Massively Parallel Quantization Implementation
Using Simulated Annealing

by

L. van Engelen

l.vanengelen@student.utwente.nl

Supervisors:
Dr. Ir. G.J.M. Smit (University of Twente)
Ir. E. Molenkamp (University of Twente)
Ir. J.W.M. Jacobs (Océ)
Drs. Z. Goey (Océ)

July 2006
University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architecture, Design and Test for Embedded Systems Group
Abstract

This thesis treats the mapping of a quantization algorithm on the Linedancer parallel processor architecture which was done at Océ. This algorithm is based on Simulated Annealing and uses a Markov Random Field image model. The mapping of the algorithm was supported by the use of the Evolutionary Design methodology.

The mapping of the algorithm was finished successfully, although the output quality was worse than that of established algorithms. Throughout development an executable prototype was used to guard the quality of the mapping. We identified four typical subphases in the top down part of the Evolutionary Design methodology.

The implementation of the quantization algorithm benefits from the implementation on the Linedancer, demonstrated by a higher processing speed when compared with a Pentium.

Finally some directions for additional research are suggested.
# List of Figures

2.1 Typical office scan: text and charts ........................................... 13
2.2 Image degradation: instead of just a couple of solid gray values, the image is blurry and displays some sort of patchwork pattern. The original image consisted of only 10 different grays, but scanning has increased this to 229. 14
2.3 Probabilities $P(y_i | q_j)$ for the case of quantization to 4 different classes ($N = 4$). 15
2.4 Approximation of a discrete probability $P(X = a)$ based on the continuous probability distribution $p(x)$. 15
2.5 Examples of images of low and high regularity .................................. 16
2.6 Image pixel (the black dot) with its neighborhood (gray dots) and corresponding set $C$ of clique pairs. Each clique has its own color. 16
2.7 Fidelity and regularity examples ................................................. 18
2.8 Acceptance probability at different temperatures ............................ 20

3.1 Path of evolutionary design through the design space ....................... 25
3.2 Transformational development .................................................. 26
3.3 Overview of Evolutionary Design: (Sub)phases are passed through from left to right in time. 26
3.4 Increasing portability by using common code templates. Algorithms are implemented in terms of these code templates. For each code template there is an implementation on each architecture. 29

4.1 Expanding a mesh (a) requires more Processor Elements (PEs) than expanding a string (b) .......................................................... 32
4.2 Transfers inside the memory hierarchy of the Linedancer ................. 33
4.3 Associative memory in action ................................................. 35
4.4 Dependency of the number of cycles on the operand width for the RTS(Mult) and RTS(Div) instructions ........................................... 39
4.5 Dependency of the number of cycles on the operand width for the remote RTS(Assign) instruction ................................................. 39

5.1 Tiling: the input image is cut into blocks fitting inside the Linedancer and processed in sequence. .................................................. 42
5.2 A 10 bits wide Linear Feedback Shift Register. A 0 is shifted out and 1 $\text{Xor} 0 = 1$ is shifted in. 43

6.1 Comparison of quantizations by (a) a commercial program (Irfanview) and (b) our algorithm ................................. 51
6.2 Tiling artifact in the output of the quantization algorithm. If you look close inside the red rectangle you can see that the blobs have horizontal and vertical borders. These are the tile edges. 53
6.3 Speedup for various image sizes and number of annealing loops ........... 54
List of Tables

4.1 Processor Architecture Taxonomy according to Flynn . . . . . . . 30
4.2 Some of the RTS instructions that can be used on the Linedancer . 36
4.3 Linedancer operations running with a constant number of cycles . 37
4.4 Linedancer operations which are linearly dependent on the bit width. 38

6.1 Number of cycles used for each stage of the algorithm. Nesting indicates loops, bold numbers indicate accumulated results. 52
6.2 Processing time of Data Stream Manager (DSM) and Instruction Stream Manager (ISM) threads per tile when one annealing loop is performed. 52
6.3 Processing time of DSM and ISM threads per tile when ten Simulated Annealing loops are performed. 54
6.4 Running times of the quantization algorithm on a Pentium and on the Linedancer. 55

A.1 J's arithmetic and comparison verbs, together with their equivalents in C and mathematical notation . 65
A.2 Operations on whole arrays . . . . . . . . . . . . . . . . . . . . . . 65
A.3 Miscellaneous verbs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
List of Algorithms

1. Simulated annealing ........................................ 19
2. Modified Metropolis Dynamics .......................... 21
Notation

\(S\)  Set of all pixels in an image.

\(s\)  Pixel in an image.

\(g\)  Quantized image.

\(g_s\)  Value of pixel \(s\) in quantized image \(g\).

\(\mu\)  Mean value.

\(\mu_{g_s}\)  Mean value of input image pixels with label \(g_s\).

\(\sigma\)  Standard deviation.

\(\sigma_{g_s}\)  Standard deviation of input image pixels with label \(g_s\).

\(y\)  Input image.

\(y_s\)  Value of pixel \(s\) in input image \(y\).

\(\delta(g_s, g_r)\)  Function returning \(-1\) if \(g_s\) equals \(g_r\), \(1\) otherwise.

[0,1)  Range of real numbers from 0 inclusive to 1 not inclusive.
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic &amp; Logic Unit: part of a computer processor performing arithmetic and logical functions.</td>
</tr>
<tr>
<td>APL</td>
<td>A Programming Language: programming language developed by K.E. Iverson, predecessor of J.</td>
</tr>
<tr>
<td>CADTES</td>
<td>Computer Architecture Design &amp; Test for Embedded Systems: Research group of the Faculty of Computer Science and the Faculty of Electrical Engineering of the University of Twente in the Netherlands.</td>
</tr>
<tr>
<td>CAM</td>
<td>Content Addressable Memory: memory addressable by its contents instead of a fixed address.</td>
</tr>
<tr>
<td>DSM</td>
<td>Data Stream Manager: Linedancer thread responsible for transferring data between SDS and PDS.</td>
</tr>
<tr>
<td>EXT</td>
<td>Extended Memory: Non-content addressable part of the memory (cf. CAM).</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array: Hardware which is reprogrammable at gate level.</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor: A processor which can be used for arbitrary tasks, such as a Pentium.</td>
</tr>
<tr>
<td>IDSM</td>
<td>Instruction and Data Stream Manager: Processor on which the DSM and ISM threads run.</td>
</tr>
<tr>
<td>ISM</td>
<td>Instruction Stream Manager: Linedancer thread responsible for controlling the processor array.</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear Feedback Shift Register: Shift register which can be used as a semi-random number generator.</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction Multiple Data: Parallel computing, where multiple instructions operate on vectors of data in parallel.</td>
</tr>
<tr>
<td>MISD</td>
<td>Multiple Instruction Single Data: Parallel computing, where multiple instructions operate on a single piece of data.</td>
</tr>
<tr>
<td>MMD</td>
<td>Modified Metropolis Dynamics: parallel optimization of Simulated Annealing.</td>
</tr>
</tbody>
</table>
LIST OF ALGORITHMS

MRF  Markov Random Field: Field over sets of random variables, with neighborhood relations.

PDS  Primary Data Store: Memory used as a buffer for transfers between SDS and PE memory.

PE   Processor Element: Single CPU of a massively parallel processor array.

SCC  System Control Computer: Aspex’s name for the host CPU of the Linedancer, usually a Pentium.

SDMC Secondary Data Movement Controller: Controller responsible for the transfer between SDS and PDS.

SDS  Secondary Data Store: Memory available to the IDSM processor.

SIMD Single Instruction Multiple Data: Parallel computing, where one instruction operates on a vector of data.

SISD Single Instruction Single Data: Nonparallel computing, where one instruction operates on a scalar.

TDS  Tertiary Data Store: Memory available to the SCC processor.

VHDL Very-High-Speed Integrated Circuit Hardware Description Language: Commonly used hardware description language.
# Contents

1 Introduction ........................................ 11
   1.1 Research Question ................................ 11
   1.2 Overview ........................................ 12

2 Quantization ........................................ 13
   2.1 Image Model ...................................... 14
      2.1.1 Fidelity .................................... 14
      2.1.2 Regularity .................................. 16
      2.1.3 Bayesian Model ............................... 17
   2.2 Simulated Annealing ............................... 19
   2.3 Modified Metropolis Dynamics ..................... 20
   2.4 Choosing Parameters .............................. 21

3 Evolutionary Design ................................ 22
   3.1 Current Practice .................................. 22
      3.1.1 OCD ......................................... 22
      3.1.2 CADTES Group ................................ 22
   3.2 Alternatives ...................................... 23
      3.2.1 Blaauw and Brooks ............................ 23
      3.2.2 Bernecky .................................... 24
   3.3 Evolutionary Design ............................... 25
      3.3.1 Incremental Prototyping ....................... 25
      3.3.2 Transformational Development ................ 26
   3.4 Intended Benefits ................................ 28
      3.4.1 Higher Productivity ......................... 28
      3.4.2 Error Recovery ................................ 28
      3.4.3 Portability ................................... 29
      3.4.4 Domain Expert Programming .................. 29

4 Linedancer .......................................... 30
   4.1 Parallel Architectures ............................. 30
      4.1.1 Processor Architecture Taxonomy .............. 30
      4.1.2 Benefits of Parallel Architectures .......... 31
   4.2 Hardware Architecture ............................ 32
      4.2.1 Scalability .................................. 32
      4.2.2 Memory Transfers ............................. 34
      4.2.3 Associativity ................................ 34
      4.2.4 Communication ............................... 34
# B Listings

## B.1 Models

B.1.1 File: functional.ijs  
B.1.2 File: functional4.ijs  
B.1.3 File: functional6.ijs  
B.1.4 File: algorithm6.ijs  
B.1.5 File: algorithm9.ijs  
B.1.6 File: architecture15.ijs  
B.1.7 File: architecture21.ijs  
B.1.8 File: templates2.ijs  

## B.2 Linedancer Code

B.2.1 File: algorithm.defs  
B.2.2 File: algorithm.tpl  
B.2.3 File: ismCode.tpl  
B.2.4 File: dsmCode.c  
B.2.5 File: sccCode.tpl  

## B.3 Python Code

B.3.1 Parser  
B.3.2 File: Parser3.c
Chapter 1

Introduction

Océ is a manufacturer of printers, copiers, scanners and software and a lot of its technology makes heavy use of image processing algorithms. These algorithms are often of a highly parallel nature, e.g. when they perform the same operation on all pixels of a scan.

Currently, these parallel image processing algorithms are prototyped in a sequential language and finally implemented in a parallel one. This is rather awkward, because when implementing the sequential prototype for a parallel problem a lot of design decisions are taken, so the developer looses a lot of freedom.

One solution to this problem is to use hardware which can be programmed in a language more close to the prototyping language. In order to research this possibility, a massively parallel hardware architecture, called the Linedancer, was introduced.

Although the Linedancer is a parallel architecture and thus lies conceptually close to Océ’s typical problems and algorithms, there still is a semantic gap between the prototype and the final implementation. To overcome this problem, a design methodology is researched, called Evolutionary Design.

Our research is focused on understanding the mapping of a parallel image processing algorithm on the Linedancer, with the help of the Evolutionary Design methodology. We needed an application and an algorithm as a test case for this mapping process, which would ideally also be of interest to Océ. Finally we decided to use image quantization as our problem to solve on the hardware, using an algorithm based on Simulated Annealing and using a Markov Random Field image model.

1.1 Research Question

From the previous we formulate the following research question:

How to map a quantization algorithm, based on Simulated Annealing and using a Markov Random Field image model, onto the Linedancer.

We make this research question more concrete with the following subquestions:
\begin{itemize}
\item Which problems do we face in mapping the algorithm and how do we face them?
\item How does the Evolutionary Design methodology help us during development?
\item How does the Linedancer compare with a General Purpose Processor?
\item How does the quality of our quantization algorithm compare with established algorithms?
\end{itemize}

During our research our main focus is on the mapping, the image quality produced by the algorithm is of less importance.

1.2 Overview

Here follows a short summary of the structure of the following chapters.

In Chapter 2 we introduce the theoretical concepts that lie at the base of our quantization algorithm. We give a mathematical description of our image model and show the algorithm which is based on Simulated Annealing.

In Chapter 3 we give a short summary of current design practices based on the situations at Océ and the \textsc{CADTES} research group. We evaluate their benefits and negative sides and introduce the design methodology we used during the implementation of the quantization algorithm.

In Chapter 4 we introduce the hardware we used to implement the quantization algorithm: the Aspex Linedancer massively parallel processor array. The special characteristics of this architecture, such as its associative memory are highlighted.

In Chapter 5 we show how the theories, ideas and technology introduced in the previous chapters are used to implement the quantization algorithm. The methodology is used as the guiding principle through the design space.

In Chapter 6 we evaluate the implementation of the quantization algorithm and the impact of the Evolutionary Design methodology on the design process.

In Chapter 7 we summarize the results of our research and give pointers for possible follow up research.
Chapter 2

Quantization

Images scanned in an office environment, such as presentation sheets and charts, often contain a limited number of colors (Figure 2.1). During scanning these get distorted. One of the possible distortions is blurring, with the effect that more colors are used in a scan than necessary (Figure 2.2). Reducing the number of colors in such scans can be useful as a first step in image compression. This process is called color quantization. Popular quantization algorithms include the median cut and octree algorithms [12]. These algorithms use a statistical approach: they count the occurrences of each color and try to assign quantized colors using only this information.

We have used the problem of color quantization as a test case for mapping an image processing algorithm on the Linedancer and developing our design methodology. However, to reduce complexity, we quantized grayscale images instead of colored ones. Our algorithm tries to incorporate pixel location information, in contrast with the aforementioned algorithms.

Figure 2.1: Typical office scan: text and charts.

[Typical office scan]
CHAPTER 2. QUANTIZATION

2.1 Image Model

In the following we assume that the original input image consists of pixels, each having a gray value of between 0 and 255 and that we want to quantize this image to \( N \) different colors. This can be interpreted as the assignment of image pixels to \( N \) different classes.

2.1.1 Fidelity

Assume that the pixel values are continuous and normally distributed. This is a common assumption in image processing [7]. The normal distribution has the following form (\( \mu \) and \( \sigma \) are the mean and standard deviation of the distribution respectively):

\[
p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left( -\frac{(x - \mu)^2}{2\sigma^2} \right) dx.
\]  

If we know the image pixel value means and standard deviations for each class we can calculate the probability of a pixel \( s \) with value \( y_s \in [a, b] \) belonging to a class \( g_s \) using

\[
P(a \leq y_s < b|g_s) = \int_a^b \frac{1}{\sqrt{2\pi}\sigma_{g_s}} \exp \left( -\frac{(y_s - \mu_{g_s})^2}{2\sigma_{g_s}^2} \right) dy_s,
\]  

where \( \mu_{g_s} \) and \( \sigma_{g_s} \) are the mean and standard deviation of pixel values in class \( g_s \) respectively (Figure 2.3). Because pixel values are actually discrete, we define the probability of a certain pixel value \( y_s \) in the discrete case as the weight of the distribution in a neighborhood of width 1 around \( y_s \) in the continuous case.

In other words, we define it as probability of the pixel having a value between \( y_s - \frac{1}{2} \) and \( y_s + \frac{1}{2} \). From Figure 2.4 it can be seen that we can approximate the discretized probability with the derivative at \( y_s \), which simply removes the integral from (2.2):

\[
P(y_s|g_s) = \frac{1}{\sqrt{2\pi}\sigma_{g_s}} \exp \left( -\frac{(y_s - \mu_{g_s})^2}{2\sigma_{g_s}^2} \right).
\]
2.1. IMAGE MODEL

Figure 2.3: Probabilities $P(y_s|g_s)$ for the case of quantization to 4 different classes ($N=4$).

Figure 2.4: Approximation of a discrete probability $P(X = a)$ based on the continuous probability distribution $p(x)$. 

$p(a + \frac{1}{2})$

$p(a - \frac{1}{2})$
CHAPTER 2. QUANTIZATION

(a) Irregular image

(b) Regular image

Figure 2.5: Examples of images of low and high regularity.

\[ C = \{ \bullet, \circ \} \]

Figure 2.6: Image pixel (the black dot) with its neighborhood (gray dots) and corresponding set \( C \) of clique pairs. Each clique has its own color.

We assume that the relation between input image and quantized image only affects corresponding pixels, which means that

\[
P(y|g) = \prod_{s \in S} P(y_s|g_s). \tag{2.4}
\]

This probability represents the concept of fidelity, i.e. the property of the quantized image that it resembles the original input image. It gives us a way to compare different quantizations by comparing probabilities. The higher the probability \( P(y|g) \), the better the quantization matches the original input.

2.1.2 Regularity

While fidelity accounts for the quantization quality of a single pixel, another property called the regularity measures the coherence between quantized pixels (Figure 2.5). This is a property of the quantized result, which is assumed as prior information. Because we focus our algorithm on business graphics we assume that the results consist of connected regions of the same color, called blobs.

To incorporate this into our model we make use of a MRF model. In this model, the value of each quantized pixel only depends on its neighborhood. These dependencies are called the local characteristics of the MRF. Sets of pixels that are all neighbors of each other are called cliques and the set of all cliques \( C \) in the entire image is denoted \( C \) (Figure 2.6).

It can be proved that the local characteristics uniquely define the joint probability \( P(g) \), and thus of the total quantized image in which we are interested. The Hammersley-Clifford theorem guarantees that we can write this.
joint probability in the form
\[ P(g) = \frac{1}{Z} \exp \left( \frac{-E(g)}{T} \right), \quad (2.5) \]
where \(E(g)\) encodes the local characteristics. \(T\) is a constant called the temperature after its meaning in statistical physics from which \(2.5\) originates. \(Z\) is a normalization constant, to make sure \(P(g)\) is a real probability.

The energy,
\[ E(g) = \sum_{C \in \mathcal{C}} V_C(g), \quad (2.6) \]
is built up by the clique potentials \(V_C(g)\). These are functions defined for each clique and only depend on the pixels inside this clique. So, it is possible to model the joint probability \(P(g)\) with clique potentials instead of local characteristics. In our case we only use the cliques \(C\) which are pairs \((s, r)\) and all clique potentials are the same:
\[ V_C(g) = V_{(s, r)}(g) = \delta(g_s, g_r). \quad (2.7) \]
The \(\delta(g_s, g_r)\) function returns \(-1\) if \(g_s\) equals \(g_r\), \(+1\) otherwise. So, similar pixels pairs contribute \(-1\) to the energy, the others contribute \(+1\).

Combining (2.5), (2.6) and (2.7) we finally arrive at:
\[ P(g) = \frac{1}{Z} \exp \left( -\frac{1}{T} \sum_{(s, r) \in \mathcal{C}_2} \delta(g_s, g_r) \right). \quad (2.8) \]
Here, \(\mathcal{C}_2\) is the set of all pair cliques.

### 2.1.3 Bayesian Model

To give an idea of the terms fidelity and regularity, Figure 2.7 shows two cases. We can interpret the (normalized) fidelity and regularity as probabilities: let \(P(y|g)\) be the fidelity (i.e. the probability of observing input image \(y\) when \(g\) is the quantized image), and \(P(g)\) be the regularity (i.e. the probability of \(g\) being a satisfying quantized image). Using Bayes law \(15\)
\[ P(g|y) = \frac{P(y|g)P(g)}{P(y)}, \quad (2.9) \]
which can now be interpreted as the probability of \(g\) being the original image when image \(y\) is observed, we obtain a measure of the suitability of an quantization given an input image in terms of the fidelity and regularity. To find the quantized image given our input we have to maximize \(P(y|g)\), the posterior distribution. The evidence, \(P(y)\) is constant for given \(y\), so this term can be ignored for the maximization.

We now explicitly derive the expression that we want to maximize, using (2.4) and (2.8):
\[ P(y|g)P(g) = \prod_{s \in S} P(y_s|g_s) \frac{1}{Z} \exp \left( -\frac{1}{T} \sum_{(s, r) \in \mathcal{C}_2} \delta(g_s, g_r) \right) \]
\[ = \frac{1}{Z} \exp \left\{ \sum_{s \in S} \left( -\ln \sqrt{2\pi \sigma_{g_s}} - \frac{(y_s - \mu_{g_s})^2}{2\sigma_{g_s}^2} \right) - \frac{1}{T} \sum_{(s, r) \in \mathcal{C}_2} \delta(g_s, g_r) \right\}. \quad (2.10) \]
CHAPTER 2. QUANTIZATION

(a) Input image.

(b) High fidelity, low regularity: result looks like the original, but does not have connected regions.

(c) High regularity, low fidelity: result has connected regions but looks less like the original.

Figure 2.7: Fidelity and regularity examples.
Because $Z^{-1}$ is constant and logarithms are strictly increasing, we can take the logarithm of (2.10) and optimize that instead. We also use $\beta = \tilde{T}^{-1}$ to denote the inverse temperature from here on. Finally we can take out a negative sign and minimize the following:

$$E(g) = \sum_{s \in S} \left( \ln \sqrt{2\pi \sigma_{g_s}} + \frac{(y_s - \mu_{g_s})^2}{2\sigma_{g_s}^2} \right) + \beta \sum_{c \in C} \delta(g_s, g_c).$$

From this we can see that $\beta$ determines the relative weight between fidelity and regularity. We can therefore control their relative importance using $\beta$ as a parameter.

### 2.2 Simulated Annealing

In order to find the quantized image minimizing the energy $E(g)$ we make use of Simulated Annealing (Algorithm 1) as described in [5]. We assume that the algorithm parameters such as $\beta$ and $\mu_{g_s}$ are known in advance (more on this later). The algorithm searches a state minimizing an energy function such as

```
x ← Random state
for $T ← T_0, T_0 \cdot C, \ldots, T_0 \cdot C^{A-1}$ do
    $\hat{x} ←$ Slightly changed state $x$
    $\Delta E ← E(\hat{x}) - E(x)$
    if $\exp \left\{ -\frac{\Delta E}{T} \right\} ≥ 1$ then \{Deterministic acceptance\}
        $x ← \hat{x}$
    else if $\exp \left\{ -\frac{\Delta E}{T} \right\} ≥ \text{Rand}[0, 1]$ then \{Probabilistic acceptance\}
        $x ← \hat{x}$
    end if
end for
```

Algorithm 1: Simulated annealing

our $E(g)$. First, it is initialized with a random state. Then, the state is changed a little (e.g. in the case of quantization by changing the classification of a single pixel) and accepted if this change decreases the state energy. If the energy does not decrease, the state is accepted with probability equal to the Boltzmann factor $e^{-\Delta E/T}$. (2.12)

A parameter $T$ (not to be confused with $\tilde{T}$ from (2.8)), called the temperature because of its foundation in physics, determines to what extent energy increases are allowed. During the running of the algorithm the temperature $T$ is decreased. The behavior of the acceptance probability for various temperatures can be seen in Figure 2.8. When the temperature is high, the probability function is mostly flat and (almost) all proposed states are accepted, which results in the visiting of random states and the algorithm behaves probabilistically. At lower temperatures the algorithm only allows transitions lowering the energy. This makes the algorithm behave deterministic. The temperature is decreased during the algorithm in order to converge to a final solution.
CHAPTER 2. QUANTIZATION

Figure 2.8: Acceptance probability at different temperatures.

The starting temperature $T_0$ determines how randomly different states are visited. It should be high enough to allow all possible states be visited at the start of the algorithm, but if chosen too high, the algorithm needs too many steps to settle down [5]. The cooling factor $C \in (0,1]$ determines the rate at which the temperature decreases. If the temperature is dropped too fast, the algorithm can get trapped in local minima.

2.3 Modified Metropolis Dynamics

To optimize Algorithm 1 for massively parallel systems we chose to use Modified Metropolis Dynamics (MMD) as given in [14]. There, the comparison with a random value from the range $[0,1)$ is substituted with a comparison with a predetermined global threshold $\alpha$ in the range $(0,1)$. This means that we can replace the exponentiation with a precalculated logarithm. Another change is that the calculation of the energy is performed local to each pixel, i.e. each pixel has its own energy $E_s = E(g_s)$ which is optimized. This results in an algorithm in which all pixels can be processed in parallel.

These changes result in Algorithm 2. The energy function now becomes ($C_{2s}$ are the pair cliques pixel $s$ belongs to)

$$E(g_s) = \left[ \ln(\sqrt{2\pi\sigma_{g_s}}) + \frac{(y_s - \mu_{g_s})^2}{2\sigma_{g_s}^2} \right] + \sum_{(s,t) \in C_{2s}} \beta \delta(g_s,g_t).$$  \hspace{1cm} (2.13)

The global constant threshold $\alpha$ has the same function as the uniform ran-
2.4. CHOOSING PARAMETERS

When we want to really implement this algorithm, we should choose values for its parameters. For the mean values this can be done by calculating them for a set of reference images, but for the parameters like $\alpha$ and $\beta$ the choice should be more ad hoc. For example, it is reasonable to expect that an $\alpha$ value near 0 gives better results for noisy input images, because this makes the algorithm behave more probabilistically.
Chapter 3

Evolutionary Design

In this chapter we describe the Evolutionary Design methodology, used in this project for the implementation of a parallel quantization algorithm.

3.1 Current Practice

In this section we describe development methodologies for parallel systems which are currently in use at the company Océ and the CADTES research group at the University of Twente.

3.1.1 Océ

Currently, functionality of parallel image processing systems is designed in C using a common framework with multiple flavors. Each employee has the freedom to some degree to use his own methods of supporting his or her development. In the case of image processing, some people choose to model their algorithm in Matlab. When they have finished this implementation and have a working model, they implement the algorithm from scratch in Very-High-Speed Integrated Circuit Hardware Description Language (VHDL) or some other target language.

The benefit of using C is that it is fast, so it is often possible to approach the speeds of the hardware implementation, where full sized images can be processed within acceptable time. However, because C is sequential, it does not map well to parallel architectures, such as Field Programmable Gate Arrays (FPGA).

The benefit of using Matlab, is that it increases development speed, because of its higher level constructs. Matlab programs are also more parallel in nature, which makes them easier to map to parallel hardware. The downside is that a Matlab implementation is not as fast as one in C and that tests therefore have to restrict themselves to downscaled versions of the problem.

Both the C and Matlab approaches lack a link between the specification of the system and the final implementation.

3.1.2 CADTES Group

At the Computer Architecture Design & Test for Embedded Systems (CADTES) group, algorithms contained in multiple parallel processes, are specified using
3.2. ALTERNATIVES

A mixture of Matlab and C code. The Matlab parts are then transformed into C code, after which a tool is used to separate parallel and sequential parts and map these to different parts of the hardware.

A recent development is the use of the Matlab tool Simulink to divide an algorithm into blocks of separate functionality. These blocks can each be implemented in a different language. Performance critical parts can be implemented in C, while less critical parts can be written in high level Matlab code. It is even possible to implement blocks using VHDL so these parts can actually run on hardware. Using this method, different parts of the system can be described at different levels of abstraction, whatever is more convenient. This also makes it easier to test a specification against the implementation in hardware.

This methodology creates a bridge between the specification of the algorithm and the implementation and can be used to model parallel systems. However, it still makes use of sequential parts (C and to a lesser degree, Matlab).

3.2 Alternatives

This section introduces two methodologies which address some of the issues mentioned in Section 3.1.

3.2.1 Blaauw and Brooks

Blaauw and Brooks distinguish three aspects of the design of a system.

- The architecture defines a minimal behavioral specification of a system.
- The implementation defines the inner workings of the system.
- The realization defines specifics of how a system is actually built.

According to Blaauw and Brooks in [4] using natural languages such as English for the specification of a computer architecture is sensitive to ambiguities. To counter this, specifications can be written in a formal language, such as Z [13]. Programming languages can be used as an executable formal language for the specification of computer architectures.

Following Blaauw and Brooks, such a language should be:

- **High level** so each desired function can be expressed directly, without having to bother with things like memory allocations and implementation of basic data structures. It also avoids suggesting an implementation, which otherwise would restrict the developer.

- **Executable** to permit demonstration to the user early in development, so the user can actively influence the functionality of the architecture. This is important, because the architecture cannot be proved or verified to be correct. It is desirable for the language to be interactive, to keep the edit-compile-run cycle short and to make it possible to easily determine the state of the system.

- **General purpose** so other tooling, such as performance measurement tools can be implemented in the same environment.
Established so that it is well tested and debugged and code can be shared with the community of language users.

Structured so it encourages a clear design structure with subordinate functions, which helps top down design.

The language Blaauw and Brooks use for their specifications is A Programming Language (APL).

Because of the use of APL, Blaauw and Brooks can develop their specification faster than they would have in C. APL being an array language, also makes it possible to model parallel algorithms.

One downside of using such a high level language is that these languages are less efficient than the target hardware language, especially when they are interpreted. This means that the problem often has to be downscaled to run the specification at a reasonable speed. Another downside of this methodology is that it does not specify how to transform the specification into an implementation on the hardware.

Conceptual Integrity

One method of specifying a system is to divide the system into parts and give the responsibility for each part to a different person. When the specifications of the different parts are ready, it may well be that the specifications are incompatible, because there was no supervision on the whole system.

Blaauw and Brooks claim that it is better to have a single person be responsible for the design of the entire system, which should keep the specification of the system consistent. This single person should then delegate the implementation of parts of the system to his subordinates.

The use of a language like APL assists this idea, because its concise and powerful notation can simplify specification and thus make it possible for a single person to grasp the design of larger systems.

3.2.2 Bernecky

In [3] Bernecky splits the design of an algorithm on hardware into three phases, each producing a different model, followed by a transcription phase. These phases are:

1. the concise model in which the problem and solution are researched and understood. This results in a functionally complete prototype, implementing the architecture.

2. the intermediate model introduces the data structures and algorithms that are to be used. This results in a prototype specifying the implementation of the system.

3. the detailed model is a reflection of the implementation as implemented in the target language.

When the detailed model is obtained, it is translated into the target hardware language in the transcription phase.

Bernecky also uses the APL language, resulting in a high speed of development and parallel capabilities.
3.3 Evolutionary Design

Our design methodology also makes use of a high level language, just like the methods of Blaauw and Brooks, and Bernecky. Evolutionary design globally consists of two phases (Figure 3.1). During the first phase, Incremental Prototyping, the problem is researched until a high level, functionally complete implementation exists. After that, during the Transformational Development phase, in a series of transformation steps, the code is refactored into a form resembling the structure of the target language. The last step in this procedure is a translation from the high level language into the final implementation language. This translation can be automated by implementing a (simple) compiler.

In order to be able to do these things, the high level language that is used has to have some extra qualities with respect to those given by Blaauw and Brooks. The language should additionally be:

- able to support programming on different abstraction levels. This makes it possible to apply Transformational Development.
- conceptually close to the target language (e.g. use an array language to develop for a parallel processor array). This reduces the effort needed for Transformational Development.

3.3.1 Incremental Prototyping

The goal of this phase is to learn about the various aspects of the development: functionality of the system, the target architecture and the description language.

The functionality of the system is built at a high abstraction level, one function at a time. The only guidance in this phase of development are the requirements of the user, so communication with the user is important. Because we use an executable description language, we can directly show the behavior of the system, without ambiguities.

If the target hardware architecture is not yet completely understood, it may be beneficial to conduct tech probes: small tests to gain insight into the workings of the architecture. Though not clearly being part of this phase, this knowl-
edge is needed in later phases to make useful design decisions, and therefore is included here.

If the programmer is not yet comfortable with the description language, this phase presents a good opportunity to get familiar with it. The functional prototyping presents an interesting learning vehicle.

After the prototype is functionally complete, the result of this phase is a first simple, but functionally complete, implementation of the system. This methodology also makes it possible to transform the specification of the algorithm into the final implementation on the hardware.

3.3.2 Transformational Development

Transformational development is the journey from a high to a low abstraction level. During this journey we step through a couple of design subphases (Figure 3.2). Each subphase results in a specific end result. Although the main development momentum is directed top-down, there is also a small bottom-up step: the template phase.

An overall overview of how Transformational Development fits into the Evolutionary Design methodology can be seen in Figure 3.3.
Trade-Off Subphase

The goal of this subphase is to change the functional description from the Incremental Prototyping phase into a description generating the same output as the hardware implementation. This means that all descriptions after this subphase have exactly the same behavior and thus can be tested by comparing their output. So, this phase is used to remove all remaining uncertainties from the description.

The activities during this phase consist of choosing specific algorithms and data types and investigate possible trade-offs, e.g. adapt bit precision to the target hardware. The influences of these activities can be measured by comparing the results with those of the Incremental Prototyping phase, which acts as a ‘perfect’ reference. The extent of the influences of the trade-offs have to be determined together with the user of the system.

The main risk in this phase is making wrong decisions because of lack of knowledge of the architecture or the problem that has to be solved. This risk can be reduced by performing tech-probes during the Incremental Prototyping phase and to perform this subphase together with the future user of the system.

At the end of this subphase there should be a description of the system for which the behavior is definitive: only the implementation can change. This is where the influence of the problem domain (and the system user) stops and from which the development is guided by the target architecture.

Reorganization Subphase

The goal of this subphase is to transform the system description into a form where all operations are performed in the same way as on the target hardware. If we interpret the Trade-Off Subphase as describing the What? of the system, the Reorganization Subphase describes the How?. Possible transformations in this subphase are changing parallel instructions as explicit loops and breaking up complex expressions.

Because in this subphase the behavior of the description is already fixed, all descriptions can be verified against each other. This makes it easy to find mistakes made during the transformation.

The result of this phase is a description of the system which mimics the final implementation both in behavior as in structure.

Template Subphase

The goal of this subphase is to transform the description of the Reorganization Subphase into a form which is easy to parse and translate into the target hardware language.

To accomplish this, blocks of code with a single identifiable function are gathered into code templates. This can be interpreted as a (small) bottom-up step in the development methodology. The templates can be developed in parallel with the previous subphases. Code templates are most often very simple and map directly to instructions in the target language, but the abstraction level of the templates does not have to be the same as the target language. For example searching a value and replacing it can be built as one code template, while the lower level implementation consists of multiple statements. This is
helpful if there are a lot of similar constructs that can be abstracted into one code template. Code templates are the link between the system description and the target hardware language. Therefore, the templates are a perfect location to check for additional requirements of the hardware, such as calling conventions. The result of this phase is a system description which can be translated into the target hardware language by a simple translator. This translator has to do little more than expanding code templates into the target hardware instructions.

Translation Subphase

The goal of this subphase is to translate the system description from the Template Subphase directly into the target hardware language. This can be done automatically by using a (simple) compiler, which is written together with the code templates. There are two possible sources of errors in this subphase: the parser implementation of the translator and the implementation of code templates into the target hardware language. The possibility of a faulty parser can be reduced by using a specialized tool for writing compilers and translators. A typing error in a code template can often be corrected, but if the error is caused by wrong assumptions about the hardware it may be needed to fall back to the Template Subphase or an even earlier phase. The result of this phase is a working implementation of the described system with identical behavior as established at the end of the Trade-O Subphase.

3.4 Intended Benefits

Here we give some of the possible benefits of Evolutionary Design.

3.4.1 Higher Productivity

Our main objective with Evolutionary Design is to make development easier and more productive. The use of an executable language gives early feedback during development, which makes development more dynamic. This makes it easier for users to make sure that they get the product they want and helps developers to verify and debug their implementations.

Using Transformational Development, we make sure that each development step is consistent and correct with respect to the previous, higher level state of the design.

3.4.2 Error Recovery

Sometimes assumptions made during the Incremental Prototyping phase and Trade-Off subphase prove to be false during one of the later subphases. This is possible, because there is no definitive notion of the correct behavior yet. However, the equivalence between all reorganization steps can help. When the error is corrected at a high level it is often not necessary to completely redo all the transformations for each subphase. Most of the time the fix for each subphase can be implemented directly into the existing code of the subphase.
3.4. **INTENDED BENEFITS**

3.4.3 **Portability**

The Template subphase can be used to identify common abstractions between different hardware implementations. This way, common code templates can be developed for different architectures. When this is achieved, development can focus on implementing towards these templates (of which there are few) instead of towards each separate piece of hardware. This way, existing algorithms can be implemented on new hardware after an implementation for the common code templates has been developed for it. Conversely, new algorithms written in terms of the common code templates can run on all machines supporting these templates (see Figure 3.4).

3.4.4 **Domain Expert Programming**

One of the possible benefits of Evolutionary Design is that all domain specific problems are solved in the Incremental Programming phase. If the higher level language is simple enough to be understood by domain experts, they can perform the Incremental Prototyping phase themselves. When the Incremental Prototyping phase is over, system engineers can take over and start at the Trade-Off subphase, with feedback from the domain experts.

After the Trade-Off subphase, the system engineers can continue on their own, because the following steps are independent from the problem domain. This way, there is less of a discontinuity between the specification from the domain professionals and the implementation of the system engineer during hardware software codesign.

Figure 3.4: Increasing portability by using common code templates. Algorithms are implemented in terms of these code templates. For each code template there is an implementation on each architecture.
Chapter 4

Linedancer

In this chapter we introduce the hardware architecture used in our research: the Linedancer, which is manufactured by Aspex, a UK based fabless semiconductor company specializing in high performance, software programmable, parallel processors based on associative technology. We give a short introduction on parallel architectures, followed by the specific properties of the Linedancer itself. The chapter concludes with performance measures of some of the instructions of the hardware.

4.1 Parallel Architectures

In this section we give a short introduction into parallel architectures.

4.1.1 Processor Architecture Taxonomy

In 1966 Flynn divided processor architectures in terms of their data and instruction multiplicity. This has lead to Flynn’s Taxonomy of four different types of computer architectures (see Table 4.1.1).

The Single Instruction Single Data (SISD) architecture is the one typically found in modern day PCs: each instruction acts on a single data unit.

The Single Instruction Multiple Data (SIMD) architecture has also made its entrance into consumer computers when Intel introduced their MMX Pentium models, later followed by AMD’s 3DNow!. These technologies provide instructions, which enables one to apply a certain instruction to multiple data units at once. E.g. vector additions could be performed using only one instruction, instead of doing the addition for each vector element explicitly.

The Multiple Instruction Single Data (MISD) is more or less a theoretical architecture because its use has been very limited. Some would place systolic

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td>SIMD</td>
<td>MISD</td>
</tr>
</tbody>
</table>

Table 4.1: Processor Architecture Taxonomy according to Flynn.
arrays into this category.

The Multiple Instruction Multiple Data (MIMD) architecture has been around for a long time. Examples can be found in specialized high performance parallel computers such as developed by the Cray supercomputer company. During the last decade MIMD also made its entrance into the PC world as multi processor systems became more commonplace. Another example of MIMD are rendering farms, which are used for rendering of computer animations and simulations. These consist of tens to hundreds of PCs and communicate through a high speed network. Finally, the trend of distributing the execution of brute force algorithms via the Internet can be seen as a form of MIMD. In this scheme, private persons are asked to donate the idle time of their computers to help in some calculation. Examples of this kind of MIMD are the SETI[2] and RC5–72[1] projects.

4.1.2 Benefits of Parallel Architectures

At first glance the use of parallel architectures only has benefits. If more processors work at a certain task, then the task is completed sooner than when only one processor is used.

While this is true (you could always use only one processor and finish the work in the same time) the benefit is not linear: doubling the number of processors generally does not half the processing time.

To understand this, we define the term speedup. If $T_{\text{seq}}$ is the time taken by the most efficient sequential algorithm to perform a task and $T_{\text{par}}(n)$ is the time taken by the most efficient parallel algorithm to perform the same task when running on $n$ processors, then the speedup $S_n$ is the ratio between these times, i.e. $S_n = T_{\text{seq}}/T_{\text{par}}(n)$. So, if a parallel algorithm running on 8 processors performs a task twice as fast as a sequential algorithm then the speedup is 2.

In practice, most parallel algorithms also have a sequential component, e.g. for managing the communication between the processors. If we call the fraction of sequential operations in an algorithm $f$, then we can derive Amdahl’s law:

$$S_n \leq \frac{1}{f + (1 - f)/n}.$$ (4.1)

When we take the limit $n \to \infty$ we see that this reduces to $S_n \leq \frac{1}{f}$. So, if our algorithm consists for 5 percent of sequential algorithm we can hope to achieve a speedup of 20 or less. This tells us that just adding more processors does not help us much and that we will have to carefully evaluate the added value of multiprocessing for each problem.

Note, however, that the most efficient sequential algorithm does not have to have the same model as the most efficient parallel algorithm. So, when developing an algorithm, care should be taken not to simply try and implement a known sequential algorithm on a parallel architecture. The most efficient parallel algorithm could be completely different. Amdahl’s law should therefore be kept in mind, but it should not discourage the developer in finding new and more efficient models for his problems.

[2]a project to find extraterrestrial intelligence by analyzing data from radio telescopes.
[3]a project to crack a 72-bit cryptology key.
4.2 Hardware Architecture

The Aspex Linedancer parallel processor array is a member of the SIMD family of processor architectures. The Linedancer is used for image processing tasks, e.g. color correction [6]. Our current system, the Linedancer development board, is mounted on a PCI card and contains 2 Linedancer chips. Both of these chips contain an array of 4096 PEs, each with its own memory of almost 200 bits. Control of the array is maintained by the processor of the machine the Linedancer is installed in and an onboard RISC processor. These are referred to as the System Control Computer (SCC) and the Instruction and Data Stream Manager (IDSM) respectively.

4.2.1 Scalability

While in most SIMD architectures the PEs are arranged in meshes, the PEs in the Linedancer are arranged in a string (Figure 4.1). So, in order to increase the number of PEs, multiple Linedancers can be connected serially, while a mesh would require adding a lot more PEs to maintain its structure. This improves the scalability of the array and reduces the complexity of the chip’s design. However, it also increases the communication time between PEs, because there are less communication channels.

4.2.2 Memory Transfers

Memory transfers from, to and inside the Linedancer can be divided over a memory hierarchy, consisting of Tertiary Data Store (TDS), Secondary Data Store (SDS), Primary Data Store (PDS) and the PE memory (Figure 4.2). TDS is the main memory of the system that contains the Linedancer (e.g. a PC). Typically this memory is only used to store the input and output of the entire program.

Each Linedancer chip has SDS memory in the form of DRAM memory modules. This memory is used for holding the data to be processed. Data can be loaded from and stored to the TDS under control of the SCC. If a lot of data is to be processed, the SDS memory can be expanded with special expansion boards. Most programs load all input data from TDS to SDS at the beginning of the program and retrieve the output from SDS to TDS at the end.
Figure 4.2: Transfers inside the memory hierarchy of the Linedancer
For each PE there is a 64 bits PDS memory. This acts as a buffer between the SDS memory and the PE’s data registers, enabling streaming by double buffering. Transfer between SDS and PDS is controlled by the Secondary Data Movement Controller (SDMC), a DMA controller that can be programmed to do all kinds of transfers and divides the SDS data between the PEs, one item at the time. E.g. the SDMC makes it possible to process an image either row- or column-wise or replicate data across multiple PEs.

4.2.3 Associativity

PE memory is divided in two parts, Extended Memory (EXT) and Content Addressable Memory (CAM). The CAM part can be addressed based on its contents instead of its physical address. Because of this, searching data in the PE is very easy. It also makes the addressing independent of the array’s string size, so your code does not have to be altered when the number of Linedancers is expanded for increased performance.

The associative memory relies on the use of a number of tag registers and one activation register for each PE. The tag register of a PE can be set based on the contents of its CAM using the Tag instruction. This is illustrated by an example.

In Figure 4.3(a) the tag register (T) is set for PE with a specific part of their memory (M) set to 010. The activation register determines which PEs can be written to. The Activate instruction can set the activation register on all PEs or based on one of the tag registers. In Figure 4.3(b) the activation register (A) is set for for those PE which did not match the pattern 010, i.e. those which do not have their tag register set and in Figure 4.3(c) the pattern 10 is written to the activated PEs.

4.2.4 Communication

Data inside the registers can be freely moved between the PEs. The number of cycles this takes depends on the distance between the PEs involved.

The IDSM can directly write scalar data to activated PE. This is done by the Write instruction.

It is possible to transfer the contents of the tag registers through the array, one PE at the time. This can be used to access PEs based on the contents of their neighbors.

4.2.5 Segmentation

By default the array behaves as a string with its extreme ends connected. The PE can be split up into multiple segments. These segments each operate independently as if they were physically distinct Linedancers, without the possibility of communication between them. This way, multiple Linedancers can be emulated on a single physical device. Segments are defined by the Segment instruction in a number of different ways [11].
4.2. HARDWARE ARCHITECTURE

(a) The tag register (T) is set for matching PEs.

(b) The activation register (A) is set for PEs where tag register (T) is not set.

(c) Data is written to PEs with set activation registers.

Figure 4.3: Associative memory in action.
### Arithmetical and Logical Operations

The only arithmetical operations that a PE can perform on the data inside its memory are addition, And, Or and Xor. These can all be performed in normal and complement mode. When in complement mode, one of the operands is complemented before the operation. Non-basic operations, such as division and subtraction are implemented in terms of the four basic operations. Because the ALU is only 2 bits wide, calculations per PE are generally slower than those done by a PC because they have to be emulated and performed serially. This emulation is done by so called RTS macros. An advantage is that PE memory can be accessed on bit level: i.e. it is possible to access and operate on 3-bit quantities inside the memory.

### Programmer’s Interface

The programmer’s interface consists of the software tools used to compile a program and a number of supporting libraries.

#### Software Tools

The programming language is C, extended with aop blocks, which contain Linedancer instructions. If the SCC and DSM are different processors, compiling results in two separate binaries; one for each processor. The SCC program is the main entry point: it initializes the Linedancer and starts the separate DSM processors.

#### Threads of Execution

The DSM process has two separate threads: the DSM and ISM threads. The DSM manages data transfer between SDS and PDS while the ISM issues the instructions for the array, including data transfer between PDS and the PE registers. Because the PDS is a shared resource of both threads, care has to be taken that access to the PDS is synchronized.
4.4 Performance

Here we present the results from testing the performance of some of the Linedancer operations that we used.

4.4.1 Constant Time Instructions

The instructions used for associative operations take constant time (Table 4.3). However, multiple occurrences of these instructions can be optimized into one occurrence.

4.4.2 Linear Time Instructions

Basic arithmetic and logical operations depend linearly on the bit widths of their operands. After some tests we were able to calculate the exact dependency (Table 4.4, $w$ is the width of the result and operands). For most operations with a cycle count dependent on the bit widths of the operands, the number of cycles is smaller if the operand width is even than when it is odd. This is because of the 2-bits ALU, which is only used if the operand widths are even.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>4</td>
</tr>
<tr>
<td>Segmentation</td>
<td>2</td>
</tr>
<tr>
<td>Write</td>
<td>2</td>
</tr>
<tr>
<td>Activate</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3: Linedancer operations running with a constant number of cycles.

4.3.3 Libraries

Aspex has written some higher level libraries on top of the basic Linedancer instructions, which can make programming the Linedancer a bit simpler.

Image Library

This library takes care of splitting image data loaded in the SDS into tiles of smaller image parts. These tiles can then be uploaded to and from the PDS. It also takes care of configuring the SDMC parameters, which can be difficult to understand.

Message Queue Library

This library makes it possible to create different message queues between processors. This can be used to transfer configuration information (e.g. from the command line) between the SCC and the DSM or to return status information from the DSM to the SCC. Because the queue transfer operations can be made to block they are also an excellent tool to synchronize the different threads.

4.4 Performance

Here we present the results from testing the performance of some of the Linedancer operations that we used.

4.4.1 Constant Time Instructions

The instructions used for associative operations take constant time (Table 4.3). However, multiple occurrences of these instructions can be optimized into one occurrence.

4.4.2 Linear Time Instructions

Basic arithmetic and logical operations depend linearly on the bit widths of their operands. After some tests we were able to calculate the exact dependency (Table 4.4, $w$ is the width of the result and operands). For most operations with a cycle count dependent on the bit widths of the operands, the number of cycles is smaller if the operand width is even than when it is odd. This is because of the 2-bits ALU, which is only used if the operand widths are even.

4 The number of cycles are for Load and Dump together.
Table 4.4: Linedancer operations which are linearly dependent on the bit width.

Because the test setup for the pdTransfer instruction demanded that both a Load and a Dump were done, we only have results from both calls together. The number of cycles taken by only one of these calls is probably one half of the number in Table 4.4 but this has not been verified.

4.4.3 Slow Instructions

In Figure 4.4 the relation between bit width and the running time in cycles is shown for the RTS(Mult) and RTS(Div) instructions. These instructions are slow: the multiplication of two eight bits numbers takes 114 cycles, while the addition of those numbers only takes 10 cycles. RTS(Mult) and RTS(Div) also have worse than linear behavior with respect to the number of bits. RTS(Div) is dependent on whether it is applied to integers (Int) or cardinals (Card). Based on these measurements, RTS(Mult) and RTS(Div) should therefore be avoided whenever possible.

In Figure 4.5 the relation between PE distance and running time in cycles is shown for the RTS(Assign) instruction. While testing, the bit width of the operands was kept constant. When we look at the number of cycles when transferring data at a distance of 64 (4226) or 128 (16642), we see that this is already a lot higher than the number used to divide two 16 bits numbers. However, this distance is exactly the distance used when emulating a mesh layout with the Linedancer, which is often the case in image processing.
4.4. PERFORMANCE

Figure 4.4: Dependency of the number of cycles on the operand width for the RTS(Mult) and RTS(Div) instructions.

Figure 4.5: Dependency of the number of cycles on the operand width for the remote RTS(Assign) instruction.
Chapter 5

Implementation

In this chapter we introduce the decisions we made while implementing the quantization algorithm from Chapter 2 on the Linedancer (Chapter 4). This implementation is supported by the Evolutionary Design methodology introduced in Chapter 3.

When the algorithm is run, the parameters are first chosen to likely correspond with the actual data. Next the quantized image is found by minimizing the energy function using Simulated Annealing.

Our methodology applied on the Linedancer has only been used to write the parallel part of the implementation, the ISM thread. This was done, because this is the most difficult part of the hardware to program, comparable with assembler code. Also, the code for the other threads is very similar for different problems, so these are a lot less sensitive to change than the ISM thread.

5.1 Preparation

We chose the J programming language as the high level language to apply our methodology with. For this we had three reasons:

1. it was already used in my working environment;
2. it was a totally different programming language than that I had been used to using, so I was interested in its use;
3. it fulfilled the requirements mentioned in Chapter 3.

J was developed as a derivative of the APL programming language, which was popularized by Kenneth Iverson, first as a mathematical notation, later as a programming language while at IBM.

J has powerful primitives and has a very mathematical inclination, which helps when specifying complex algorithms. J is a language with arrays as primary data type, which results in a parallel approach of the manipulation of data. Allocation and deallocation happen automatically and all data accesses are by value, so there is no use of pointers. These qualities make J easier to use and less error prone than languages such as C. Compared to e.g. Matlab, J is very concise in notation and is very light weight: the installation is only 20
5.2. INCREMENTAL PROTOTYPING

MB. Functions are first citizens in the J language, but it is not a strict functional language, such as Haskell or Miranda, because it allows for side effects, has variables and control flow constructs such as If, For, etc.

The makers of J have optimized it quite a bit; e.g. some combinations of keywords are recognized and have an optimized implementation.

This chapter contains snippets of J code to give a general idea of what kind of transformations are done. It does not matter if the code is not entirely understood. In Appendix A you can find a short introduction to J, which should help understand the J code used here.

5.2 Incremental Prototyping

Before the implementation of the quantization algorithm we tried different implementations of Simulated Annealing programs found in literature to learn about the theory behind it. Because I wrote these test programs in J, this was also a way to get familiar with the programming language. During my internship I had already worked with the Linedancer, so I had a general idea how the hardware architecture behaves.

The first implementation in J can be found in Section 5.1.1. The annealing procedure is located in the lines 1–22. This implementation serves as the root of our development efforts.

5.3 Transformational Development

We now give an overview of how the different subphases of Transformational Development were passed through during the implementation of the quantization algorithm. This is done using concrete examples from the implementations in J. For an introduction to J primitives, we refer to [9]. Comments in J start with NB.

5.3.1 Trade-Off Subphase

The goal of this subphase is to change the functional description from the Incremental Prototyping phase into a description generating the same output as the hardware implementation. We quantized some test images using the program Irfanview and used these results as a reference. We chose a threshold of 5% misqualified pixels to represent a ‘good enough’ quantization. This was used to compare different trade-offs in this subphase.

Tiling

Our implementation uses one PE for each pixel in the input image, because this is the massively parallel part of the algorithm. The number of PEs in the Linedancer is limited (although it can be expanded), so the number of pixels that can be processed together is also limited.

To make it possible to process larger images, we used tiling, i.e. we cut the image in small chunks that do fit on the Linedancer (see Figure 5.1) and optimize each tile once. We chose to do it this way because:
Figure 5.1: Tiling: the input image is cut into blocks fitting inside the Linedancer and processed in sequence by the Simulated Annealing algorithm.

- It is a simple method, so it would not complicate the algorithm any further.
- Some tests on an early J model gave the impression that non-overlapping tiling would not have a large influence on the algorithm.

We chose to minimize the number of SDMC transfers, and not communicate between tiles in order to maximize the number of parallel calculations within a tile. This was done to maximize calculations, which the Linedancer is good at, and minimize communication overhead.

Note that this way of tiling increases the locality of the algorithm, because there is no influence between data in different tiles.

**Round Off**

Although hinted on in the documentation, the Linedancer does not support floating point numbers in the PEs. This was confirmed by someone from Aspex. Therefore, we would have to round off values or use fixed point arithmetic if more than integer precision would be needed.

The only parameters that are candidates for round off are:

\[ \tilde{\beta} \]  
the inverse of the Simulated Annealing temperature, which can be interpreted as the weighting factor between fidelity and regularity;

\[ \mu_{gs} \]  
the mean values for each quantization class;

\[ -T \cdot \ln \alpha \]  
the threshold in the Simulated Annealing algorithm.

We tested the range of useful \( \tilde{\beta} \) values for our test set of images. We chose a set of possible \( \tilde{\beta} \) values and determined the value for which misclassification was minimal for each image. This gave us a collection of \( \tilde{\beta} \) values, which we assumed as the range needed for our algorithm. This range lay between 20 and 88 and it was found that integer values are sufficient and that the use of fixed point math is unnecessary. 7 Bits are sufficient to represent each integer value within this range.

We tested the effects of using integer values for \( \mu_{gs} \). The misclassification rates of all but the first image fell below the 5\% threshold. We therefore found it suitable to use integer values for \( \mu_{gs} \).
5.3. TRANSFORMATIONAL DEVELOPMENT

We tried to find a suitable rounding for $-T \cdot \ln \alpha$ but we could not find any. Even replacing this value with 0 had no effect. This essentially means that transforming the algorithm into a deterministic one (one which only lowers the energy) did not influence the outcome. However, in order to not over-simplify the algorithm used to evaluate the methodology, we rounded off threshold values to integers instead of replacing them with 0.

Random Number Generator

We chose to use a Linear Feedback Shift Register (LFSR) to generate pseudo random numbers on the Linedancer. Figure 5.2 shows the schematic of a LFSR. A LFSR is a rotating shift register, with a bit width of $W$. Its state is initialized with a non-zero value, the seed. Pseudo-random numbers are generated by calculating a new state, which depends on the values at some of the bit positions; the taps. The calculation of a state involves Xor-ing the bits at the tap locations and inserting the result at the least significant bit position, shifting the other bits one position to the right.

Some combinations of bit widths and tap positions give a sequence of states covering all possible states, except zero: these combinations are called maximal. Maximality is a desirable property for the random number generator, because it maximizes the period of possible generated numbers. More on Linear Feedback Shift Registers can be found at [16].

Because an LFSR is $W$ bits wide, it has a range of $\{1, 2, \ldots, 2^W - 1\}$. If
the LFSR is maximal, all states except zero are possible, but if it is seeded with zero, it stays zero. When we have \( N \) different states, states lie within a range of \( \{0, N-1\} \). This means that the values obtained from the LFSR have to be transformed into the range of the states.

Our first attempt at this, was by decreasing the value of the LFSR with 1 and use the remainder after division by \( N \). This can be found in Section B.1.2 in lines 13–19 from which the following lines have been taken, with some comments added for clarification (\( y \) is the input image, \texttt{random} is a function implementing a LFSR):

\[
\begin{align*}
\text{NB. Initialize seed to random values.} \\
\text{\texttt{seed=, ? ($ y.$) $ (# , y.)}} \\
\text{NB. Calculate new seed using a LFSR of 10 bits wide and taps at} \\
\text{NB. bits 10 and 7.} \\
\text{\texttt{seed=. (10 ; 10 7) random seed}} \\
\text{NB. Map seed to state by subtracting one and calculate the} \\
\text{NB. modulus with the number of labels LABELS.} \\
\text{\texttt{nstate=. LABELS | <: seed}} \\
\end{align*}
\]

This approach was wrong. First of all it uses the number of elements in the image to determine the maximum seed value, for which there is no good reason. Second, it still allows for seeds being initialized to 0. This error was only found and corrected in a later subphase, which is discussed later.

We tried to find a suitable bit width for the random number generator. We compared the output of our model using the LFSR implementation with a version using the built in J version by looking at the images. This made us decide on a bit width of 12 for the LFSR.

### 5.3.2 Reorganization Subphase

The goal of this subphase is to transform the system description into a form where all operations are performed in the same way as on the target hardware.

#### Simulation Length

Because we calculate everything in the same way and with the same precision as we will do on the Linedancer, we can reduce the number of iterations of the algorithm during tests. This reduces the time needed to verify the correctness of a translation step.

#### Globalization

All functionality is gathered within a single function and all variables representing memory fields are made global. This is done to mirror the situation on the Linedancer, where all memory fields are also essentially shared. This may look as bad programming practice, but actually this is the introduction of a lower implementation level.

#### Resource Assignment

Because the Linedancer has restrictions on how to use its operations and because memory for each PE is limited, we must be careful with our assignment of
variables to memory fields. Therefore, we used an Excel sheet for ‘keeping score’: keeping track of which bits of PE memory were used for which variable in the J model and warning us when we used too much space. When going through the Reorganization subphase, we bring our model code closer to the assignments used in the Excel sheet. We rename variables so their names include the offset and size they will have in PE memory. A variable name DUMMY is changed to DUMMY_4_16 meaning that it is a 16 bit wide value, located at bit offset 4.

Resource Sharing

During the reorganization subphase, the code is getting more and more disconnected with the original algorithm and becomes a sequence of very simple instructions. Sometimes this actually helps in simplifying the code. For example the variable mu in Section B.1.4 lines 19–49 can be merged with the use of the variable a. The result of this operation can be seen in Section B.1.5, lines 23–53. Because variables correspond to memory fields during the translation, this simplification frees up some extra bits in each PE for other uses.

Expansion

We expand parallel constructs that will have to be done in loops on the Linedancer. One example is the expansion of the code which calculates the energy of a state. In Section B.1.2 the following code can be found in line 21:

NB. Calculate the likelihood part of the energy by squaring the differences of the pixel values and label means.
lhood=. *: y. - state { MU

During the reorganization subphase this is transformed to the code in Section B.1.5 between lines 23–29:

NB. Fetch the pixel value.
a_14_10=. y_74_8
NB. For each label do...
for_l. i. LABELS do.
    NB. Tag the pixels with this label.
eq=. state_10_4 = l
neq=. -. eq
NB. Subtract means from pixel values.
a_14_10=. a_14_10 + (eq * - l { MU) + (neq * 0)
end.
NB. Square the difference between pixel values and label means.
e_82_20=. a_14_10 * a_14_10

Expression Reduction

In this subphase we also make sure that each line only does one operation for which there is a Linedancer equivalent. For example, take the calculation of fidelity from Section B.1.2 line 21:

NB. Calculate the likelihood part of the energy.
lhood=. *: y. - state { MU
The Linedancer does not have a square function, and it can only do one operation at a time, so the code is translated to:

\[
e = y - \text{state} \{ \text{MU} \\
e = e * e
\]

Normalization

We normalize loops, so they all have a simple, common form which will be easy to translate later during the Translation Phase. In Section B.1.3 lines 27–33 we find the following loop:

```plaintext
for_n. NBORS do.
    nb = n |:.0 state
    nb = state ^: nb
    eq = nb = 0
    neq = -. eq
    e = e + (eq * - BETA) + (neq * BETA)
end.
```

In this code, NBORS is a collection of neighbor offsets. This is normalized into a form where there is iterated over a list of increasing integers. After the translation, the offsets from a pixel to its neighbor are stored in a separate array and indexed with the iterated value (code taken from Section B.1.5 lines 31–37):

```plaintext
for_n. i. NBCOUNT do.
    a_10 = (n { NBORS) |:.0 state_10
    a_10 = state_10 ^: a_10
    eq = a_10 = 0
    neq = -. eq
    e_20 = e_20 + (eq * - BETA) + neq * BETA
end.
```

Here, NBCOUNT is the number of neighbor offsets.

Precalculation of Constants

Some constants can be precalculated in advance. The different temperatures and energy thresholds can be calculated in a table, and indexed during the loop. However, this does of course increase the space needed to run the algorithm, but it has the advantage that the calculation can be done on the SCC.

We try to identify values in the algorithm which can be precalculated. An example is the precalculation of \(-T \cdot \ln \alpha\), which is calculated and stored in a lookup table. This can be implemented on the Linedancer as an array, with an element for each time step.

5.3.3 Template Subphase

The goal of this subphase is to transform the description of the Reorganization Subphase into a form which is easy to parse and translate into the target hardware language. This means that we replace code by functions representing code templates.
5.3. TRANSFORMATIONAL DEVELOPMENT

Code templates can be used to perform checks on the data, such as checking calling conventions. For instance, the code in Section 5.1.8 lines 71–98, emulates the RTS(Mul) instruction of the Linedancer, but it also checks the arguments against the calling conventions as found in the Linedancer programmer's manual.

Code templates abstract away from common combinations of operations. For example, the following construct can be found in lines 50–52 in Section B.1.5:

```
NB. Tag values equal to 0.
  eq=. a_14_4 = 0
  neq=. -. eq
NB. Subtract BETA from all tagged values and add BETA to all NB. other values.
ne_24_20=. ne_24_20 + (eq * - BETA) + neq * BETA
```

These lines of code represent the concept of selecting a subset of values from an array equal to a constant (in this case 0) and subtracting a constant BETA from the selected values and adding this constant to others. This concept is encapsulated in the J code as a function with the hopefully descriptive name SelectiveAddSubValue and called like this (this should actually be on one line to be valid J code):

```
SelectiveAddSubValue 'ne_24_20' ; 'ne_24_20' ; 'a_14_4' ; 0 ; BETA ; 'Int'
```

The SelectiveAddSubValue itself can be translated into Linedancer aop instructions. The result of the translation can be found in Section B.2.2 lines 141–148:

```
// Activate all PEs.
Activate(All,-,-,-),
// Write 1 to ab0 field, activating RTS operations on all PEs.
Write(Binary, 1, ab, 0),
// Clear 0 to ab1 field on all PEs.
Write(Binary, 0, ab, 1),
// Tag PEs where memory field @{4,14} equals 0.
Tag(Binary, 0, 4, 14, tr1, -),
// Activate the matching PEs.
Activate(Matching, -, tr1, -),
// Write 1 to ab1 fields on activated PEs.
Write(Binary, 1, ab, 1),
// Add BETA to memory field @{20,24} where ab1 equals 0 and
// subtract BETA from it everywhere else.
RTS(AddSub, Int, -, @{20,24}, @{20,24}, BETA),
```

For our purposes, we only needed one tag register of the Linedancer. Therefore, we used the register tr0 everywhere.

The bits of PE memory can be individually addressed. PE memory is not emulated on a bit level, only on a value level. This means that for these kinds of operations a workaround had to be found. We did this by assigning results to variables not involved in the translation.
We chose this solution because we did not have a lot of places where we had to work on individual bits and because a bit-wise representation would have a negative impact on performance.

Parts of J code can be excluded from the translation. Only J code within the comment pair \texttt{NB.\{ and \texttt{NB.\})} is translated. This makes it possible to do things ‘behind the screens’ which should not be directly translated. For example, the first part of the following code fragment forces a one dimensional list of random numbers, which are used as seed values for the random number generator, into a two dimensional, rectangular form needed by the J implementation. This part is not translated. The second part of the code assigns these values to another memory field and does get translated.

\texttt{NB. Force seed\_0\_5 into the right shape.}
\texttt{seed\_0\_5=\ ($ y.\ ) \$ seed\_0\_5 \quad \texttt{NB. not translated}}
\texttt{\texttt{NB.\{} \texttt{AssignField 'state\_10\_4'; 'seed\_0\_4' \quad \texttt{NB. translated}}
\texttt{\texttt{NB.}}}\texttt{\texttt{\}}}\texttt{\texttt{\}}

We created a code template for encapsulating the data transfer between PDS and PE memory. One of the places this was used to load initial seed values for the random number generator. This is an example of a code template having a somewhat different role in the J model and in the Linedancer code.

\textbf{Initial Randomization}

The initial random labeling is done by loading a file with a precalculated list of random values. This made it easier to compare results between the J model and Linedancer implementation, because the same file could be used in both.

\textbf{Problem Resolution}

The version using the LFSR had some disturbing single dots in the result, for which we did not know the cause at first. Only later in the implementation we found out that the LFSR state was wrongly initialized. This bug was only found, when we were well inside the Template Subphase. The reason why this bug survived for so long, was that the random number generator was only tested by comparing it by eye with results obtained from the built in J random number generator. Eventually the problem was resolved by increasing the initial seed values by one and letting the seed value range depend on the width of the LFSR (Section \ref{sec:randomization} lines 12,16,17):

\texttt{NB. Initialize seed with a number between 1 and 2\^5 - 1}
\texttt{seed=. >: ? ($ y.\ ) \$ <: 2\^5}
\texttt{NB. Calculate new seed using a LFSR of 5 bits wide and taps at}
\texttt{NB. bits 5 and 3.}
\texttt{seed=. (5 ; 5 3) random seed}
\texttt{NB. Map seed to state by calculate the modulus with the number}
\texttt{NB. of labels LABELS.}
\texttt{nstate=. LABELS | seed}

After fixing the problem with the LFSR we were able to drop the width of the LFSR to only 5 bits while still getting reasonable results.
This is an illustration of how faults in the design can still get into the template subphase. Here the fault was easily fixed in the last version of the trade-off subphase. The fault was also easily fixed in the last version of the reorganization subphase. These two versions should now be totally equivalent (if both fixes were correct), so this can be tested. This means that all intermediate transformation steps can be skipped. This also holds true for the step from reorganization to template subphase.

5.3.4 Translation Subphase

The goal of this subphase is to translate the system description from the Template Subphase directly into the target hardware language.

The Parser

For the translation a simple parser was written using Yapps, a tool for writing compilers running in Python. Yapps is very suitable for writing quick and dirty compilers and translators for problems too difficult to solve with regular expressions.

The implemented translator takes specially formatted J code as input and translates this into code that can be compiled using the Linedancer toolchain. The parser takes as input the J code of the model and a definition file containing declarations of variables (Section 4.2.1). The parser then creates a file that can be included into a wrapper file (Section 4.2.3). The wrapper file is the actually file that is compiled using the Linedancer compiler. The wrapper is a separate entity because it does not change very often.

The translator can translate loops in the J model into loops in Linedancer code. For example, the following code (Section 4.1.7, lines 53–63)

```j
for_n. i. NBCOUNT do.
    FetchNeighbor 'a_14_10'; 'state_10_4'; (n { NBORS)
    a_14_4=: LABELS | a_14_10
    Xor 'a_14_4'; 'state_10_4'; 'a_14_4'
    SelectiveAddSubValue 'e_82_20'; 'e_82_20'; 'a_14_4'; 0 ; BETA ; 'Int'
end.
```

is translated into (Section 4.2.2, lines 79–101, omitting some less important code):

```c
for (n = 0; n < NBCOUNT; ++n) {
    aop{ /* ... */ );
    temp = NBORS[n];
    aop{
        /* ... */
        RTS(Assign, Bitset, co{Get,temp}, @{10,14}, @{4,10}, -),
        /* ... */
    }
}
```

Note that the loop variable `n` is translated correctly. Because it is not possible to use arrays inside RTS instructions, the value of `NBORS[n]` is first assigned to a temporary variable. This case is also automatically translated.
Interface To Implementation Language

Some templates take extra parameters not used in the J model, but used as parameters in the generated code. One example is the `LoadFile` template, which takes a rendezvous setting as third parameter (Section B.1.7 line 4):

```
LoadFile 'seed_0_5' ; 'seed31.dat' ; 'ro{pdtRdv,sdtRdv}'
```

The translator copies this argument verbatim as an argument for the `pdTransfer` instruction (Section B.2.2 lines 26 and 27):

```
Activate(All,-,-,-),
pdTransfer(Load, 5, 0, -, ro{pdtRdv,sdtRdv}),
```

This copying is useful, because it allows us to change these arguments in the same file as the rest of the source code.

Testing The Implementation

During the translation subphase, final bugs in the translation are exposed. It is best to incrementally test the translation by started with a translation of a model with almost all functionality commented out. The equivalence between the J model and the Linedancer implementation is tested. After the tests are successful, more code is uncommented and this process continues until the whole program can be translated and run successfully on the Linedancer.
Chapter 6

Evaluation

6.1 Quantization

In this section we show some quantitative and qualitative results concerning our quantization algorithm.

6.1.1 Algorithm Load Distribution

To give an idea how many cycles the different parts of the algorithm take, we measured the number of cycles taken for different stages of the algorithm. The results can be seen in Table 6.1. The stage taking the most cycles is the processing of neighbors, where $\beta$ is added or subtracted from the energy.

6.1.2 Algorithm Output Quality

The quality of the output of our quantization algorithm is not as good as that of existing algorithms.

When we compare the output of our algorithm with a reference quantization obtained with a commercial program such as Irfanview, we get results as can be seen in Figure 6.1. The reference quantization has a smooth, somewhat cloudy

![Figure 6.1: Comparison of quantizations by (a) a commercial program (Irfanview) and (b) our algorithm.](image)

51
CHAPTER 6. EVALUATION

Preparation

Processing one tile

Calculate a new random labeling 44
For each label 352
\[ y - \mu_{g_s} \]
Square 164
For each neighbor 3592
  Add or subtract \( \beta \)
Load \( y \)
For each label 352
\[ y - \mu_{g_s} \]
Square 164
For each neighbor 3592
  Add or subtract \( \beta \)
Subtract energies, threshold values and update 70

Dump result 338

Table 6.1: Number of cycles used for each stage of the algorithm. Nesting indicates loops, bold numbers indicate accumulated results.

<table>
<thead>
<tr>
<th># Tiles</th>
<th>DSM Cycles / Tile</th>
<th>%</th>
<th>ISM Cycles / Tile</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>25475</td>
<td>74.4252</td>
<td>8754</td>
<td>25.5748</td>
</tr>
<tr>
<td>8</td>
<td>22550.2</td>
<td>72.0357</td>
<td>8754</td>
<td>27.9643</td>
</tr>
<tr>
<td>28</td>
<td>21824.6</td>
<td>71.3721</td>
<td>8754</td>
<td>28.6279</td>
</tr>
<tr>
<td>98</td>
<td>21686.1</td>
<td>71.2419</td>
<td>8754</td>
<td>28.7581</td>
</tr>
<tr>
<td>378</td>
<td>21687.2</td>
<td>71.2429</td>
<td>8754</td>
<td>28.7571</td>
</tr>
</tbody>
</table>

Table 6.2: Processing time of DSM and ISM threads per tile when one annealing loop is performed.

appearance, while the output of our algorithm contains blobs with uniform color. This can be explained by the different approaches taken by the algorithms. The algorithm used by Irfanview only calculates a mapping from original color values to quantized color values, while our algorithm tries to create regions of uniform color.

Because the data is split up into tiles fitting inside the Linedancer, tiling artifacts can be visible in the output files. An example can be seen in Figure 6.2. Inside the red rectangle the sharp border of the tile can be seen.

6.1.3 Performance Analysis

We have measured the processing time of DSM (communication) and ISM (processing) threads to get an idea of their relative running time. When we let the algorithm perform only one annealing loop, most processing is done inside the DSM thread (Table 6.2). Here the SDMC data transfers form the bottleneck of the implementation. Note that the time spent inside the DSM thread decreases slightly when there are more tiles to process, but that it basically needs a constant number of cycles for each tile. The number of cycles spent inside the ISM thread is constant, because it only depends on the number of Simulated
Figure 6.2: Tiling artifact in the output of the quantization algorithm. If you look close inside the red rectangle you can see that the blobs have horizontal and vertical borders. These are the tile edges.
### EVALUATION

#### Tiles

<table>
<thead>
<tr>
<th># Tiles</th>
<th>DSM</th>
<th>ISM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles / Tile</td>
<td>%</td>
<td>Cycles / Tile</td>
</tr>
<tr>
<td>2</td>
<td>25558</td>
<td>23.3564</td>
</tr>
<tr>
<td>8</td>
<td>22671.5</td>
<td>21.2799</td>
</tr>
<tr>
<td>28</td>
<td>21959.2</td>
<td>20.7501</td>
</tr>
<tr>
<td>98</td>
<td>21794.5</td>
<td>20.6265</td>
</tr>
<tr>
<td>378</td>
<td>21797.7</td>
<td>20.6289</td>
</tr>
</tbody>
</table>

Table 6.3: Processing time of DSM and ISM threads per tile when ten Simulated Annealing loops are performed.

#### Speedup

![Figure 6.3: Speedup for various image sizes and number of annealing loops.](image)

After we increase the load inside the ISM thread by performing ten annealing loops, the Simulated Annealing forms the bottleneck (Table 6.3). The number of cycles per tile needed for the ISM thread in this case is 9.58 times more than the case with only one Simulated Annealing loop. This is of course due to the factor 10 difference in the number of loops.

### 6.1.4 Speedup

We have implemented a sequential reference implementation in C to determine the speedup, the gain in processing speed by using a parallel implementation instead of a sequential one. This implementation followed the parallel implementation as closely as possible. The results can be seen in Figure 6.3 for three different numbers of annealing loops and five image sizes. These numbers were calculated by dividing the running time of the sequential implementation by the running time of the parallel implementation on the Linedancer. The sequential implementation was run on an Intel Pentium Xeon running at 2 GHz. In Table 6.4, the actual running times are shown.

We can conclude that the parallel implementation is always faster. However, the output quality is not as good as can be achieved using a program like Irfanview on the Pentium, which is also faster than the implementation of our algorithm on the Linedancer. Smaller images and running with small annealing.
### 6.1. QUANTIZATION

<table>
<thead>
<tr>
<th># Pixels</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pentium</td>
</tr>
<tr>
<td>10000</td>
<td>5.75</td>
</tr>
<tr>
<td>40000</td>
<td>22.5</td>
</tr>
<tr>
<td>160000</td>
<td>91.3</td>
</tr>
<tr>
<td>640000</td>
<td>374</td>
</tr>
<tr>
<td>2560000</td>
<td>1500</td>
</tr>
</tbody>
</table>

(a) 1 Run

<table>
<thead>
<tr>
<th># Pixels</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pentium</td>
</tr>
<tr>
<td>10000</td>
<td>52.5</td>
</tr>
<tr>
<td>40000</td>
<td>211</td>
</tr>
<tr>
<td>160000</td>
<td>856</td>
</tr>
<tr>
<td>640000</td>
<td>3530</td>
</tr>
<tr>
<td>2560000</td>
<td>14100</td>
</tr>
</tbody>
</table>

(b) 10 Runs

<table>
<thead>
<tr>
<th># Pixels</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pentium</td>
</tr>
<tr>
<td>10000</td>
<td>517</td>
</tr>
<tr>
<td>40000</td>
<td>2070</td>
</tr>
<tr>
<td>160000</td>
<td>8420</td>
</tr>
<tr>
<td>640000</td>
<td>34600</td>
</tr>
<tr>
<td>2560000</td>
<td>138000</td>
</tr>
</tbody>
</table>

(c) 100 Runs

Table 6.4: Running times of the quantization algorithm on a Pentium and on the Linedancer.
cycles have less benefit from the parallelization. Increasing the number of annealing steps increases the time spent in the parallel part of the algorithm and therefore has a positive impact on the speedup.

6.2 Methodology

6.2.1 Design and Development

It was easy to evaluate design decisions. We can give the implementation of the random number generator as an example. When replacing the native randomization function with our own version, it was easy to make a first, separate implementation of a LFSR. This part could easily be tested separately from the rest of the code, until we were sure its functionality was correct. Later it could be incorporated with the rest of the code, and the only thing left was to determine the actual width of the LFSR.

Another benefit was the easy comparison of output during the different phases. All output could be accessed within the J interpreter and during the translation phase it could also be compared with the output of the Linedancer.

6.2.2 The J Language

We used the J language for the incremental prototyping and transformational development. This language has some nice benefits, but also some shortcomings. J is a very concise and expressive language, where a lot of functionality can be hidden in a few lines of code. This has the benefit of expressing ideas in very little space, which helps getting a grip on the problem (conceptual integrity). However, this conciseness also has a downside: J code can be very hard to understand for newcomers to the language, and even for people with more experience in the language.

6.3 Hardware

We had very good experiences with the support of the Linedancer. We could mail every question to Aspex, and we would receive an helpful answer in a short time.

The documentation of the Linedancer was in worse condition. Some functionality mentioned in the documentation, was not available on the hardware itself. An example of this are floating point memory fields. Other parts of the documentation where very vague, such as the description of SDMC transfers.

Because the Linedancer compiler is actually a wrapper around the GCC compiler with some extra translation steps, error messages can get very cryptic. An example occurs when compiling code not fulfilling the calling conventions. Also, because of the translation steps, line numbers in the compiler output are not always accurate. These things can make debugging rather difficult.
6.4 Possible Enhancements

Here we will mention some possible enhancements we thought of during the development of the quantization algorithm or found in the literature.

6.4.1 Algorithm

Multiscale

In the literature (e.g. [14]), MRF and Simulated Annealing algorithms benefit from a multiscale implementation. This means that the algorithm is run at various resolutions, starting using a course grid and moving to finer grids later in the algorithm. Such a top down approach could make the algorithm more robust against errors at pixel level because the processing at a lower level starts with the more or less good approximation of a higher level.

Other Optimization Target

The median cut and octree quantization algorithms use the relative frequency of colors within an image to calculate a mapping from the original color space to the quantized color space. Our quantization algorithm maps individual pixels to labels, so it is possible for pixels with the same original color to end up having different quantized values. If we throw away this extra freedom, and instead optimize a mapping of pixel values to pixel labels, the algorithm is not dependent on the number of pixels anymore, but only on the number of original colors. The number of different states is then reduced which could improve the Simulated Annealing results.

Learning Algorithm

At this point, the algorithm needs all its parameters given in advance. This is not a realistic scenario. Therefore it would be better if the algorithm could learn these parameters from training data. We tried one approach of learning the mean values, by starting with means evenly divided over the possible labels and calculate new means during the algorithm. However, implementing this calculation on the Linedancer proved to be difficult within the given time frame.

Alternative Randomizing

The algorithm might give better results if, instead of using a random quantization to initialize the state at the beginning of the algorithm, we initialize it using the input quantized with a simple and fast but less accurate quantization algorithm.

This could be combined with the use of a different proposal distribution than the uniform distribution used in our algorithm. For instance, a normal distribution dependent on the current state of a pixel could be used, so new labellings are chosen close to the current state.
6.4.2 Hardware

Optimizing Transfers

Our focus was with implementing the ISM part of the Linedancer code and then only the part running on the PEs. So, we did not fully optimize the data transfers between the different threads. A lot can still be gained here.

Our current implementation transfers data between the SDS and PDS one byte at the time. The Linedancer permits the transfer of eight bytes at the time. This could give a factor 8 speed increase for these transfers, but this only helps if the number of Simulated Annealing steps is low, because otherwise the Simulated Annealing is the bottleneck. So in order to profit from this optimization, streaming also has to be optimized using double buffering, so transfers and Simulated Annealing can be done concurrently.
Chapter 7  

Conclusions and Recommendations

7.1 Mapping and Methodology

We have successfully mapped the quantization algorithm to the Linedancer: the algorithm was first implemented in a functional model and this eventually resulted into a final running implementation on the Linedancer.

We have extended the Evolutionary Design methodology by stratifying the Transformational Development phase into four subphases: Trade-Off, Reorganization, Template and Translation. These extensions give a more systematic understanding of the Transformational Development phase and in this way offer extra support to overcome the relative complexity of design.

Throughout the Transformational Development we have been able to guard the quality of our prototype by keeping it executable. Finally, code for the Linedancer, generated from the prototype could be checked against the prototype in a bit true way.

The use of the J programming language proved to fit the methodology well. It made it possible to examine the state of the prototypes and made it easy to emulate the parallelism of the Linedancer. However, it may be difficult to introduce J into other projects, because people are hesitant to learn a new programming language. Especially when its syntax is as Spartan as that of J.

7.2 Quantization

The quality of our quantization algorithm is not on par with that of current algorithms as used by commercial products like Irfanview. Especially the tiling artifacts and the appearance of blobs in the final result have a negative impact on the quality.

However, the probabilistic approach taken by the algorithm is an interesting way to specify algorithms. Instead of generatively specifying how the output is constructed from the input, an output is searched using an energy function. This has the potential to reduce design complexity, because it is only based on the result we want to achieve, and nothing more. Therefore it may be worthwhile...
to investigate these kinds of algorithms further. The art of finding a suitable energy function for a given problem should be the first thing to be researched.

The Simulated Annealing optimization and MRF image model introduce a lot of parameters. This makes it difficult to apply the algorithm, because the influence of these parameters is not very clear.

An interesting improvement of our algorithm would be to make use of a multiscale approach. This should make the algorithm behave better on a global level and prevent influences from disturbances at pixel level, which is needed to compete with global algorithms such as median cut and octree.

7.3 Linedancer

The Linedancer makes it possible to write an implementation of our quantization algorithm which runs multiple factors faster than on a Pentium. Our Linedancer implementation relies heavily on communication between PEs, which forms the largest bottleneck during Simulated Annealing.

It might be beneficial for the quantization algorithm to have more communication between tiles. Having the borders of the tiles overlap implies performing multiple passes over the whole image instead of treating each tile in one pass. This may give better output results, such as reduction of tiling artifacts.

The optimization of SDMC transfers should also be investigated, together with the possibilities of double buffering. This could result in a better balance between data transfer and calculations times. Optimized transfers are especially interesting when performing multiple passes on the image.
Bibliography


Appendix A

Introduction to J

In this chapter we will give a short introduction in J which might be helpful for understanding the code excerpts in the text. We have tried to make this introduction short and focused on the use of the J language within this text. This meant that we had to make some simplifications and generalizations. The interested reader can refer to [8, 9] for a more complete introduction into the language.

A.1 Comments

Comments in J start with NB. and continue to the end of the line:

NB. This line is ignored

A.2 Verbs and Nouns

In J, the equivalents to functions and variables are called verbs and nouns respectively.

A.2.1 Verbs

Verbs are operations on either one or two nouns. Verbs operating on one noun are called monads, verbs operating on two nouns are called dyads. Verbs are evaluated right to left, which means that a phrase like

\[ 2 * 3 + 4 \]

evaluates to 14 instead of 10. This may seem strange at first, but has the benefit that the phrase can be read as ‘two times the sum of 3 and 4’ and is also evaluated this way. Another benefit is that this evaluation method applies to all verbs, so operator precedence is really simple. Contrast this to operator precedence in C, where one probably has to search for an operator precedence table to see what is (probably) wrong with

\[ x & 1 == 0 \]

Just like in C, the precedence order can be influenced by using parentheses:
(2 * 3) + 4
evaluates to 10.

A.2.2 Nouns
Each noun is an array with a certain shape. For instance, a $10 \times 8$ matrix is represented by an array with shape $10$ 8, a vector of size 10 is represented by an array with shape 10 and a scalar value is represented by an array with an empty shape. To create a matrix of size $10 \times 8$ (an array with shape $10$ 8) filled with sevens, we can use the dyadic verb $\$ \$ like in the following J code:

NB. Create a (10,8) matrix filled with 7s.

$10$ 8 $\$ \$ 7$

If we want to know the shape of a noun $y$ we can use the monadic version of the verb $\$:

NB. Find out the shape of the noun $x$

$\$ x

Nouns can also be strings, delimited by single quotes:

s=: 'This is a string'

A.3 Assignment
J has two assignment operators, $\_.$ and $\_=$. The only difference is, that $\_.$ assigns a value in a local scope, while $\_=$: assigns a value in a global scope.

NB. Assign 1 to a global noun 'oneG'

oneG=: 1

NB. Assign 1 to a local noun 'oneL'

oneL=. 1

A.4 Arithmetic and Comparison
Arithmetic operators, such as addition and subtraction, operate on single elements of nouns. For example, if $a$ and $b$ each are vectors with dimension 10 (in J speak: arrays with shape 10), then the following adds them and puts the result in $r$:

NB. Add nouns $a$ and $b$, put result in $r$.

r=: a + b

Trying to add two arrays with different shapes is an error.

J treats comparison operators like $\lt;\lt;\lt;\lt;\lt$ as ordinary arithmetic functions returning 0 or 1 to represent true and false. This results in the common idiom for assigning to $r$ elements of $v1$ for which condition $b$ holds, and assigning elements of $v2$ where $b$ does not hold:

NB. Assign $v1$ to $r$ where $v1$ equals 2, and assign $v2$ everywhere else.

b=: v1 = 2
r=: b * v1 + (-. b) * v2
A.5. ARRAY OPERATIONS

<table>
<thead>
<tr>
<th>J Verb</th>
<th>C</th>
<th>Mathematical description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*: y</td>
<td>pow(y,2.0)</td>
<td>$y^2$</td>
</tr>
<tr>
<td>x + y</td>
<td>+</td>
<td>$x + y$</td>
</tr>
<tr>
<td>x - y</td>
<td>-</td>
<td>$x - y$</td>
</tr>
<tr>
<td>x = y</td>
<td>==</td>
<td>$x = y$</td>
</tr>
<tr>
<td>x * y</td>
<td>*</td>
<td>$x \cdot y$</td>
</tr>
<tr>
<td>x ^ y</td>
<td>pow(x,y)</td>
<td>$x^y$</td>
</tr>
<tr>
<td>&gt;: y</td>
<td>+ 1</td>
<td>$y + 1$</td>
</tr>
<tr>
<td>&lt;: y</td>
<td>- 1</td>
<td>$y - 1$</td>
</tr>
<tr>
<td>x</td>
<td>. y</td>
<td>mod x</td>
</tr>
<tr>
<td>-. y</td>
<td>! y</td>
<td>$-y$</td>
</tr>
<tr>
<td>x ^: y</td>
<td>x != y</td>
<td>$x \neq y$</td>
</tr>
<tr>
<td>x &lt;: y</td>
<td>x &lt;= y</td>
<td>$x \leq y$</td>
</tr>
</tbody>
</table>

Table A.1: J’s arithmetic and comparison verbs, together with their equivalents in C and mathematical notation.

<table>
<thead>
<tr>
<th>J Verb</th>
<th>C</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>. y</td>
<td>Rotate the contents of an array</td>
</tr>
<tr>
<td>x</td>
<td>.!.0 y</td>
<td>Shift the contents of an array</td>
</tr>
<tr>
<td>? y</td>
<td>rand()</td>
<td>For each value $v$ in $y$, generate a random integer in the range ${0,1,\ldots,v-1}$</td>
</tr>
<tr>
<td>x { y</td>
<td>y[x]</td>
<td>Extract an element located by $x$ from array $y$</td>
</tr>
</tbody>
</table>

Table A.2: Operations on whole arrays.

The most important arithmetic functions for understanding this text, can be found in Table A.1 together with their counterparts in C and in mathematical notation.

A.5 Array Operations

Some operations operate on whole arrays (Table A.2). The verb |. rotates the values in its right argument in a way indicated by its left argument. This can be used to simulate a rotating register. The verb |.!.0 does the same, but shifts instead of rotates, filling undefined locations with 0. This can be used to simulate a shift register.

NB. 1 0 1 1 is eleven in binary notation
NB. This results in 1 1 0 1
1 |. 1 0 1 1
NB. This results in 0 1 1 0
1 (|.!.0) 0 1 0 1

It is also possible to generate arrays with randomly generated values, lying within a given range. For example:

NB. Generate 5 random numbers in the range $\{0,1,\ldots,3\}$
? 4 4 4 4 4
APPENDIX A. INTRODUCTION TO J

<table>
<thead>
<tr>
<th>J Verb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.</td>
<td>Generate list of integers</td>
</tr>
<tr>
<td>;</td>
<td>Concatenate objects of arbitrary type</td>
</tr>
<tr>
<td>,</td>
<td>Concatenate arrays of same rank</td>
</tr>
<tr>
<td>,:</td>
<td>Concatenate arrays of different rank</td>
</tr>
</tbody>
</table>

Table A.3: Miscellaneous verbs.

A.6 Miscellaneous Verbs

In Table A.3 we list some verbs we used in the text which cannot be fit into the previous categories.

The phrase i. y generates an increasing vector of integers of size y, starting with 0:

\[
\text{NB. Assign to a the values 0 1 2 3 4}
\]

\[
a =. i. 5
\]

In order to concatenate values of different types, we can use the verb ;. To concatenate two arrays of the same shape resulting in an array of greater shape, we can use the verb ,: to concatenate two arrays of the same shape, resulting in an array of the same shape, we can use the verb ,.

J also supports for loops:

\[
\text{for_t. 2 3 4 do.}
\]

\[
\text{end.}
\]

is the same as the following C code:

\[
\text{for (t = 2; t <= 4; ++t) { }
\]

\[
}
\]
Appendix B

Listings

This appendix contains source code of the different models (written in J) and Linedancer implementations (written in C/Linedancer code). The source code in these listings has been cleaned up to contain only the functions that might be interesting to the reader. Unnecessary comments, driver code and debugging code have been removed.

B.1 Models

These source files were written during the Incremental Development phases of our design methodology. The files were written before we defined a clear definition of the subphases, so the filenames might not be very clear, although a mapping can be made between the filenames used here and the subphases we finally identified:

<table>
<thead>
<tr>
<th>Subphase</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trade-Off</td>
<td>functional*.ijs</td>
</tr>
<tr>
<td>Reorganization</td>
<td>algorithm*.ijs</td>
</tr>
<tr>
<td>Template</td>
<td>architecture*.ijs</td>
</tr>
</tbody>
</table>

For each model only the implementation of the quantization verb is shown, because this contains the actual algorithm.

B.1.1 File: functional.ijs

This is the first complete implementation of our quantization algorithm.

```
1 annealing=: 3 : 0
2   (9!:1) 7^5
3
4 state=. ? ($ y.) $ LABELS
5 for_t. TEMPO * COOLING " i. RUNS do.
6   nstate=. ? ($ y.) $ LABELS
7
8 lhood=. *: y. - state { MU
9 prior=. BETA * +/ state (-.0:+:0:=)_1 NBORS _1 state
10 e=. lhood + prior
```

67
APPENDIX B. LISTINGS

B.1.2 File: functional4.ijs

In this file, we removed the dependency on the J random number generator and replaced it with our LFSR implementation. The use of the LFSR contains a flaw, because we try to avoid a register state of 0 in a wrong way.

```j
random=: 4 : 0"_ 0
'len taps'=. x.
taps=. <: taps
state=. (- len) {. #: y.
b=. ":/ taps { state
#. len {. b , state
)
annealing=: 3 : 0
(9!:1) 7^5
NB. Give all pixels a unique starting value.
seed=. ? ($ y.) $ (# , y.)
state=. LABELS | seed
for_t. TEMPO * COOLING ^ i. RUNS do.
seed=. (10;10 7) random seed
nstate=. LABELS | <: seed
lhood=. *: y. - state { MU
prior=. BETA * / state (-.0:+0:=)"_ _1 NBORS |."0"1 state
e=. lhood + prior
lhood=. *: y. - nstate { MU
prior=. BETA * / nstate (-.0:+0:=)"_ _1 NBORS |."0"1 state
e=. lhood + prior
de=. ne - e
accept=. de <: - t * ^. ALPHA
state=. (nstate * accept) + state * -. accept
end.
state
```

| lhood=. *: y. - nstate { MU |
| prior=. BETA * / nstate (-.0:+0:=)"_ _1 NBORS |."1 _ state |
| ne=. lhood + prior |
| de=. ne - e |
| accept=. de <: - t * ^. ALPHA |
| state=. (nstate * accept) + state * -. accept |
| end. |
| state |
B.1.3 File: functional6.ijs

In this file, we fixed the LFSR flaw from the previous implementation. We also decreased the width of the LFSR.

```
random=: 4 : 0"_ 0
  'len taps'=. x.
  taps=. <: taps
  state=. (- len) #: y.
  b=: ":/ taps { state
  #: len { b , state
)
```

```
annealing=: 3 : 0
  (9!:1) 7"5
  seed=. >: ? ($ y.) $ <: 2^5
  state=. LABELS | seed
  for_t. TEMPO * COOLING ^ i. RUNS do.
    seed=. (5;S 3) random seed
    nstate=. LABELS | seed
    lhood=. *: y. - state { MU
    prior=. BETA * +/- state (-.0:+.0=)_1 NBORS | .! 0"1 _ state
    e=. lhood + prior
    lhood=. *: y. - nstate { MU
    prior=. BETA * +/- nstate (-.0:+.0=)_1 NBORS | .! 0"1 _ state
    ne=. lhood + prior
    de=. ne - e
    accept=. de <: <. - t * ^ ALPHA
    state=. (nstate * accept) + state * -. accept
  end.
  state
)```

B.1.4 File: algorithm6.ijs

In this file, based on the implementation from file functional4.ijs, we replaced the call to the verb implementing the LFSR with inline code. We also replaced the implicit loops contained in the calculations of lhood and prior with explicit ones.

```
annealing=: 3 : 0
  (9!:1) 7"5
  NB. Give all pixels a unique starting value.
  seed=. ? ($ y.) $ (# , y.)
  state=. LABELS | seed
```
for_t. i. RUNS do.
    s=. _10 {."1 #: seed
    b1=. TAP1 {"1 s
    b2=. TAP2 {"1 s
    b=. b1 ~: b2
    ns=. 9 {."1 s
    seed=. #. b ,"_2 ns
    seed=. seed - 1
    nstate=. LABELS | seed
    seed=. seed + 1
    mu=. 0
    for_l. i. LABELS do.
        eq=. state = l
        neq=. -. eq
        mu=. mu + eq * l { MU
    end.
    a=. y. - mu
    e=. a * a
    for_n. NBORS do.
        nb=. n |.!.0 state
        eq=. state "=: nb
        eq=. nb = 0
        neq=. -. eq
        e=. e + (eq * - BETA) + (neq * BETA)
    end.
    mu=. 0
    for_l. i. LABELS do.
        eq=. nstate = l
        neq=. -. eq
        mu=. mu + eq * l { MU
    end.
    a=. y. - mu
    ne=. a * a
    for_n. NBORS do.
        nb=. n |.!.0 state
        eq=. nstate "=: nb
        eq=. nb = 0
        neq=. -. eq
        ne=. ne + (eq * - BETA) + (neq * BETA)
    end.
    ne=. ne - e
    accept=. ne <: t { ALPHAs
    state=. (nstate * accept) + state * -. accept
    end.
    state}
B.1. MODELS

B.1.5 File: algorithm9.ijs

In this file, based on the implementation from file algorithm6.ijs, we added offset and size information to all variables modeling memory fields. We also came to the conclusion that we did not need a separate mu variable and thus removed it.

```
annealing=: 3 : 0
(9!:1) 7^5

NB. Give all pixels a unique starting value.
seed_0_10=. ? ($ y.) $ (# , y.)
state_10_4=. LABELS | seed_0_10

NB. Give pixel value a name
y_74_8=. y.

for_t. i. RUNS do.
    NB. Randomize new state
    s=. _10 .*'1 #: seed_0_10
    seed_0_1=. TAP1 .*'1 s
    seed_3_1=. TAP2 .*'1 s
    b_64_1=. seed_0_1 " : seed_3_1
    ns_65_9=. 9 .*'1 s
    seed_0_10=. #. b_64_1 ,"2 ns_65_9
    seed_0_10=. seed_0_10 - 1
    seed_0_10=. seed_0_10 + 1

    NB. Calculate current state energy
    a_14_10=. y_74_8
    for_l. i. LABELS do.
        eq=. state_10_4 = l
        neq=. -. eq
        a_14_10=. a_14_10 + (eq * - l { MU) + (neq * 0)
    end.
    e_82_20=. a_14_10 * a_14_10
    for_n. i. NBCOUNT do.
        a_14_10=. (n { NBORS) |.!.0 state_10_4
        eq=. a_14_10 = 0
        neq=. -. eq
        e_82_20=. e_82_20 + (eq * - BETA) + neq * BETA
    end.

    NB. Calculate new state energy
    a_14_10=. y_74_8
    for_l. i. LABELS do.
        eq=. seed_0_4 = l
        neq=. -. eq
        a_14_10=. a_14_10 + (eq * - l { MU) + neq * 0
    end.
    ne_24_20=. a_14_10 * a_14_10
    for_n. i. NBCOUNT do.
        a_14_10=. (n { NBORS) |.!.0 state_10_4
```

APPENDIX B. LISTINGS

```
    a_14_10=. seed_0_4 "=: a_14_10
    eq=. a_14_10 = 0
    neq=. -. eq
    ne_24_20=. ne_24_20 + (eq * - BETA) + neq * BETA
end.

    NB. Compare energies
    ne_24_20=. ne_24_20 - e_82_20
    accept_1004_1=. ne_24_20 <: t { ALPHAs
    eq=. accept_1004_1 = 1
    neq=. -. eq
    state_10_4=. (eq * seed_0_4) + neq * state_10_4
end.

    state_10_4
```

B.1.6 File: architecture15.ijs

This file was derived from file algorithm9.ijs. Code templates have been substituted for blocks of code.

```
load jpath’~addons/image3/image3.ijs’
load jpath’~addons/image3/view_m.ijs’
load jpath’d:/leroy/trunk/quantization/models/templates2.ijs’
annealing=: 3 : 0
NB.{{
    NB. Give all pixels a unique starting value.
    LoadFile 'seed_0_10' ; 'd:/leroy/trunk/quantization/seed1023.dat' ; 'ro{pdtRdv,sdtRdv)
NB.}}
    NB. Force seed_0_10 into the right shape.
    seed_0_10=: ($ y.) $ seed_0_10
    NB. Explicitly assign bits to seed_0_4
    seed_0_4=: LABELS | seed_0_10
NB.{{
    AssignField 'state_10_4' ; 'seed_0_4'
NB.}}
    pixels=. y.
NB.{{
    NB. Give pixel value a name
    LoadField 'y_74_8' ; pixels ; 'ro{pdtRdv'
NB.}}
    NB.{{
        for_t. i. RUNS do.
        NB. Randomize new state
        NB. Get the bit representation so we can work
        NB. at bit level.
        s=. _10 {."1 #: seed_0_10
        NB. These assignments are used as an alias for these
        NB. bits so they can be used in Rts instructions.
        NB. seed_0_10 does not change during their use.
        ```
B.1. MODELS

    seed_0_1=: TAP1 "1 s
    seed_3_1=: TAP2 "1 s
    seed_1_9=: 9 "1 s

NB.{{
    Xor 'b_64_1' ; 'seed_0_1' ; 'seed_3_1'  
    AssignField 'ns_65_9' ; 'seed_1_9'
    AssignField 'seed_0_9' ; 'ns_65_9'
    AssignField 'seed_9_1' ; 'b_64_1'
}}

NB. Update seed_0_10 to reflect the changes in its bits.  
    seed_0_10=: #. seed_9_1 ,"_2 seed_0_9

NB. Another labeling for some bits.  
NB. seed_0_10 does not change during its use.  
    seed_0_4=: LABELS | seed_0_10

NB.{{
    NB. Calculate current state energy
    AssignField 'a_14_10' ; 'y_74_8'
    for_l. i. LABELS do.
        SelectiveSubValue 'a_14_10' ; 'a_14_10' ; 'state_10_4' ; l ; (l { MU) ; 'Int'
    end.
    Mul 'e_82_20' ; 'a_14_10' ; 'a_14_10' ; 'Int'
    for_n. i. NBCOUNT do.
        FetchNeighbor 'a_14_10' ; 'state_10_4' ; (n { NBORS)
    end.
}}

NB. Alias a_14_4
NB. a_14_10 is overwritten after this.  
    a_14_4=: LABELS | a_14_10

NB.{{
    Xor 'a_14_4' ; 'state_10_4' ; 'a_14_4'
    SelectiveAddSubValue 'e_82_20' ; 'e_82_20' ; 'a_14_4' ; 0 ; BETA ; 'Int'
}}

NB. Calculate new state energy
    AssignField 'a_14_10' ; 'y_74_8'
    for_l. i. LABELS do.
        SelectiveSubValue 'a_14_10' ; 'a_14_10' ; 'seed_0_4' ; l ; (l { MU) ; 'Int'
    end.
    Mul 'ne_24_20' ; 'a_14_10' ; 'a_14_10' ; 'Int'
    for_n. i. NBCOUNT do.
        FetchNeighbor 'a_14_10' ; 'state_10_4' ; (n { NBORS)
    end.
}}

NB. Alias a_14_4
NB. a_14_10 is overwritten after this.  
    a_14_4=: LABELS | a_14_10

NB.{{
    Xor 'a_14_4' ; 'seed_0_4' ; 'a_14_4'
    SelectiveAddSubValue 'ne_24_20' ; 'ne_24_20' ; 'a_14_4' ; 0 ; BETA ; 'Int'
}}
APPENDIX B. LISTINGS

B.1.7 File: architecture21.ijs

This file was derived from file architecture15.ijs after the LFSR error was corrected in the corresponding files from the Trade-Off and Reorganization sub-phases.

annealing=: 3 : 0
  NB. Give all pixels a unique starting value.
  LoadFile 'seed_0_5' ; 'd:/leroy/trunk/quantization/seed31.dat' ; 'ro{pdtRdv,sdtRdv}'
  NB.}

  NB. Force seed_0_5 into the right shape.
  seed_0_5=: ($ y.) $ seed_0_5
  NB. Explicitly assign bits to seed_0_4
  seed_0_4=: LABELS | seed_0_5
  NB.}

  pixels=. y.
  NB.}

  NB. Give pixel value a name
  LoadField 'y_74_8' ; pixels ; 'ro{pdtRdv}'

  for_t. i. RUNS do.
  NB. Randomize new state
  NB. Get the bit representation so we can work
  NB. at bit level.
  s=. _5 ."1 #: seed_0_5

  NB. These assignments are used as an alias for these
  NB. bits so they can be used in Rts instructions.
B.1. MODELS

NB. seed_0_10 does not change during their use.

seed_0_1=: TAP1 "1 s
seed_2_1=: TAP2 "1 s
seed_1_4=: 4 {."1 s

NB.{{
Xor 'b_64_1'; 'seed_0_1'; 'seed_2_1'
AssignField 'ns_65_4'; 'seed_1_4'
AssignField 'seed_0_4'; 'ns_65_4'
AssignField 'seed_4_1'; 'b_64_1'
NB.

NB. Update seed_0_10 to reflect the changes in its bits.
seed_0_5=: #. seed_4_1 ,"_2 seed_0_4

NB. Another labeling for some bits.
NB. seed_0_10 does not change during its use.
seed_0_4=: LABELS | seed_0_5
NB.{{
NB. Calculate current state energy
AssignField 'a_14_10'; 'y_74_8'
for_l. i. LABELS do.
SelectiveSubValue 'a_14_10'; 'a_14_10'; 'state_10_4'; l; (l { MU); 'Int'
end.
Mul 'e_82_20'; 'a_14_10'; 'a_14_10'; 'Int'
for_n. i. NBCOUNT do.
FetchNeighbor 'a_14_10'; 'state_10_4'; (n { NBORS)
NB.

NB. Alias a_14_4
NB. a_14_10 is overwritten after this.
a_14_4=: LABELS | a_14_10

NB.{{
Xor 'a_14_4'; 'state_10_4'; 'a_14_4'
SelectiveAddSubValue 'e_82_20'; 'e_82_20'; 'a_14_4'; 0; BETA; 'Int'
end.

NB. Calculate new state energy
AssignField 'a_14_10'; 'y_74_8'
for_l. i. LABELS do.
SelectiveSubValue 'a_14_10'; 'a_14_10'; 'seed_0_4'; l; (l { MU); 'Int'
end.
Mul 'ne_24_20'; 'a_14_10'; 'a_14_10'; 'Int'
for_n. i. NBCOUNT do.
FetchNeighbor 'a_14_10'; 'state_10_4'; (n { NBORS)
NB.

NB. Alias a_14_4
NB. a_14_10 is overwritten after this.
a_14_4=: LABELS | a_14_10

NB.{{
Xor 'a_14_4'; 'seed_0_4'; 'a_14_4'
SelectiveAddSubValue 'ne_24_20'; 'ne_24_20'; 'a_14_4'; 0; BETA; 'Int'
end.
APPENDIX B. LISTINGS

B.1.8 File: templates2.ijs

This file contains the J code implementing the code templates.

```j
NB. Compare energies
Sub 'ne_24_20' ; 'ne_24_20' ; 'e_82_20' ; 'Int'
LEValue 'accept_6_1' ; 'ne_24_20' ; (t { ALPHAs) ; 'Int'
SelectiveUpdate 'state_10_4' ; 'seed_0_4' ; 'accept_6_1' ; 1
end.
NB. Debugging assignment
AddValue 'debug_32_32' ; 'state_10_4' ; 0 ; 'Card'
NB.}}{{
NB. The Linedancer produces two times the same result.
dump_32_32=: ,~ dump_32_32
NB.}}{{
NB. DumpFile 'state_10_4' ; '' ; 'ro{sdtRdv}'
DumpFile 'debug_32_32' ; 'd:\leroy\trunk\quantization\output.dump' ; 'ro{pdtRdv,sdtRdv'
NB.})
state_10_4
)
```

B.1.8 File: templates2.ijs

This file contains the J code implementing the code templates.

```j
NB. ===========================================================
NB. Constraints checking
NB. Check RTS operation calling conventions.
NB. Returns (offset,size) pairs for <dst>, <src> and <src2>.
NB. Missing field can be given as empty string ''. This is
NB. returned as 0.
RtsCheck=: 3 : 0
'dst src1 src2'=. y.
NB. Get (offset,size) pair for src1, src2 and dst.
if. src1 -: '' do.
us1=. 0
nls1=. 0
else.
us1=. Unpack src1
nls1=. 1
end.
if. src2 -: '' do.
us2=. 0
nls2=. 0
else.
us2=. Unpack src2
nls2=. 1
end.
ud=. Unpack dst
NB. Check for overlap
NB. Between src1 and dst
if. nls1 do.
NB. When result and operand overlap, they must share the same
NB. base address.
```
if. us1 Overlap ud do. assert. {. ud = us1 end.
end.

NB. Between src2 and dst
if. nls2 do.

NB. When result and operand overlap, they must share the same
NB. base address.
if. us2 Overlap ud do. assert. {. ud = us2 end.
end.

NB. Only one vector operand can come from EXT.
if. nls1 *. nls2 do.

assert. 2 > +/  64 < +/"1 us1 ,: us2
end.

ud ; us1 ; us2

NB. Get (offset, size) pair from register name
Unpack=: =&'_'".;._1 ]
NB. Offset lhs lies between rhs.
Between=: ([: >: {.@]) *. ([< +/@])
NB. Check if two (offset, size) pairs overlap.
Overlap=: (Between~ {.)+. (Between~ {.))~

NB. Check logical RTS operation calling conventions.
RtsLogicalCheck=: 3 : 0
RtsCheck y.

NB. The size of the vector(s) Operand1 and/or Operand2 must always
NB. match the size of the Result.
if. -. us1 -: 0 do. assert. 1 { ud = us1 end.
if. -. us2 -: 0 do. assert. 1 { ud = us2 end.
0

NB. Arithmetic operations

NB. Multiply field <src1> with field <src2> and put the result
NB. in <dst>.

NB. Activate(All,-,-,-),
NB. Write(Binary, 1, ab, 0),
NB. RTS(Mult, type, -, dst, src1, src2),
Mul=: 3 : 0
'dst src1 src2 type'=:. y.
'ud us1 us2'=:. RtsCheck dst ; src1 ; src2

NB. Result size must be equal to sum of size of operands.
assert. 1 { ud = +/- us1 ,: us2

m=: us1 Multiplicand us2
assert. 2 > +/- 64 < +/-"1 ud ,: m
Tests showed that we cannot do $A\_OUT \leftarrow A\_IN \times B$, where $A\_IN$ and $A\_OUT$ have the same base address and $\text{size}(A\_OUT) = \text{size}(A\_IN) + \text{size}(B)$.

Basically this means that the result cannot have the same base address as any of the operands. RtsCheck already takes care of the overlap situation.

```
assert. {. ud "=: us1
assert. {. ud "=: us2
```

Return the multiplicand of a multiplication as a pair $(o,s)$.

```
Multiplicand=: 1&{ @: < { ,:
```

Add value `<delta>` to field `<src>` and put the result in field `<dst>`.

```
AddValue=: 3 : 0
'dst src delta type'=. y.
RtsCheck dst ; src ; ''
". dst , "=: ', src , ' + delta''
```

Subtract field `<src2>` from field `<src1>` and put the result in field `<dst>`.

```
Sub=: 3 : 0
'dst src1 src2 type'=. y.
RtsCheck dst ; src1 ; src2
". dst , "=: ', src1 , ' - ', src2''
```

Subtract value `<delta>` from field `<src>` and put the result in field `<dst>`.

```
SubValue=: 3 : 0
'dst src delta type'=. y.
RtsCheck dst ; src ; ''
". dst , "=: ', src , ' - delta''
```

Perform XOR on field `<src1>` and field `<src2>` and put the result in `<dst>`.

```
```

This implementation does not actually perform a binary
NB. XOR, but only compares the values. This is enough for
NB. our purposes.

NB. Activate(All,-,-,-),
NB. Write(Binary, 1, ab, 0),
NB. RTS(Xor, Bitset, -, dst, src1, src2),
Xor=: 3 : 0
'y.' dst src1 src2'=. y.
'td us1 us2'=. RtsCheck dst ; src1 ; src2
RtsLogicalCheck ud ; us1 ; us2

". dst , '=: ' , src1 , ' ~: ' , src2
)

NB. Set field <dst> to 1 where field <src> <: value <val>,
NB. and to 0 everywhere else.

NB.
NB. Activate(All,-,-,-),
NB. Write(Binary, 1, ab, 0),
NB. RTS(Le, type, -, dst, src, val),
LEValue=: 3 : 0
'y.' dst src val type'=. y.
'td us1 us2'=. RtsCheck dst ; src ; ''

NB. <dst> must have size 1.
assert. 1 = 1 { ud

". dst , '=: ' , src , ' <: val'
)

NB. =============================================================

NB. Assignment operations

NB. Assign the contents of field <src> to field <dst>.
NB.
NB. Activate(All,-,-,-),
NB. Write(Binary, 1, ab, 0),
NB. RTS(Assign, Bitset, -, dst, src, -),
AssignField=: 3 : 0
'y.' dst src'=. y.
'td us1 us2'=. RtsCheck dst ; src ; ''

NB. Check if size of src <= size of dst.
NB. This is for convenience and not imposed by the architecture.
assert. 1 { ud >: us1

". dst , '=: ' , src
)

NB. Fetch field <src> of neighbor <nbor> and assign its value
NB. to field <dst>.
NB.
NB. Activate(All,-,-,-),
NB. Write(Binary, 1, ab, 0),
APPENDIX B. LISTINGS

195 NB. RTS(Assign, Bitset, nbor, dst, src, -),
196 FetchNeighbor=: 3 : 0
197 'dst src nbor'=. y.
198 'ud us1 us2'=. RtsCheck dst ; src ; ''
199 '. dst , '=': nbor |. .! .0 , src
200 )
201
203 NB. Selective operations
204
205 NB. Subtract value <delta> from field <src> where field <cond>
206 NB. equals value <val> and put the result in field <dst>.
207 NB.
208 NB. Activate(All,-,-,-),
209 NB. Write(Binary, 0, ab, 0),
210 NB. Tag(Binary, val, cond, tr1, -),
211 NB. Activate(Matching, -, tr1, -),
212 NB. Write(Binary, 1, ab, 0),
213 NB. RTS(Sub, type, -, dst, src, delta),
214 SelectiveSubValue=: 3 : 0
215 'dst src cond val delta type'=. y.
216 RtsCheck dst ; src ; ''
217 " 'eq=' , cond , '=' val'
218 neq=. -. eq
219 " . dst , '=': (eq * , src , '-' delta) + (neq * , src , ')''
220 )
222 NB. Subtract value <delta> from field <src> where field <cond>
223 NB. equals value <val> and add value <delta> to field <src>
224 NB. everywhere else.
225 NB.
228 NB. Activate(All,-,-,-),
229 NB. Write(Binary, 1, ab, 0),
230 NB. Write(Binary, 0, ab, 1),
231 NB. Tag(Binary, val, cond, tr1, -)
232 NB. Activate(Matching, -, tr1, -),
233 NB. Write(Binary, 1, ab, 1),
234 NB. RTS(AddSub, type, -, dst, src, delta),
235 SelectiveAddSubValue=: 3 : 0
236 'dst src cond val delta type'=. y.
237 RtsCheck dst ; src ; ''
238 " 'eq=' , cond , '=' val'
239 neq=. -. eq
240 " . dst , '=': (eq * , src , '-' delta) + (neq * , src , '+ delta)''
241 )
245 NB. Assign field <src> to field <dst> where field <cond> equals
246 NB. value <val>, keep the rest of field <dst> intact.
247 NB.
248 NB. Activate(All,-,-,-),
249 NB. Write(Binary, 0, ab, 0),
B.1. MODELS

<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>249</td>
<td>NB. Tag(Binary, val, cond, tr1, -),</td>
</tr>
<tr>
<td>250</td>
<td>NB. Activate(Matching, -, tr1, -),</td>
</tr>
<tr>
<td>251</td>
<td>NB. Write(Binary, 1, ab, 0),</td>
</tr>
<tr>
<td>252</td>
<td>NB. RTS(Assign, Bitset, -, dst, src, -),</td>
</tr>
<tr>
<td>253</td>
<td>SelectiveUpdate=: 3 : 0</td>
</tr>
<tr>
<td>254</td>
<td>'dst src cond val'=. y.</td>
</tr>
<tr>
<td>255</td>
<td>RtsCheck dst ; src ; ''</td>
</tr>
<tr>
<td>256</td>
<td>'. 'eq='. ', cond ', '= val'</td>
</tr>
<tr>
<td>257</td>
<td>neq=. -. eq</td>
</tr>
<tr>
<td>258</td>
<td>'.. dst ', '=:(eq * ', src ', ') + (neq * ', dst ', ')'</td>
</tr>
<tr>
<td>259</td>
<td>}</td>
</tr>
<tr>
<td>260</td>
<td>NB. =============================================================</td>
</tr>
<tr>
<td>261</td>
<td>NB. Data loading / dumping</td>
</tr>
<tr>
<td>262</td>
<td>NB. Set field &lt;dst&gt; to the values of variable &lt;src&gt;.</td>
</tr>
<tr>
<td>263</td>
<td>NB. Activate(All,-,-,-),</td>
</tr>
<tr>
<td>264</td>
<td>NB. pdTransfer(Load, dst, -, flags),</td>
</tr>
<tr>
<td>265</td>
<td>LoadField=: 3 : 0</td>
</tr>
<tr>
<td>266</td>
<td>'dst src flags'=. y.</td>
</tr>
<tr>
<td>267</td>
<td>'.. dst ', '=: src'</td>
</tr>
<tr>
<td>268</td>
<td>}</td>
</tr>
<tr>
<td>269</td>
<td>NB. Set field &lt;dst&gt; to the contents of file &lt;file&gt;.</td>
</tr>
<tr>
<td>270</td>
<td>NB. Activate(All,-,-,-),</td>
</tr>
<tr>
<td>271</td>
<td>NB. pdTransfer(Load, dst, -, flags),</td>
</tr>
<tr>
<td>272</td>
<td>LoadFile=: 3 : 0</td>
</tr>
<tr>
<td>273</td>
<td>'dst file flags'=. y.</td>
</tr>
<tr>
<td>274</td>
<td>'.. dst ', '=: _. '. (1!:1) &lt; jpath file'</td>
</tr>
<tr>
<td>275</td>
<td>}</td>
</tr>
<tr>
<td>276</td>
<td>NB. If &lt;file&gt; -: '', compare field &lt;src&gt; to the contents of</td>
</tr>
<tr>
<td>277</td>
<td>NB. file &lt;file&gt; and raise an assertion failure if there is</td>
</tr>
<tr>
<td>278</td>
<td>NB. a mitchmatch.</td>
</tr>
<tr>
<td>279</td>
<td>NB. Also assign the result of the comparison to the global</td>
</tr>
<tr>
<td>280</td>
<td>NB. variable DUMP.</td>
</tr>
<tr>
<td>281</td>
<td>NB.</td>
</tr>
<tr>
<td>282</td>
<td>NB. Activate(All,-,-,-),</td>
</tr>
<tr>
<td>283</td>
<td>NB. pdTransfer(Dump, src, -, flags),</td>
</tr>
<tr>
<td>284</td>
<td>DumpFile=: 3 : 0</td>
</tr>
<tr>
<td>285</td>
<td>'src file flags'=. y.</td>
</tr>
<tr>
<td>286</td>
<td>val=.. '. src</td>
</tr>
<tr>
<td>287</td>
<td>if. -. file -: '' do.</td>
</tr>
<tr>
<td>288</td>
<td>dat=.. '. (1!:1) &lt; jpath file</td>
</tr>
<tr>
<td>289</td>
<td>DATA=: dat=. ($ val) $ dat</td>
</tr>
<tr>
<td>290</td>
<td>DUMP=: chk= val = dat</td>
</tr>
<tr>
<td>291</td>
<td>assert. val == dat</td>
</tr>
<tr>
<td>292</td>
<td>end.</td>
</tr>
<tr>
<td>293</td>
<td>val</td>
</tr>
<tr>
<td>294</td>
<td>)</td>
</tr>
</tbody>
</table>
B.2 Linedancer Code

These files form the final implementation of the quantization algorithm on the Linedancer. Only the upper level thread functions have been included to just give an idea of how the implementation works.

B.2.1 File: algorithm.defs

This file contains declarations for variables used in the output generated by the translation process of the Translation subphase.

```c
/* Array parameters */
#define BLOCK_TO_PTR(x) sdsAddress(x, OWN_CHANNEL)

const uint8 *MU = (const uint8 *)BLOCK_TO_PTR(parms->ip_block_mu);
const uint16 *ALPHAs = (const uint16 *)BLOCK_TO_PTR(parms->ip_block_thres);
const int16 *NBORS = (const int16 *)BLOCK_TO_PTR(parms->ip_block_nbor);

/* Scalar parameters */
uint16 BETA = parms->ip_beta;
uint32 RUNS = parms->ip_stepcount;
uint8 LABELS = parms->ip_labelcount;
uint8 NBCOUNT = parms->ip_nborcount;

/* Loop variables */
int32 t, l, n;
/* Temporary variable for complicated values */
int32 temp;
```

B.2.2 File: algorithm.tpl

This is the file generated from architecture21.ijs in the Translation subphase.

```c
void
algorithm(struct ism_parms *parms)
{

/* Array parameters */
#define BLOCK_TO_PTR(x) sdsAddress(x, OWN_CHANNEL)

const uint8 *MU = (const uint8 *)BLOCK_TO_PTR(parms->ip_block_mu);
const uint16 *ALPHAs = (const uint16 *)BLOCK_TO_PTR(parms->ip_block_thres);
const int16 *NBORS = (const int16 *)BLOCK_TO_PTR(parms->ip_block_nbor);

/* Scalar parameters */
uint16 BETA = parms->ip_beta;
uint32 RUNS = parms->ip_stepcount;
uint8 LABELS = parms->ip_labelcount;
uint8 NBCOUNT = parms->ip_nborcount;

/* Loop variables */
int32 t, l, n;
/* Temporary variable for complicated values */
int32 temp;
```
aop{
    /* Linedancer initialization */
    Segmentation(ioChannel, -),
    /* Load */
    Activate(All, -,-,-),
    pdTransfer(Load, 5, 0, -, ro{pdtRdv,sdtRdv}),
    /* AssignField */
    Activate(All, -,-,-),
    Write(Binary, 1, ab, 0),
    RTS(Assign, Bitset, -, @{4,10}, @4, 0, -),
    /* Load */
    Activate(All, -,-,-),
    pdTransfer(Load, 8, 74, -, ro{pdtRdv}),
};

for (t = 0; t < RUNS; ++t) {
    aop{
        /* Xor */
        Activate(All, -,-,-),
        Write(Binary, 1, ab, 0),
        RTS(Xor, Bitset, -, @{1,64}, @4, 0, @1,2),
        /* AssignField */
        Activate(All, -,-,-),
        Write(Binary, 1, ab, 0),
        RTS(Assign, Bitset, -, @{4,65}, @4, 1, -),
        /* AssignField */
        Activate(All, -,-,-),
        Write(Binary, 1, ab, 0),
        RTS(Assign, Bitset, -, @{4,0}, @4, 65, -),
        /* AssignField */
        Activate(All, -,-,-),
        Write(Binary, 1, ab, 0),
        RTS(Assign, Bitset, -, @{1,4}, @1, 64, -),
        /* AssignField */
        Activate(All, -,-,-),
        Write(Binary, 1, ab, 0),
        RTS(Assign, Bitset, -, @{10,14}, @8, 74, -),
    }
    for (l = 0; l < LABELS; ++l) {
        aop{
            /* SelectiveSubValue */
            Activate(All, -,-,-),
            Write(Binary, 0, ab, 0),
            Tag(Binary, l, 4, 10, tr1, -),
            Activate(Matching, - tr1, -),
            Write(Binary, 1, ab, 0),
        }
        temp = MU[l];
        aop{
            RTS(Sub, Int, -, @{10,14}, @10, 14, temp),
        }
    }
    aop{
        /* Mul */
APPENDIX B. LISTINGS

75  Activate(All,-,-,-),
76  Write(Binary, 1, ab, 0),
77  RTS(Mult, Int, -, @20,82, @10,14, @10,14)),
78  }
79  for (n = 0; n < NBCOUNT; ++n) {
80    aop{
81      /* FetchNeighbor */
82      Activate(All,-,-,-),
83      Write(Binary, 1, ab, 0),
84    }
85    temp = NBORS[n];
86    aop{
87      RTS(Assign, Bitset, co{Get,temp}, @10,14, @4,10, -),
88      /* Xor */
89      Activate(All,-,-,-),
90      Write(Binary, 1, ab, 0),
91      RTS(Xor, Bitset, -, @4,14, @4,10, @4,14)),
92      /* SelectiveAddSubValue */
93      Activate(All,-,-,-),
94      Write(Binary, 1, ab, 0),
95      Write(Binary, 0, ab, 1),
96      Tag(Binary, 0, 4, 14, tr1, -),
97      Activate(Matching, -, tr1, -),
98      Write(Binary, 1, ab, 1),
99      RTS(AddSub, Int, -, @20,82, @20,82, BETA),
100    }
101  }
102  aop{
103    /* AssignField */
104    Activate(All,-,-,-),
105    Write(Binary, 1, ab, 0),
106    RTS(Assign, Bitset, -, @10,14, @8,74, -),
107  }
108  for (l = 0; l < LABELS; ++l) {
109    aop{
110      /* SelectiveSubValue */
111      Activate(All,-,-,-),
112      Write(Binary, 0, ab, 0),
113      Tag(Binary, l, 4, 0, tr1, -),
114      Activate(Matching, -, tr1, -),
115      Write(Binary, 1, ab, 0),
116    }
117    temp = MU[l];
118    aop{
119      RTS(Sub, Int, -, @10,14, @10,14, temp),
120    }
121  }
122  aop{
123    /* Mul */
124    Activate(All,-,-,-),
125    Write(Binary, 1, ab, 0),
126    RTS(Mult, Int, -, @20,24, @10,14, @10,14)),
127  }
128  for (n = 0; n < NBCOUNT; ++n) {
B.2. LINEDANCER CODE

```plaintext
aop{
    /* FetchNeighbor */
    Activate(All,-,-,-),
    Write(Binary, 1, ab, 0),
};
temp = NBORS[n];
aop{
    RTS(Assign, Bitset, co{Get,temp}, @\{10,14\}, @\{4,10\}, -),
    /* Xor */
    Activate(All,-,-,-),
    Write(Binary, 1, ab, 0),
    RTS(Xor, Bitset, -, @\{4,14\}, @\{4,0\}, @\{4,14\}),
    /* SelectiveAddSubValue */
    Activate(All,-,-,-),
    Write(Binary, 1, ab, 0),
    Write(Binary, 0, ab, 1),
    Tag(Binary, 0, 4, 14, tr1, -),
    Activate(Matching, -, tr1, -),
    Write(Binary, 1, ab, 0),
    RTS(AddSub, Int, -, @\{20,24\}, @\{20,24\}, BETA),
};
}
aop{
    /* Sub */
    Activate(All,-,-,-),
    Write(Binary, 1, ab, 0),
    RTS(Sub, Int, -, @\{20,24\}, @\{20,24\}, @\{20,82\}),
    /* LEValue */
    Activate(All,-,-,-),
    Write(Binary, 1, ab, 0),
};
temp = ALPHAs[t];
aop{
    RTS(le, Int, -, @\{1,6\}, @\{20,24\}, temp),
    /* SelectiveUpdate */
    Activate(All,-,-,-),
    Write(Binary, 0, ab, 0),
    Tag(Binary, 1, 1, 6, tr1, -),
    Activate(Matching, -, tr1, -),
    Write(Binary, 1, ab, 0),
    RTS(Assign, Bitset, -, @\{4,10\}, @\{4,0\}, -),
};
}
aop{
    /* AddValue */
    Activate(All,-,-,-),
    Write(Binary, 1, ab, 0),
    RTS(Add, Card, -, @\{32,32\}, @\{4,10\}, 0),
    /* Dump */
    pdTransfer(Dump, 4, 10, -, ro{sdtRdv}),
    /* Dump */
    Activate(All,-,-,-),
    pdTransfer(Dump, 32, 32, -, ro{pdtRdv,sdtRdv}),
```
B.2.3 File: ismCode.tpl

This is the driver code for the ISM thread of the Linedancer for the quantization algorithm. The generated code from algorithm.tpl is included in this file and the function declared in that file is called.

```c
aop(InsertImports);

#include "algorithm.tpl"

void
ismMain(void)
{
    struct ism_parms ismparms;
    mqId_t sccq, dsmq;
    int i, ntiles;
    setup_queues(&sccq, &dsmq);
    mqGet(sccq, &ismparms, WAIT_FOREVER);
    mqGet(dsmq, &ntiles, WAIT_FOREVER);
    for (i = 0; i < ntiles; ++i) {
        algorithm(&ismparms);
    }
    thrExit(0);
}
```

B.2.4 File: dsmCode.c

This is the driver code for the DSM thread of the Linedancer for the quantization algorithm. It takes care of splitting the image into separate tiles and sending them to the ISM thread.

```c
void
dsmMain(void)
{
    mqId_t sccq, ismq;
    uint32 dsmpresult = 0;
    int i, ntiles;
    sdmcParameters_t sdmc;
    struct tile_info ti;
    setup_queues(&sccq, &ismq);
    mqGet(sccq, &dsmparms, WAIT_FOREVER);
    tile_init(&ti,
        dsmparms.dp_width,
        dsmparms.dp_height,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmparms.dp_border,
        dsmpars...
B.2. LINEDANCER CODE

PDT_RDV | SDT_RDV | D8 | JUMP1 | PACK1);
ntiles = tile_count(&ti);

mqPut(ismq, &ntiles, WAIT_FOREVER);
for (i = 0; i < ntiles; ++i) {
    upload_seed(dsmparms.dp_block_seed);
    setup_tile_transfer(&ti, dsmparms.dp_block_in, PDS_LOAD, &sdmc);
    sdmcQueueTransfer(OWN_CHANNEL, &sdmc, NULL);
    setup_tile_transfer(&ti, dsmparms.dp_block_out, PDS_DUMP, &sdmc);
    sdmcQueueTransfer(OWN_CHANNEL, &sdmc, NULL);
    download_debug(dsmparms.dp_block_dbg);
}
tile_next(&ti);

sdmcWait(OWN_CHANNEL);
mqPut(sccq, &dsmresult, WAIT_FOREVER);
thrExit(0);

B.2.5 File: sccCode.tpl

This is the driver code for the SCC thread of the Linedancer. It takes care of reading parameters for the algorithm, and precalculation and rounding of parameters.

int main(int argc, char **argv)
{
    mqId_t dsmq[MAX_SDT_CHANNELS];
    mqId_t ismq[MAX_SDT_CHANNELS];
    struct ism_parms ismparms;
    struct dsm_parms dsmparms;
    ld_init();

    attach_queues(dsmq, ismq);
    load_config(&dsmparms, &ismparms);
    parse_arguments(argc, argv, &dsmparms, &ismparms);
    dsmparms.dp_block_seed = upload_seed();
    upload_image(inputfile,
        &dsmparms.dp_width,
        &dsmparms.dp_height,
        &dsmparms.dp_chsize,
        &dsmparms.dp_block_in,
        &dsmparms.dp_block_out);
    dsmparms.dp_block_dbg = alloc_debug();
APPENDIX B. LISTINGS

send_parameters(dsmq, &dsmparms, ismq, &ismparms);

wait_for_ld(dsmq);

if (outputfile != NULL) {
    download_image(outputfile, 
        dsmparms.dp_width, 
        dsmparms.dp_chsize, 
        dsmparms.dp_block_out);
}

download_dbg("output.dump", dsmparms.dp_block_dbg);

destroy_sds_buffers(&dsmparms, &ismparms);

ld_quit();

return 0;

B.3 Python Code

This is the Python code implementing the translator used in the Translation
subphase. It uses Yapps to generate a parser.

B.3.1 Parser

This is the grammar file of the parser. The driver code and initialization of
variables has been removed for space considerations. The top part implements
the emitting of Linedancer code for each code template, while the grammar
itself is contained in the bottom half.

B.3.2 File: Parser3.g

# Representation of memory field.
class Field:
    def __init__(self, offset, size):
        self.offset = offset
        self.size = size

# String representation of a value.
# Values can be complicated, which means they cannot occur in aop operations
# directly. We work around this by assigning them to a temporary variable first.
class Value:
    def __init__(self, complicated, strrep):
        self.complicated = complicated
        self.strrep = strrep

# Return string representation for a field (offset, size) pair.
# If offset = 100x it is taken as a ab register with size x.
def FieldStr(f):
    if f.offset >= 1000:
B.3. PYTHON CODE

```
return 'ab,' + str(f.offset - 1000)
else:
    return str(f.size) + ', ' + str(f.offset)

# Return string representation for a field (offset, size) pair,
# in a RTS command context.
# If offset = 100x it is taken as a ab register with size x.
def RtsFieldStr(f):
    if f.offset >= 1000:
        return '@{ab,' + str(f.offset - 1000) + '}'
    else:
        return '@{' + str(f.size) + ',' + str(f.offset) + '}'

# Emit a line at the current indentation level.
def Emit(s):
    global EmitIndent, Output
    Output = Output + EmitIndent * '	' + s + '

# Emit the opening of the generated function.
def EmitOpen():
    global FuncName, FuncType, FuncArgs, FuncDefs
    Emit(FuncType)
    Emit(FuncName + '(' + FuncArgs + ')')
    Emit('{')
    IndentOpen()
    Emit(FuncDefs)
    EmitAopOpen()
    Emit('/* Linedancer initialization */')
    Emit('Segmentation(ioChannel, -),')

# Emit the closing of the generated function.
def EmitClose():
    EmitAopClose()
    IndentClose()
    Emit('}')

# Open an aop{ ... }; context if we are not already in one.
def EmitAopOpen():
    global InAop
    if not InAop:
        Emit('aop{')
        IndentOpen()
        InAop = True

# Close an aop{ ... }; context if we are already in one.
def EmitAopClose():
    global InAop
    if InAop:
        IndentClose()
        Emit('};')
```
InAop = False

# Open a new indentation level.
def IndentOpen():
    global EmitIndent
    EmitIndent = EmitIndent + 1

# Close current indentation level.
def IndentClose():
    global EmitIndent
    EmitIndent = EmitIndent - 1

# Emit the opening of a FOR loop.
def EmitForOpen(loopvar, count):
    EmitAopClose()
    Emit('for (' + loopvar + ' = 0; ' + loopvar + ' < ' + count + '; ++' + loopvar + ') {'
    IndentOpen()

# Emit the closing of a FOR loop.
def EmitForClose():
    EmitAopClose()
    IndentClose()
    Emit('}')

# Get the final string representation of value <val> for use in Aop
# instructions.
# Assignment to a temporary variable for complicated values is emitted if
# needed.
def AopRep(val):
    if val.complicated:
        v = ComplicatedTemp
        EmitAopClose()
        Emit(v + ' = ' + val.strrep + ';')
    else:
        v = val.strrep
        EmitAopOpen()
    return v

# Emit a write of value <val> to field <f> for all PEs, taking into account the
# difference between CAM and EXT memory.
def EmitWriteAll(val, f):
    v = EmitVal(val)
    Emit('Activate(All, -, -, -),')
    if f.offset >= 64:
        Emit('Write(Binary, 1, ab, 0),')
        Emit('RTS(Assign, Bitset, -, ' + RtsFieldStr(f) + ', ' + v + ', -),')
    else:
        Emit('Write(Binary, ' + v + ', ' + FieldStr(f) + '),')

# Emit a write of value <val> to field <f> for only the PEs matching tr1,
# taking into account the difference between CAM and EXT memory.
def EmitWriteMatching(val, f):
v = EmitVal(val)

if f.offset >= 64:
    Emit("Activate(All, -", -, ")
    Emit("Write(Binary, 0, ab, 0),")
    Emit("Activate(Matching, -", tr1, ")
    Emit("Write(Binary, 1, ab, 0),")
    Emit("RTS(Assign, Bitset, ", " + RtsFieldStr(f) + ",", " + v +", ",")
else:
    Emit("Activate(Matching, -, tr1, ")
    Emit("Write(Binary, ", " + v +", " + FieldStr(f) + ",")

##
## Emit functions for code templates.
##

def EmitMul(dst, src1, src2, type):
    EmitAopOpen()
    Emit("/* Mul */")
    Emit("Activate(All,-,-,-),")
    Emit("Write(Binary, 1, ab, 0),")
    Emit("RTS(Mult, " + type + \
          ","," + RtsFieldStr(dst) + \
          ","," + RtsFieldStr(src1) + \
          ","," + RtsFieldStr(src2) + \
          "),")

def EmitAddValue(dst, src, delta, type):
    EmitAopOpen()
    Emit("/* AddValue */")
    Emit("Activate(All,-,-,-),")
    Emit("Write(Binary, 1, ab, 0),")
    rep = AopRep(delta)
    Emit("RTS(Add, " + type + \
          ",","," + RtsFieldStr(dst) + \
          ",","," + RtsFieldStr(src) + \
          ",","," + rep + \
          "),")

def EmitSub(dst, src1, src2, type):
    EmitAopOpen()
    Emit("/* Sub */")
    Emit("Activate(All,-,-,-),")
    Emit("Write(Binary, 1, ab, 0),")
    Emit("RTS(Sub, " + type + \
          ",","," + RtsFieldStr(dst) + \
          ",","," + RtsFieldStr(src1) + \
          ",","," + RtsFieldStr(src2) + \
          "),")

def EmitSubValue(dst, src, delta, type):
    EmitAopOpen()
    Emit("/* SubValue */")
    Emit("Activate(All,-,-,-),")
    Emit("Write(Binary, 1, ab, 0),")
    rep = AopRep(delta)
def EmitXor(dst, src1, src2):
    EmitAopOpen()
    Emit('/* Xor */')
    Emit('Activate(All,-,-,-),')
    Emit('Write(Binary, 1, ab, 0),')
    Emit('RTS(Xor, Bitset' +
          ', -, ' + RtsFieldStr(dst) +
          ', ' + RtsFieldStr(src1) +
          ', ' + RtsFieldStr(src2) +
          '),')

def EmitLEValue(dst, src, val, type):
    EmitAopOpen()
    Emit('/* LEValue */')
    Emit('Activate(All,-,-,-),')
    Emit('Write(Binary, 1, ab, 0),')
    rep = AopRep(val)
    Emit('RTS(le, ' + type +
          ', -, ' + RtsFieldStr(dst) +
          ', ' + RtsFieldStr(src) +
          ', ' + rep +
          '),')

def EmitAssignField(dst, src):
    EmitAopOpen()
    Emit('/* AssignField */')
    Emit('Activate(All,-,-,-),')
    Emit('Write(Binary, 1, ab, 0),')
    Emit('RTS(Assign, Bitset' +
          ', -, ' + RtsFieldStr(dst) +
          ', ' + RtsFieldStr(src) +
          '),')

def EmitAssignValue(dst, val):
    EmitAopOpen()
    Emit('/* AssignField */')
    Emit('Activate(All,-,-,-),')
    Emit('Write(Binary, 1, ab, 0),')
    rep = AopRep(val)
    Emit('RTS(Assign, Bitset' +
          ', -, ' + RtsFieldStr(dst) +
          ', ' + rep +
          '),')

def EmitFetchNeighbor(dst, src, nbor):
    EmitAopOpen()
    Emit('/* FetchNeighbor */')
    Emit('Activate(All,-,-,-),')
B.3. PYTHON CODE

```python
Emit('Write(Binary, 1, ab, 0),')
rep = AopRep(nbor)
Emit('RTS(Assign, Bitset, co:Get,' + rep + ' },
   ', '+ RtsFieldStr(dst) 
   ', '+ RtsFieldStr(src) 
   ', '-', ')')

def EmitSelectiveSubValue(dst, src, cond, val, delta, type):
    EmitAopOpen()
    Emit('/* SelectiveSubValue */')
    Emit('Activate(All,-,-,-),')
    Emit('Write(Binary, 0, ab, 0),')
    rep = AopRep(val)
    Emit('Tag(Binary, ' + rep + ', ' + FieldStr(cond) + ', tr1, -),')
    Emit('Activate(Matching, -, tr1, -),')
    Emit('Write(Binary, 1, ab, 0),')
    rep = AopRep(delta)
    Emit('RTS(Sub, ' + type + ' +
   ', '-', '+ RtsFieldStr(dst) 
   ', '+ RtsFieldStr(src) 
   ', '+ rep +
   ') ,')

def EmitSelectiveAddSubValue(dst, src, cond, val, delta, type):
    EmitAopOpen()
    Emit('/* SelectiveAddSubValue */')
    Emit('Activate(All,-,-,-),')
    Emit('Write(Binary, 1, ab, 0),')
    Emit('Write(Binary, 0, ab, 1),')
    rep = AopRep(val)
    Emit('Tag(Binary, ' + rep + ', ' + FieldStr(cond) + ', tr1, -),')
    Emit('Activate(Matching, -, tr1, -),')
    Emit('Write(Binary, 1, ab, 1),')
    rep = AopRep(delta)
    Emit('RTS(AddSub, ' + type + ' +
   ', '-', '+ RtsFieldStr(dst) 
   ', '+ RtsFieldStr(src) 
   ', '+ rep +
   ') ,')

def EmitSelectiveUpdate(dst, src, cond, val):
    EmitAopOpen()
    Emit('/* SelectiveUpdate */')
    Emit('Activate(All,-,-,-),')
    Emit('Write(Binary, 0, ab, 0),')
    rep = AopRep(val)
    Emit('Tag(Binary, ' + rep + ', ' + FieldStr(cond) + ', tr1, -),')
    Emit('Activate(Matching, -, tr1, -),')
    Emit('Write(Binary, 1, ab, 0),')
    Emit('RTS(Assign, Bitset, -, ' + RtsFieldStr(dst) + 
```
def EmitLoad(dst, flags):
    EmitAopOpen()
    Emit('/* Load */
        Activate(All,-,-,-),
        pdTransfer(Load, ' + FieldStr(dst) + \
        ',-, ' + flags + \
        '),')

def EmitDump(src, flags):
    EmitAopOpen()
    Emit('/* Dump */
        Activate(All,-,-,-),
        pdTransfer(Dump, ' + FieldStr(src) + \
        ',-, ' + flags + \
        '),')

##
## The parser
##

parser Linedancer:
    token END: '$'
    token NAME: '[a-zA-Z][a-zA-Z0-9]*'
    token NUM: '[0-9]+'
    token ID: '[a-zA-Z][a-zA-Z0-9-]*'
    token STR: '[^\"]*'
    ignore: 'NB\..*
    ignore: '\[ \t\]'++

rule program: {{ EmitOpen() }} command* {{ EmitClose() }} END {{ return Output }}

rule command: 'Mul' field {{ dst = field }}
    ';' field {{ src1 = field }}
    ';' field {{ src2 = field }}
    ';' string {{ type = string }}
    {{ EmitMul(dst, src1, src2, type) }}
    | 'AddValue' field {{ dst = field }}
    ';' field {{ src = field }}
    ';' expr {{ delta = expr }}
    ';' string {{ type = string }}
    {{ EmitAddValue(dst, src, delta, type) }}
    | 'Sub' field {{ dst = field }}
    ';' field {{ src1 = field }}
    ';' field {{ src2 = field }}
    ';' string {{ type = string }}
    {{ EmitSub(dst, src1, src2, type) }}
    | 'SubValue' field {{ dst = field }}
    ';' field {{ src = field }}
B.3. PYTHON CODE

```python
';' expr {{ delta = expr }}
';' string {{ type = string }}
{{ EmitSubValue(dst, src, delta, type) }}
| 'Xor' field {{ dst = field }}
';' field {{ src1 = field }}
';' field {{ src2 = field }}
{{ EmitXor(dst, src1, src2) }}
| 'LEValue' field {{ dst = field }}
';' field {{ src = field }}
';' expr {{ val = expr }}
';' string {{ type = string }}
{{ EmitLEValue(dst, src, val, type) }}
| 'AssignField' field {{ dst = field }}
';' field {{ src = field }}
{{ EmitAssignField(dst, src) }}
| 'AssignValue' field {{ dst = field }}
';' expr {{ val = expr }}
{{ EmitAssignValue(dst, val) }}
| 'FetchNeighbor' field {{ dst = field }}
';' field {{ src = field }}
';' expr {{ nbor = expr }}
{{ EmitFetchNeighbor(dst, src, nbor) }}
| 'SelectiveSubValue' field {{ dst = field }}
';' field {{ src = field }}
';' field {{ cond = field }}
';' expr {{ val = expr }}
';' expr {{ delta = expr }}
';' string {{ type = string }}
{{ EmitSelectiveSubValue(dst, src, cond, val, delta, type) }}
| 'SelectiveAddSubValue' field {{ dst = field }}
';' field {{ src = field }}
';' field {{ cond = field }}
';' expr {{ val = expr }}
';' expr {{ delta = expr }}
';' string {{ type = string }}
{{ EmitSelectiveAddSubValue(dst, src, cond, val, delta, type) }}
| 'SelectiveUpdate' field {{ dst = field }}
';' field {{ src = field }}
';' field {{ cond = field }}
';' expr {{ val = expr }}
{{ EmitSelectiveUpdate(dst, src, cond, val) }}
| 'LoadField' field {{ dst = field }}
';' expr
';' string {{ flags = string }}
{{ EmitLoad(dst, flags) }}
| 'LoadFile' field {{ dst = field }}
';' string
';' string {{ flags = string }}
{{ EmitLoad(dst, flags) }}
| 'DumpFile' field {{ src = field }}
';' string
';' string {{ flags = string }}
{{ EmitDump(src, flags) }}
| for_loop
```
rule for_loop: 'for_'
  ID {{ loopvar = ID }}
  'i.' expr
  'do.' {{ EmitForOpen(loopvar, expr.strrep) }}
  command*
  'end.' {{ EmitForClose() }}

rule expr: operand {{ val = operand }}
  [ '
i.' expr
  ] {{ return val }}
  [ 'i.' expr
  ] {{ return val }}
  [ 'i.' expr
  ] {{ return val }}

rule operand: NUM {{ return Value(False, NUM) }}
  ID {{ return Value(False, ID) }}
  'i.' expr {{ return expr }}

rule field: 'i.' NAME 'i.'
  NUM {{ offset = int(NUM) }}
  'i.'
  NUM {{ size = int(NUM) }}
  'i.'
  {{ return Field(offset, size) }}

rule string: 'i.' {{ s = '' }}
  STR {{ s = STR }}
  'i.'
  {{ return s }}

%%