Playing with the BEAST

Efficient Error Control Coding

using the

Cell Broadband Engine Architecture

Daniel Johnsson
Fredrik Bjärkeson

Department of Electrical and Information Technology
Lund Institute of Technology

Advisor: Florian Hug

January 11, 2010

Printed in Sweden
E-huset, Lund, 2010

Abstract

Information coding is used in communication systems to reliably transfer data from one point to another. The focus of this master thesis is to aid in the searching of new codes used in channel coding. Codes are ranked using property metrics and finding new and better codes improves communication as more errors can be corrected. The new codes are found using algorithms optimized for the Cell Broadband Engine Architecture, emphasizing the Bidirectional Efficient Algorithm for Searching code Trees (BEAST), and taking full advantage of the parallelism facilitated by the architecture.

Acknowledgements

We are now done with our thesis and would like to take the opportunity to thank Florian Hug, our supervisor, for our many discussions, his guidance and help in writing this thesis. Also, we would like to thank our examiner Martin Hell for his guidance and support, as well as for his involvement in this thesis. We would also like to thank Rolf Johannesson for proof-reading and also the staff and PhD students at the department of Electrical and Information Technology for providing us with a friendly and stimulating workplace.

Special thanks to everyone who has supported me during my work on this thesis.

Fredrik

Last but not least, to my dear Caroline, parents and sister, thank you for your support and encouragement without which I would not be half way of where I consider myself to be today. Filip Olsson, thank you for our many discussions and for your assistance in conjuring the iterative method described in Section 5.3.5.

Daniel

Lund, January 11, 2010

Contents

Chapter 1

Introduction

Digital transmission and storage systems demand fast and reliable data transfer from one point to another. Figure 1.1 illustrates a simple model of a communication system, with information source and destination.

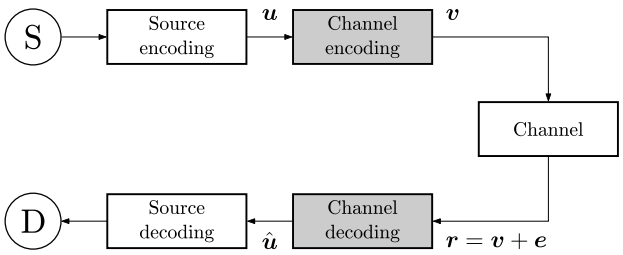


Figure 1.1: Simple model of a communication system where the source S transmits information to the destination D. In this thesis, focus is on the grayed boxes.

In many applications the raw source information is initially compressed (lossless) which is illustrated by the source encoding block in Figure 1.1. Thereby, the initial raw source information is represented by a minimum number of bits, while still being reconstructable without an loss of information. The source information is then fed into a channel encoder, which can be expressed as a sequence of information *symbols* $\boldsymbol{u}$. The channel encoder adds redundancy in a controlled way and outputs a sequence of *code symbols* $\boldsymbol{v}$, which for example can be converted to an electromagnetic wave form and transmitted over a wireless channel. Different kinds of noise and errors $\boldsymbol{e}$ are introduced by the channel and change the code symbols before they are received by the channel decoding block. The purpose of the decoder is to exploit the added redundancy and try to recover the original code sequence $\boldsymbol{v}$ from the received sequence $\boldsymbol{r}$. The channel decoder outputs an information sequence decision $\hat{\boldsymbol{u}}$, which is forwarded to the source decoder. The source decoder reconstructs the original information (decompressing) and delivers it to its final destination.

Cell Broadband Engine

The microprocessor industry faces a new era where physical limitations, heat dissipation and power consumption force manufacturers toward a multi-core design. In a conventional *homogeneous* multi-core system all computational units are identical, while in a less common *heterogeneous* design the computational units have different architectural features.

The Cell Broadband Engine (designed by Sony, Toshiba and IBM) is an innovative heterogeneous approach to a multi-core system containing 9 computational units with an impressive floating-point computational power. The Cell processor was developed with an emphasis on peak performance at the expense of simple software development, and thus to fully exploit the hardware potential an expert knowledge is needed in both programming and its architectural features.

In November 2006 Sony released the PlayStation 3 (PS3) gaming console, which is equipped with a stripped down Cell processor. Although it was not meant for scientific computing, it has gained a lot of attention in the high performance computing community since the PS3 console is cost-competitive with other commodity processors. Considering the large volume of production and the price of a single gaming console, the PS3 platform makes a good candidate for building Cell-based clusters.

Reader's Guideline

In *Chapter 2* a brief introduction to error control coding is given. We also discuss important properties and their relation to performance and ability to correct errors.

In *Chapter 3* the BEAST is introduced, which is needed to determine properties for evaluating codes. To find the best codes, an exhaustive code search must be performed on a huge set of codes. Important steps involved in an exhaustive code search are discussed, such as reducing the original set of codes using the Row Distance-Sieve.

Chapter 4 introduces the Cell Broadband Engine Architecture and gives a brief description of the hardware.

In *Chapter 5* we discuss specific implementation details of the BEAST, and present two different implementations. We also give a detailed description of the Row Distance-Sieve, and other important aspects of parallelizing an exhaustive code search using multiple Cell processors.

Chapter 6 demonstrates the performance of our BEAST implementation, such as memory usage and comparison between other architectures. We present new results of encoders with properties better than previously known, obtained by exploiting discoveries made when visualizing encoders with good properties in a two-dimensional space. This graphical representation is known as a *heat* map and we refer to the encoder clustering pattern as *hot* encoders. Finally, we also list preliminary results from the exhaustive code search for memory 26.

Chapter 2
Error Control Coding

Channel coding is a way to add redundancy to the information symbols in such a way that errors can be detected and possibly corrected. The information and code symbols are taken from a source alphabet, which in most digital systems is the Galois field $GF(2)$. The code symbols can be grouped into blocks (*codewords*) or be treated as a semi-infinite *code sequence*, where the set of all codewords or sequences is called the channel *code* $\mathcal{C}$. In this thesis we only consider symbols from the Galois field $GF(2)$ that is the information and code symbols are *bits*.

2.1 Error Probability

A simple model of a communication channel is the *binary symmetric channel* (BSC) with binary inputs and outputs. A bit carried over a BSC channel is transmitted erroneously with the *crossover probability* ϵ and correctly with probability $1-\epsilon$ as illustrated by Figure 2.1. The theory behind channel coding is for example used to determine an upper bound on the error probability when transmitting information across such a channel.

For decoding, a *maximum likelihood* (ML) algorithm, the *Viterbi algorithm* [18], is used in many applications such as satellite and deep-space communications. The basic idea for ML-decoding is to output a code sequence $\boldsymbol{v}$ that maximizes the conditional probability $P(\boldsymbol{r}|\boldsymbol{v})$ on the received sequence $\boldsymbol{r}$. When designing an encoder (choosing a code $\mathcal{C}$) there are two important code properties which have a big impact on the performance of an encoder[1], the *free distance* (d_{free}) or the *minimum distance* $(d_{\min})$ and *the distance spectrum* $(n_{d_{\text{free}}+i},\ i = 0, 1, 2, \ldots)$ introduced in Section 2.6.1 and Section 2.6.5, respectively. Further analysis into the concept of communications channels and digital modulations is beyond the scope of this thesis. For our purposes it is sufficient to consider the union-bound on the burst error probability P_{B} for convolutional codes in ML-decoding on a BSC, given by

$$P_B < \sum_{w=d_{\text{free}}}^{\infty} n_w \left(2\sqrt{\epsilon\,(1-\epsilon)}\right)^w. \tag{2.1}$$

[1] It should be emphasized that a property of a code $\mathcal{C}$ does not coincide with a property of an encoder.

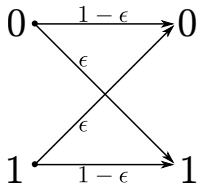


Figure 2.1: A binary symmetric channel with crossover probability ϵ.

To achieve a low burst-error probability one should maximize the free distance d_{free}, which is the dominating term, and minimize each spectral component n_w. Note, a similar upper-bound on the burst-error probability for block codes leads to the same conclusions for the *minimum distance* $d_{\min}$ and the spectral components, respectively.

2.2 Block Codes

In a block code, the sequence of information symbols is structured into independent blocks and encoded separately. Each information block $\boldsymbol{u}$ consists of a k-bit binary tuple $\boldsymbol{u} = (u_0, u_1, \ldots, u_{k-1})$. There are $M = 2^k$ different information blocks which are mapped one-to-one to individual codewords. Each codeword is an n-bit binary tuple $\boldsymbol{v} = (v_0, v_1, \ldots, v_{n-1})$. The difference $n - k$ between the number of symbols in $\boldsymbol{u}$ and $\boldsymbol{v}$ corresponds to the added redundancy and satisfies $n - k > 0$. The *rate* of a code is defined as $R = k/n$ and shows how much redundancy is added to each information symbol (i.e., the relation between the number of input bits and output bits of the encoder).

If k and n are large (i.e., there are many different codewords) there is a need to simplify the encoding process. Therefore *linear* block codes are mostly used because of their linear encoding property. In an (n, k) linear block code $\mathcal{C}$ all $M = 2^k$ codewords form a k-dimensional subspace of the vector space of all n-tuples over the field $GF(2)$. In other words, every codeword $\boldsymbol{v}$ in $\mathcal{C}$ is a linear combination of k linearly independent basis vectors $\boldsymbol{g}_i$ of length n forming a $k \times n$ *generator matrix*:

$$G = \begin{pmatrix} \boldsymbol{g}_0 \\ \boldsymbol{g}_1 \\ \vdots \\ \boldsymbol{g}_{k-1} \end{pmatrix} = \begin{pmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & & g_{1,n-1} \\ \vdots & & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{pmatrix}. \tag{2.2}$$

A linear block code is completely defined by its generator matrix G. Any set of k linearly independent vectors, out of the set of all codewords, can be used to form the rows of G, thus, the same block code $\mathcal{C}$ can be represented by different generator matrices.

Given the information sequence $\boldsymbol{u} = (\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{k-1})$ its codeword $\boldsymbol{v}$ is determined by the matrix multiplication

$$\begin{aligned} \boldsymbol{v} &= \boldsymbol{u}G \\ &= (u_0, u_1, \ldots, u_{k-1}) \times \begin{pmatrix} \boldsymbol{g}_0 \\ \boldsymbol{g}_1 \\ \vdots \\ \boldsymbol{g}_{k-1} \end{pmatrix} \\ &= u_0\boldsymbol{g}_0 + u_1\boldsymbol{g}_1 + \ldots + u_{k-1}\boldsymbol{g}_{k-1.} \end{aligned} \tag{2.3}$$

By (2.3) it follows that the all-zero information sequence $\boldsymbol{u} = \mathbf{0}$ always maps to the all-zero codeword $\boldsymbol{v} = \mathbf{0}$ which is a necessary but not a sufficient condition for a code to be linear [13].

2.2.1 Block Code Properties

A fundamental parameter of block codes is the so-called *minimum distance* $d_{\min}$ which determines the error-detecting and error-correcting capabilities of a code.

The *Hamming weight* w_{H} is defined as the number of nonzero components in a sequence of symbols. A codeword $\boldsymbol{v}$ of length n with binary symbols has the Hamming weight

$$w_{\mathrm{H}}(\boldsymbol{v}) = \sum_{i=0}^{n-1} \boldsymbol{v}_{\mathrm{i}}. \tag{2.4}$$

The closely related *Hamming distance* d_{H} between two sequences of symbols is defined as the number of positions they differ. For example, given the two codewords $\boldsymbol{v} = (1000)$ and $\boldsymbol{w} = (1011)$, the Hamming distance is $d_{\mathrm{H}}(v, w) = 2$, since the last two bits differ.

Now, the *minimum distance* of a block code $\mathcal{C}$ can be defined as

$$d_{\min} = d_{\mathrm{H}_{\min}} = \min_{\boldsymbol{v},\boldsymbol{w}\in\mathcal{C},\boldsymbol{v}\neq\boldsymbol{w}} \{d_{\mathrm{H}}(\boldsymbol{v}, \boldsymbol{w})\} = \{d_{\min}(\boldsymbol{v} + \boldsymbol{w})\}. \tag{2.5}$$

As previously mentioned, for linear block codes, the all-zero message $\boldsymbol{u} = \mathbf{0}$ maps to the all-zero codeword $\boldsymbol{v} = \mathbf{0}$, and because of this linearity (2.5) can be simplified to

$$d_{\min} = \min_{\boldsymbol{v}\in\mathcal{C},\boldsymbol{v}\neq\mathbf{0}} \{w_{\mathrm{H}}(\boldsymbol{v})\}. \tag{2.6}$$

In other words, the minimum distance of a linear block code $\mathcal{C}$ is the smallest Hamming weight of all non-zero codewords in $\mathcal{C}$.

When a codeword $\boldsymbol{v}$ of length n is transferred over a noisy communication channel the received sequence is $\boldsymbol{r} = \boldsymbol{v} + \boldsymbol{e}$ where $\boldsymbol{e} = (e_0, e_1, \ldots, e_{n-1})$ denotes the error pattern induced by the channel. The number of errors in $\boldsymbol{r}$ is the difference between $\boldsymbol{r}$ and the codeword $\boldsymbol{v}$, that is,

$$d_{\mathrm{H}}(\boldsymbol{r}, \boldsymbol{v}) = w_{\mathrm{H}}(\boldsymbol{e}). \tag{2.7}$$

Since $d_{\min}$ is the smallest number of positions in which any two codewords differ in a block code $\mathcal{C}$, every error pattern $\boldsymbol{e}$ with $w_{\mathrm{H}}(\boldsymbol{e}) \leq d_{\min} - 1$ cannot change the

received sequence $\boldsymbol{r}$ into another codeword $\boldsymbol{v}' \in \mathcal{C}$. Thus, up to $d_{\min} - 1$ of errors can be detected. Let ϵ_t be all error patterns with t or fewer errors, that is,

$$\epsilon_t = \{\boldsymbol{e} | w_H(\boldsymbol{e}) \leq t\} \tag{2.8}$$

then the block code $\mathcal{C}$ can correct all patterns in ϵ_t if and only if

$$d_{\min} > 2t.$$

This is the so called *error-correcting capability* [12] of the block code $\mathcal{C}$ and often written as

$$t \leq \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor. \tag{2.9}$$

2.3 Convolutional Codes

Convolutional codes, which were introduced by Elias [15] in 1955, is another type of code used in error-control coding. This alternative to block codes is used in many applications, such as cellular and satellite transmissions. Convolutional codes introduce the concept of encoder memory m, and as opposed to block codes, each code tuple depends not only on the current information tuple but also on the m previous information tuples. An information sequence for a convolutional code with rate $R = b/c$ is ordered into tuples of length b, which are fed into an encoder. A code sequence is then generated in a continuous fashion, ordered in tuples of length c, where b and c are typically small integers. In our thesis we will focus on the most simple and common class of convolutional codes with rate $R = 1/2$.

A linear convolutional encoder can be realized with a simple linear sequential circuit, constructed by basic memory (delay) units and XOR-gates (modulo-2 adders). For our purposes we only consider *time invariant* realizations in the commonly used *controller canonical form* without feedback, in which the delay elements form a *shift register*. Figure 2.2 shows such a realization of a rate $R = 1/2$ convolutional encoder with memory $m = 2$. At each time instance an information bit u together with m previous information bits generate two output bits $v^{(0)}$ and $v^{(1)}$, which are multiplexed into the code sequence $\boldsymbol{v}$.

2.4 The Delay Operator

Denote the impulse response of the ith input and jth output of a rate $R = b/c$ convolutional code $\mathcal{C}$ by g_{ij}. Then the $b \times c$ convolutional generator matrix G_{conv} is given by

$$G_{\text{conv}} = \begin{pmatrix} g_{11} & \cdots & g_{1c} \\ \vdots & \ddots & \vdots \\ g_{b1} & \cdots & g_{bc} \end{pmatrix}. \tag{2.10}$$

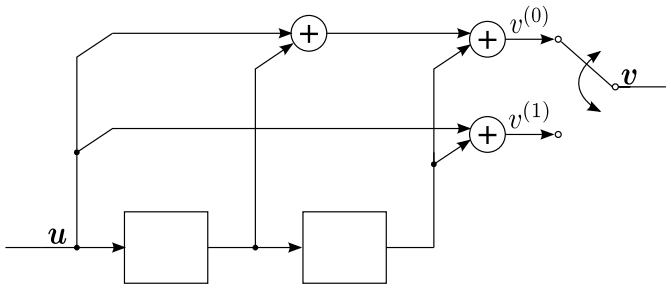


Figure 2.2: A convolutional encoder with rate $R = 1/2$ and memory $m = 2$.

Using G_{conv}, the relation between the b input sequences $\boldsymbol{u}^{(i)}$, $i = 1, 2, \ldots, b$, and the c output sequences $\boldsymbol{v}^{(j)}$, $j = 1, 2, \ldots, c$, can be expressed by as

$$\left(\boldsymbol{v}^{(1)}, \boldsymbol{v}^{(2)}, \ldots, \boldsymbol{v}^{(c)}\right) = \left(\boldsymbol{u}^{(1)}, \boldsymbol{u}^{(2)}, \ldots, \boldsymbol{u}^{(b)}\right) \times G_{\text{conv}}.$$

Since a convolutional encoder is a linear system, information and code sequences can be represented in polynomial form. For a rate $R = b/c$ convolutional code $\mathcal{C}$ with memory m, a generator matrix G can be expressed in terms of the delay operator D, that is,

$$G(D) = \begin{pmatrix} g_{11}(D) & g_{12}(D) & \ldots & g_{1c}(D) \\ g_{21}(D) & g_{22}(D) & \ldots & g_{2c}(D) \\ \vdots & \vdots & \ddots & \vdots \\ g_{b1}(D) & g_{b2}(D) & \ldots & g_{bc}(D) \end{pmatrix}$$

with the generator polynomials

$$g_{ij}(D) = g_{ij}^{(0)} + g_{ij}^{(1)} D + g_{ij}^{(2)} D^2 + \ldots + g_{ij}^{(m)} D^m$$

where $i = 1, 2, \ldots, b$ and $j = 1, 2, \ldots, c$. Note that the maximum degree of the generator polynomial is equal to the memory m. Then the relation between the information sequence and the code sequence, both expressed in terms of the delay operator D, is given by

$$\boldsymbol{v}(D) = \boldsymbol{u}(D) G(D)$$

with

$$\begin{aligned} \boldsymbol{u}(D) &= \boldsymbol{u}_0 + \boldsymbol{u}_1 D + \boldsymbol{u}_2 D + \ldots \\ \boldsymbol{v}(D) &= \boldsymbol{v}_0 + \boldsymbol{v}_1 D + \boldsymbol{v}_2 D + \ldots \end{aligned}$$

where $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ denotes the input b-tuple and output c-tuple at time instance t, respectively.

Expressing the generator matrix $G(D)$ as

$$G(D) = G_0 + G_1 D + G_2 D + \ldots + G_m D^m$$

we obtain $m+1$ sub-matrices of size $b \times c$, where the sub-matrix G_i is associated with all impulse responses at time instance $t = i$. Using these sub-matrices the semi-infinite generator matrix G is given by

$$G = \begin{pmatrix} G_0 & G_1 & \dots & G_m & 0 & 0 & \dots \\ 0 & G_0 & G_1 & \dots & G_m & 0 & \dots \\ 0 & 0 & G_0 & G_1 & \dots & G_m & \dots \\ \vdots & \vdots & & \ddots & \ddots & \dots & \ddots \end{pmatrix}. \tag{2.11}$$

Together with the semi-finite information sequence

$$\boldsymbol{u} = (\boldsymbol{u}_0, \boldsymbol{u}_1, \dots) = \left(u_0^{(1)} u_0^{(2)} \dots u_0^{(b)},\, u_1^{(1)} u_1^{(2)} \dots u_1^{(b)}, \dots\right)$$

and the semi-infinite code sequence

$$\boldsymbol{v} = (\boldsymbol{v}_0, \boldsymbol{v}_1, \dots) = \left(v_0^{(1)} v_0^{(2)} \dots v_0^{(c)},\, v_1^{(1)} v_1^{(2)} \dots v_1^{(c)}, \dots\right)$$

with the b-tuples $\boldsymbol{u}_t$ and the c-tuples $\boldsymbol{v}_t$ for the consecutive time instances $t = 0, 1, \dots$, the input-output relation follows as

$$\boldsymbol{v} = \boldsymbol{u}G.$$

Commonly, generator polynomials are described as octal numbers, which we will use throughout this thesis. In this format, each digit represents three bits, that is $2^3 = 8$ different values. Any generator polynomial hereinafter is expressed with its highest degree to the right.

Example
Given a rate $R = 1/2$ convolutional code with memory $m = 3$ and generator matrix $G = (g_{11}\ g_{12}) = (74\ 54)_8$, in a left-aligned octal form, the generator polynomials follow as

$$\begin{array}{rcrcrcl} 74_8 & = & 111100_2 & \Rightarrow & 1111_2 & = & 1 + D + D^2 + D^3 \\ 54_8 & = & 101100_2 & \Rightarrow & 1011_2 & = & 1 + D^2 + D^3. \end{array}$$

2.5 Generating the Code Sequence

Consider again the block diagram in Figure 2.2 showing a rate $R = 1/2$ convolutional code with memory $m = 2$ and generator matrix $G = (7\ 5)_8$, or in polynomial form

$$G(D) = \left(1 + D + D^2 \quad 1 + D^2\right). \tag{2.12}$$

When an encoder is realized in controller canonical form, the generator polynomials can be seen directly in the block diagram. The D-terms in a generator polynomial (e.g., D and D^2) correspond to a memory unit, starting with the overall highest degree on the right.

Assuming the memory elements to be initially filled with zeros, the outputs $v_t^{(0)}$ and $v_t^{(1)}$ are generated at each time instance t as follows:

(i) With the first polynomial, being $g_1(D) = 1 + D + D^2$, the current input u_t as well as the two previous input bits u_{t-1} and u_{t-2} are added together by modulo-2 adders to form the first output bit $v_t^{(0)}$, as illustrated in Figure 2.2.

(ii) Similarly, the second output $v_t^{(1)}$ is associated with the second polynomial $g_2(D) = 1 + D^2$, that is, the current input bit v_t is added together with the input bit two time instances earlier, u_{t-2}, currently stored in the second memory element (counted from left to right).

For a rate $R = 1/2$ convolutional code an input sequence $\boldsymbol{u} = (u_0, u_1, \ldots)$ generates two output sequences $\boldsymbol{v}^{(0)} = (v_0^{(0)}, v_1^{(0)}, \ldots)$ and $\boldsymbol{v}^{(1)} = (v_0^{(1)}, v_1^{(1)}, \ldots)$ which are the convolution between the input sequence and the *impulse responses* of the encoder.

Example
Consider a rate $R = 1/2$ convolutional code $\mathcal{C}$ with memory $m = 2$ and generator matrix $G = \left(1 + D + D^2 \; 1 + D^2\right)$. Given an information sequence $\boldsymbol{u} = (11\ 01)$ or in polynomial form $\boldsymbol{u}(D) = 1 + D + D^3$, the codeword is obtained as

$$\begin{aligned} \boldsymbol{v}(D) &= \boldsymbol{u}(D) \times G(D) \\ &= (1 + D + D^3) \times (1 + D + D^2 \quad 1 + D^2) \\ &= (1 + D^4 + D^5 \quad 1 + D + D^2 + D^5). \end{aligned}$$

By multiplexing the outputs, a codeword can be expressed in the following way:

$$\boldsymbol{v}(D) = v^{(0)}(D^2) + Dv^{(1)}(D^2)$$

and in our example we get the codeword

$$\boldsymbol{v}(D) = 1 + D + D^3 + D^5 + D^8 + D^{10} + D^{11}.$$

2.6 Distance Properties of Convolutional Codes

To ensure small error probabilities in the simple model of a communication channel (from Section (2.1)) the most important properties of a convolutional code $\mathcal{C}$ are the free distance (d_{free}) and the distance spectrum $n_{d_{\text{free}}}$ which govern the error probability in (2.1).

2.6.1 Free Distance

The free distance is similar to the minimum distance ($d_{\min}$) property for block codes (introduced in Section 2.2.1). The Hamming weight property introduced in (2.4) still holds for convolutional codes, but the codeword can now be viewed as a code sequence, since a convolutional encoder can generate code sequences of infinite length. Thereby, the distance property d_{free} now becomes the minimum Hamming distance between any two code sequences (i.e., the number of positions in which they differ) of possible infinite length.

In order to obtain small error probabilities it follows from (2.1) that convolutional codes with large free distances should be used. Therefore, determining the free distance of a convolutional code is a commonly encountered problem. Since we only consider linear codes, the free distance of a code $\mathcal{C}$ can be determined by finding the code sequence $\boldsymbol{v}$ of minimum weight

$$d_{\text{free}} = \min_{\boldsymbol{v} \in \mathcal{C}, \boldsymbol{v} \neq \boldsymbol{0}} \{w_{\text{H}}(\boldsymbol{v})\}. \tag{2.13}$$

As with block codes the distance parameter determines the error-correcting and error-detecting capabilities of a code $\mathcal{C}$. Any error pattern ϵ_t as defined in (2.8) can be detected if $t < d_{\text{free}}$ [9], and is guaranteed to be corrected if and only if

$$t \leq \left\lfloor \frac{d_{\text{free}} - 1}{2} \right\rfloor. \tag{2.14}$$

Note that it may be possible to correct some errors of larger weight, while this is not guaranteed in general. Moreover, for rate $R = b/c$ convolutional codes with same memory m only a small fraction of all generator matrices G have the largest free distance.

2.6.2 Catastrophic Generator Matrices

A generator matrix can cause catastrophic error propagation. Catastrophic error propagation occurs if an information sequence of an infinite number of non-zero bits at the input $w_{\text{H}}(\boldsymbol{u}) = \infty$ generates a sequence of a finite number of non-zero bits at the output $w_{\text{H}}(\boldsymbol{v}) < \infty$. In practical terms this means that a finite number of errors (e.g., induced by channel noise) are sufficient in order for a received sequence $\boldsymbol{r}$ to obtain an unlimited number of decoding errors. Catastrophic behavior is a generator matrix property and discarding such matrices is of great importance when designing an encoder. A necessary and sufficient condition have been provided by Massey and Sain [14] to ensure non-catastrophic behavior. For a rate $R = 1/2$ convolutional generator matrix

$$G(D) = (g_1(D) \quad g_2(D))$$

$G(D)$ is non-catastrophic if and only if

$$\gcd(g_1(D), g_2(D)) = D^l \tag{2.15}$$

for some non-negative integer l (gcd denotes the greatest common divisor). In our thesis we assume that at least one generator polynomial is *delay free*, that is, $g_1(0) \neq 0$ or $g_2(0) \neq 0$, and thereby simplifying the condition to

$$\gcd(g_1(D), g_2(D)) = 1. \tag{2.16}$$

2.6.3 Column Distance

Another common distance measure of a convolutional code is its jth order *column distance* d_j^{c}, which is defined as

$$d_{\text{j}}^{\text{c}} = \min_{\boldsymbol{u}, \boldsymbol{u}_0 \neq \boldsymbol{0}} \left\{ w_{\text{H}}(\boldsymbol{v}_{[0,\,j]}) \right\} \tag{2.17}$$

where $\boldsymbol{v}_{[0,j]}$ denotes a truncated code sequence resulting from the information sequences $\boldsymbol{u}_{[0,j]} = (\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_j)$ with a non-zero $\boldsymbol{u}_0$. The column distance d_j^{c} is a non-decreasing function and converges to the free distance

$$d_j^{\mathrm{c}} \leq d_{j+1}^{\mathrm{c}},\ j = 0, 1, 2 \ldots$$

$$\lim_{j\to\infty} d_j^{c} = d_{\infty}^{\mathrm{c}} = d_{\mathrm{free}}.$$

It has been observed that good computational performance for so-called *sequential decoding* is achieved if the column distance function grows as rapidly as possible [9]. A *distance profile* for a generator matrix G with memory m is given by the $m+1$ column distance tuple

$$\boldsymbol{d}^p = (d_0^{\mathrm{c}},\ d_1^{\mathrm{c}},\ d_2^{\mathrm{c}},\ \ldots,\ d_m^{\mathrm{c}}).$$

A distance profile $\boldsymbol{d}^p$ is equal or superior to another distance profile $\boldsymbol{d}^{p\prime}$ if there is a j_0 such that

$$d_j^{c} \begin{cases} = d_j^{c\prime}, & j = 0, 1, \ldots, j_0 - 1 \\ > d_j^{c\prime}, & j = j_0,\ j \leqq m \end{cases}$$

that is, if $\boldsymbol{d}^p > \boldsymbol{d}^{p\prime}$ the column distance function grows faster with j for $\boldsymbol{d}^p$ than for $\boldsymbol{d}^{p\prime}$. If the distance profile of a generator matrix G is superior to any other generator matrix of same rate and memory m, it is denoted an *optimum distance profile* (ODP) generator matrix [9].

2.6.4 Row Distance

If an information sequence is truncated at depth j, $\boldsymbol{u}_{[0,j]} = (\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_j)$, the memory units of the encoder are still filled with m previous information bits. By forcing the encoder back to the all-zero state (i.e., input m zeros) we terminate it with a so called *zero-tail* (ZT) termination. Using a ZT terminated information sequence $\boldsymbol{u}_{[0,j+m]}^{ZT} = (\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{j+m})$ with $\boldsymbol{u}_{[j+1,m]} = \boldsymbol{0}$, the jth order *row distance* is defined as

$$d_j^{\mathrm{r}} = \min_{\boldsymbol{u}_{[0,j]} \neq \boldsymbol{0}} \left\{ w_{\mathrm{H}} \left(\boldsymbol{u}_{[0,j+m]}^{ZT} G \right) \right\} = \min_{\boldsymbol{u}_{[0,j]} \neq \boldsymbol{0}} \left\{ w_{\mathrm{H}}(\boldsymbol{v}_{[0,j+m]}) \right\} \tag{2.18}$$

where $\boldsymbol{v}_{[0,j+m]}$ is the code sequence resulting from the ZT information sequence $\boldsymbol{u}_{[0,j+m]}^{ZT}$.

The row distance d_j^{r} is a non-increasing function and converges to the free distance for non-catastrophic generator matrices

$$d_j^{\mathrm{r}} \geq d_{j+1}^{\mathrm{r}}, \quad j = 0, 1, 2, \ldots$$

$$\lim_{j\to\infty} d_j^{\mathrm{r}} \begin{cases} = d_{\mathrm{free}}, & \mathrm{non-catastrophic} \\ \geq d_{\mathrm{free}}, & \mathrm{catastrophic.} \end{cases}$$

Under the assumption that we have non-catastrophic generator matrices we obtain the following lower and upper bounds on the free distance

$$d_0^{\mathrm{c}} \leq d_1^{\mathrm{c}} \leq d_2^{\mathrm{c}} \leq \ldots \leq d_\infty^{\mathrm{c}} = d_{\mathrm{free}} = d_\infty^{\mathrm{r}} \leq \ldots \leq d_2^{\mathrm{r}} \leq d_1^{\mathrm{r}} \leq d_0^{\mathrm{r}} \tag{2.19}$$

which will be exploited when performing an exhaustive code search later on.

2.6.5 Distance Spectrum

The number of codewords of a specific Hamming weight is the secondary most important property (2.1). The distance spectrum for a generator matrix G is given by

$$n_{d_{\mathrm{free}}+i}, \quad i = 0, 1, 2, \ldots \tag{2.20}$$

where $n_{d_{\mathrm{free}}+i}$ is the enumeration of all code sequences of weight $d_{\mathrm{free}} + i$. Each $n_{d_{\mathrm{free}}+i}$, also denoted the $(i+1)$th, spectral component diverge from the all zero sequence at time instance zero and after they re-merge with the all-zero sequence, they will not diverge again. It is interesting that the distance spectrum is an encoder property while the free distance as well as the burst error probability are code properties . [4].

Chapter 3

Exhaustive Code Search

The aim of this thesis is to perform an exhaustive code search for rate $R = 1/2$ generator matrices with memory $m = 26$. As described in Section 2.1 the free distance d_{free} is the most important criteria for evaluating the performance of convolutional codes. Finding convolutional codes with the *maximum* free distance, also denoted $d_{\text{free}}^{\max}$, for a given memory m is performed by an exhaustive code search. In order to reduce the number of encoder matrices we exploit their symmetrical properties and apply different rejection rules. Afterwards we use the BEAST (see Section 3.2) - Bidirectional Efficient Algorithm for Searching code Trees - to determine the free distance and spectral components of the remaining generator matrices.

3.1 Rejection Rules

For rate $R = 1/2$ convolutional codes with memory m there exist 2^{2m} possible generator matrices $G = (g_1(D)\ g_2(D))$. Because of the huge number of possible matrices, the complete ensemble needs to be greatly reduced before applying the BEAST. A couple of limitations on the original number of generator matrices can be made:

- Clearly $G = (g_1(D)\ g_2(D))$ and $G' = (g_2(D)\ g_1(D))$ are symmetrical and obviously have the same properties. Consequently the total number of matrices can be reduced by 50%.
- As we search for generator matrices of memory m, we consider only generator matrices $G = (g_1(D)\ g_2(D))$ where at least one of the polynomials has a D^m-term, i.e., $g_1(D)$ or $g_2(D)$ is odd in a numerical notation. This will discard another 25%.

After these limitations, the original ensemble of 2^{2m} encoder matrices is now reduced to $3 \cdot 2^{2m-3} - 2^{m-2}$ [9]. However, this number is still too large and approximately grows with a factor 4 for each m. Table 3.1 shows the number of generator matrices that need to be checked for memory m, after applying the initial limitations.

However, the free distance is upper-bounded by the row distance d_j^{r} as shown by (2.19). Therefore, it is possible to use the row distances as a fast rejection rule.

m	# generator matrices	m	# generator matrices
2	5	23	26388276969472
3	22	24	105553112072192
4	92	25	422212456677376
5	376	26	1688849843486720
6	1520	27	6755399407501312
7	6112	28	27021597697114112

Table 3.1: The total number of $R = 1/2$ generator matrices that must be checked for some values of m, after applying initial limitations.

However, we identify a certain target free distance (d^t_{free}), i.e., we are limiting our search to generator matrices with $d_{\text{free}} \geq d^t_{\text{free}}$. Then we can use the jth order row distance d^r_j as an upper bound on the free distance d_{free} and reject all generator matrices if $d^r_j < d^t_{\text{free}}$ for any $j = 0, 1, 2, \ldots$. All generator matrices that survived the row distance rejection rules, or *row distance-sieve*, will be checked for catastrophic behavior by verifying that (2.16) is satisfied. Note that approximately 10% of all surviving generator matrices will be discarded by (2.16).

3.2 The BEAST

The Bidirectional Efficient Algorithm for Searching code Trees (BEAST), introduced in [2, 3] is, in its simplest and original version, a method for finding the code distance spectrum. Every convolutional code $\mathcal{C}$ of rate $R = b/c$ can be represented by a graph, constructed from *nodes* and *edges*. The BEAST is used to greatly reduce the amount of nodes needed visiting [13].

The state σ of an encoder of a rate $R = 1/2$ convolutional code $\mathcal{C}$ at time t, specifies the previous m input bits, and is together with the current input bit u_t sufficient to determine the next state, as well as the output-sequence associated with this state transition.

3.2.1 State-Transition Diagram

The state-transition diagram of a rate $R = 1/2$ convolutional encoder with memory $m = 2$ and generator matrix $G = (1 + D + D^2 \; 1 + D^2)$ in Figure 2.2 is illustrated in Figure 3.1, containing $2^m = 4$ encoder states. Every node ξ is labeled by one of the $2^m = 4$ encoder states with edges between them. The edges correspond to state-transitions and are labeled by the corresponding input bit and output 2-tuple u/v.

For example, starting at the all-zero state $\boldsymbol{\sigma} = (00)$, at time $t = 0$, and using the input-sequence of $\boldsymbol{u} = (1\ 0\ 1\ 0\ 0\ \ldots)$ will result in visiting the states $\boldsymbol{\sigma} = (00\ 10\ 01\ 10\ 01\ 00\ \ldots)$ and yield the code sequence $\boldsymbol{v} = (11\ 10\ 00\ 10\ 11\ \ldots)$.

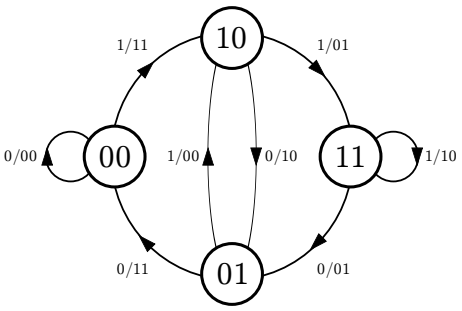


Figure 3.1: State-transition diagram of rate $R = 1/2$ convolutional encoder with $G = (7\ 5)_8$ and memory $m = 2$.

3.2.2 Code Tree

Every convolutional code $\mathcal{C}$ can also be represented by either a forward or a backward tree as illustrated in Figure 5.6. Every node ξ in the tree is characterized by three parameters: the state $\sigma(\xi)$, the tree depth $\ell(\xi)$, and the weight $w(\xi)$. Clearly, every node ξ has a single parent node ξ^{P} and 2^b child nodes ξ^{C}.

The forward tree stems from the root node ξ_{root} at depth 0 and is indicated by the subscript f. The node weight $w_f(\xi)$ is the accumulated Hamming weight of the path from the root node ξ_{root} to the current node ξ

$$w_f(\xi) = \sum_{t=1}^{\ell_f(\xi)} w_{\mathrm{H}}(\boldsymbol{v}_t) \tag{3.1}$$

where $\ell_f(\xi_{\mathrm{root}}) = 0$, $w_f(\xi_{\mathrm{root}}) = 0$, and $\sigma(\xi_{\mathrm{root}}) = \mathbf{0}$, and where $\boldsymbol{v}_t$ denotes the output c-tuple for the state-transition at depth t counting from the root, and leading to the current node ξ.

The backward tree stems from toor node ξ_{toor}[1] and is indicated by the subscript b. Similar to a node in the forward tree, a node in the backward tree has weight $w_b(\xi)$, that is the accumulated Hamming weight of the path from the toor node ξ_{toor} to the node ξ

$$w_b(\xi) = \sum_{t=1}^{\ell_b(\xi)} w_{\mathrm{H}}(\boldsymbol{v}_t) \tag{3.2}$$

where $\ell_b(\xi_{\mathrm{toor}}) = 0$, $w_b(\xi_{\mathrm{toor}}) = 0$ and $\sigma(\xi_{\mathrm{toor}}) = \mathbf{0}$, and $\boldsymbol{v}_t$ denotes the output c-tuple for the state transition at depth t (counting from the toor) and leading to the current node ξ.

3.2.3 Finding the Code Spectrum

All codewords of a non-catastrophic generator matrix of a rate $R = b/c$ convolutional code with memory m can be represented by a path through the code

[1] The toor node is the root of the backward tree.

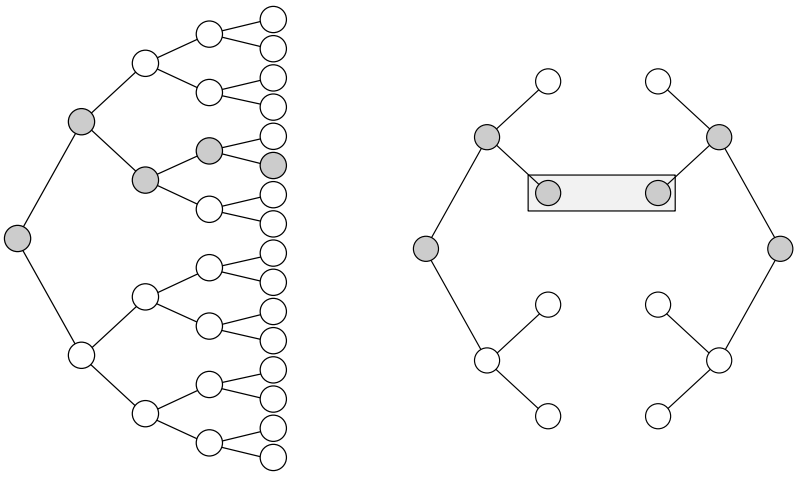


Figure 3.2: Illustration of a unidirectional and a bidirectional code tree search. The gray nodes represent the shortest path found between ξ_{root} and ξ_{toor}, yielding the d_{free}. The two nodes in the rectangle, in the bidirectional search, are actually the same node found in both the forward and the backward tree, indicating a match.

tree. Such a path leaves the all-zero state from the root node ξ_{root} at time instance $t = 0$, and after it re-merges with the all-zero state at the toor node ξ_{toor}, it does not diverge again.

This implies that for every path $\xi_{\text{root}} \rightarrow \xi_{\text{toor}}$ of weight w, there exist an intermediate node ξ such that $w_f(\xi) = f_w + j$ and $w_b(\xi) = b_w - j$, $j = 0, 1, \ldots, c-1$, fulfilling

$$f_w + b_w = w \tag{3.3}$$

where $w_f(\xi)$ and $w_b(\xi)$ are the accumulated Hamming weights of the code sequence segments $\xi_{\text{root}} \rightarrow \xi$ and $\xi \rightarrow \xi_{\text{toor}}$, respectively.

In order to find the number of codewords of weight w, the BEAST performs an independent forward and backward search obtaining the forward and backward sets $\mathcal{F}_{+j}$ and $\mathcal{B}_{-j}$

$$\mathcal{F}_{+j} = \left\{ \xi | w_f(\xi) = f_w + j, w_f(\xi^P) < f_w, \sigma(\xi) \neq 0 \right\} \tag{3.4}$$

$$\mathcal{B}_{-j} = \left\{ \xi | w_b(\xi) = b_w - j, w_b(\xi^C) > b_w, \sigma(\xi) \neq 0 \right\}. \tag{3.5}$$

Although f_w and b_w may be selected freely as detailed in (3.3), an unbalanced distribution would result in one set containing more nodes than the other one, decreasing the efficiency of the BEAST.

Figure 3.2 illustrates the amount of nodes to be visited in order to find the shortest path through the code tree for a unidirectional and a bidirectional tree search.

Finally, the spectral component corresponding to weight w is defined as the number of codewords of weight w and is given by

$$n_w = \sum_{j=0}^{c-1} \sum_{(\xi_f, \xi_b) \in \mathcal{F}_{+j} \times \mathcal{B}_{-j}} \chi(\xi_f, \xi_b) \tag{3.6}$$

where ξ_f is a node from the forward set, ξ_b a node from the backward set, and χ is the function for indicating a match, defined as

$$\chi(\xi_f, \xi_b) = \begin{cases} 1, & \text{if } \sigma(\xi_f) = \sigma(\xi_b) \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

Chapter 4

The Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture was developed by an alliance known as *STI* (Sony, Toshiba and IBM) to meet requests by Sony and Toshiba. The requests called for a high-performing chip with a high performance-to-power ratio, initially targeting gaming consoles.

The Cell architecture consists of one 64-bit IBM PowerPC called the Power Processor Element (PPE) which is used for running an operating system and managing system resources. It also includes eight co-processors called Synergistic Processor Elements (SPE)[1], which are connected to each other and to the PowerPC Processor Element through the high bandwidth Element Interconnect Bus (EIB). Figure 4.1 illustrates how the processors, memory and I/O controllers are connected using the EIB.

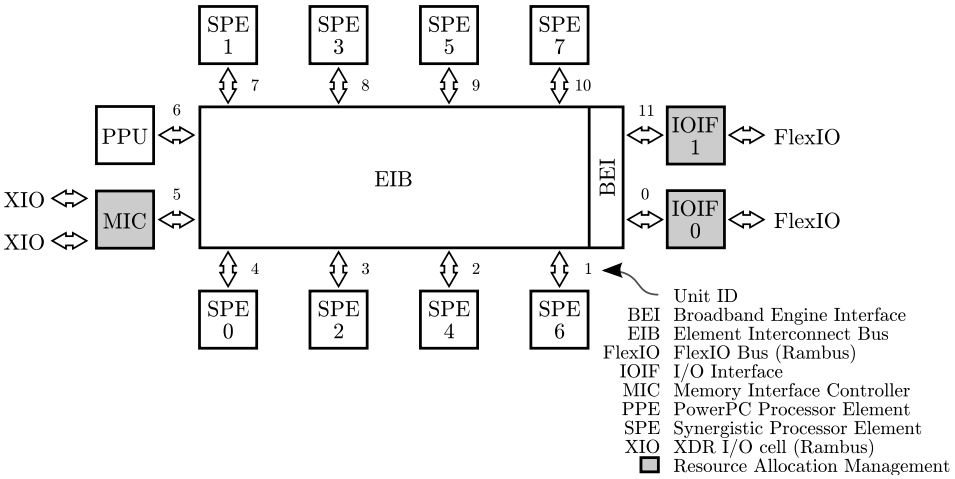


Figure 4.1: A schematic overview of the processor layout.

[1]Current processors contain 7 SPEs in order to increase production yield.

4.1 The PowerPC Processor Element

The PowerPC Processor Element (PPE) is a general-purpose processor and conforms to the PowerPC Architecture, introduced by IBM in the late 1970s. This allows applications written for the 970-family (e.g., Power Mac G5) to run on the processor without modification. The main processing unit in the PPE is the PowerPC Processor Unit (PPU), which is a 64-bit, dual-threaded Reduced Instruction Set Computing (RISC) processor supporting both the PowerPC and the Vector/SIMD[2] Multimedia Extension (VMX) Instruction Set Architectures (ISA) . It can operate in either 32- or 64-bit mode and it has 32 64-bit general purpose registers, 32 64-bit floating-point registers and 32 128-bit vector registers.

Included in the processor is the Power Processor Storage Subsystem (PPSS) which handles memory requests to and from the PPE. The PPSS contains a unified[3] 512 KB level 2 cache, various queues and a bus interface unit that handles communication with the EIB.

4.1.1 The Cache, Pipeline and Branch Prediction

The PPE has a 32 KB level 1 (L1) instruction cache and a 32 KB L1 data cache; both of which are shared between two threads and allow a block to be loaded by one thread and used by the other. It can execute up to two instructions per cycle, and pre-fetch up to four. Additional processor components include an Instruction-control Unit (IU)[4] which interprets the instructions and directs them to correct operational units and a Load- and Store Unit (LSU) coordinating memory access. Additionally, the PPU also contains a Fixed-Point Unit (FXU) and a Vector/Scalar Unit (VSU)[5]. As the PPE has only one set of functional units, only one thread may use one unit at a time. Table 4.1 shows the permitted functional unit combinations for two threads. Dual-issue occurs when both threads use separate functional units, allowing two instructions to complete for every one cycle. Stalls occur when one thread has to wait for the other to complete its use of a functional unit.

Branch misses can be prevented by hinting likely/unlikely[6] outcomes, but a missed branch comes with a 23-cycle penalty so it is often better to allow the PPU to use its own branch prediction.

4.2 The Synergistic Processor Element

The Synergistic Processor Element (SPE) consists of a Synergistic Processor Unit (SPU) and a 256 KB Local Store (LS) as well as a Memory Flow Controller (MFC) for moving data in and out of the LS. It is a SIMD RISC processor with 32-bit fixed length instructions, operating on 128-bit data. The processor has 128 general-purpose registers that are used by both floating-point and integer instructions. It

[2]Single-Instruction, Multiple-Data (see Section 4.4).

[3]Both instruction and data share the same cache.

[4]The IU contains the Branch Unit (BRU) handling branch statements.

[5]Containing Floating-Point Unit (FPU).

[6]By the use of the compiler directive `__builtin_expect`.

	FXU	LSU	BRU	VSU 1	VSU 2
FXU		√	√	√	√
LSU	√		√		
BRU	√	√		√	√
VSU 1	√	√	√		√
VSU 2	√	√	√	√	

Table 4.1: Combinations of functional units allowing dual-issue. The VSU 2 refer the instructions using the VSU for loading, storing and permuting data. VSU 1 refer to the remaining instructions, not handled by VSU 2 [17].

does *not* handle dynamic branching but instead relies on the compiler's branching instructions as detailed by [8].

The Local Store is a SPU-private memory, containing the SPU-application, static memory and stack. It is an SRAM-array which is byte-addressed, but only permits 16-byte aligned data to be loaded and stored. The LS completes a read instruction in six cycles and a write instruction in four. Operations have different priorities and are, in descending priority: DMA[7] read/writes, load/stores and instruction pre-fetching.

4.2.1 Pipeline and Branching

The SPU has two pipelines called the even and the odd pipeline. Each pipeline handle different types of instructions, allowing two instructions to complete per cycle (one in each pipeline). Provided there are no dependencies between the two instructions, dual issue can occur, which can be utilized to increase performance.

As previously mentioned, the SPU does not (like a general-purpose processor such as the PPU) have a branch-prediction cache. Branches are assumed not to be taken, and when a branch is taken, the processor stalls which has a negative impact on performance. Note that stalls due to branches can be avoided using the previously mentioned hinting compiler directive `__builtin_expect`.

4.2.2 Synergistic Memory Flow Control

The SPU stores and retrieves data between main memory and the LS by issuing requests to the Memory Flow Controller (MFC). This is achieved by giving the MFC one location in main memory and one in the LS along with an amount of data to be copied. The data transfers are asynchronous, which allows the SPU to continue working while memory is being copied. One request can issue a copy of up to 16 KB of data, identified using group identification number with data transferred at a rate of 16 byte per cycle. The group identification numbers are used to identify data transfers and are used by the SPU to check for completeness

[7] Direct Memory Access is a feature that allow the SPUs memory access independently of the PPU.

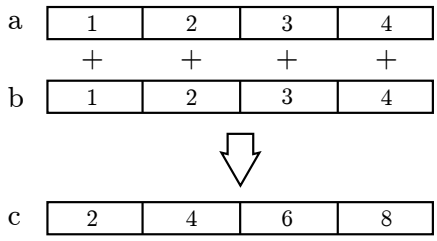


Figure 4.2: Adding two vectors a and b using the intrinsic `vec_add`.

of the same. Requests may complete in a different order than in which they were issued, and to enforce control, fence and barrier instructions exist if needed.

4.3 The Element Interconnect Bus

The Element Interconnect Bus (EIB) handles all command and data communications between the processor elements and the controllers for memory and input/output operations. The EIB consists of four data rings, each 16 byte wide and able to carry 128 bytes at a time. Each connected element has a unit identification number (see Figure 4.1) which enables a programmer to minimize latency by addressing specific elements. The EIB has a maximum internal bandwidth of 96 bytes per cycle and each ring can have multiple outstanding DMA requests between a SPE local store and the main memory.

4.4 Single Instruction, Multiple Data

Single Instruction Multiple Data (SIMD) is a technique to enable data parallelism. SIMD-enabled processing units have been designed to handle data in vectors, which in the case of current Cell processors are 128-bit vector types. This means that as opposed to a scalar 32-bit data type (e.g., `float`), four 32-bit operations can be carried out in parallel by using a `vector float`. An example of a vector intrinsic (from the VMX instruction set) is `vec_add`, which is used to add together two vector types (all element-wise), illustrated in Figure 4.2. All these instructions are made accessible through C and C++ language extensions and intrinsics[8].

As the PPU and SPU are of different architectures, they also differ in their supported instruction sets. Two important differences (detailed in [17]) between the two processors are:

- The SPU support 64-bit vector types (i.e., `vector double` and `vector long long`) and the PPU does not.
- The PPU support the `vector pixel` data type, which is used to represent pixel information, and the SPU does not.

[8] An intrinsic is an abstraction of machine level instructions.

Unlike the PPU Floating-Point Unit (FPU), the PPU SIMD Execution Unit (VXU[9]) sacrifices precision for speed (*graphics rounding mode*); resulting in a number of behavioral differences from IEEE 754, namely:

- Subnormal[10] and underflow numbers are automatically rounded to zero.
- Infinity and NaN[11] are processed as if normal.
- The positive overflow value is set to `0x7FFFFFFF`.
- The negative overflow value is set to `0xFFFFFFFF`.

4.5 The PlayStation 3

The PlayStation 3 contains 256 MB of Extreme Data Rate Dynamic Random Access Memory (XDR DRAM). When running a guest operating system on the PlayStation 3, it is run in an Hypervisor environment supervised by Sony's operating system GameOS. The GameOS reserves the use of one SPU, making in total six SPUs available to a guest operating system. Similarly, not all 256 MB of XDR DRAM is available to the guest operating system.

The PlayStation 3 is also fitted with a Reality Synthesizer (RSX) Graphics Processor Unit (GPU) from nVidia, with 256 MB of GDDR3 RAM. However, no access to the accelerated GPU functions is given when running in the Hypervisor environment.

Finally, the PlayStation 3 offers a disc drive capable of reading Blu-ray discs, DVDs and CDs.

4.5.1 Limitations of the PlayStation 3 Gaming Console

The PlayStation 3 consoles provide an affordable solution to achieving high performance. However, there are a number of limitations imposed by using the PS3 for high performance computing. The main memory, where approximately 200 MB is available for Linux OS and user applications, can in some applications impose a limitation on the performance. The SPU only has 256 KB of memory, which often result in memory being copied back and forth from main memory, adversely affecting performance. Furthermore, the application binary which is executed by the SPU must also fit in the LS, therefore additional programming aspects such as code optimizations and code size also must be addressed.

A key factor when developing applications for the Cell processor is that all hardware functionality is available to the programmer. This makes it possible to achieve peak performance in a predictable way, as opposed to a conventional processor where cache memories can not be controlled directly. Note, that writing fast code that achieves near peak performance is not a trivial task and requires a deep knowledge of the architecture and low level programming [1].

[9] Vector Multimedia eXtension instruction set (also known as AltiVec).

[10] A number that is closer to zero than the smallest possible `float` value, emulated in software.

[11] Not a Number which signals a result of an operation with invalid input.

Chapter 5
Implementation Details

To perform an exhaustive code search, both the row distance-sieve algorithm and the BEAST have been implemented for the Cell Broadband Engine Architecture, using the architecture's inherent parallelism.

5.1 Parallelism

It is possible for us to split the full ensemble of encoders in arbitrary sized stand-alone sub sets, because each encoder is independent of another. In order to fully utilize the parallelism offered by the multiple Cell processors in our setup, our implementation utilizes the following:

Five Cell processors are available in our setup of PlayStation 3 consoles, effectively reducing the total time needed for our calculations by a factor of five. This is possible due the lack of data dependency between encoders.

SIMD instructions allow us to process multiple data where we would otherwise only be able to process single data.

Dual PPU threads facilitate dual issue on the PPU.

Six SPUs increase the throughput of encoders by a factor of almost six (see Section 6.2.3), again related to independent nature of each encoder.

Dual SPU pipeline facilitate dual issue on the SPU.

Asynchronous DMA avoids halting the SPU when moving data in and out of the SPU local store.

5.2 Row Distance-Sieve

Assume a time-invariant generator matrix and consider the row distance definition in (2.18). Let S_j denote all information sequences $\boldsymbol{u} = (u_0, u_1, \ldots, u_j)$ where $u_0 = 1$ and $u_j = 1$, at row distance depth j. The cardinality $|S_j|$ of S_j is then

$$|S_j| = \begin{cases} 2^{j-1}, & j > 1 \\ 1, & 0 \leq j \leq 1. \end{cases} \tag{5.1}$$

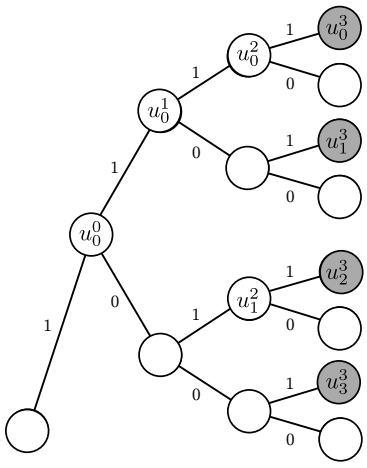


Figure 5.1: A code tree illustrating the row distance calculation.

Denote $\boldsymbol{v}_i$ the code sequence resulting from $\boldsymbol{u}_i$, then for a rate $R = 1/2$ convolutional code with time invariant generator matrix $G = (g_1\ g_2)$, the row distances d_j^{r}, $j = 0, 1, \ldots$, are calculated as follows:

1. Set $j = 1$, generate the set S_j and convolve each $\boldsymbol{u}_i \in S_j$, $i = 1, 2, \ldots, \left|S_j\right|$ with the two generator polynomials resulting in two sequences $\boldsymbol{v}_i^{(0)}$ and $\boldsymbol{v}_i^{(1)}$, that is,
$$\begin{aligned} \boldsymbol{v}_i^{(0)} &= \boldsymbol{u}_i * g_1 \\ \boldsymbol{v}_i^{(1)} &= \boldsymbol{u}_i * g_2. \end{aligned} \tag{5.2}$$
Compute the Hamming weight of the resulting code sequence $w_i = w_{\mathrm{H}}(\boldsymbol{v}_i^{(0)}) + w_{\mathrm{H}}(\boldsymbol{v}_i^{(1)}), \quad i = 1, 2, \ldots, \left|S_j\right|$.
2. Set the jth order row distance $d_j^{\mathrm{r}} = \min(w_i)$, $i = 1, 2, \ldots, \left|S_j\right|$. Afterward set $j = j + 1$ and repeat algorithm starting with step 2.

For performance, our implementation is limited to 32-bit data types which imposes a limit on the generator matrices, where the generator polynomials in $G = (g_{11}\ g_{12})$ can consume at most 32 bits. In other words, only generator matrices with memory $m < 32$ is supported since the generator polynomials must be represented by a 32-bit data type.

Since all encoding operations are over the Galois field $GF(2)$ the convolution given in (5.2) is a *discrete convolution*, which can be calculated with a few XOR- and shift-operations. The resulting sequence $\boldsymbol{v}_i$ spans over $j + m$ bits. Therefore, all cases where $m + j > 32$ can be handled properly by simply counting the truncated bits. Furthermore, the SPU instruction set provides a special instruction for counting bits which is used extensively[1], for example when computing the Hamming weight of each convolved sequence $\boldsymbol{v}_i$.

[1]The `spu_cntb`-instruction counts the number of ones on an 8-bit boundary.

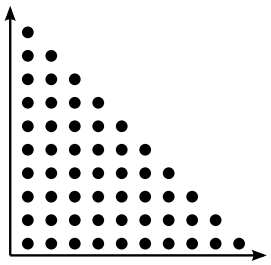


Figure 5.2: A triangular iteration space.

The row distance-sieve is trivial and is a *breadth-first* traversal of a code tree (see Section 3.2.2). When traversing the tree, the number of nodes that need to be processed at each jth order row distance is given by (5.1). From the traversal, the sequences $\boldsymbol{u} \in S_j$ can be processed sequentially (i.e., $\boldsymbol{u}_i, \boldsymbol{u}_{i+1}, \boldsymbol{u}_{i+2}, \ldots$) and independently which is a desired behavior for SIMD-instructions. Furthermore, if $j \geq 3$ then $\left|S_j\right| \geq 4$ (5.1) and four sequences $(\boldsymbol{u}_i, \boldsymbol{u}_{i+1}, \boldsymbol{u}_{i+2}, \boldsymbol{u}_{i+3})$ can be processed simultaneously (SIMD).

Figure 5.1 illustrates the calculation of the row distances d_j^{r}, $j = 0, 1, 2, 3$ in an example code tree for a rate $R = 1/2$ encoder with memory $m = 3$. Several nodes are marked with an information sequence $\boldsymbol{u}_i^j$ where j is the row distance order. These are the actual nodes that will be processed by the row distance-sieve. The dark-shaded nodes correspond to d_3^{r} where SIMD-instructions can be utilized fully.

5.2.1 Parallel Work Distribution

The problem of generating all generator matrices fits very well for parallel computing and multi-core systems. Since there is no dependency between the generator matrices, their generation can be implemented within a series of nested loops using a *data parallelism* programming model where each processor performs identical operations on distinct data simultaneously[16]. This is a simple and common way to decompose a problem and is often encountered in scientific applications.

The nested loops create a *triangular* iteration space that consists of an outer and inner loop. The inner loop can be executed independently of all others but depends on the index of the outer loop. Figure 5.2 shows the iteration space where each dot corresponds to an iteration in the inner loop. Clearly, a simple partitioning scheme where each vertical vector of dots is assigned to a processor p would yield an uneven workload.

In order to achieve an optimal work distribution, an *approximation-based near optimal partitioning* approach from [10] is chosen, that works well for work distribution in homogeneous systems (i.e., identical types of computational units). Even though CBEA is a heterogeneous platform, this scheme is applicable since the partitions will be executed exclusively on the SPUs and hence on the same instruction set architecture. Using the concept with dots from Figure 5.2 this scheme would decompose the outer loop into partitions containing an approx-

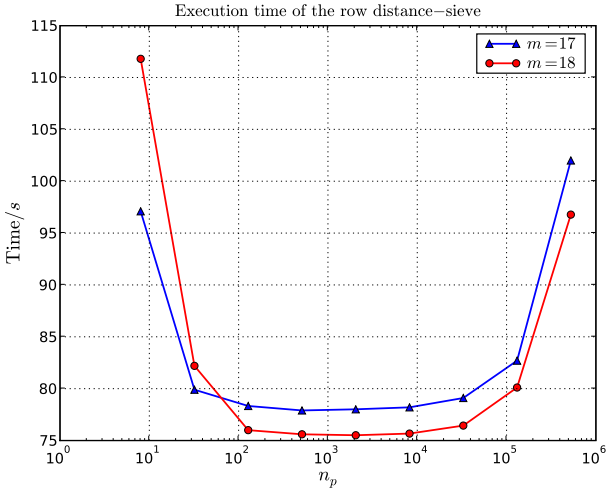


Figure 5.3: Execution time of the row distance-sieve with iteration space decomposed into different number of partitions n_p.

imately equal number of dots (i.e., iterations in the inner loop). Let i_k be the outer loop index for partition k (denoted p_k), $i_{\max}$ the upper limit of the outer loop, and n_p the number of partitions. Then the loop bounds for partition p_k are given by (i_k, i_{k+1}) where i_k follows as

$$i_k = \text{round}\left(i_{\max}\left(1 - \sqrt{\frac{p_k}{n_p}}\right)\right). \tag{5.3}$$

This partitioning scheme is used to distribute work among the SPUs on a single Cell processor, where we typically used $n_p = 512$ partitions.

Furthermore, each partition can be decomposed into j sub-partitions, denoted q_j. The loop bounds (i_j, i_{j+1}) for each sub-partition q_j, with n_q being the number of sub-partitions, are given by

$$i_j = \left\lfloor \frac{(i_{k+1} - i_k)}{n_q} \right\rfloor. \tag{5.4}$$

By decomposing the complete iteration space into disjoint partitions and sub-partitions, computations can be distributed among an arbitrary number of Cell processors, where each Cell would be *statically* assigned a distinct sub-partition. Although the actual computational time for each partition (and sub-partition) is non-deterministic, it is in fact quite easy to achieve a near optimal time with regard to the number of partitions. This is illustrated in Figure 5.3 where the complete ensemble of generator matrices with memory 17 and 18 has been decomposed into different number of partitions, ranging from $n_p = 8$ to $n_p = 524288$ and then processed with the row distance-sieve. Obviously, a near optimal number of partitions is $128 \lesssim n_p \lesssim 1024$.

5.3 The BEAST

In order to perform an exhaustive code search for generator matrices with memory 26, a lot of calculations (see Table 3.1) have to be performed. The row distance algorithm in Section 5.2 greatly reduces the amount of generator matrices but in order to finally determine the actual free distance, an algorithm like the BEAST is needed. The BEAST can also be used to obtain the code spectrum which is needed to rank the generator matrix and finally obtain the optimum free distance (OFD) generator matrix.

5.3.1 Problem Partitioning

The problem of determining the free distance can be broken down into three main parts: *generating* a result set, *sorting* it and *matching* it to another set. Generating the result sets is a task well suited for the SPU, as it requires very little input-data, a fair amount of calculations, and has a relatively low output rate. The result sets often requires more memory than available in the LS, which means that the generated data needs to be offloaded into main memory, while the LS is reused to allow the SPU to complete calculating the result set.

Additionally the SPU is assigned the task of sorting the result sets. This is possible without the full sets by splitting the result sets into partitions of a suitable size, known by both the SPU and the PPU.

The matching of different result sets is an operation well suited for the PPU as it has access to the full, partitioned, sets and it is possible to utilize its L1 data cache effectively.

In a naive parallelization, the forward and the backward result sets are obtained by simultaneous calculations on two individual processing units. Assuming that both, forward and backward sets, take the same amount of time to be calculated, the amount of time needed has effectively been halved. In reality however, the sets are rarely balanced, resulting in one processing unit stalling while the other completes its computations. This is a parallelization strategy that may only be suitable for quickly processing *one* encoder. However, in processing many encoders, stalling needs to be kept to a minimum which is made possible by calculating both the forward and the backward set on one processing unit, reducing the overall execution time.

5.3.2 Critical Path

The critical path represent the events that have to be performed in a specific sequence. It solely dictates how a given implementation performs, meaning a shorter critical path would shorten the overall time needed. A prerequisite for shortening a critical path, given a number of operations to be carried out, is that multiple operations can be carried out in simultaneously.

Figure 5.4 illustrates the critical path for calculating a target w. The figure illustrates how data needs to be prepared by the PPU before the SPU starts its processing of the tree. Once the SPU has started traversing the code tree, only the event of having generated 16 KB of result set data will interrupt its execution.

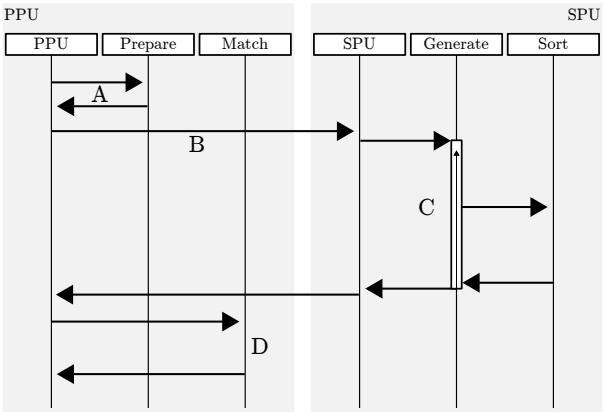


Figure 5.4: Sequence diagram of the execution when calculating a result set. In *A* the root node ξ_{root} is expanded into a `vector` type containing four nodes. *B* includes signaling and a DMA operation. *C* represents the possibly repeated generating and sorting of the result set in 16 KB chunks. At *D* the complete result set has been generated and is matched.

When the interrupt occurs, the result set is sorted and a DMA operation is issued to have it asynchronously copied into main memory. In other words, the copying of data occurs *while* the SPU continues to traverse the code tree. Once the entire code tree has been processed, control is handed over to the PPU for matching of the newly generated result set to previously generated result sets.

5.3.3 Traversing Code Trees

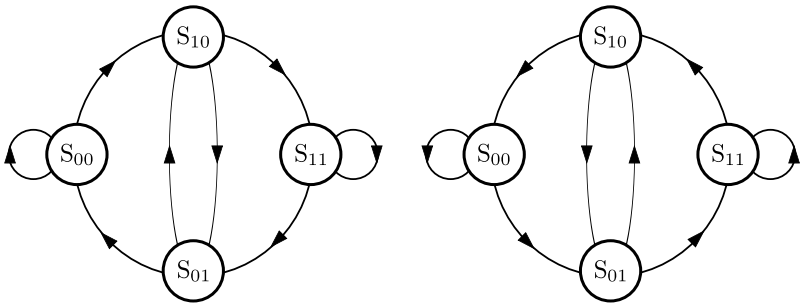
The state σ as introduced in Section 3.2.2 is expressed as a vector of delay elements $(d^{(1)}, d^{(2)}, \ldots, d^{(m)})$. For a rate $R = b/c$ encoder and memory m, the input u as well as every delay element $d^{(i)}$ are comprised of b bits. We define a function $\sigma_{f_{\ell+1}}$ for moving in a forward direction in the forward tree and also a function $\sigma_{f_{\ell-1}}$ for moving in a backward direction in the *forward* tree by

$$\sigma_{f_{\ell+1}} = (u, d^{(1)}, d^{(2)}, \ldots, d^{(m-1)}) \tag{5.5}$$

$$\sigma_{f_{\ell-1}} = (d^{(2)}, d^{(3)}, \ldots, d^{(m)}, u') \tag{5.6}$$

where u and u' are the inputs for moving forward and backward respectively. Corresponding functions for moving in the backward tree are defined by $\sigma_{b_{\ell+1}} = \sigma_{f_{\ell-1}}$ and $\sigma_{b_{\ell-1}} = \sigma_{f_{\ell+1}}$.

An example of a rate $R = 1/2$ encoder with memory $m = 2$ is illustrated in the state transition diagram in Figure 5.5 which shows the four possible states and the possible transitions between the states.



(a) $\sigma_{f_{\ell+1}}$ **(b)** $\sigma_{f_{\ell-1}}$

Figure 5.5: State transition diagrams illustrating $\sigma_{f_{\ell+1}}$ and $\sigma_{f_{\ell-1}}$ for a rate $R = 1/2$ convolutional encoder with memory $m = 2$.

Figure 5.6 shows the same encoder, as presented in Figure 5.5, in a code tree where the functions $\sigma_{f_{\ell+1}}$ and $\sigma_{f_{\ell-1}}$ are applied to traverse the tree in a forward direction starting from the ξ_{root} and in a backward direction starting from the ξ_{toor}. The edge labels represent the input u and u' respectively.

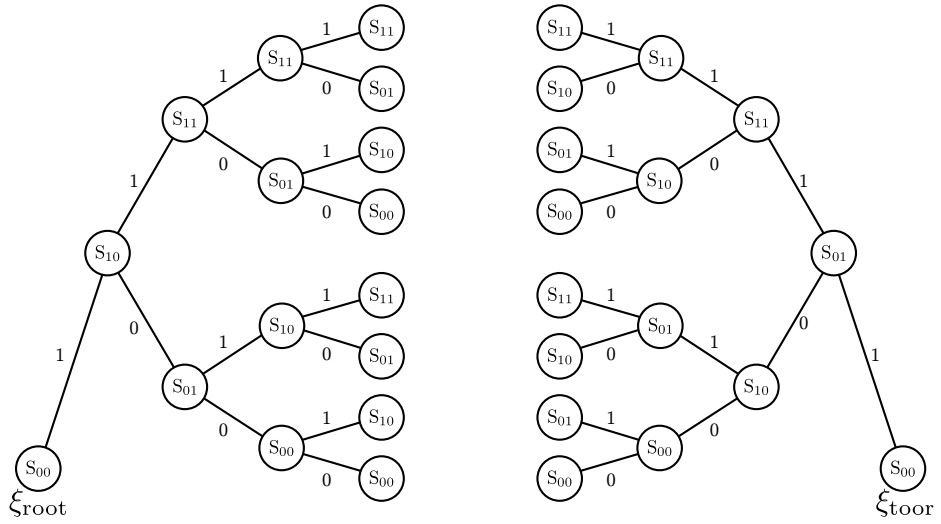


Figure 5.6: Example showing a forward and backward code tree.

5.3.4 Recursion vs. Iteration

Writing a recursive function is often a very expressive way of solving a particular problem. This is true for many computational problems and particularly true for processing tree structures. Since the BEAST uses two trees, recursion seems like

the natural pattern and a recursive implementation of the BEAST is possible to write with a moderate amount of code. However, recursions often come at a price of a larger amount of memory needed for every successive call to the function. Additionally there is no way of predicting the size of the result set and in turn no way of predicting how deep a code tree has to be processed.

As memory is a critical resource on the PlayStation 3, and in particular on the SPU, processing deep trees using recursion is a highly undesirable method. In Section 5.3.5 an iterative method is described which inhibits stack-consumption. It also provides a platform for easily monitoring the required amount of memory (which greatly differs between different generator matrices) but comes at the expense of doing extra calculations.

5.3.5 Iterative Depth-First Search of Code Trees

A standard approach to traversing trees in a depth-first manner, also known as Depth-First Search (DFS), is as mentioned recursive. However, it is possible to *avoid* recursions by performing calculations which would be otherwise implicitly handled by the call stack. This method involves maintaining a stack data structure[2] of nodes yet to be visited. Figure 5.7 shows a simple example performing a DFS in a code tree with seven nodes by using a stack. Using a stack data structure for storing states will result in less memory being consumed as there is no function call taking place. It will, however, still consume plenty of memory when processing deep trees as most of the memory is consumed by keeping track of the current state and not by the function call itself.

A characteristic of traversing a code tree instead of a general tree structure (with arbitrary nodes and edges) is that the code tree is processed in a deterministic manner. As it is evident from the example in Figure 5.5a, being at state S_{01} and going backwards, there are two possible parent nodes. Since there is only one *actual* parent node, a method like the Iterative Depth-first search of Code Trees (IDCT) is required for mapping the path needed for backtracking.

Using the fact that it is possible to backtrack in the tree as long as we know the path of how we got to a particular state, combined with the methodology of the iterative DFS we are able to construct the IDCT and effectively reduce the information needed to be saved for every depth of the tree from tens of bytes to single bits. The program listing presented in Algorithm 1 uses a *bit-stack*, which is a stack data structure able to store individual bits in a highly compact format[3].

Clearly we are interested in storing the bits shifted *out* when calculating the following state. These bits represent the path needed to backtrack the code tree, and are thus used as input when moving backward using $\sigma_{f_{\ell-1}}$.

For every visited node, the weight is accumulated which also constitutes part of the program-state that consumes memory. It is possible to trade in this memory usage at the cost of doing extra calculations in the form of calculating the transitional weight a second time when moving in the reverse direction, and subtracting it from the accumulated weight.

[2] Replacing the need for a *call stack*.

[3] As opposed to storing all 32 or 64 bits which the data type may actually hold.

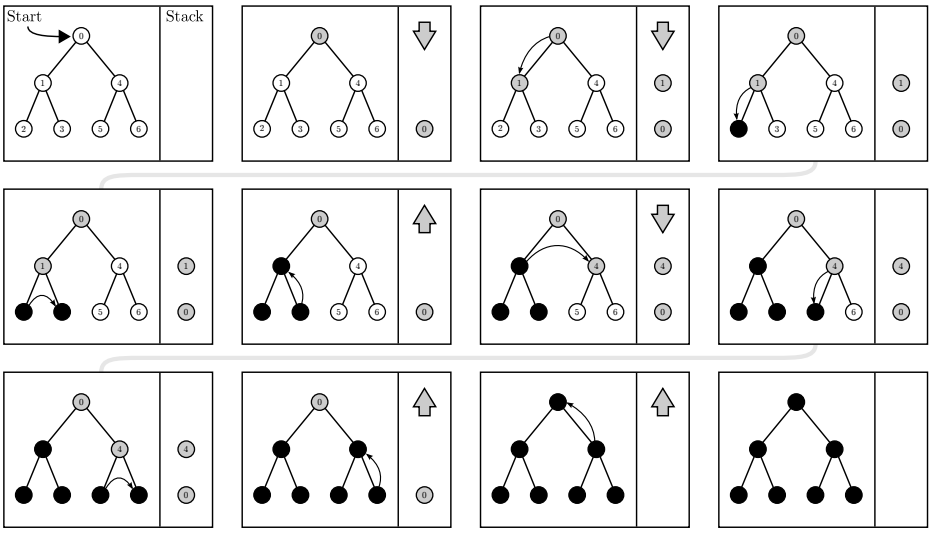


Figure 5.7: DFS using a stack. The grayed nodes are stored on the stack (on the right) and the black nodes represent processed nodes.

Algorithm 1 C-inspired program listing realizing $\sigma_{\ell+1}$ and $\sigma_{\ell-1}$ using IDCT. The operations <<, >>, | and & are the bit-wise operations `SHIFT LEFT`, `SHIFT RIGHT`, `OR` and `AND` respectively.

```
state_next(state, u, nbr_input_bits, delay_elements)
{
    i = (u << delay_elements) | state;
    m = (1 << nbr_input_bits) - 1;

    bitstack_push(i & m);

    return (i >> nbr_input_bits);
}

state_prev(state, nbr_input_bits, delay_elements)
{
    t = bitstack_pop();
    i = (state << nbr_input_bits) | t;
    m = (2 << delay_elements) - 1;

    return (i & m);
}
```

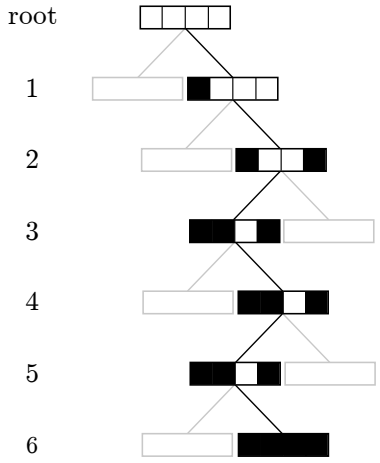


Figure 5.8: Processing a code tree using Singe Instruction, Multiple Data.

Additional features of the IDCT are that it is suitable for work-sharing where one processor interrupts another one to assist in computing the same problem. It also enables storing and restoring execution-state quickly using very little memory[4], should an execution need to be interrupted and resumed.

5.3.6 The Result Set

Using a straight-forward SIMD-based implementation, one might imagine a four fold[5] decrease in the time needed for obtaining the corresponding sets. In reality however, branches terminate at different depths while processing is continued until the last of the four simultaneously processed branches is terminated as shown in Figure 5.8.

This is not an issue in the scalar implementation of the BEAST, as all states in the resulting sets are obtained in serial. For a SIMD implementation however, four states are processed at a time. Simply storing the `vectors` would result in "false" states being stored. Therefore these states are filtered out by setting them to zero before storing, which signals that the states actually do not belong to the result set[6].

However as the extra zero-states still consume memory, the implementation suitably removes them as early as possible. Filtering out the extra zero-states greatly improves performance by reducing the size of the result set and in turn, reducing the amount of information needed to be sorted, copied and matched.

[4] E.g., by flushing the bit-stack, the current state $\sigma(\xi)$ and the accumulated weight $w(\xi)$ to a storage medium.

[5] Using 32-bit data types.

[6] By definition in (3.4) and (3.5).

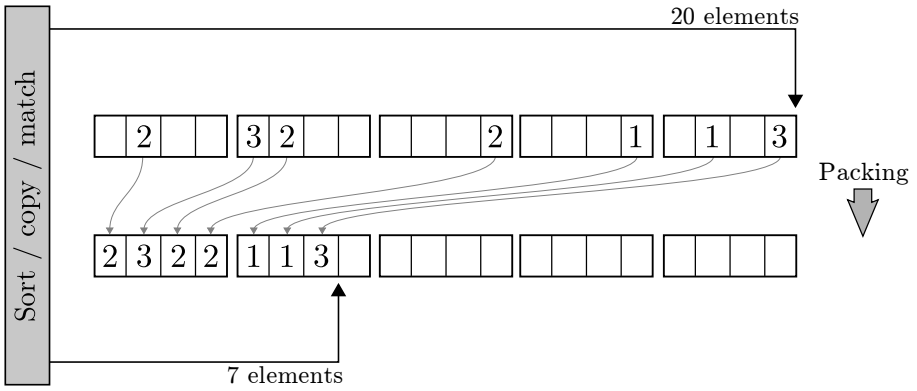


Figure 5.9: The difference in memory requirements between unpacked and packed data. The empty positions are introduced by processing the code tree in SIMD.

Subsets are generated by the BEAST algorithm in sets of $j = 0, \ldots, c-1$. Merging j and the state σ before storing the information in the result set allows us to handle several smaller subsets as one big set. The program listing in Algorithm 2 details how merging and subset extraction is carried out to and from an element in the result set. See Section 5.3.7 below for details on selecting the data type with sufficient bit positions.

Sorting is performed on the SPU using the quick-sort algorithm introduced by C.A.R Hoare [6]. A preferable replacement for the default quick-sort algorithm, available through glibc[7], may be given by the Bitonic sorting algorithm [5], reducing the number of branching instructions and hence increase speed.

Matching is done on the PPU on subsets split into 16 KB sized partitions. The reason for this is given by the fact that the MFC transfers at most 16 KB per operation, making this a natural boundary for partitioning the result set. Additionally, this enables the SPU to sort each partition prior to issuing the DMA command.

5.3.7 Data Types

Selecting an appropriate data type size for the state representation is crucial. Using more bits than required consumes unnecessary amounts of memory and may adversely affect the performance of the implementation[8]. As mentioned in Section 5.3.6, the information regarding which set a state belongs to is encoded into

[7] The GNU C Library.

[8] For example by doing 64-bit calculations on a 32-bit system.

Algorithm 2 The function `encode_subset` creates an element by merging the `state` and the subset information (`j`). The function `decode_subset` extracts the subset information from a previously merged element. The operations `<<`, `>> and |` are the bit-wise operations `SHIFT LEFT`, `SHIFT RIGHT` and `OR` respectively.

```
encode_subset(state, j, delay_elements)
{
    return ((j << delay_elements) | state);
}

decode_subset(element, delay_elements)
{
    return (element >> delay_elements);
}
```

the state itself to allow unique identification of each state in the different j sets (a state may appear multiple times in one or both $\mathcal{F}_{+j}$ and $\mathcal{B}_{-j}$ sets). To represent a state in a code tree, generated by an encoder of rate $R = 1/2$ and memory m, $m+1$ bits are needed. Moreover, additional b bits have to be stored temporarily during a transition, as well as the set-information of $\lceil \log_2(c) \rceil$ bits during storage, leading to an upper bound of

$$\mathrm{T}_{n_{\mathrm{bits}}} \leq m + 1 + \max\left\{\lceil \log_2(c) \rceil, b\right\} \tag{5.7}$$

where $\mathrm{T}_{n_{\mathrm{bits}}}$ is the amount of bits for every member data type in the result set.

Example
A rate $R = 1/2$ convolutional code for memory $m = 26$, which is the topic of this thesis, yields $\mathrm{T}_{n_{\mathrm{bits}}} = 26 + 1 + \max\{1, 1\} = 28$. The required amount of bits exceed that which is available in a 16-bit `short`, but needs less than a 32-bit `int`; hence an unsigned 32-bit `int` is used.

Chapter 6
Results

As generating all matrices for a given memory m is closely coupled with the BEAST, it is of great importance to aim for an optimal computation time, both with regard to BEAST and the rejection rules.

Three versions of the GNU Compiler Collection (gcc) were used in compiling the C source code:

- Synergistic Processor Unit: spu-gcc (GCC) 4.1.1
- PowerPC Processor Unit: ppu-gcc (GCC) 4.1.1
- Generic platform: gcc 4.3.0

6.1 Row Distance-Sieve

Figure 6.1 and 6.2 illustrates the efficiency of the row distance-sieve on the total ensemble of rate $R = 1/2$ generator matrices for some memory m. In Figure 6.1 all memory 16 generator matrices, with the maximum free distance $d_{\text{free}}^{\max} = 20$, have been generated for three target distances $d_{\text{free}}^{\text{t}} = \{19, 20, 21\}$. Note, that the maximum free distance $d_{\text{free}}^{\max}$ is only available *after* performing an exhaustive code search. In Figure 6.2 all generator matrices have been generated with a target distance equal to the maximum free distance, i.e., $d_{\text{free}}^{\text{t}} = d_{\text{free}}^{\max}$. In both figures the distance-sieve has been applied for order $j = 0, 1, 2, \ldots, 15$ and $j = 0, 1, 2, \ldots, 20$ after the initial limitations (c.f., Section 3.1).

Let $n_{d_j^{\text{r}}}$ denote the number of generator matrices surviving the row distance-sieve at order j, and $n_{d_{\text{free}}^{\text{t}}}$ denote the number of generator matrices with the target free distance $d_{\text{free}}^{\text{t}}$. As illustrated in Figure 6.1, $n_{d_j^{\text{r}}}$ decreases rapidly for a low order and converges to the total number of generator matrices with free distances ranging from $d_{\text{free}}^{\text{t}}$to $d_{\text{free}}^{\max}$, that is,

$$\lim_{j\to\infty} n_{d_j^{\text{r}}} = \begin{cases} 0, & \text{d}_{\text{free}}^{\text{t}} > d_{\text{free}}^{\max} \\ n_{d_{\text{free}}^{\text{t}}} + n_{d_{\text{free}}^{\text{t}+1}} + \ldots + n_{d_{\text{free}}^{\max}}, & \text{d}_{\text{free}}^{\text{t}} \leq \text{d}_{\text{free}}^{\max} \end{cases}. \tag{6.1}$$

Clearly, as seen in Figure 6.1, $d_{\text{free}}^{\text{t}}$ has a big impact on the number of surviving matrices at each jth order. Choosing a too optimistic target distance $d_{\text{free}}^{\text{t}}$, will

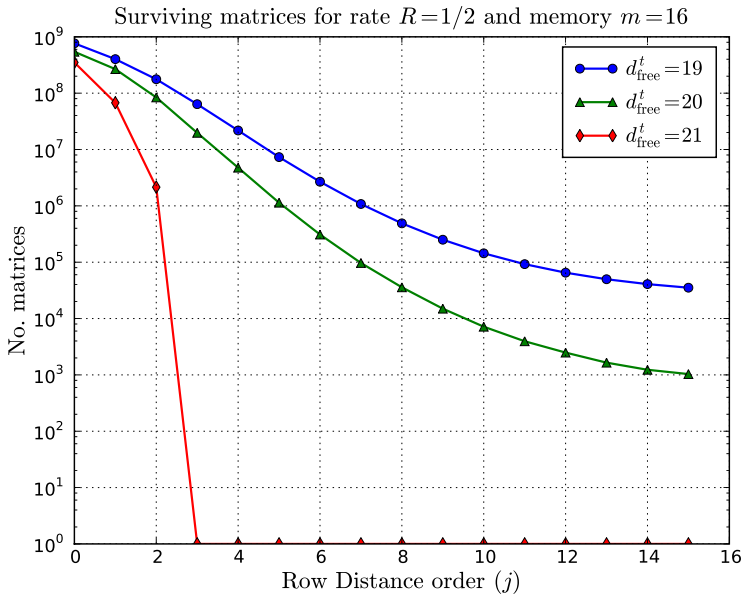


Figure 6.1: The number of surviving generator matrices for three free distance estimates, where $d^{\max}_{\text{free}} = 20$.

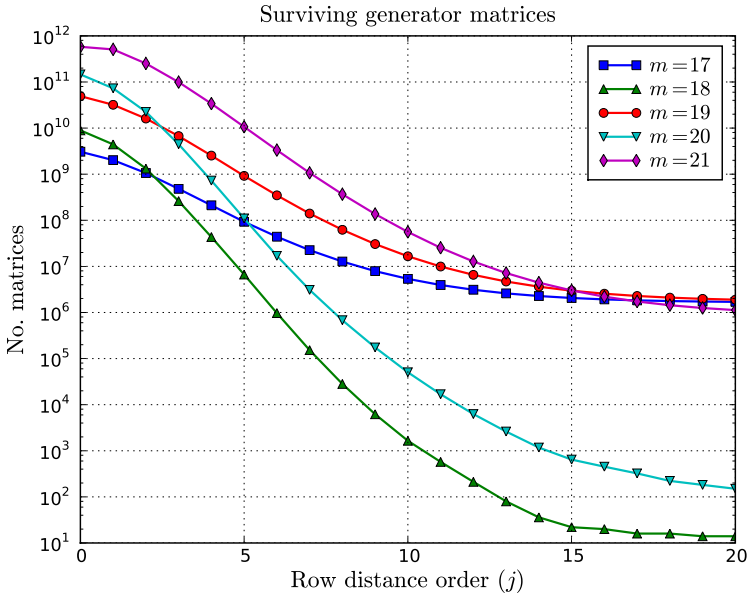


Figure 6.2: The number of surviving generator matrices for some memory m.

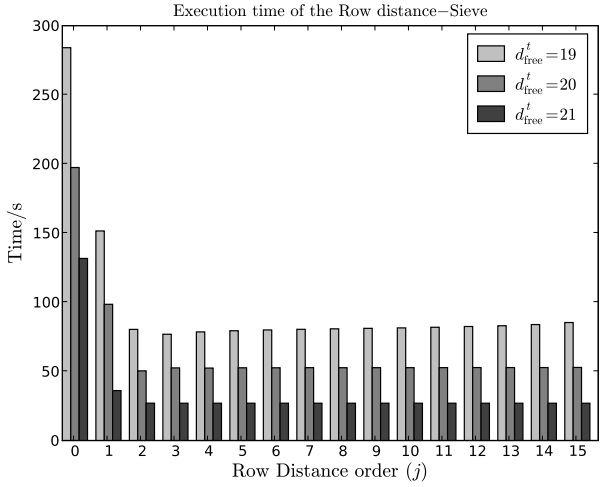


Figure 6.3: Execution time of the row distance-sieve for three estimates on d_{free}, where $d^{\max}_{\text{free}} = 20$.

result in no surviving matrices, i.e., $n_{d^r_j} = 0$ (c.f. $d^{\text{t}}_{\text{free}} = 21$ in Figure 6.1). Note, we have used previously published results to measure and illustrate the performance of the implemented rejection rules.

6.1.1 Execution Time

Intuitively one might suspect that the execution time, for an increasing jth order of the row distance, would increase rapidly since the number of input doubles at each order (i.e., extending the code tree one more level). Figure 6.3 illustrates the execution time needed to generate $n_{d^{\text{r}}_j}$ with $j = 0, 1, \ldots, 15$, for the rate $R = 1/2$ and $m = 16$ encoder ensemble using $d^{\text{t}}_{\text{free}} = \{19, 20, 21\}$. However, for a sufficiently small order, i.e., $j \lesssim 16$ the execution time is governed by $n_{d^{\text{r}}_j}$ and not the order j (c.f. Figure 6.2). Consequently, it is only when $n_{d^{\text{r}}_j}$ approaches $\left(n_{d^{\text{t}}_{\text{free}}} + n_{d^{\text{t}+1}_{\text{free}}} + \ldots + n_{d^{\max}_{\text{free}}}\right)$, that the execution time will increase at each larger jth order.

Consider for example $d^{\text{r}}_j = 0$ in Figure 6.1, where the amount of surviving generator matrices is approximately $n_{d^r_0} = 10^9$. A generator matrix $G = (g_{11}\, g_{12})$ for a rate $R = 1/2$ convolutional code with memory $m < 32$ can be represented with two 32-bit data types, that is, 8 bytes. 10^9 generator matrices would need $10^9 \cdot 8 = 8$ GB of data to be stored for later processing. Therefore, the extra time penalty during execution for $d^{\text{r}}_j = \{0, 1, 2\}$ in Figure 6.3 is due to the extra overhead (i.e., file I/O) needed when storing each generator matrix on disk.

For comparison and to illustrate how the SPU instruction set can be utilized in computations, a simple implementation of the row distance-sieve was done

Figure 6.4: Comparison between one SPU core and one AMD Athlon 64 X2 Dual Core 5200+.

for the SPU and x86 architecture. Sample code for the SPU implementation[1] is included in A.2. Figure 6.4 illustrates execution times for the two different architectures, where both implementations was run on a single core. The complete ensemble of generator matrices was filtered with the row distance-sieve at depth $j = 10$ (d^{r}_{10}) for $m = 13, \ldots 20$, and the target distances was selected as the known maximum free distance, $d^{t}_{\mathrm{free}} = d^{\max}_{\mathrm{free}}$. It is clear that for memories with a large amount of surviving matrices (e.g., $m = 17$ or $m = 19$) the SPU implementation gains a lot of performance by using SIMD instructions.

6.2 Running the BEAST

In the following sections, two sample sets are used for illustrating BEAST-implementation specific properties. Both sample sets, let us call them $\mathcal{R}_A$ and $\mathcal{R}_B$, are made up of 5,000 and 500,000 respectively, randomly selected memory 26 encoders. The encoders are selected uniformly from the entire ensemble using the R250 random number generator introduced by Kirkpatrick and Stoll [11], which has a period of almost 2^{250}. The results below are presented using a 32-bit data type, meaning *four* simultaneously handled code tree branches when using a 128-bit `vector` type.

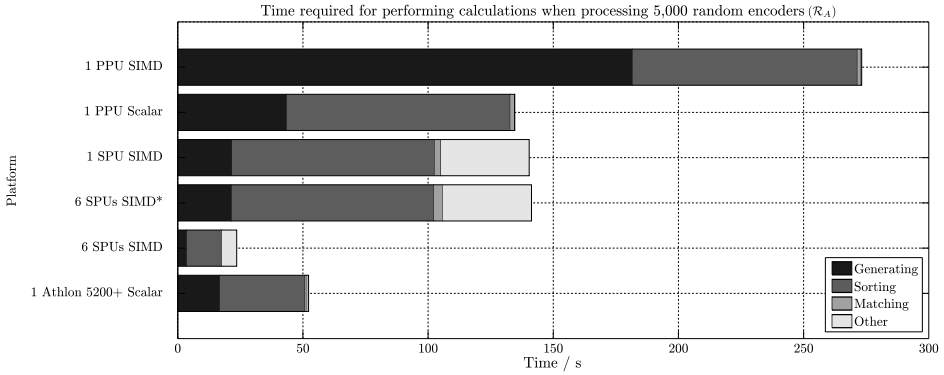


Figure 6.5: Required processing time for the random set of 5,000 encoders in $\mathcal{R}_A$. The data marked with an asterisk (*) shows the accumulated time of all 6 SPUs.

6.2.1 Algorithmic Components

Illustrated in Figure 6.5 is the work-load distribution of the three major algorithmic components, namely *generating*, *sorting* and *matching*. The time spent outside of those areas are summarized in *other*. The figure reveals that the PPU SIMD implementation requires much more time in generating the result set than all the other implementations. As is detailed in Appendix A.1, there is no *single* SIMD instruction for counting bits on the PPU[2] and hence no cheap[3] way of calculating the Hamming weight of elements of a `vector` type. This means that every time the Hamming weight of a `vector` type is calculated, the scalar method for calculating the weight is applied, four times. Combined with having to calculate weights that are immediately discarded (as detailed in Section 5.3.6) this yields a heavy penalty.

Evident from Figure 6.5 and the SPU implementation, is that a significant amount of time (approximately 25%) makes up *other*; which includes tasks such as inter-processor communication and synchronization.

The input data is exactly the same ($\mathcal{R}_A$) on all platforms, yielding exactly the same result sets regardless implementation. As expected, the result sets require the same amount of time to be sorted on a core, regardless of how they were generated. It can be noted that the SPU `qsort` implementation has matching performance to the PPU implementation.

The binary running on the "Athlon 5200+" core is compiled from identical source code used in creating the "PPU Scalar" binary. Although different versions of GCC were used in compiling code for the PPU and the Athlon, the performance on the Athlon is attributed to it being of a different and more recent design than the Cell processor.

[1]The x86 implementation using scalar instructions is conceptually identical.
[2]As is the case with the SPU.
[3]Cheap in terms of a single or very few instructions.

6.2.2 Performance and Memory Footprint

Figure 6.6 presents a comparison between the total time needed for processing the 5,000 encoders in $\mathcal{R}_A$ using a recursive and an iterative implementation. Although the recursive method outperforms the iterative in all implementations, the iterative method is a much more suitable basis for an implementation calculating higher memories, particularly in memory-starved environments such as on the SPU.

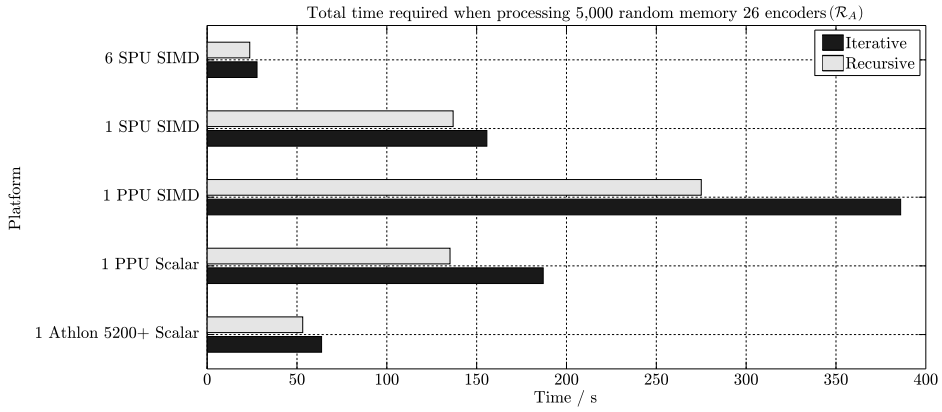


Figure 6.6: Plot of execution time using different implementations for calculating the free distance.

Our implementation of the recursive method uses 160 bytes for every depth (ℓ) in the code tree, when handling four simultaneous branches. The iterative method uses considerably less stack memory with a footprint of 10 *bits* of memory for every depth in the code tree (a factor 128 less than the recursive). The mean stack memory usage when processing the 500,000 encoders in $\mathcal{R}_B$, using the recursive and the iterative method, is illustrated in Figure 6.7. The figure also shows the distribution of stack usage for the recursive method, with an upper limit of roughly 16 KB. Note that the maximum free distance in $\mathcal{R}_B$ is 26, and is therefore likely not to include the worst case scenario with the maximum stack usage needed in performing the exhaustive code search for memory $m = 26$.

In order to utilize the local store effectively, a buffered DMA strategy is used, which triggers a MFC-put operation every time a 16 KB chunk has been generated[4]. Using 6 buffers of 16 KB each, a total of 96 KB is needed, leaving room for a recursive method and its use of a call-stack. Note that even though the size of the local store suffices for the recursive method with memory $m = 26$ for the rate $R = 1/2$ encoders, it may quickly present a problem for higher memories as there is no way of predicting how deep a tree needs to be processed.

[4] 16 KB is the maximum amount of data a single DMA operation can issue a transfer of.

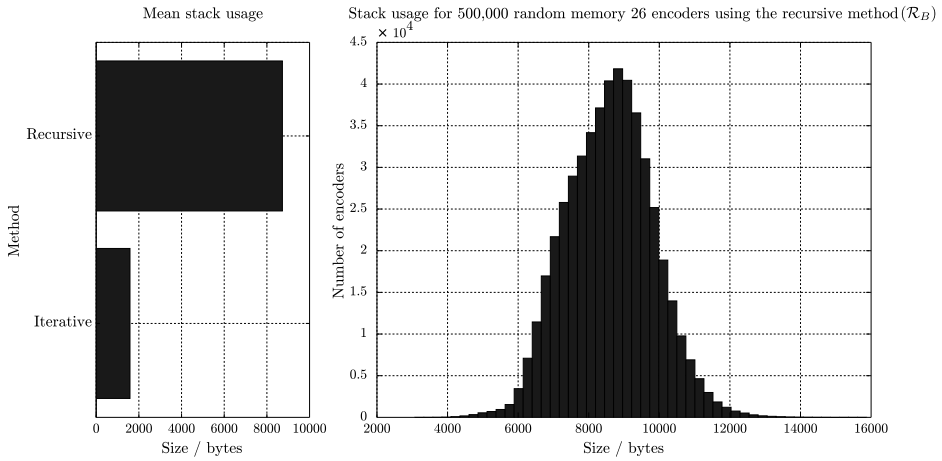


Figure 6.7: The stack usage of the 500,000 random encoders ($\mathcal{R}_A$) when finding the best generator matrix using the spectral components.

6.2.3 Scaling

Figure 6.8 illustrates is the BEAST implementation, running on a varied amount of SPUs. The figure shows that for every doubling of the amount of processor elements, the time needed for calculations is *nearly* halved.

6.3 Exhaustive Code Search

As described in Section 6.1, choosing the right parameters for the row distance-sieve has a great impact on the computational time and surviving generator matrices. Table 6.1 lists generator matrices for memory 16 to 29, where some results were verified by us, and memory $m = 23$ to $m = 25$ are OFD generator matrices from previously published results [7]. By considering for example $m = 24$ or $m = 25$ in Table 6.1, a naive estimate of the free distance for memory $m = 26$ could be $d_{\text{free}}^{\text{t}} = 29$. However, running the row distance-sieve with such a target distance resulted in *no* surviving matrices[5]. We thereby can conclude that the rate $R = 1/2$ OFD generator matrix of memory $m = 26$ has a free distance of $d_{\text{free}} = 28$ while its generator polynomials are still unknown. Meanwhile an exhaustive code search with target free distance $d_{\text{free}}^{\text{t}} = 28$ for memory $m = 26$ was started, leading to the preliminary results marked "$*$" in Table 6.1, where a generator matrix with better distance spectrum was found.

Furthermore, by selecting a subset within one of the "hot" areas (c.f., Section 6.5) for memories 27, 28 and 29. Then performing a search within each sub-

[5]Using five PS3 consoles the exhaustive code search completed in approximately 12 days.

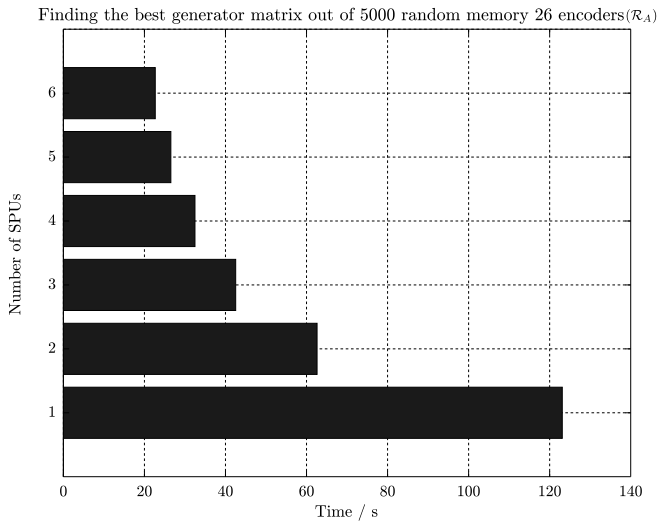


Figure 6.8: Plot of time needed for finding the best generator matrix in $\mathcal{R}_A$ by calculating spectral components, using a varied amount of SPUs.

set containing only $1/\left(2 \cdot 16^4\right)$ of the original ensemble. We were able to obtain generator matrices that are better than the previously known optimum distance (ODP) generator matrices (e.g., in [7]), these new results are marked by "$\circ$" in Table 6.1. Note, their OFD property has not been determined yet.

6.4 Time Trade-Offs

Consider the generator matrices with memory $m = 17, 19, 21$ in Table 6.1 which have the same free distance d_{free} as memory $m = 16, 18, 20$ respectively. In such cases a large amount of generator matrices survives the row distance rejection rules, leading to a greatly increased run-time. On the other hand, if the free distance d_{free} increases with increasing memory, only a few generator matrices pass the rejection rules, leading to a shorter running time.

However, choosing the target free distance too optimistic results in no surviving generator matrices, as for example illustrated in Figure 6.1 with $d_{\text{free}}^t = 21$.

Note, the row distance-sieve is just the first step in an exhaustive code search and all surviving encoder matrices need to be checked with the BEAST in order to determine their actual d_{free} (and spectral components). Therefore one should switch from the row distance-sieve to BEAST at the right point, to achieve an optimal overall running time on the exhaustive search. Figure 6.9 shows the "golden cross", by using a small subset (e.g., the first sub-partition q_1 with $n_q = 131072$) for memory $m = 26$, where the computational time for the row-distance sieve will exceed the computational time for the BEAST.

m	$g_1(D)$	$g_2(D)$	d_{free}	$i=0$	1	2	Remarks
16	626656	463642	20	43	0	265	OFD
17	611675	550363	20	4	24	76	OFD
18	4551474	6354344	22	65	0	349	OFD
19	7504432	4625676	22	5	52	116	OFD
20	6717423	5056615	24	145	0	225	OFD
21	63646524	57112134	24	17	95	136	OFD
22	64353362	41471446	25	47	88	137	OFD
23	75420671	45452137	26	45	0	365	OFD [7]
24	766446634	540125704	27	50	135	118	OFD [7]
25	662537146	505722162	28	71	196	112	OFD [7]
26	727322321	424667027	28	11	60	150	$*$
27	6276631214	5475602164	29	19	63	185	$\circ$
28	5762423076	6005305632	30	53	0	341	$\circ$
29	5713575517	6026566375	31	64	164	68	$\circ$

Table 6.1: The currently best known rate $R = 1/2$ generator matrices and spectral components $n_{d_{\text{free}}+i}$.

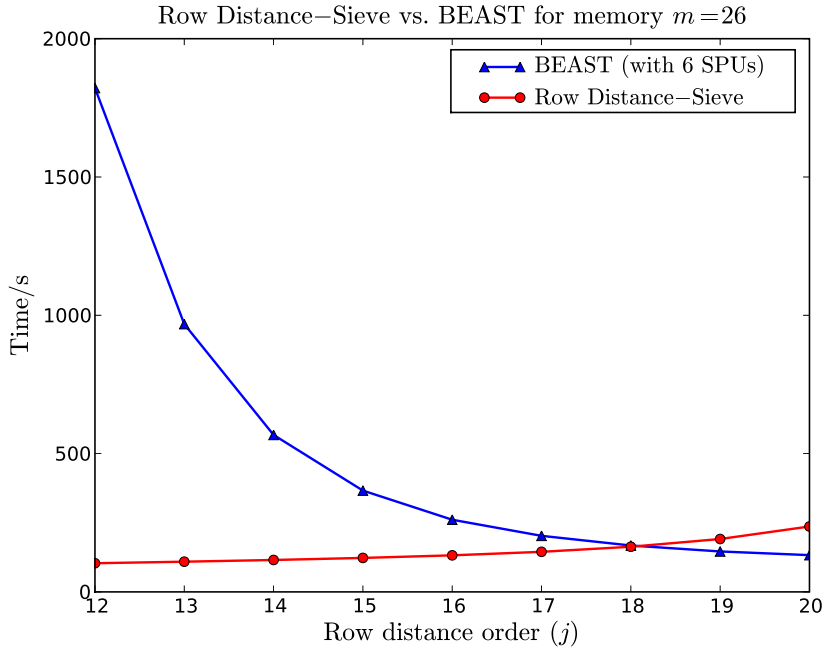


Figure 6.9: The golden cross between the BEAST and row distance-sieve.

m	$n_{d_{\text{free}}^{\max}}$
16	200
18	8
19	104295
20	12
22	2
23	28692
24	22

Table 6.2: Generator matrices with the maximum free distance.

6.5 Hot Encoders

An exhaustive code search for a rate $R = 1/2$ convolutional codes with memory m will result in a number of generator matrices with the maximum free distance, $n_{d_{\text{free}}^{\max}}$. Table 6.2 lists these numbers for some values of m, as a result of an exhaustive code search.

Using their polynomial representation (c.f. Section 2.4), each generator sequence in $G = (g_{11}\ g_{12})$ for a rate $R = 1/2$ convolutional code with memory m will in binary notation satisfy

$$2^m \leq g_{1j} < 2^{m+1},\ j = 1, 2. \tag{6.2}$$

Example
A rate $R = 1/2$ generator matrix $G = (74\ 54)_8$ with memory $m = 3$, can be represented with decimal numbers as

$$G = (111100\ 101100)_2 = (1111\ 1011)_2 = (15\ 11)_{10}$$

and thus, with $g_{11} = 15$ and $g_{12} = 11$, (6.2) is satisfied.

Pixel Representation
A function κ is used to transform a polynomial g of memory m into a pixel coordinate

$$\kappa = \left\lfloor \left(\frac{g}{2^m} - 1 \right) \cdot p \right\rfloor \tag{6.3}$$

in the pixel space $p \times p$.

6.5.1 Heat Maps

A heat map is a graphical visualization of two-dimensional data. Areas of the map will be colored more intensely if there is more data in that particular region. If the complete ensemble of generator matrices for a memory m is rasterized with an arbitrary raster size (e.g., 256) and each generator $G = (g_{11}\ g_{12})$ with $d_{\text{free}}^{\max}$ is mapped onto the rasterized data matrix, a pattern occurs if $n_{d_{\text{free}}^{\max}}$

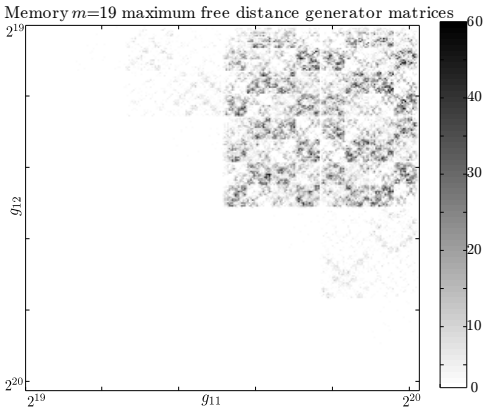


Figure 6.10: Visualizing maximum free distance generator matrices for memory $m = 19$.

is large enough (i.e., $n_{d_{\text{free}}^{\max}} \gtrsim 1000$). Figure 6.10 shows a heat map where all rate $R = 1/2$ generator matrices for memory $m = 19$ with the maximum free distance are visualized, with each pixel representing an equal portion of the complete encoder ensemble. A triangular shape appears because of the symmetry between $G(D) = (g_1(D)\ g_2(D))$ and $G'(D) = (g_2(D)\ g_1(D))$ which have equivalent properties, and hence only $G(D)$ is considered. The dark areas indicate dense clusters of generator matrices generating codes with the maximum free distance $d_{\text{free}}^{\max}$. A rate $R = 1/2$ generator matrix $G = (g_{11}\ g_{12})$ with memory m that satisfy the following numerical relations

$$\begin{aligned} 2^m + 2^{m-1} < \quad & g_{11} < \quad 2^{m+1} \\ 2^m < \quad & g_{12} < \quad 2^m + 2^{m-2} \end{aligned} \tag{6.4}$$

will be located within a more dense clustered area, as observed in Figure 6.10. By graphically representing matrices with the maximum free distance for other memories the same patterns appear of different intensity, depending on $n_{d_{\text{free}}^{\max}}$. With memory $m = 16$ for instance, with only $n_{d_{\text{free}}^{\max}} = 200$, the pattern is not as apparent as for $m = 19$, but most matrices with the maximum free distance are still clustered within in the same area (i.e., satisfy (6.4)) as illustrated in Figure 6.11. This also leads to the suggestion on focusing in this area when performing a random code search.

Another promising result is that a pattern appears to repeat itself when the resolution is increased, as illustrated in Figure 6.12. In this example a square is "zoomed" in, illustrating a similar pattern as the original, but now only covering a small subset of the original encoder ensemble. Furthermore, by repeating this process several iterations, it is possible to greatly reduce original number of generator matrices and conduct a heuristic code search.

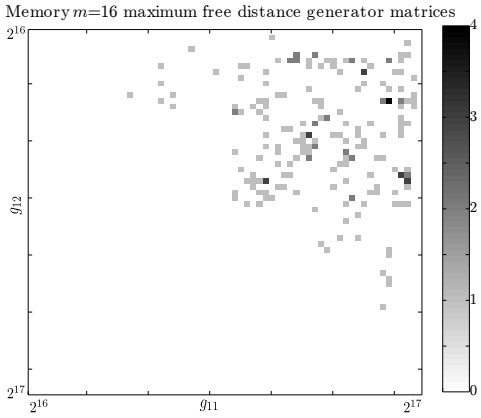


Figure 6.11: Visualizing maximum free distance generator matrices for memory $m = 16$.

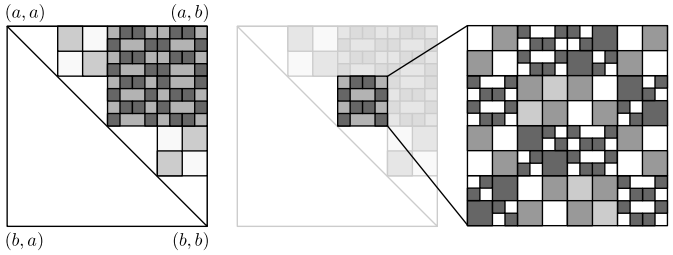


Figure 6.12: Illustration of the *hot* encoder pattern observed for memory $m = 19$, including the pattern appearing when the resolution is increased. In polynomial space, $a = 2^m$ and $b = 2^{m+1} - 1$. In the $p \times p$ pixel space, a and b represent 0 and $p - 1$, respectively.

Chapter 7
Conclusions

The Cell Broadband Engine Architecture was utilized in exhaustive code search for a rate $R = 1/2$ convolutional code with the best possible free distance and spectra. Because of the huge size of the full memory 26 ensemble, their properties were used to reduce the size of the set; therein an algorithm exploiting the row distance property. The BEAST - Bidirectional Efficient Algorithm for Searching code Trees - was applied to the greatly reduced ensemble in order to finally determine the free distance and spectra was presented. Preliminary results in the exhaustive memory 26 code search include an encoder with the currently best known free distance and spectra.

Additionally, a pattern in the distribution of encoders with preferable properties was discovered and used to find encoders with *better* properties than previously known for memories 27, 28 and 29.

The Cell architecture is very complex and is only suitable for certain types of problems. We have shown that the problem of exhaustively searching for convolutional codes is a task well suited for the architecture due to its highly parallelizable nature.

An iterative method for overcoming the immediate memory restriction on the Synergistic Processor Units has been presented, especially useful when processing larger memories.

Outlooks

We have shown that the algorithms used in an exhaustive code search can be successfully implemented effectively on the Cell Broadband Engine Architecture. In order to be able to perform an exhaustive code search using encoders with properties not bounded by the data type in (5.7), a 64-bit data type would need to be used.

The evidence in Section 6.2.1 exposes the need for further optimizations, such as using a different scheduling. If possible, moving the *matching* out of the SPU's critical path should provide ample speed-up. Further optimization can be carried out by exchanging the quick-sort sorting method with a Bitonic sorting method, reducing the amount of branching on the SPU and increasing speed.

Finally, the pattern of ”hot encoders” warrants an in-depth investigation, as the evidence in this thesis hints at properties that might be used in order to quickly find codes of larger memories with desirable properties.

Bibliography

[1] Jack Dongarna Alfredo Buttari and Jakub Kurzak. Limitations of the playstation 3 for high performance cluster computing. Technical report, Tech. rep., University of Tennessee Computer Science, 2007.

[2] Irina E. Bocharova, Marc Handlery, Rolf Johannesson, and Boris D. Kudryashov. A BEAST for prowling in trees. In *Proc. 39th Annual Allerton Conf. Commun., Control, and Computing*, 2001.

[3] Irina E. Bocharova, Marc Handlery, Rolf Johannesson, and Boris D. Kudryashov. A BEAST for prowling in trees. *IEEE Transactions on Information Theory*, 50(6):1295–1302, 2004.

[4] Irina E. Bocharova, Florian Hug, Rolf Johannesson, and Boris D. Kudryashov. A note on convolutional codes: Equivalences, macwilliams identity, and more. submitted to IEEE Trans. on Inf. Theory.

[5] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. Cellsort: high performance sorting on the cell processor. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1286–1297. VLDB Endowment, 2007.

[6] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[7] Florian Hug. On Graph-Based Convolutional Codes. Master's thesis, Lund University, Lund, Sweden, 2008.

[8] IBM. The cell architecture, innovation matters. Webpage, June 2009. http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.in- novation.html.

[9] Rolf Johannesson and K. S. Zigangirov. *Fundamentals of Convolutional Coding*. Wiley-IEEE Press, 1999.

[10] Nedal Kafri and Jawad Abu Sbeih. Simple near optimal partitioning approach to perfect triangular iteration space. In *HPCS*, 2008. http://cisedu.us/storage/hpcs/2008/nkafri_at_science.alquds.edu-2008.04.16-06.04.27-hpcs08ahg_Kafri_Jawad_April_16.pdf.

[11] Scott Kirkpatrick and Erich P. Stoll. A very fast shift-register sequence random number generator. *Journal of Computational Physics*, 40(2):517–526, April 1981.

[12] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[13] Maja Lončar. *Taming of the BEAST*. PhD Thesis, Lund University, 2007. ISBN 9171670459.

[14] J.L. Massey and M.K. Sain. Inverses of linear sequential circuits. *IEEE Transactions on Computers*, 17(4):330–337, 1968.

[15] P.Elias. Coding for noisy channels. In *IRE Conv. Record*, volume 4, pages 37–47, 1955.

[16] IBM Redbooks. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. Vervante, 2008.

[17] Matthew Scarpino. *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[18] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, 1967.

Appendix A

Sample Code and Comparisons

A.1 Hamming Weight

The Hamming weight function is used extensively when running the BEAST. The following sample code and comparisons aims to illustrate the importance of effectively determining the hamming weight during calculations, which is illustrated in Figure A.1 for different platforms.

Normally, one of the most effective ways to determine the hamming weight is to use a lookup-table. This was used in the experimental test runs in Figure A.1, except for the SPU which provides an internal instruction for counting bits.

SPU SIMD

```
#define spu_hamweight(va) ((vec_uint4)\
  (spu_sumb(spu_cntb((vec_uchar16) (va)),\
   spu_splats((uint8) 0))))
```

Scalar

```
static inline uint8_t hamweight(uint32_t a)
{
  uint8_t c = 0;

  for (c=0; a; c++)
    a &= (a - 1);

  return c;
}

#define hamweight(a) (__hw_lookup_table[(a)>>16] +\
                      __hw_lookup_table[(a)&0xffff])

#define hamweight(a) __builtin_popcount((a))
```

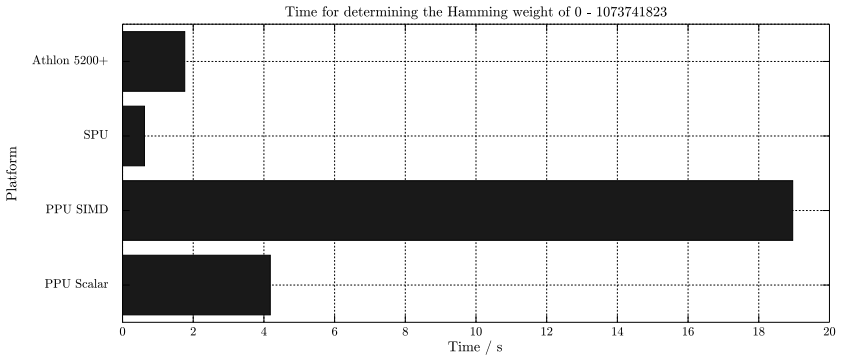


Figure A.1: Time needed to determine Hamming weight (using a 16-bit lookup-table on all platforms but the SPU).

A.2 Simple Row Distance-Sieve Implementation

SPU Code

```
inline uint32_t spu_row_rejection(uint32_t target,
  uint32_t sequences,
  vec_uint4 g1,
  vec_uint4 g2)
{

  uint32_t i,j,len;
  vec_uint4 zeros = spu_splats((uint32_t)0);

  for (i = 1; i < sequences; i += 8) {

    vec_uint4 t = {i,i+2,i+4,i+6};
    vec_uint4 v0 = zeros;
    vec_uint4 v1 = zeros;
    len = 32-lz(i+6);

    for (j = 0; j < len; j++) {
      v0 = spu_sl(v0,1);
      v1 = spu_sl(v1,1);

      vec_uint4 bitmask = spu_and(t,spu_splats((uint32_t)1<<j));
      vec_uint4 xormask = spu_cmpgt(bitmask,zeros);

      v0 = spu_xor(v0,spu_sel(zeros,g1,xormask));
      v1 = spu_xor(v1,spu_sel(zeros,g2,xormask));
    }
```

```
    vec_uint4 weights = spu_add(spu_hamweight(v0),
        spu_hamweight(v1));

    if (__builtin_expect(!spu_allgt(weights,target-1)),1)
      return 0;
  }

  return 1;
}
```