the receiver side is able to detect and correct bit errors, up to some maximum number, in the received sequence. FEC enables high data rates in the presence of fading and interference from other wireless channels. Described below are two types of FEC codes: convolutional codes, which are supported by most wireless standards, and turbo codes, an improved FEC scheme making use of convolutional codes.

### 2.2.1 Convolutional Codes

Convolutional codes are a popular type of forward error-correcting codes, and are supported by the majority of wireless standards [Erg09]. Decoding convolutional codes is performed using the Viterbi algorithm, first introduced in 1967, which has a fixed decoding time [For73]. A convolutional encoder performs forward error-correction by

introducing parity bits associated with a binary input sequence using a shift-register process and modulo-2 adders. A shift register is simply a chain of flip-flops or storage elements connected together with the output of one flip-flop connected to the input of the next flip-flop. At every clock edge, the first flip-flop captures a new value provided to it, while the rest of the values are shifted by one position.

A convolutional encoder is typically classified in terms of code rate, constraint length, one or more generator polynomials, and a feedback polynomial. The code rate is $R = k/n$, where $k$ is the number of information bits provided to the encoder, while $n$ is the total number of bits to be transmitted after encoding. An encoder consists of at least $n$ modulo-2 adders, and the generator polynomials specify the shift-register connections to the adders. If there is no feedback polynomial, the code is *recursive*, otherwise it is *non-recursive*. If the input to the encoder is included in the output sequence in addition to the parity bits, the code is said to be *systematic*; otherwise it is a *non-systematic* code. The constraint length parameter is $K$, which is the number of cycles an input bit affects the output of the encoder.

As input bits are provided to the encoder, the values contained in the flip-flops change, and it is these values that correspond to the encoder state. An encoder with $m$ flip-flops has $2^m$ states. Each input bit provided to the encoder has an effect on $K$ consecutive output symbols. It is this characteristic that gives convolutional codes their error-correcting power. Figure 2.1 provides an example of a non-recursive, non-systematic convolutional encoder. The polynomials defining the connections to the modulo-2 adders for parity bits $x^{p_1}$, $x^{p_2}$, and $x^{p_3}$ are $G_1 = 1 + D + D^2$, $G_2 = D + D^2$, and $G_3 = 1 + D^2$, respectively, in symbolic notation. There is no feedback polynomial. The encoder has rate $R = 1/n = 1/3$ and constraint length $K = 3$.

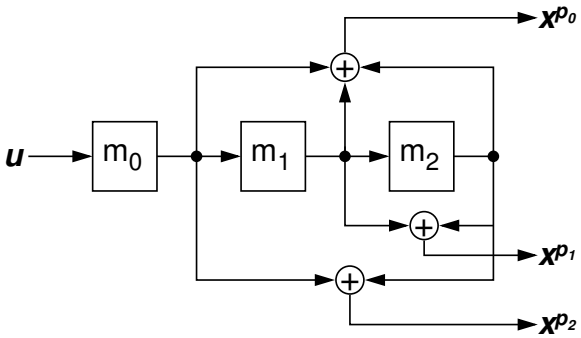


Figure 2.1: An example of a convolutional encoder.

### 2.2.2 Turbo Codes

Turbo codes, first introduced in 1993, use two or more recursive systematic convolutional (RSC) codes that are concatenated in parallel via an interleaver [Ber93]. Turbo codes have been shown to offer superior bit-error rate (BER) performance to that of convolution codes because of their structure based on interleaving in conjunction with concatenated coding and iterative decoding, at the expense of increased complexity. Many current and evolving wireless standards support the use of binary turbo codes, including UMTS (using W-CDMA), HSDPA, and LTE. The basic structure of a binary turbo encoder, using two RSC encoders in parallel, is shown in Figure 2.2. For each information bit provided to the encoder, $u$, two parity bits are generated, $x^{p_1}$ and $x^{p_2}$. The systematic bit $x^s$ is simply equal to the information bit, giving an overall encoder rate of $R = 1/3$.

The output sequence produced by the turbo encoder is typically interleaved and punctured. Interleaving improves BER performance in the presence of noise bursts, by reordering the coded bits to be transmitted in a pseudo-random fashion. Burst errors will affect some consecutive number of transmitted symbols. BER performance

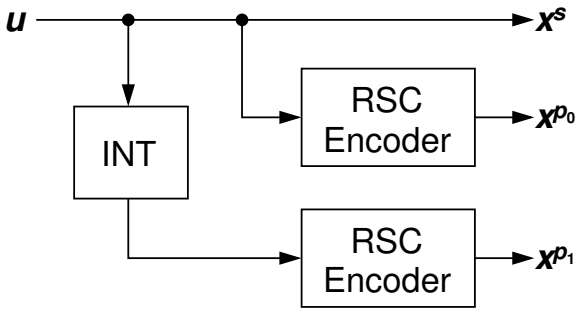


Figure 2.2: Basic structure of a turbo encoder.

is improved because prior to decoding, the received symbols are de-interleaved, therefore spreading burst errors over the entire transmitted sequence. Puncturing simply removes some of the coded bits in a sequence, typically parity bits, to increase the rate of the code that is actually transmitted and thereby use the channel more efficiently.

A recent enhancement to turbo codes is the use of double-binary symbols as opposed to single bits when encoding and decoding, which offers better BER performance by further reducing correlation effects, at the expense of increased complexity [Ber99]. The WiMAX standard and the DVB-for-return-channel-via-satellite (DVB-RCS) standard both support enhanced double-binary turbo codes [IEE04, Dou00]. The details of the WiMAX turbo code standard and encoder are discussed in detail in Section 3.1. Encoding of binary turbo codes is performed by feeding single bits to the encoder sequentially, whereas encoding double-binary turbo codes is performed by encoding two bits in parallel. The computational complexity of the encoder is much lower in comparison with the computational demand required by turbo decoders.

Decoding turbo codes is performed in an iterative manner using two soft-input soft-output (SISO) RSC component decoders connected by an interleaver and de-interleaver, which exchange information to cooperatively decoder a frame. Decoding performs optimally with uncorrelated symbols, and interleaving and de-interleaving

provides a sequence of symbols to each component decoder that are largely independent. Binary turbo decoding has been shown to contribute, depending on the implementation platform, around 40% to the total computational complexity of the physical layer [Lin06b], while double-binary turbo decoding would contribute more due to its increased complexity. For supporting turbo decoding in a software-defined radio environment (SDR), an efficient software implementation is critical to achieving optimal performance in an SDR device. An overview of the turbo decoding algorithm, as well as various hardware and SDR decoder implementations, are discussed in the next section.

## 2.3 Decoding Turbo Codes

A typical turbo decoder consists of two soft-input soft-output (SISO) component decoders that are serially concatenated via an interleaver and de-interleaver [Ber93, Val01]. Each component decoder implements either the soft-output Viterbi algorithm (SOVA) or the maximum *a posteriori* (MAP) algorithm. The MAP algorithm is the preferred algorithm for turbo decoding, as it offers better BER performance when compared with the SOVA. Turbo decoder implementations typically use a simplified version of the MAP algorithm, referred to as log-MAP, which deals with the logarithm of the probabilities used in the calculations, rather than the probabilities themselves, to avoid expensive multiplication and division operations [Rob95]. Various log-MAP variants exist, depending on the chosen max* operator, an operation used extensively throughout log-MAP decoding. The particular max* implementation chosen affects both BER performance and decoder complexity, as discussed in detail in Section 3.4. The simplest implementation is the max-log-MAP algorithm, common among

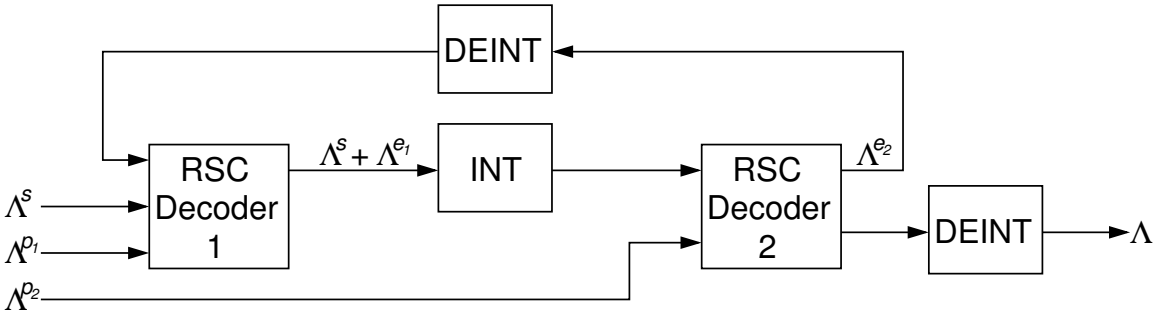


Figure 2.3: Turbo decoder structure.

turbo decoder implementations. Turbo decoding for WiMAX is discussed in detail in Chapter 3, but a brief overview of the decoding process is provided here. The basic turbo decoder structure is illustrated in Figure 2.3 [Ber93].

The ultimate goal of the turbo decoder is to find the *a posteriori* probability of the information bits, $\Lambda$. The inputs to the encoder are soft-input probability values, associated with the coded output of the encoder after transmission over a noisy channel. The probabilities are $\Lambda^s$, $\Lambda^{p_1}$, and $\Lambda^{p_2}$ associated with the systematic bits, parity bits for RSC decoder 1, and parity bits for RSC decoder 2, respectively. A turbo decoder typically performs a specified number of iterations, before making hard bit decisions. Each iteration consists of two half-iterations, during which only one RSC decoder is performing log-MAP decoding. After each half-iteration, *extrinsic* information is provided to the other RSC decoder, $\Lambda^{e_1}$ and $\Lambda^{e_2}$ in Figure 2.3, before the next half-iteration begins. It is this exchange of information between component decoders that improves the BER performance of the decoder with each additional iteration. In implementations, only one component decoder needs to actually be implemented because they operate in a serial manner. Once a specified number of iterations have been performed, the *a posteriori* probabilities of the information bits, $\Lambda$, are used to make hard bit decisions.

Decoding using the log-MAP algorithm involves computing a number of metrics associated with each information bit in a frame, using the soft-input values provided to the decoder. Two metrics of interest include $A$ metrics, which are defined recursively in the *forward* direction, and $B$ metrics, which are defined recursively in the *reverse* direction. The sliding window technique is used to reduce overall memory requirements, as explained in detail in Section 3.6, and it is commonly used among hardware implementations. Essentially, a frame is split into a number of sub-blocks, and log-MAP decoding is applied to each sub-block independently. Memory requirements are reduced, as metrics only need to be stored for the sub-block currently being decoded. Some aggressive implementations use a parallel version of the sliding window algorithm, where sub-blocks are decoded in parallel rather than serially. Dedicated hardware implementations typically use fixed-point number representations to achieve a good trade-off between memory requirements and BER performance.

The interleaver is involved in both encoding and decoding turbo codes, and is dependent on frame size. The most straightforward approach involves storing the interleaver patterns in memory, however this can lead to a prohibitively large memory requirement, resulting in significant silicon area and power consumption. A more advanced approach, which eliminates the need of an interleaver memory, involves generating interleaved addresses on-the-fly [MC07].

### 2.3.1 Dedicated Hardware Implementations

A dedicated double-binary turbo decoder ASIC implementation supporting WiMAX was recently presented, which uses the max-log-MAP algorithm [Kim08]. The sliding window technique is used, and due to the recursive nature and order of the metrics

calculated, $B$ metrics must be initialized at the border of each sub-block. The initialization method used makes use of a border memory to store $B$ metrics at the border of each sub-block, encoded using a small number of bits to reduce the memory size. $B$ metrics are initialized from this border memory using the border values from the previous iteration. The reduced accuracy from encoding the values stored was shown to have a minimal effect on BER performance. Using the border memory was shown to reduce energy consumed by 26.2% over another method that performs additional *dummy* calculations to obtain reliable border metrics with similar BER results. A dedicated hardware interleaver is used to generate interleaved addresses on-the-fly, eliminating the need for an interleaver memory.

A more flexible turbo decoder implementation supporting multiple standards was designed and implemented, consisting of dedicated hardware for SISO decoding and a SIMD processor for interleaving [MC07]. The flexibility needed to support W-CDMA and CDMA2000 turbo codes dictated the need for some degree of reconfigurability, especially for the interleaver, as each standard utilizes a unique and complex interleaver. A hybrid architecture was used, with dedicated hardware performing computationally intensive but regular tasks, and software running on a programmable single-instruction multiple data (SIMD) processor for tasks that require flexibility. The two standards use similar RSC turbo codes, hence dedicated hardware with minimal reconfigurability is suitable for the SISO decoder implementation, while the differing interleavers dictated the need to implement them in software. Again, a sliding window technique is used, while additional *dummy* calculations are performed to obtain reliable $B$ border metrics. The component decoder supports both the max-log-MAP algorithm and an approximated log-MAP algorithm using a look-up table,

with the most suitable one chosen based on the channel signal-to-noise ratio (SNR) estimation.

### 2.3.2 SDR Implementations

The previously described SODA SDR platform implementing W-CDMA used four processing elements (PEs), with one entire PE dedicated solely to turbo decoding, illustrating the computation demanded by decoding [Lin07]. The targeted W-CDMA standard used only binary turbo codes, while double-binary codes present higher computational complexity, requiring additional processing power.

A family of reconfigurable application-specific instruction-set processors (ASIPs), controlled by software and supporting decoding in an SDR environment, has recently been presented [Vog08]. The system supports binary convolutional decoding, and turbo decoding for binary as well as double-binary turbo codes for a number of current and upcoming wireless standards. A thorough analysis of the different decoders utilized in the various mobile wireless standards was conducted, and commonalities were identified and exploited for the decoder's architecture. The end result was an ASIP with a fully customized 15-stage pipeline, with dedicated communication and memory infrastructure, and a specialized instruction set suitable for decoding. Data-level parallelism is provided to perform all required metric calculations for a given frame index in a single cycle. Various sliding window and frame termination schemes are provided for turbo, log-MAP, and Viterbi decoding. The ASIP offers design-time reconfigurability, which allows a customized design to implement only a subset of the supported decoding algorithms, allowing increased performance and reduced area compared with a full implementation. Run-time configurability allows the design to

switch between channel codes with minimal overhead. This aggressive ASIP implementation is able to provide a maximum turbo decoding throughput of 34 Mbps, while consuming significantly less power and area than other SDR decoder architectures. Although this custom turbo decoding architecture offers good performance, it offers less flexibility and a more complex programming model than more general-purpose architectures.

The algorithm-architecture co-design of a turbo decoder supporting SDR using an enhanced DSP was previously presented [Lin06a]. The custom DSP architecture consists of a 32-wide SIMD pipeline, which was chosen to meet the performance requirements of the W-CDMA protocol implemented. In addition, the architecture included a scalar pipeline for sequential operations, and a programmable DMA unit to allow transfers between between the SIMD, scalar, and global memories. A parallel max-log-MAP SISO design was employed, similar to the sliding window technique, where sub-blocks are processed in parallel rather than serially. Additional *dummy* calculations are used to initialize $A$ and $B$ values at the borders of sub-blocks. The details of performing $A$ and $B$ calculations in parallel for four sub-blocks are discussed, as the W-CDMA turbo code has eight states, for a total of 32 simultaneous metric calculations. The custom-DSP architecture was shown to provide 2 Mbps throughput, while consuming sub-watt power.

An efficient UMTS turbo decoder implementation suitable for implementation in an SDR receiver is explained in [Val01]. Variations of the log-MAP decoding algorithm are discussed, implemented, and compared, including a linear approximation that results in only a negligible decrease in BER performance. Some methods that can be employed to stop the decoder dynamically as opposed to using a fixed number

of iterations are also proposed. A method involving the normalization of the $A$ and $B$ branch metrics was presented, which resulted in a memory savings of 12.5%. The BER performance of the decoder was evaluated using different variants of the log-MAP algorithm, including the average number of iterations for each algorithm to converge, as well as various techniques for dynamic halting. The turbo decoder design and the results presented in this work are useful when designing an SDR decoder implementation.

## 2.4 Computer Architecture Design Considerations for SDR

Although there has been some research illustrating the design challenges of SDR-based architectures, they are typically designs dedicated to turbo decoding only. Flexible single-chip architectures that can be used to implement an entire SDR receiver, such as AsAP [Baa07] and SODA [Lin07], will undoubtedly increase in importance as transistor sizes shrink, allowing acceptable power consumption levels required for that of mobile devices. As a result, the optimization of software for computationally-intensive turbo decoders will increase in importance, particularly for double-binary turbo decoders. Various computer architecture considerations to support SDR are discussed briefly below.

### 2.4.1 Multiprocessor System-on-Chip

The design challenges facing SDR turbo decoder implementations differ largely from that of dedicated hardware-based implementations. As discussed, multiprocessor systems consisting of multiple processing-elements are well-suited to SDR implementations, and will increase in importance as transistors continue to shrink in size. Integrating a multiprocessor system on a single chip, referred to as a multiprocessor system-on-chip (MPSoC), is a standard practise for application-specific systems, as it reduces system area, power consumption, and device cost.

Dedicated turbo decoder hardware implementations allow more fine-grained control of the needed calculations, data-flow, and memory operations. SDR architectures cannot achieve the same granularity of control, particularly for general MPSoC architectures that provide a common platform for the entire transmitter and receiver. For example, the majority of dedicated hardware implementations and some reconfigurable architectures assume dedicated $A$ and $B$ processors executing concurrently each with their own dedicated memory. A general multiprocessor architecture requires the calculations for $A$ and $B$ be expressed as individual threads, with memory operations performed with specific load and store instructions. In dedicated hardware implementations, the system would be designed in such a way that prerequisite calculations have been performed, and data is ready. In a general multiprocessor architecture, steps must be taken using synchronization constructs to ensure threads proceed only when appropriate.

The processing elements (PEs) in a multiprocessor system vary in complexity, depending on the design methodology sought. The AsAP system made use of simpler PEs [Baa07], requiring a significant number of them to implement a transmitter or

receiver. Using a number of simpler PEs requires an efficient communication network, particularly to keep the system scalable, hence a mesh topology was chosen. On the other hand, the SODA SDR architecture uses complex PEs, each with 32-wide SIMD pipeline, a scalar pipeline, local SIMD and scalar memories, and a DMA unit [Lin07]. As a result, only four PEs are needed to implement a complete receiver. Communication between PEs uses a simple bus-based network, with communication performed though DMA transfers to a global memory.

### 2.4.2 Memory and Caches

Neither of the two architectures described above make use of caches. However the target architecture used for this work makes use of conventional processing elements, with instruction and data caches, and a single global memory, all connected with a shared bus. Hence, an explanation of caches and the interaction between caches and memory and their effect on performance in a multiprocessor system is in order. A brief discussion is provided here, however a more thorough discussion on the subject is available [Pat05].

Introducing caches into a processor reduces delays associated with reads and writes, as cache accesses offer reduced access latencies than that of memory. Data is transfered between caches and main memory in units of *cache blocks*. Cache blocks typically range in size between 16 and 128 bytes. Depending on the data type, data in multiple memory locations can be associated with the same cache block. For example, four single-precision 32-bit floating-point values would comprise one 16-byte cache block. Recently accessed data is written into the cache, and subsequent reads and writes to the same data are performed via the cache, provided that the data

is still valid. Caches can substantially reduce the latency associated with memory operations, thereby potentially increasing the performance of a MPSoC considerably. For the following discussion, *direct-mapped* caches are assumed for simplicity, where a particular block in memory directly maps to only one location in the cache. As multiple blocks in memory map to one location in the cache, a *tag* is used to identify each block in the cache, which is a subset of the address bits of the block's address in memory.

With the addition of caches in a processor, there is going to exist some penalty when cache misses occur, because there is a delay when retrieving a cache block either from memory or from another processor's cache. There are a variety of causes that contribute to cache misses. For a single processor system, there are two types of misses that can occur. First, a miss occurs when the cache is initially empty and a memory location is referenced for the first time. Second, a read/write that maps to a cache location containing a valid cache block where the tag does not match the address of the request also results in a cache miss. This second type of cache miss can also trigger a write-back to memory if the existing valid cache block has been modified from a store by the host processor. In this case, the existing cache block must be written back to memory, prior to placing a new block in the same location in the cache. The frequency of write-backs depends on the size of the cache, as a larger cache reduces the number of memory locations that map to a particular location in the cache.

In a multiprocessor system with caches, there are additional sources of misses associated with data sharing, where reads and writes occur to the same memory location

by different processors. Whatever the sequence of reads and writes are as multiprocessor execution proceeds, *cache coherence* is enforced through requests issued and received by each processor's cache controller. Cache coherence ensures that a read to a particular memory location returns the most up-to-date data, regardless of where it exists. For example, a write by one processor to a particular memory location and a subsequent read to that same location by another processor will cause a cache miss. A miss will be incurred regardless of whether the processor performing the read previously had a copy of the associated cache block, because a write to any cache block will cause all copies in other caches to be invalidated. For this type of miss, the processor's cache with the modified data supplies the block, because memory does not have the most up-to-date data.

### 2.4.3 Vector Processing

Vector processing is implemented with a single instruction, multiple data (SIMD) processor architecture, where a single instruction operates on multiple data items simultaneously, achieving data-level parallelism [Pat05]. Many of the architectures described in Section 2.3 make use of SIMD processors, as they are well suited to the repetitive calculations required for turbo decoding. A single instruction performs an operation on multiple data elements, or a vector with a particular number of elements. Vector lengths anywhere from four to thirty-two words are common, with some architectures offering a variable vector length depending on the data type. In addition to operating on multiple data elements with a single instruction, vector processors typical execute fewer instructions, fewer overall operations, and fewer memory accesses, when compared with a traditional scalar microprocessor [Qui98].

Vector processors support conditional execution through the use of conditional instructions that use a vector mask register to specify which vector elements actually get modified [Qui98]. A vector instruction which performs a comparison, such as *set less than*, would modify the contents of the mask register, setting the $i$th position of the mask register for every $i$th vector element comparison that evaluated to true. A conditional vector instruction, such as a *conditional add*, would be carried out, but only the vector elements that have the $i$th position in the vector mask register set would ultimately get updated. Vector elements that have a zero bit in the mask register would be unmodified by the conditional instruction.

## 2.5 Chapter Summary

This chapter has presented background information pertinent to turbo decoding in a SDR environment. The emergence of the SDR design philosophy was explained, due to increasing use of multiple wireless standards in mobile devices, and the details of two general multiprocessor SDR architecture were presented. An introduction to FEC codes was presented, followed by a more detailed discussion of convolutional codes and turbo codes. An overview of turbo decoding using the MAP algorithm was provided, including design details required for hardware implementations. Next, the design details of some example turbo decoders were discussed, including both dedicated hardware turbo decoders and more flexible SDR-based turbo decoder architectures. Finally, computer architecture design considerations for supporting SDR pursuant to this research effort was discussed, including the use of MPSoCs, memory and caches in a multiprocessor system, and a brief discussion of vector processing.

# Chapter 3

# Double-Binary Turbo Codes

This chapter discusses topics pertinent to decoding WiMAX convolutional turbo codes (CTC) [IEE04]. Section 3.1 describes the WiMAX encoder structure and the operations performed to encode a sequence of data bits. A thorough description of the problem facing decoders and an understanding of Viterbi algorithm, which can be found in a tutorial paper by Forney [For73], would aid the reader in comprehending the nature of the turbo decoding algorithm. A clear and explanation of the structure of a WiMAX turbo decoder and its operation is important for understanding its efficient implementation in either hardware or software, and for identifying opportunities for enhancement. A number of papers attempt to describe the operation of a turbo decoder, all of which vary in the level of detail and clarity provided [Ber93, Ber05, Kim08, Lin06a, Rob95, MC07, Val01, Vit98, Zha06, Zha04]. The matter is further complicated by the use of double-binary turbo codes defined in the WiMAX standard, where the decoder operates on information associated with double-binary symbols as opposed to single bits. An attempt to provide as clear and thorough an explanation as possible of the turbo decoder structure and algorithm

details are broken down across Sections 3.2, 3.3, 3.4, and 3.5.

When a frame is encoded, the sequence of systematic and parity bits generated by the encoder can be represented as a unique path through a trellis, as explained in Section 3.1. The function of the decoder is to take into account the possible trellis paths representing the bits that were transmitted, by using probabilities associated with the received bits generated from the previous communication stage. Section 3.2 provides a top-level view of the double-binary turbo decoder structure, and the feedback nature of two component decoders connected together to cooperatively decode a frame. Each component decoder implements a simplified version of the maximum *a posteriori* (MAP) algorithm [Rob95], as explained in Section 3.3. An important operation called max* is used extensively throughout a particular MAP algorithm implementation, and different versions of max* have a direct impact on the BER performance of the decoder, as described in Section 3.4. Details associated with the trellis termination scheme utilized by a WiMAX encoder, and how it affects the operation of the decoder, are described in Section 3.5. Lastly, an efficient decoding technique called sliding window, common among hardware implementations, is described in Section 3.6.

## 3.1 The WiMAX Turbo Code

The turbo code defined by the WiMAX standard is a double-binary circular recursive systematic convolutional (CRSC) code [IEE04, Erg09]. The WiMAX convolutional turbo code (CTC) encoder and its constituent recursive encoder, with a constraint length of four, are shown in Figure 3.1. Essentially, the CTC encoder is constructed by the parallel concatenation of two identical recursive systematic convolutional (RSC)

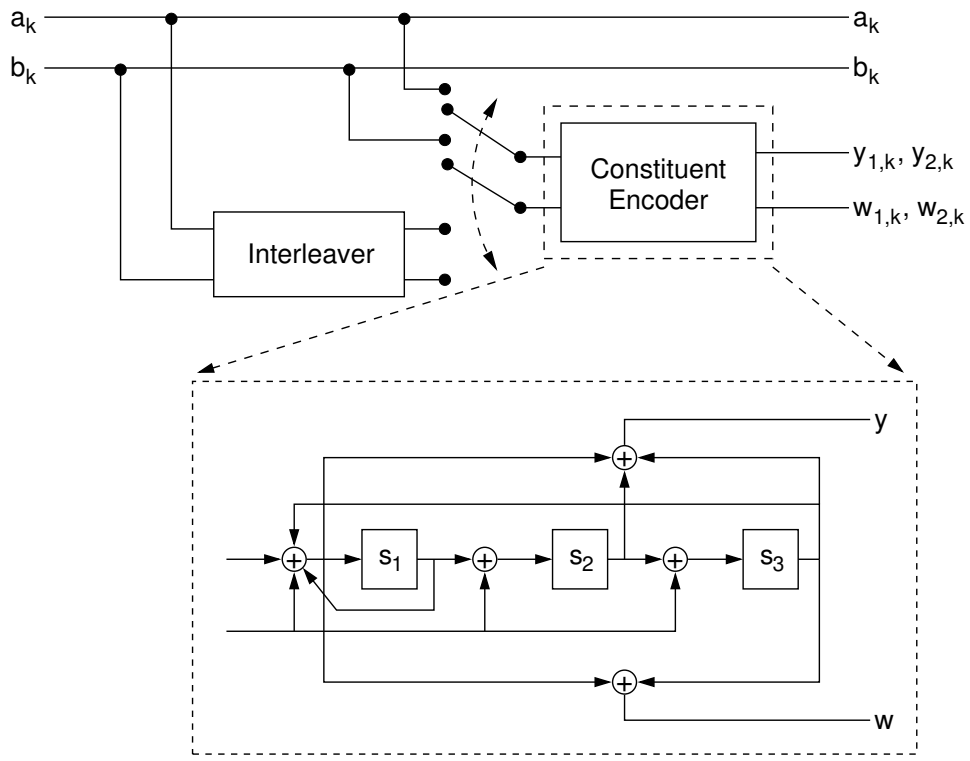


Figure 3.1: WiMAX turbo encoder.

encoders, where one RSC encoder operates on the input data sequence in natural order, while the other operates on the input data sequence in an interleaved order [Ber93]. Consecutive data bit pairs, $a_k$ and $b_k$, are fed to the encoder in blocks of $2N$ bits or $N$ couples, where $a_k$ and $b_k$ are the $k$th data bit pair at time $k$. In symbolic notation, the polynomials defining the connections are $1 + D + D^3$ for the feedback branch, $1 + D^2 + D^3$ for the $y_k$ parity bit, and $1 + D^3$ for the $w_k$ parity bit, where $D$ is a delay or storage element. The parity bit pair $\{y_{1,k},\ w_{1,k}\}$ is generated from the data sequence in natural order, whereas $\{y_{2,k},\ w_{2,k}\}$ is the parity bit pair generated from the interleaved sequence.

Two consecutive bits are used as simultaneous inputs, hence the double-binary encoder has four possible state transitions, as opposed to two possible state transitions

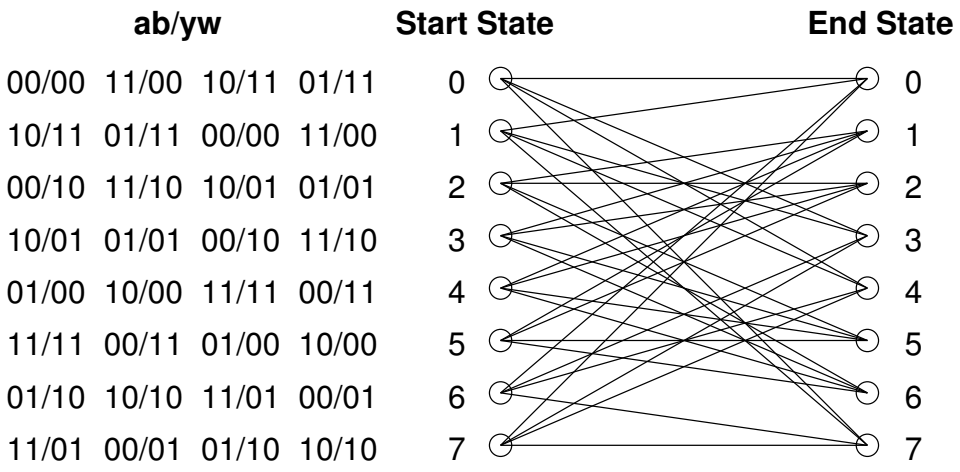


Figure 3.2: WiMAX turbo code trellis diagram.

for a single-binary turbo encoder. There are eight possible encoder states, where the current state is defined by $S = 4s_1 + 2s_2 + s_3$. The trellis diagram for the WiMAX turbo code, with all possible state transitions for a given input data bit pair ($ab$), along with its corresponding parity bit pair ($yw$), is illustrated in Figure 3.2. Because the WiMAX encoder standard utilizes a circular code, the start and end states of the encoded data sequence are forced to be the same.

Once a frame of size $N$ pairs of bits is encoded, the data output from the encoder consists of six sub-blocks:

- systematic bit pairs, $A = \{a_k\}_{k=0}^{N-1}$ and $B = \{b_k\}_{k=0}^{N-1}$,
- parity bit pairs of the systematic bits in natural order,
  $Y_1 = \{y_{1,k}\}_{k=0}^{N-1}$ and $W_1 = \{w_{1,k}\}_{k=0}^{N-1}$, and
- parity bit pairs of the systematic bits in interleaved order,
  $Y_2 = \{y_{2,k}\}_{k=0}^{N-1}$ and $W_2 = \{y_{2,k}\}_{k=0}^{N-1}$.

Because each sub-block contains the same number of bits, the natural code rate of

the encoder is 1/3. In other words, three bits are generated by the encoder for each data bit provided to it. Following encoding, each sub-block is interleaved separately, followed by puncturing the sub-blocks containing the parity bits using a specific puncturing pattern to achieve the target code rate. The puncturing of sub-blocks is simply the removal of some bits from a sub-block so that that fewer bits are actually transmitted. The interleaved and punctured sub-blocks are passed on to the symbol mapping stage, where a binary bit sequence is mapped to complex-valued symbols using the chosen constellation. Appendix A describes the different coding and modulation schemes defined for the WiMAX turbo code.

## 3.2 Decoding Double-Binary Turbo Codes

A typical turbo decoder consists of two soft-input soft-output (SISO) component decoders that are serially concatenated via an interleaver [Ber93, Val01]. For WiMAX, double-binary circular recursive systematic convolutional (CRSC) component decoders are used, as shown in Figure 3.3. The inputs to the turbo decoder provided from the symbol de-mapping stage are assumed to be in bit log-likelihood ratio (LLR) form, where the LLR for the $k$th bit $u_k$ is

$$L_k(u_k) = \ln\left(\frac{P(u_k = 1)}{P(u_k = 0)}\right). \tag{3.1}$$

The bit LLR values are a measure of the probability of a bit being 0 or 1, following transmission through a finite memory-less noisy channel, either an additive white Gaussian noise (AWGN) or Rayleigh fading channel. Following sub-block depuncturing and de-interleaving, the bit LLRs are converted to double-binary symbol

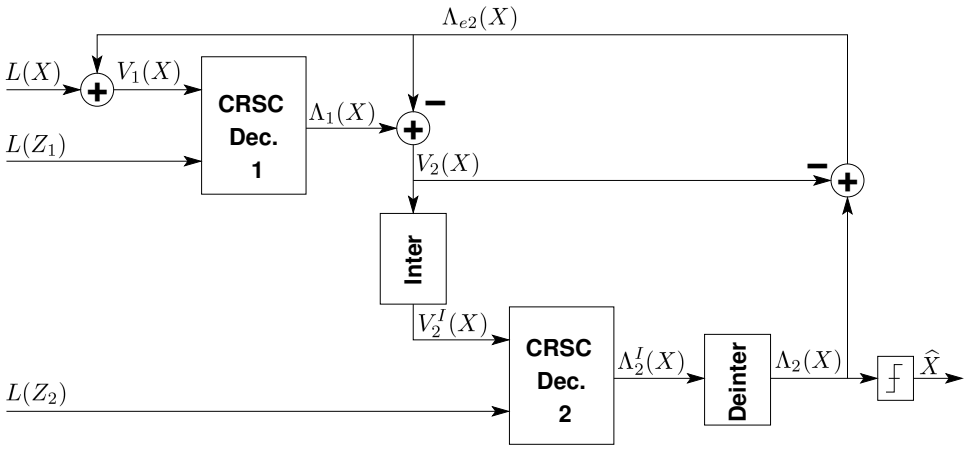


Figure 3.3: WiMAX turbo decoder.

LLRs, which are defined as

$$L_k(U_k) = \ln\left(\frac{P(U_k = z)}{P(U_k = 00)}\right) \tag{3.2}$$

for the $k$th symbol $U_k$, where z belongs to $\phi = \{01, 10, 11\}$. Therefore, for each symbol $U_k$, there are three associated channel LLR values. The reason for this conversion is because decoding is done using double-binary symbols rather than single bits.

There are three distinct sets of LLRs provided to the decoder:

- the systematic channel LLRs, $L(X)$,
- the parity channel LLRs used by CRSC decoder 1, $L(Z_1)$, and
- the parity channel LLRs used by CRSC decoder 2, $L(Z_2)$.

During decoding, each CRSC decoder receives *extrinsic* systematic LLRs, $\Lambda_e(X)$, obtained from the other CRSC decoder and computed as

$$\Lambda_e(X) = \Lambda(X) - V(X). \tag{3.3}$$

Thus, each CRSC decoder receives systematic symbol LLRs, $V(X)$, computed as

$$V(X) = L(X) + \Lambda_e(X), \tag{3.4}$$

along with parity channel LLRs, $L(Z)$. The output of each CRSC decoder is the *a posteriori* probability of each systematic symbol in LLR form, $\Lambda(X)$. The decoder operates for a specified number of iterations before making hard bit decisions, where each additional iteration improves the decoder's BER performance due to its feedback nature. Each decoder iteration consists of two half-iterations, during which only one SISO unit is performing CRSC decoding. Typically, turbo decoders perform between four and ten iterations, as each additional iteration offers a decreasing BER performance improvement.

The WiMAX CTC interleaver patterns are generated in a two-step process, and are dependent on the frame size, modulation, and code rate [IEE04]. These patterns are used when symbol LLRs are exchanged between decoders, namely to generate $V_2^I(X)$ by interleaving $V_2(X)$ for use by CRSC decoder 2, and to generate $\Lambda_2(X)$ by de-interleaving $\Lambda_2^I(X)$ for use by CRSC decoder 1.

## 3.3 The log-MAP Algorithm

Hardware implementations of turbo decoders typically use a simplified version of the maximum *a posteriori* (MAP) algorithm [Ber93]. The log-MAP algorithm performs MAP decoding in the logarithmic domain to avoid expensive multiplication and exponential operations and number representation problems [Rob95]. After each half-iteration, the *a posteriori* LLR output of each CRSC decoder for symbol $X_k$ is

expressed as

$$\Lambda_k(X_k) = \max^*_{(s_k,s_{k+1},z)} \{A_k(s_k) + \Gamma_{k+1}(s_k, s_{k+1}) + B_{k+1}(s_{k+1})\} - \max^*_{(s_k,s_{k+1},00)} \{A_k(s_k) + \Gamma_{k+1}(s_k, s_{k+1}) + B_{k+1}(s_{k+1})\}, \quad (3.5)$$

where $z$ belongs to $\phi$, and $s_k$ and $s_{k+1}$ are the start and end states, respectively, of a particular trellis branch associated with input symbol $z$ (refer to Figure 3.2). $A$, $B$, and $\Gamma$ are the forward, backward, and branch metrics, respectively. There are four branches associated with each symbol $z$ and symbol 00, thus the max* operator is performed over four values for each symbol. The details of the max* operator are discussed in Section 3.4.

The forward branch metric, $A_k(s_k)$, is a measure of the probability of the current state $s_k$ based on LLR values prior to time $k$. The backward branch metric, $B_{k+1}(s_{k+1})$, is a measure of the probability of the state $s_{k+1}$ given future LLR values after time $k$. The branch metric $\Gamma_{k+1}(s_k, s_{k+1})$ is a measure of the probability of the current state transition between $s_k$ and $s_{k+1}$. The $A$ and $B$ metrics are defined recursively as

$$A_k(s_k) = \max^*_{s_{k-1} \in s_A} \{A_{k-1}(s_{k-1}) + \Gamma(s_{k-1}, s_k)\} \quad (3.6)$$

and

$$B_k(s_k) = \max^*_{s_{k+1} \in s_B} \{B_{k+1}(s_{k+1}) + \Gamma(s_k, s_{k+1})\}, \quad (3.7)$$

where $s_A$ is the set of states at time $k-1$ connected to state $s_k$, and $s_B$ is the set of states at time $k+1$ connected to state $s_k$. For each trellis branch connecting states $s_k$ and $s_{k+1}$, $\Gamma$ is defined as:

$$\Gamma_k(s_k, s_{k+1}) = V_k(X_k) + L_k(Z_k) \quad (3.8)$$

where $V_k(X)$ contains the systematic symbol LLRs.

During every decoder iteration, each CRSC decoder calculates $\Gamma$, $A$, and $B$ values for the entire frame being decoded, followed by $\Lambda_k(X_k)$ calculations for each symbol $X_k$. An efficient technique to implement the log-MAP algorithm is described in Section 3.6. Once the decoder has operated for the specified number of iterations, the *a posteriori* symbol LLR values, $\Lambda_k(X_k)$, are converted to bit LLR values, which are then used to make hard bit decisions.

## 3.4 The max* Operator

The max* operator is employed extensively throughout the log-MAP algorithm, and its particular implementation affects both decoder complexity and BER performance. The max* operator is defined as

$$\max_i{}^*(\lambda_i) \triangleq \ln\left(\sum_i e^{\lambda_i}\right), \tag{3.9}$$

where $\lambda_i$ is a real number. This operation with multiple arguments can be decomposed into a recursive form using a max* operator with only two arguments [Erf94], such as

$$\max^*(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = \max^*(\max^*(\lambda_1, \lambda_2), \max^*(\lambda_3, \lambda_4)), \tag{3.10}$$

for example. Applying the Jacobian logarithm, a two-input max* operator can be expressed in the form [Erf94]

$$\begin{aligned}\max^*(\lambda_1, \lambda_2) &= \max(\lambda_1, \lambda_2) + \ln(1 + e^{-|\lambda_1 - \lambda_2|}) \\ &\triangleq \max(\lambda_1, \lambda_2) + f_c(|\lambda_1 - \lambda_2|).\end{aligned} \tag{3.11}$$

That is, the max* operator is equivalent to finding the maximum of the two input arguments, then adding a correction term, $f_c(|\lambda_1 - \lambda_2|)$. Computing the correction

term directly as $\ln(1 + e^{-|\lambda_1 - \lambda_2|})$ results in high computational complexity that is not suitable for implementation. As such, there has been much effort given to finding low-complexity versions of $f_c$, which offer varying degrees of accuracy.

The most accurate max* operator implementation is simply referred to as log-MAP, because the correction function computes the Jacobian logarithm exactly as it is defined in Equation 3.11. In software, the correction function can either be implemented using the `log` and `exp` function calls in C (or the equivalent in other languages), or by using a large look-up table [Rob95]. This is the most computationally expensive max* operator to implement in software, and other alternatives exist which have reduced complexity at the expense of slightly reduced accuracy.

Although it is the least accurate, the max-log-MAP max* algorithm is popular among hardware implementations because it is the least complex of the max* variants [Rob95]. This algorithm simply sets the correction function to zero, resulting in

$$\max^*\{\lambda_1, \lambda_2\} \approx \max(\lambda_1, \lambda_2). \tag{3.12}$$

A version slightly more complex than the max-log-MAP implementation is the constant-log-MAP algorithm, which uses a constant correction term [Gro98]. It approximates the Jacobian logarithm as

$$\max^*\{\lambda_1, \lambda_2\} \approx \max(\lambda_1, \lambda_2) + \begin{cases} 0, & \text{if } |\lambda_2 - \lambda_1| > T \\ a, & \text{if } |\lambda_2 - \lambda_1| \leq T \end{cases}, \tag{3.13}$$

where the constants $a = 0.5$ and $T = 1.5$ are optimal [Val01]. Essentially, the constant-log-MAP max* operator is equivalent to the log-MAP max* operator implemented using a two-element look-up table. The performance of the constant-log-MAP algorithm is between that of the max-log-MAP and log-MAP algorithms.

A further enhancement is the more complex linear-log-MAP algorithm, which offers one of the best trade-offs in terms of complexity and performance among the different max* variants [Che00, Val01]. It achieves an approximation very close to that of the log-MAP max* implementation by using a linear correction function,

$$\max^*\{\lambda_1, \lambda_2\} \approx \max(\lambda_1, \lambda_2) + \begin{cases} 0, & \text{if } |\lambda_2 - \lambda_1| > T \\ a(|\lambda_2 - \lambda_1| - T), & \text{if } |\lambda_2 - \lambda_1| \leq T \end{cases} . \quad (3.14)$$

In other research, parameters $a = -0.24904$ and $T = 2.5068$ were found to minimize the total squared error between the exact correction function and its linear approximation, when using floating-point operations to implement the decoder [Val01].

## 3.5 Tailbiting Turbo Codes

The WiMAX turbo code is a circular code, where the encoder uses a technique called tailbiting to force the encoder start state to be equal to the end state [IEE04]. Encoders that use tailbiting when encoding a frame are more bandwidth-efficient than using flush bits, which force the encoder start and end states to 0 [And98]. The encoded frame can be viewed as a path around a circular trellis, however the start and end states are unknown to the decoder. Because the $A_k$ and $B_k$ metrics are defined recursively in Equations 3.6 and 3.7, $A_0$ and $B_{N-1}$ must be initialized for all states at the start of each half-iteration, for a frame size of $N$ pairs of bits. Two methods are generally preferred, with the first method performing *dummy* metric calculations to obtain reliable initial border metrics [Dou00]. A second method uses the metrics computed during the previous iteration, which has been shown to offer comparable BER performance to the first method with lower computational complexity [Zha06].

Thus, $A_0$ and $B_N$ are initialized using:

$$A_0(s) = \begin{cases} 0 & \text{for the first iteration} \\ A'_N(s) & \text{otherwise} \end{cases} \tag{3.15}$$

$$B_N(s) = \begin{cases} 0 & \text{for the first iteration} \\ B'_0(s) & \text{otherwise} \end{cases} \tag{3.16}$$

In the above formulation, $s$ denotes the encoder state and $A'_N(s)$ and $B'_0(s)$ are the metrics from the previous iteration. Initializing $A_0(s)$ and $B_N(s)$ to 0 for the first iteration reflects that the circular state is unknown, and therefore all states have the same probability.

## 3.6 The Sliding Window Decoding Technique

The majority of turbo decoder implementations use the sliding window technique to operate on a portion of a frame at a time so as to minimize memory requirements [Vit98]. A large frame is divided into a number of sub-blocks, and the MAP algorithm is applied to each sub-block independently. Memory requirements are reduced because branch metrics only have to be stored for the sub-block currently being decoded, rather than for the entire frame. Decoding each sub-block typically proceeds by calculating either the $A$ or the $B$ metric for the entire sub-block, and then calculating the other metric ($B$ or $A$) and the LLR values $\Lambda$ for each trellis index. The second metric calculated ($B$ or $A$) does not have to be stored for the entire sub-block; only the metrics for two trellis indexes need to be stored. Figure 3.4 illustrates how decoding proceeds using the sliding window technique with the original frame divided into four

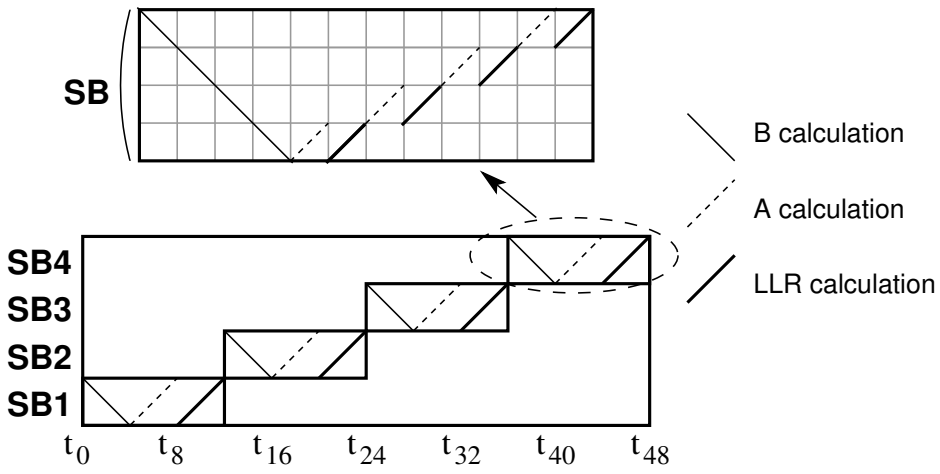


Figure 3.4: The sliding window technique using four sub-blocks, with the details of a single sub-block shown.

sub-blocks. The details of a single sub-block are shown separately to highlight the difference in calculations for the first and second metrics. For each individual sub-block shown, $B$ metrics are calculated first for the entire sub-block, followed by $A$ and LLR calculations. In this case, $A$ metrics only need to be stored for the current and previous trellis index, as the $A$ metrics from earlier indexes are not needed. Because the sub-blocks are decoded serially, the horizontal time scale in Figure 3.4 indicates that a total of 48 time units are required.

Similar to the tailbiting technique, one of the metrics at the border of each sub-block must be initialized when utilizing a sliding window. For the frame shown in Figure 3.4, the $B$ metrics must be initialized for each sub-block, because they are defined recursively in the reverse direction, while the $A$ metrics can be initialized from the previous sub-block. Initialization can be accomplished by either performing dummy calculations, which involves performing calculations on the frame outside the current sub-block [Zha04], or by using metrics from the previous iteration. Initialization using dummy calculations increases decoder latency, while using metrics from the

previous iteration requires a border memory to store the border values, so a trade-off is made when using either technique. A double-binary turbo decoder implementation has shown that the two techniques provide almost identical BER performance [Kim08]. Because one of the goals of this work is to reduce decoder latency and thereby increase decoder throughput, using metrics from the previous iteration is the selected technique for the decoder in this thesis.

## 3.7 Chapter Summary

This chapter has provided an overview of topics for double-binary turbo codes as they apply to the WiMAX standard. A description of the encoding process was provided, including the details of the WiMAX turbo code and the structure and operation of the encoder. Next, decoding turbo codes using two CRSC component decoders was discussed, including the generation of LLRs for both input and output to each component decoder. Each component decoder implements the log-MAP decoding algorithm, and its details including the required calculation of various metrics were discussed. Different log-MAP implementations, which make us of different max* operators of various complexities and accuracies, were described. The details of the WiMAX tailbiting termination scheme, which implements a circular code, was discussed. Lastly, a popular turbo decoder implementation technique called sliding window was presented.

# Chapter 4

# Decoder Adaptation and Optimizations

This chapter discusses the adaptation, implementation and optimization of the original WiMAX software turbo decoder, and it provides results on execution time and bit-error rate (BER) performance. The initial implementation of a stand-alone decoder in C, which was adapted from an existing open-source decoder written in MATLAB with C extensions, is discussed in Section 4.1. Section 4.2 then discusses various optimizations that were implemented to reduce the execution time of the decoder when decoding a frame. A version of the decoder that operates exclusively using integer data and arithmetic is described in Section 4.3. The decoder's simulated BER performance is discussed in Section 4.4. Finally, the decoder's execution performance and the speedup resulting from each optimization implemented are summarized in Section 4.5.

## 4.1 Decoder Adaptation

The WiMAX double-binary turbo decoder reflecting the design in Figure 3.3 was implemented in C, where each circular recursive systematic convolutional (CRSC) component decoder utilizes the log-MAP algorithm described in Section 3.3. The initial turbo decoder software was adapted from the open-source Coded Modulation Library (CML) [Val08], which provided sequential encoder and decoder implementations in MATLAB with C extensions, suitable for BER simulation. For the original CML WiMAX decoder, the operations performed by each CRSC decoder were implemented as C extensions, while the rest of the decoder functionality was implemented in MATLAB. The CML provided all of the variants of the max* operator discussed in Section 3.4. In particular, two variants of the log-MAP algorithm were provided, one which calculates the correction function exactly as defined in Equation 3.9 using `log` and `exp` C function calls [Mul00], and the other which uses an eight-element non-uniform look-up table with linear interpolation. To support the flexibility of choosing the max* operator at run-time in the original CML implementation, the different max* functions are called via an array of function pointers, where each element is a pointer to a particular max* function variant. An argument supplied at run-time specifies the array index of the desired max* function to use during decoding.

The initial effort for this thesis involved implementing a stand-alone decoder in C by adapting the decoder functionality originally written in MATLAB. Referring to Figure 3.3, the functionality needed outside each CRSC decoder was translated to C from MATLAB for this work, including interleaver/de-interleaver operations,

the generation of $V_1(X)$, $V_2(X)$, and $\Lambda_{e2}(X)$, and the ability to make hard bit decisions. The CML provides the necessary functionality to generate the patterns required for interleaving/de-interleaving and puncturing/de-puncturing, as defined in the WiMAX standard [IEE04]. For a software-defined radio (SDR) decoder implementation, these patterns only need to be generated once, prior to decoding a frame. It was decided that an efficient SDR implementation to generate the WiMAX turbo code interleaver/de-interleaver patterns and perform sub-block de-interleaving and de-puncturing would be beyond the focus of this work, but could be given future consideration. Efforts would primarily be focused on the actual decoding algorithm itself and its efficient implementation.

Given that CML provides an accurate wireless simulation environment, it was used to provide test data for the stand-alone C decoder implemented. The data provided includes the original binary data, the interleaver/de-interleaver pattern, and symbol log-likelihood ratio (LLR) values, in the form of a file to be read by the stand-alone C decoder. In an SDR environment, the demodulation stage of a wireless communication pipeline would be responsible for writing symbol LLR values to memory, to be read by the forward error-correction stage. The symbol LLRs are in double-binary form, and the corresponding sub-blocks have already been de-interleaved and de-punctured, as discussed in Section 3.1.

### 4.1.1 Halting Condition

The original CML decoder software used an ideal halting condition, where the decoder stops when either all frame errors are corrected, or a specified maximum number of iterations had been reached. The maximum number of iterations was set to ten. This

approach reduces execution time when performing BER simulations, where a large number of frames are processed, as the number of iterations performed decoding each frame is reduced. Ideal halting requires *a priori* knowledge of the original data, to determine if errors exist in the decoded binary data, hence it is not suitable for a real-world decoder implementation. The decoder software was modified so that a fixed number of iterations were used, as is common among the majority of turbo decoder implementations. Typically, turbo decoders perform between four and ten iterations, as each additional iteration offers a decreasing BER performance improvement. The fixed number of iterations used was set to eight for the initial C decoder implementation.

## 4.2 Reduction of Execution Overhead

After the initial adaptation of the stand-alone decoder in C, various algorithmic optimizations were applied in order to reduce the execution time for decoding a frame. This section describes the optimizations that were performed; the resulting performance improvement of each optimization is assessed in Section 4.5.

### 4.2.1 Elimination of Dynamic Memory Allocation

In the original implementation, various global arrays were being allocated dynamically during each CRSC decoder half-iteration using calls to the C library function `malloc(...)` [Mul00], including arrays for $A$, $B$, and $\Gamma$ metrics, as well as temporary storage. The size of the arrays allocated were dependent on the size of the frame currently being decoded, as well as the number of states and state transitions. For

a particular coding scheme, such as the WiMAX turbo code, the number of states and state transitions from a particular state remain constant. To eliminate execution time overhead required for dynamic memory allocation, arrays were simply globally declared to accommodate the largest WiMAX turbo code frame size of 60 bytes, as indicated in Appendix A. Global variables are allocated on program startup, and continue to exist for the duration of the main program. Thus, no overhead is incurred by using global variables, entirely eliminating any overhead associated with memory allocation.

### 4.2.2 WiMAX Turbo Code Trellis Generation

The trellis information associated with the WiMAX turbo code is generated by making a function call in the original implementation, with the encoder generator polynomials as inputs. With reference to Figure 3.2, the trellis information associated with each trellis branch consists of a start state, end state, systematic bit pair, and parity bit pair. In the original code, the trellis information is generated every decoder half-iteration, however only the trellis end state and parity bit pair are stored in arrays. In contrast, the trellis start state and systematic bit pair are generated on demand, in particular when performing $A$, $B$, $\Gamma$, and bit LLR calculations. Because there are 32 branches associated with the WiMAX turbo code trellis, a loop is used to perform calculations associated with each branch. There are eight trellis states and four branches associated with each state, thus the trellis start state is simply calculated using integer division as $trellis\_start\_state = i/4$, where $0 \leq i < 32$ is the loop counter. Similarly, the systematic bit pair or trellis input is calculated using modular division as $trellis\_input = i \mod 4$.

The trellis information for a particular code does not change from iteration to iteration, as it is characteristic of the code constraint length and encoder generator polynomials. Therefore, the generation of the WiMAX turbo code information was changed so that it was only generated on program startup, rather than every half-iteration. The function that generates the trellis information was modified so that the trellis start state and systematic bit pair information was stored in arrays for use later when needed, rather than performing the calculations every time this information was required.

### 4.2.3 Decoder Half-Iteration Complexity Reduction

The original decoder code ran for some number of cycles each half-iteration when performing $A$, $B$, and $\Gamma$ calculations, as given in Equations 3.6, 3.7, and 3.8, respectively. For the first cycle, border $A$ and $B$ metrics were initialized to zero, while for the second and subsequent cycles, the border metrics were set to that of the previous cycle, as done similarly in Equation 3.15 due to the tailbiting property. Every cycle, a comparison was made between the new and previous metrics, and if the difference between any pair exceeded some threshold, another cycle was executed. As a result, at least two cycles were executed every half-iteration, with the total number of cycles executed varying each half-iteration, up to some maximum number. $A$ and $B$ calculations performed during the first cycle could be viewed as *dummy* metric calculations, a technique mentioned in Section 3.5 to obtain reliable initial metrics. However, *dummy* metric calculations are typically performed over only a few trellis indexes, and not the entire frame [Dou00]. In addition, the original decoder software did not initialize border $A$ and $B$ metrics from the previous iteration, thus additional time

was being wasted (at least one cycle) determining reliable border metrics by setting them to zero at the start of each half-iteration. Additional time was also being wasted by performing $\Gamma$ calculations twice per cycle; once during the calculation of $A$ metrics and again for the calculation of $B$ metrics. In addition to *a posteriori* systematic LLRs, $\Lambda(X)$, parity LLR values were being calculated by each CRSC decoder. The latter values are never used.

The check between cycles was likely done in the original decoder to ensure that the correct start and end states were correctly identified. If they were incorrectly identified, the new and previous metrics would differ by some margin. However, this is why multiple iterations are used in the first place; multiple iterations increase the reliability of the metrics calculated, which ultimately improve the BER performance of the decoder. Most, if not all, turbo decoder implementations calculate $A$, $B$, and $\Gamma$ metrics only once per decoder half-iteration. Modifications were made to the decoder to avoid redundant calculations, along with eliminating the check between new and previous metrics, so that effectively only one cycle was performed each half-iteration. Also, the initialization of $A$ and $B$ border metrics was modified to use the metrics of the previous iteration, properly implementing the tailbiting initialization scheme given in Equation 3.15. The calculation of $\Gamma$ values was modified so that they were only being done once per half-iteration, rather than twice. The calculation of *a posteriori* parity LLRs was eliminated as they are never used.

### 4.2.4 Sliding Window and Other Optimizations

The decoder design up to this point calculates $\Gamma$, $A$, and $B$ metrics individually, followed by *a posteriori* symbol LLRs. This method implements the log-MAP turbo

decoding algorithm properly, as described in Section 3.3, however unnecessary memory is being used to store both $A$ and $B$ metrics for the entire frame. The sliding window (SW) decoding scheme, explained in Section 3.6, requires only $A$ or $B$ metrics to be stored for a portion of the frame or current sub-block, while the other branch metric only needs to be stored for only two frame indexes. To avoid any BER degradation resulting from splitting the frame into sub-blocks, the decoder was modified so that $A$ metrics are calculated for the entire frame, followed by $B$ metrics and output LLRs. This is equivalent to the sliding window scheme using only one sub-block the size of the entire frame. The need to initialize one of the branch metrics, as discussed in Section 3.6, is avoided by using this method. Thus, instead of storing $B$ metrics for the entire frame, as done in the original decoder implementation, they are only stored for two frame indexes.

The original code calculated the bit LLRs every iteration, whereas they only need to be calculated when making hard bit decisions. The adapted decoder software operates for a fixed number of iterations before making bit decisions, hence it was modified to only calculate bit LLRs on the second half of the last iteration. Border $A$ and $B$ values were being stored at the end of every iteration to be used as the initial metric values for the following iteration, due to the tailbiting property described in Section 3.5. The border values stored on the last iteration are not used, hence this unnecessary execution overhead was eliminated. These optimizations were combined with various other minor optimizations, such as enhancing the efficiency of $\Gamma$ calculations.

### 4.2.5 Enhanced Decoder Software Structure

With all of the optimizations discussed in this section to this point applied to the adapted stand-alone C decoder implementation, the structure of the resulting software is illustrated in Figure 4.1. The figure shows the hierarchy of function calls, and a brief explanation of the work performed by each function.

### 4.2.6 Reduction of max* Function Overhead

The decoder software profile statistics, to be presented in Section 4.5.5, suggest that optimizing the max* function would stand to increase performance significantly, as it contributes the most to execution time. As mentioned in Section 4.1, the decoder software up to this point calls the desired max* variant from an array of function pointers, using an argument supplied at run-time. If a particular max* operator is chosen prior to run-time, its corresponding function can be called directly rather than through a function pointer. Each time the max* operator is used, there is run-time overhead associated with the array look-up, as well as the pointer indirection to resolve the actual location of the function in memory. Compile-time selection of the max* operator allows the corresponding function to be called directly during execution, eliminating this extra overhead.

To further reduce the overhead associated with the max* function, the operations of the selected max* operator can be inlined in the code, thus completely eliminating the overhead associated with a function call.

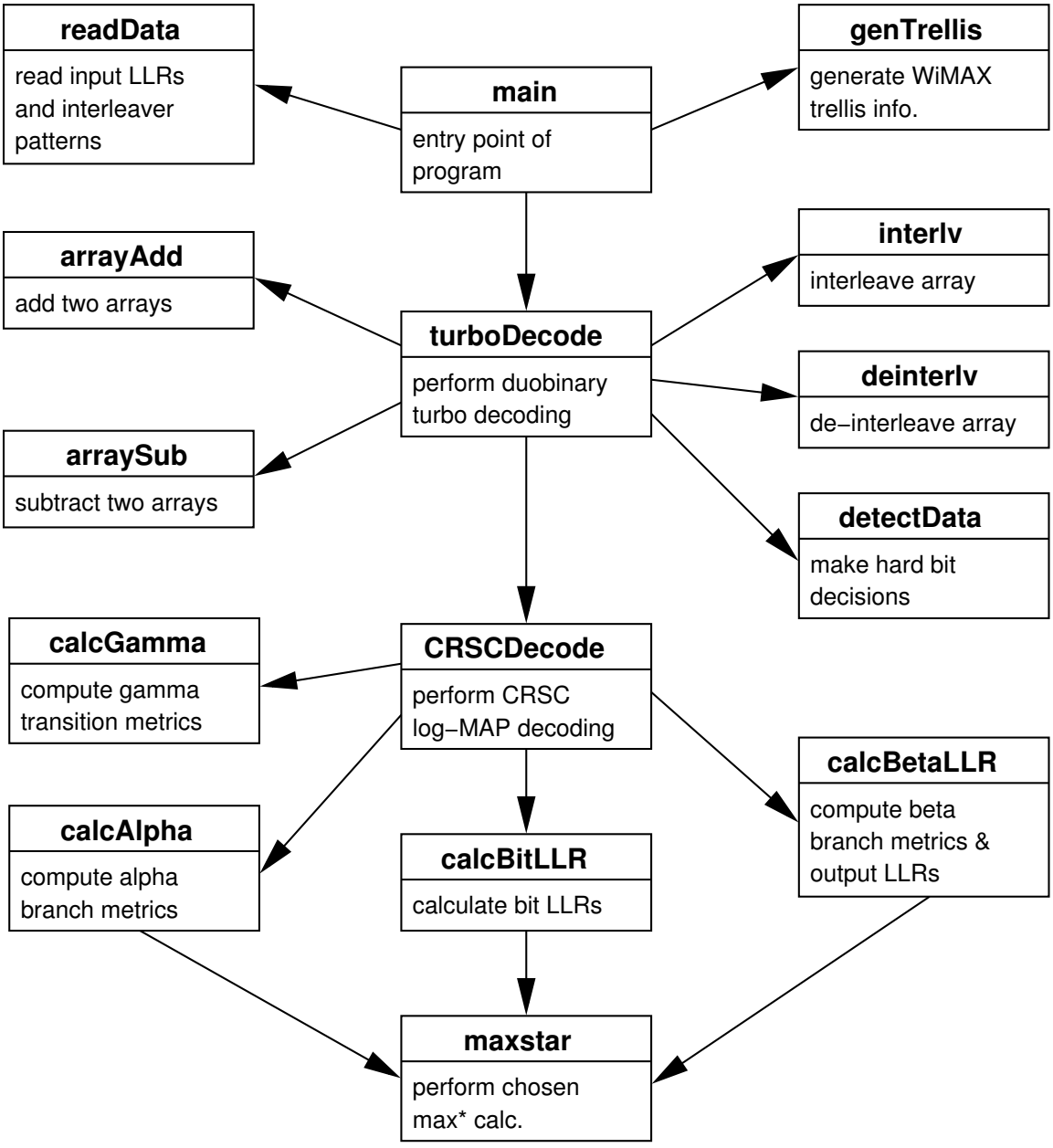


Figure 4.1: Optimized turbo decoder software structure.

## 4.3 Integer Implementation

In hardware, integer arithmetic is typically faster than the more complicated floating-point arithmetic. Some low-cost architectures only offer integer arithmetic capabilities to reduce chip area. Fixed-point number representation, which is common among dedicated hardware turbo decoders, uses a fixed number of bits before and after the radix point to represent real numbers. Integer number representation is similar to fixed-point representation, except that values are scaled by some factor, for example $1.234 = 1234/1000$; here, the scaling factor is 1000. Thus, fixed-point and integer representations using the same number of bits can each represent the same number of unique numbers. Integer and fixed-point representations cannot represent nearly as wide a range of values as floating-point representation. However, if the range of values needed to be represented is known, this does not pose a problem.

A turbo decoder implemented to operate using only integer data could offer further improvement in execution performance, while supporting a broader range of architectures. For the turbo decoder software up to this point, the input LLRs, calculated metrics, and output LLRs are manipulated and stored using floating-point values. The range of values of the input LLRs vary depending on the channel SNR, modulation scheme, and frame size, but the values are real numbers typically in the range of $\{-50 \ldots 50\}$. The input LLRs can be converted to integers by scaling them by some factor $\alpha$, and truncating the fractional portion. Thus, the scaled integer LLRs would be in the range of $\{-50\alpha \ldots 50\alpha\}$. Although elimination of the fractional portion introduces some error, decoding essentially proceeds exactly the same using integer values instead of float-point values. In Section 4.5, the execution time performance is evaluated for the integer turbo decoder with $\alpha = 100$, for simplicity, using inlined

max-log-MAP and linear-log-MAP max* variants with all optimizations discussed up to this point. The integer decoder's BER performance is discussed in Section 4.4.1.

When scaled integer LLRs are used, the max-log-MAP max* operator would function exactly the same as defined in Equation 3.12, by simply taking the maximum of the two arguments. Substituting the input arguments in Equation 3.14 with arguments scaled by some factor $\alpha$, the linear-log-MAP max* operator becomes

$$\max^*\{\alpha\lambda_1, \alpha\lambda_2\} \approx \alpha \max(\lambda_1, \lambda_2) + \begin{cases} 0, & \text{if } \alpha|\lambda_2 - \lambda_1| > T \\ a(\alpha|\lambda_2 - \lambda_1| - T), & \text{if } \alpha|\lambda_2 - \lambda_1| \leq T \end{cases}. \quad (4.1)$$

For the linear approximation to hold, the constant $T$ must be scaled by $\alpha$ as well. As mentioned previously in Section 3.4, the optimal values for $a$ and $T$ were $a = -0.24904$ and $T = 2.5068$. Because multiplication by $a$ is closely approximated by multiplying by $-1/4$, it can also be performed by dividing by $-4$. In hardware, dividing by $-4$ can be implemented by performing a right-shift by two bit positions for binary representation, and negating the result. Thus, any floating point operations can be avoided, and Equation 4.1 becomes

$$\max^*\{\lambda_1, \lambda_2\} \approx \max(\lambda_1, \lambda_2) + \begin{cases} 0, & \text{if } |\lambda_2 - \lambda_1| > \alpha T \\ (|\lambda_2 - \lambda_1| - \alpha T)/a, & \text{if } |\lambda_2 - \lambda_1| \leq \alpha T \end{cases}, \quad (4.2)$$

assuming $\lambda_1$ and $\lambda_2$ have already been scaled by $\alpha$ and $a = -4$. For $\alpha = 100$, $100T$ is approximated by $100T \approx 251$.

## 4.4 Simulated Decoder BER Performance

For verification purposes, BER performance simulations were conducted using the Coded Modulation Library (CML) simulation environment. This section discusses the

BER performance of the original, optimized, and integer decoder implementations, as well as the effect that the choice of different max* operators has on decoder performance. Simulations were conducted for an additive white Gaussian noise (AWGN) channel for rate-1/2 frame sizes using quadrature phase-shift keying (QPSK) modulation. The results presented below are for decoding 36-byte frames, annotated as QPSK (36,72), however the results are similar for other frame sizes, modulations, and code rates. Smaller frame sizes exhibit worse BER performance while larger frame sizes exhibit better BER performance, for a particular SNR. Using higher code rates and higher order modulations result in worse BER performance than rate-1/2 QPSK frames, but show BER trends similar to QPSK (36,72) frames. Enough frames were simulated to generate at least 100 frame errors at a particular SNR for each case considered.

### 4.4.1 Decoder BER Performance Comparison

The BER of the decoder after each optimization was applied in Section 4.2 was compared with that of the original CML decoder to ensure that no BER degradation had occurred. Also, the BER of the integer decoder, described in Section 4.3, was simulated to determine the effect that rounding the input LLRs has on performance. BER simulation results are presented in Figure 4.2 for the original, fully optimized, and integer decoders, for both the max-log-MAP and linear-log-MAP variants, when decoding QPSK (36,72) frames. The BER performance of the three decoders was found to be almost identical, especially at low $Eb/No$ SNR values, for both the max-log-MAP and linear-log-MAP max* variants. This demonstrates that the optimizations performed to the decoder have not affected its BER performance. Also, the rounding

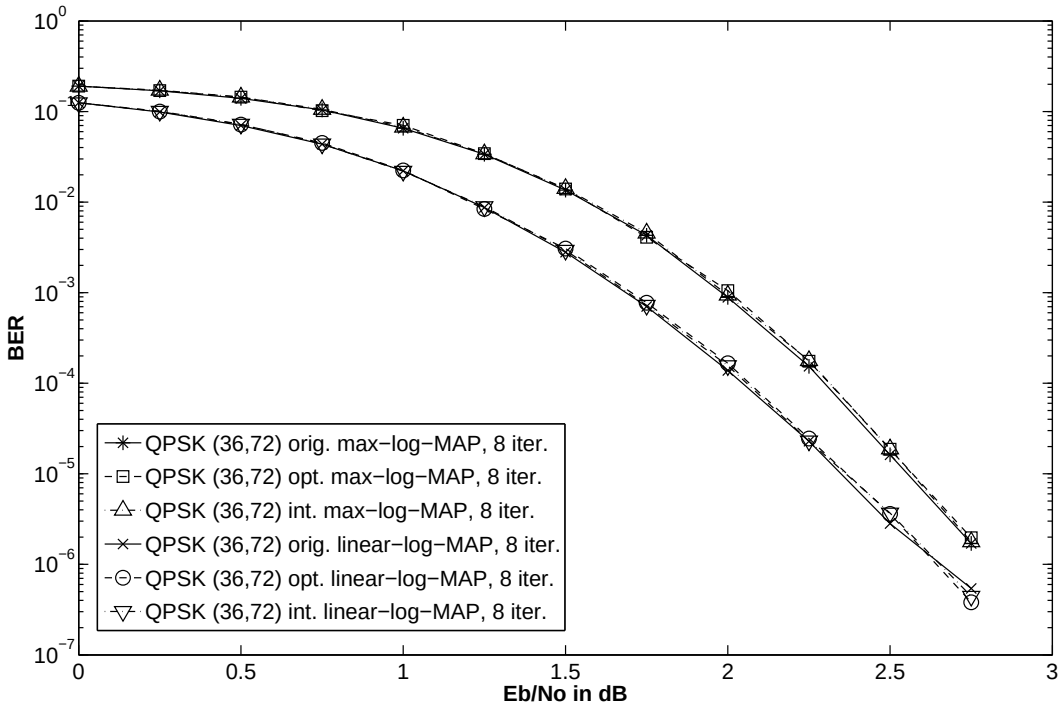


Figure 4.2: BER performance of the original, optimized, and integer decoders.

of input LLRs performed by the integer decoder, as well as the modified linear-log-MAP max* operation used, exhibit little to no impact on its BER performance.

To illustrate that the results obtained for decoding QPSK (36,72) frames are consistent with the results obtained for other frame sizes, modulations, and code rates, an additional graph is shown in Figure 4.3 using the optimized decoder. Here, in addition to the QPSK (36,72) BER results, results are shown for QPSK (18,36) and QPSK (60,120) frames, all using the max-log-MAP algorithm. In comparison to QPSK (36,72) frames, the BER for QPSK (18,36) frames is slightly worse, while the BER for QPSK (60,120) frames is slightly better, as expected. Results for decoding 16-QAM (36,72) frames and rate-3/4 QPSK (36,48) frames are shown for both the max-log-MAP and linear-log-MAP decoders using 8 iterations. Decoding QPSK

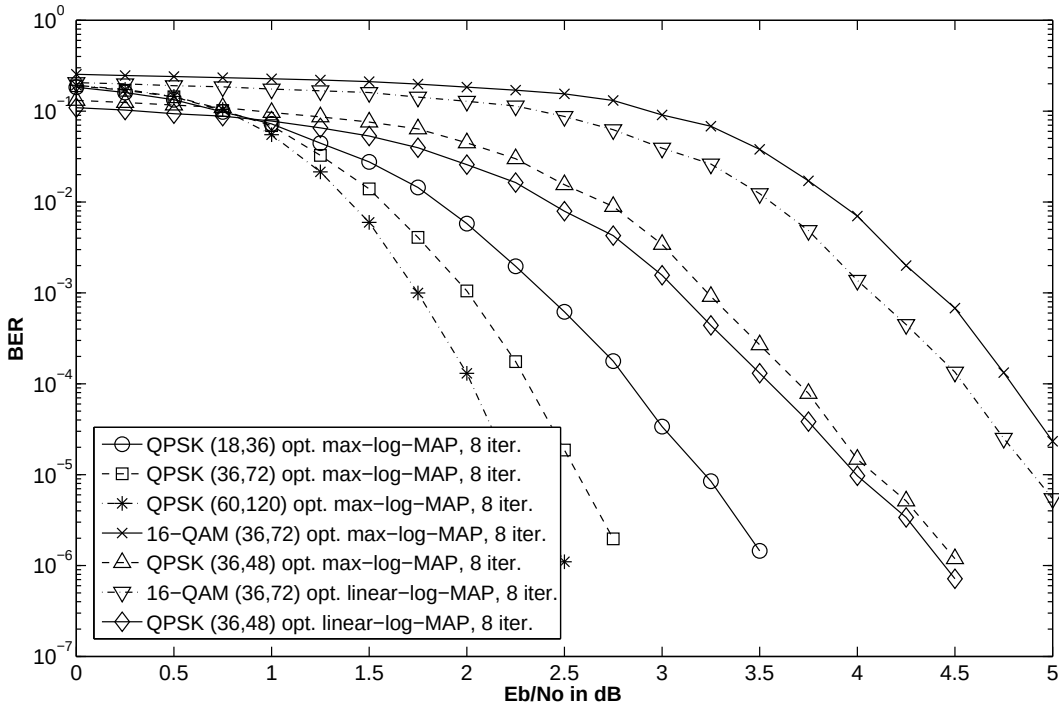


Figure 4.3: BER performance comparison of rate-1/2 QPSK frames, and rate-3/4 QPSK and 16-QAM frames.

(36,48) frames exhibited significantly worse BER performance than the other rate-1/2 QPSK frames, while decoding 16-QAM (36,72) frames exhibited even worse BER performance still. These results are expected, as encoding rates higher than 1/2 and higher order modulations are used only when channel conditions are good, i.e. when the SNR is high. The linear-log-MAP decoder still had better BER performance than the max-log-MAP decoder, with a BER trend similar to the QPSK (36,72) simulations discussed above.

### 4.4.2 Effect of max* Operator on BER Performance

To characterize the effect that the different max* operators have on the turbo decoder's BER performance, MATLAB simulations using CML were run using decoder

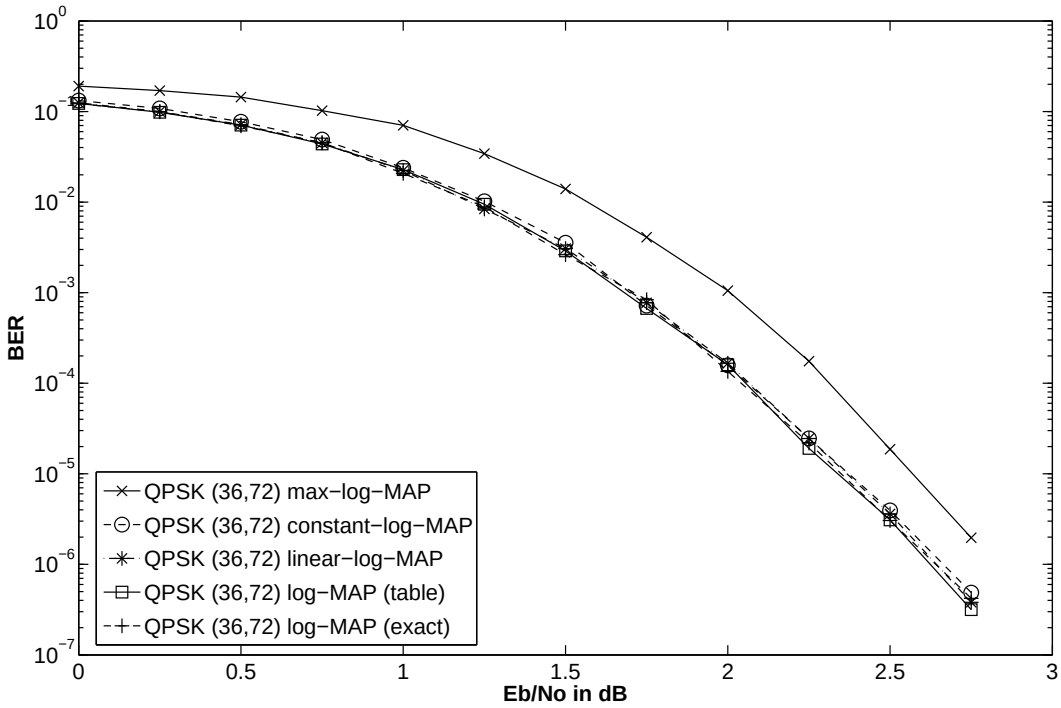


Figure 4.4: BER comparison of max* operators for decoding a QPSK (36,72) frame using eight iterations.

implementations utilizing each of the max* variants presented in Section 3.4 with eight iterations. As mentioned earlier, two variants of the log-MAP algorithm were provided, one which calculates the correction function exactly as defined in Equation 3.9, and the other which uses an eight-element non-uniform look-up table with linear interpolation. BER simulation results are presented in Figure 4.4 for decoding a QPSK (36,72) frame. As expected, the max-log-MAP decoder implementation consistently performed the worst because the correction function is set to zero for reduced complexity. The other, more complex implementations performed better, being clustered within 0.1 dB of each other for a given BER. The constant-log-MAP did perform slightly worse than the linear-log-MAP and both log-MAP implementations.

A more complex max* variant can achieve the same BER performance as the least

complex max-log-MAP max* variant using fewer decoder iterations, which could potentially offer better execution-time performance. The linear-log-MAP decoder was chosen for comparison with the max-log-MAP decoder because it offers comparable BER performance with that of the more accurate log-MAP implementations, while being less complex. Decoder BER simulations comparing the max-log-MAP algorithm using 8 iterations and the linear-log-MAP algorithm using a varying number of iterations were performed, with the results displayed in Figure 4.5. The simulations revealed that decoding using the max-log-MAP algorithm with 8 iterations performs worse than the linear-log-MAP algorithm using both 4 and 6 iterations. Decoding using the max-log-MAP algorithm with 8 iterations and the linear-log-MAP algorithm with 4 iterations perform comparably for large values of $E_b/N_0$. It can be expected that integer implementations of the two decoders would also perform comparably in terms in BER. The execution performance of the max-log-MAP decoder using 8 iterations and the linear-log-MAP decoder using 4 iterations is compared in Section 4.5.6.

## 4.5 Simulated Execution Performance

This section discusses the decoder's execution time performance of the original adapted decoder and the decoders resulting from implementing each of the optimizations presented in Section 4.2. The hardware used for simulation, and the methodology used to obtain execution-time measurements are described in Section 4.5.1. Section 4.5.2 presents the adapted decoder's execution time and speedup results for the various optimizations implemented. Execution-time profile results are discussed in Section 4.5.5 for the decoder with all optimizations described up to Section 4.2.4. Lastly, the

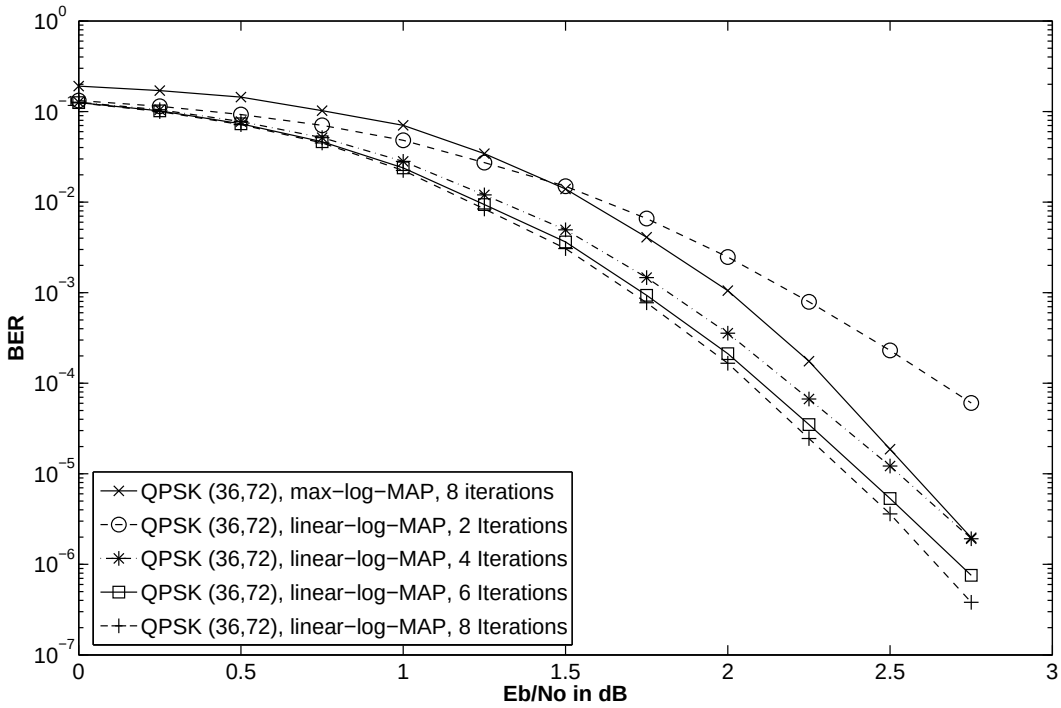


Figure 4.5: BER performance comparison of the max-log-MAP and linear-log-MAP decoders for decoding a QPSK (36,72) frame.

execution performance of the max-log-MAP decoder using eight iterations and the linear-log-MAP decoder using four iterations are compared in Section 4.5.6.

### 4.5.1 Simulation Environment

A personal computer was used to conduct the execution time performance simulations. The system consisted of an AMD Athlon 64 X2 4400+ dual-core processor, with 64 KB of level 1 and 1 MB of level 2 per core, and 2 GB of memory. The system was configured with the Ubuntu distribution of the Linux operating system, version 7.04. The adapted and optimized decoder software in C was compiled using the `gcc` compiler, version 4.1.3 with optimization level `-O3`. To obtain execution times, the Linux UNIX `/usr/bin/time` command was used.

To make execution times lengthy enough so that small variances have an insignificant impact on results, a sufficient number of frames was used. Simulation data was generated by the original CML software, utilizing QPSK modulation and an AWGN channel. The simulation data used covered 41 SNR points ranging between 0.0 and 10.0 dB, in increments of 0.25 dB, with 10 frames per SNR point. Input data covering a wide range of SNR values was used so that the input LLRs varied as much as possible, as the input LLRs are larger for higher SNR. Every frame was decoded ten times to further lengthen the decoding time and to mitigate the overhead of the compiled C code reading the frame data from a file. Frame sizes of 6, 18, 36, and 60 bytes were used to evaluate the performance improvement resulting from each optimization, to characterize the effect that frame size has on decoding time. The times reported are the average of three runs for each optimization and max* implementation.

### 4.5.2 Execution Performance Results

This section discusses the execution-time performance, for the original decoder and the decoder with each of the optimizations described in Section 4.2. Results for the max-log-MAP, constant-log-MAP, linear-log-MAP, and two versions of the log-MAP decoder implementations are presented for comparison, with each using eight iterations. As the linear-log-MAP decoder offers almost identical BER performance of the two log-MAP decoders, it is the preferred decoder implementation of the three; however, results for the two log-MAP decoders are presented for comparison. The optimizations were implemented incrementally, and the resulting decoder performance with each optimization includes the previous optimizations. Tables 4.1 and 4.2 present the total decoder execution time for decoding each of the 410 frames ten

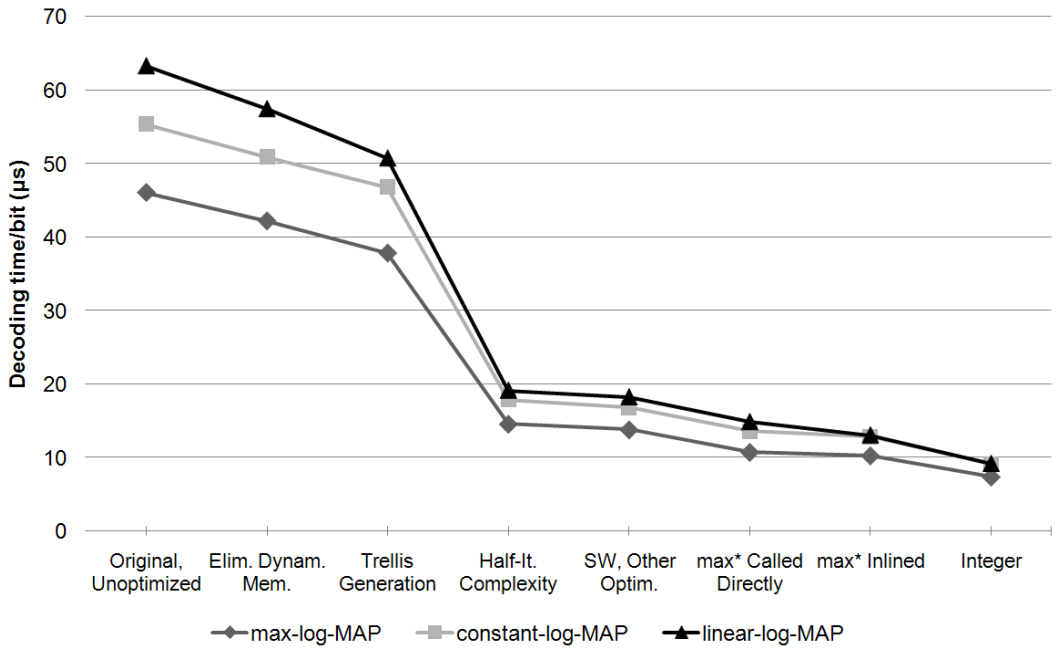


Figure 4.6: Decoding time/bit for QPSK (6,12) frame.

times, whose details were described in Section 4.5.1, using each max* variant and the speedup resulting from each optimization. Table 4.1 presents the results for decoding QPSK(6,12) frames, while Table 4.2 presents the results for decoding QPSK(36,72) frames. These two WiMAX scenarios were selected to illustrate the differences in decoding time for shorter and longer frame sizes. Results for the QPSK(18,36) and QPSK(60,120) scenarios are provided in Appendix B, and are consistent with the results for the scenarios presented here. The corresponding decoding times per bit for QPSK(6,12) and QPSK(36,72) frames after each optimization was applied are illustrated in Figures 4.6 and 4.7, respectively. Each figure compares the results for the max-log-MAP, constant-log-MAP, and linear-log-MAP decoders. The use of the decoding time per bit statistic allows the decoder performance for the shorter and longer frame sizes to be compared directly.

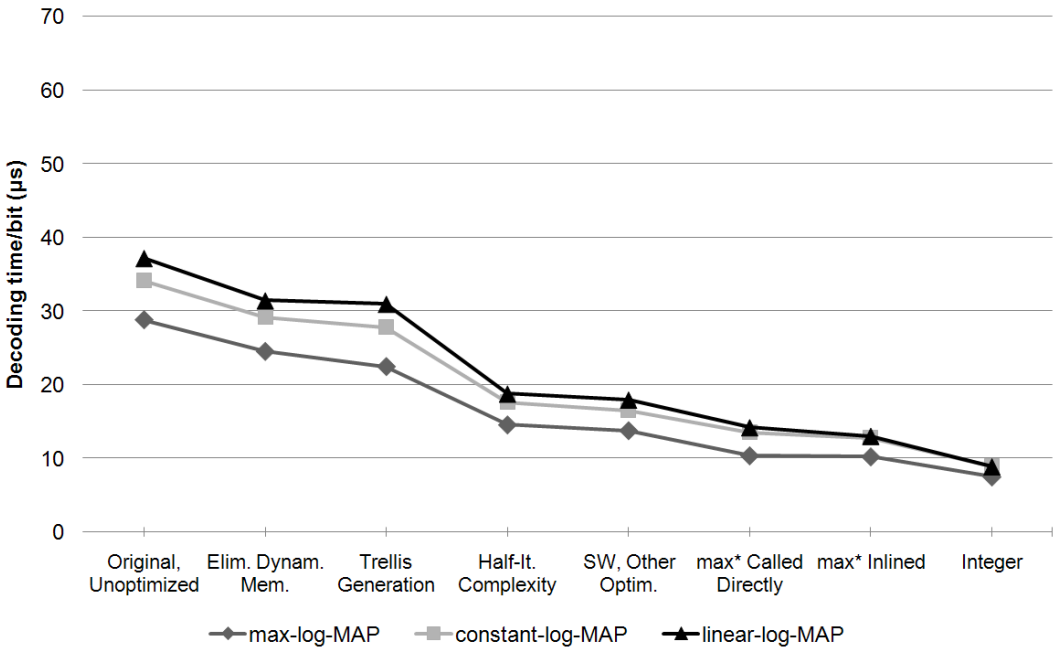


Figure 4.7: Decoding time/bit for QPSK (36,72) frame.

As expected, the decoding times provided in Tables 4.1 and 4.2 increase with the complexity of the max* variant used. The decoder utilizing the least complex max-log-MAP max* variant consistently offered the lowest execution time, while the most complex log-MAP decoder using `log` and `exp` function calls consistently offered the highest execution time. The corresponding decoding time per bit of the three max* variants illustrated in Figures 4.6 and 4.7 is consistently lowest for the max-log-MAP implementation and consistently highest for the linear-log-MAP implementation, as expected. Comparing the two frame sizes, decoding the QPSK(6,12) frame had a significantly higher time/bit compared with decoding the QPSK(36,72), up until the reduced half-iteration complexity optimization was applied. After the reduced half-iteration complexity optimization was applied, the decoding time/bit was much closer for the two frame sizes, however the smaller frame still exhibited

a slightly higher decoding time/bit. The significance of the reduced half-iteration complexity optimization results are explained in detail in Section 4.5.4. Decoding larger frames results in less time/bit since more time is spent in each of the turbo decoding functions, indicated in Figure 4.1, performing calculations for each frame index. Spending more time in a function mitigates the overhead of the function call. Processor cache effects could also explain the better decoding performance of larger frames, however this investigation is not pursued until the next chapter.

The elimination of the dynamic memory allocation optimization, explained in Section 4.2.1, provided a greater speedup for the larger frame size for each decoder implementation, ranging from 1.08 to 1.18, compared to 1.06 to 1.10 for the smaller frame size. The optimization to generate the trellis information only once on startup, rather than every decoder half-iteration, as explained in Section 4.2.2, provided a greater speedup for the smaller frame size than the larger frame size. This is expected, because generating the trellis information by calling the function `genTrellis` takes the same amount of time regardless of the frame size. Thus, `genTrellis` would contribute more to execution time for a smaller frame size than for a larger frame size. By generating the trellis information only once on startup, the execution time contributed by `genTrellis` is essentially eliminated.

The reduced half-iteration complexity optimization, explained in Section 4.2.3 provided the greatest speedup of all the optimizations, ranging between 2.60 and 2.72 for the small frame, and between 1.54 and 1.61 for the large frame. A detailed explanation of these results is provided in Section 4.5.4. The sliding window and other optimizations, described in Section 4.2.4, provided a relatively constant speedup across all implementations for both frame sizes, ranging from 1.05 to 1.08.

Table 4.1: Execution times (s) and speedups associated with optimizations for decoding QPSK (6,12) frames.

| **Optimization** | **max-log-MAP** | | **constant-log-MAP** | | **linear-log-MAP** | | **log-MAP (tbl)** | | **log-MAP (exact)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup |
| **Original, Unoptim.** | 9.07 | - | 10.90 | - | 12.46 | - | 14.04 | - | 79.29 | - |
| **Elim. Dynam. Mem.** | 8.31 | 1.09 | 10.03 | 1.09 | 11.31 | 1.10 | 12.86 | 1.09 | 74.90 | 1.06 |
| **Trellis Gen. Once** | 7.45 | 1.12 | 9.22 | 1.09 | 9.99 | 1.13 | 11.57 | 1.11 | 73.42 | 1.02 |
| **Half-It. Complexity** | 2.87 | 2.60 | 3.52 | 2.62 | 3.78 | 2.64 | 4.26 | 2.72 | 27.08 | 2.71 |
| **SW, Other Optim.** | 2.72 | 1.06 | 3.31 | 1.06 | 3.60 | 1.05 | 3.93 | 1.08 | 25.36 | 1.07 |
| **max* Called Dir.** | 2.12 | 1.29 | 2.69 | 1.23 | 2.94 | 1.23 | - | - | - | - |
| **max* Inlined** | 2.02 | 1.05 | 2.52 | 1.06 | 2.58 | 1.14 | - | - | - | - |
| **Integer Implem.** | 1.46 | 1.39 | 1.80 | 1.40 | 1.82 | 1.42 | - | - | - | - |

Table 4.2: Execution times (s) and speedups associated with optimizations for decoding QPSK (36,72) frames.

| **Optimization** | **max-log-MAP** | | **constant-log-MAP** | | **linear-log-MAP** | | **log-MAP (tbl)** | | **log-MAP (exact)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup |
| **Original, Unoptim.** | 34.01 | - | 40.30 | - | 43.94 | - | 48.47 | - | 275.54 | - |
| **Elim. Dynam. Mem.** | 28.95 | 1.17 | 34.45 | 1.17 | 37.14 | 1.18 | 41.23 | 1.16 | 254.94 | 1.08 |
| **Trellis Gen. Once** | 26.53 | 1.09 | 32.80 | 1.05 | 34.78 | 1.07 | 39.45 | 1.06 | 251.61 | 1.01 |
| **Half-It. Complexity** | 17.19 | 1.54 | 20.77 | 1.58 | 22.22 | 1.57 | 24.46 | 1.61 | 161.60 | 1.56 |
| **SW, Other Optim.** | 16.27 | 1.06 | 19.52 | 1.06 | 21.24 | 1.06 | 23.00 | 1.06 | 151.63 | 1.07 |
| **max* Called Dir.** | 12.25 | 1.33 | 16.03 | 1.22 | 16.81 | 1.26 | - | - | - | - |
| **max* Inlined** | 12.09 | 1.01 | 15.16 | 1.06 | 15.44 | 1.09 | - | - | - | - |
| **Integer Implem.** | 8.87 | 1.36 | 10.56 | 1.44 | 10.54 | 1.47 | - | - | - | - |

The cumulative speedup, averaged over all implementations, with all optimizations discussed up to this point is 3.36 for the QPSK (6,12) frame and 2.03 for the QPSK (36,72) frame size.

Calling the max* function directly rather than via a function pointer, an optimization discussed in Section 4.2.6, resulted in a fairly constant speedup averaging 1.26 for both frame sizes. Further optimization of the max* function by inlining resulted in average speedups of 1.03, 1.06, and 1.12 for the max-log-MAP, constant-log-MAP, and linear-log-MAP implementations, respectively. Compiler optimizations are likely the reason why the more complex max* variants achieved greater speedups as a result of inlining. Overall, the average speedup resulting from inlining the max* function over selecting it at run-time, with all optimizations up to Section 4.2.6, was 35% for the different frame sizes. This corresponds to an overall average speedup of 4.54 for decoding the smaller frame and 2.77 for decoding the larger frame over the original decoder implementation. The integer decoder implementation provided significant speedup, with speedups of ranging between 1.36 to 1.47 for the two frame sizes over an equivalent floating-point implementation. The integer decoder's execution performance is discussed in detail in Section 4.5.3.

### 4.5.3 Integer Decoder Execution Performance

Execution performance results of the integer decoder implementation for the max-log-MAP and linear-log-MAP implementations, discussed in Section 4.3, were significant. For the two frame sizes, the integer decoder implementation achieved speedups from 1.36 to 1.47, over the floating-point decoder implementation with all optimizations

discussed thus far. These speedups illustrate the performance advantage that integer arithmetic has over floating-point arithmetic for the particular system used. The linear-log-MAP decoder experienced the greatest speedup, as the calculations performed by its max* function are the most complex. As such, the linear-log-MAP max* function contributes a greater portion of the decoder's overall execution time than the less complex max* functions, as will be indicated in Section 4.5.5, and stood to gain the most improvement from integer implementation.

### 4.5.4 Improvement with Reduced Half-Iteration Complexity

The work performed each half-iteration, prior to the reduced half-iteration complexity optimization described in Section 4.2.3, was non-deterministic, as anywhere between two and four cycles could be performed. Each cycle, $\Gamma$, $A$, and $B$ metrics are calculated, and are thus calculated between two and four times each half-iteration. More than two cycles are performed if the $A$ or $B$ metrics differ by some margin from the values calculated the previous cycle. The $A$ and $B$ metrics are more likely to change between cycles when a low-SNR channel, a higher code rate, or a higher order modulation scheme is used, as the input LLRs are more noisy and less reliable. Also, for shorter frame sizes it was more likely that more than two cycles would be run, since it was less likely that the circular trellis state would be identified correctly, as a result of the tailbiting property discussed in Section 3.5. This is because for the first cycle, $A$ and $B$ metrics are initialized to zero, because they are unknown, while for subsequent cycles, $A$ and $B$ metrics are initialized to that of the previous cycle. For a shorter frame, the circular branch metrics are less reliable than that of a longer frame after the first cycle, because calculations are performed over fewer frame indexes.

Following the reduced half-iteration complexity optimization, $\Gamma$, $A$, and $B$ metrics are calculated only once each half-iteration. Therefore, following this optimization, the work performed by the turbo decoder is deterministic for a given frame size; the actual input LLRs and modulation used do not affect decoding. As mentioned, the speedup following the reduced half-iteration complexity optimization resulted in speedups ranging between 2.60 and 2.72 for the small frame, and between 1.54 and 1.61 for the large frame. The small frame benefited more from the optimization, as the original half-iteration design was more likely to run for greater than two cycles. Each of the different decoder implementations benefited similarly from the optimization for a given frame size.

### 4.5.5 Execution Time Profile

To direct subsequent optimization efforts of the software, the performance of the decoder was analyzed to determine which portions of the turbo decoding software contribute significantly to the overall execution time. The decoding software with all optimizations presented up to Section 4.2.4 was compiled and linked using the `-pg` flag of the `gcc` compiler to enable profiling. Profile statistics were obtained using the GNU profiler `gprof`, which allows the performance of a program to be examined on a per-function basis in terms of time, number of calls, and the percentage contribution to overall execution time [Fen00]. Profile data was collected for frame sizes of 6, 18, 36, and 60 bytes. The turbo decoder software structure, including the hierarchy of function calls, is illustrated in Section 4.2.5. Figure 4.8 shows the percentage contribution to the total execution time for the most significant functions when decoding a 36-byte frame using the max-log-MAP, constant-log-MAP, and linear-log-MAP max*

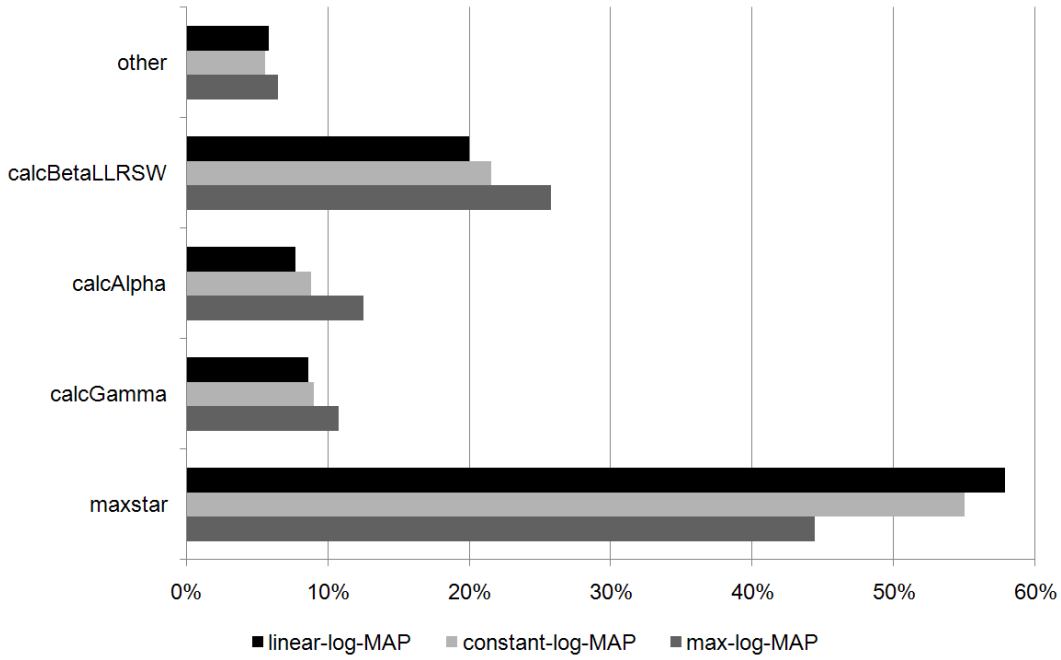


Figure 4.8: Function contribution to overall execution time.

variants. The profile statistics show a similar trend for the other frame sizes. For all max* variants, the max* function contributes the most to the total execution time, with the more complex max* variants contributing more. Profiling for the log-MAP table and log-MAP exact max* decoder implementations show that the max* function constitutes an even higher portion of the overall execution time, as expected.

### 4.5.6 Max-Log-MAP and Linear-Log-MAP Decoders

The max-log-MAP decoder using eight iterations and the linear-log-MAP decoding using four iterations offer comparable BER performance, as indicated earlier in Section 4.4.2. Therefore, a comparison of the execution performance of these two decoder implementations was conducted. For QPSK rate-1/2 frame sizes of 6, 18, 36, and 60

bytes, the performance exhibited by the linear-log-MAP decoder implementation using four iterations was approximately 40% better than that of the max-log-MAP decoder using eight iterations. For integer implementations of the two decoders, the linear-log-MAP decoder using four iterations performed on average approximately 63% better than the max-log-MAP decoder using eight iterations. The linear-log-MAP decoder benefited the most from integer implementation, as discussed in Section 4.5.3, thus this result is expected.

## 4.6 Chapter Summary

This chapter has discussed the adaptation of the decoder software from the CML into a stand-alone C decoder implementation. Following adaptation, a number of optimizations were performed to reduce the execution time of the decoding software, while preserving its BER performance. An integer decoder implementation was pursued, including the adaptation of linear-log-MAP max* variant to support operations on integer LLRs. BER simulations of the original, fully optimized, and integer decoders were performed, and the effect that a chosen max* operator has on performance was examined. The reduction in execution time that resulted from each optimization was evaluated through simulations performed on a general-purpose PC, and any significant results were explained.

# Chapter 5

# Enhanced Turbo Decoding

Following the software-only optimizations that were pursued and evaluated using a general-purpose PC in the previous chapter, enhancements that are targeted towards more customized hardware implementations were considered, and are discussed in this chapter. The previous software optimizations were necessary in bringing the performance of the decoding software to the highest level possible, before using any special hardware support. The enhancements to the optimized decoder software in this chapter take advantage of special hardware support, including multiprocessor and vector functionality.

The various enhancements that were pursued include the introduction of special instructions to accelerate operations commonly performed during decoding using the log-MAP algorithm, described in Section 3.3, assuming a simple single-issue reduced instruction set computer (RISC) architecture with ideal one-cycle-per-instruction execution. A special instruction to speed up the retrieval of trellis information is discussed in Section 5.1. Accelerating the frequently performed max* calculation with a special instruction for both the max-log-MAP and linear-log-MAP implementations

is discussed in Section 5.2. Following enhancements using special instructions, a parallel turbo decoder design was pursued and subsequently optimized, for execution on a multiprocessor system architecture. Details of the designed flexible parallel MAP decoding algorithm are discussed in Section 5.3, while implementing details using multithreading are discussed in Section 5.4. Further optimization of the parallel decoder and its implementation are described in Section 5.5. Finally, the potential performance benefit of turbo decoding using vector processing is discussed in Section 5.7. The execution performance improvements from the enhancements are evaluated in the following chapter using a custom multiprocessor simulator.

## 5.1 Special Instruction for Trellis Look-up

During log-MAP decoding, information on a particular code's trellis is used extensively in the calculation of $\Gamma$, $A$, $B$, and $\Lambda$, as defined in Equations 3.8, 3.6, 3.7, and 3.5, respectively. Referring to Figure 3.2, the trellis information includes the start state, the double-binary input associated with a particular branch ($ab$), referred to as the trellis input, the parity bits associated with a particular branch ($yw$), referred to as the trellis output, and the end state. Following the optimizations described in Section 4.2.2, trellis information is generated on program startup and stored in arrays, requiring array look-ups during decoding to obtain the required information. For example, the calculation of $A_k(s_k)$ for a particular state $s_k$ at trellis index $k$ requires 8 trellis information array look-ups. This means that $A$ calculations require $8 \times 8 = 64$ array look-ups for each trellis index, for a total of $64N$ look-ups in each half-iteration for a frame with $N$ pairs of bits. Similarly, $64N$ trellis array look-ups are required for the calculation of $\Gamma$ and $B$ metrics, while $\Lambda$ calculations require $96N$ array look-ups,

in each half-iteration. Therefore, a total of $2(64N + 64N + 64N + 96N)I = 576NI$ trellis array look-ups are required for decoding a frame of size $N$ using $I$ decoder iterations.

Providing a method to speed up the retrieval of trellis information would certainly improve decoder performance, given the extensive use of trellis information when decoding a frame. If it is assumed that the base address of a trellis array is already present in a register, a RISC architecture would take one cycle to calculate the actual address of the array element needed, plus another cycle to perform the load. A special instruction could be implemented to allow trellis information to be obtained in a single cycle. The instruction would require two input operands in registers, one to specify the desired trellis information and an index to specify a particular trellis branch. In hardware, a look-up table or special memory embedded in the processor would store the trellis information, which would be accessed by this special instruction. The result would be returned in an output register. For the WiMAX turbo code, because there are 32 branches associated with each trellis stage and four pieces of information associated with each branch, only 128 values need to be stored. The values are all integers in the range $0 \ldots 7$, so 3 bits would be needed for each value, resulting in a total of 48 bytes of storage required.

## 5.2 Accelerating the max* Calculation

The max* operation is used extensively throughout the MAP algorithm, as outlined in Section 3.3. The calculation of all $A$ metrics for each frame index requires 32 max* function calls per half iteration. For a frame with $N$ pairs of bits requiring $I$ decoder iterations ($2I$ half-iterations), this calculation results in a total of $2 \times I \times 32N = 64NI$

max* operations. Similarly, there are $64N$ max* operations used in the calculation of each $B$ and $\Lambda$, along with $4N$ calls made during the calculation of bit LLRs during the last decoder iteration. As a result, a total of $(192I+4)N$ max* operations required during the decoding of a frame. Therefore, there is an opportunity to increase decoder throughput by reducing the cycles required to perform all of these max* operations. To accelerate the max* calculation, a special RISC-style instruction could be introduced, which would eliminate the use of more general C code to implement the required functionality. Such an instruction would have two input operands in registers, while the result would be returned in an output register.

The max* operation in the max-log-MAP algorithm performs a single floating-point maximum operation for two input values. In the original C decoder, it is implemented as a comparison to select the larger value, requiring a number of cycles to compute when compiled to normal machine instructions. Implementation of this functionality in a single special instruction and inlining it could allow the max-log-MAP max* operation to be performed in a single cycle, rather than multiple instructions using general C code.

As the linear-log-MAP algorithm offers a good trade-off between BER performance and complexity, it was desirable to investigate what performance benefit could be achieved by providing a special instruction to implement it as well. Referring to Equation 3.14, the linear-log-MAP computation potentially requires a multiplication, a maximum operation, and three additions/subtractions to generate the result from two input values. Special hardware could be introduced to perform the required calculations as described below, which would be invoked by the special instruction.

As done for the integer version of the linear-log-MAP max* operation in Section

4.3, multiplication by $a = -0.24904$ can be approximated by division by $b = -4$. Division by $b = -4$ can be performed in hardware by performing a right shift by two bit positions, and taking the negative of the result. The right shift operation does not need to be explicitly done; it is sufficient to discard the least significant two bits of the operand, saving a cycle of execution. An annotated version of the linear-log-MAP operation is provided below, re-arranged so that $b = 4$ is used, with the cycle in which each operation could be performed indicated as superscripts:

$$\max^*(\lambda_1, \lambda_2) \approx \begin{cases} \max(\lambda_1, \lambda_2)^1, & \{\text{if } |\lambda_2 - \lambda_1|^1 > T\}^2 \\ \{\max(\lambda_1, \lambda_2)^1 + \{(T - |\lambda_2 - \lambda_1|^1)/b\}^2\}^3, & \{\text{if } |\lambda_2 - \lambda_1|^1 \leq T\}^2 \end{cases}, \tag{5.1}$$

where again $T = 2.5068$. All potentially needed computations are performed in parallel, and after comparisons are done in the second cycle, it is known which return value will provide the result of the two possibilities, and what calculation must be done during the third cycle, if any. From this analysis, it was conservatively assumed that the linear-log-MAP max* instruction would require a constant 3 cycles for execution, due to the sequence of operations outlined above.

## 5.3 Parallel MAP Decoding

Maximizing the throughput of a turbo decoder can be accomplished by exploiting the parallelism of the MAP algorithm. This is accomplished using a technique similar to the sliding window, as discussed in Section 3.6, where a frame is divided into a number of sub-blocks and the MAP algorithm is applied to each sub-block independently. Parallelism is exploited by processing all sub-blocks simultaneously. As each CRSC decoder performs MAP decoding to generate its output LLRs, the time spent during

each half-iteration by a CRSC decoder constitutes the parallel execution portion of the algorithm described below. The sequential portion of the algorithm is composed of less complex tasks performed between half-iterations, as indicated in Figure 3.3, which includes the generation of input LLRs and extrinsic information, interleaving and de-interleaving, and making hard bit decisions.

To maximize the flexibility of the software turbo decoder, it was desirable to support an arbitrary degree of parallelism, which translates into decoding a frame in parallel using an arbitrary number of sub-blocks. There are a number of ways to apply the MAP algorithm to each sub-block by varying the order and timing of the $A$, $B$, and $\Lambda$ calculations. A number of parallel turbo decoding structures have been proposed in previous work, including an analysis of their design trade-offs [Zha04]. The original scheme was presented using four sub-blocks, and border metrics were initialized using extra *dummy* metric calculations. The parallel turbo decoding algorithm in this paper was implemented using a design similar to the double-windowing scheme, but without the need for *dummy* calculations and adapted to support an arbitrary number of sub-blocks, as described below.

Figure 5.1 shows the same four sub-blocks previously shown in Figure 3.4, but now they are processed in parallel to ideally reduce the time required by a factor of four, neglecting any serial overhead. As shown in Figure 5.1, there are two types of sub-blocks identified for the parallel MAP algorithm. For type I sub-blocks (SB1 and SB3 in Figure 5.1), $A$ metrics for the entire sub-block are calculated first in the forward direction, followed by $B$ and $\Lambda$ calculations in the reverse direction. For type II sub-blocks (SB2 and SB4 in Figure 5.1), $B$ metrics are calculated first for the entire sub-block in the reverse direction, then $A$ and $\Lambda$ calculations are performed in the

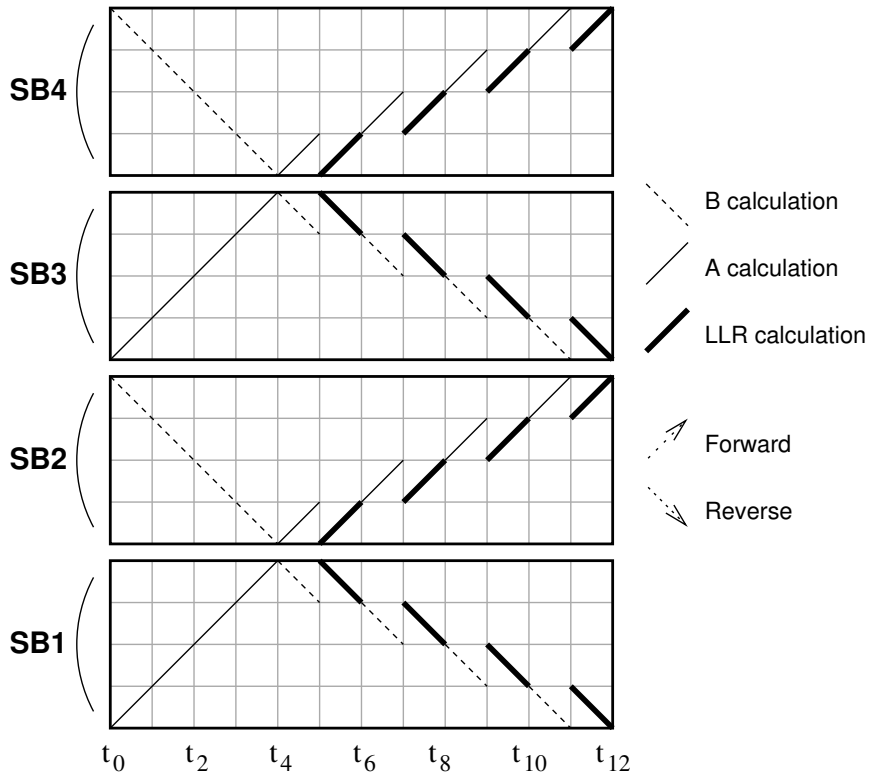


Figure 5.1: The parallel double-window decoding scheme.

forward direction.

When $A$ and $B$ metric calculations are started somewhere in the middle of the frame, they must be initialized as done in the sliding window technique. In Figure 5.1, SB2 $B$ metrics and SB3 $A$ metrics must be initialized at the start of the iteration. To obtain reliable metrics at the borders of a sub-block, the decoder uses metrics from the previous iteration, which has been shown to perform well [Kim08]. Again, border $A$ and $B$ metrics for all states are initialized to zero, because no prior information is available. At time $t_4$ in Figure 5.1, $A$ and $B$ metrics must be initialized before decoding can proceed. These border values can be obtained from adjacent sub-blocks,

where metrics have already been calculated up to the border. SB1 obtains $B$ initialization values from SB2, while SB2 obtains $A$ initialization values from SB1. In a similar manner, SB3 and SB4 exchange metrics at time $t_4$. The initialization and exchange of metrics between sub-blocks has been generalized to an arbitrary number of sub-blocks for the parallel software turbo decoder described in this work.

A parallel decoder's BER performance may exhibit some degradation as the degree of parallelism is increased. This behaviour is due to more unknown border metrics requiring initial estimates as the number of sub-blocks is increased. Results for BER simulations will be presented in Section 6.4.1 to assess this aspect.

## 5.4 Multithreaded Turbo Decoder Design

To support multithreaded execution using the parallel MAP decoding algorithm described in the previous section, a new version of the decoder software was created that is based on the sequential version as optimized in Chapter 4. In the sequential decoder software, the sliding window algorithm was implemented to first perform $A$ calculations for the entire frame, followed by $B$ and $\Lambda$ calculations for each trellis index. This is the same order that calculations are performed for type I sub-blocks, thus the functions `calcAlpha` and `calcBetaLLR`, as shown in Figure 4.1, were modified to support multithreaded execution by performing the calculations on a particular sub-block instead of the entire frame. In addition, two new functions were implemented for multithreaded execution to support the calculations performed by type II sub-blocks, namely `calcBeta` and `calcAlphaLLR`. `calcBeta` first calculates $B$ values for the entire sub-block, while `calcAlphaLLR` calculates $A$ and $\Lambda$ values incrementally for each trellis index.

All of the necessary software threads are created on program startup, and persist throughout execution, as the algorithm is performed the same way over multiple iterations. For any serial phases of activity, one thread can be responsible for that computation, and the other threads can busy-wait on a barrier. As explained in the previous section, parallel activity occurs during each CRSC decoder half-iteration, while serial activity consists of less complex tasks between half-iterations. Thus, threads busy-wait on a barrier before a half-iteration begins, to ensure the required serial computations have completed. Similarly, threads busy-wait on a barrier at the end of a half-iteration before serial activity resumes, to ensure all threads have completed their respective calculations during the parallel portion of the algorithm. Parallel efficiency is high when the amount of work in the parallel phases is much larger than in the serial phases, which is the expectation for the multithreaded turbo decoder.

Threads are assigned to a particular sub-block, and once each has calculated either $A$ or $B$ values for their entire sub-block, via calls to `calcAlpha` or `calcBeta`, threads exchange border values with their neighbours. A barrier synchronization event occurs to ensure threads have completed their respective calculations up to this point, and have written their border values to memory. Once all threads have reached the barrier, each one reads the required border metrics from memory, and proceeds with the second set of calculations, with calls to either `calcBetaLLR` or `calcAlphaLLR`.

## 5.5 Optimization of the Parallel Decoder

Following the original multithreaded decoder implementation utilizing the described parallel algorithm, further optimizations were pursued to improve performance. Referring to Figure 3.3, the serial activity performed between half-iterations includes the generation of $V(X)$ and $\Lambda_e(X)$, interleaving and de-interleaving, and making hard bit decisions. Increasing the parallelism of the decoder could be done by shifting work that is done serially and distributing it among all threads to be done in a parallel manner. The reduced serial execution time would allow a greater speedup to be obtained by the parallel decoding software.

The generation of $V(X)$ and $\Lambda_e(X)$ involves simple array addition/subtraction, which is performed serially by a single thread. $V(X)$ is used in the calculation of $\Gamma$, as defined in Equation 3.8. $\Gamma$ calculations are performed by threads for their respective sub-blocks, thus the generation of $V(X)$ can be included in the calculation of $\Gamma$, as defined by the revised equation

$$\Gamma_k(s_k, s_{k+1}) = V_k(X_k) + L_k(Z_k) = L_k(X_k) + \Lambda_{e,k}(X_k) + L_k(Z_k). \tag{5.2}$$

Doing so effectively parallelizes the calculation of $V(X)$. Similarly, the calculation of $\Lambda_e(X)$ can be shared by all threads, where each thread calculates the extrinsic information associated with its respective sub-block. On the last iteration, the second CRSC decoder outputs *a posterior* LLRs, $\Lambda_2^I(X)$, instead of extrinsic LLRs, to be de-interleaved and hard bit decisions made. The revised calculations result in each CRSC decoder now receiving three sets of LLRs as input:

- the systematic channel LLRs, $L(X)$,
- the parity channel LLRs, $L(Z)$, and

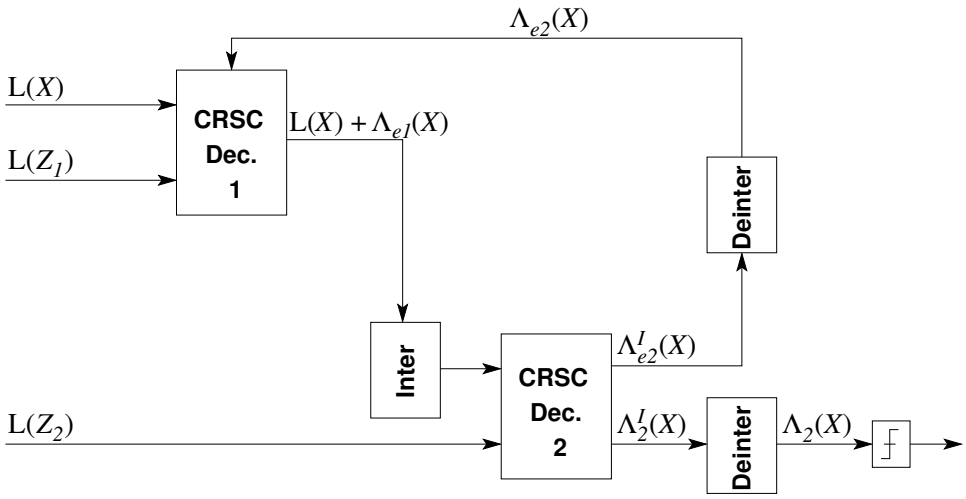


Figure 5.2: Parallel turbo decoder design following optimizations.

- extrinsic information provided from the other CRSC decoder, $\Lambda_e(X)$.

The updated decoder design following the described optimizations is shown in Figure 5.2. CRSC decoder 1 outputs both systematic channel and extrinsic LLRs to be interleaved and used by CRSC decoder 2, and are calculated as:

$$L(X) + \Lambda_{e1}(X) = \Lambda(X) - \Lambda_{e2}(X). \tag{5.3}$$

This avoids having to add $L(X)$ to $\Lambda_{e1}$ during the serial phase of computation, before interleaving is performed.

## 5.6 More Accurate Modelling of Execution

When evaluating the decoder's performance through simulation, ideal execution can be assumed for simplicity, where one instruction is executed every cycle. In an actual system implementation, memory operations associated with loads and stores introduce additional latency, as discussed in Section 2.4. This is because the memory

latency in a single-chip system typically causes the processor to idle while a load or store is being performed. Thus, in the absence of caches, a processor must wait several cycles for a load or store to complete before execution continues. Modelling execution performance without delays associated with memory simplifies simulation, and still gives a good impression of execution performance for comparison as enhancements are made to the software and custom architectural features are added. Modelling the interaction of caches and memory in a multiprocessor system, and their inherent delays, through simulation provides a more realistic assessment of the performance for an actual hardware implementation.

### 5.6.1 Parallel Turbo Decoder Data Sharing

For a multiprocessor system with caches, additional cache misses are incurred as a result of data sharing among processors beyond the misses for first-time accesses or limited cache capacity. In the parallel turbo decoding algorithm described in Section 5.3, there is data shared between pairs of threads, and data shared between the thread performing serial computations and all other threads. Data shared between pairs of threads includes metrics exchanged during a decoder half-iteration, as well as metrics shared between half-iterations. Pairs of threads exchange border $A$ and $B$ metrics during each decoder half-iteration, where a particular thread writes one set of border metrics ($A$ or $B$) for reading by an adjacent thread, followed by reading one set of border metrics ($B$ or $A$) previously written by the same adjacent thread. At the end of a particular half-iteration (first or second), each thread writes border values for the metric calculated last, due to either tailbiting or parallel decoding, for initialization by an adjacent thread at the start of the following same half-iteration (first or second).

By referring to Figure 5.2, data that is shared between the serial and parallel decoder phases of the optimized parallel decoder software primarily involves the extrinsic information generated by each CRSC decoder. During a half-iteration, each thread accesses a subset of the shared data associated with its assigned sub-block, while during serial phases one thread accesses data for the entire frame. For example, one serial thread writes de-interleaved extrinsic values, $\Lambda_{e2}(X)$, of which a subset are then read by each thread associated with the sub-block assigned to it during CRSC decoder 1 processing. During CRSC decoder 1 processing, each thread writes output LLRs associated with their sub-block, which are subsequently read by one thread when interleaving is performed serially. Similar data sharing is associated with CRSC decoder 2 processing. On the second half of the last iteration, bit LLRs instead of extrinsic symbol LLRs are generated by each thread for their associated sub-block, which are read when hard bit decisions are made during the following serial phase.

Recognizing the data sharing patterns inherent in the parallel turbo decoding algorithm, and modelling the associated cache, memory and bus arbitration behaviours in a multiprocessor system, gives a more accurate assessment of the turbo decoder's expected execution performance on actual hardware.

## 5.7 Vectorizing the Turbo Decoder

Following the addition of special instructions and parallelization to speed up execution of the turbo decoder software, an initial investigation into using vector processing to further enhance performance was conducted. Algorithms that perform vector-type operations can benefit substantially on a single-instruction, multiple-data processor

(SIMD) architecture, where a single instruction operates on multiple data items simultaneously, as described in Section 2.4.3. Vector processing adds an additional degree of parallelism to the execution of the decoder software, with finer granularity than using parallel threads. Both approaches require additional hardware; using more threads for parallel execution requires additional processors, while adding vector support or increasing the vector length used requires additional vector computation units within a processor.

Depending on the desired performance of the turbo decoder, a combination of multiple processors and processing using vector operations of a particular length can be exploited to increase the performance of the decoding software. Given the iterative nature of the log-MAP decoding algorithm discussed in Section 3.3, where the same calculations are performed over multiple states and trellis branches for every frame index, execution of the parallel decoder software on a vector architecture could further increase performance. An initial investigation into the performance benefit that can be achieved by combining parallel execution using multiple processors with vector execution was desired. There have been recent turbo decoder implementations that make use of SIMD vector architectures, however these systems either consist of complex processing elements or are architectures designed specifically for turbo decoding. The approach used in this research involves using a general multiprocessor architecture, and adding vector support to simple processing elements.

### 5.7.1 Vector Nature of Turbo Decoding

The majority of calculations by each CRSC decoder are executed in an iterative manner. The calculation of $\Gamma$ metrics at each frame index is performed for each trellis

branch, for a total of 32 calculations per index. $A$ and $B$ calculations are performed for each trellis state at each index, or eight metric calculations per index. The calculation of output LLRs requires iterating over every trellis branch for each frame index. Essentially, the majority of execution during each CRSC decoder iteration consists of performing repetitive calculations using `for` loops, which could be vectorized to reduce execution time. The serial computation performed between half-iterations mainly involves array manipulation, which could also benefit from vectorization.

To illustrate vectorization of the types of `for` loops mentioned above, the example below shows pseudo-code for a simple loop that simply adds two arrays of 32 numbers each, A and B, multiplies them by some constant, `MULT`, and stores the result in another array, C. The pseudo-code of the described operations on a simple scalar processor is provided below:

```
initialize counter i to 0
execute this loop 32 times
      load array element of A at index i, A[i]
      load array element of B at index i, B[i]
      add A[i] and B[i]
      multiply sum by MULT
      store result in array element of C at index i, C[i]
      increment i by 1
end loop
```

Assuming that a vector architecture is available with a vector length of 8, the pseudo-code for the described operations looks like this:

```
initialize counter i to 0
execute this loop 4 times
      load 8 array elements of A starting at index i, A[i..i+7]
      load 8 array elements of B starting at index i, B[i..i+7]
      add A[i..i+7] and B[i..i+7], element-by-element
      multiply sums by MULT
      store result in C array starting at index i, C[i..i+7]
      increment i by 8
end loop
```

For the vectorized version of the pseudo-code, the entire loop gets executed only four times instead of thirty-two, as each vector operation operates on eight array elements. Initialization of and incrementing the counter variable remains a scalar operation, thus the speedup obtained by the vectorized code would be less than eight, as a result of this loop overhead. Identifying areas of the parallel turbo decoder software that are vectorizable in a similar manner and assessing the potential performance improvement from doing so are investigated in the next chapter.

## 5.8 Chapter Summary

This chapter discussed various enhancements to the decoder software that were pursued, which take advantage of special hardware support and multiprocessor execution.

Special instruction support for commonly performed operations were presented, including a max* instruction for both the max-log-MAP and linear-log-MAP variants, and an instruction to accelerate the retrieval of trellis information. Next, a parallel decoder design was presented, followed by the necessary multithreaded design aspects required to implement the parallel algorithm for execution on a multiprocessor platform. More accurate modelling of execution of the multithreaded decoder software was discussed, including accounting for memory and cache delays, and the nature of data sharing of the designed parallel decoder software. Finally, the vector nature of the parallel turbo decoding software was briefly described, and the potential for performance improvement that exists with execution on a vector processor architecture. The resulting performance improvements from simulation of the various enhancements described in this chapter are presented in Chapter 6.

# Chapter 6

# Enhanced Turbo Decoding Results

The chapter provides simulation results for the decoder enhancements described in Chapter 5. A custom multiprocessor simulator was used to evaluate the decoder's execution-time performance, whose details are described in Section 6.1. A baseline for decoder performance is established in Section 6.2, prior to the addition of any enhancements or use of multiprocessing. The resulting performance improvement of the decoder software on a single processor with the addition of special instructions for accelerating both trellis information look-ups and max* operations is presented in Section 6.3. The multiprocessor performance of the parallel implementation of the decoder software is discussed in Section 6.4, along with an assessment of the BER performance from multithreaded execution. The performance of the parallel decoder with more accurate modelling of delays associated with memory and caches is presented in Section 6.5. Finally, initial results for decoding using vector processing are presented in Section 6.6.

Ideal simulation is used initially for the evaluation of the improvement resulting from the inclusion of special instructions and parallel execution, for simplicity.

Essentially, the ideal simulation results do not assume a particular multiprocessor architecture and communication network. Simulation results using more accurate simulation, described in Section 5.6, would be expected to show similar results, however ideal simulation still provides a good assessment of the available parallelism and the potential improvement from enhancements. Parallel simulated execution times that account for the delays associated with processor caches and a single main memory using a shared bus communication network are investigated to determine the viability of such a system organization.

## 6.1 Custom Multiprocessor Simulator

To evaluate the software turbo decoder's performance, a customized version of the SimpleScalar tool-set supporting multiprocessor simulation was used [Man01]. The modelled shared-memory multiprocessor architecture consists of single-issue processors with ideal one-instruction-per-cycle execution. The decoder code was compiled using the `gcc` compiler version 2.6.3, with optimization level `-O2`, into the MIPS-derived portable instruction set architecture (PISA) for interpretation by the SimpleScalar simulator. The SimpleScalar simulator provides detailed statistics of program execution, including the total number of execution cycles and the number of instructions executed by each thread. The special instructions for trellis information, described in Section 5.1, and the max* calculation, described in Section 5.2, were implemented in SimpleScalar in order to assess their benefit. The speedup values obtained for idealized functional simulation were found to be largely independent of frame size.

### 6.1.1 Simulator Modifications

The output of the SimpleScalar simulator provides detailed statistics of execution, including the total number of execution cycles, the number of instructions executed by each thread, and information regarding memory usage. By analyzing these statistics, the performance of the decoder can be characterized in terms of $T_N$, the total execution cycles using $N$ threads, $T_O$, the execution overhead associated with thread creation and termination, program initialization, file I/O, etc., $T_S$, the serial portion of decoder execution, and $T_P$, the parallel portion of decoder execution. In an actual SDR decoder implementation, $T_O$ would have an insignificant impact on execution time, since any program overhead would happen rarely on occasional startup/shutdown, while the time spent decoding frames would dominate. To minimize the simulation time, a single frame is decoded using a specified number of iterations. The execution time of decoding a single frame can be expressed as:

$$T_N = T_O + T_S + T_P \tag{6.1}$$

$$T_N* = T_S + T_P \tag{6.2}$$

where $T_N*$ is the execution time ignoring the overhead described previously. $T_P$ is the time spent performing CRSC decoding in parallel using the algorithm described in Sections 5.3 and 5.4, while $T_S$ is the time spent in between decoder half-iterations, when the generation of extrinsic information, interleaving and de-interleaving, and making hard bit decisions are performed, as indicated previously in Figure 3.3. The results in Sections 6.2 and 6.3 are for the sequential decoder, hence $T_P = 0$ and $T_N* = T_S$. Later results in Section 6.4 involve parallel execution to reduce $T_N*$.

The SimpleScalar simulator initially would only provide the total execution cycles, from program initiation to termination. Therefore, the execution cycles reported was

$T_N$, while the desired execution cycles to more accurately assess performance is $T_N*$. To obtain simulation execution cycles associated with decoding only, and to ignore execution cycles from overhead, $T_O$, the simulator was modified. A special instruction was implemented in SimpleScalar which allowed the instruction count for a particular processor to be reset to zero. This instruction is executed in the simulated application code prior to the start of decoding a frame. Even though the execution cycles counted after being reset still include program termination, that contribution was found to be small in comparison with the total cycles, and hence would have a negligible contribution. Thus, the execution cycles presented for the remainder of this chapter are $T_N*$, as measured with this modification to the simulator.

## 6.2 Baseline Decoder Performance

To establish a performance baseline for comparison, simulations were performed with the SimpleScalar simulator using the sequential decoding software with all optimizations from Chapter 4, but none of the enhancements in Chapter 5. Both the max-log-MAP with 8 iterations and the linear-log-MAP with 4 iterations decoder variants were simulated, as these two implementations have similar bit-error rate (BER) performance, as discussed in Section 4.4.2. The execution cycles and cycles per bit for decoding rate-1/2 QPSK frames of sizes 6, 18, 36, and 60 bytes are presented in Table 6.1. The amount of work performed by the decoding software only depends on frame size, while the particular modulation scheme has no effect.

The performance of the linear-log-MAP decoder was uniformly better by 82%than that of the max-log-MAP decoder. An important aspect to note from the results in Table 6.1 is that the cycles executed per bit for the different frame sizes are relatively

Table 6.1: Execution cycles and cycles/bit (both in thousands) for the max-log-MAP and linear-log-MAP baseline decoders.

| **Frame size** | **max-log-MAP** | | **linear-log-MAP** | |
|---|---|---|---|---|
| | cycles, $T_S$ | cycles/bit | cycles, $T_S$ | cycles/bit |
| **QPSK (6,12)** | 1759 | 36.6 | 968 | 20.2 |
| **QPSK (18,36)** | 5234 | 36.3 | 2873 | 19.9 |
| **QPSK (36,72)** | 10438 | 36.2 | 5724 | 19.9 |
| **QPSK (60,120)** | 17414 | 36.3 | 9542 | 19.9 |

constant for each of the decoder implementations. In other words, although more time is spent in total decoding larger frames, the amount of work required per bit does not change. Thus, execution performance of the decoder does not actually depend on the frame size or the modulation used. As a result, to evaluate the performance of the decoder from this point forward, a single frame size is used for convenience, as evaluation using other frame sizes yields similar results.

## 6.3 Speedup from Special Instructions

This section presents simulation results obtained from implementing special instructions that accelerate commonly performed decoding operations, which would be supported in an actual system implementation through the inclusion of a modest amount of special hardware. The resulting execution performance for inclusion of the special instructions described below are for decoding a QPSK (18,36) frame, for the max-log-MAP decoder with 8 iterations and the linear-log-MAP decoder with 4 iterations. All of the optimizations in Chapter 4 are included for these results.

A new instruction was implemented in SimpleScalar to allow trellis information

to be obtained in a single cycle, an enhancement described in Section 5.1. The two decoder implementations were modified to make use of the trellis instruction and again simulated in SimpleScalar using a single thread. The performance obtained for both implementations showed a fairly uniform improvement of approximately 11% over the baseline decoder. For the max-log-MAP decoder, the cycle savings resulting from the new instruction was 566,222 for decoding a frame size of $N = 72$ pairs of bits (18 bytes). Using the relation derived in Section 5.1, the total number of trellis information look-ups required is $576NI = 331,776$, using $I = 8$ decoder iterations and $N = 72$ pairs of bits. Therefore, the average savings per trellis information look-up is $566,222/331,776 = 1.71$ cycles by using the special instruction. This is more than the one-cycle savings that was predicted, but could be explained if the base array address has to be reloaded often into a register during some loop iterations.

Analysis of the assembly code produced by the compiler indicated that each max-log-MAP max* function call requires 6 cycles to execute, with call/return statements. Because the max-log-MAP max* operation performs a single floating-point maximum operation, common among other architectures, it was assumed it could be performed in a single cycle by one inlined instruction, an enhancement described in Section 5.2. The decoder software was modified to use the max-log-MAP max* special instruction and simulated in SimpleScalar, which resulted in an overall speedup of 19%. A savings of 754,193 cycles was realized for decoding a frame size of $N = 72$. Using the relation described in Section 5.2, decoding an $N = 72$ frame requires $(192I + 4)N = 110,880$ max* operations using 8 decoder iterations. This translates into an average savings of $754,193/110,880 = 6.8$ cycles per max* operation, more than the 5 cycles expected. Analysis of the assembly code with and without the special

instruction revealed that the C function implementation of the max* function requires three registers: two for the function arguments and one for the return value. Use of the special instruction also requires three registers, however the return value is stored in one of the registers used for the input arguments. The compiler does not allow functions to return values in registers used for the input arguments. Therefore, using the special instruction frees up a register, which could explain the better-than-expected performance improvement.

The linear-log-MAP max* instruction was also implemented in SimpleScalar, assuming a constant 3 cycles for execution, as determined in Section 5.2. The assembly code of the C function revealed that it takes up to 20 cycles to execute, with call/return statements, so a substantial savings was predicted. Simulation results for the linear-log-MAP decoder indicated a 30% improvement as a result of introducing a special instruction to perform the corresponding max* calculation. The number of max* operations required using a frame size of $N = 72$ and $I = 4$ decoder iterations is $(192I + 4)N = 55,584$. Given that the cycles saved as a result of the linear-log-MAP max* instruction was 627,049, the average savings per max* operation is $627,049/55,584 = 11.3$ cycles.

The discussion from inclusion of special instructions thus far characterized the individual improvement each instruction had by itself on decoder performance. When the trellis and max* instructions were used together in the max-log-MAP decoder implementation, the resulting performance improvement was 29% over the baseline performance. Similarly, when both instructions were used together in the linear-log-MAP decoder implementation, the overall performance improvement was 40%. When the performance of the two decoder implementations with both special instructions

are compared, the linear-log-MAP decoder exhibits approximately 90% higher performance than the max-log-MAP decoder. This is an increase from the 82% difference obtained with the baseline decoder, due to a more substantial cycle savings obtained from the linear-log-MAP max* instruction.

## 6.4 Parallel Decoder Performance

This section provides simulation results on the parallel turbo decoder software, whose design was discussed in Sections 5.3 and 5.4, in terms of both BER and execution performance. As indicated earlier, the algorithmic optimizations from Chapter 4 are already included for all of these results.

### 6.4.1 BER Performance

BER simulations were performed with the designed parallel decoder software using CML to observe to what extent BER degradation occurs as decoder parallelism is increased, an expected characteristic explained in Section 5.3. Due to multithreading issues with MATLAB, BER simulations were performed with a serial version of the designed parallel decoding algorithm. The algorithm is the very same one used to evaluate the execution-time performance, however sub-blocks are processed serially by a single-thread. The functions used are the same as the multithreaded decoder, and border metrics are initialized as described in Section 5.3. Thus, the BER performance of the decoder used for MATLAB simulations is the same as the decoder used to evaluate its parallel execution-time performance. Figure 6.1 illustrates BER performance for parallel decoding of an 18-byte frame with QPSK modulation using

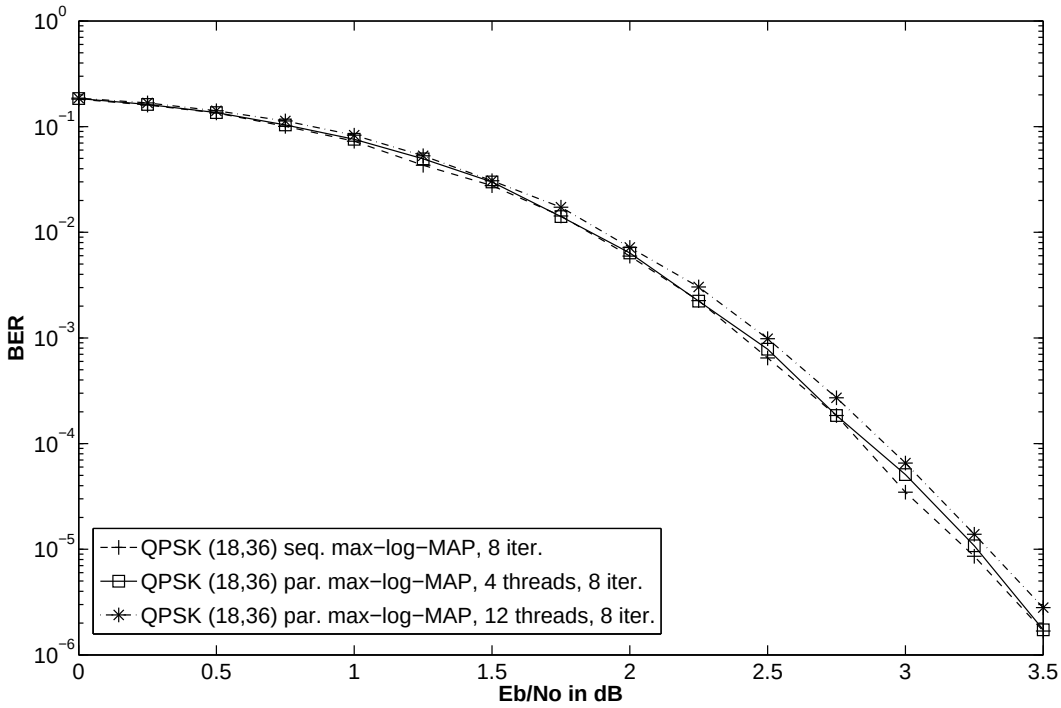


Figure 6.1: BER performance of the parallel turbo decoder.

the max-log-MAP max* variant with eight iterations. The BER degradation as parallelism is increased is not significant, with the maximum degradation using twelve sub-blocks being less than 0.1 dB for a particular BER. The same trend is observed for the parallel linear-log-MAP decoder using four iterations, with a parallel BER graph provided in Appendix C.

### 6.4.2 Execution Performance

This section provides SimpleScalar simulation results of the multithreaded turbo decoder in order to characterize the reduction in execution cycles for parallel decoding. All results presented are for decoding 18-byte frames. A comparison of the multithreaded decoder performance with and without special instructions for the

max-log-MAP and linear-log-MAP decoders was performed. The max-log-MAP case uses 8 iterations and the linear-log-MAP case uses 4 iterations for comparable BER performance, as discussed in Section 4.4. With special instructions, the improvement with a given number of processors for the max-log-MAP decoder was 23% to 29% over the parallel baseline implementation without special instructions. Similarly, the speedup from including special instructions for the linear-log-MAP decoder was 34% to 40%.

Table 1 provides simulation statistics for the multithreaded max-log-MAP decoder using 8 iterations with special instructions. The total execution cycles for decoding a single frame using the algorithm described in Section 5.3 is $T_N*$, as defined in Equation 6.2, which is the sum of serial execution cycles, $T_S$, and parallel execution cycles, $T_P$. $S_N$ is the speedup obtained using $N$ threads over the single-threaded decoder, $f_S$ is the serial fraction of execution time using $N$ threads, calculated as $f_S = T_S/T_N*$, and $E_N$ is the parallel efficiency of the multithreaded decoder, where $E_N = S_N/N$. The parallel efficiency values in Table 6.2, being 80% for 12 threads and at least 91% for 8 or fewer threads, indicate that the parallel decoding algorithm is highly parallel and the performance benefit of multithreaded turbo decoding is significant. The speedup and efficiency values are comparable for other frame sizes. Results for the linear-log-MAP decoder are similar.

Simulations were also conducted using the parallel decoder with the additional optimizations described in Section 5.5. Work that was previously done serially was pushed to threads to be performed in parallel, thus greater speedup results were expected. The extent of the improvement increased as more threads were used for parallel execution.

Table 6.2: Simulation statistics of the parallel max-log-MAP turbo decoder for decoding 18-byte frames (cycles in thousands).

| $N$ | $T_N*$ | $T_S$ | $T_P$ | $f_S$ | $S_N$ | $E_N$ |
|---|---|---|---|---|---|---|
| 1 | 4227 | 4227 | 0 | 1.00 | 1.00 | 1.00 |
| 2 | 2140 | 93 | 2048 | 0.05 | 1.97 | 0.99 |
| 4 | 1119 | 92 | 1027 | 0.08 | 3.78 | 0.94 |
| 8 | 581 | 92 | 489 | 0.19 | 7.27 | 0.91 |
| 12 | 439 | 92 | 346 | 0.27 | 9.64 | 0.80 |

Table 6.3: Simulation statistics of the optimized parallel max-log-MAP turbo decoder for decoding 18-byte frames (cycles in thousands).

| $N$ | $T_N*$ | $T_S$ | $T_P$ | $f_S$ | $S_N$ | $E_N$ |
|---|---|---|---|---|---|---|
| 1 | 4227 | 4227 | 0 | 1.00 | 1.00 | 1.00 |
| 2 | 2144 | 43 | 2101 | 0.02 | 1.97 | 0.99 |
| 4 | 1096 | 42 | 1054 | 0.04 | 3.86 | 0.96 |
| 8 | 545 | 42 | 502 | 0.08 | 7.76 | 0.96 |
| 12 | 398 | 42 | 356 | 0.12 | 10.63 | 0.89 |

Relative to the parallel decoder without the optimizations in Section 5.5, the improvements for two and four threads were 2% or less, but the improvements for eight and twelve threads were 7% and 12%, respectively, using the max-log-MAP decoder.

Results were similar for the linear-log-MAP decoder. Because the serial fraction of execution is much higher for eight and twelve threads (see Table 6.2), they gained the most from the optimization. Table 6.3 shows statistics for the optimized parallel decoder. Speedup and parallel efficiency values have increased for four, eight, and twelve threads, as a result of the optimization, when compared with the values in Table 6.2.

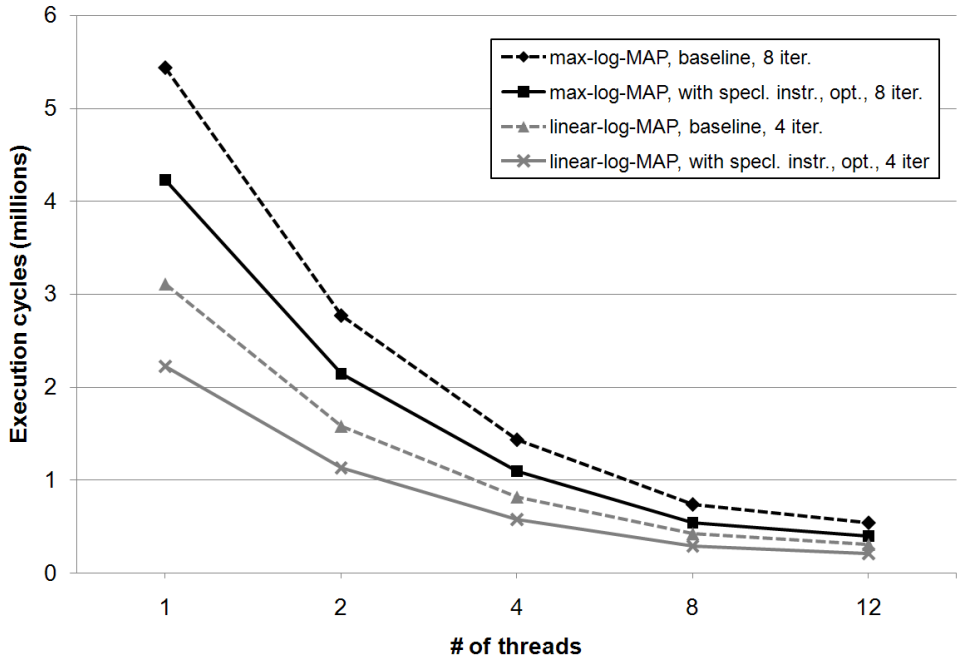


Figure 6.2: Execution cycles comparison between max-log-MAP and linear-log-MAP decoders.

Figure 6.2 shows the reduced execution time for both the optimized parallel max-log-MAP and linear-log-MAP decoders with special instructions over the baseline implementation without special instructions. Speedup values with twelve threads were 10.63 and 10.54 for the max-log-MAP and linear-log-MAP decoders, respectively, over the sequential decoder. The results again demonstrate good parallel efficiency for both optimized decoder implementations with special instructions, being 88% or higher. Because it performs fewer iterations, the linear-log-MAP decoder offers a throughput that is approximately 90% higher than the max-log-MAP decoder, making it the superior choice for our particular software WiMAX turbo decoder. The number of threads used to perform turbo decoding can be chosen to suit the particular target architecture and desired decoder throughput, and the flexibility of the parallel turbo decoding algorithm can allow this number to change dynamically.

## 6.5 Modelling Multiprocessor Caches and Memory

Simulations performed thus far have assumed ideal execution, where one instruction is executed every cycle, and memory, cache, and bus arbitration delays have not been accounted for. Ideal execution still provides a useful indication of parallel performance and the improvement that is obtained with various optimizations and architectural enhancements. It was, however, desirable to see the effect that accounting for these various delays has on the performance of the parallel decoder software, with more accurate simulation and analysis. Section 5.6 discussed the performance impact stemming from the presence of caches and the nature of data sharing in a multiprocessor system.

Results presented in this section use the max-log-MAP parallel decoder software with all optimizations up to this point. First, the cache size and miss rate are compared for the multithreaded turbo decoder. Second, simulations that accurately account for delays were performed, and the execution performance is analyzed.

### 6.5.1 Cache Size and Miss Rate

To evaluate the effect of data cache size on cache miss rate for the parallel decoder software, a version of the multiprocessor SimpleScalar simulator that models one level of direct-mapped caches was used, with processors connected via a shared bus with a single memory [Man01]. The simulator keeps track of tags associated with blocks currently in the data cache, as well as actions related to multiprocessor cache coherence. Cache statistics are collected for the multiprocessor system running the

decoding software, and include all cache accesses and misses, reads from memory, and cache-to-cache transfers. The simulated execution still reflects ideal behaviour, however. Simulations were performed so that prior to decoding, there was no valid data in the cache. As discussed in Section 5.6, cache misses can occur from initial accesses to data, data that was previously in the cache but was subsequently replaced, and invalidated data as a result of data sharing. An analysis of the data shared between threads was also provided in Section 5.6.

Statistics from simulations illustrating the effect of data cache size on cache miss rate for different numbers of threads are illustrated in Figure 6.3, for decoding a QPSK(18,36) frame. The results presented take into account the total number of cache access and misses stemming from each processor. The results are averages of all processors weighted by their proportion of total cache activity, for a given number of threads. Statistics for other frame sizes follow the same trend as for QPSK (18,36) frames, with similar miss rates. In general, miss rates decrease as the cache size is increased. This characteristic is expected, and reflects that fewer cache misses are occurring due to capacity, as a smaller cache size will incur more replacements due to two blocks mapping to the same location. Although there are some anomalies, generally for a given cache size, using more processors results in a lower miss rate, as each processor handles less data as more processors are added.

A more in-depth exploration of the sources of cache misses would explain any anomalies present in the graph. One explanation of the anomalies could be that a particular cache size and number of threads used at a given point caused a higher percentage of data to map to the same location, causing more cache misses due to data being replaced in the cache. For a cache size of 2 kilobytes (KB), for example,

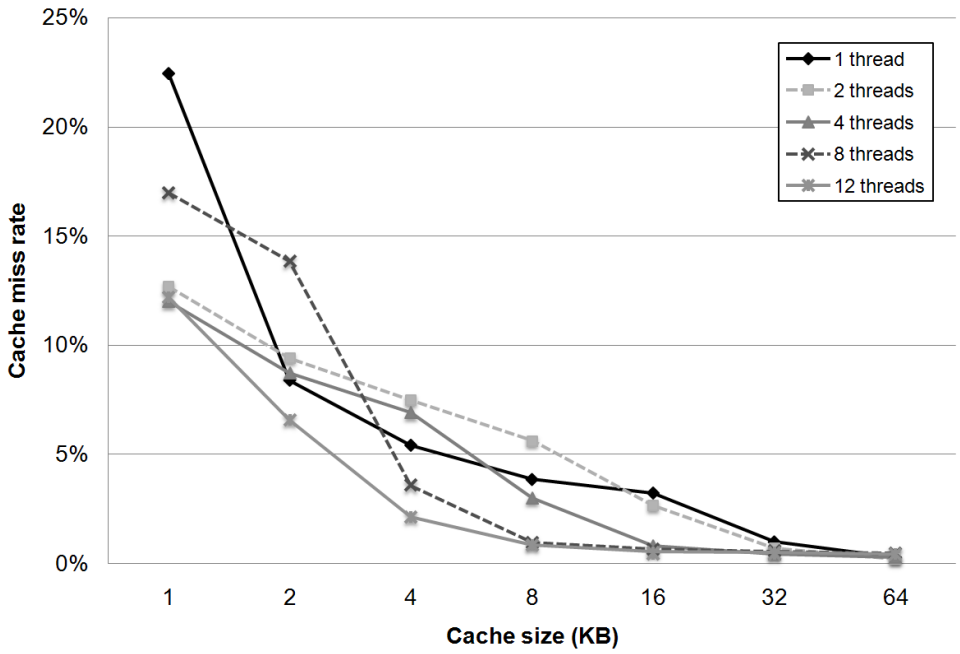


Figure 6.3: Multiprocessor cache miss rate for different cache sizes.

using 8 threads resulted in the highest miss rate. Analyzing the cache statistics revealed significantly more cache requests are made using 8 threads for a 2-KB cache size, suggesting a higher percentage of data that map to the same location in such a small cache. For the different curves, a point is reached where each curve flattens out, meaning an increase in cache size results in a negligible decrease in the cache miss rate. The point before the curve flattens out reflects the minimum desirable cache size for a given number of processors. To achieve miss rates of less than 1%, for 1, 2, 4, 8, and 12 processors, the desirable cache sizes are 64, 32, 16, 8, and 8 KB, respectively.

### 6.5.2 Cache Misses and Execution Performance

Cache misses introduce additional latency in an actual system for parallel execution over the ideal behaviour used in simulations up to this point. Statistics from the ideal multiprocessor cache simulations presented above can be used to analytically predict best-case and worst-case bounds on parallel execution time using $N$ processors, reflecting bus and memory delays associated with cache misses. The cache of processor zero experiences more activity than other processor's caches due its greater workload associated with serial computation. Thus, processor zero's cache misses and write-backs actually determine the bounds on parallel execution time with delays. The best-case performance with caches assumes there is no contention for the bus. For a memory access latency of $M$ cycles, the best-case time required to service each cache miss or write-back is $M$ cycles. Worst-case performance assumes that for every cache miss or write-back, the other $N-1$ processors are contending for the bus with higher priority. Therefore, servicing a cache request from memory requires $(N-1)M+M = NM$ cycles. This worst-case analysis assumes that every cache miss is serviced from memory, when in some cases a cache miss will be serviced by another processor which has a modified copy of the data, via a cache-to-cache transfer over the bus, and in turn will require less than $M$ cycles.

For the analysis in this section, a data cache size of 8 KB was chosen, as this is the desirable cache size for 8 and 12 threads. With fewer threads, this cache size still exhibits good cache performance (i.e. 6% or less miss rate). Table 6.4 shows the cache misses and write-backs for processor zero obtained from the multiprocessor cache simulations. Using a relatively long on-chip memory latency of $M = 20$ cycles, Table 6.4 also shows the ideal, best-case, and worst-case parallel execution times

Table 6.4: Ideal, best-case and worst-case time (in thousands of cycles) when decoding QPSK (18,36) frames.

| **Threads** | **1** | **2** | **4** | **8** | **12** |
|---|---|---|---|---|---|
| **Misses** | 37559 | 19256 | 8628 | 3750 | 3389 |
| **Write-backs** | 17023 | 7814 | 3497 | 1443 | 1189 |
| **Ideal time** | 4227 | 2144 | 1096 | 545 | 398 |
| **Best time** | 5318 | 2685 | 1337 | 648 | 489 |
| **Worst time** | 5318 | 3227 | 2066 | 1375 | 1496 |

determined using the methodology described above. For a small number of threads, e.g. 2, the ideal, best, and worst execution times do not differ by a wide margin. For more threads, e.g. 12, the worst-case time is significantly higher the ideal and best-case times, due to the worst-case scenario assuming that processor zero has the lowest bus arbitration priority, and must wait for all other processors to use the bus.

Additional simulations were then performed using an enhanced version of the SimpleScalar multiprocessor cache simulator that models the delays associated with cache misses, including the delay for reading/writing to/from main memory, and bus contention [Pau07]. Figure 6.4 shows the execution cycles obtained with the enhanced simulator, as well as the ideal, best-case, and worst-case execution performance for comparison. The same memory latency of M = 20 cycles is used in all cases. The best-case and worst-case execution times were calculated using the methodology described above. With the exception of the results for two processors, the execution performance with more accurate simulation is close to that of the best case. For two processors, the best and worst case times are close to each other, thus performance is still good. These findings suggest that with a low cache miss rate, the amount of data sharing and bus activity due to servicing cache requests does not have a significant impact on decoder

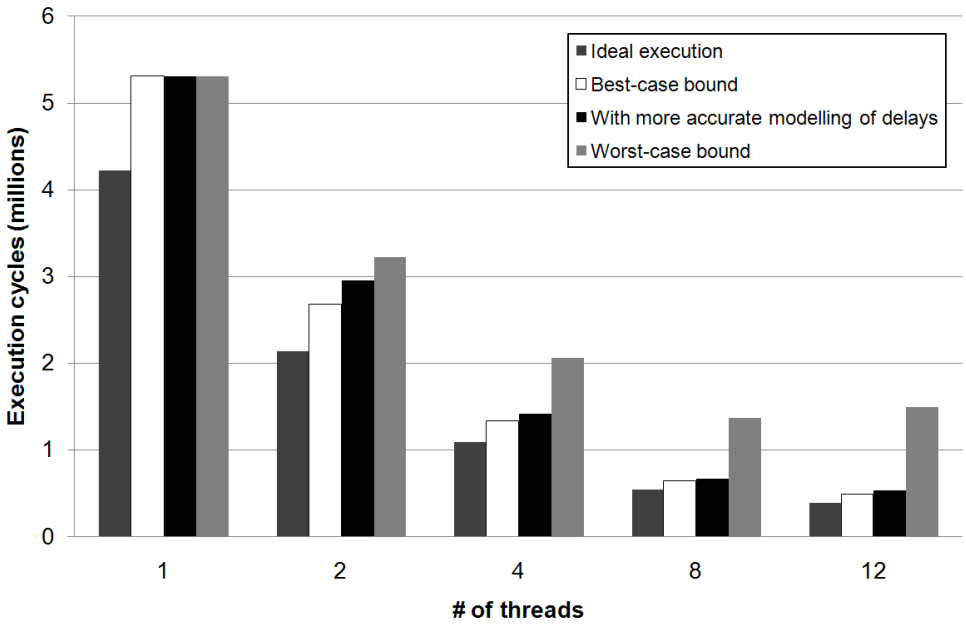


Figure 6.4: Multithreaded execution performance with caches.

performance. From these simulation results with more accurate modelling of delays that would be present in an actual hardware implementation, it can be concluded that a bus-based multiprocessor system with modest cache sizes is a suitable architecture for parallel turbo decoding.

## 6.6 Vectorized Turbo Decoder Initial Results

Vector processing has the potential to speed up parallel execution of the turbo decoder considerably, given the iterative nature of the log-MAP algorithm, as described in Section 5.7. Presented here is an initial investigation into the potential performance benefit that can be obtained through vectorizing the turbo decoder software. The execution results presented here again assume ideal execution for simplicity.

```
/* Calculates gamma values for the sub-block starting at frame index start to
 * index end.  MAX_LLR is the number of LLR values associated with the largest
 * frame size.  NUM_T is the number of threads being used.  MAX_TRELLIS is the
 * number of trellis branches (= 32).  LLR_HEIGHT are the number of LLRs
 * associated with each symbol (= 3).
 */
void calcGamma( float gamma[MAX_LLR/NUM_T][MAX_TRELLIS], int start, int end )
{
    int i, j, k, index;
    for ( i = start; i <= end; i++ ) /* loop over sub-block frame indexes */
    {
        j = i - start; /* gamma array index for sub-block */
        for ( k = 0; k < MAX_TRELLIS; k++ ) /* loop over each trellis branch */
        {
            gamma[j][k] = 0;
            if (trellis_input[k] != 0)
            {
                index = trellis_input[k] + i*LLR_HEIGHT - 1;
                gamma[j][k] += inX[index];
            }
            if (trellis_output[k] != 0)
            {
                index = trellis_output[k] + i*LLR_HEIGHT - 1;
                gamma[j][k] += inZ[index];
            }
        }
    }
}
```

Figure 6.5: C code for calcGamma.

### 6.6.1 Vectorizing the Gamma Calculations

For an initial investigation into the performance improvement that vector processing can achieve, it was desirable to translate a version of the scalar assembly code function responsible for performing $\Gamma$ calculations (`calcGamma` in Figure 4.1) into vector code. Execution-time profiling on a general-purpose PC, as discussed in Section 4.5.5, showed that this function constitutes around 10% of the total execution time required to decode a frame. The C code for `calcGamma` is provided in Figure 6.5. The C code in Figure 6.5 was compiled with optimization level `-O2`, as described earlier for SimpleScalar, with the resulting MIPS-derived portable instruction-set architecture (PISA) assembly code of `calcGamma` shown in Figure 6.6. As a first step in assessing the potential benefit of vectorization, the effect of vectorization in the body of a loop can be approximated directly from this scalar code.

```
Scalar calcGamma:

    move    $11,$5              # [1]       # <$11> = i = start
    slt     $2,$6,$11           # [1]       # <$6> = end, <$2> = (end < start)
    bne     $2,$0,done          # [1]       # if (end < start) then branch to done
    la      $15,trellis_input   # [1]       # <$15> = *trellis_input
    la      $13,inX             # [1]       # <$13> = *inX
    mtc1    $0,$f4              # [1]       # <$f4> = 0
    la      $14,trellis_output  # [1]       # <$14> = *trellis_output
    la      $12,inZ             # [1]       # <$12> = *inZ
    sll     $2,$11,1            # [1]       # <$2> = 2*i
    addu    $10,$2,$11          # [1]       # <$10> = 2*i + i = 3*i = i*LLR_HEIGHT
                                            # **** for ( i = start; i <= end; i++ ) {
outer_loop:                                 # **** j = i - start;
    move    $9,$0               # [72]      # <$9> = 0
    subu    $2,$11,$5           # [72]      # <$2> = j = i - start (initially 0)
    sll     $2,$2,7             # [72]      # <$2> = 128*j = 4*32*j
    addu    $3,$2,$4            # [72]      # <$3> = 128*j + *gamma = gamma[j][k]
    move    $8,$14              # [72]      # <$8> = *trellis_output
    move    $7,$15              # [72]      # <$7> = *trellis_input

inner_loop:                                 # **** for ( k = 0; k < MAX_TRELLIS; k++ ) {
                                            # **** gamma[j][k] = 0;
    sw      $0,0($3)            # [32*72]   # gamma[j][k] = 0
                                            # **** if (trellis_input[k] != 0) {
    lw      $2,0($7)            # [32*72]   # <$2> = trellis_input[k]
    beq     $2,$0,cond2         # [32*72]   # if (trellis_input[k] == 0) branch to cond2
                                            # condition is true only 1/4 of the time
                                            # **** index = trellis_input[k] + i*LLR_HEIGHT - 1;
    addu    $2,$10,$2           # [24*72]   # <$2> = trellis_input[k] + i*LLR_HEIGHT
    subu    $2,$2,1             # [24*72]   # <$2> = trellis_input[k] + i*LLR_HEIGHT - 1
                                            # **** gamma[j][k] += inX[index];
    sll     $2,$2,2             # [24*72]   # <$2> = 4*(trellis_input[k] + i*LLR_HEIGHT - 1)
    addu    $2,$2,$13           # [24*72]   # <$2> = *inX + 4*(trellis_input[k] + i*LLR_HEIGHT - 1)
    l.s     $f0,0($2)           # [24*72]   # <$f0> = inX[index]
    add.s   $f0,$f0,$f4         # [24*72]   # <$f0> = gamma[j][k] + inX[index]
    s.s     $f0,0($3)           # [24*72]   # store gamma[j][k] += inX[index]

cond2:                                      # **** if (trellis_output[k] != 0) {
    lw      $2,0($8)            # [32*72]   # <$2> = trellis_output[k]
    beq     $2,$0,next          # [32*72]   # if (trellis_output[k] == 0) branch to next
                                            # condition is true only 1/4 of the time
                                            # **** index = trellis_output[k] + i*LLR_HEIGHT - 1;
    addu    $2,$10,$2           # [24*72]   # <$2> = trellis_output[k] + i*LLR_HEIGHT
    subu    $2,$2,1             # [24*72]   # <$1> = trellis_output[k] + i*LLR_HEIGHT
                                            # **** gamma[j][k] += inZ[index];
    sll     $2,$2,2             # [24*72]   # <$2> = 4*(trellis_output[k] + i*LLR_HEIGHT)
    addu    $2,$2,$12           # [24*72]   # <$2> = *inZ + 4*(trellis_output[k] + i*LLR_HEIGHT - 1)
    l.s     $f0,0($3)           # [24*72]   # load gamma[j][k]
    l.s     $f2,0($2)           # [24*72]   # load inZ[index]
    add.s   $f0,$f0,$f2         # [24*72]   # gamma[j][k] += inZ[index]
    s.s     $f0,0($3)           # [24*72]   # store gamma[j][k]

next:
    addu    $3,$3,4             # [32*72]   # <$3> = *gamma[j][k] + 4
    addu    $8,$8,4             # [32*72]   # <$8> = *trellis_output[j][k] + 4
    addu    $7,$7,4             # [32*72]   # <$7> = *trellis_input[j][k] + 4
    addu    $9,$9,1             # [32*72]   # <$9> = j += 1
    slt     $2,$9,32            # [32*72]   # <$2> = (j < 32)
    bne     $2,$0,inner_loop    # [32*72]   # if (j < 32) branch to inner_loop
    addu    $10,$10,3           # [72]      # <$10> = (i+1)*LLR_HEIGHT
    addu    $11,$11,1           # [72]      # <$11> = i += 1
    slt     $2,$6,$11           # [72]      # <$2> = (end < i)
    beq     $2,$0,outer_loop    # [72]      # if (end < i) branch to outer_loop

done:
    j           $31             # [1]       # return
```

Figure 6.6: Scalar assembly code for calcGamma.

Table 6.5: Estimated and actual execution cycles and speedup for vectorized Γ calculations.

| | Estimated | | Actual | |
|---|---|---|---|---|
| **VLEN** | Cycles | Speedup | Cycles | Speedup |
| **1** | 51995 | 1.00 | 51418 | 1.00 |
| **2** | 26363 | 1.97 | 26074 | 1.97 |
| **4** | 13547 | 3.84 | 13402 | 3.84 |
| **8** | 7139 | 7.28 | 7066 | 7.28 |
| **16** | 3935 | 13.21 | 3898 | 13.19 |
| **32** | 2333 | 22.29 | 2314 | 22.22 |

The total number of times each instruction is executed is indicated in square brackets beside each instruction in Figure 6.6. The total number of execution cycles required for `calcGamma` during decoding of an 18-byte (N = 72) frame using a single thread is $1\times11+10\times72+11(32\times72)+15(24\times72) = 51995$. The inner `for` loop takes up the majority of execution time with $11(32 \times 72) + 15(24 \times 72) = 51264$ cycles. To initially estimate the potential benefit of vectorizing the `calcGamma` function, the inner `for` loop was incremented by the desired vector length, $L$, to reflect the conversion of scalar memory and arithmetic instructions in the loop body to vector instructions. Using this technique, the estimated total number of instructions executed is reduced to $731 + 51264/L$. Applying this formula for different values of $L$, the estimated execution cycles and the speedup obtained for a single thread is provided in Table 6.5.

A more detailed analysis of actual vector code can also be pursued for comparison with the estimated results from the scalar code. A vectorized version of `calcGamma`

Table 6.6: Vector instructions needed for vectorized Γ calculations.

| | |
|---|---|
| LV $V_t$,*offset*($R_s$) | Load data from memory address $R_s$+*offset* into vector register $V_t$. |
| LV.S $V_{Ft}$,*offset*($R_s$) | Load data from memory address $R_s$+*offset* into floating-point vector register $V_{Ft}$. |
| SV.S $V_{Ft}$,*offset*($R_s$) | Store data from floating-point vector register $V_{Ft}$ to memory address $R_s$+*offset*. |
| VADD $V_d$,$V_s$,$V_t$ | Perform $V_s$ + $V_t$, element-by-element, and place the result in $V_d$. |
| VSUB $V_d$,$V_s$,$V_t$ | Perform $V_s$ - $V_t$, element-by-element, and place the result in $V_d$. |
| VCADD.S $V_{Fd}$,$V_{Fs}$,$V_{Ft}$ | Perform $V_{Fs} + V_{Ft}$, element-by-element, and place the result in $V_{Fd}$ only if corresponding bit is set in $VCR$. |
| VSLL $V_d$,$V_s$,*shamt* | Shift left logical $V_s$ by *shamt*, element-by-element, and place the result in $V_d$. |
| VFSLT $V_s$,$V_t$ | Modifies $VCR$ by doing the comparison $V_s < V_t$, element-by-element (either 0 or 1). |
| VMTC1 $R_t$,$V_{Fs}$ | Move from integer register $R_t$ to vector floating point register $V_{Fs}$. |

was translated from the scalar assembly code, assuming a MIPS-style vector instruction set, similar to the scalar instruction set used by the multiprocessor simulator described in Section 6.1 [Bur97]. A vector condition register, $VCR$, is assumed to exist, and conditional vector instructions are used to execute conditional C statements. The proposed vector instruction set assumes scalar integer registers, denoted by $R$, vector integer registers, denoted by $V$, and vector floating-point registers, denoted by $V_F$. A summary of the instructions required for the vectorized version of `calcGamma` is shown in Table 6.6.

The vectorized assembly code of `calcGamma` is shown in Figure 6.7, with the number of times each line is executed shown in square brackets. $T_i$ is the number

of times the inner loop is executed, and is defined as $T_i = 72(32/L)$. The total execution cycles for the vectorized `calcGamma` function, using a vector length of $L$, is $T = 10 \times 1 + 10 \times 72 + 22T_i = 730 + 50688/L$. The actual execution cycles and speedups of the vectorized code for different $L$ are summarized in Table 6.5. As the results suggest, the approximated analysis through incrementing the inner `for` loop by $L$ in the scalar assembly code provides estimated results quite similar to the actual vectorized assembly code. The speedup values obtained illustrate the benefit of vectorizing inner loops in the decoding software for a vector architecture. The results presented assume that the memory (or cache) width is equal to the vector length, allowing a vector load or store to be ideally performed in only one cycle.

### 6.6.2 Preliminary Execution Results

Having shown the potential of vectorization for a specific portion of the decoding software, an initial investigation into the benefit of vectorizing the entire decoding software was conducted. As the results above suggest, incrementing the inner `for` loops in the scalar code by $L$ can provide results similar to actual vectorized code. The parallel decoder software was modified to increment all vectorizable inner `for` loops by $L$, and simulations were executed using SimpleScalar for different values of $L$ and different numbers of threads. Results are presented in Figure 6.8 for decoding 18-byte frames. As the vector length $L$ is increased, performance improves, albeit with diminishing returns for large values of $L$. The initial results shown in Figure 6.8 give an indication of the reduction in execution cycles that could be obtained with a completely vectorized version of the decoder software running on a vector architecture.

```
Vectorized calcGamma:

    move    $11,$5              # [1]       # <$11> = i = start
    slt     $2,$6,$11           # [1]       # <$6> = end, <$2> = (end < start)
    bne     $2,$0,done          # [1]       # if (end < start) then branch to done
    la      $78,trellis_input   # [1]       # <$15> = *trellis_input
    la      $13,inX             # [1]       # <$13> = *inX
    la      $14,trellis_output  # [1]       # <$14> = *trellis_output
    la      $12,inZ             # [1]       # <$12> = *inZ
    sll     $2,$11,1            # [1]       # <$2> = 2*i
    addu    $10,$2,$11          # [1]       # <$10> = 2*i + i = 3*i = i*LLR_HEIGHT
                                            # **** for ( i = start; i <= end; i++ ) {
outer_loop:                                 # **** j = i - start;
    move    $9,$0               # [72]      # <$9> = 0
    sub     $2,$11,$5           # [72]      # <$2> = j = i - start (initially 0)
    sll     $2,$2,7             # [72]      # <$2> = 128*j = 4*32*j
    add     $3,$2,$4            # [72]      # <$3> = 128*j + *gamma = gamma[j][k]
    move    $8,$14              # [72]      # <$8> = *trellis_output
    move    $7,$15              # [72]      # <$7> = *trellis_input

inner_loop:                                 # **** for ( k = 0; k < MAX_TRELLIS; k++ ) {
                                            # **** gamma[j][k..(k+VLEN-1] = 0;
    vmtc1   $0,$vf0             # [Ti]      # set <$vf4> = gamma[j][k..(k+VLEN-1] = 0
                                            # **** if (trellis_input[k..(k+VLEN-1)] != 0) {
    lv      $v0,0($7)           # [Ti]      # <$v2> = trellis_input[k..(k+VLEN-1)]
    vfslt   $0,$v0              # [Ti]      # VCR[i] = (0 < trellis_input[k..(k+VLEN-1)])
                                            # want VCR[i] set for values not zero
                                            # **** index = trellis_input[k] + i*LLR_HEIGHT - 1;
    vadd    $v0,$v0,$10         # [Ti]      # <$v2> = index[0..(VLEN-1)]
                                            #       = trellis_input[k..(k+VLEN-1)] + i*LLR_HEIGHT]
    vsub    $v0,$v0,1           # [Ti]      # <$v2> = index[0..(VLEN-1)] -= 1
                                            # **** gamma[j][k] += inX[index];
    vsll    $v0,$v0,2           # [Ti]      # <$v2> = index[0..(VLEN-1)] *= 4
    lv.s    $vf1,$v0($13)       # [Ti]      # <$f0> = inX[index[0..(VLEN-1)]
    vcadd.s $vf0,$vf0,$vf1      # [Ti]      # <$f0> = gamma[j][k] += inX[index[0..(VLEN-1)]]
                                            # **** if (trellis_output[k] != 0) {
    lv      $v0,0($8)           # [Ti]      # <$v2> = trellis_output[k..(k+VLEN-1)]
    vfslt   $0,$v0              # [Ti]      # VCR[i] = (0 < trellis_output[k..(k+VLEN-1)])
                                            # want VCR[i] set for values not zero
    vadd    $v0,$v0,$10         # [Ti]      # <$v2> = index[0..(VLEN-1)]
                                            #       = trellis_output[k..(k+VLEN-1)] + i*LLR_HEIGHT]
    vsub    $v0,$v0,1           # [Ti]      # <$v2> = index[0..(VLEN-1)] -= 1
                                            # **** gamma[j][k] += inZ[index];
    vsll    $v0,$v0,2           # [Ti]      # <$v2> = index[0..(VLEN-1)] *= 4
    lv.s    $vf1,$v0($12)       # [Ti]      # <$f0> = inZ[index[0..(VLEN-1)]
    vcadd.s $vf0,$vf0,$vf1      # [Ti]      # <$f0> = gamma[j][k] += inZ[index[0..(VLEN-1)]]
    sv.s        $vf0,0($3)      # [Ti]      # store gamma[j][k]

next:
    addu    $3,$3,4*VLEN        # [Ti]      # <$3> = *gamma[j][k] + 4*VLEN
    addu    $8,$8,4*VLEN        # [Ti]      # <$8> = *trellis_output[j][k] + 4*VLEN
    addu    $7,$7,4*VLEN        # [Ti]      # <$7> = *trellis_input[j][k] + 4*VLEN
    addu    $9,$9,VLEN          # [Ti]      # <$9> = j += VLEN
    slt     $2,$9,32            # [Ti]      # <$2> = (j < 32)
    bne     $2,$0,inner_loop    # [Ti]      # if (j < 32) branch to inner_loop
    addu    $10,$10,3           # [72]      # <$10> = (i+1)*LLR_HEIGHT
    addu    $11,$11,1           # [72]      # <$11> = i += 1
    slt     $2,$6,$11           # [72]      # <$2> = (end < i)
    beq     $2,$0,outer_loop    # [72]      # if (end < i) branch to outer_loop

done:
    j           $31             # [1]       # return
```

Figure 6.7: Vector assembly code for calcGamma.

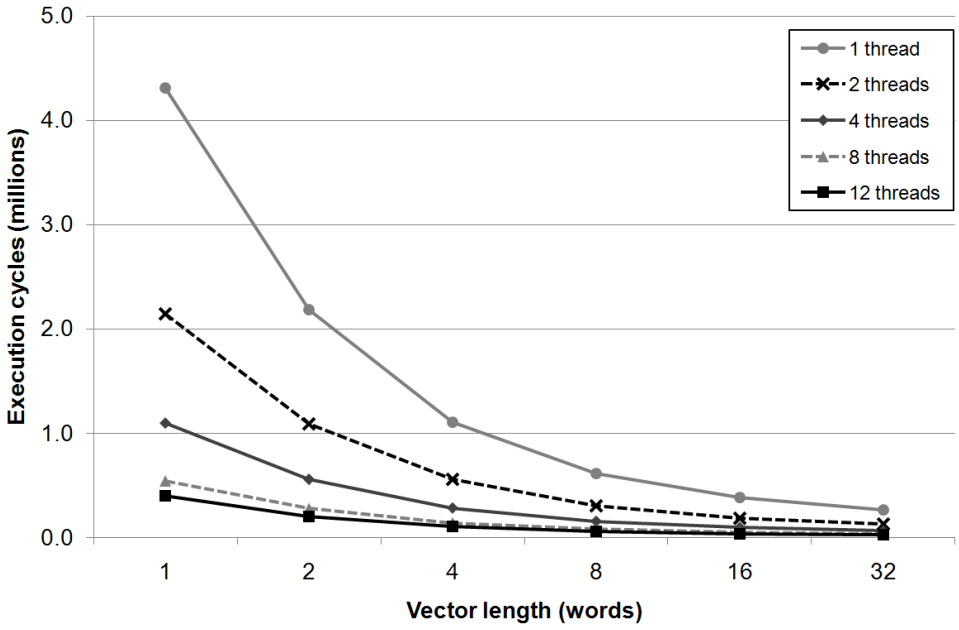


Figure 6.8: Preliminary execution results for vector decoding.

For a more specific assessment, vector speedup and efficiency values of the preliminary vector results are shown in Table 6.7 using four threads. Speedup using a vector length of $L$ is calculated with respect to $L = 1$, and is defined as $S_L = T_1/T_L$. Vector efficiency provides a measure of effectiveness for a certain vector length with regards to the resulting performance improvement, and is calculated as $E_L = S_L/L$. The vectorization speedup and efficiency using fewer threads are slightly higher, and slightly lower using more threads, but otherwise the trends are similar. The ideal vector length is eight, as it still provides good efficiency. The speedup and efficiency values drop off using a vector length of 16 and 32, thus the improvement offered by using a vector length greater than eight would not be worth the additional hardware required. It is not surprising that a vector length of eight is ideal, as the calculation of $A$ and $B$ metrics involves looping over each of the eight states of the trellis for every frame index. Using a larger vector length would mean some of the vector

Table 6.7: Vector speedup and efficiency using 4 threads.

| $L$ | **1** | **2** | **4** | **8** | **16** | **32** |
|---|---|---|---|---|---|---|
| $S_L$ | 1.00 | 1.96 | 3.85 | 6.94 | 11.19 | 16.13 |
| $E_L$ | 1.00 | 0.98 | 0.96 | 0.87 | 0.70 | 0.50 |

units are idle when executing vector instructions, decreasing the resulting speedup and efficiency.

## 6.7 Chapter Summary

This chapter presented simulation results for hardware-oriented enhancements to support the decoding software described in Chapter 5. A custom version of the SimpleScalar simulator was used, which supports multiprocessor execution and cache coherence, to gather execution statistics for analysis of performance. First, the baseline performance of the decoder was established with two decoder implementations: a max-log-MAP version with 8 iterations and a linear-log-MAP version with 4 iterations. Special instruction support was implemented for both the retrieval of trellis information and for performing max* operations. The performance improvement from both instructions for the max-log-MAP and linear-log-MAP decoders was 29% and 40%, respectively. The decoder software was parallelized and its BER and execution performance was examined. BER simulations revealed that the BER degradation due to parallelism was not significant, being at most 0.1 dB. The execution performance of the multithreaded decoder software, after optimizations were performed, revealed good speedup and efficiency for ideal execution, illustrating the high degree of parallelism and the benefit of parallel decoding. Consideration of the delays associated

with caches was subsequently pursued. Simulations showing the effect of data cache size on miss rate revealed that a modest cache size of 8 KB results in a good cache miss rate of less than 6% for configurations using one to twelve threads. An enhanced version of the simulator was then used, that accounts for delays associated with caches, memory, and bus arbitration, to obtain more accurately modelled execution times. Simulations using an 8 KB cache revealed that the more accurately modelled performance of the parallel decoder was close to that of the analytical best-case execution time, illustrating that a bus-based interconnect is a good choice for a turbo decoder system architecture. Finally, both analysis and simulation were performed to investigate the opportunity for reduced execution time available with a vector architecture. Preliminary results suggest that using vector processing with a vector width of eight provides good speedup and efficiency of the parallel decoder software.

# Chapter 7

# Conclusion

This thesis has described the enhancement of WiMAX-capable turbo decoding software through both software and hardware-oriented optimizations. To maximize flexibility, a multiprocessor system-on-chip architecture was targeted, as such a system is expected to become increasingly feasible for SDR-supported mobile devices as transistor sizes shrink. With information distilled from multiple sources, a clear and concise explanation of decoding double-binary turbo codes using the maximum *a posteriori* (MAP) algorithm was presented. The insight gained through investigation of the decoding algorithm aided the adaptation and optimization of the decoding software used in this thesis. The software and hardware enhancements pursued can be applied to any SDR turbo decoder implementation to increase execution performance.

Adapted from open-source turbo decoder software suitable for simulation purposes, a stand-alone C decoder implementation was developed and subsequently optimized. Different max* variants with different accuracies and complexities were evaluated in terms of both execution-time and BER performance. Numerous software enhancements were pursued, which significantly reduced the computational complexity

of the decoding software, without compromising bit-error rate (BER) performance. The software optimizations were implemented in the turbo decoding software and the resulting performance improvement was evaluated using a PC. The most significant software optimization involved reducing the work performed during each half-iteration, which demonstrated speedups of between 154% and 272% over the same decoder software without this optimization. The combined performance improvement of all software optimizations for the different max* variants was 277% for a larger frame size and 454% for a smaller frame size.

It was demonstrated that a more complex max* variant using fewer iterations can achieve the same BER performance of a simpler max* variant. In particular, the more complex linear-log-MAP algorithm using four iterations was shown through simulation to provide similar BER performance to that of the simple max-log-MAP algorithm using eight iterations. Additionally, integer decoder implementations based on the linear-log-MAP and max-log-MAP algorithms were developed, and through simulation were shown to only suffer from minimal BER degradation, while offering a performance advantage over their floating-point implementations.

Following the software optimizations, hardware-oriented enhancements were pursued to aid turbo decoding. These enhancements were implemented and their performance benefit was evaluated using a custom multiprocessor simulator. Special instruction support for frequently performed decoder operations was pursued. The max* operator is used extensively throughout log-MAP decoding, and software profiling indicated that upwards of 50% of the total execution time of the decoder is spent performing max* calculations. Thus, special instructions were implemented to accelerate its calculation for both the max-log-MAP decoder using eight iterations

and the linear-log-MAP decoder using four iterations. Simulations revealed using a special instruction to perform the max* operation for the max-log-MAP and linear-log-MAP decoders resulted in improvements of 19% and 30%, respectively, using a single thread. Trellis information is used extensively when computing the various metrics required during log-MAP decoding, thus speeding up its retrieval via a special instruction was investigated. Simulations showed that the execution performance of the decoding software improved by a further 11% with the inclusion of a specialized trellis instruction, again using a single thread. Comparing the two decoder implementations revealed that the linear-log-MAP decoder using four iterations exhibited 90% higher throughput than the max-log-MAP decoder using eight iterations.

In addition to special instructions, a flexible parallel MAP decoder was designed and implemented in software. Ideal simulation results using two to twelve threads indicated that the parallel decoder implementation exhibited high parallel efficiency, with efficiency values of at least 89% obtained, following optimizations. To more accurately assess the performance of the decoder software on a multiprocessor system, the latencies associated with caches, memory, and bus arbitration were considered. Cache simulation results revealed that only modest cache sizes are required to achieve good cache miss rates (below 1%) on a modelled multiprocessor system with a shared bus and a global memory. An enhanced version of the multiprocessor cache simulator was used, which more accurately models parallel execution, taking into account the cache, memory, and bus arbitration delays. Simulation results using more accurately modelled execution revealed that performance was close to that of the ideal case, using a modest cache size of 8 KB and a relatively long memory latency. The use of vector processing to enhance the performance of the turbo decoding software, which adds an

additional degree of execution parallelism, was investigated. Using an approximated method to mimic the work that would be performed with a vectorized version of the software, results indicated that the iterative nature of turbo decoding can allow it to benefit substantially from execution on a single-instruction multiple-data (SIMD) vector architecture.

Overall, the enhancement of both software and hardware can be pursued to increase turbo decoding performance in an SDR environment. A multiprocessor bus-based architecture was shown through accurate simulation to offer good performance in spite of the data sharing inherent in the implemented parallel MAP algorithm. Parallel execution and vector processing combined, along with custom hardware support, can be exploited to improve the performance of an SDR turbo decoder.

## 7.1 Future Work

Considerations for future work include the continued evolution of the modelled multiprocessor system, and implementing additional decoder functionality in software, including de-puncturing and interleaving/de-interleaving for WiMAX, in addition to implementing support for other turbo code standards. Implementing de-puncturing is performed by re-introducing parity bits that were removed during puncturing by the encoder prior to transmission. Puncturing and de-puncturing are performed from a pre-determined pattern, defined by a particular standard. Thus the decoder software would have to be modified to generate these puncturing patterns and perform de-puncturing, which simply involves re-introducing probabilities in the received sequence. Interleaving is more complex to implement, as discussed in Section 2.3. Generating interleaver addresses on-the-fly is desirable to eliminate the need to store

the interleaver patterns in memory. Generating the interleaver addresses on-the-fly in hardware is not straight-forward, and can result in memory access collisions given the pseudo-random nature of the patterns. Implementing interleaving in software would be equally challenging, but would provide more complete decoder functionality for the designed decoder software.

When considering the evolution of the modelled multiprocessor system, it is important to note that the decoder software was evaluated on a system simulated in software, not actual hardware. Evaluating execution performance on an actual hardware architecture, such as a multiprocessor system implemented in a field-programmable gate array (FPGA), can be pursued to obtain concrete implementation and execution performance results. Throughout evaluation of the hardware enhancements, the multiprocessor system consisted of simple single-issue processor elements and one-instruction-per-cycle execution. Another research direction can involve implementing a multiprocessor system for SDR consisting of more aggressive hardware, either simulated or an actual hardware implementation.

Two directions with respect to the continued enhancement of the modelled multiprocessor system can be pursued. The fine-grained approach would involve using additional simple processors to perform turbo decoding. This direction would require refining the parallel algorithm even further, such as having two processors cooperatively decode each sub-block. Using more processors may also require an enhanced communication network other than a shared bus, and this necessity could be determined through simulation. The coarse-grained approach would involve using processing elements with increased complexity, such as SIMD processors, which initial

results indicated would offer a promising performance improvement. Other potentials for enhanced processing units include superscalar execution, where more than one instruction is issued at a time, achieving instruction-level parallelism. Enhanced memory organization could also be pursued, such as providing each processor with their own local memory in addition to a global memory, and/or direct-memory access (DMA) transfers, which would allow the processor to continue executing while memory operations are performed.

# Bibliography

[And98] J. B. Anderson and S. M. Hladik, "Tailbiting MAP decoders," *IEEE J. Sel. Areas Commun.*, vol. 16, no. 2, pp. 297–302, Feb 1998.

[Baa07] B. Baas, Z. Yu, M. Meeuwsen, O. Sattari, R. Apperson, E. Work, J. Webb, M. Lai, T. Mohsenin, D. Truong, and J. Cheung, "AsAP: A fine-grained many-core platform for DSP applications," *IEEE Micro*, vol. 27, no. 2, pp. 34–45, Mar.-Apr. 2007.

[Ber93] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *Proc. IEEE Int. Conf. Commun. (ICC)*, vol. 2, May 1993, pp. 1064–1070.

[Ber99] C. Berrou and M. Jezequel, "Non-binary convolutional codes for turbo coding," *Electron. Letters*, vol. 35, no. 1, pp. 39–40, Jan. 1999.

[Ber05] C. Berrou, R. Pyndiah, P. Adde, C. Douillard, and R. Le Bidan, "An overview of turbo codes and their applications," in *Proc. European Conf. Wireless Tech.*, Oct. 2005, pp. 1–9.

[Bur97] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[Bur00] E. Buracchini, "The software radio concept," *IEEE Commun. Mag.*, vol. 38, no. 9, pp. 138–143, Sept. 2000.

[Che00] J.-F. Cheng and T. Ottosson, "Linearly approximated log-MAP algorithms for turbo decoding," in *Proc. IEEE Vehicular Technol. Conf. (VTC)*, vol. 3, 2000, pp. 2252–2256.

[Dou00] C. Douillard, M. Jezequel, and C. Berrou, "The turbo code standard for DVB-RCS," in *Int. Symp. Turbo Codes and Related Topics*, vol. 3, Sept. 2000, pp. 535–538.

[Erf94] J. Erfanian, S. Pasupathy, and G. Gulak, "Reduced complexity symbol detectors with parallel structure for ISI channels," *IEEE Trans. Comm.*, vol. 42, no. 234, pp. 1661–1671, Feb/Mar/Apr 1994.

[Erg09] M. Ergen, *Mobile Broadband Including WiMAX and LTE.* Springer USA, 2009.

[Fen00] J. Fenlason and R. Stallman, *GNU gprof: The GNU Profiler*, Free Software Foundation, Inc., 2000, available at http://www.skyfree.org/linux/references/gprof.pdf.

[For73] J. Forney, G.D., "The Viterbi algorithm," *Proc. IEEE*, vol. 61, no. 3, pp. 268–278, March 1973.

[Gro98] W. Gross and P. Gulak, "Simplified MAP algorithm suitable for implementation of turbo decoders," *Electron. Letters*, vol. 34, no. 16, pp. 1577–1578, Aug 1998.

[IEE04] IEEE, *Standard for Local and Metropolitan Area Networks, Part 16: Air Interface for Fixed Broadband Wireless Access Systems*, Std. 802.16-2004, Oct. 2004.

[Inf09] Infineon Technologies, *X-GOLD SDR 20: Programmable Baseband Processor for Multi-Standard Cell Phones Product Brief*, 2009, available at http://www.infineon.com/dgdl/X-GOLD_SDR-2009-pb.pdf.

[Kim08] J.-H. Kim and I.-C. Park, "Double-binary circular turbo decoding based on border metric encoding," *IEEE Trans. Circuits Syst. II*, vol. 55, no. 1, pp. 79–83, Jan. 2008.

[Lin06a] Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, A. Reid, and K. Flautner, "Design and implementation of turbo decoders for software defined radio," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2006, pp. 22–27.

[Lin06b] Y. Lin, H. Lee, M. Who, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A low-power architecture for software radio," in *Proc. Int. Symp. Comput. Arch. (ISCA)*, 0-0 2006, pp. 89–101.

[Lin07] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A high-performance DSP architecture for software-defined radio," *IEEE Micro*, vol. 27, no. 1, pp. 114–123, Jan.-Feb. 2007.

[Lin08] C.-H. Lin, C.-Y. Chen, and A.-Y. Wu, "High-throughput 12-mode CTC decoder for WiMAX standard," in *Proc. IEEE Int. Symp. VLSI Design, Autom. Test (VLSI-DAT)*, April 2008, pp. 216–219.

[Man01] N. Manjikian, "Multiprocessor enhancements of the SimpleScalar tool set," *SIGARCH Comput. Archit. News*, vol. 29, no. 1, pp. 8–15, 2001.

[MC07] S. Myoung-Cheol and I.-C. Park, "SIMD processor-based turbo decoder supporting multiple third-generation wireless standards," *IEEE Trans. VLSI Syst.*, vol. 15, no. 7, pp. 801–810, July 2007.

[Mul00] T. Muldner, *C for Java Programmers*. Addison Wesley Longman, Inc., 2000.

[Nik05] A. Niktash, R. Maestre, and N. Bagherzadeh, "A case study of performing OFDM kernels on a novel reconfigurable DSP architecture," in *Proc. IEEE Military Commun. Conf. (MILCOM)*, vol. 3, Oct. 2005, pp. 1813–1818.

[Pat05] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2005.

[Pau07] R. Pau, *ELEC 871 Project Report*, Dept. of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, April 2007, unpublished documentation of project to enhance multiprocessor cache simulation software.

[Qui98] F. Quintana, R. Espasa, and M. Valero, "An ISA comparison between superscalar and vector processors," in *Proc. Int. Conf. Vector and Parallel Process. (VECPAR)*. Springer-Verlag, 1998, pp. 21–23.

[Ram07] U. Ramacher, "Software-defined radio prospects for multistandard mobile phones," *IEEE Computer*, vol. 40, no. 10, pp. 62–69, Oct. 2007.

[Rob95] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. IEEE Int. Conf. Comm. (ICC)*, vol. 2, Jun 1995, pp. 1009–1013.

[Val01] M. C. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software-defined radios," *Proc. Int. J. Wireless Inf. Netw.*, vol. 8, no. 4, pp. 203–215, Oct. 2001.

[Val08] M. C. Valenti, *A Guided Tour of CML, the Coded Modulation Library*, Iterative Solutions, Feb. 2008, available at http://www.iterativesolutions.com/user/image/cmloverview.pdf.

[Vit98] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Sel. Areas in Commun.*, vol. 16, no. 2, pp. 260–264, Feb. 1998.

[Vog08] T. Vogt and N. Wehn, "A reconfigurable ASIP for convolutional and turbo decoding in an SDR environment," *IEEE Trans. VLSI Syst.*, vol. 16, no. 10, pp. 1309–1320, 2008.

[Zha04] Y. Zhang and K. K. Parhi, "Parallel turbo decoding," in *Proc. Int. Symp. Circuits Syst.*, vol. 2, May 2004, pp. 509–512.

[Zha06] C. Zhan, T. Arslan, A. Erdogan, and S. MacDougall, "An efficient decoder scheme for double binary circular turbo codes," in *Proc. IEEE Int. Conf. Speech Signal Process.*, vol. 4, May 2006, pp. IV–229–232.

# Appendix A

# WiMAX Turbo Code Standard

Table A.1 shows the different data block sizes and the corresponding modulation schemes supported by the WiMAX convolutional turbo code (CTC) standard. $N$ is the data block size in pairs of bits, and is an important parameter for WiMAX CTC decoder implementations. The notation used for the different modulation schemes is $\{modulation\}$-$\{code\ rate\}$.

Table A.1: CTC channel coding block sizes and modulation schemes.

| **Data block size (bytes)** | **N** | **Modulation Schemes** |
|---|---|---|
| 6 | 24 | QPSK-1/2 |
| 9 | 36 | QPSK-3/4 |
| 12 | 48 | QPSK-1/2, 16QAM-1/2 |
| 18 | 72 | QPSK-1/2, QPSK-3/4, 16QAM-3/4, 64QAM-1/2 |
| 24 | 96 | QPSK-1/2, 16QAM-1/2, 64QAM-1/3 |
| 27 | 108 | QPSK-3/4, 64QAM-3/4 |
| 30 | 120 | QPSK-1/2, 64QAM-5/6 |
| 36 | 144 | QPSK-1/2, QPSK-3/4, 16QAM-1/2, 16QAM-3/4, 64QAM-1/2 |
| 45 | 180 | QPSK-3/4 |
| 48 | 192 | QPSK-1/2, 16QAM-1/2, 64QAM-3/4 |
| 54 | 216 | QPSK-1/2, QPSK-3/4, 16QAM-3/4, 64QAM-1/3, 64QAM-3/4 |
| 60 | 240 | QPSK-1/2, 16QAM-1/2, 64QAM-5/6 |

# Appendix B

# Additional Execution Performance Results

Additional decoder execution performance results for the original decoder implementation and after various optimizations were applied, as discussed in Chapter 4, are provided here. Decoding times for each of the max* decoder implementations, along with the associated speedups following the various optimizations, are provided. Table B.1 provides the results for decoding QPSK (18,36) frames, while Table B.2 provides results for decoding QPSK (60,120) frames. The results for the two frame sizes given here are consistent with the results obtained for the QPSK (6,12) and QPSK (36,72) frame sizes, as discussed in Section 4.5.

Table B.1: Execution times (s) and speedups associated with optimizations for decoding QPSK (18,36) frames.

| **Optimization** | **max-log-MAP** | | **constant-log-MAP** | | **linear-log-MAP** | | **log-MAP (tbl)** | | **log-MAP (exact)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup |
| **Original, Unoptim.** | 19.06 | - | 22.54 | - | 25.19 | - | 28.15 | - | 157.72 | - |
| **Elim. Dynam. Mem.** | 16.36 | 1.17 | 19.65 | 1.15 | 21.60 | 1.17 | 24.34 | 1.16 | 143.89 | 1.10 |
| **Trellis Gen. Once** | 15.45 | 1.06 | 18.47 | 1.06 | 20.16 | 1.07 | 23.17 | 1.05 | 144.26 | 1.00 |
| **Half-It. Complexity** | 8.70 | 1.78 | 10.53 | 1.75 | 11.07 | 1.82 | 12.34 | 1.88 | 79.71 | 1.81 |
| **SW, Other Optim.** | 8.11 | 1.07 | 10.00 | 1.05 | 10.79 | 1.03 | 11.65 | 1.06 | 78.09 | 1.02 |
| **max* Called Dir.** | 6.35 | 1.28 | 8.12 | 1.23 | 8.64 | 1.25 | - | - | - | - |
| **max* Inlined** | 6.00 | 1.06 | 7.55 | 1.08 | 7.74 | 1.12 | - | - | - | - |
| **Integer Implem.** | 4.36 | 1.38 | 5.32 | 1.42 | 5.27 | 1.47 | - | - | - | - |

Table B.2: Execution times (s) and speedups associated with optimizations for decoding QPSK (60,120) frames.

| **Optimization** | **max-log-MAP** | | **constant-log-MAP** | | **linear-log-MAP** | | **log-MAP (tbl)** | | **log-MAP (exact)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup | Time | Speedup |
| **Original, Unoptim.** | 53.89 | - | 63.28 | - | 68.40 | - | 77.44 | - | 431.63 | - |
| **Elim. Dynam. Mem.** | 46.03 | 1.17 | 54.13 | 1.17 | 58.05 | 1.18 | 65.12 | 1.19 | 389.41 | 1.11 |
| **Trellis Gen. Once** | 42.91 | 1.07 | 51.58 | 1.05 | 54.85 | 1.06 | 62.25 | 1.05 | 392.90 | 1.00 |
| **Half-It. Complexity** | 28.80 | 1.49 | 34.60 | 1.49 | 37.18 | 1.48 | 41.09 | 1.51 | 263.61 | 1.49 |
| **SW, Other Optim.** | 26.98 | 1.07 | 32.55 | 1.06 | 35.41 | 1.05 | 38.45 | 1.07 | 253.72 | 1.04 |
| **max* Called Dir.** | 20.58 | 1.33 | 26.74 | 1.22 | 27.79 | 1.27 | - | - | - | - |
| **max* Inlined** | 20.42 | 1.01 | 25.22 | 1.06 | 25.78 | 1.08 | - | - | - | - |
| **Integer Implem.** | 15.01 | 1.36 | 17.5 | 1.44 | 17.50 | 1.47 | - | - | - | - |

# Appendix C

# Additional Parallel BER Performance Results

Simulations demonstrating the parallel bit-error rate (BER) performance of the linear-log-MAP decoding using four iterations are shown in Figure C.1. Similar to the parallel max-log-MAP decoder using eight iterations, the degradation due to parallelism is less than 0.1 dB for a particular BER.

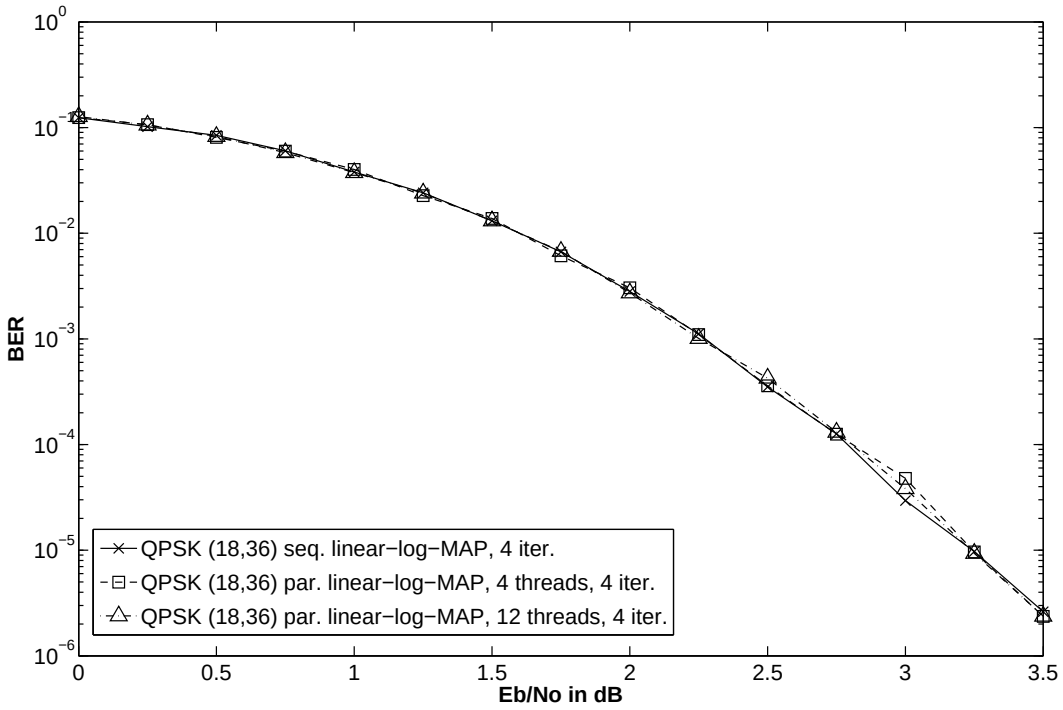


Figure C.1: Parallel BER performance of the linear-log-MAP decoder.