Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Evaluation of the Turbo-decoder Coprocessor on a TMS320C64x Digital Signal Processor

Examensarbete utfört i Kommunikationssystem
vid Tekniska högskolan i Linköping
av

Johan Ahlqvist

LiTH-ISY-EX--11/4522--SE

Linköping 2011

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Evaluation of the Turbo-decoder Coprocessor on a TMS320C64x Digital Signal Processor

Examensarbete utfört i Kommunikationssystem
vid Tekniska högskolan i Linköping
av

Johan Ahlqvist

LiTH-ISY-EX--11/4522--SE

Handledare: **Reza Moosavi**
ISY, Linköpings universitet
Khalid Goyan
Saab AB
Examinator: **Mikael Olofsson**
ISY, Linköpings universitet

Linköping, 19 Oktober, 2011

Presentationsdatum 2011-10-14 **Publiceringsdatum (elektronisk version)** 2011-10-28	**Institution och avdelning** **Institutionen för systemteknik** **Department of Electrical Engineering**

Språk ___ Svenska X Engelska **Antal sidor** 84	**Typ av publikation** ___ Licentiatavhandling X Examensarbete ___ C-uppsats ___ D-uppsats ___ Rapport ___ Annat (ange nedan) ________	**ISBN (licentiatavhandling)** **ISRN** LiTH-ISY-EX--11/4522--SE **Serietitel (licentiatavhandling)** **Serienummer/ISSN (licentiatavhandling)**

URL för elektronisk version
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-71656

Publikationens titel
Evaluation of the Turbo-decoder Coprocessor on a TMS320C64x Digital Signal Processor

Författare
Johan Ahlqvist

Sammanfattning
En teknik som används för att minska de fel som en signal utsätts för vid transmission över en brusig kanal är felrättande kodning. Ett exempel på sådan kodning som ger ett mycket bra resultat är turbokodning. I några digitalsignalprocessorer, av sorten TMS320C64x^{TM}, finns en inbyggd coprocessor som utför turboavkodning.
Denna uppsats är utförd åt Communication Development inom Saab AB och presenterar en utvärdering av denna coprocessor. Utvärderingen avser såväl minnesförbrukning som datatakt och innehåller även en jämförelse med en implementering av turbokodning utan att använda coprocessorn.

One technique that is used to reduce the errors brought upon signals, when transmitted over noisy channels, is error control coding. One type of such coding, which has a good performance, is turbo coding. In some of the TMS320C64x^{TM} digital signal processors there is a built in coprocessor that performs turbo decoding.
This thesis is performed on the account of Communication Developments, within Saab AB and presents an evaluation of this coprocessor. The evaluation deals with both the memory consumption as well as the data rate. The result is also compared to an implementation of turbo coding that does not use the coprocessor.

Nyckelord
Turbo decoding, Turbo-decoder Coprocessor, Digital Signal Processor, C64x

Abstract

One technique that is used to reduce the errors brought upon signals, when transmitted over noisy channels, is error control coding. One type of such coding, which has a good performance, is turbo coding. In some of the TMS320C64x^{TM} digital signal processors there is a built in coprocessor that performs turbo decoding.

This thesis is performed on the account of Communication Developments, within Saab AB and presents an evaluation of this coprocessor. The evaluation deals with both the memory consumption as well as the data rate. The result is also compared to an implementation of turbo coding that does not use the coprocessor.

Sammanfattning

En teknik som används för att minska de fel som en signal utsätts för vid transmission över en brusig kanal är felrättande kodning. Ett exempel på sådan kodning som ger ett mycket bra resultat är turbokodning. I några digitalsignalprocessorer, av sorten TMS320C64x^{TM}, finns en inbyggd coprocessor som utför turboavkodning.

Denna uppsats är utförd åt Communication Development inom Saab AB och presenterar en utvärdering av denna coprocessor. Utvärderingen avser såväl minnesförbrukning som datatakt och innehåller även en jämförelse med en implementering av turbokodning utan att använda coprocessorn.

Acknowledgements

First of all I would like to thank my family and friends for supporting me throughout this thesis. Special gratuity is also given to:

My supervisors *Reza Moosavi*, at Linköpings universitet, and *Khalid Goyan*, at Saab AB, for their highly appreciated guidance and help during the thesis

My examiner *Mikael Olofsson* for giving me the opportunity to perform this thesis

Finally I would like to thank all the inspiring people both at Communication Development – Saab AB and at Communication systems – Linköpings universitet.

Johan Ahlqvist
Stockholm, 2011

Contents

Abbreviation

3GPP	3rd Generation Partnership Project
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AWGN	Additive White Gaussian Noise
BCJR	Bahl, Cocke, Jalinek and Raviv
BPSK	Binary Phase Shift Keying
C64x	TMS320C64x^{TM}
CCS	Code Composer Studio^{TM}
CSL	Chip Support Library
DSP	Digital Signal Processor
EDMA	Enhanced Direct-Memory-Access
IDE	Integrated Development Environment
MAC	Multiple Accumulate
MAP	Maximum a Posteriori
ML	Maximum Likelihood
RAM	Random Access Memory
SISO	Soft Input Soft Output
SNR	Signal to Noise Ratio
SOVA	Soft Output Viterbi Algorithm
TCP	Turbo-decoded Coprocessor
TI	Texas Instruments
VLIW	Very Long Instruction Words

Chapter 1

Introduction

When transmitting data over a noisy channel, errors will arise on the received signal. A technique called error control coding has been shown to reduce these errors. There are different ways of doing this, but they have one factor in common; they add redundancy to the signal at the transmitter.

One of the error control coding techniques with a high performance is the so-called turbo coding, which has two decoding devices, which share information over several runs to improve the overall performance.

One problem that often makes turbo coding unusable in practice is that it has a high computational complexity, making it slow. Therefore, there is a coprocessor, the Turbo-decoder Coprocessor (TCP), built-in in some of the TSM320C64x^{TM} (C64x) digital signal processors (DSPs), which can increase the speed of the decoding.

1.1 Thesis Objectives

The main objective of this thesis is to evaluate the performance of the TCP by comparing it to an implementation using a conventional C-algorithm. The report will give answers to the following questions:

- Is it efficient in terms of memory usage and execution speed to use turbo coding on a C64x DSP without using the TCP?
- How much memory usage and speed could be gained by using the TCP instead of the C-based algorithm, if any?
- Is it practically applicable to use the TCP for turbo decoding on a C64x DSP?

1.2 Procedure

The thesis project can be grouped into the following activities:

- Background studies of error control coding, with special focus on turbo coding
- Background studies of the C64x DSP and especially the TCP
- Implementation of a turbo encoder, a C-based turbo decoder and the TCP
- Evaluation of the implementations
- Comparison between the results
- Improvements of the results
- Analysis of the results

1.3 Thesis Outline

Chapter 2 provides a deeper background of error control coding, the 3rd Generation Partnership Project (3GPP) Turbo Coding Standards and the C64x DSP. To fully understand the upcoming chapters, it is important to read this chapter and get the main ideas. In **Chapter 3** the problem is defined and also a suggestion on how to solve it. **Chapter 4** explains the implementation and **Chapter 5** presents the results of the evaluation. In **Chapter 6** an improvement of the implementation is examined. Finally **Chapter 7** states the conclusions and **Chapter 8** suggests some future work.

Chapter 2

Background

2.1 Error Control Coding

Claude Shannon showed in 1948 that it is possible to reduce the errors brought upon signals, when transmitted over noisy channels, to substantially low levels without decreasing the information data rate, as long as the rate is less than the channel's capacity [1]. One technique that tries to accomplish this is error control coding or channel coding, which has been an efficient way of lowering the impact of noise, interference, fading and other channel impairments for a long time now and it has improved the bit error rate performance of communication systems. [2, Ch.2]

The main idea of error control coding is to introduce redundancy in the transmitted signal, which makes it possible for the receiver to detect and correct the errors from the channel. This is performed by mapping of the information sequence, prior to the transmission, onto an encoded sequence called a codeword. [3]

Nowadays there are many different techniques available and they are often split into two groups: (i) block codes and (ii) convolutional codes [3]. Both of the groups however add redundancy to the information sequence. Turbo codes are based upon convolutional coding. Therefore block codes will not be described in this thesis and instead the focus lies on convolutional coding.

2.1.1 Convolutional Encoding

Convolutional and turbo-like codes have a straightforward implementation and they give a good result in terms of bit error rate, this is the reason why they have revolutionized communication systems [3]. The output from a convolutional encoder depends on the input at that time instant as well as on the input on previous time instants. Therefore the implementation can be interpreted as sending the information bits through a time discrete filter [4].

The information bit sequence is entering the encoder in k serial input streams [4]. The encoding then consists of k different shift registers summed up in n modulo-2 adders giving n output streams, where $n > k$. This gives a rate, which is defined as $R = k/n$. Lastly a multiplexer converts the n output streams into one stream of codewords. The encoders are often referred to as (n,k)-encoders.

The memory m_i of each shift register in the encoder is given by the number of delay elements in each of the k shift registers [2, Ch.11]. The memory of the encoder v is defined as the sum of all the delay elements, and the memory order m is defined as the maximum value of all the m_i:s. The memory is an important factor for convolutional codes and it has been shown that increasing m could get a lower error probability.

An example of a convolutional encoder is given in Figure 2-1. It has one input and two outputs, that is $k = 1$ and $n = 2$. The rate is therefore $R = ½$. There are also two delay elements, which gives $m = v = 2$.

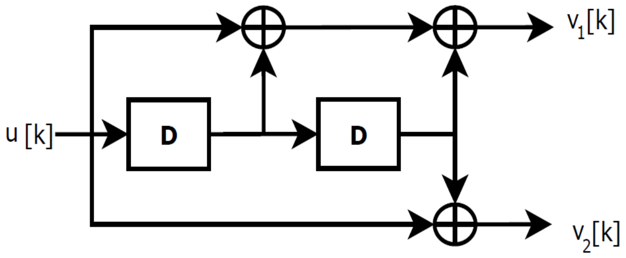

Figure 2-1: A rate ½ non-recursive and non-systematic convolutional encoder.

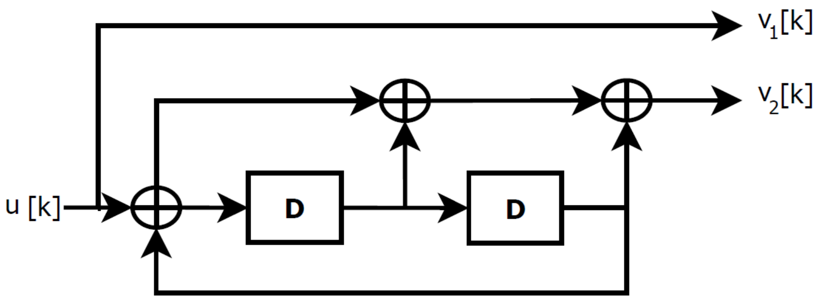

Figure 2-2: A rate ½ recursive and systematic convolutional encoder.

An encoder could be defined as *systematic* or *non-systematic* and *recursive* or *non-recursive* [2, Ch.11]. A systematic encoding means that the first k bits of the output are the same as the k input bits, i.e. the encoder does not affect them. These k bits are often called *systematic bits* and the other $n-k$ bits *parity bits*. The opposite of a systematic encoder is a non-systematic encoder, which means that the input sequence cannot be found in the encoded sequence. An encoder is also called recursive if it uses a feedback and non-recursive or feedforward encoder otherwise. The encoder in Figure 2-1 is an example of a non-recursive and non-systematic encoder. On the other hand, Figure 2-2 illustrates a recursive and systematic encoder, since v_1[k] = u[k] and it consist of a feedback loop.

The operation of a convolutional encoder could be interpreted as a state diagram, since it consists of shift registers which are changed for every new set of k arriving input bits [2, Ch.11]. The state of the encoder could be defined as the content in the shift registers. The total number of states in the state diagram is related to the number of delay elements as: $N = 2^v$.

From the encoder in Figure 2-1, the state diagram shown in Figure 2-3 is given, which has $N = 2^2 = 4$ states. The branches are labelled with u/v_1v_2, where u is the input to the encoder in a certain state making the diagram move via that branch to the next state and outputting v_1 and v_2.

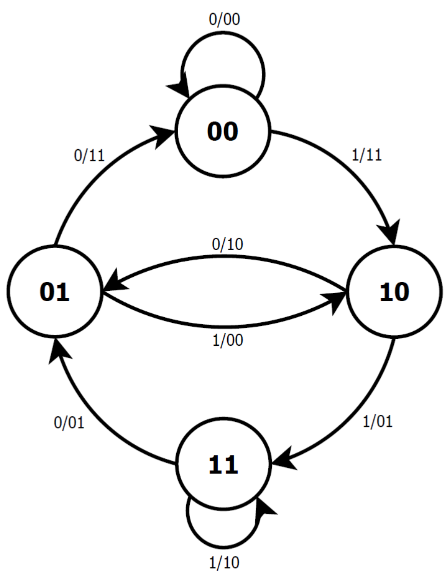

Figure 2-3: The state diagram of the encoder in Figure 2-1.

2.1.1.1 Distance properties

From the state diagram, given that it initially starts in the all-zero state, it is possible to find the codewords from the corresponding information sequence by following the path given by the information sequence and deriving the branch labels.

To find the Hamming weights, i.e. the number of ones in the different codewords, it is proper to modify the state diagram so that it starts and ends in the all-zero state. This is illustrated in Figure 2-4.

Given this modified diagram it is possible to obtain all non-zero codewords from the different paths that start in the all-zero state and diverge from it once, and ends in the all-zero state as they remerge with it [3].

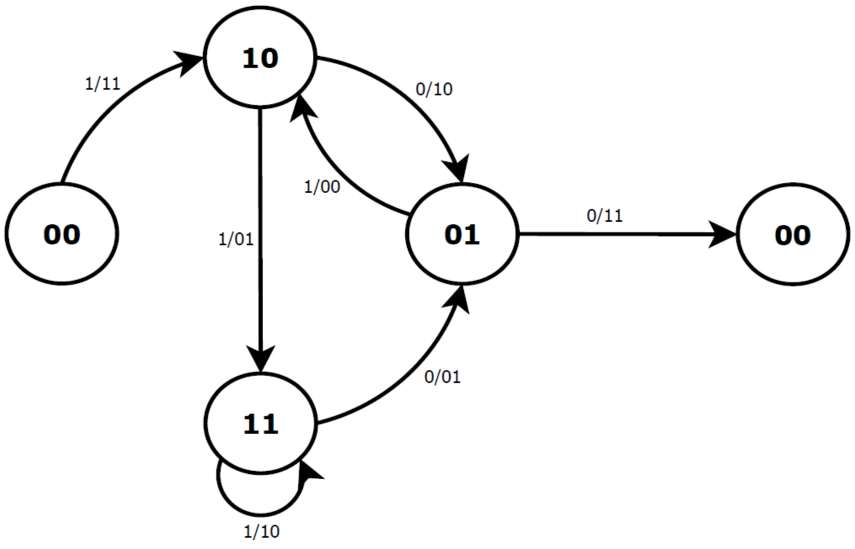

Figure 2-4: a modified state diagram, starting and ending in the all-zero state.

In such a path, the encoder will start outputting ones, and then it is possible to find the minimum weight codeword by finding the path with the fewest ones that takes us through the diagram. The free Hamming distance could now be defined as, since the coding is linear, the non-zero codeword with the minimum Hamming weight [2, Ch.11]. The free Hamming distance is the most important measurement of the performance of the coding and a higher free distance gives a better code. From Figure 2-4, it is possible to follow the path that gives the fewest number of ones. Stepping through the states: 00-10-01-00, with an input sequence of: 100, gives an output sequence of: 111011 which have five ones, thus the free Hamming distance, for this illustrative example, is five.

2.1.2 Convolutional Decoding

Sequential decoding, threshold decoding and Viterbi decoding are some of the different decoding algorithms listed in [4]. The algorithm used in turbo decoding is based on the Bahl, Cocke, Jalinek and Raviv (BCJR) algorithm, which in turn is based on Viterbi-like algorithms; therefore these two algorithms will be described in the following subsections.

2.1.2.1 The Viterbi algorithm

The Viterbi algorithm is said to be an optimum decoding algorithm, given that the input bits are 0 or 1 with equal probability, since it searches through the received sequence and finds the maximum likelihood (ML) sequence [2, Ch.12]. It is therefore called an ML-decoding algorithm. The algorithm finds the most likely codeword in the received sequence and minimizes the probability that the received sequence is not equal to the encoded sequence and thus it will find the most likely codeword.

A good way of understanding the Viterbi algorithm is to start by introducing a trellis diagram [2, Ch.12]. A trellis diagram is an expansion of the state diagram; showing how the different input bits will affect the path between states in time, i.e. now separate state diagrams are shown for each time instant. There will be 2^k branches entering and leaving each state, and each branch could be labelled with the output of the encoder. Using the state diagram in Figure 2-3, the trellis diagram in Figure 2-5 is obtained.

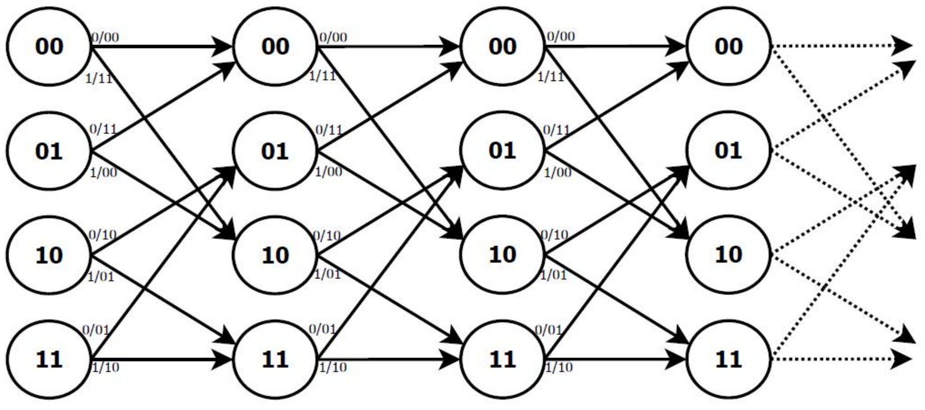

Figure 2-5: a trellis diagram, obtained from the convolutional encoder in Figure 2-1.

The trellis could start and end in any state, but usually the encoding is initiated with zeros in the shift registers forcing the trellis to start in the all-zero state [4]. If m additional zeros is appended at the end of the information sequence, the trellis diagram is also forced to end in the all zero state. This will make the rate of the encoding to increase since a tail of *m* extra bits are sent for each block of bits.

Another way of terminating the encoding, called tail biting, does not append any extra bits [2, Ch.12]. Instead the starting and ending states of the trellis should be the same. This gives the decoder the opportunity to try the different starting and ending states, to find the best choice.

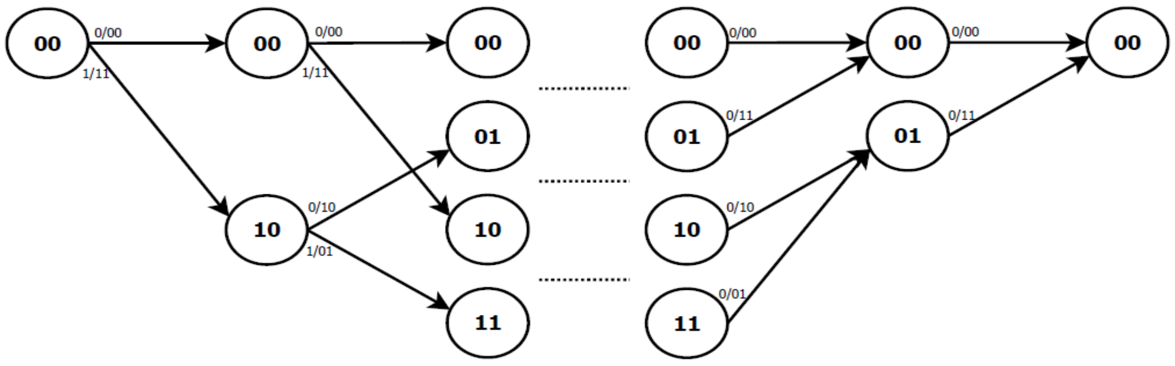

Figure 2-6: A terminated trellis starting in the all-zero state.

The trellis diagram is used in the Viterbi decoding. Under the assumption that the encoder is terminated with some tail bits, the sequence would start and end in the all-zero state [3]. The start and end of such a trellis is shown in Figure 2-6, showing that there are a limited number of paths through the trellis. The purpose of the algorithm is to find the optimum path through the trellis. A useful metric in the decoding is the branch metric, which could be calculated in two different ways, either by using the log-likelihood function to maximize the transition probability or by minimizing the Hamming distance. These two are given as:

$$M_{\mathrm{k,i}} = \ln \mathrm{P}(r_{\mathrm{k}} \mid c_{\mathrm{i}}),$$

$$M_{\mathrm{k,i}} = \mathrm{d}(r_{\mathrm{k}}, c_{\mathrm{i}}).$$

The first metric is the log of the probability that *r* is received given that the codeword *c* was sent and the second is the Hamming distance defined by the number of differing bits between the received *r* and the sent codeword *c*.

It is possible to use the Viterbi algorithm for both hard decoding, i.e. using the signs of the received sequence, as well as soft decoding, i.e. using the log-likelihood values of the received sequence.

Along the way it is possible to discard some of the sequences by noticing that there are several paths entering each state, so if one of these is considered better than the others the rest could be discarded. This means that it is only necessary to keep, as many survivors for each step through the trellis as there are number of states in the diagram.

The following reasoning is adopted and summarized from [2, Ch.12] and [4], for a full description refer to them. Given a received sequence corresponding from an information sequence of length *S*, the Viterbi algorithm could be defined in a couple of steps. Note that depending on which of the branch metric that is used, this is either a maximization problem (log-likelihood function) or a minimization problem (Hamming distance).

Step 1: From the starting time $t = m$, calculate all the branch metrics entering all the states. Keep the branch with the highest/lowest metric as the survivor and save the metric as a state metric.

Step 2: Increase the time t to m+1, calculate all the branch metrics and add them to the state metric from where they originate. Keep the paths with the highest/lowest total metric as the survivor at each state and save the metric as a state metric, discard the rest of the paths.

Step 3: If $t < S + m$, repeat step 2. Otherwise, choose the estimated sequence as the path with the highest/lowest total metric. If the ending state is known, the other states could be discarded.

In practice, to avoid the need of going backwards through the trellis to find the estimated information sequence corresponding to a path, the information sequence and not the surviving path is stored in step 1 and 2.

2.1.2.2 The BCJR Algorithm

The Viterbi decoder is an optimum ML-decoder in the sense that it finds the most likely codeword, but the algorithm is not optimum in terms of bit error rate performance so instead there was a maximum a posteriori (MAP) decoding algorithm introduced in 1974 by Bahl, Cocke, Jalinek and Raviv, called the BCJR algorithm [3]. The BCJR algorithm does not find the most likely codeword; instead it finds the most likely bit in each codeword by estimating the posteriori distribution of the bits, since it is often better to minimize the bit error rate rather than the word error rate [2, Ch.12]. One way of minimizing the bit error rate is to choose an encoder, which maps low-weight information sequences to low-weight codewords. One type of encoder that has this behaviour is a systematic encoder.

The BCJR algorithm, based on the received signal and its predecessors, has as its main task to estimate the posterior probability that a particular branch will be crossed due to the transmitted codeword [3]. And from these posterior probabilities of all branches, it is possible to calculate the posterior probabilities for each bit related to these branches, roughly speaking.

The BCJR algorithm runs Viterbi-like algorithms twice, once forward through the trellis and once backwards. This gives the BCJR algorithm a higher computational complexity than the Viterbi algorithm. With a decoding that runs for several iterations and updating the a priori probabilities of the bits in every run, the BCJR algorithm has a better performance than the Viterbi algorithm.

Definitions

The following definitions are a summary of the reasoning given in both [2, Ch.12] and [3]. For more details refer to them.

The BCJR algorithm is based on hypothesis testing, where a received information bit is either 0 or 1, which gives the log-likelihood function as:

$$\mathrm{L}(u_{\mathrm{i}}) = \ln \frac{\mathrm{P}[u_{\mathrm{i}} = 0 \mid \mathbf{y}]}{\mathrm{P}[u_{\mathrm{i}} = 1 \mid \mathbf{y}]}.$$

Here, $\mathbf{y}$ is the received signal and u_{i} is the information bit at time instant i. From this it is possible to make a “hard” MAP-decision as:

$$\hat{u}_{\mathrm{i}} = \begin{cases} 0, & \mathrm{L}(u_{\mathrm{i}}) > 0, \\ 1, & \mathrm{L}(u_{\mathrm{i}}) < 0. \end{cases}$$

A specific value of u_{i} corresponds to crossings of specific branches in the trellis and the posterior probability of a branch is proportional to the joint probability. From this the log-likelihood function could be expressed as:

$$\mathrm{L}(u_{\mathrm{i}}) = \ln\left(\frac{\sum_{(s_{\mathrm{i}},s_{\mathrm{i+1}})\in U_0} \mathrm{P}(s_{\mathrm{i}} = s', s_{\mathrm{i+1}} = s, \mathbf{y})}{\sum_{(s_{\mathrm{i}},s_{\mathrm{i+1}})\in U_1} \mathrm{P}(s_{\mathrm{i}} = s', s_{\mathrm{i+1}} = s, \mathbf{y})} \right),$$

where U_0 corresponds to the branches in the trellis where u_{i} is zero, U_1 to the branches in the trellis where u_{i} is one and s_{i} and $s_{\mathrm{i+1}}$ are states in the two following steps in the trellis. Given that the received signal **y** consists of S different received codewords as:

$$\mathbf{y} = [y_1, y_2, \cdots, y_S] = y_{1\to S},$$

and using the chain rule for joint probabilities:

$$\mathrm{P}(x_1, x_2, \cdots, x_{\mathrm{n}}) = \mathrm{P}(x_{\mathrm{n}} \mid x_1, x_2, \cdots, x_{\mathrm{n}-1}) \cdot \mathrm{P}(x_1, x_2, \cdots, x_{\mathrm{n}-1}),$$

the following expression could be derived:

$$\begin{aligned} &\mathrm{P}(s', s, \mathbf{y}) = \mathrm{P}(s_{\mathrm{i}} = s', s_{\mathrm{i+1}} = s, y_{1\to S}) = \\ &= \mathrm{P}(y_{\mathrm{i+1}\to S} \mid s_{\mathrm{i}} = s', s_{\mathrm{i+1}} = s, y_{1\to \mathrm{i}})\, \mathrm{P}(s_{\mathrm{i+1}} = s, y_{\mathrm{i}} \mid s_{\mathrm{i}} = s', y_{1\to \mathrm{i}-1})\, \mathrm{P}(s_{\mathrm{i}} = s', y_{1\to \mathrm{i}-1}). \end{aligned}$$

Then by using the fact that the channel is memoryless for simplicity, i.e. the received value at time t is independent of the received values from previous times ($< t$), it is possible to simplify the expression to:

$$\begin{aligned} &\mathrm{P}(y_{\mathrm{i+1}\to S} \mid s_{\mathrm{i}} = s', s_{\mathrm{i+1}} = s, y_{1\to \mathrm{i}})\, \mathrm{P}(s_{\mathrm{i+1}} = s, y_{\mathrm{i}} \mid s_{\mathrm{i}} = s', y_{1\to \mathrm{i}-1})\, \mathrm{P}(s_{\mathrm{i}} = s', y_{1\to \mathrm{i}-1}) \\ &= \mathrm{P}(y_{\mathrm{i+1}\to S} \mid s_{\mathrm{i+1}} = s)\, \mathrm{P}(s_{\mathrm{i+1}} = s, y_{\mathrm{i}} \mid s_{\mathrm{i}} = s')\, \mathrm{P}(s_{\mathrm{i}} = s', y_{1\to \mathrm{i}-1}). \end{aligned}$$

The following quantities are now defined:

$$\beta_{\mathrm{i}}(\mathrm{s}) = \mathrm{P}(y_{\mathrm{i+1}\to S} \mid s_{\mathrm{i+1}} = s),$$

$$\gamma_{\mathrm{i}}(s', s) = \mathrm{P}(s_{\mathrm{i+1}} = s, y_{\mathrm{i}} \mid s_{\mathrm{i}} = s'),$$

$$\alpha_{\mathrm{i}-1}(s') = \mathrm{P}(s_{\mathrm{i}} = s', y_{1\to \mathrm{i}-1}).$$

It is possible to rewrite the expressions for α and β as:

$$\alpha_{\mathrm{i}}(\mathrm{s}) = \sum_{s'} \gamma_{\mathrm{i}}(s', s)\alpha_{\mathrm{i}-1}(s'),$$

$$\beta_{\mathrm{i}-1}(s') = \sum_{s} \gamma_{\mathrm{i}}(s', s)\beta_{\mathrm{i}}(s).$$

α is called the forward metric since running through the trellis forward could derive it and β is called the backward metric since running through the trellis backwards could derive it. Using these new notations, P(s’, s, **y**) could now be expressed as:

$$\mathrm{P}(s', s, \mathbf{y}) = \beta_{\mathrm{i}}(s)\gamma_{\mathrm{i}}(s', s)\alpha_{\mathrm{i}-1}(s').$$

Left to be calculated is all γ_{i}, called the branch metrics. A rewriting could be done like:

$$\gamma_\mathrm{i}(s',s) = \mathrm{P}(s_{\mathrm{i}+1} = s, y_\mathrm{i} \mid s_\mathrm{i} = s') = \mathrm{P}(s_{\mathrm{i}+1} = s \mid s_\mathrm{i} = s')\mathrm{P}(y_\mathrm{i} \mid s_{\mathrm{i}+1} = s, s_\mathrm{i} = s').$$

The first probability of the right hand side is the a priori probability that a state *s*' is traversed to *s* in the trellis. The second probability on the right hand side is related to the modulation and demodulation used, as well as the channel model, which is the same probability calculated for the branch metrics in the Viterbi algorithm.

The **max* operation** could be useful in the BCJR algorithm. The operation is defined for real numbers x_1, x_{2},…, x_n as:

$$\max{}^*(\mathrm{x}_1, \mathrm{x}_2, \cdots, \mathrm{x}_\mathrm{n}) \equiv \ln(\mathrm{e}^{\mathrm{x}_1} + \mathrm{e}^{\mathrm{x}_2} + \cdots + \mathrm{e}^{\mathrm{x}_\mathrm{n}}).$$

The operation has the property of associativity:

$$\max{}^*(\mathrm{x},\mathrm{y},\mathrm{z}) = \max{}^*(\max{}^*(\mathrm{x},\mathrm{y}),\mathrm{z}).$$

With proof:

$$\max{}^*(\mathrm{x},\mathrm{y},\mathrm{z}) = \ln(\mathrm{e}^\mathrm{x} + \mathrm{e}^\mathrm{y} + \mathrm{e}^\mathrm{z}) = \ln(\mathrm{e}^{\ln(\mathrm{e}^\mathrm{x}+\mathrm{e}^\mathrm{y})} + \mathrm{e}^\mathrm{z}) = \max{}^*(\max{}^*(\mathrm{x},\mathrm{y}),\mathrm{z}).$$

For two arguments the operation could be expressed as:

$$\max{}^*(\mathrm{x},\mathrm{y}) = \ln(\mathrm{e}^\mathrm{x} + \mathrm{e}^\mathrm{y}) = \max(\mathrm{x},\mathrm{y}) + \ln(1 + \mathrm{e}^{-|\mathrm{x}-\mathrm{y}|}),$$

which is proved in [3]. In this expression the second part of the sum ranges between 0 and ln 2 and could therefore be computed via a look-up table, which would make it more effective to compute than the original exponential expression.

The Algorithm

The BCJR algorithm, like the Viterbi algorithm, can be described in a couple of steps. This is summarized from [2, Ch.12] and [3]. In practice, this is often done in a slightly modified log-domain algorithm, called log-MAP, to get more effective computations:

Step 1: Initialization of the forward and backward metrics. Assuming that the initial and the terminal states are known, these are initialized according to:

$$a_0(s) \equiv \ln(\alpha_0(s)) = \begin{cases} 0, & s = 0, \\ -\infty, & s \neq 0, \end{cases}$$

$$b_S(s) \equiv \ln(\beta_S(s)) = \begin{cases} 0, & s = 0, \\ -\infty, & s \neq 0. \end{cases}$$

Step 2: The branch metrics are calculated as:

$$g_\mathrm{i}(s',s) \equiv \ln(\gamma_\mathrm{i}(s',s)).$$

Step 3: The forward metrics are calculated as:

$$a_\mathrm{i}(s) \equiv \ln(\alpha_\mathrm{i}(s)) = \ln\sum_{s'}\gamma_\mathrm{i}(s',s)\alpha_{\mathrm{i}-1}(s') = \ln\sum_{s'}\mathrm{e}^{g_\mathrm{i}(s',s)+a_{\mathrm{i}-1}(s')} = \max_{s'}{}^*(a_{\mathrm{i}-1}(s') + g_\mathrm{i}(s',s)).$$

Step 4: The backward metrics are calculated as:

$$b_{\mathrm{i}-1}(s') \equiv \ln(\beta_{\mathrm{i}-1}(s')) = \ln\sum_{s}\gamma_\mathrm{i}(s',s)\beta_\mathrm{i}(s) = \ln\sum_{s}\mathrm{e}^{g_\mathrm{i}(s',s)+b_\mathrm{i}(s)} = \max_{s}{}^*(b_\mathrm{i}(s) + g_\mathrm{i}(s',s)).$$

Step 5: The log-likelihood functions are calculated for all k. With the definitions of *a*, *b* and *g*, it is now possible to express $\mathrm{P}(s', s, \mathbf{y})$ as:

$$\mathrm{P}(s', s, \mathbf{y}) = \mathrm{e}^{b_{\mathrm{i}}(s) + g_{\mathrm{i}}(s', s) + a_{\mathrm{i}-1}(s')}.$$

And the log-likelihood function as:

$$\mathrm{L}(u_{\mathrm{k}}) = \ln\left\{ \sum_{(s_{\mathrm{i}}, s_{\mathrm{i}+1}) \in U_0} \mathrm{e}^{b_{\mathrm{i}}(s) + g_{\mathrm{i}}(s', s) + a_{\mathrm{i}-1}(s')} \right\} - \ln\left\{ \sum_{(s_{\mathrm{i}}, s_{\mathrm{i}+1}) \in U_1} \mathrm{e}^{b_{\mathrm{i}}(s) + g_{\mathrm{i}}(s', s) + a_{\mathrm{i}-1}(s')} \right\} =$$
$$= \max^{*}_{(s_{\mathrm{i}}, s_{\mathrm{i}+1}) \in U_0} \left[b_{\mathrm{i}}(s) + g_{\mathrm{i}}(s', s) + a_{\mathrm{i}-1}(s') \right] - \max^{*}_{(s_{\mathrm{i}}, s_{\mathrm{i}+1}) \in U_1} \left[b_{\mathrm{i}}(s) + g_{\mathrm{i}}(s', s) + a_{\mathrm{i}-1}(s') \right].$$

Step 6: Finally the “hard” decisions could (optionally) be calculated using:

$$\hat{u}_{\mathrm{i}} = \begin{cases} 0, \mathrm{L}(u_{\mathrm{i}}) > 0, \\ 1, \mathrm{L}(u_{\mathrm{i}}) < 0. \end{cases}$$

Modification

According to [2, Ch.12], it is possible to obtain a simpler algorithm if the max* operation is approximated as:

$$\max{}^{*}(x, y) \approx \max(x, y).$$

This is a good approximation if $\max(x, y)$ is relatively large, since $\ln(1 + \mathrm{e}^{-|x-y|})$ is bounded by 0 and $\ln(2) \approx 0.693$. Using this approximation gives an algorithm called the Max-log-MAP algorithm.

2.1.3 Turbo Coding

Turbo coding is an error control coding with a performance that can get very close to Shannon’s theoretical limit [3]. Its iterative decoding, where two decoding devises share information between each other over several rounds, was a breakthrough for error control coding. The two fundamental ideas of turbo coding are the creation of a random-like code and iterative decoding.

2.1.3.1 Encoding

The encoding is done using two convolutional encoders, one is used to encode the information sequence and the other one is used to encode an interleaved version of the information sequence. It has been shown that at high signal-to-noise ratio’s (SNR's), the bit-error rate achieved when using non-systematic encoders is lower than that with systematic encoders while at low SNR's it is opposite [5]. Also, it has been shown that high rate turbo codes obtained by using recursive and systematic convolutional encoders perform better than the turbo codes obtained from non-systematic encoders at any operating SNR.

Often both encoders are equal, this is more for convenience than performance and the best codes often have a relatively short memory length, typically four or less [2, Ch.16]. After the two encoders, a puncturing scheme could be introduced to get a higher code rate, which means that some of the redundant outputs are deleted according to a predefined pattern.

An example of a turbo encoder is shown in Figure 2-7. The two identical encoders are recursive and systematic encoders with four memory cells each.

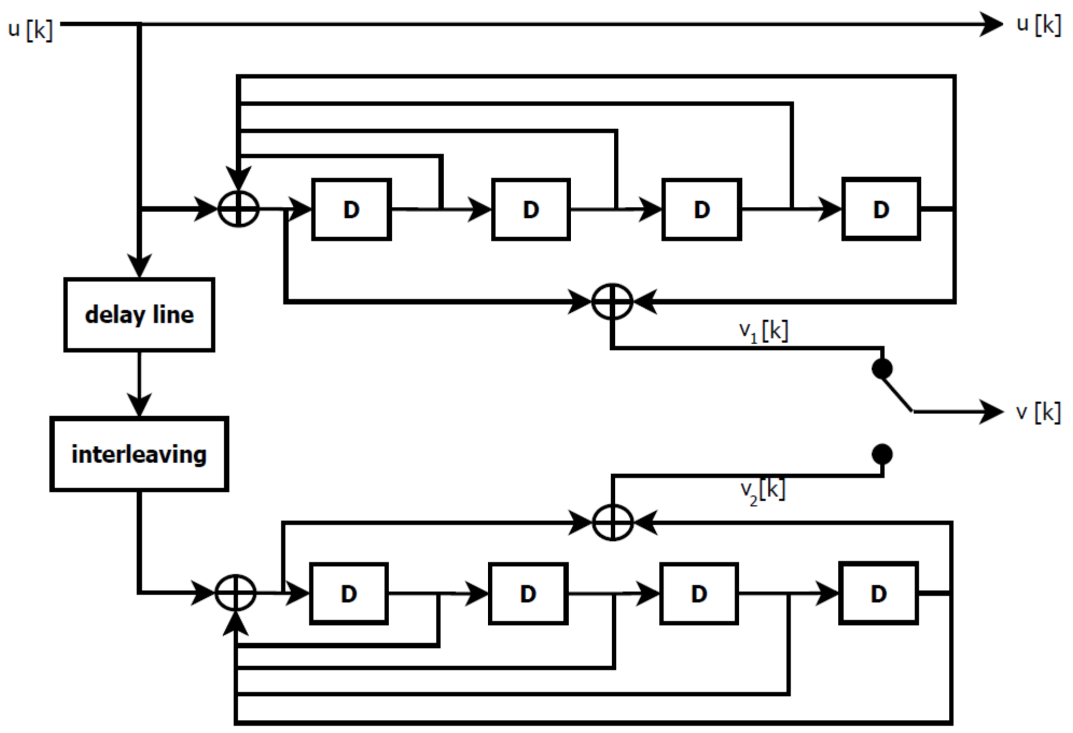

Figure 2-7: Recursive Systematic codes with parallel concatenation. This figure is produced freely from [5, Fig. 2].

2.1.3.2 Interleaving

The input bits are written to a square matrix row-wise and then read pseudo-randomly column-wise, rearranging the bits in an irregular but pre-described way [5]. These long interleavers are what give the code a random-like structure, which has been proposed to gain capacity. To get a good performance the interleaving depth should be large, often a couple of thousand bits [2, Ch.16].

One important observation is that even though the first encoder, encoding the non-interleaved sequence, is terminated, the second encoder, due to the interleaving, might not be terminated just by adding zeros [2, Ch.16]. This could however be modified if desired.

2.1.3.3 Decoding

It has been shown that soft decoding is better than hard decoding, and by using the BCJR algorithm, the turbo decoding could calculate soft a posteriori probability of each bit based on the received signal [3]. The received signal form the first encoder of the non-interleaved information sequence is MAP decoded and the log-likelihood ratio outputs are used as a priori, soft inputs, to the second decoder, using an interleaver in between.

It is then possible to improve the result by using several iterations via a feedback loop, i.e. the second decoder outputs log-likelihood ratios, which are deinterleaved and then fed back to the first decoder [2, Ch.16]. The loop continues for either a predefined number of iterations or until a satisfactory convergence has been reached. Figure 2-8 shows the improvement when increasing the number of iterations in the decoding. Herein the bit error rates of the different number of iterations have been plotted and compared to each other at different SNRs. The encoder is the same as the ½-rate code in Figure 2-7 and the channel is an Additive White Gaussian Noise (AWGN) channel.

When a puncturing scheme is used, the decoder considers the punctured bits as erasures, i.e. it considers them to be zero or one with equal probability [3].

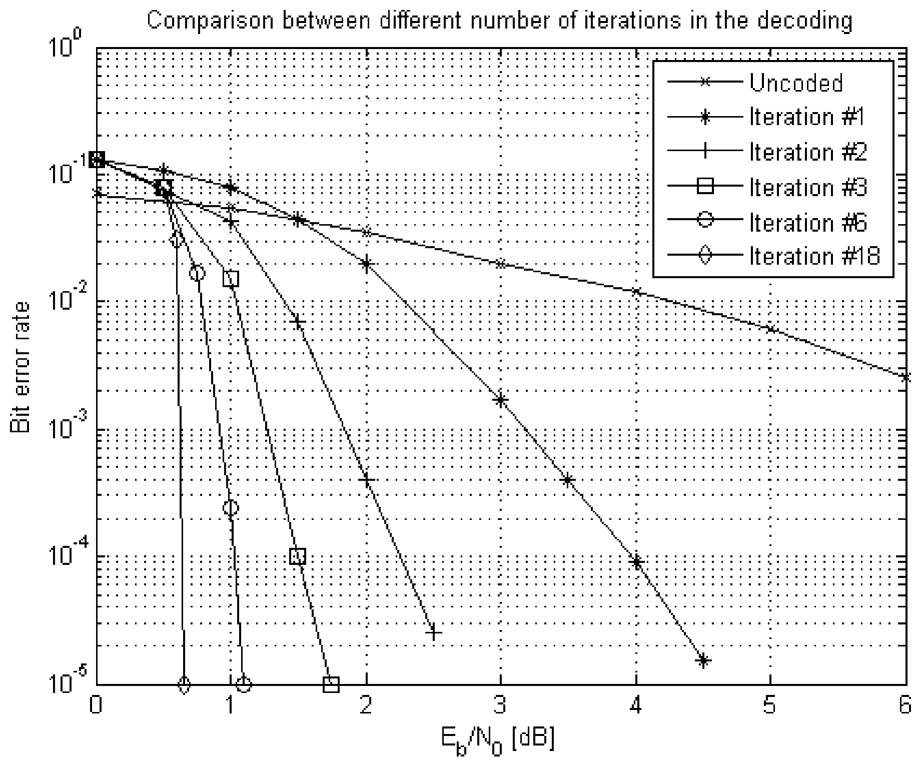

Figure 2-8: Binary error rate given by iterative decoding (p=1,…, 18) of code of fig. 2 (Figure 2-7) (rate:1/2); interleaving 256x256. This figure is produced freely from [5, Fig. 5].

According to section 2.1.2, there are different decoding methods that can be used for turbo decoding. In the following section a comparison between four different decoding algorithms will be presented. The results have been taken from [6], where two rate $R = ½$, $m = 4$ encoders has been used to create one rate $R = ⅓$ turbo encoder. The encoded bits where modulated using Binary Phase Shift Keying (BPSK) and transmitted through an AWGN channel. The four soft input soft output (SISO) decoding algorithms compared are the soft output Viterbi algorithm (SOVA), the Max-Log-MAP, the Log-MAP and the MAP algorithm. The different algorithms could be implemented as depicted in Figure 2-9, with two decoders, using one of the four decoding algorithms, sharing information with each other. The performance and the computational complexity comparison between the different algorithms are given in Figure 2-10 and 2-11. Here it has been shown that it is possible to gain a lot of time complexity by using one of the simplified algorithms at the cost of higher bit error rate.

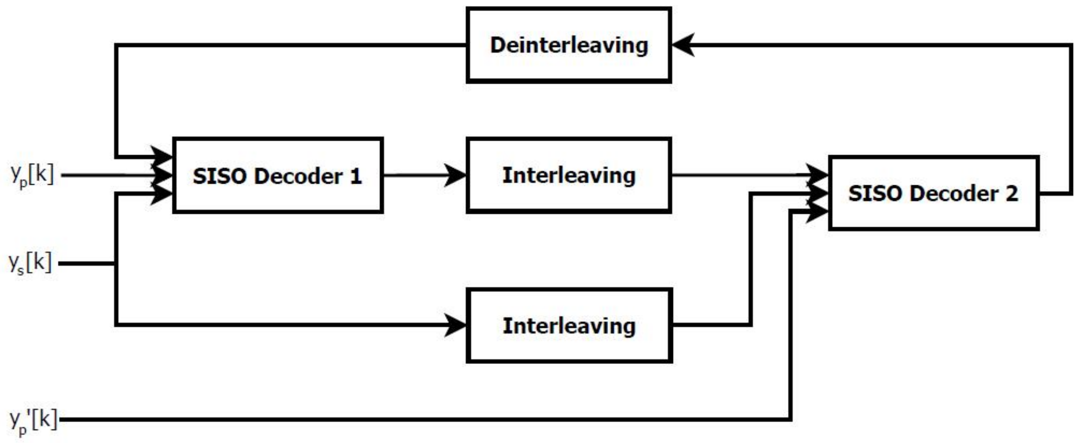

Figure 2-9: Block diagram of iterative (turbo) decoder. This figure is produced freely from [6, Fig. 2].

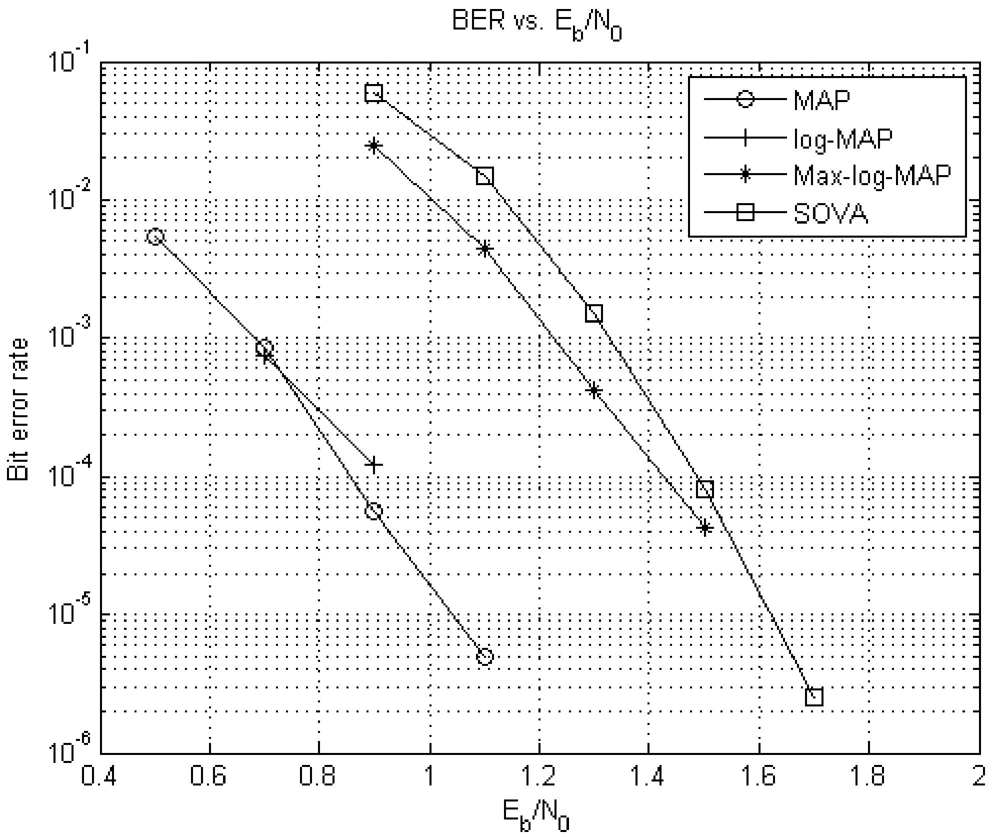

Figure 2-10: Turbo decoding performance comparison among different SISO algorithms. This figure is produced freely from [6, Fig. 4].

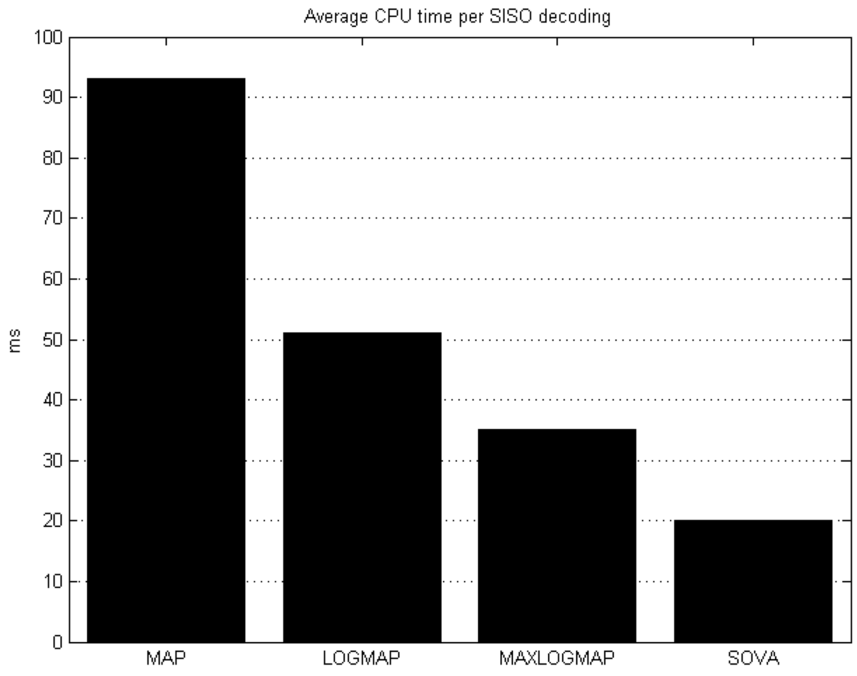

Figure 2-11: Execution time comparison of different SISO algorithms. This figure is produced freely from [6, Fig. 5].

2.2 3GPP Turbo Coding Standards

The 3GPP is a project involving six different communication bodies [7]. The project started with the production of technical specifications and reports regarding the 3G mobile systems. The different bodies in the project are:

- **ARIB –** The Association of Radio Industries and Businesses, Japan
- **ATIS –** The Alliance for Telecommunications Industry Solutions, USA
- **CCSA –** China Communications Standards Association
- **ETSI –** The European Telecommunications Standards Institute
- **TTA –** Telecommunications Technology Association, Korea
- **TTC –** Telecommunication Technology Committee, Japan

By now the 3GPP has produced a couple of standards for wireless transmission and in the following subsections the turbo coding standards is presented.

2.2.1 Encoding

This section is summarized from [8] which define the encoding of the 3GPP turbo coding.

The encoding of the 3GPP turbo coding is done in blocks of K bits, where $40 \le K \le 5114$. It consists of two identical recursive and systematic convolutional encoders with the rate ½. The first one encodes the input sequence and outputs X and A, where X is equal to the input sequence. The other encoder encodes an interleaved version of the input sequence and only outputs A', since the other output X' already exists by interleaving X. This is illustrated in Figure 2-12.

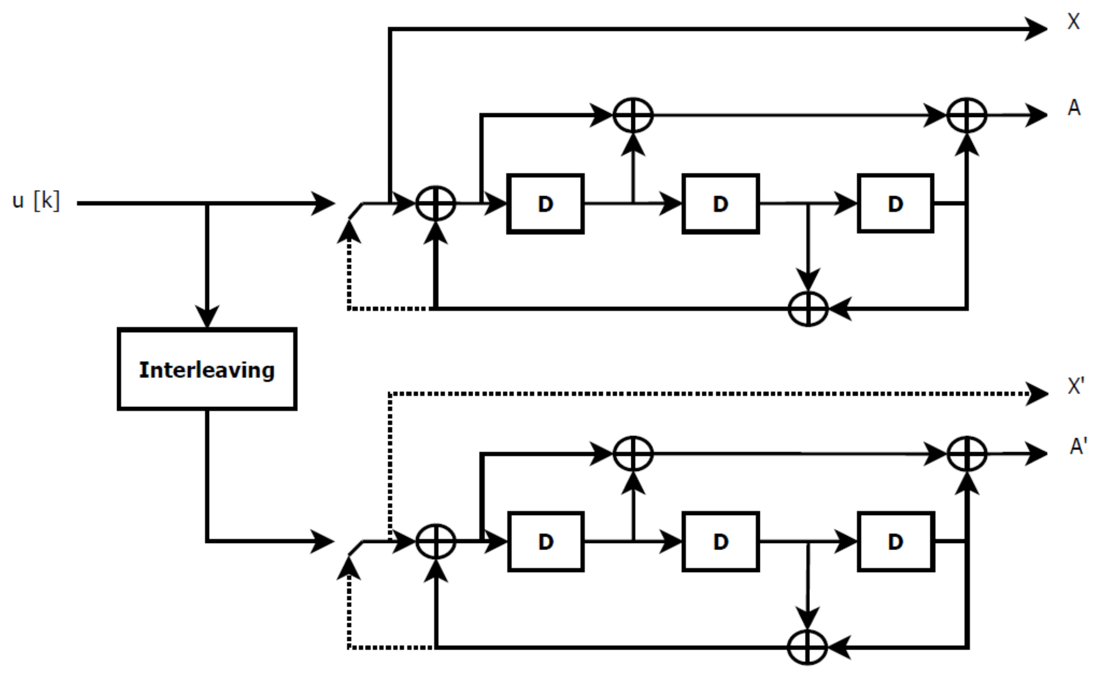

Figure 2-12: Structure of rate 1/3 Turbo coder (dotted lines apply for trellis termination only). This figure is produced freely from [8, Figure 4].

The output sequence from the turbo encoding is:

- X_1, A_1, A'_1, X_2, A_2, A'_2, … , X_K, A_K, A'_K

The termination is done by changing the switches to its lower position, illustrated by the dashed line in the picture, where three zero bits is inserted by adding the fed back bit to itself. The termination bits are outputted as:

- X_{K+1}, A_{K+1}, X_{K+2}, A_{K+2}, X_{K+3}, A_{K+3}, X'_{K+1}, A'_{K+1}, X'_{K+2}, A'_{K+2}, X'_{K+3}, A'_{K+3}

2.2.2 Interleaving

The following subsections summarize how the interleaving is presented in [8].

2.2.2.1 Initialization

The input sequence is arranged in a $R \times C$ rectangular matrix, where R denotes the number of rows and C the number of columns. The arranging of the matrix is done in the following three steps:

1. Decide the number of rows R, numbered from 0 to R-1 from top to bottom as:

$$R = \begin{cases} 5, & \text{if } (40 \le K \le 159) \\ 10, & \text{if } ((160 \le K \le 200) \text{ or } (481 \le K \le 530)) \\ 20, & \text{if } (K = \text{any other value}) \end{cases}$$

2. Determine a prime number p and the number of columns C numbered from 0 to C-1 from left to right as:
   - If $(481 \le K \le 530)$: $p = C = 53$.
   - Otherwise, from Table 2-1, find a prime number p that fulfils: $K \le R \times (p+1)$ and determine C as:

$$C = \begin{cases} p-1, & \text{if } K \le R \times (p-1) \\ p, & \text{if } R \times (p-1) < K \le R \times p \\ p+1, & \text{if } R \times p < K \end{cases}$$

p	*v*	*p*	*v*	*p*	*v*	*p*	*v*	*p*	*v*
7	3	47	5	101	2	157	5	223	3
11	2	53	2	103	5	163	2	227	2
13	2	59	2	107	2	167	5	229	6
17	3	61	2	109	6	173	2	233	3
19	2	67	2	113	3	179	2	239	7
23	5	71	7	127	3	181	2	241	7
29	2	73	5	131	2	191	19	251	6
31	3	79	3	137	3	193	5	257	3
37	2	83	2	139	2	197	2		
41	6	89	3	149	2	199	3		
43	3	97	5	151	6	211	2		

Table 2-1: List of prime number *p* and associated primitive root *v*. This table is produced freely from [8, Table 2]

3. Write the bits into the matrix row by row and end with possibly a couple of dummy bits (0 or 1) at the end to fill the matrix.

2.2.2.2 Permutations

After the bits have been written to the matrix they are intra-row and inter-row permuted according to six steps:

1. Depending on the prime *p* chosen in the initialization, choose a primitive root *v*, associated to *p*. This could be done from Table 2-1.
2. Build a base sequence *s*(*j*) from *v* and *s*(0)=1 as:

   $s(j) = (v \times s(j-1)) \bmod p$, $j = 1, 2, \ldots, p-2$.
3. Determine q_i, for $i=0, 1, \ldots, R-1$, for $q_0=1$ as:

   $q_i > 6$ and $q_i > q_{i-1}$ should be a least prime integer such that g.c.d[*](q_i, p-1)=1.
4. Construct the sequence r_i, $i=0, 1, \ldots, R-1$ as:

   $r_{T(i)} = q_i$, where $T(i)$ is a predefined pattern shown in Table 2-2 below.

[*] g.c.d. is the greatest common divisor, which is the largest positive integer that divides both numbers without a remainder.

Number of input bits K	Number of rows R	Inter-row permutation patterns <$T(0)$, $T(1)$, …, $T(R-1)$>
$40 \leq K \leq 159$	5	<4, 3, 2, 1, 0>
($160 \leq K \leq 200$) or ($481 \leq K \leq 530$)	10	<9, 8, 7, 6, 5, 4, 3, 2, 1, 0>
($2281 \leq K \leq 2480$) or ($3161 \leq K \leq 3210$)	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 16, 13, 17, 15, 3, 1, 6, 11, 8, 10>
K = any other value	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 10, 8, 13, 17, 3, 1, 16, 6, 15, 11>

Table 2-2: Inter-row permutation patterns for Turbo code internal interleaver. This table is produced freely from [8, Table 3]

5. Perform the intra-row permutation for i=0, 1, …, R-1, for different values of C as:

   – ($C=p$):

   $U_i(j)=s[(j \times r_i) \bmod (p-1)]$, j=0, 1, …, (p-2), and $U_i(p-1)=0$, where $U_i(j)$ is the input bit position of j-th permuted bit of row i.

   – ($C=p+1$):

   $U_i(j)=s[(j \times r_i) \bmod (p-1)]$, j=0, 1, …, (p-2), and $U_i(p-1)=0$ and $U_i(p-1)=0$, where $U_i(j)$ is the input bit position of j-th permuted bit of row i, and if $K=RC$ then $U_{R-1}(p)$ should be exchanged with $U_{R-1}(0)$.

   – ($C=p-1$):

   $U_i(j)=s[(j \times r_i) \bmod (p-1)]-1$, j=0, 1, …, (p-2), where $U_i(j)$ is the input bit position of j-th permuted bit of row i.

6. Use T(i) to perform inter-row permutations, where $T(i)$ is the rows original position of the permuted row i.

2.2.2.3 Output

After the permutations, the matrix is read column-wise starting at column 0 and row 0 and ending at column C-1 and row R-1. The permuted values corresponding to dummy values are removed from the output.

2.2.3 Decoding

The documentation of the 3GPP channel coding does not describe any standard decoding method. A guess would be that as long as the decoder follows the scheme of the encoder it could use any of the existing decoding methods. This would give the implementer a greater freedom in designing the decoder that fits its requirements of speed and error rate performance.

2.3 C64x DSP

2.3.1 Introduction

The C64x DSP family is described in more detail in [9]. The C64x DSP is a generation in the TMS320C6000^{TM} DSP platform, in which these fixed-point DSPs are the ones with the highest performance. They are developed by Texas Instruments (TI) and can be used in wireless applications.

With the use of the advanced architecture of very-long-instruction-word (VLIW) and eight independent functional units, it has cost-effective solutions. Out of the eight functional units, two are multipliers, which are capable of producing four 16-bit or eight 8-bit multiply-accumulates (MACs) per cycle. The other six are arithmetic logic units (ALUs).

The DSPs also contain 64 32-bit General-Purpose Registers and some of them, e.g. the TMS320C6416, have two built-in coprocessors, one Viterbi-decoder Coprocessor and one Turbo-decoder Coprocessor, making it possible to speed up the channel decoding. These two coprocessors communicate with the DSP via Enhanced Direct-Memory-Access (EDMA).

2.3.1.1 VLIW

VLIW is an architecture that has been very successful in DSPs, thanks to its parallelism that increases the performance of the processors [10]. Using a relatively simple technique and without consuming especially much power, it is able to achieve a high instruction level parallelism and run multiple operations every cycle. Instead of running one operation each cycle, there are multiple functional units that can handle different operations in parallel and therefore the speed of the program could be increased [11].

2.3.1.2 MAC

In DSPs the MAC operation is important, when realizing digital filters and convolution [12]. It is also very expensive. The operation is a multiplication of two numbers, which are added and stored to an accumulator, according to:

$$A \leftarrow A + (B \times C).$$

2.3.1.3 EDMA

The EDMA controls the communication between the level-two cash/memory and the device peripherals [13]. It has a programmable priority and allows data transfers to/from addresses in the memory space, peripherals and external memory. It is also possible to chain and link data transfers to each other.

2.3.2 TCP

For a complete description of the TCP, refer to [14]. The TCP is a coprocessor, which is built-in in some of the TMS320C6000^{TM} DSPs, e.g. the C6416 DSP. It has been developed to perform the turbo decoding for the 3GPP and IS2000 wireless standards. This thesis only deals with 3GPP turbo coding, the IS2000 standards will not be discussed further.

2.3.2.1 Features

The TCP comes with the following features:

- Performance:
    - The TCP consists of a paralleled architecture resulting in a low processing delay.
    - The possibility of enabling a stopping criteria algorithm can reduce the processing delay even further.
    - It is possible for the TCP and the DSP to run at full speed at the same time.
- Optimization:
    - By performing the turbo decoding on the TCP, both the board space and the power consumption is reduced.
    - The TCP has an own optimized working memory.
- Flexibility:
    - It accepts frame sizes from 40 up to 20730 and all the rates and polynomials used in the 3GPP and IS2000 coding standards.
    - Any kind of interleaver can be used.

2.3.2.2 Encoding

The encoding in the TCP consists of two identical systematic and recursive rate $^1/_3$ convolutional encoders, which use termination bits. One encodes the information sequence and the other an interleaved version of the information sequence. The encoded bits are then punctured and repeated before being outputted. The encoder is illustrated in Figure 2-13.

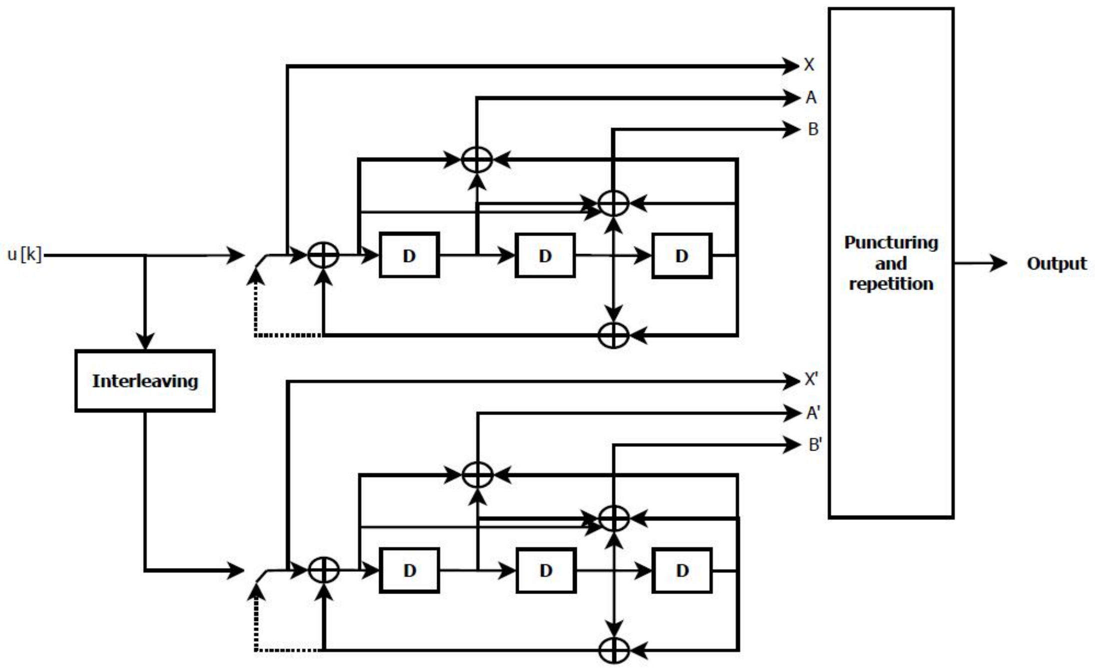

Figure 2-13: 3GPP and IS2000 Turbo Encoder Block Diagram. This figure is produced freely from [14, Figure 1].

Given a frame size F and the notations from Figure 2-13, the puncturing and repetition is done according to the following scheme:

- Data rate ½ (2×*F* bits):

– $X_0A_0X_1A'_1X_2A_2X_3A'_3$…

- Data rate $^1/_3$ (3×*F* bits):

– $X_0A_0A'_0X_1A_1A'_1X_2A_2A'_2X_3A_3A'_3$ …

- Date rate $^1/_4$ (4×*F* bits):

– $X_0A_0B_0B'_0X_1A_1A'_1B'_1X_2A_2B_2B'_2X_3A_3A'_3B'_3$…

When all the F information bits have been shifted into the encoder, the switch switches to its lower position and the tail bits are punctured and repeated as:

- IS2000 tail rate ½ and 3GPP tail rate $^1/_3$: 12 bits

– $X_FA_FX_{F+1}A_{F+1}X_{F+2}A_{F+2}X'_FA'_FX'_{F+1}A'_{F+1}X'_{F+2}A'_{F+2}$

- IS2000 tail rate $^1/_3$: 18 bits (systematic bit repeated twice)

– $X_FX_FA_FX_{F+1}X_{F+1}A_{F+1}X_{F+2}X_{F+2}A_{F+2}X'_FX'_FA'_FX'_{F+1}X'_{F+1}A'_{F+1}X'_{F+2}X'_{F+2}A'_{F+2}$

- IS2000 tail rate $^1/_4$: 24 bits (systematic bit repeated twice)

– $X_FX_FA_FB_FX_{F+1}X_{F+1}A_{F+1}B_{F+1}X_{F+2}X_{F+2}A_{F+2}B_{F+2}X'_FX'_FA'_FB'_FX'_{F+1}X'_{F+1}A'_{F+1}B'_{F+1}X'_{F+2}X'_{F+2}A'_{F+2}B'_{F+2}$

2.3.2.3 Decoding

The decoding in the TCP is based on the log-MAP algorithm, which, as discussed earlier, is an approximation of the BCJR algorithm. Two decoders iteratively decode the two different encoder's bits. The inputs to the decoders are the received signal and the a priori probabilities. For the first iteration the a priori probabilities are approximated values based on the received but corrupted information bits. The output of the decoders are soft values of the a posteriori probabilities which are first interleaved and passed as a priori probability inputs to the other decoder. The iteration continues either for a predefined number of iterations or until the decoding has reached satisfactory convergence. Finally it outputs the estimated hard values of the information sequence. The decoding is illustrated in Figure 2-14.

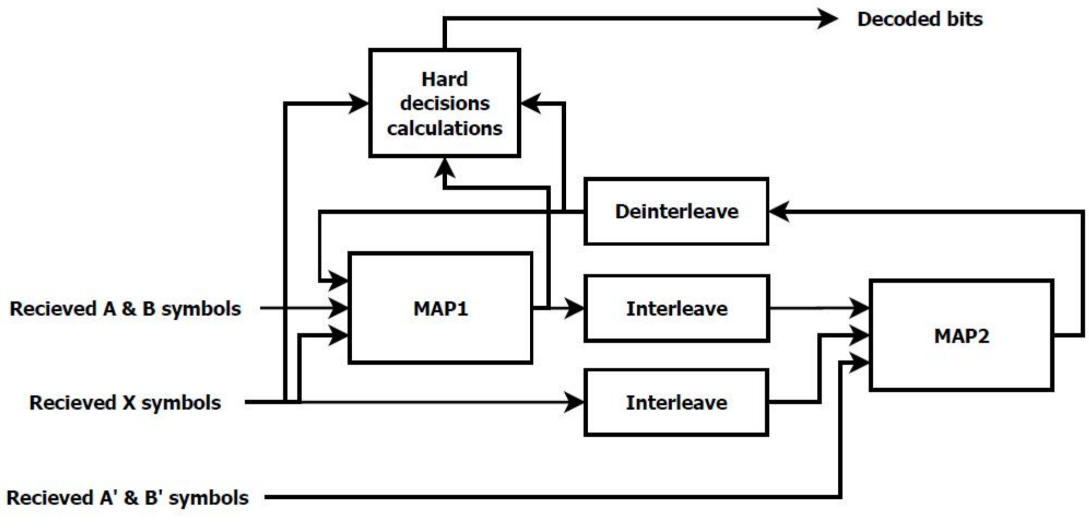

Figure 2-14: 3GPP and IS2000 Turbo Decoder Block Diagram. This figure is produced freely from [14, Figure 2].

2.3.3 Code Composer Studio^{TM}

Code Composer Studio (CCS) is an integrated development environment (IDE) developed by TI for their DSPs, microcontrollers and application processors [15]. It contains tools for developing and debugging embedded applications and includes e.g. compilers for different TI devises, simulators, a source code editor and a debugger.

In CCS it is possible to implement applications using C, C++ or assembly code and, once built, it is possible to simulate and step through all the instructions and view the content of the different memories.

2.3.3.1 Optimization

There are two parameters to set the optimization for the compiler in CCS [15]. The first represents the compilers optimization level, which speeds up the code by removing unnecessary codes. The other represents optimization for the code size. This does not increase the speed, but minimizes the memory needed by the code.

Each optimizer consists of a grading of five possible optimization levels. For the optimization level the following are possible to choose:

- Nothing – no optimization
- O0 – optimizes registers
- O1 – same as O0 plus adding local optimizations
- O2 – same as O1 plus adding global optimizations
- O3 – same as O2 plus adding interprocedure optimization

Chapter 3

Problem Description

3.1 Evaluation

When implementing algorithms on a DSP, there are a couple of important factors that need to be considered. These often deal with the memory usage and execution speed of the code. The code should run as fast as possible, while using a small amount of memory.

As the evaluation criteria, three important factors are considered: code density, random access memory (RAM) usage and data rate.

3.1.1 Code Density

The code density is the amount of code (in bytes) that is generated when the source code is compiled. This includes variable declaration and the list of instructions. This value could be calculated by building the code to an out file. The size of this file is the code density.

3.1.2 RAM Usage

The RAM usage is the amount of memory, in bytes, other than that of the actual code, required by the application when the code executes. Here the variables that use dynamic memory allocation, e.g. vectors and arrays, are stored. This value could be estimated by gathering all the objects in the code which use dynamic memory allocation and check how much memory they occupy.

The input and output buffers will be designed to contain enough memory to accept one block of data. It is however possible to increase these buffers to be able to load uncoded data to, and read encoded data from the buffers while the present block is being coded. This would of course increase the RAM usage.

3.1.3 Data Rate

The data rate is one way of defining the efficiency of the code. It measures the number of bits that could run through the algorithm without a delay. When the code is a part of a larger system it is important that the data rate fulfils the demand of the entire system.

The data rate *R* could be calculated using a functionality of CCS called Profiler, where it is possible to collect the number of clock cycles used by each function. Taking the number of coded bits and dividing by the number of clock cycles gives a value of how many bits that can be coded for each clock cycle. Using the knowledge that the C64x DSP can run in f = 600 MHz, i.e. 600×10^6 clock cycles per second, it is possible to calculate the bit rate, in bits/s, of the coding according to:

$$R = \frac{\#\,\text{bits}}{\#\,\text{clock\_cycl es}} f \ .$$

For the encoder, #bits are the number of input bits and *R* is a measurement of how many bits it could encode per second. For the decoders, #bits are the number of output bits and *R* is a measurement of how many bits they could decode per second. Since the encoder will have more output bits than input bits, it is important to notice that the output data rate will be higher than the input. The opposite holds for the decoder.

This rate does not include the delay where the system waits for the input bits, i.e. it assumes that all the bits in the same block that should be encoded and decoded are received at the same time. In a real system however, using large in and out buffers, as mentioned in section 3.1.2, can minimize this delay.

3.2 Tasks

The problem is divided into four different tasks:

1. Evaluation of turbo encoding on a C64x DSP.
2. Evaluation of turbo decoding on a C64x DSP, implemented as a conventional C-based algorithm.
3. Evaluation of turbo decoding on a C64x DSP, implemented using the TCP.
4. Comparison between the two decodings.

3.2.1 Encoding

To be able to evaluate the performance of the two different decoding, a turbo encoder will be implemented, using a C-based algorithm.

3.2.2 C-based Decoding

The evaluation of the C-based decoding will consider the performance of turbo coding on a C64x DSP. The implementation will follow the 3GPP wireless standards and be implemented as a conventional C-based algorithm.

3.2.3 TCP Decoding

The evaluation of the TCP decoding will consider the performance of turbo coding on a C64x DSP. The C64x TCP has been designed to perform the turbo decoding operations for the 3GPP wireless standards.

Chapter 4

Implementation

The implementation consists of three different parts, encoding, C-based decoding and TCP decoding. The first two are written in C++ and the last is performed with the help of the coprocessor and EDMA communication.

The C-based algorithms are implemented based on IT++, an open source C++ library mainly used for simulation and performance research in communication systems [16]. The IT++ library includes turbo coding and the implementation consisted of modifying this code to cope with CCS and the C64x DSP.

The C-based decoding has been implemented for comparison with the TCP, i.e. the same state diagram and interleaving has been used and the decoding algorithm is the log-MAP algorithm running over several iterations.

To be able to simulate the decoding of soft data, a dummy input stream, modulation and AWGN channel has been implemented. The input stream is a simple implementation outputting 0 or 1, with equal probability, using an ordinary random function. Each zero is then mapped to +1.0 and each one to -1.0, which can illustrate the use of BPSK modulation. To simulate the AWGN channel, a white Gaussian noise is added to represent the disturbance of the channel. The noise is implemented using an algorithm in which it is possible to decide the variance of the noise.

The channel is also scaled at the receiver and the received data is modified to function with the decoding. The scaling for BPSK follows the reasoning in [3], where the log-likelihood ratio of the demodulated bit *b* from the corresponding received signal *y* is:

$$\mathrm{L}_{\text{channel}}(b) = \frac{2A}{\sigma^2} y\,,$$

where A is the amplitude of the signal and σ^2 the variance of the noise. With an amplitude $A = 1$ and the one-sided power spectral density of the noise defined according to [3] as: $N_0 \equiv \sigma^2/2$, this gives:

$$\mathrm{L}_{\text{channel}}(b) = \frac{4}{N_0} y\,.$$

4.1 Encoding

The turbo encoding in IT++ consists mainly of two different parts, interleaving and encoding. The encoding is done according to Figure 4-1. The input data is encoded block-wise with a predefined block length using the encoder illustrated in Figure 4-2. The input data is then interleaved and encoded once more. The systematic (input), parity and tail bits are then combined according to the turbo-decoder coprocessors puncturing and repetition scheme, using the rate $R = {}^1/_3$. The reason for this is that the IT++ library has a “speed optimized decoder for $R = {}^1/_3$”.

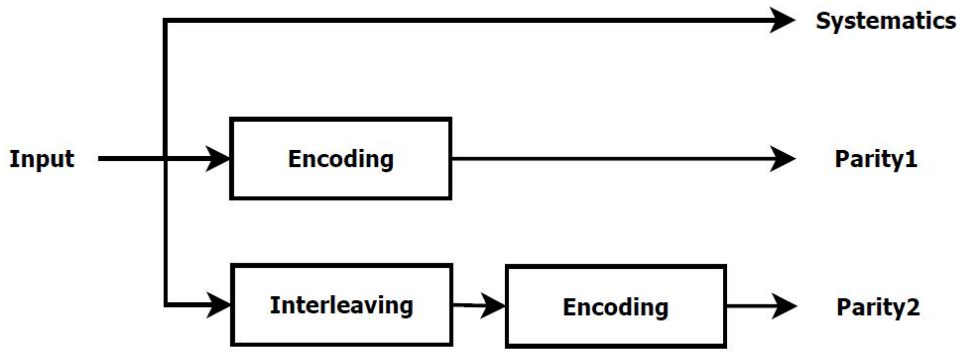

Figure 4-1: Block diagram of the encoder.

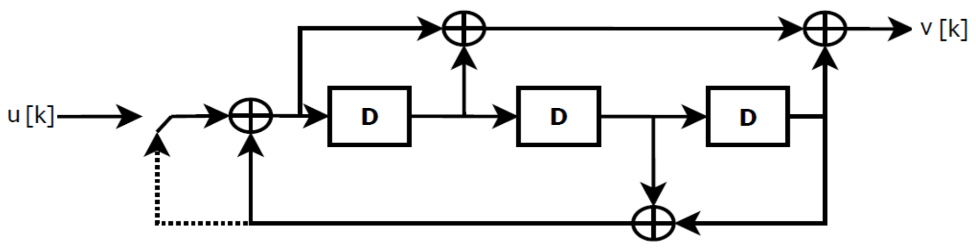

Figure 4-2: The encoding part.

4.1.1 Interleaving

The interleaving can interleave binary vectors. It uses an interleaving sequence to map each position in the input vector to the interleaved output vector according to:

out[i] = in[interleaving sequence[i]].

The interleaving sequence is built according to the 3GPP Turbo Coding Standards described in section 2.2.2 and calculated in the initialization of the encoding.

4.1.2 Encoding

A state transition table as well as an output parity table based on the shift registers are built in the initialization of the encoding. This has been illustrated in Figure 4-3 as a trellis diagram. These tables, the input bit and the present state are used to find the corresponding output bit and the next state. This is done for all the bits in the block before it forces the state diagram to zero by inserting some additional tail bits. This part outputs the parity and tail bits.

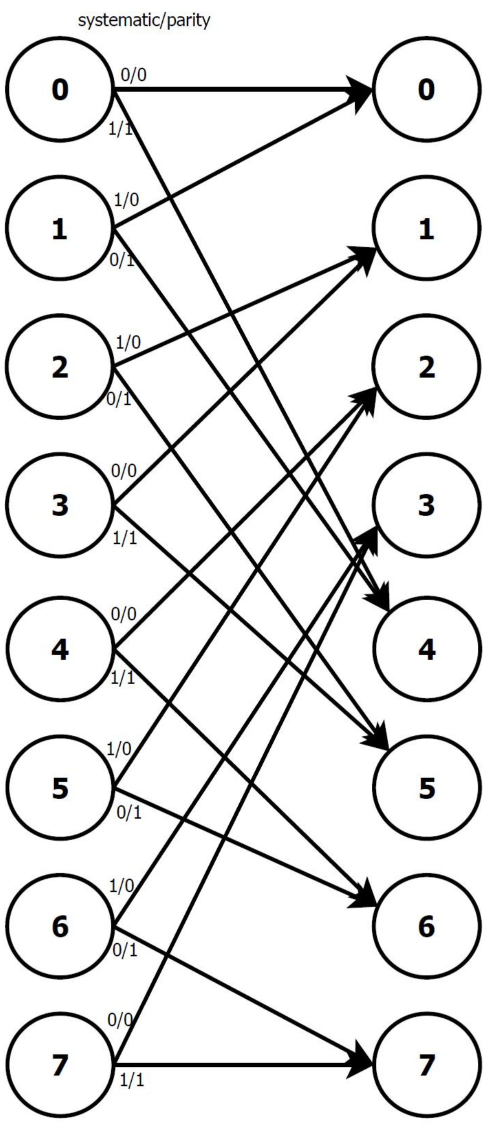

Figure 4-3: State transition and output parities.

4.2 C-based decoding

The turbo decoding in IT++ mainly consists of three different parts, interleaving, deinterleaving and calculation of log-likelihood values of each bit (extrinsic outputs) using the log-MAP algorithm. The decoding is illustrated in Figure 4-4.

The turbo decoding in IT++ is a “speed optimized decoder for R=$^1/_3$”, which receives soft valued data. A negative value is considered to be a probable 1 and a positive value a probable 0. It is possible to decide how many iterations the decoding should run.

The decoding starts by separating the systematic bits from the parity bits and the systematic bits are interleaved to represent the output from the second encoder. Then each run consists of a calculation of the extrinsic outputs of non-interleaved data, interleaving, calculation of the extrinsic outputs of interleaved data followed by a deinterleaving. For the first extrinsic calculation in the first iteration the a priori probabilities (extrinsic inputs) are assumed to be all zero, illustrated by a dashed line in Figure 4-4. Thereafter the interleaved or deinterleaved extrinsic outputs are used as extrinsic inputs.

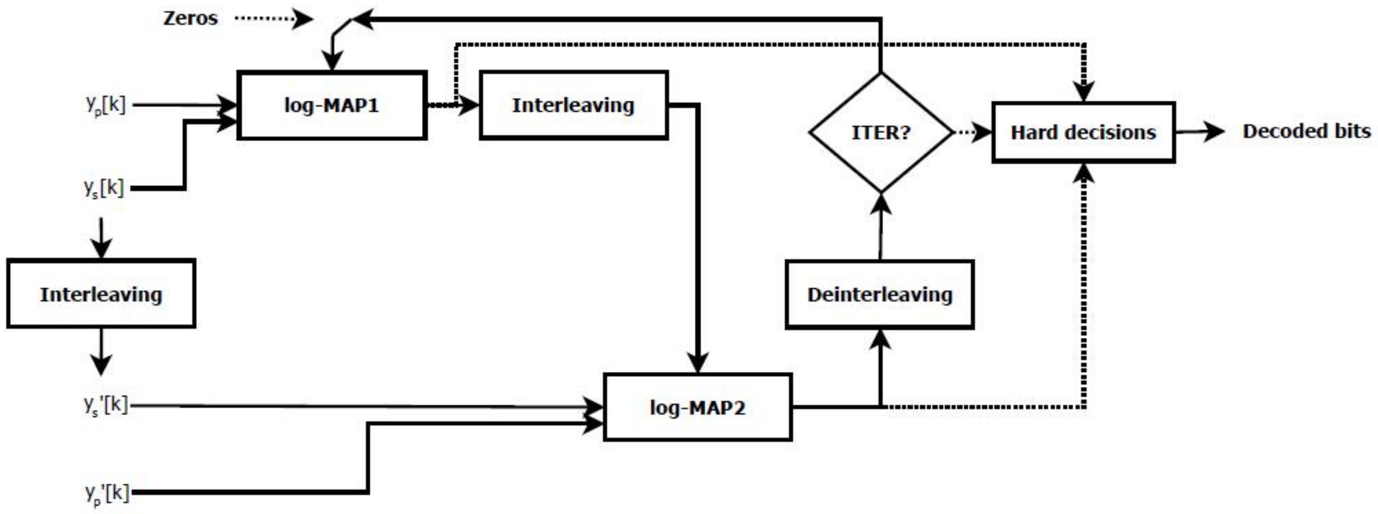

Figure 4-4: C-based decoding.

This is performed over the predefined number of iterations. After the last run the turbo decoder takes a hard decision based on the received data and the extrinsic output values from the last two extrinsic calculations, which is illustrated by the dashed lines to the hard decision block in Figure 4-4.

4.2.1 Interleaving and deinterleaving

This part is mainly the same as the one described in section 4.1.1. A deinterleaving of a vector is performed as:

out[interleaving_sequence[i]] = in[i].

4.2.2 Calculation of extrinsic output

The calculations of the extrinsic outputs are made according to the log-MAP algorithm described in section 2.1.2.2. The receiver uses the received data and the extrinsic inputs, a state transition and an output parity table (calculated the same way as in the encoding) to calculate the γ values. Using the initial values of α and β and all the γ values, the other α and β values are calculated. The extrinsic outputs are then calculated using α, β and γ.

4.3 Coprocessor

The coprocessor can be used in two different modes, stand alone and shared-processing. It is however recommended to use the stand alone mode for block sizes between 40 and 5114 as is the case for 3GPP turbo coding. Therefore this implementation only uses the stand alone mode.

The stand alone mode requires three types of EDMA data transmissions namely: (i) input configuration parameters, (ii) systematic and parities and (iii) interleaver indexes. Also the hard decision data needs to be received. Optionally some output parameters can also be received.

The programming is made with the help of a chip support library (CSL), which provides an application programming interface (API) described in [17].

4.3.1 Input configuration parameters

The input configuration parameters are 12 32-bit registers [14]. They are used to control and define different parts in the decoding, e.g. mode, rate, number of iterations and block length. Also the different tail bits are placed in these registers. For details about all the registers refer to [14].

According to the API described in [17], the input configuration parameters could be built, mainly using the two functions TCP_genParams(), and TCP_genIC(). These functions use some information given by the programmer to construct all the 12 registers represented as a struct. For details on what the functions do, refer to [17].

4.3.2 Systematic and parity data

The systematic and parity data are represented in eight bits as: SIIII.FFF where S is a sign bit, the four I represent an integer value between 0 and 15 and the three F represent the fraction. Together they range from -15.875 (10000000), which is a highly probable 0, to 15.875 (01111111), which is a highly probable 1.

The data should be aligned according to Table 4-1, where the systematic and parity data represented as above are saved in the register, starting at a base address. Since it is a 32-bit register, the data is arranged in groups of four, where “X” represent the systematic data, “A” the parity data from the first encoder and “A'” the parity data from the second encoder.

	Data			
Address (bytes)	**MSB**			**LSB**
Base	X1	A'0	A0	X0
Base + 4h	A2	X2	A'1	A1
Base + 8h	…			A'2

Table 4-1: Rate ⅓ Systematic/Parity Data. This table is produced freely from [14, Table 3].

4.3.3 Interleaver index

The interleaver index consists of 13-bit values saved as 16 bits right justified according to Table 4-2, i.e. in groups of two indexes per address. The interleaver table is built according to the 3GPP Turbo Coding Standard described earlier and calculated in the initialization of the decoding.

	Data	
Address (bytes)	**MSB**	**LSB**
Base	Index1	Index0
Base + 4h	Index3	Index2
Base + 8h	…	

Table 4-2: Interleaver data. This table is produced freely from [14, Table 5].

4.3.4 Hard decision data

The output data is received from the coprocessor as 32-bit words with the first symbols hard decision bit in the least significant bit-position in the first word.

4.3.5 Output parameters

Optionally the output parameters could be received. This is a 32-bit word, where the most significant 16 bits represent the number of used iterations and the least significant 16 bits are reserved.

4.3.6 EDMA Programming

There are two available EDMA channel events designated to the TCP, a receive event and a transmission event. Each event is based on EDMA parameters, which are 32-bit words illustrated in Figure 4-5, with the OPT field illustrated in Figure 4-6. For more details about each field, refer to [13].

To decode using the TCP, these four (optionally five) events needs to be initiated by creating the corresponding EDMA parameters. This is mainly done by using the API functions EDMA_config() and EDMA_link() along with the options in the following subsections. Finally by writing a START command to the TCP execution register TCPEXE the EDMA transmission will start.

EDMA Channel Option Parameters (OPT)	
EDMA Channel Source Address (SRC)	
Array/Frame count (FRMCNT)	Element count (ELECNT)
EDMA Channel Destination Address (DST)	
Array/Frame index (FRMIDX)	Element index (ELEIDX)
Element count reload (ELERLD)	Link address (LINK)

Figure 4-5: EDMA Parameters Structure. This figure is produced freely from [14, Figure 26 (a)]

31 29	28 27	26	25 24	23	22 21	20	19 16
PRI	ESIZE	2DS	SUM	2DD	DUM	TCINT	TCC

15	14 13	12	11	10 5	4	3	2	1	0
-	TCCM	ATCINT	-	ATCC	-	PDTS	PDTD	LINK	FS

Figure 4-6: EDMA Channel Option Parameters (OPT). This figure is produced freely from [14, Figure 26 (b)]

4.3.6.1 Input Configuration Parameters Transfer

This transfer is a transmission event sending the 12 input configuration registers. The parameters should be set as:

- OPT:
  - ESIZE = 00
  - 2DS = 2DD = 0
  - SUM = 01
  - DUM = 00
  - LINK = 1
  - FS = 1
- SRC: start address of the input configuration parameters
- ELECNT: 000Ch
- FRMCNT: 0000h
- DST: TCPIC0 (5800 0000h)
- ELEIDX: does not matter
- FRMIDX: does not matter
- LINK: address to the EDMA parameters associated with the systematics and parities
- ELERLD: does not matter

4.3.6.2 Systematic and Parities Transfer

This transfer is a transmission event sending the systematic and parity. The parameters should be set as:

- OPT:
  - ESIZE = 00
  - 2DS = 2DD = 0
  - SUM = 01
  - DUM = 00
  - LINK = 1
  - FS = 1
- SRC: start address of the systematic and parity words
- ELECNT: $2 \times \text{ceil}(\frac{F \times R}{8 \times (\text{FRMCNT} + 1)})$, with *F* as the frame length and *R* the rate
- FRMCNT: $\text{ceil}(\frac{F \times R}{(\text{NWDSYPAR} \times 4)}) - 1$, where NWDSYPAR is set in TCPIC3
- DST: TCPSP (5802 0000h)
- ELEIDX: does not matter
- FRMIDX: does not matter
- LINK: address to the EDMA parameters associated with the interleaver table
- ELERLD: does not matter

4.3.6.3 Interleaver Indexes Transfer

This transfer is a transmission event sending the interleaver table. The parameters should be set as:

- OPT:
  - ESIZE = 00
  - 2DS = 2DD = 0
  - SUM = 01
  - DUM = 00
  - LINK = 1
  - FS = 1
- SRC: start address of the interleaver table
- ELECNT: $2 \times \text{ceil}(\frac{F}{4 \times (\text{FRMCNT} + 1)})$, where F is the frame length
- FRMCNT: $\text{ceil}(\frac{F}{(\text{NWDINTER} \times 2)}) - 1$, where NWDINTER is set in TCPIC3
- DST: TCPINTER (5808 0000h)
- ELEIDX: does not matter
- FRMIDX: does not matter
- LINK: address to a null EDMA parameter (with all zeros)
- ELERLD: does not matter

4.3.6.4 Hard-decision Transfer

This transfer is a receive event receiving the hard decision output bits. The parameters should be set as:

- OPT:
  - ESIZE = 00
  - 2DS = 2DD = 0
  - SUM = 00
  - DUM = 01
  - LINK = 1
  - FS = 1
- SRC: TCPHD (580A 0000h)
- ELECNT: $2 \times \text{ceil}(\frac{F}{64 \times (\text{FRMCNT} + 1)})$, where *F* is the frame length
- FRMCNT: $\text{ceil}(\frac{F}{(\text{NWDHD} \times 32)}) - 1$, where NWDHD is set in TCPIC5
- DST: start address of the output bits
- ELEIDX: does not matter
- FRMIDX: does not matter
- LINK: either an address to a null EDMA parameter (with all zeros) if output parameters should not be sent, otherwise the address to the EDMA parameters associated with the output parameters
- ELERLD: does not matter

4.3.6.5 Output Parameters Transfer

This transfer is optional and depends on an OUTF bit in TCPIC0. It is a receive event receiving of output parameters. The parameters should be set as:

- OPT:
  - ESIZE = 00
  - 2DS = 2DD = 0
  - SUM = 01
  - DUM = 01
  - LINK = 1
  - FS = 1
- SRC: TCPOUT (5800 0030h)
- ELECNT: 0002h
- FSMCNT: 0000h
- DST: start address of the output parameters
- ELEIDX: does not matter
- FRMIDX: does not matter
- LINK: address to a null EDMA parameter (with all zeros)
- ELERLD: does not matter

Chapter 5

Evaluation

5.1 Performance

To check whether the two different turbo decodings seemed reasonable or not the following test was created. A random input vector gets encoded in blocks of 1000 bits using the encoder, described in section 4.1, passed through the dummy channel and then decoded, using various numbers of iterations (1, 3, 6 and 13). The noise variance of the channel was varied to give a desired bit-SNR, given that the energy of each input bit is one (i.e. the energy of each encoded bit is ≈1/3).

The C-based decoding test was, for reduction of the simulation time, built by the g++ compiler instead of using CCS. This made it possible to use 1000000 bits as input and the result is shown in Figure 5-1. The result is also compared to a transmission without any coding.

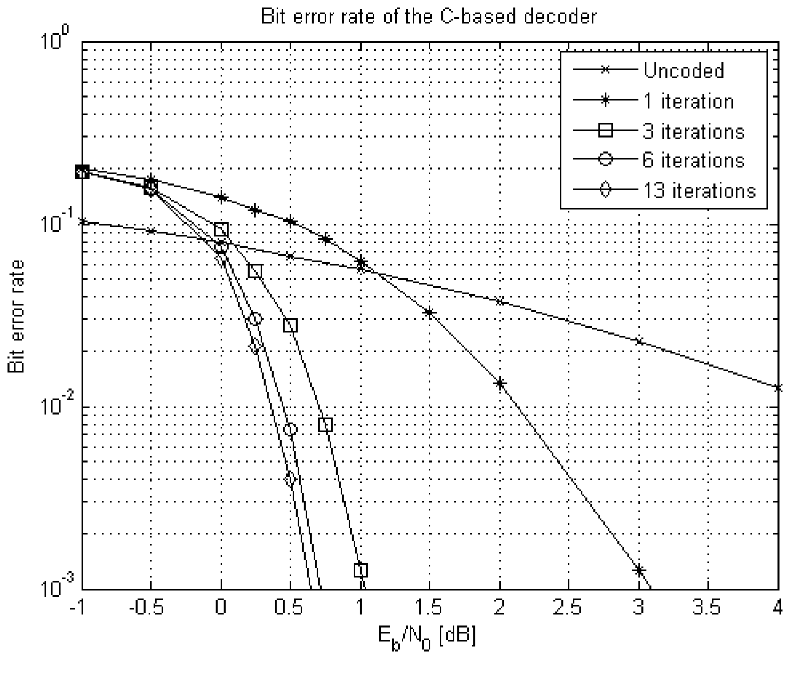

Figure 5-1: Bit error rate as a function of number of iterations.

The TCP decoding where performed in CCS and for reduction of the simulation time using only 10000 bits as input. The result is shown in Figure 5-2. This result is also compared to a transmission without any coding.

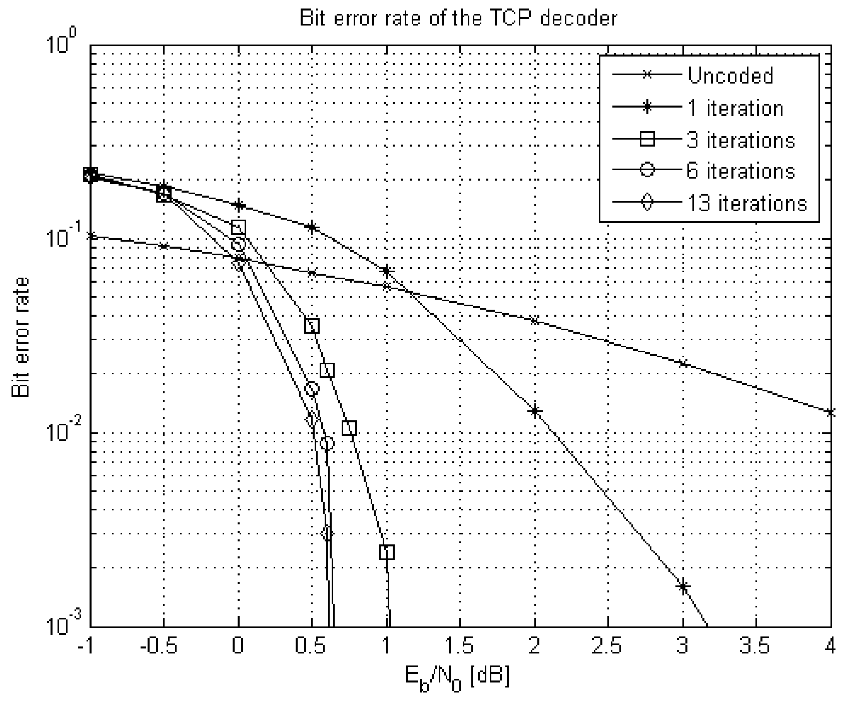

Figure 5-2: Bit error rate as a function of number of iterations.

The plots only show the performance down to a bit error rate of 10^{-3}, after that it would take even more input bits to get a proper result. Unfortunately the simulation time in code composer studio does make it hard to use more than 10000 bits as input to the TCP decoding, making this graph a bit edgy. Despite this, the plots are good enough to prove if the decoding works or not.

Given that the two test results resemble a lot to each other and also to Figure 2-8, which is given in [5]; it is safe to believe that the decoding works as it should. The difference between using one and three iterations is large, the gain of using even more iterations decreases for each iteration. It is also noticeable that the coding is tenable for SNRs below 0 dB, thereafter the bit error rate is rapidly decreasing compared to the uncoded transmission, at least when using three iterations or more.

5.1.1 Comparison

Combining the bit error rate tests from both tests the result shown in Figure 5-3 is obtained.

This result indicates that the C-based decoding is slightly better than the TCP decoding. However, the C-based decoding uses 100 times as many bits in the test, making it more reliable. So instead considering that six iterations of the TCP are better than three of the C-based decoding does imply that the difference is not that great and that their performance is approximately the same.

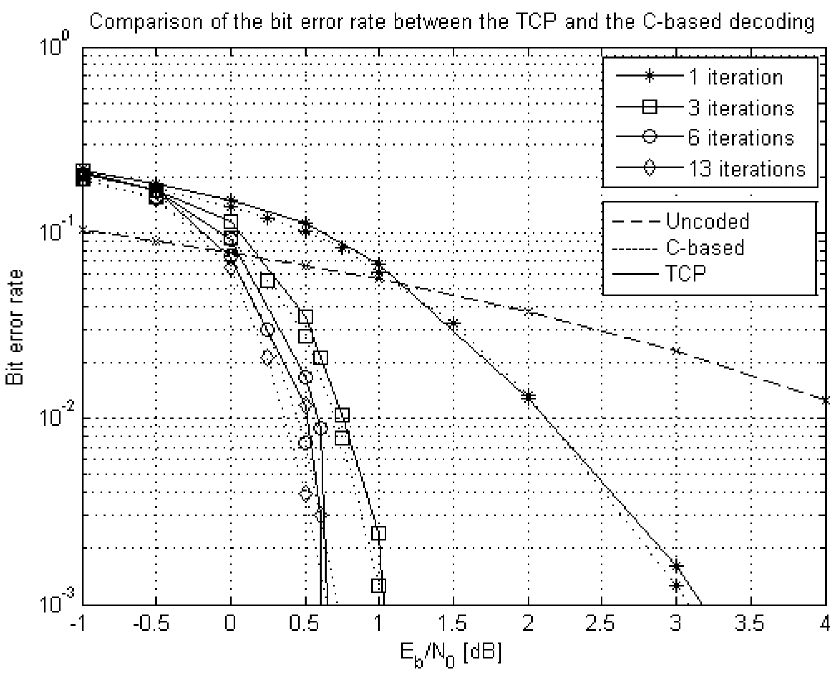

Figure 5-3: Comparison of the bit error rate.

5.1.2 Max-log-MAP

Since the C-based decoding, when using the log-MAP algorithm seems a bit better than the TCP, a similar test has been run using the Max-log-MAP instead of the log-MAP algorithm. The result, compared to the TCP and an uncoded transmission is shown in Figure 5-4, where the different lines illustrate different iterations (1, 3, 6 and 13) as earlier. This should, according to the theory, give a faster but more inaccurate decoding.

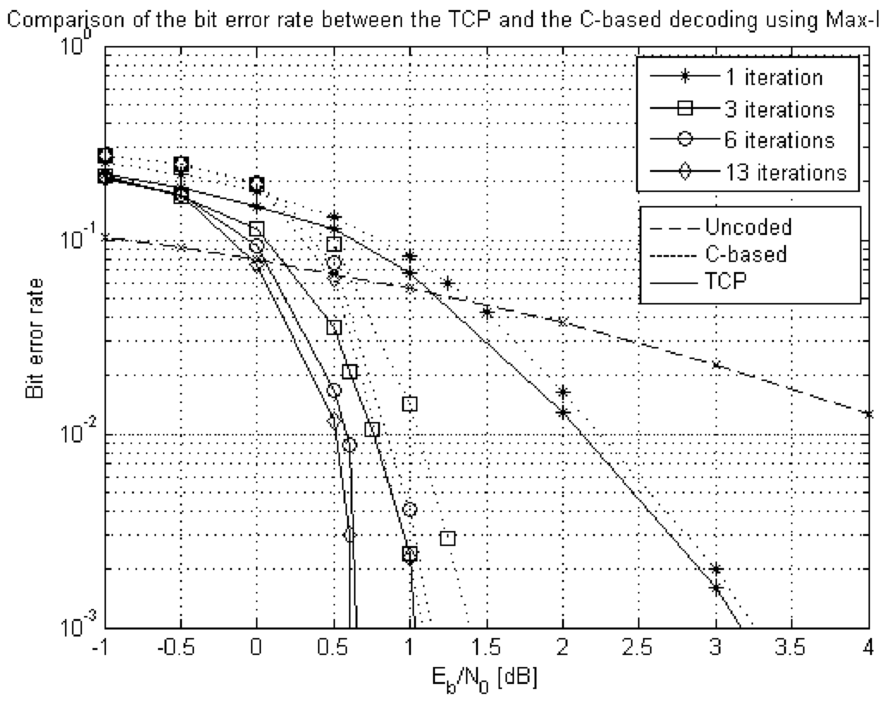

Figure 5-4: Comparison of the bit error rate when using the Max-log-MAP algorithm.

From this result it is easy to determine that the TCP is better than the C-based decoding using the Max-log-MAP algorithm. This implies that the TCP is actually using the log-MAP algorithm, even though the previous results indicated that the C-based algorithm is slightly better.

5.2 Code density

Using all different combinations of optimization when building the code, it is possible to get the code densities of the three codings. The result is illustrated for the encoding in Figure 5-5, the C-based decoding in Figure 5-6 and for the TCP in Figure 5-7.

Each line in the figures represents the different code optimization levels and the x-axis shows different optimizations for code size. The code density is displayed in kB and an initial guess is that the optimization for code size should decrease the code density.

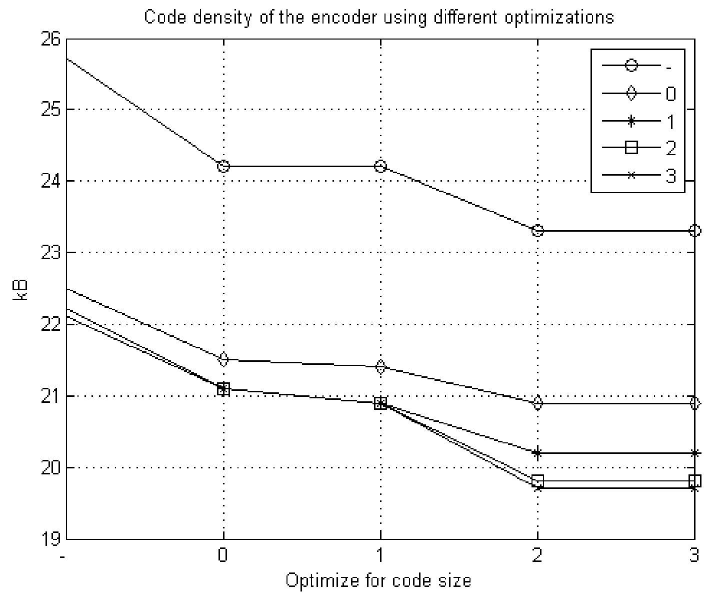

Figure 5-5: The code density used by the encoding.

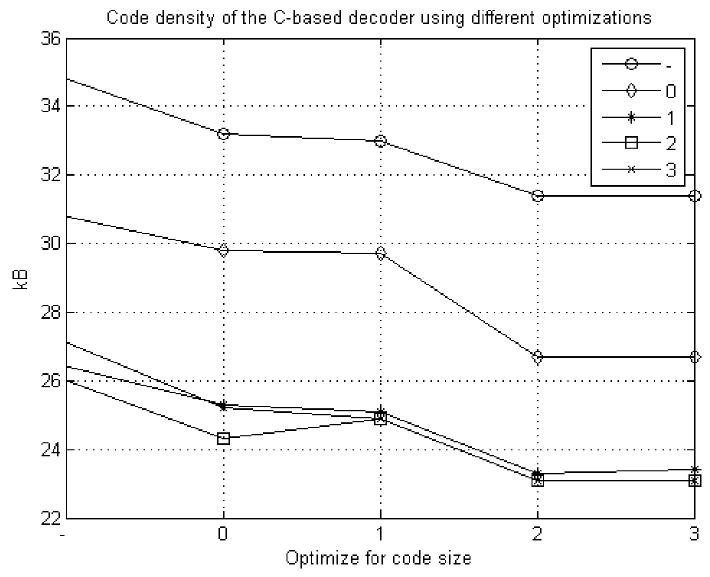

Figure 5-6: The code density used by the C-based decoding.

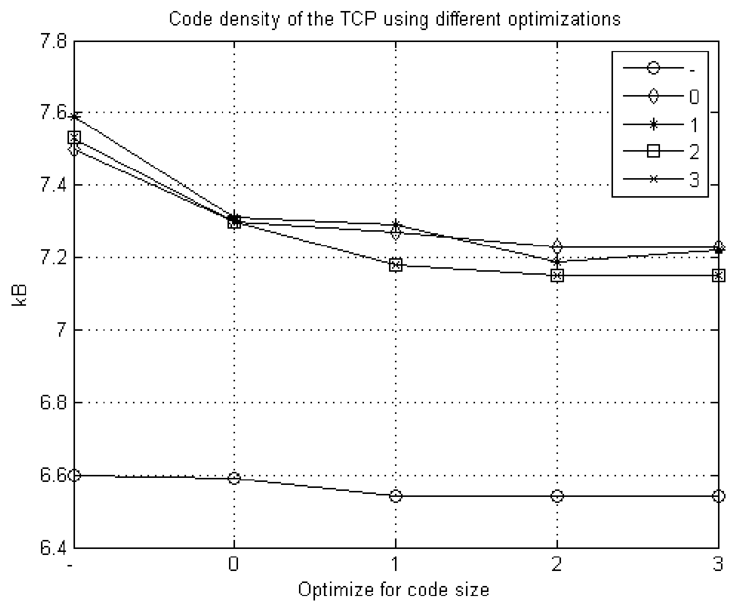

Figure 5-7: The code density used by the TCP decoding.

It is obvious that the optimization for code size reduces the code density and for the encoding and C-based decoding the optimization level also decreases the code density. A step from no optimization to the first levels of optimization gives a large gain in the code density while for the last levels the size does not improve especially much. The only odd factor about the C-based decoding is that there is a dip when using optimization level 2 and optimization 0 for code size. This behaviour is hard to explain and might just be a coincidence.

The same holds for the code density of the TCP, it acts as expected and gains in using both optimizations, at least for optimization levels 0-3 but remarkably, the best result is given when using no optimization level. The reason for this is probably due to the straightforward implementation of the TCP, giving an optimization of the code a worse result than using none.

The encoding has a code density, without optimization, of almost 26 kB, which is reduced down to 20 kB with high optimization. The difference of the code density for the C-based decoding is, 35 kB as highest and the 24 kB as lowest. The highest code density of the TCP is 7.6 kB, while the lowest is 6.6 kB.

5.2.1 Comparison

This result shows that the TCP decoding has the smallest code density. This makes sense considering that the main part of the code is performed in the coprocessor, which does not use any memory of the DSP. The C-based decoding has the highest code density, which also is expected since it is the largest algorithm.

Using the C-based decoding together with the encoding would make the total code density end up at 45-60 kB, while using the TCP decoding would give a total code density of approximately 30 kB, which is a reduction with a factor 2.

5.3 RAM Usage

Table 5-1 to 5-3 list the different objects in the encoding, the TCP and the C-based decoding, which use dynamic memory allocation, along with their data types and sizes. Many of them are dependent on the length of the input block, l_B.

Name	Data Type	Size
Input	Binary (1/8 Byte)	l_B
Output	Binary (1/8 Byte)	$3l_B + 12$
Interleaving sequence	Short (2 Bytes)	l_B
State transition table	Char (1 Byte)	16
Output parity table	Binary (1/8 Byte)	16
Interleaved input	Binary (1/8 Byte)	l_B
Parity bits from first rscc	Binary (1/8 Byte)	$l_B + 3$
Parity bits from second rscc	Binary (1/8 Byte)	$l_B + 3$
Tail from first rscc	Binary (1/8 Byte)	3
Tail from second rscc	Binary (1/8 Byte)	3

Table 5-1: The objects used by the encoder.

Name	Type	Size
Input	char (1 Byte)	$3l_B + 12$
Output	Binary (1/8 Byte)	l_B
Interleaving sequence	Short (2 Bytes)	l_B
Input configuration	32-bit word (4 Bytes)	12
Input configuration parameters	32-bit word (4 Bytes)	6
Systematic and parity parameters	32-bit word (4 Bytes)	6
Interleaver index parameters	32-bit word (4 Bytes)	6
Hard decision parameters	32-bit word (4 Bytes)	6
Output parameters	32-bit word (4 Bytes)	6

Table 5-2: The objects used in the TCP decoding.

Name	**Data Type**	**Size**
Input	Float (4 Bytes)	$3l_B + 12$
Output	Binary (1/8 Byte)	l_B
Interleaving sequence	Short (2 Bytes)	l_B
State transition table	Char (1 Byte)	16
Rev. state transition table	Char (1 Byte)	16
Output parity table	Binary (1/8 Byte)	16
Rev. output parity table	Binary (1/8 Byte)	16
Received systematics	Float (4 Bytes)	$l_B + 3$
Interleaved systematics	Float (4 Bytes)	$l_B + 3$
Rec parity from first rscc	Float (4 Bytes)	$l_B + 3$
Rec parity from second rscc	Float (4 Bytes)	$l_B + 3$
Extrinsic input	Float (4 Bytes)	l_B
Interleaved extrinsic input	Float (4 Bytes)	l_B
Extrinsic output	Float (4 Bytes)	l_B
Interleaved extrinsic output	Float (4 Bytes)	l_B
Alpha	Float (4 Bytes)	$8l_B + 8$
Beta	Float (4 Bytes)	$8l_B + 8$
Gamma	Float (4 Bytes)	$16l_B + 16$
Log likelihood values	Float (4 Bytes)	l_B

Table 5-3: The objects used by the C-based decoder.

Taking into account that only full bytes could be saved properly, Table 5-1 makes it possible to derive the following relation for the encoding:

$$\mathrm{RAM}_{\mathrm{ENC}} = 2 * l_B + \mathrm{ceil}(\frac{3}{8} l_B) + 4 * \mathrm{ceil}(\frac{1}{8} l_B) + 24 \ [\mathrm{Bytes}].$$

Table 5-2 gives the following relation for the TCP:

$$\mathrm{RAM}_{\mathrm{TCP}} = 5 * l_B + \mathrm{ceil}(\frac{1}{8} l_B) + 156 \ [\mathrm{Bytes}].$$

Table 5-3 gives the following relation for the C-based decoding:

$$\mathrm{RAM}_{\mathrm{C}} = 178 * l_B + \mathrm{ceil}(\frac{1}{8} l_B) + 260 \ [\mathrm{Bytes}].$$

The result is shown, in kB, in Figure 5-8 (encoding), Figure 5-9 (C-based decoding) and Figure 5-10 (TCP).

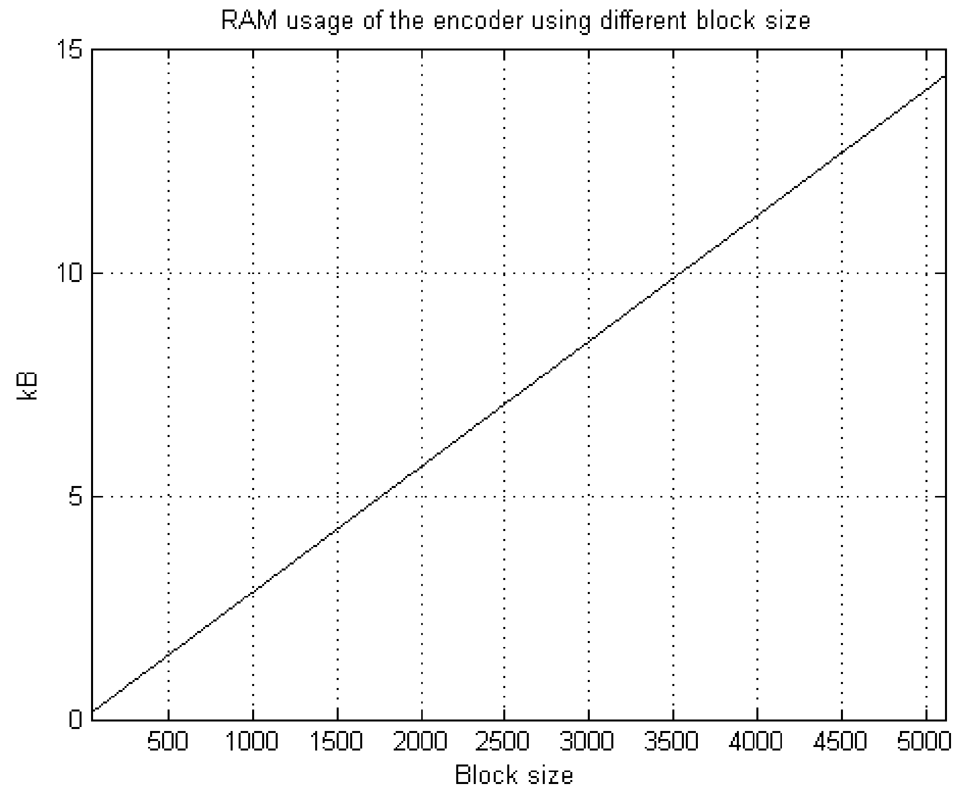

Figure 5-8: The RAM used by the encoder at different block size.

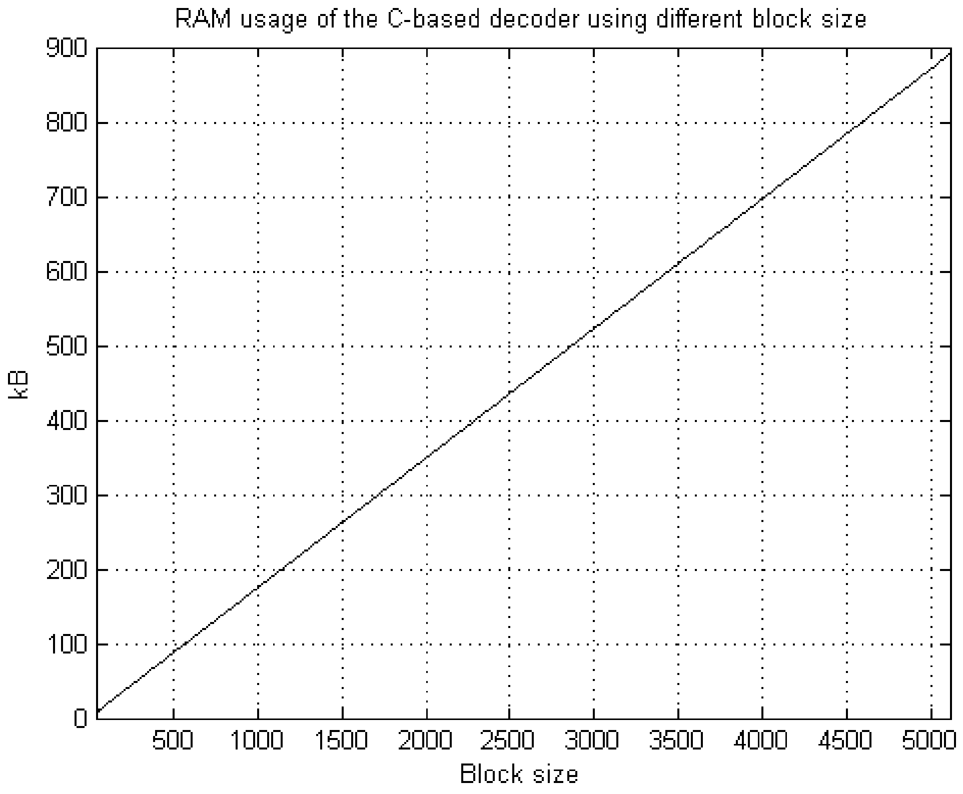

Figure 5-9: The RAM used by the C-based decoder at different block size.

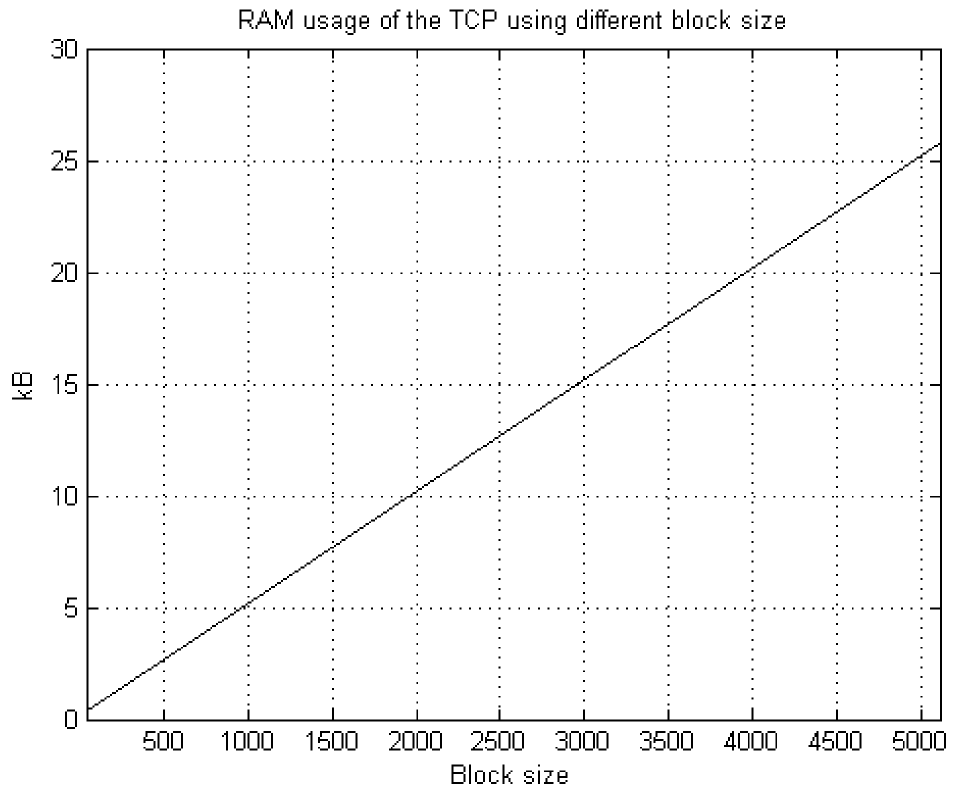

Figure 5-10: The RAM used by the TCP decoding at different block size.

The RAM usages of the different codings behave like linear functions, which all depend on the block size. The RAM usage of the encoder starts below 1 kB for small block sizes and goes up to 14 kB for large block sizes. The C-based decoding, using a small block size, only demands a couple of kB of RAM space and using higher block size takes up to 900 kB RAM space. For the TCP, the smallest block sizes only need a couple of kB, which goes up to 25 kB for the largest ones.

5.3.1 Comparison

The RAM usages from the three codings are shown together in Figure 5-11.

It is possible to see a great difference between the codings. The C-based decoding has a lot of large objects, which in turn depends on the size of the input blocks, making it more memory consuming. When using the TCP, these large objects seen in the C-based algorithm, are saved in the coprocessor instead, making the processors RAM usage much smaller. It is also easy to see that the increasing of the block size has a very large impact on the RAM usage.

When using C-based coding, the RAM usage of the encoding would not have a very large impact since the RAM usage of the C-based decoding is a lot greater. The RAM usage of the TCP and the encoding are approximately the same, making the total RAM usage about twice as high.

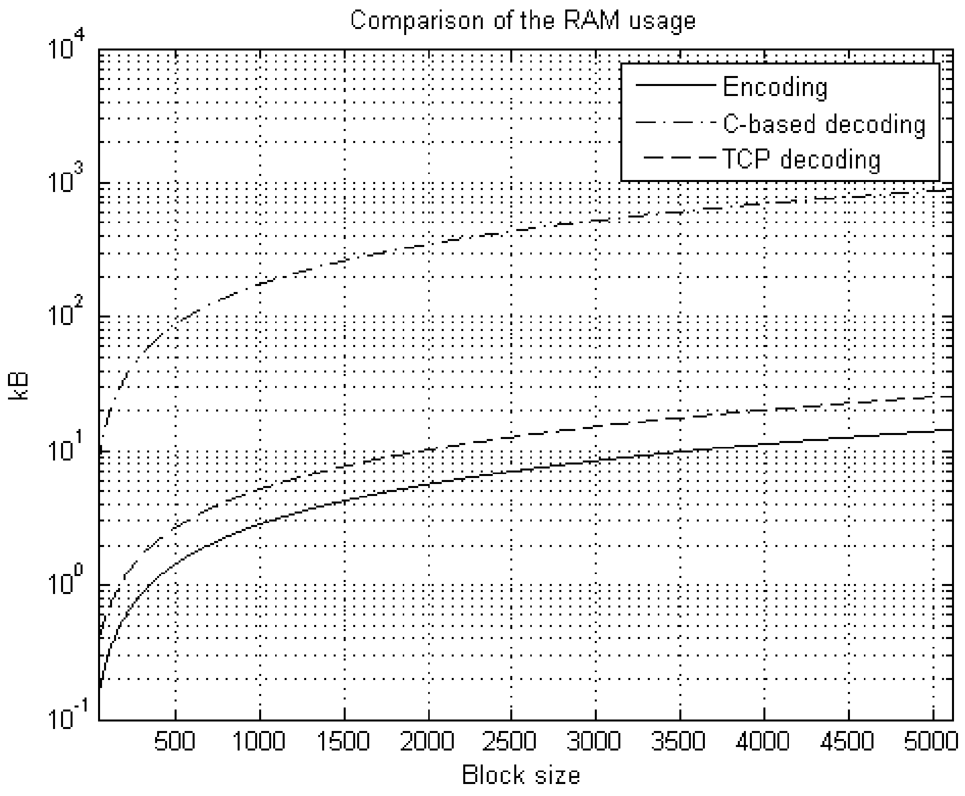

Figure 5-11: Comparison of the RAM usage.

5.4 Total Memory Usage

Combining the results of the code density and the RAM usage for the two decoding methods and including the use of the encoder gives a total memory usage as illustrated in Figure 5-12 and 5-13. The RAM usage is optimization independent, which is not the case for the code density and these results have been calculated using no optimization.

For the C-based decoder, it is easy to see that the code density is only a small part of the total memory usage, at least for large block sizes. The code density of the TCP coding influences the total memory usage a lot more, since the RAM usage is not as high as the C-based coding.

The C-based coding has a total memory usage starting just below 100 kB and going up to almost 1000 kB, which is almost the same as the RAM usage of the C-based decoder. For the TCP, the total memory usage goes from 35 kB for small block sizes and up to 70 kB.

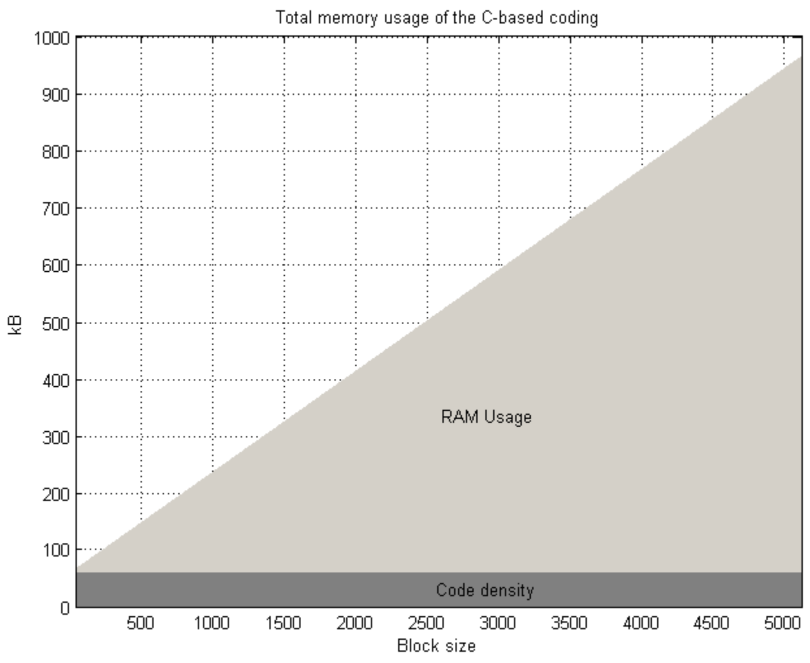

Figure 5-12: Total memory usage of the C-based decoding with the encoding included.

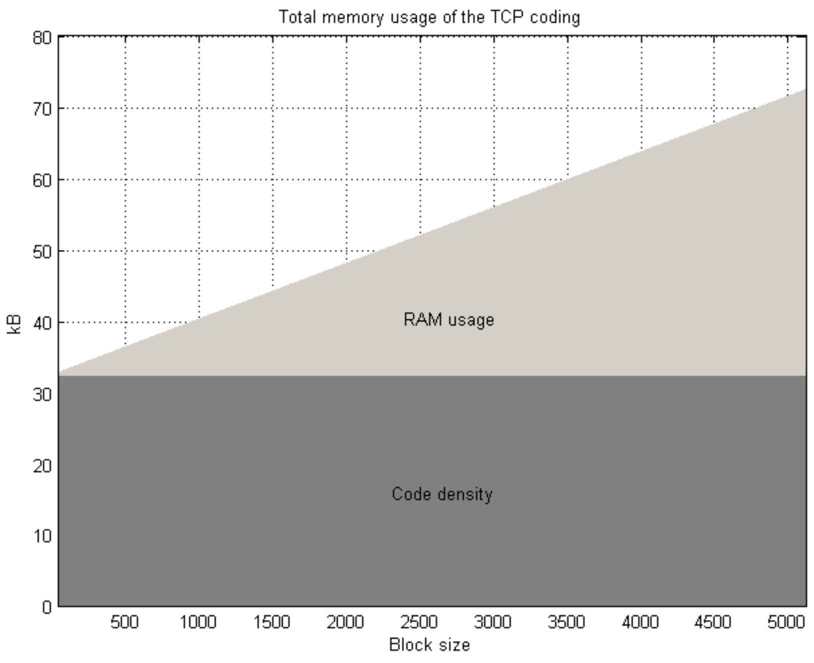

Figure 5-13: Total memory usage of the TCP with the encoding included.

5.4.1 Comparison

Figure 5-14 gives a comparison of the total memory usage of the C-based coding and the TCP coding, when both is including the memory usage of the encoding.

As can be seen, both codings have roughly the same total memory usage when using small block sizes. Increasing the code size makes the C-based decoding a lot greater. It is hardly any doubt that the TCP coding has the lowest overall memory usage.

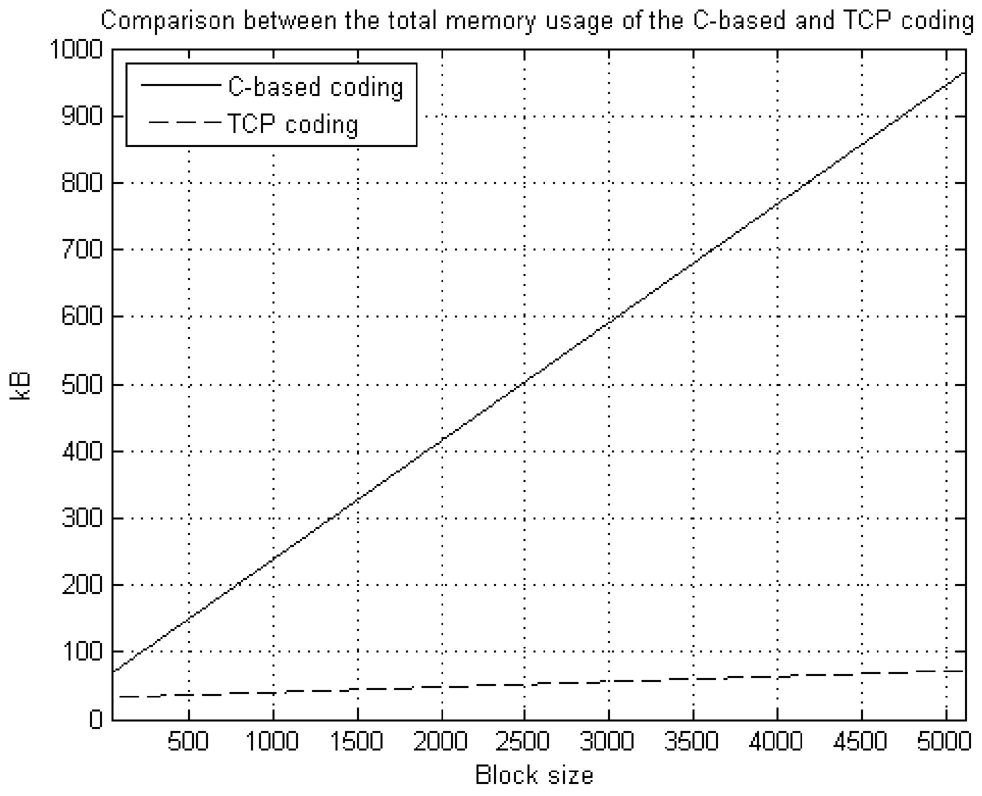

Figure 5-14: Comparison between the total memory usages of the two coding.

5.5 Data Rate

The data rate might be dependent on the optimization, the number of iterations, the block size and the number of blocks. All of the following results have been collected with the help of the Profiler in CCS. Using the reasoning earlier, the different bit rates where calculated, using *f*=600 MHz, from the number of clock cycles as:

$$R = \frac{\#\text{bits}}{\#\text{clock\_cycl es}} f \ .$$

The results are all given in kbit/s.

5.5.1 Optimization

By encoding and decoding a block of 40 bits using different optimization levels, Figure 5-15 to 5-17 are produced that show the resulting bit rates for the encoding, C-based decoding and the TCP respectively.

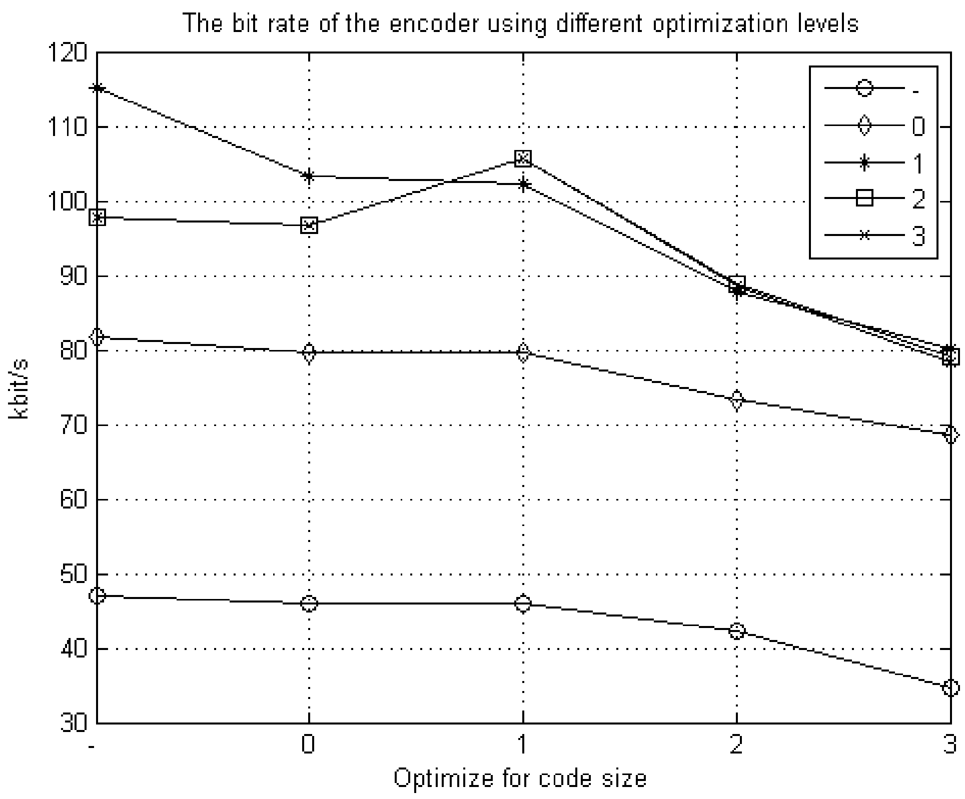

Figure 5-15: The bit rate of the encoder using different optimizations.

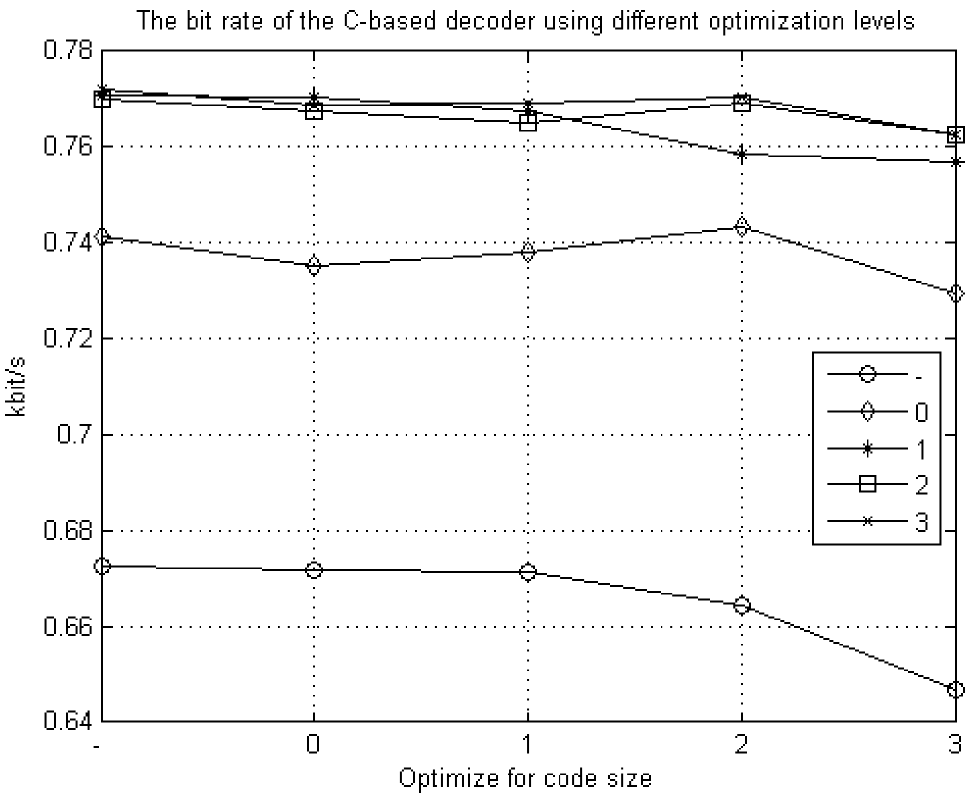

Figure 5-16: The data rate of the C-based decoder.

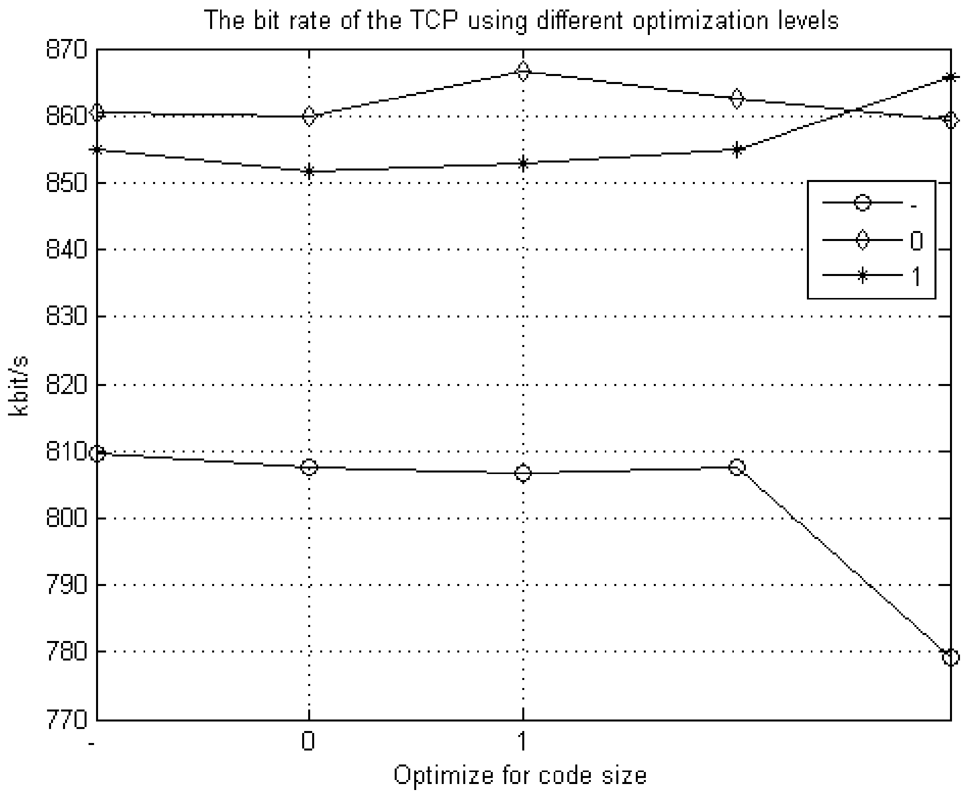

Figure 5-17: The data rate of the TCP decoding using different optimization levels.

Using a higher optimization level increases the bit rate, first a lot when going from no optimization to level 0, but for the highest levels there is not much difference between the results. The optimization for code size does not affect the code rate especially much in either direction; instead it looks a bit random if increasing this would give a bit higher or a bit lower data rates.

The highest bit rate of the encoder is almost 120 kbit/s, while the lowest notation is below 40 kbit/s. The C-based decoder could get as high as 0.78 kbit/s and as low as 0.65 kbit/s.

For the TCP, the optimization levels 2 and 3 corrupted the decoding and the TCP never returned any hard decisions. For the optimizations where it worked, the bit rate where between 780 and 865 kbit/s.

5.5.2 Blocks

To check whether the choice of block size and number of consecutive blocks influence the data rate, two tests were performed, one based on different block sizes and the other on different number of blocks.

5.5.2.1 Block size

In the first test, different block sizes where encoded and decoded, using only one block. The code was built without optimization and the decoding uses one iteration. The result is given for the encoder, C-based decoder and TCP in Figure 5-18 to 5-20.

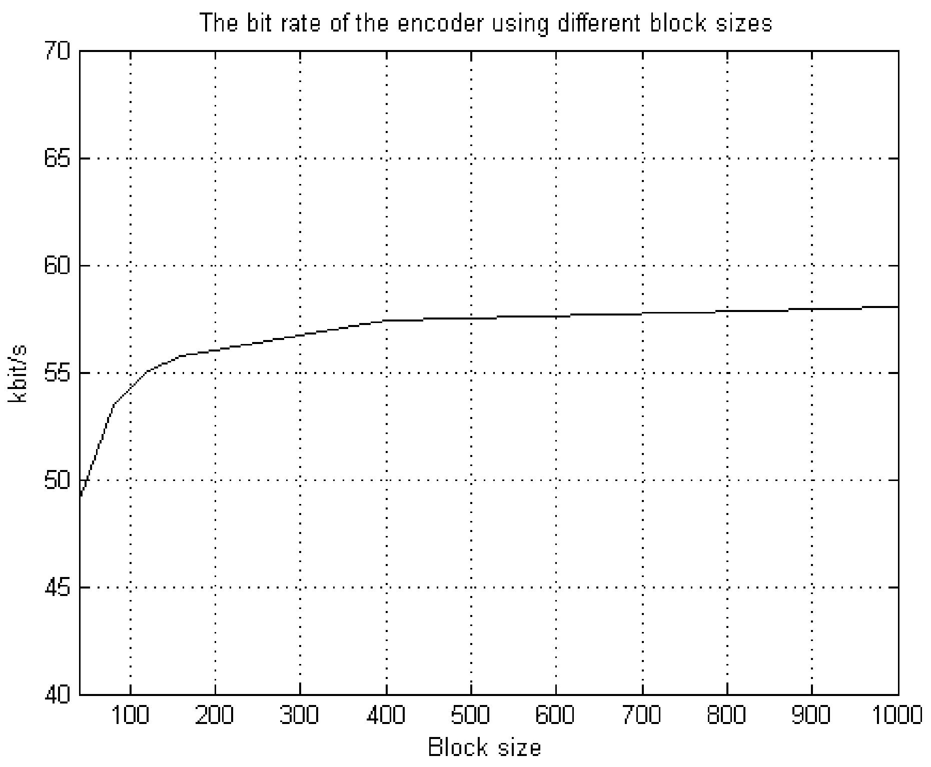

Figure 5-18: Bit rate of the encoder using different block sizes.

The encoding gets a slightly increased bit rate by increasing the block size. This indicates that the initializing and closuring of the encoding affects the smaller blocks more than the larger blocks. The encoding of each bit is probably equally fast, regardless of the block size.

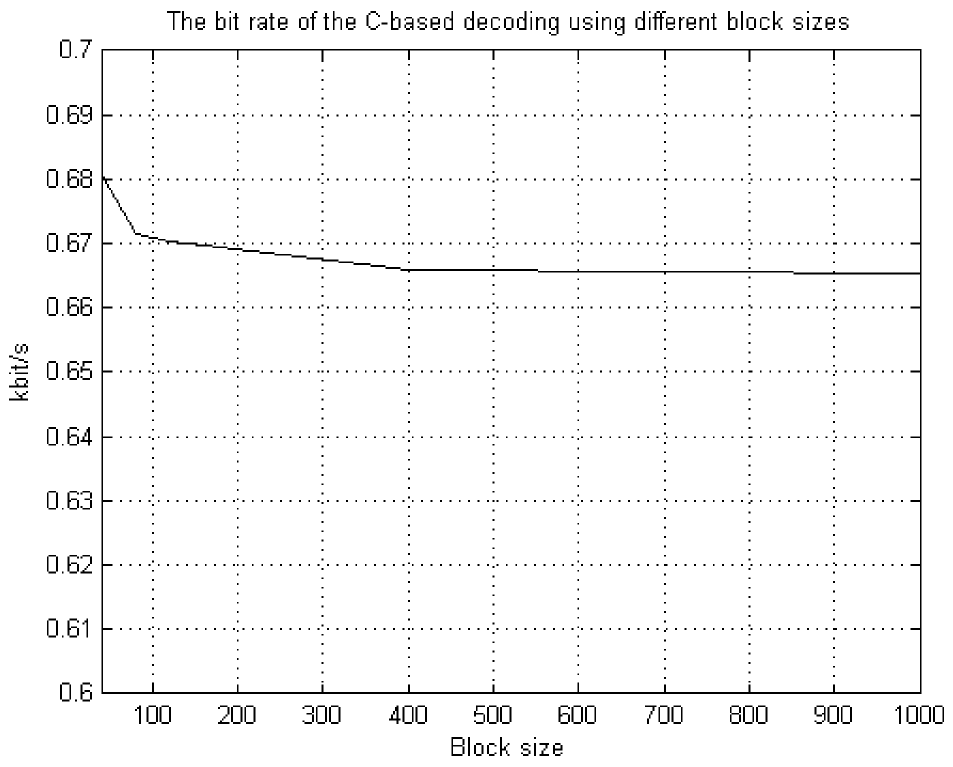

Figure 5-19: Bit rate of the C-based decoder using different block sizes.

Looking at the result from the C-based decoder it is possible to see a slight decrease in the bit rate when going from smaller to a bit larger block sizes. Otherwise the bit rate is almost constant. This has probably to do with the fact that the decoding is very heavy, which does not make the initialization and closuring influence that much. The reason why it is a small decrease when increasing the block size is unclear. A guess would be that the reading of all the large vectors used in the decoding gets slower for larger vectors.

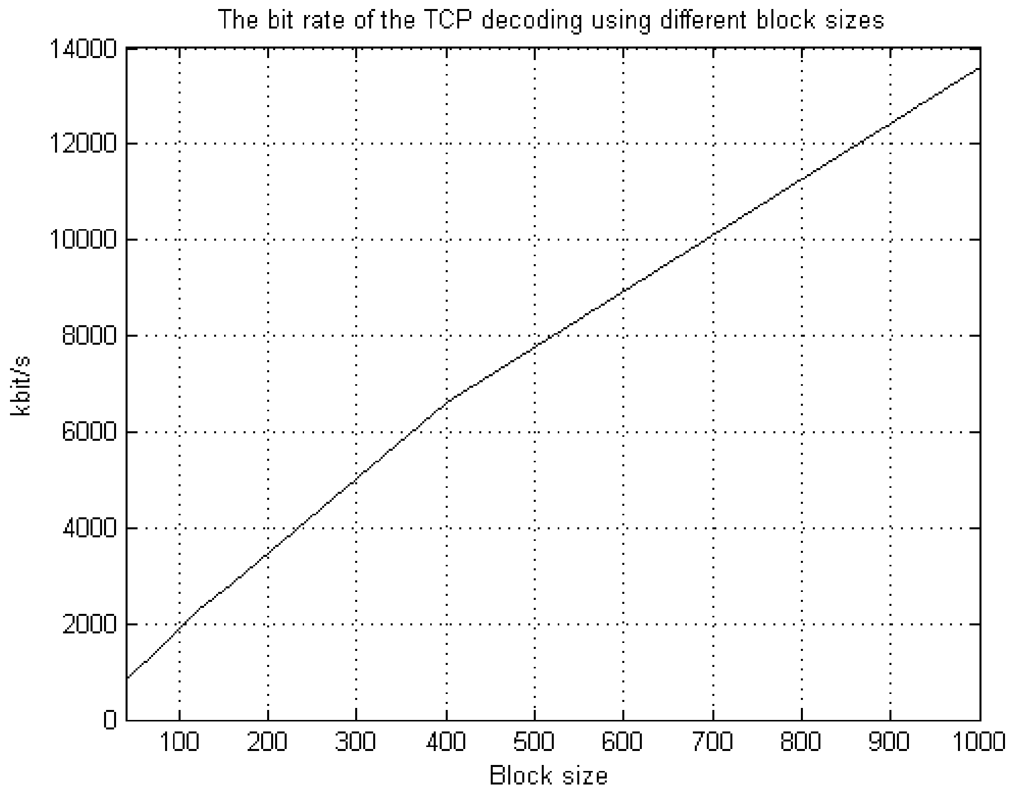

Figure 5-20: Bit rate of the TCP using different block sizes.

The TCP is the one that gains the most from using larger block sizes. The reason for this must be that the coprocessor is really fast regardless of how many bits it needs to decode. What slows down the TCP for smaller block sizes is the initialization and closuring of the decoding.

5.5.2.2 Number of blocks

In the second test, different number of blocks where encoded and decoded, using a block size of 40 bits. The code was built without optimization and the decoding uses one iteration. The result is given for the encoder, C-based decoder and TCP in Figure 5-21 to 5-23.

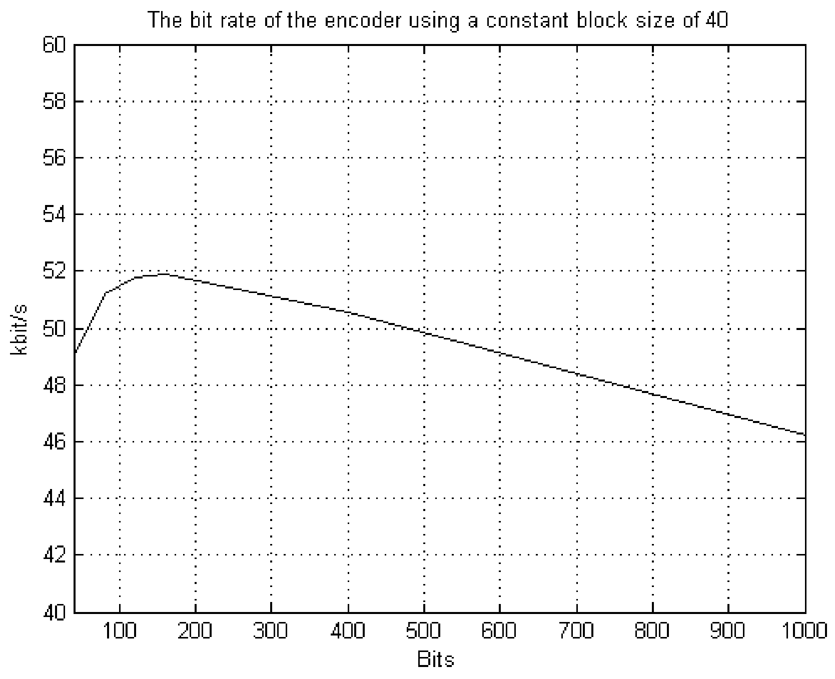

Figure 5-21: Bit rate of the encoder using a constant block size of 40.

This result indicates that something is slowing the encoder down when it is fed too many blocks. Since this is a loop, running through all the blocks, it must be something inside this loop and the only factor that separates the different runs from each other is a copying of the input bits to a temporary input vector. A guess would be that this copying is slowing the encoder down.

The increase of the bit rate that is seen for small block sizes is more expected and if it were not for the vector copying, this increase would probably have continued.

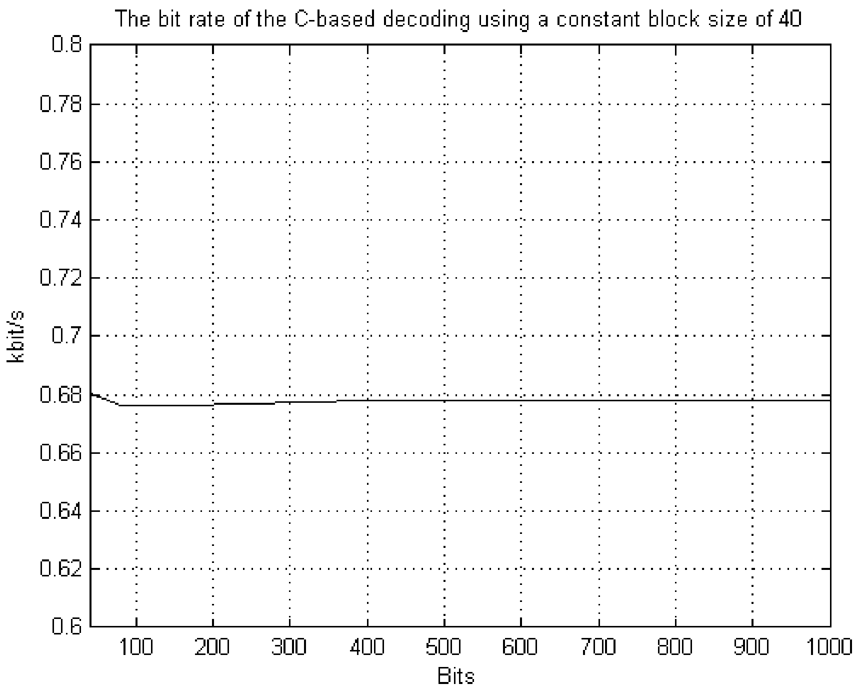

Figure 5-22: Bit rate of the C-based decoder using a constant block size of 40.

The data rate of the C-based decoding is not affected by how many blocks that are fed into the decoder. This again indicates that the actual decoding is the part that is slowing down this decoder.

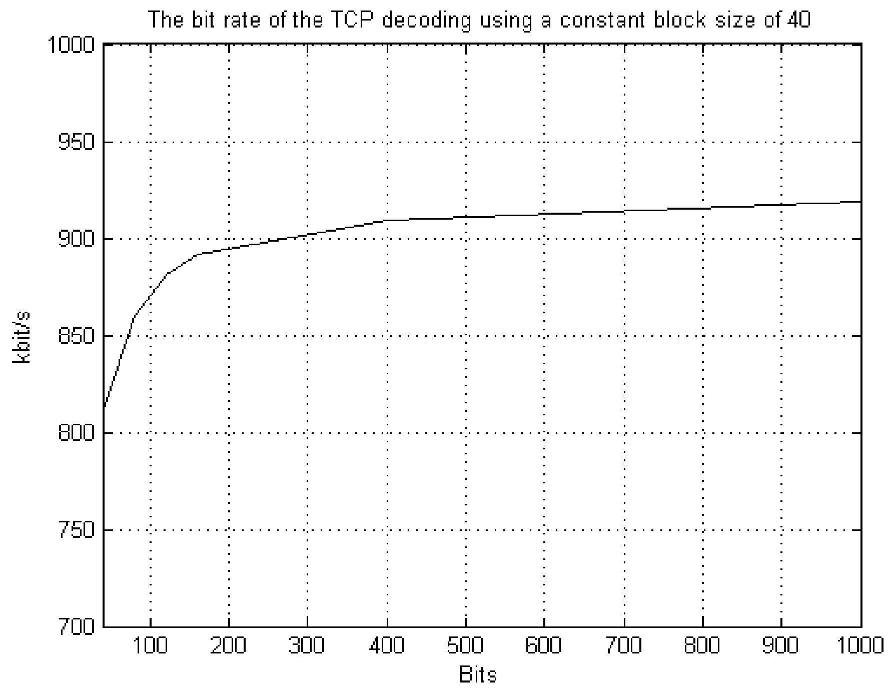

Figure 5-23: Bit rate of the TCP using a constant block size of 40.

The TCP gains a bit from using several blocks. Again this indicates that the initialization and closuring of the decoding is the slow part of the TCP and not the work done by the coprocessor.

5.5.3 Iterations

To see how much impact the number of iterations has on the bit rate, one block of 40 bits where decoded using 1, 2, 3, 4 and 5 iterations. Figure 5-24 and 5-25 shows the bit rate of the C-based decoder and the TCP when no optimization has been used. An initial guess would be that 2 iterations would take approximately two times as long as one.

The result from the C-based decoder follows the initial guess and the bit rate is roughly halved each iteration. The data rate of the TCP is of course also decreased for each iteration but not especially much. This, one more time, indicates that it is the initialization and closuring done at the processor prior to the decoding and not the coprocessor that is time consuming.

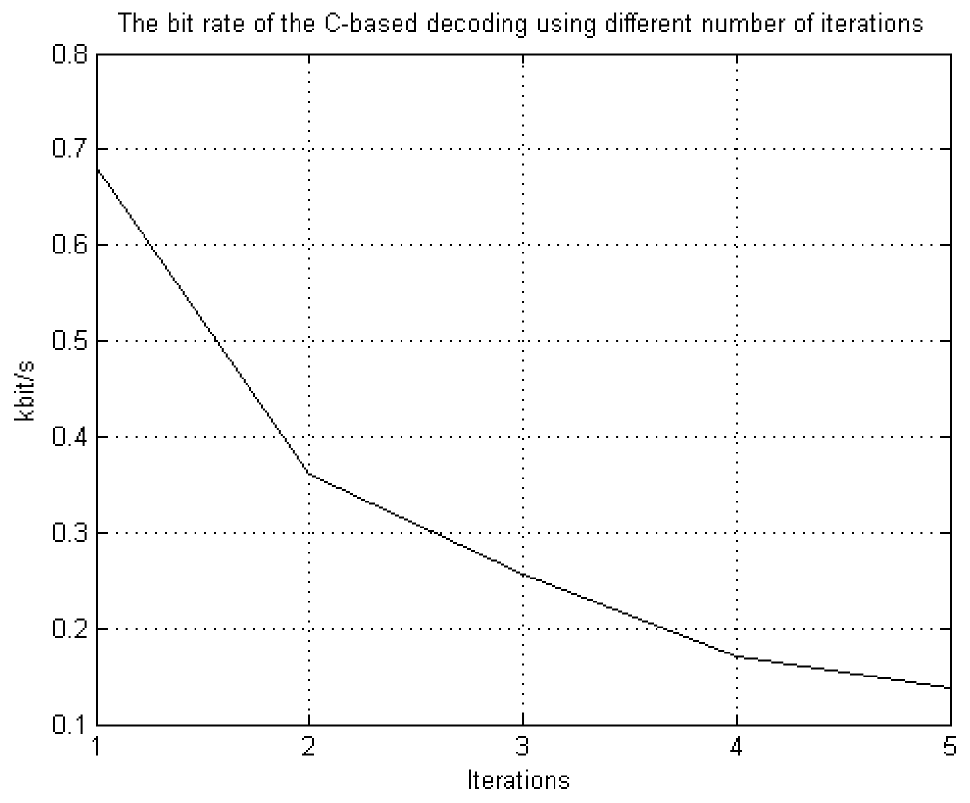

Figure 5-24: Bit rate of the C-based decoder using different number of iterations.

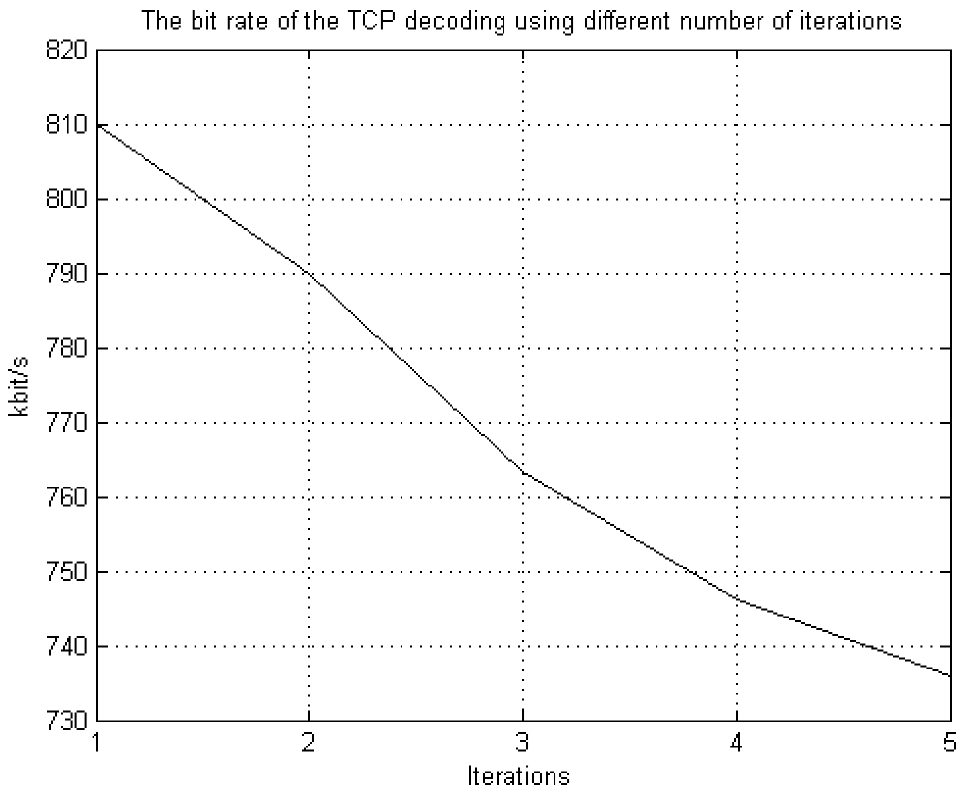

Figure 5-25: The bit rate of the TCP using different number of iterations.

5.5.4 Comparisons

5.5.4.1 Optimization

When using different optimizations the TCP has the highest bit rate, approximately 10 times faster than the encoder and approximately 1000 times faster than the C-based decoder.

5.5.4.2 Blocks

Looking at the different block constellations, it is only the TCP that is gaining on using both larger block sizes and several blocks. The encoder and the C-based decoder are also influenced but the difference is not very large. The C-based decoder seems to have a constant bit rate of 0.68 kbit/s using this set up, the encoder ends up around 50 kbit/s, while the TCP could increase to almost 14 Mbit/s only by using a block size of 1000 bits.

5.5.4.3 Iterations

Notable here is that while the bit rate of the C-based decoding is almost halved each iteration, this is not the case for the TCP. The bit rate of the C-based decoding drops from 0.68 to 0.15 kbit/s, when increasing the number of iterations from 1 to 5. The TCP instead drops from 810 to 736 kbit/s. Recall that the bit error rate performance when using only one iteration is quite bad and that the decoding would need a couple of iterations to be really useful.

5.5.4.4 Total

One important fact to remark is that a coding system is only as fast as its slowest link, i.e. when using the C-based decoding with the encoding the data rate can not be higher than that of the C-based decoding; otherwise the incoming bits to the decoding will be over flooded. Using the TCP, it is instead the encoding which seems to be the bottleneck.

The encoding could be both optimized and make use of larger block sizes to get a bit rate above 100 kbit/s. For the C-based decoding, it is hard to gain anything except from the optimization, giving it a maximum bit rate of less than 1 kbit/s, when only using one iteration. The TCP could gain the most from using larger block sizes making it able to have a bit rate above 1000 kbit/s, when using several number of iterations.

5.5.5 Max-log-MAP algorithm

Using the faster but more inaccurate Max-log-MAP algorithm would instead give the bit rate shown in Figure 5-26, where 1000 bits have been decoded using different block sizes. The number of iterations is one and it has been built using level 1 optimization and no optimization for code size.

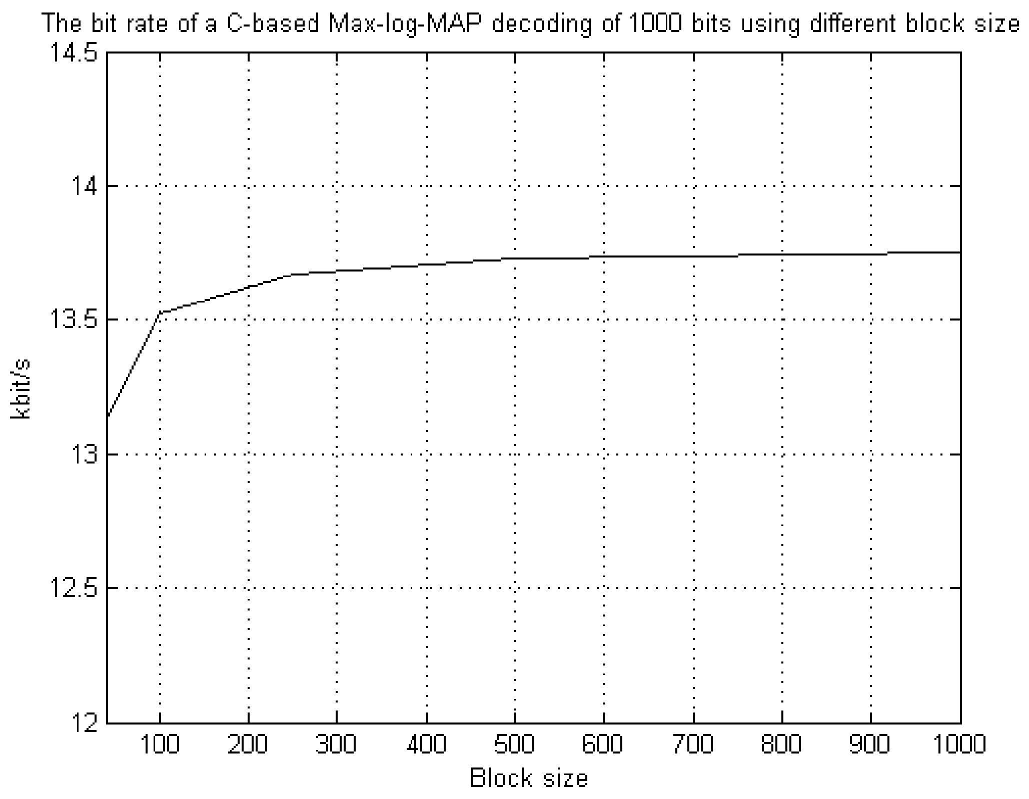

Figure 5-26: The bit rate of the C-based decoding using the Max-log-MAP algorithm.

The gain in using the Max-log-MAP decoding is that the bit rate gets increased roughly 10 times by changing the max*-function to the faster max-function, but it is still slower than both the encoding and the TCP decoding.

Chapter 6

Improving the Data Rate

6.1 Introduction

The results from section 5.5.2.2 indicated that the use of vectors is slowing the encoder down. Also when using a bit vector, all the bits are saved in 32-bit words, meaning that there are a lot of bit operations, which are slow compared to using larger data types. Therefore, a test was created, where the implementation was adjusted so that all the bit-vectors where changed to char-arrays. Of course only one of the two modifications could be performed independently to increase the data rates.

It is clear that the encoder is the coding with most bit vectors and should also be the one gaining the most from the new set. The cost for this modification will of course be a larger RAM usage. The question is, if the gain in data rate is enough to compensate the increase of memory usage.

To see whether the data rate is increased or not, the same test same as in section 5.5.1, was created, i.e. testing how the coding responds to the different optimizations. The result is given in Figure 6-1 to 6-3.

Comparing these figures to Figure 5-15 to 5-17, the coding gain could be seen readily. The encoder improves from about 100 kbit/s to almost 5000 kbit/s, the C-based decoder from just below 0.8 kbit/s to slightly above and the TCP from a peak of 860 kbit/s to over 900 kbit/s. It is also obvious that the encoder is the only one with a really large gain. Therefore the performance of the encoder will be deeper examined in the following sections.

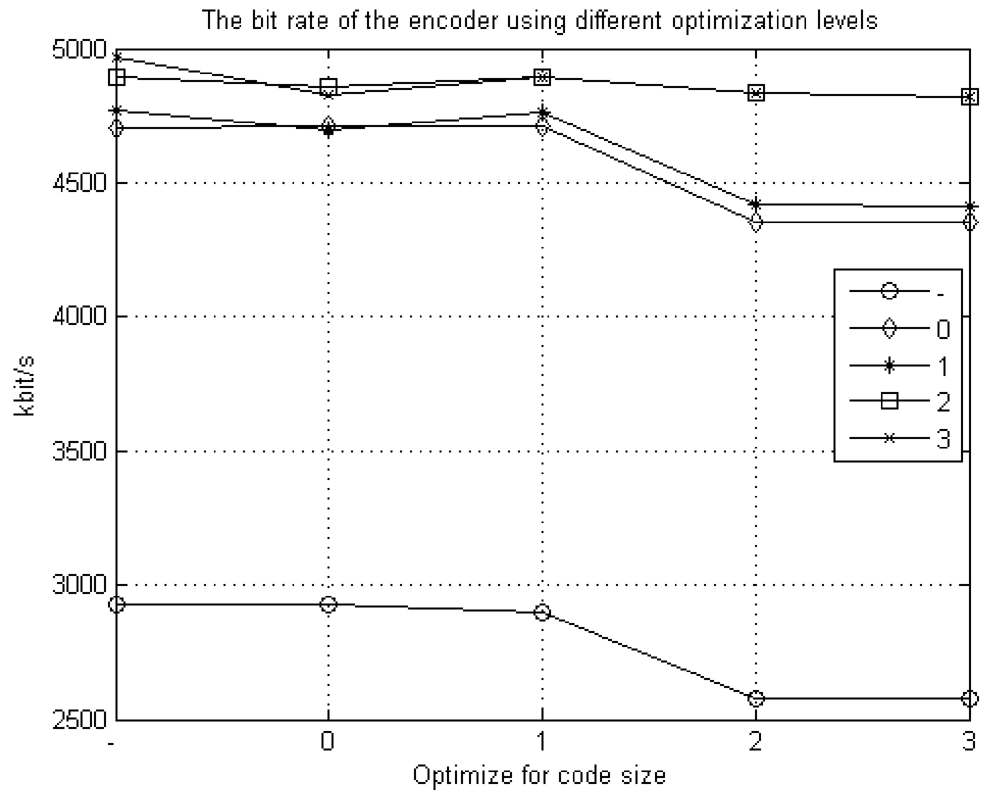

Figure 6-1: The new data rate of the encoder when using different optimizations.

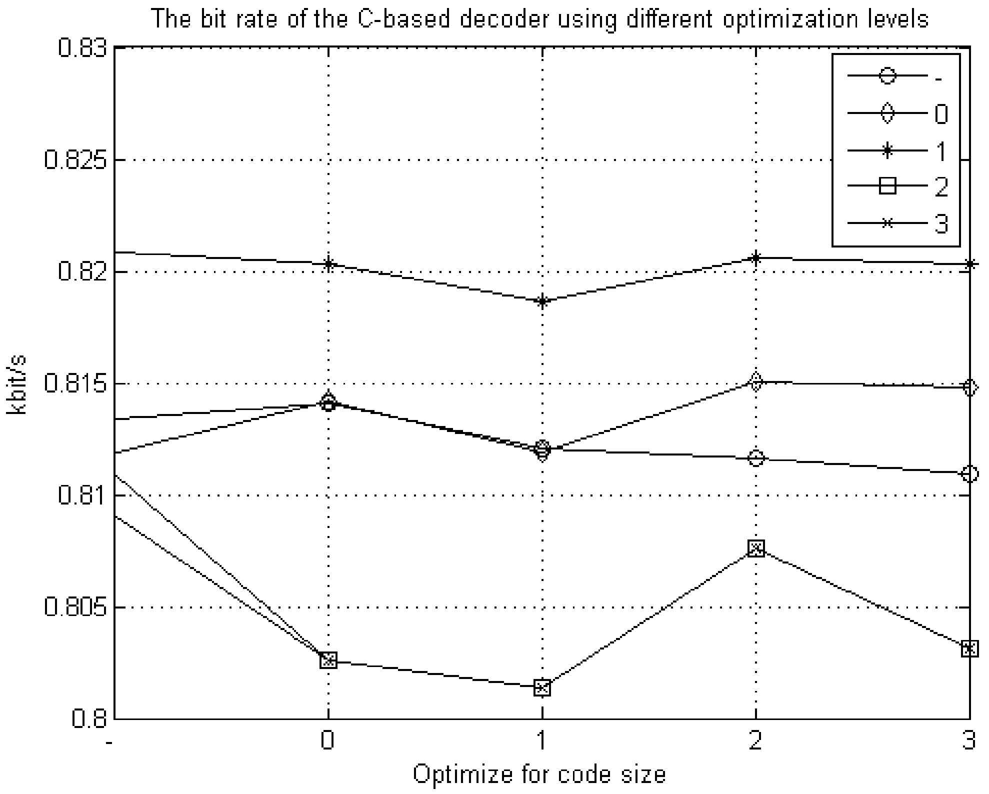

Figure 6-2: The new data rate of the C-based decoder when using different optimizations.

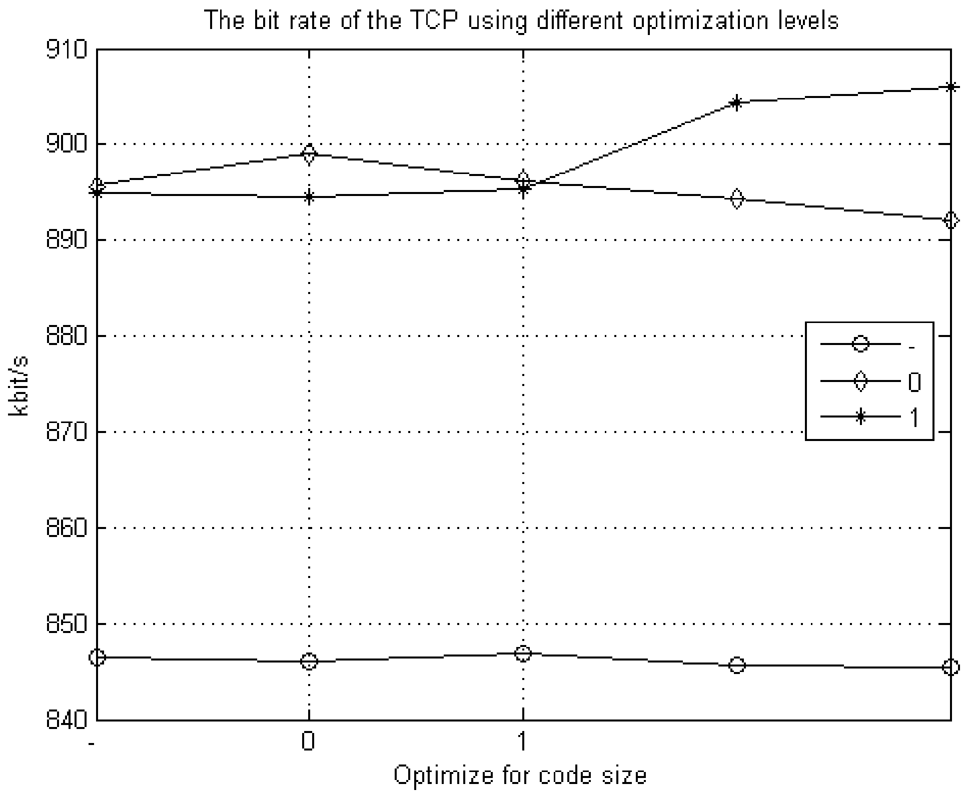

Figure 6-3: The new data rate of the TCP when using different optimizations.

6.2 Code density

The same test as in section 5.2.1, using different optimizations, was performed for the encoder giving the result shown in Figure 6-4. Apparently the code density of the encoder gets reduced by approximately one third compared to the previous encoder.

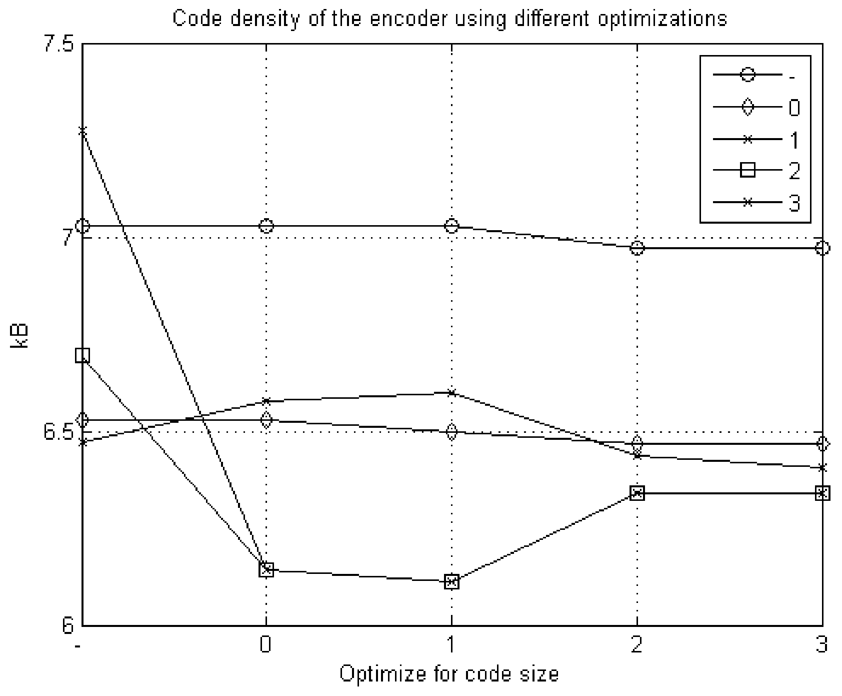

Figure 6-4: The new code density of the encoder at different optimization levels.

6.3 RAM Usage

This is the part where the cost of this data rate improvement is paid. The different object from the encoding, listed in Table 5-1 earlier, is again listed in Table 6-1 with their new data types.

Name	**Data Type**	**Size**
Input	Char (1 Byte)	l_B
Output	Char (1 Byte)	$3l_B + 12$
Interleaving sequence	Short (2 Bytes)	l_B
State transition table	Char (1 Byte)	16
Output parity table	Char (1 Byte)	16
Interleaved input	Char (1 Byte)	l_B
Parity bits from first rscc	Char (1 Byte)	$l_B + 3$
Parity bits from second rscc	Char (1 Byte)	$l_B + 3$
Tail from first rscc	Char (1 Byte)	3
Tail from second rscc	Char (1 Byte)	3

Table 6-1: The objects in the new encoding which uses dynamic memory allocation.

From this it is possible to derive the following expression for the RAM usage of the encoder:

$$\mathrm{RAM}_{\mathrm{ENC}} = 8 * l_{\mathrm{B}} + 56 \; [\mathrm{B}].$$

This is illustrated in Figure 6-5, where it is possible to see an increase of the RAM usage compared to Figure 5-8 for the previous encoder.

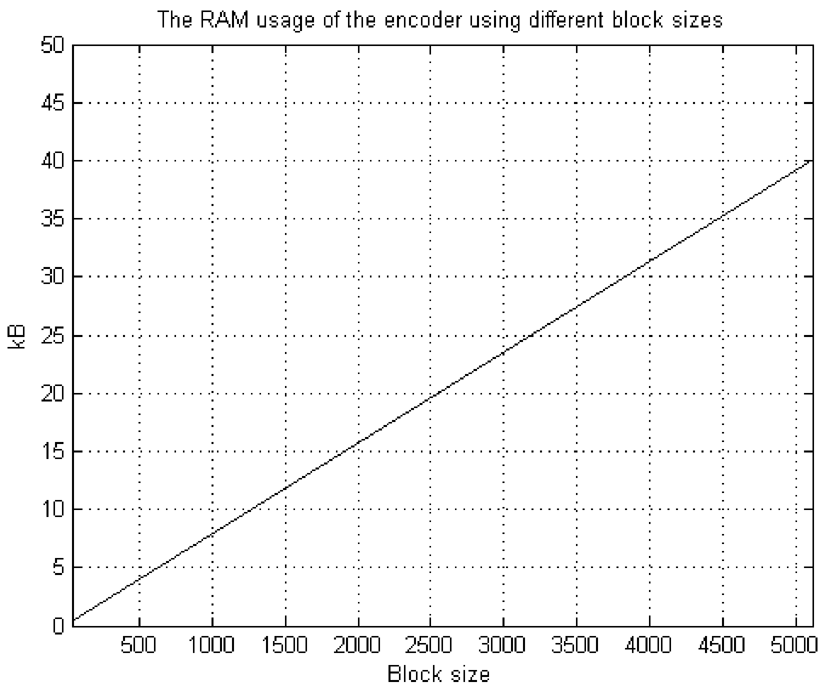

Figure 6-5: New RAM usage of the encoder.

6.4 Total memory usage

The totally memory usage of the encoder is gathered and compared to the previous result in Figure 6-6, where no optimization is included.

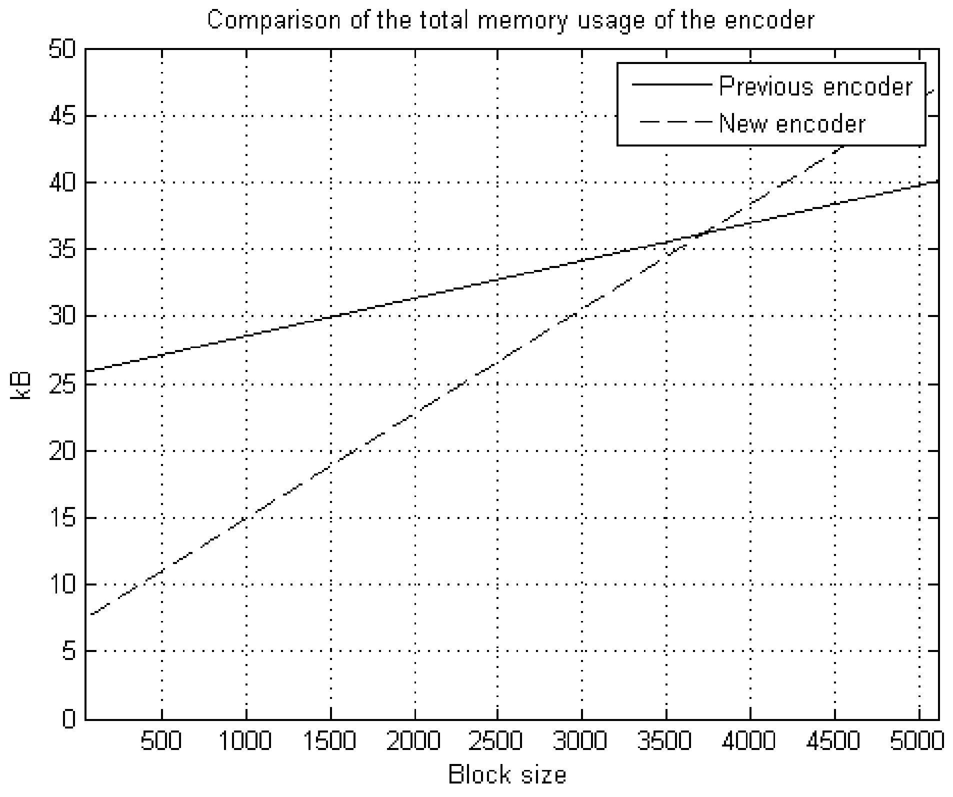

Figure 6-6: Comparison of the previous and new encoders total memory usage.

The code density of the previous encoder is that much larger than that of the new encoder, giving a larger total memory usage for block sizes of less than 3700 bits. This is true regardless of that the new encoder has a lot larger RAM usage. A guess is that, even though the memory used by the vectors is allocated from the RAM, the creation of a vector that is included in the code density is memory consuming as well. It has probably to do with their need for overhead information.

6.5 Data Rate

The result when using different optimizations has already been presented. The dependency of using different block sizes and number of blocks is instead given here in Figure 6-7 and 6-8. Using this new set gives a gain in using both larger block sizes and several numbers of blocks. This differs from the previous encoding as can be seen in section 5.5.2.

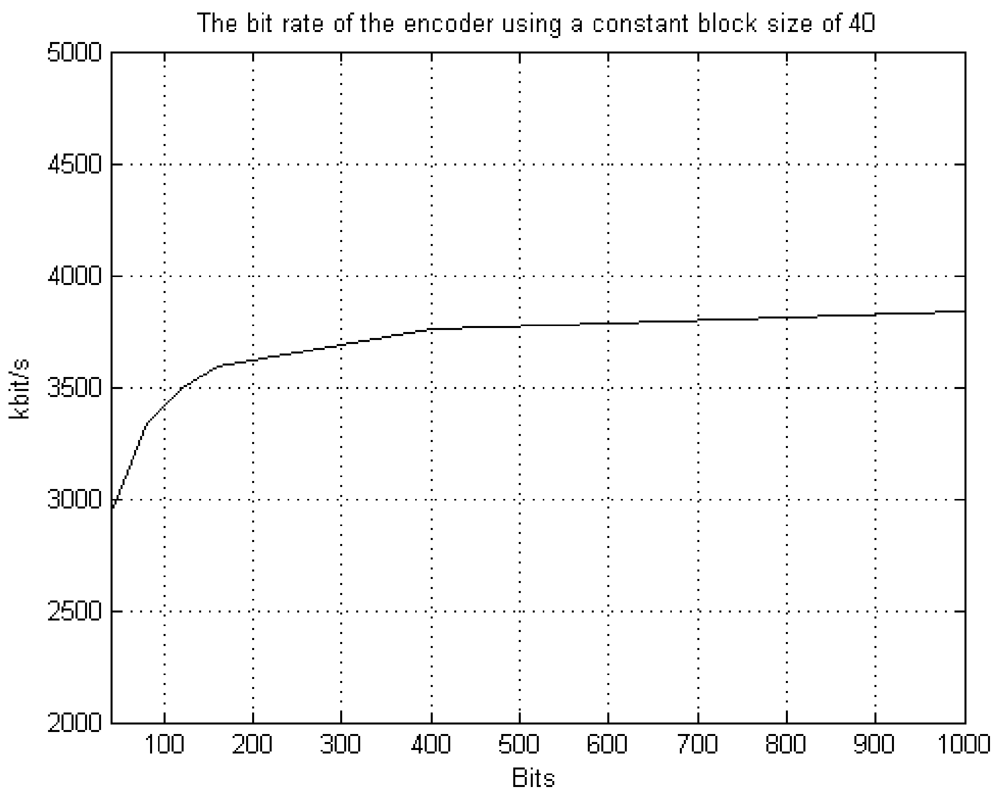

Figure 6-7: The new data rate of the encoder using a constant block size.

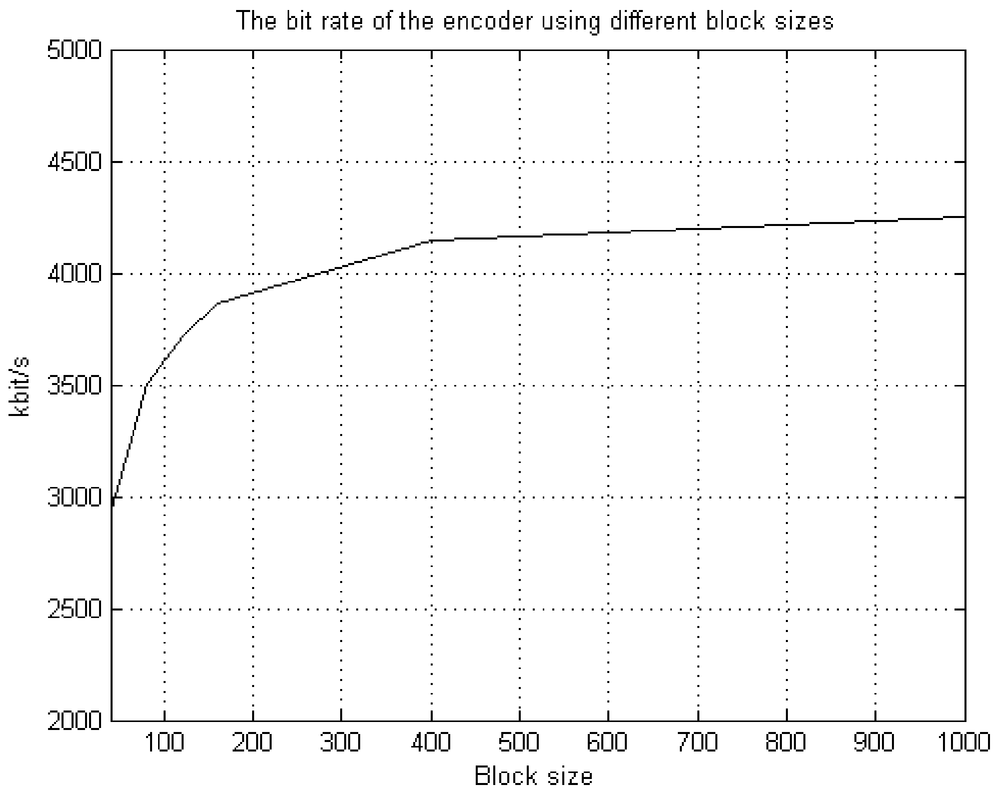

Figure 6-8: The new data rate of the encoder using different block sizes.

6.6 Total System

The C-based implementation was previously bounded by the speed of the decoder. This could be improved a bit but not particularly much. The TCP on the other hand was bounded by the speed of the encoder. With this new improved encoding this is no longer the case.

The encoder is about as fast as the TCP giving an opportunity to optimize them both to get a good result. Since the TCP can get really fast using large block sizes, Table 6-2 shows the numbers of decoding iterations allowed at different block sizes to make the encoding and TCP equally fast. The data rate here is, when using optimization, above 8 Mbit/s.

# Iterations	**Block Size**
1	440 bits
2	500 bits
3	550 bits
4	600 bits
5	650 bits
6	800 bits
7	900 bits
8	1100 bits
9	1300 bits
10	1450 bits
11	1800 bits
12	2300 bits
13	3000 bits
14	4000 bits
15	5114 bits

Table 6-2: The numbers of decoding iterations allowed at different block sizes.

Chapter 7

Conclusions

7.1 Memory

Using the TCP would give roughly half the code density compared to the C-based decoding but neither of the code densities is especially high. Compared to the, for the C-based decoder, larger RAM usage the code density will not occupy any greater part of the total memory.

Since turbo coding requires a lot of memory, the TCP, which uses external memory during the decoding, will most likely use less RAM than the C-based decoding. By looking at the results there is not any big difference when using the smallest block size. But the difference increases a lot, already after a small increase in the block size.

Using the new set of the encoding will not affect the total memory usage especially much. Depending on the block size it will be either a bit decreased or increased.

7.2 Data Rate

The evaluation shows that it is possible to run the encoding and C-based decoding in a bit less than 1 kbit/s. Using the encoding and the TCP instead, the bit rate could be approximately 100 kbit/s, here the encoding is the bottleneck. Improving the data rate of the encoder gave a bit rate of a couple of Mbit/s and together the encoding and TCP could run, using several decoding iterations, in as high as 8 Mbit/s.

7.3 Answers

Given the questions in the thesis objectives these are the answers that the evaluation has come up with:

- Is it efficient in terms of memory usage and execution speed to use turbo coding on a C64x DSP without using the TCP?

It is not inefficient in terms of memory usage to use the C-based algorithm for turbo coding, especially if a small block size is chosen. It is however definitely inefficient to use the decoding in terms of execution speed. A wireless application, which runs in a bit rate below 1 kbit/s, would not last very long, despite how good error correction it provides.

- How much memory usage and speed could be gained by using the TCP instead of the C-based algorithm, if any?

The gain of the memory usage depends on the block size. For a block size of 40 bits, the gain is approximately 30 kB. Using 5114 bits as block size instead would give a total memory gain of approximately 900 kB.

The data rate of the TCP is over 1000 times faster than the C-based decoding, which is a large gain. Instead of running in less than 1 kbit/s it is possible to reach a couple of Mbit/s.

- Is it practically applicable to use the TCP for turbo decoding on a C64x DSP?

Using a total memory usage of less than 100 kB and reaching a bit rate of 8 Mbit/s are enough for many kinds of wireless application. Comparing this to the different standards shown in Table 7-1, the coprocessor would be enough for GSM and with the faster encoder also 3G. However, it is still too slow to use in LTE.

GSM:	14.4 kbit/s [18]
3G:	Peak bit rates of 1 Mbit/s [19]
LTE:	Peak bit rates of 100 Mbit/s [20]

Table 7-1: Different standards and their bit rates.

Chapter 8

Future Work

The first work that would be interesting to do is to implement the TCP in a real system to see if this evaluation holds or not. Then of course it would be interesting to try to push both the encoder and the TCP to higher data rates.

Unfortunately it would be very hard to increase the data rate of the TCP. A solution might be to increase the frequency of the DSP. Another possibility to increase the speed of the decoding is to use two DSPs or one DSP containing two TCPs.

There is also a newer version of the TCP called the TCP2, which is included in some newer DSPs. It would be interesting to evaluate this decoder's performance as well, and a guess would be that it could reach even higher data rates than the TCP does.

Bibliography

[1] C. E. Shannon, “A Mathematical Theory of Communication,” *Bell Syst. Tech. J.*, vol. 27, pp. 397-423, 623-656, July/Oct. 1948.

[2] S. Lin & D. Costello, *Error Control Coding: Fundamentals and Applications*, Upper Saddle River, NJ: Pearson Prentice Hall, ISBN: 0-13-017073-6, 2004.

[3] U. Madhow, *Fundamentals of Digital Communication*, Chapter 7, New York: Cambridge University Press, ISBN: 978-0-521-87414-4, 2008.

[4] L. Ahlin, J. Zander & B. Slimane. *Principles of Wireless Communication*, Chapter 6, Lund: Studentlitteratur, ISBN: 978-91-44-03080-7, 2008.

[5] C. Berrou, A. Glavieux and P. Thitimajshima. “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes,” *(Proceedings of) IEEE International Conference on Communications, Communications*, vol. 2, 1993, pp. 1064-1070.

[6] Y. Tong, T-H. Yeap and J-Y. Chouinard. “VHDL Implementation of a Turbo Decoder with Log-MAP-Based Iterative Decoding,” *Transactions on Instrumentation and Measurement*, vol. 53, no. 4, August 2004, pp. 1268-1278.

[7] “3GPP - Organizational Partners,” web page, URL: www.3gpp.org/partners, accessed 2011-10-04.

[8] “3rd Generation Partnership Project, Technical Specification Group Radio Access Network; Multiplexing and Channel Coding (FDD),” *3GPP TS 25.212*, v. 9.3.0 Release 9, Sept. 2010. URL: http://www.quintillion.co.jp/3GPP/Specs/25212-930.pdf, accessed 2011-10-25.

[9] “TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-Point Digital Signal Processors,” Dallas Texas: Texas Instruments, 2009. URL: ttp://www.ti.com/lit/ds/symlink/tms320c6416t.pdf , accessed 2011-10-04.

[10] P. Wang, J. Yang and B. Wang. “Simple-VLIW: A Fundamental VLIW Architectural Simulation Platform,” *(Proceedings of) 7th International Conference on System Simulation and Scientific Computing*, 2008, pp. 1258-1266.

[11] J. Fisher. “Very Long Instruction Word architectures and the ELI-512,” *(Proceedings of) the 10th annual international symposium on Computer architecture*, 1983, pp. 140-150.

[12] P. Stelling and V. Oklobdzija. “Implementing Multiply-Accumulate Operation in Multiplication Time,” *(Proceedings of) 13th IEEE Symposium on Computer Arithmetic*, 1997, pp. 99-106.

[13] “TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Reference Guide,” Dallas Texas: Texas Instruments, 2006. URL: http://www.ti.com.cn/cn/lit/ug/spru234c/spru234c.pdf, accessed 2011-10-04.

[14] “TMS320C64x DSP Turbo-Decoder Coprocessor (TCP) Reference Guide,” Dallas Texas: Texas Instruments, 2004. URL: http://www.ti.com/lit/ug/spru534b/spru534b.pdf, accessed 2011-10-04.

[15] “IDEs including CCStudio,” web page, URL: http://www.ti.com/lsds/ti/dsp/support/dev_tool/ccs_overview.page, accessed 2011-10-04.

[16] “SourceForge.net: Project itpp – About,” web page, URL: sourceforge.net/apps/wordpress/itpp/about/, accessed 2011-10-04.

[17] “TMS320C6000 Chip Support Library API Reference Guide,” Dallas Texas: Texas Instruments, 2004. URL: http://archer.ee.nctu.edu.tw/class/dsplab/07s/reference/spru401i.pdf, accessed 2011-10-04.

[18] “Functionality in early GSM releases,” web page, URL: http://www.3gpp.org/article/ functionality-in-early-gsm, accessed 2011-10-04.

[19] “Overview of 3GPP, Release 7,” v. 0.9.14, June 2011, downloadable from URL: http://www.3gpp.org/ftp/Information/WORK_PLAN/Description_Releases/, accessed 2011-10-04.

[20] Beming, P et. al., “LTE-SAE Architecture and Performance,” Ericsson Review No. 3, 2007. URL: http://www.ericsson.com/ericsson/corpinfo/publications/review/2007_03/files/5_LTE_SAE.pdf, accessed: 2011-10-04.