Master's Thesis

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 displ
2.3 Probabilities $P(y_s|g_s)$ 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 distri
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 $\mathcal{C}$ of cliq
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.
3.4 Increasing portability by using common code templates. Algorithms are implemented in term

4.1 Expanding a mesh (a) requires more Processor Elements (PEs) than expanding a string (b).
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)
4.5 Dependency of the number of cycles on the operand width for the remote RTS(Assign) instru

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

6.1 Comparison of quantizations by (a) a commercial program (Irfanview) and (b) our algorithm
6.2 Tiling artifact in the output of the quantization algorithm. If you look close inside the red re
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 ac
6.2 Processing time of Data Stream Manager (DSM) and Instruction Stream Manager (ISM) threads per tile
6.3 Processing time of DSM and ISM threads per tile when ten Simulated Annealing loops are performed. 5
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.
μ	Mean value.
μ_{g_s}	Mean value of input image pixels with label g_s.
σ	Standard deviation.
σ_{g_s}	Standard deviation of input image pixels with label g_s.
y	Input image.
y, 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

ALU Arithmetic & Logic Unit: part of a computer processor performing arithmetic and logical functions.

APL A Programming Language: programming language developed by K.E. Iverson, predecessor of J.

CADTES Computer Architecture Design & 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.

CAM Content Addressable Memory: PE memory addressable by its contents instead of a fixed address.

DSM Data Stream Manager: Linedancer thread responsible for transferring data between SDS and PDS.

EXT Extended Memory: Non-content addressable part of the PE memory (cf. CAM).

FPGA Field Programmable Gate Array: Hardware which is reprogrammable at gate level.

GPP General Purpose Processor: A processor which can be used for arbitrary tasks, such as a Pentium.

IDSM Instruction and Data Stream Manager: Processor on which the DSM and ISM threads run.

ISM Instruction Stream Manager: Linedancer thread responsible for controlling the processor array.

LFSR Linear Feedback Shift Register: Shift register which can be used as a semi-random number generator.

MIMD Multiple Instruction Multiple Data: Parallel computing, where multiple instructions operate on vectors of data in parallel.

MISD Multiple Instruction Single Data: Parallel computing, where multiple instructions operate on a single piece of data.

MMD Modified Metropolis Dynamics: parallel optimization of Simulated Annealing.

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 Océ 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 32
4.2.3 Associativity 34
4.2.4 Communication 34

4.2.5 Segmentation 34
4.2.6 Arithmetical and Logical Operations 36
4.3 Programmer's Interface 36
4.3.1 Software Tools 36
4.3.2 Threads of Execution 36
4.3.3 Libraries 37
4.4 Performance 37
4.4.1 Constant Time Instructions 37
4.4.2 Linear Time Instructions 37
4.4.3 Slow Instructions 38

5 Implementation **40**
5.1 Preparation 40
5.2 Incremental Prototyping 41
5.3 Transformational Development 41
5.3.1 Trade-Off Subphase 41
5.3.2 Reorganization Subphase 44
5.3.3 Template Subphase 46
5.3.4 Translation Subphase 49

6 Evaluation **51**
6.1 Quantization 51
6.1.1 Algorithm Load Distribution 51
6.1.2 Algorithm Output Quality 51
6.1.3 Performance Analysis 52
6.1.4 Speedup 54
6.2 Methodology 56
6.2.1 Design and Development 56
6.2.2 The J Language 56
6.3 Hardware 56
6.4 Possible Enhancements 57
6.4.1 Algorithm 57
6.4.2 Hardware 58

7 Conclusions and Recommendations **59**
7.1 Mapping and Methodology 59
7.2 Quantization 59
7.3 Linedancer 60

A Introduction to J **63**
A.1 Comments 63
A.2 Verbs and Nouns 63
A.2.1 Verbs 63
A.2.2 Nouns 64
A.3 Assignment 64
A.4 Arithmetic and Comparison 64
A.5 Array Operations 65
A.6 Miscellaneous Verbs 66

B Listings **67**
B.1 Models 67
B.1.1 File: `functional.ijs` 67
B.1.2 File: `functional4.ijs` 68
B.1.3 File: `functional6.ijs` 69
B.1.4 File: `algorithm6.ijs` 69
B.1.5 File: `algorithm9.ijs` 71
B.1.6 File: `architecture15.ijs` 72
B.1.7 File: `architecture21.ijs` 74
B.1.8 File: `templates2.ijs` 76
B.2 Linedancer Code 82
B.2.1 File: `algorithm.defs` 82
B.2.2 File: `algorithm.tpl` 82
B.2.3 File: `ismCode.tpl` 86
B.2.4 File: `dsmCode.c` 86
B.2.5 File: `sccCode.tpl` 87
B.3 Python Code 88
B.3.1 Parser 88
B.3.2 File: `Parser3.g` 88

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 (MRF) image model, onto the Linedancer.

We make this research question more concrete with the following subquestions:

- Which problems do we face in mapping the algorithm and how do we face them?
- How does the Evolutionary Design methodology help us during development?
- How does the Linedancer compare with a General Purpose Processor?
- How does the quality of our quantization algorithm compare with established algorithms?

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 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.

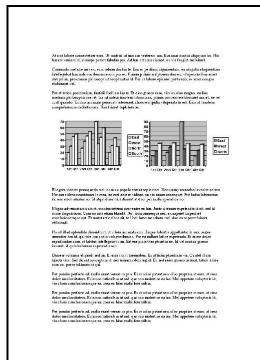


Figure 2.1: Typical office scan: text and charts.
[Typical office scan]

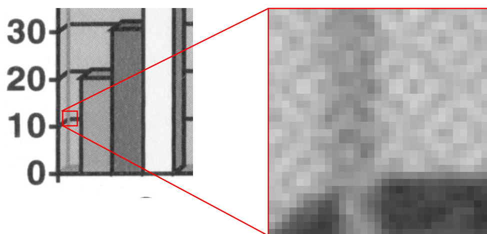


Figure 2.2: Image degradation: instead of just a couple of solid gray values, the image is blurry and displays some sort of patchwork pattern. The original image consisted of only 10 different grays, but scanning has increased this to 229.

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 (μ and σ are the mean and standard deviation of the distribution respectively):

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) \mathrm{d}x, \tag{2.1}$$

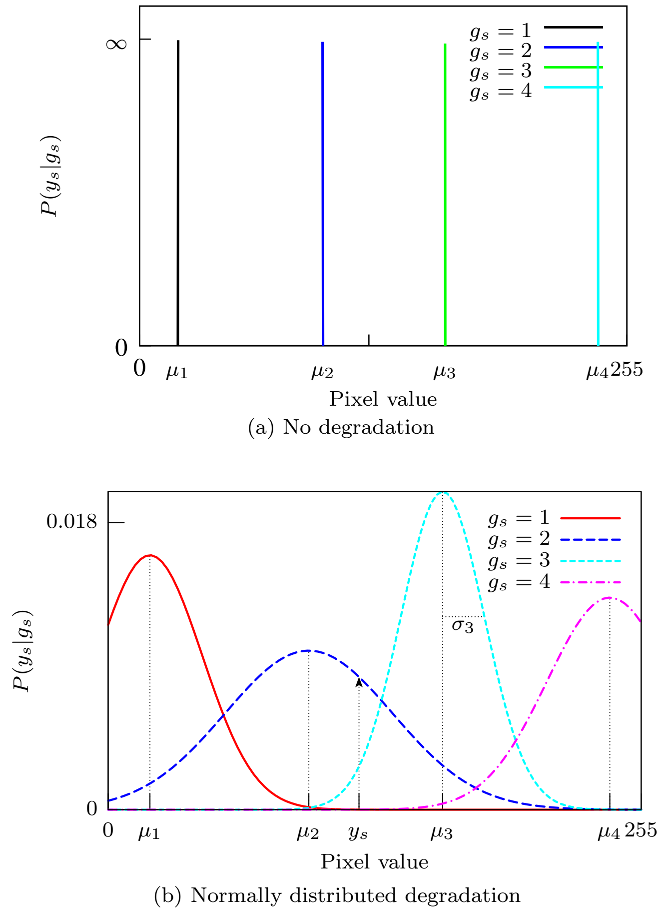
If we know the image pixel value means and standard deviations for each class we can calculate the probability of a pixel s with value $y_s \in [a, b)$ belonging to a class g_s using

$$P(a \leq y_s < b|g_s) = \int_a^b \frac{1}{\sqrt{2\pi}\sigma_{g_s}} \exp\left(\frac{-(y_s-\mu_{g_s})^2}{2{\sigma_{g_s}}^2}\right) \mathrm{d}y_s, \tag{2.2}$$

where μ_{g_s} and σ_{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):

$$P(y_s|g_s) = \frac{1}{\sqrt{2\pi}\sigma_{g_s}} \exp\left(\frac{-(y_s-\mu_{g_s})^2}{2{\sigma_{g_s}}^2}\right). \tag{2.3}$$



(a) No degradation

(b) Normally distributed degradation

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

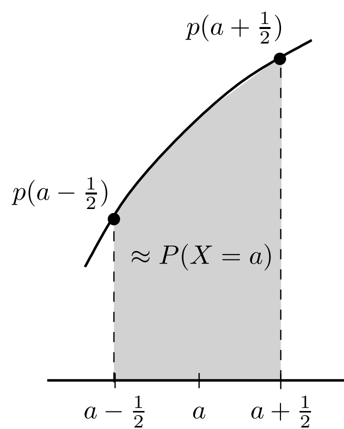
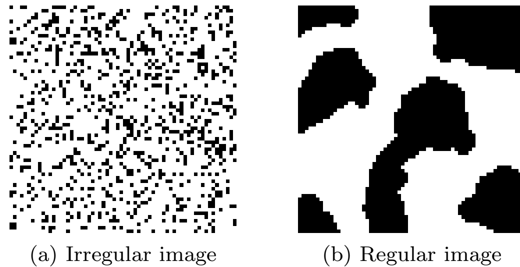


Figure 2.4: Approximation of a discrete probability $P(X = a)$ based on the continuous probability distribution $p(x)$.



(a) Irregular image (b) Regular image

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

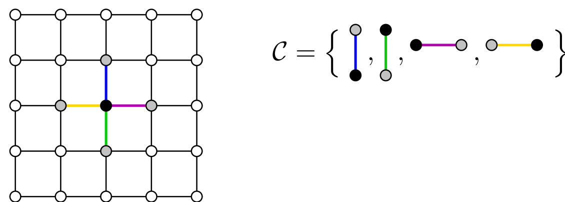


Figure 2.6: Image pixel (the black dot) with its neighborhood (gray dots) and corresponding set $\mathcal{C}$ of clique pairs. Each clique has its own color.

We assume that the relation between input image and quantized image only affects corresponding pixels, which means that [10]

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

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 $\mathcal{C}$ (Figure 2.6).

It can be proved [17] 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 [17] guarantees that we can write this

joint probability in the form

$$P(g) = \frac{1}{Z} \exp\left(\frac{-E(g)}{\hat{T}}\right), \tag{2.5}$$

where $E(g)$ encodes the local characteristics. $\hat{T}$ 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*,

$$E(g) = \sum_{C \in \mathcal{C}} V_C(g), \tag{2.6}$$

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:

$$V_C(g) = V_{(s,r)}(g) = \delta(g_s, g_r). \tag{2.7}$$

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:

$$P(g) = \frac{1}{Z} \exp\left(-\frac{1}{\hat{T}} \sum_{(s,r) \in \mathcal{C}_2} \delta(g_s, g_r)\right). \tag{2.8}$$

Here, $\mathcal{C}_2$ is the set of all pair cliques.

2.1.3 Bayesian Model

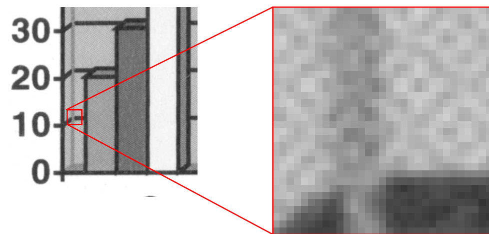
To give an idea of the terms fidelity and regularity, Figure 2.7 shows two cases. We can interpret the (normalized) fidelity and regularity as probabilities: let $P(y|g)$ be the fidelity (i.e. the probability of observing input image y when g is the quantized image), and $P(g)$ be the regularity (i.e. the probability of g being a satisfying quantized image). Using Bayes law [15]

$$P(g|y) = \frac{P(y|g)P(g)}{P(y)}, \tag{2.9}$$

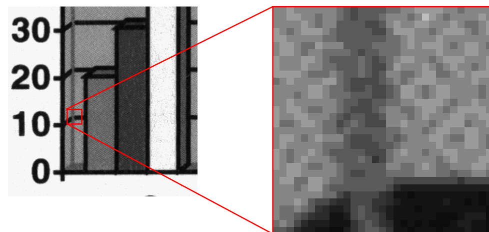
which can now be interpreted as the probability of g being the original image when image y is observed, we obtain a measure of the suitability of an quantization given an input image in terms of the fidelity and regularity. To find the quantized image given our input we have to maximize $P(g|y)$, the *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):

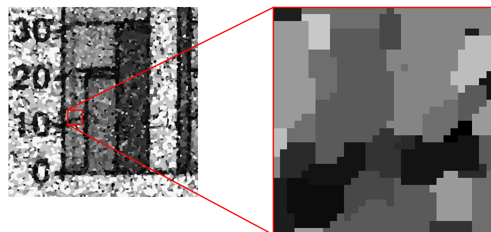
$$\begin{aligned} P(y|g)P(g) &= \prod_{s \in S} P(y_s|g_s) \frac{1}{Z} \exp\left(-\frac{1}{\hat{T}} \sum_{(s,r) \in \mathcal{C}_2} \delta(g_s, g_r)\right) \\ &= \frac{1}{Z} \exp\left\{ \sum_{s \in S} \left(-\ln \sqrt{2\pi}\sigma_{g_s} - \frac{(y_s - \mu_{g_s})^2}{2\sigma_{g_s}^2}\right) - \frac{1}{\hat{T}} \sum_{(s,r) \in \mathcal{C}_2} \delta(g_s, g_r) \right\}. \end{aligned} \tag{2.10}$$



(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 $\beta = \hat{T}^{-1}$ to denote the inverse temperature from here on. Finally we can take out a negative sign and minimize the following:

$$E(g) = \underbrace{\sum_{s \in S} \left(\ln \sqrt{2\pi}\sigma_{g_s} + \frac{(y_s - \mu_{g_s})^2}{2\sigma_{g_s}^2} \right)}_{\text{fidelity}} + \beta \underbrace{\sum_{C \in \mathcal{C}} \delta(g_s, g_r)}_{\text{regularity}}. \tag{2.11}$$

From this we can see that β determines the relative weight between fidelity and regularity. We can therefore control their relative importance using β 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 β and μ_{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 · C, ..., T_0 · C^{A−1} do
  x̂ ← Slightly changed state x
  ΔE ← E(x̂) − E(x)
  if exp{−ΔE/T} ≥ 1 then {Deterministic acceptance}
    x ← x̂
  else if exp{−ΔE/T} ≥ Rand[0, 1) then {Probabilistic acceptance}
    x ← x̂
  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 [5]

$$e^{-\Delta E/T}. \tag{2.12}$$

A parameter T (not to be confused with $\hat{T}$ 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.

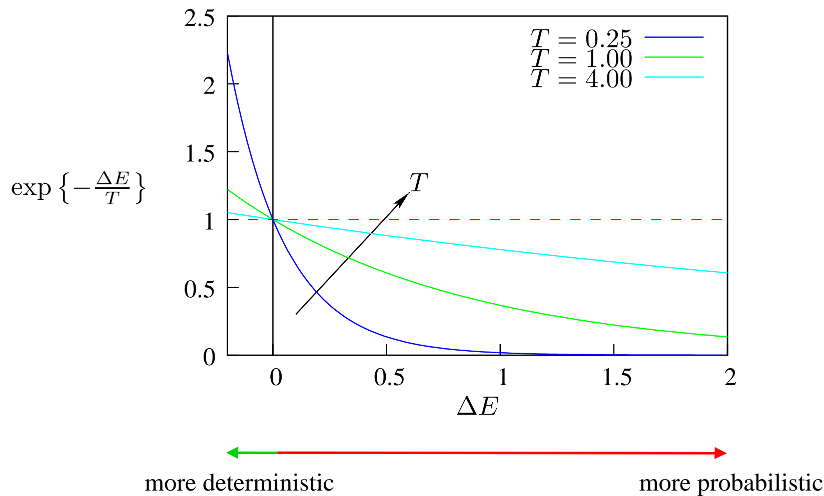


Figure 2.8: Acceptance probability at different temperatures.

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 α 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)

$$E(g_s) = \underbrace{\left(\ln(\sqrt{2\pi}\sigma_{g_s}) + \frac{(y_s - \mu_{g_s})^2}{2{\sigma_{g_s}}^2}\right)}_{\text{fidelity}} + \underbrace{\sum_{(s,t)\in C_{2s}} \beta\delta(g_s, g_t)}_{\text{regularity}}. \tag{2.13}$$

The global constant threshold α has the same function as the uniform ran-

```
g ← Random quantized image
for T ← T_0, T_0 · C, ..., T_0 · C^{A−1} do
  ĝ ← Slightly changed quantized image g
  for s ∈ S do
    ΔE_s ← E(ĝ_s) − E(g_s)
    if ΔE_s ≤ 0 then {Deterministic acceptance}
      g_s ← ĝ_s
    else if ΔE_s ≤ T · − ln α then {Probabilistic acceptance}
      g_s ← ĝ_s
    end if
  end for
end for
```

Algorithm 2: Modified Metropolis Dynamics

dom value in Algorithm 1: the comparison

$$\exp\left(-\frac{\Delta E}{T}\right) \geq \text{Rand}[0,1)$$

becomes

$$\Delta E_s \leq T \cdot - \ln \alpha. \tag{2.14}$$

This means that for α close to 0 all new states are accepted, while for $\alpha = 1$ only energy decreases are accepted.

If the standard deviations of quantization classes within input images do not vary much, then we can simplify the energy function, assuming the standard deviations are constant. Therefore we define the following energy function:

$$\tilde{E}(g_s) = (y_s - \mu_{g_s})^2 + \sum_{(s,t)\in C_{2s}} \tilde{\beta}\delta(g_s, g_t), \tag{2.15}$$

where $\tilde{\beta} = (2{\sigma_{g_s}}^2)\beta$. The logarithmic terms eliminate each other in the calculation of ΔE_s. $\tilde{E}(g_s)$ minimizes $(2{\sigma_{g_s}}^2)E(g_s)$, which has the same minimum as $E(g_s)$ if all standard deviations are the same.

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 α and β the choice should be more ad hoc. For example, it is reasonable to expect that an α 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 (FPGAs).

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

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.

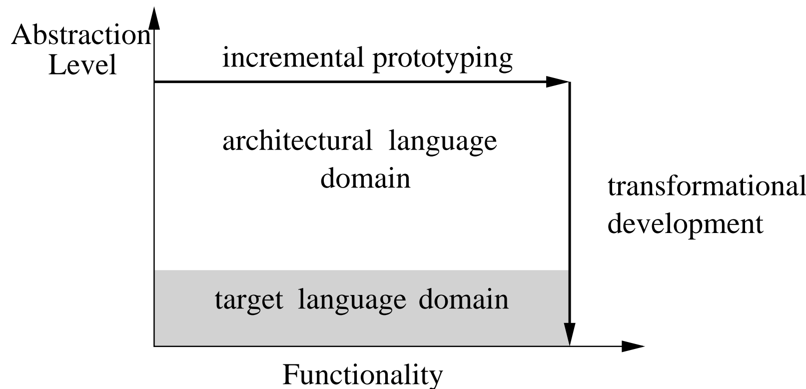


Figure 3.1: Path of evolutionary design through the design space.

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-

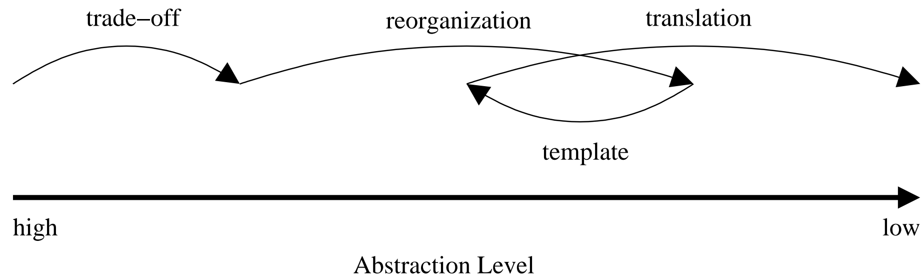


Figure 3.2: Transformational development

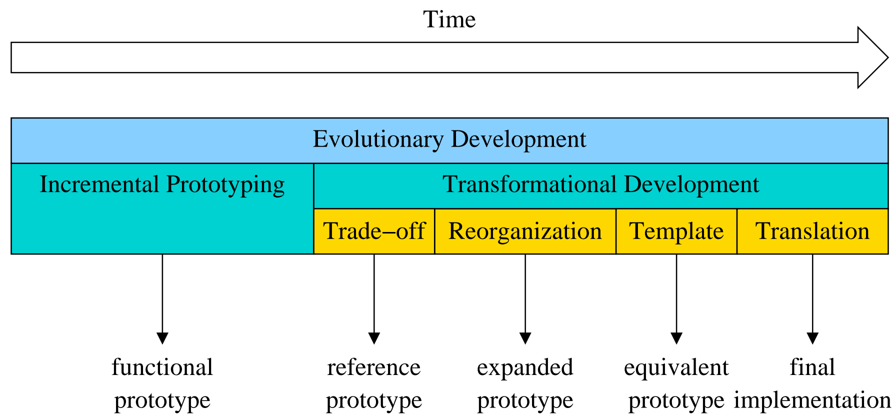


Figure 3.3: Overview of Evolutionary Design: (Sub)phases are passed through from left to right in time. The nesting of the different subphases are also shown.

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-Off 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.

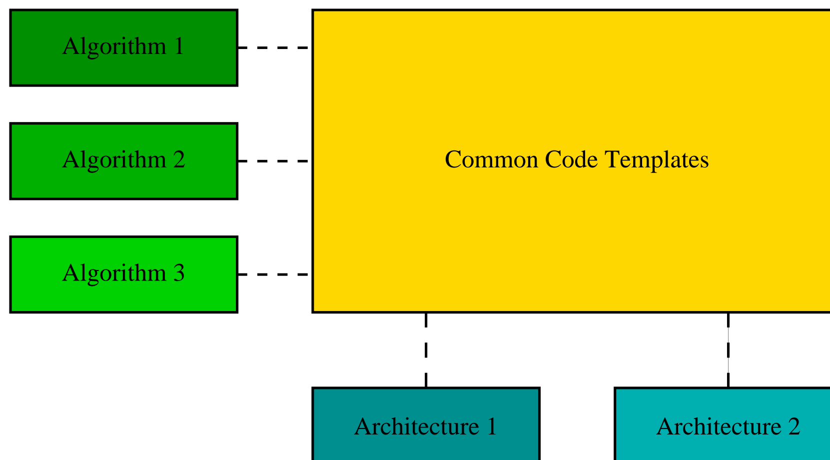


Figure 3.4: Increasing portability by using common code templates. Algorithms are implemented in terms of these code templates. For each code template there is an implementation on each architecture.

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.

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

		Instructions	
		Single	**Multiple**
Data	**Single**	SISD	MISD
	Multiple	SIMD	MIMD

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[1] supercomputer company. During the last decennium 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] [2] and RC5–72 [3] [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_{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*:

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

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.

[1] `http://www.cray.com`

[2] a project to find extraterrestrial intelligence by analyzing data from radio telescopes.

[3] a project to crack a 72-bit cryptology key.

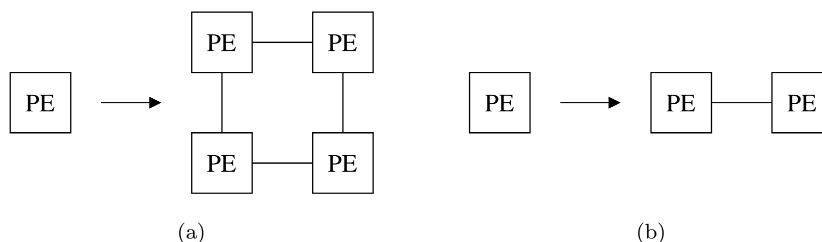


Figure 4.1: Expanding a mesh (a) requires more PEs than expanding a string (b).

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.

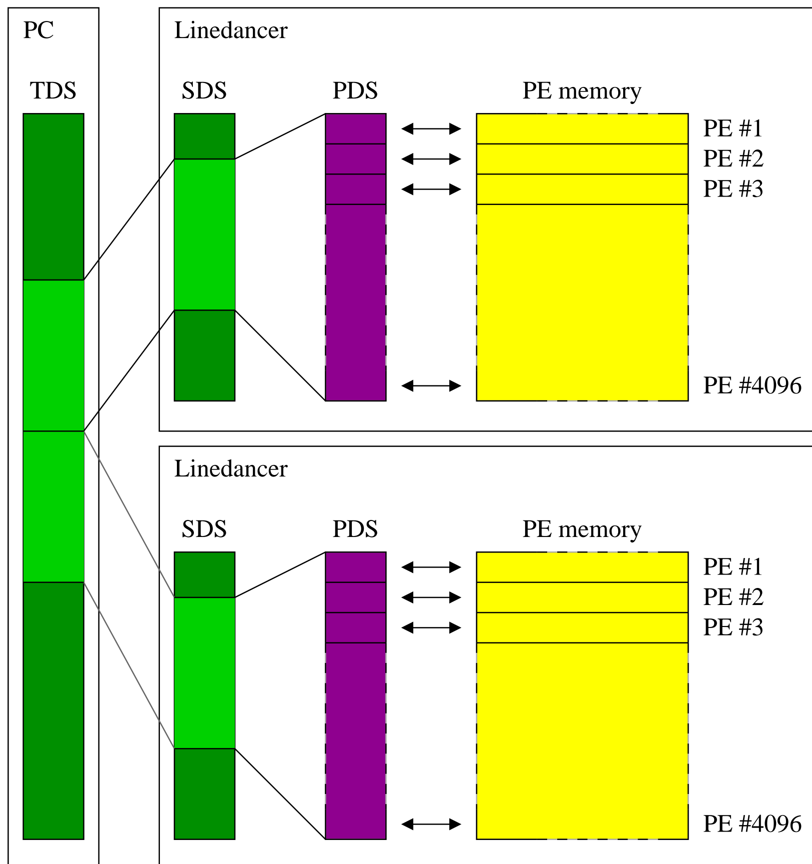


Figure 4.2: Transfers inside the memory hierarchy of the Linedancer

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 PEs 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 PEs 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

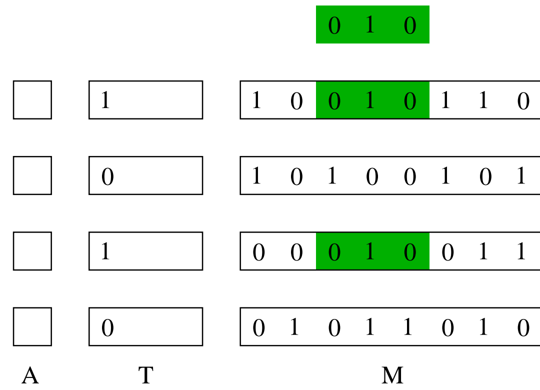
Data inside the registers can be freely moved between the PEs. The number of cycles this takes depends on the distance between the PEs involved.

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

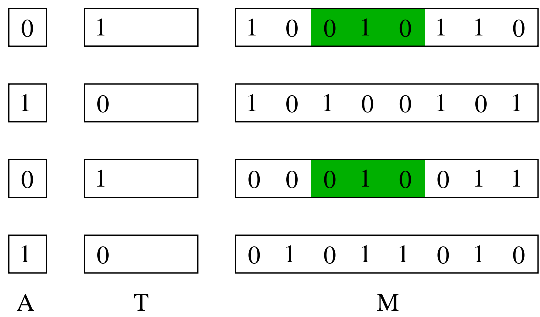
It is possible to transfer the contents of the tag registers through the array, one PE at the time. This can be used to access PEs based on the contents of their neighbors.

4.2.5 Segmentation

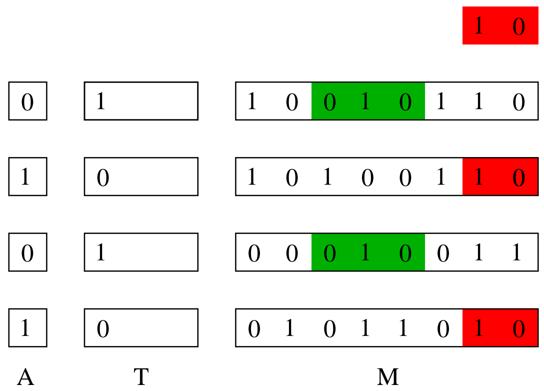
By default the array behaves as a string with its extreme ends connected. The PEs can be split up into multiple *segments*. These segments each operate independently as if they were physically distinct Linedancers, without the possibility of communication between them. This way, multiple Linedancers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



(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.

Instruction	Description
RTS(Add)	Addition
RTS(Sub)	Subtraction
RTS(AddSub)	Either addition or subtraction, depending on a special bit
RTS(Assign)	Assignment, within a single PE or from one PE to another
RTS(le)	Less than of equal comparison
RTS(Xor)	Bitwise Xor

Table 4.2: Some of the RTS instructions that can be used on the Linedancer.

4.2.6 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 Table 4.2. 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.

4.3 Programmer's Interface

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

4.3.1 Software Tools

The programming language is C, extended with aop blocks, which contain Linedancer instructions. If the SCC and IDSM 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 IDSM processors.

4.3.2 Threads of Execution

The IDSM 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.

Instruction	Cycles
Tag	4
Segmentation	2
Write	2
Activate	2

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 IDSM or to return status information from the IDSM 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.

Instruction	Cycles	
	Even Width	Odd Width
pdTransfer[4]	$14 + 12w$	$14 + 12w$
RTS(Add)	$2 + w$	$2 + 2w$
RTS(Sub)	$2 + w$	$2 + 2w$
RTS(AddSub)	$2 + 2w$	$2 + 4w$
RTS(Assign) (local)	$2 + w$	$2 + 2w$
RTS(le)	$6 + w$	$8 + 2w$
RTS(Xor)	$4 + w$	$4 + 2w$

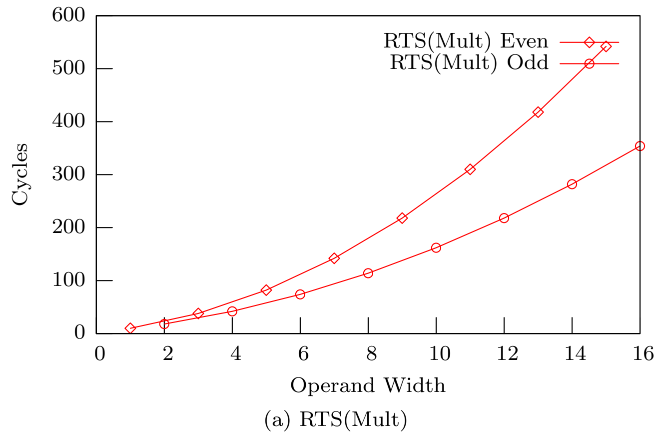
Table 4.4: Linedancer operations which are linearly dependent on the bit width.

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.

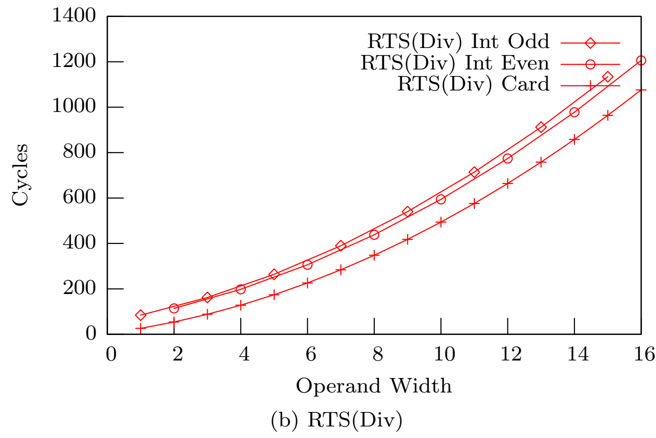
4.4.3 Slow Instructions

In Figure 4.4 the relation between bit width and the running time in cycles is shown for the RTSMult 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.



(a) RTS(Mult)



(b) RTS(Div)

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

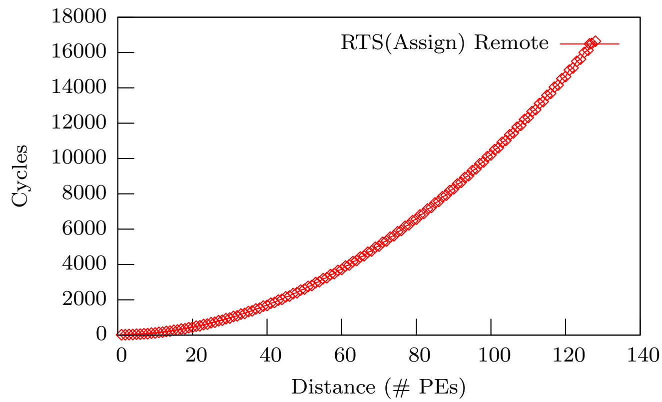


Figure 4.5: Dependency of the number of cycles on the operand width for the remote RTS(Assign) instruction.

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 [8] 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

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 B.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:

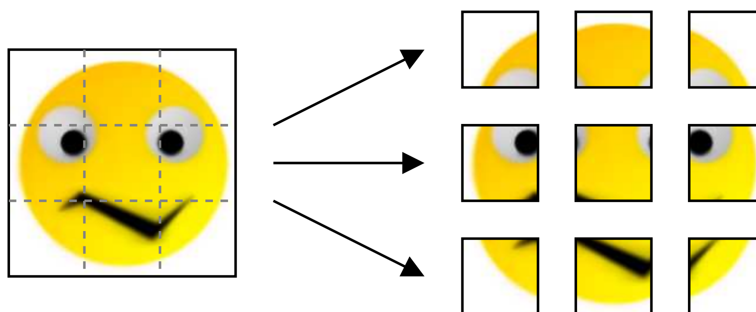


Figure 5.1: Tiling: the input image is cut into blocks fitting inside the Linedancer and processed in sequence by the Simulated Annealing algorithm.

- 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:

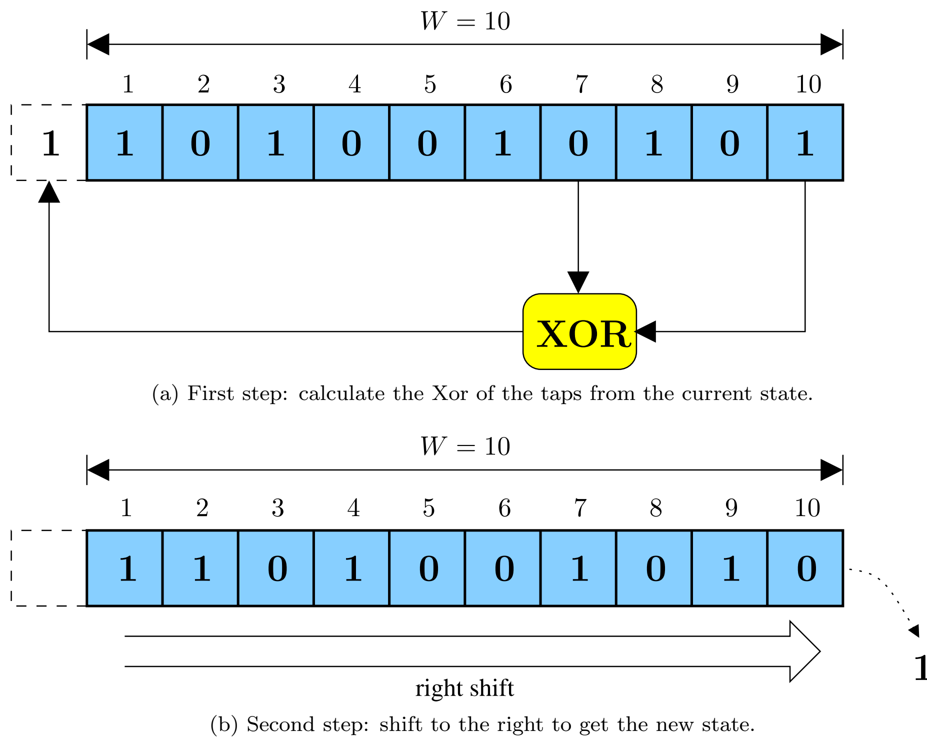
$\tilde{\beta}$ the inverse of the Simulated Annealing temperature, which can be interpreted as the weighting factor between fidelity and regularity;

μ_{g_s} 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 μ_{g_s}. The misclassification rates of all but the first image fell below the 5% threshold. We therefore found it suitable to use integer values for μ_{g_s}.



(a) First step: calculate the Xor of the taps from the current state.

(b) Second step: shift to the right to get the new state.

Figure 5.2: A 10 bits wide Linear Feedback Shift Register. A 0 is shifted out and $1 \,\mathrm{Xor}\, 0 = 1$ is shifted in.

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, `random` is a function implementing a LFSR):

```
NB. Initialize seed to random values.
seed=. ? ($ y.) $ (# , y.)
NB. Calculate new seed using a LFSR of 10 bits wide and taps at
NB. bits 10 and 7.
seed=. (10 ; 10 7) random seed
NB. Map seed to state by subtracting one and calculate the
NB. modulus with the number of labels LABELS.
nstate=. LABELS | <: seed
```

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
NB. 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. - state { 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.1.4, lines 27–33 we find the following loop:

```
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):

```
NB. For each n in {0,1, ... NBCOUNT-1}
for_n. i. NBCOUNT do.
  a_14_10=. (n { NBORS) |.!.0 state_10_4
  a_14_10=. state_10_4 ~: a_14_10
  eq=. a_14_10 = 0
  neq=. -. eq
  e_82_20=. e_82_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.

Code templates can be used to perform checks on the data, such as checking calling conventions. For instance, the code in Section B.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 `NB.{{` and `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.

```
NB. Force seed_0_5 into the right shape.
seed_0_5=: ($ y.) $ seed_0_5              NB. not translated
NB.{{
  AssignField 'state_10_4' ; 'seed_0_4'   NB. translated
NB.}}
```

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.

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.

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 B.1.3, lines 12,16,17):

```
NB. Initialize seed with a number between 1 and 2^5 - 1
seed=. >: ? ($ y.) $ <: 2^5
NB. Calculate new seed using a LFSR of 5 bits wide and taps at
NB. bits 5 and 3.
seed=. (5 ; 5 3) random seed
NB. Map seed to state by calculate the modulus with the number
NB. of labels LABELS.
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 where 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 B.2.1). The parser then creates a file that can be included into a wrapper file (Section B.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 B.1.7, lines 53–63)

```
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 B.2.2, lines 79–101, omitting some less important code):

```
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.

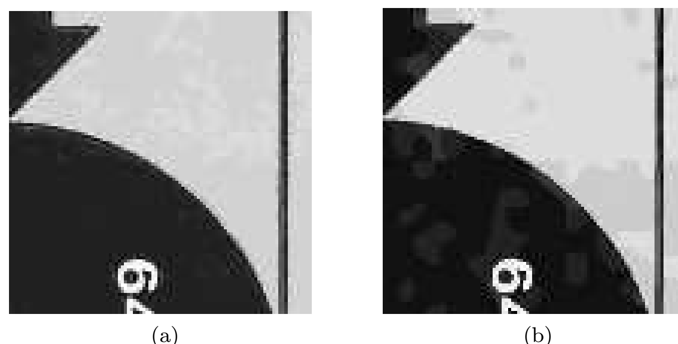
6.1.1 Algorithm Load Distribution

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 β 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



(a) (b)

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

Preparation	**70**
Processing one tile	**8346**
Calculate a new random labeling	44
For each label	352
$y - \mu_{g_s}$	22
Square	164
For each neighbor	3592
Add or subtract β	596
Load y	16
For each label	352
$y - \mu_{g_s}$	22
Square	164
For each neighbor	3592
Add or subtract β	596
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.

# Tiles	DSM		ISM	
	Cycles / Tile	%	Cycles / Tile	%
2	25475	74.4252	8754	25.5748
8	22550.2	72.0357	8754	27.9643
28	21824.6	71.3721	8754	28.6279
98	21686.1	71.2419	8754	28.7581
378	21687.2	71.2429	8754	28.7571

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

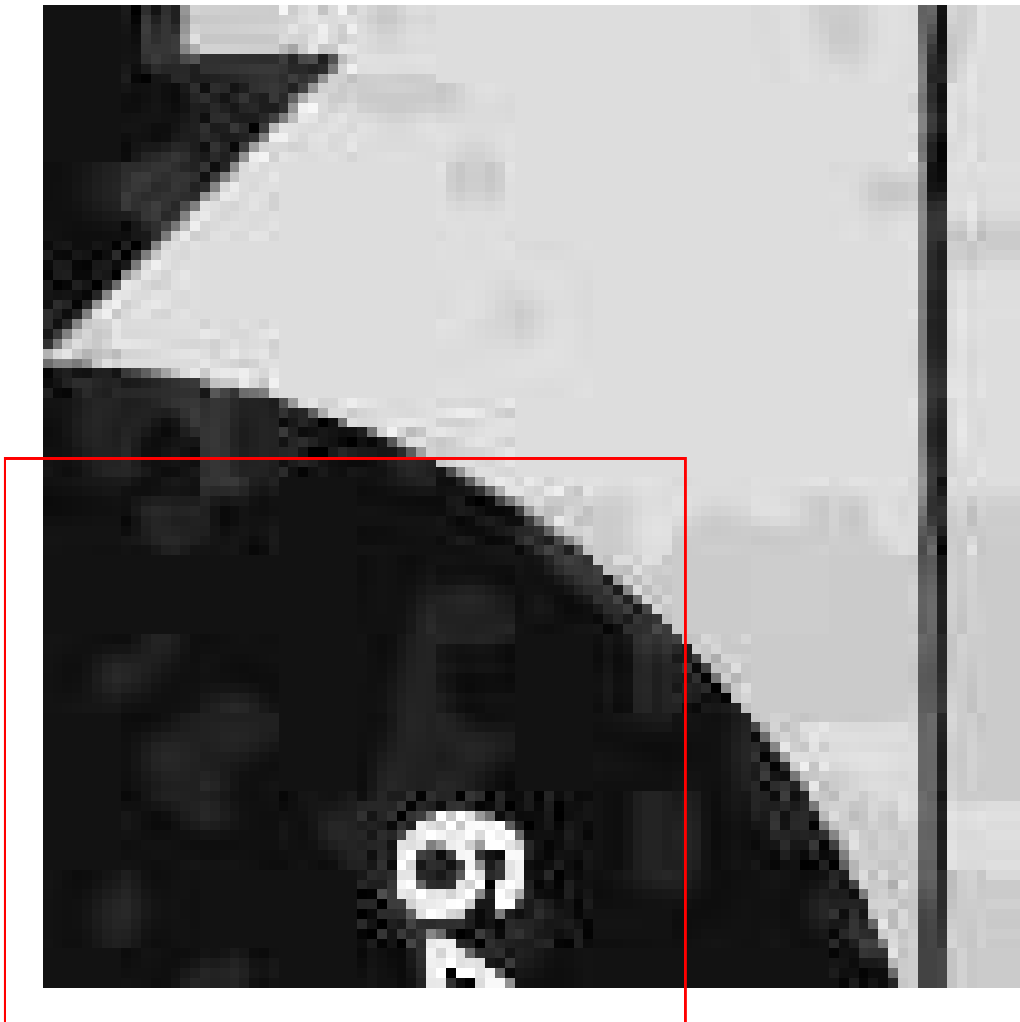


Figure 6.2: Tiling artifact in the output of the quantization algorithm. If you look close inside the red rectangle you can see that the blobs have horizontal and vertical borders. These are the tile edges.

# Tiles	DSM		ISM	
	Cycles / Tile	%	Cycles / Tile	%
2	25558	23.3564	83868	76.6436
8	22671.5	21.2799	83868	78.7201
28	21959.2	20.7501	83868	79.2499
98	21794.5	20.6265	83868	79.3735
378	21797.7	20.6289	83868	79.3711

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

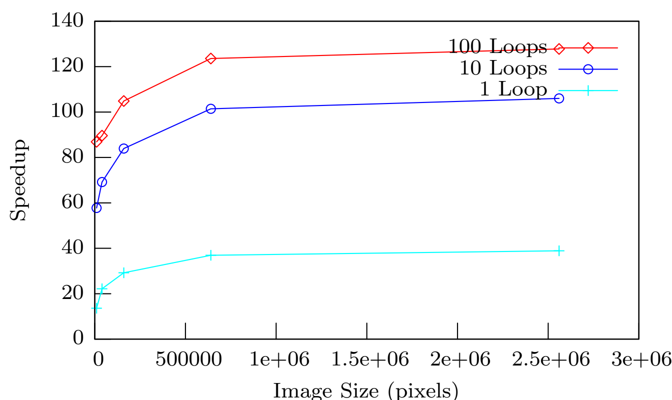


Figure 6.3: Speedup for various image sizes and number of annealing loops.

Annealing loops performed.

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

# Pixels	Time (ms)	
	Pentium	Linedancer
10000	5.75	0.422
40000	22.5	1.01
160000	91.3	3.13
640000	374	10.1
2560000	1500	38.6

(a) 1 Run

# Pixels	Time (ms)	
	Pentium	Linedancer
10000	52.5	0.909
40000	211	305
160000	856	10.2
640000	3530	34.8
2560000	14100	133

(b) 10 Runs

# Pixels	Time (ms)	
	Pentium	Linedancer
10000	517	5.95
40000	2070	23.1
160000	8420	80.3
640000	34600	280
2560000	138000	1080

(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

[1] Distributed.net. `http://www.distributed.net/index.php.en`.

[2] Seti@home. `http://setiathome.ssl.berkeley.edu/`.

[3] Robert Bernecky. APL: A prototyping language. In *Proceedings of the international conference on APL*, pages 221–228, New York, NY, USA, July 7, 1986. SIGAPL, ACM Press.

[4] Gerrit A. Blaauw and Frederick P. Brooks Jr. *Computer Architecture, Concepts and Evolution*. Addison Wesley Longman, Inc., 1997.

[5] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, Inc., 2001.

[6] EETimesUK. Aspex linedancer used for video processing, February 3, 2006. [Online; accessed 2-May-2006].

[7] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*, pages 222–223. Prentice-Hall, Inc., second edition, 2002.

[8] Jsoftware Inc. J home, 2006. [Online; accessed 28-June-2006].

[9] Jsoftware Inc. J vocabulary, 2006. [Online; accessed 28-June-2006].

[10] Zoltan Kato. *Modelisations markoviennes multiresolutions en vision par ordinateur. Application a la segmentation d'images SPOT*. PhD thesis, University of Nice, December 1994. [English translation].

[11] Aspex Semiconductor Limited. Activate release 2.2.1. `SP221-APRG.chm`, 2005.

[12] Steven Segenchuk. An overview of color quantization techniques, 1997. [Online; accessed 30-June-2006].

[13] J. M. Spivey. *The Z Notation — A Reference Manual*. Oriel College, Oxford, 1998. [Online; accessed 2-July-2006].

[14] Tamás Szirányi et al. Image segmentation using Markov random field model in fully parallel cellular network architectures. *Real-Time Imaging*, 2000.

[15] Wikipedia. Bayes' theorem — wikipedia, the free encyclopedia, 2006. [Online; accessed 30-June-2006].

[16] Wikipedia. Linear feedback shift register — wikipedia, the free encyclopedia, 2006. [Online; accessed 20-March-2006].

[17] Gerhard Winkler. *Image Analysis, Random Fields and Dynamic Monte Carlo Methods: A Mathematical Introduction.* Springer-Verlag, 1995.

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 10×8 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 10×8 (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 $<, \leq, =, \geq, >$ 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
```

<table>
<tr><th>J Verb</th><th>C</th><th>Mathematical description</th></tr>
<tr><td><code>*: y</code></td><td><code>pow(y,2.0)</code></td><td>y^2</td></tr>
<tr><td><code>x + y</code></td><td><code>x + y</code></td><td>$x + y$</td></tr>
<tr><td><code>x - y</code></td><td><code>x - y</code></td><td>$x - y$</td></tr>
<tr><td><code>x = y</code></td><td><code>x == y</code></td><td>$x = y$</td></tr>
<tr><td><code>x * y</code></td><td><code>x * y</code></td><td>$x \cdot y$</td></tr>
<tr><td><code>x ^ y</code></td><td><code>pow(x,y)</code></td><td>x^y</td></tr>
<tr><td><code>>: y</code></td><td><code>y + 1</code></td><td>$y + 1$</td></tr>
<tr><td><code><: y</code></td><td><code>y - 1</code></td><td>$y - 1$</td></tr>
<tr><td><code>x | y</code></td><td><code>y % x</code></td><td>$y \bmod x$</td></tr>
<tr><td><code>-. y</code></td><td><code>! y</code></td><td>$\neg y$</td></tr>
<tr><td><code>x ~: y</code></td><td><code>x != y</code></td><td>$x \neq y$</td></tr>
<tr><td><code>x >: y</code></td><td><code>x >= y</code></td><td>$x \geq y$</td></tr>
<tr><td><code>x <: y</code></td><td><code>x <= y</code></td><td>$x \leq y$</td></tr>
</table>

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

<table>
<tr><th>J Verb</th><th>C</th><th>Description</th></tr>
<tr><td><code>x |. y</code></td><td></td><td>Rotate the contents of an array</td></tr>
<tr><td><code>x |.!.0 y</code></td><td><code><<, >></code></td><td>Shift the contents of an array</td></tr>
<tr><td><code>? y</code></td><td><code>rand()</code></td><td>For each value v in <code>y</code>, generate a random integer in the range $\{0, 1, \ldots, v - 1\}$</td></tr>
<tr><td><code>x { y</code></td><td><code>y[x]</code></td><td>Extract an element located by <code>x</code> from array <code>y</code></td></tr>
</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,..,3}
? 4 4 4 4 4
```

J Verb	Description
`i.`	Generate list of integers
`;`	Concatenate objects of arbitrary type
`,`	Concatenate arrays of same rank
`,:`	Concatenate arrays of different rank

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:

```
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:

```
for_t. 2 3 4 do.
end.
```

is the same as the following C code:

```
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:

Subphase	Filename
Trade-Off	`functional*.ijs`
Reorganization	`algorithm*.ijs`
Template	`architecture*.ijs`

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 (-.@:+:@:=)"_ _1 NBORS |."1 _ state
10                 e=. lhood + prior
```

```
11
12                  lhood=. *: y. - nstate { MU
13                  prior=. BETA * +/ nstate (-.@:+:@:=)"_ _1 NBORS |."1 _ state
14                  ne=. lhood + prior
15
16                  de=. ne - e
17                  accept=. de <: - t * ^. ALPHA
18
19                  state=. (nstate * accept) + state * -. accept
20          end.
21          state
22  )
```

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.

```
1   random=: 4 : 0"_ 0
2           'len taps'=. x.
3           taps=. <: taps
4           state=. (- len) {. #: y.
5           b=. ~:/ taps { state
6           #. len {. b , state
7   )
8
9   annealing=: 3 : 0
10          (9!:1) 7^5
11
12          NB. Give all pixels a unique starting value.
13          seed=. ? ($ y.) $ (# , y.)
14          state=. LABELS | seed
15
16          for_t. TEMPO * COOLING ^ i. RUNS do.
17                  seed=. (10;10 7) random seed
18
19                  nstate=. LABELS | <: seed
20
21                  lhood=. *: y. - state { MU
22                  prior=. BETA * +/ state (-.@:+:@:=)"_ _1 NBORS |.!.0"1 _ state
23                  e=. lhood + prior
24
25                  lhood=. *: y. - nstate { MU
26                  prior=. BETA * +/ nstate (-.@:+:@:=)"_ _1 NBORS |.!.0"1 _ state
27                  ne=. lhood + prior
28
29                  de=. ne - e
30                  accept=. de <: <. - t * ^. ALPHA
31
32                  state=. (nstate * accept) + state * -. accept
33          end.
34          state
35  )
```

```
36
```

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;5 3) random seed
                nstate=. LABELS | seed

                lhood=. *: y. - state { MU
                prior=. BETA * +/ state (-.@:+:@:=)"_ _1 NBORS |.!.0"1 _ state
                e=. lhood + prior

                lhood=. *: y. - nstate { MU
                prior=. BETA * +/ nstate (-.@:+:@:=)"_ _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
```

```
7
8           for_t. i. RUNS do.
9                   s=. _10 {."1 #: seed
10                  b1=. TAP1 {"1 s
11                  b2=. TAP2 {"1 s
12                  b=. b1 ~: b2
13                  ns=. 9 {."1 s
14                  seed=. #. b ,"_2 ns
15                  seed=. seed - 1
16                  nstate=. LABELS | seed
17                  seed=. seed + 1
18
19                  mu=. 0
20                  for_l. i. LABELS do.
21                          eq=. state = l
22                          neq=. -. eq
23                          mu=. mu + eq * l { MU
24                  end.
25                  a=. y. - mu
26                  e=. a * a
27                  for_n. NBORS do.
28                          nb=. n |.!.0 state
29                          nb=. state ~: nb
30                          eq=. nb = 0
31                          neq=. -. eq
32                          e=. e + (eq * - BETA) + (neq * BETA)
33                  end.
34
35                  mu=. 0
36                  for_l. i. LABELS do.
37                          eq=. nstate = l
38                          neq=. -. eq
39                          mu=. mu + eq * l { MU
40                  end.
41                  a=. y. - mu
42                  ne=. a * a
43                  for_n. NBORS do.
44                          nb=. n |.!.0 state
45                          nb=. nstate ~: nb
46                          eq=. nb = 0
47                          neq=. -. eq
48                          ne=. ne + (eq * - BETA) + (neq * BETA)
49                  end.
50
51                  ne=. ne - e
52                  accept=. ne <: t { ALPHAs
53
54                  state=. (nstate * accept) + state * -. accept
55          end.
56          state
57  )
```

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_4=. LABELS | seed_0_10
                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
                        a_14_10=. state_10_4 ~: a_14_10
                        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
```

```
49                          a_14_10=. seed_0_4 ~: a_14_10
50                          eq=. a_14_10 = 0
51                          neq=. -. eq
52                          ne_24_20=. ne_24_20 + (eq * - BETA) + neq * BETA
53                  end.
54
55                  NB. Compare energies
56                  ne_24_20=. ne_24_20 - e_82_20
57                  accept_1004_1=. ne_24_20 <: t { ALPHAs
58                  eq=. accept_1004_1 = 1
59                  neq=. -. eq
60                  state_10_4=. (eq * seed_0_4) + neq * state_10_4
61          end.
62          state_10_4
63  )
```

B.1.6 File: architecture15.ijs

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

```
1   load jpath'~addons/image3/image3.ijs'
2   load jpath'~addons/image3/view_m.ijs'
3
4   load jpath'd:/leroy/trunk/quantization/models/templates2.ijs'
5
6   annealing=: 3 : 0
7   NB.{{
8           NB. Give all pixels a unique starting value.
9           LoadFile 'seed_0_10' ; 'd:/leroy/trunk/quantization/seed1023.dat' ; 'ro{pdtRdv,sdtRdv}
10  NB.}}
11          NB. Force seed_0_10 into the right shape.
12          seed_0_10=: ($ y.) $ seed_0_10
13          NB. Explicitly assign bits to seed_0_4
14          seed_0_4=: LABELS | seed_0_10
15  NB.{{
16          AssignField 'state_10_4' ; 'seed_0_4'
17  NB.}}
18          pixels=. y.
19  NB.{{
20          NB. Give pixel value a name
21          LoadField 'y_74_8' ; pixels ; 'ro{pdtRdv}'
22  NB.}}
23  NB.{{
24          for_t. i. RUNS do.
25  NB.}}
26                  NB. Randomize new state
27                  NB. Get the bit representation so we can work
28                  NB. at bit level.
29                  s=. _10 {."1 #: seed_0_10
30
31                  NB. These assignments are used as an alias for these
32                  NB. bits so they can be used in Rts instructions.
33                  NB. seed_0_10 does not change during their use.
```

```
34                  seed_0_1=: TAP1 {"1 s
35                  seed_3_1=: TAP2 {"1 s
36                  seed_1_9=: 9 {."1 s
37  NB.{{
38                  Xor 'b_64_1' ; 'seed_0_1' ; 'seed_3_1'
39
40                  AssignField 'ns_65_9' ; 'seed_1_9'
41                  AssignField 'seed_0_9' ; 'ns_65_9'
42                  AssignField 'seed_9_1' ; 'b_64_1'
43  NB.}}
44                  NB. Update seed_0_10 to reflect the changes in its bits.
45                  seed_0_10=: #. seed_9_1 ,"_2 seed_0_9
46
47                  NB. Another labeling for some bits.
48                  NB. seed_0_10 does not change during its use.
49                  seed_0_4=: LABELS | seed_0_10
50
51  NB.{{
52                  NB. Calculate current state energy
53                  AssignField 'a_14_10' ; 'y_74_8'
54                  for_l. i. LABELS do.
55                          SelectiveSubValue 'a_14_10' ; 'a_14_10' ; 'state_10_4' ; l ; (l { MU) ; 'Int'
56                  end.
57                  Mul 'e_82_20' ; 'a_14_10' ; 'a_14_10' ; 'Int'
58                  for_n. i. NBCOUNT do.
59                          FetchNeighbor 'a_14_10' ; 'state_10_4' ; (n { NBORS)
60  NB.}}
61                          NB. Alias a_14_4
62                          NB. a_14_10 is overwritten after this.
63                          a_14_4=: LABELS | a_14_10
64  NB.{{
65                          Xor 'a_14_4' ; 'state_10_4' ; 'a_14_4'
66
67                          SelectiveAddSubValue 'e_82_20' ; 'e_82_20' ; 'a_14_4' ; 0 ; BETA ; 'Int'
68                  end.
69
70                  NB. Calculate new state energy
71                  AssignField 'a_14_10' ; 'y_74_8'
72                  for_l. i. LABELS do.
73                                  SelectiveSubValue 'a_14_10' ; 'a_14_10' ; 'seed_0_4' ; l ; (l { MU) ; 'In
74                  end.
75                  Mul 'ne_24_20' ; 'a_14_10' ; 'a_14_10' ; 'Int'
76                  for_n. i. NBCOUNT do.
77                          FetchNeighbor 'a_14_10' ; 'state_10_4' ; (n { NBORS)
78  NB.}}
79                          NB. Alias a_14_4
80                          NB. a_14_10 is overwritten after this.
81                          a_14_4=: LABELS | a_14_10
82  NB.{{
83                          Xor 'a_14_4' ; 'seed_0_4' ; 'a_14_4'
84
85                          SelectiveAddSubValue 'ne_24_20' ; 'ne_24_20' ; 'a_14_4' ; 0 ; BETA ; 'Int'
86                  end.
87
```

```
88                  NB. Compare energies
89                  Sub 'ne_24_20' ; 'ne_24_20' ; 'e_82_20' ; 'Int'
90
91                  LEValue 'accept_1004_1' ; 'ne_24_20' ; (t { ALPHAs) ; 'Int'
92                  SelectiveUpdate 'state_10_4' ; 'seed_0_4' ; 'accept_1004_1' ; 1
93
94                  AddValue 'seed_0_10' ; 'seed_0_10' ; 1 ; 'Card'
95          end.
96  NB.{{
97          AssignField 'debug_32_32' ; 'seed_0_10'
98  NB.}}
99          NB. Double the debug output.
100         debug_32_32=: ,~ debug_32_32
101 NB.{{
102         DumpFile 'state_10_4' ; '' ; 'ro{sdtRdv}'
103         DumpFile 'debug_32_32' ; '' ; 'ro{pdtRdv,sdtRdv}'
104 NB.}}
105         state_10_4
106 )
```

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 subphases.

```
1   annealing=: 3 : 0
2   NB.{{
3           NB. Give all pixels a unique starting value.
4           LoadFile 'seed_0_5' ; 'd:/leroy/trunk/quantization/seed31.dat' ; 'ro{pdtRdv,sdtRdv}'
5   NB.}}
6
7           NB. Force seed_0_5 into the right shape.
8           seed_0_5=: ($ y.) $ seed_0_5
9
10          NB. Explicitly assign bits to seed_0_4
11          seed_0_4=: LABELS | seed_0_5
12  NB.{{
13          AssignField 'state_10_4' ; 'seed_0_4'
14  NB.}}
15          pixels=. y.
16  NB.{{
17          NB. Give pixel value a name
18          LoadField 'y_74_8' ; pixels ; 'ro{pdtRdv}'
19
20          for_t. i. RUNS do.
21  NB.}}
22                  NB. Randomize new state
23                  NB. Get the bit representation so we can work
24                  NB. at bit level.
25                  s=. _5 {."1 #: seed_0_5
26
27                  NB. These assignments are used as an alias for these
28                  NB. bits so they can be used in Rts instructions.
```

```
29                  NB. seed_0_10 does not change during their use.
30                  seed_0_1=: TAP1 {"1 s
31                  seed_2_1=: TAP2 {"1 s
32                  seed_1_4=: 4 {."1 s
33  NB.{{
34                  Xor 'b_64_1' ; 'seed_0_1' ; 'seed_2_1'
35
36                  AssignField 'ns_65_4' ; 'seed_1_4'
37                  AssignField 'seed_0_4' ; 'ns_65_4'
38                  AssignField 'seed_4_1' ; 'b_64_1'
39  NB.}}
40                  NB. Update seed_0_10 to reflect the changes in its bits.
41                  seed_0_5=: #. seed_4_1 ,"_2 seed_0_4
42
43                  NB. Another labeling for some bits.
44                  NB. seed_0_10 does not change during its use.
45                  seed_0_4=: LABELS | seed_0_5
46  NB.{{
47                  NB. Calculate current state energy
48                  AssignField 'a_14_10' ; 'y_74_8'
49          for_l. i. LABELS do.
50                  SelectiveSubValue 'a_14_10' ; 'a_14_10' ; 'state_10_4' ; l ; (l { MU) ; 'Int'
51          end.
52          Mul 'e_82_20' ; 'a_14_10' ; 'a_14_10' ; 'Int'
53                  for_n. i. NBCOUNT do.
54                          FetchNeighbor 'a_14_10' ; 'state_10_4' ; (n { NBORS)
55  NB.}}
56                          NB. Alias a_14_4
57                          NB. a_14_10 is overwritten after this.
58                          a_14_4=: LABELS | a_14_10
59  NB.{{
60                          Xor 'a_14_4' ; 'state_10_4' ; 'a_14_4'
61
62                          SelectiveAddSubValue 'e_82_20' ; 'e_82_20' ; 'a_14_4' ; 0 ; BETA ; 'Int'
63                  end.
64
65                  NB. Calculate new state energy
66                  AssignField 'a_14_10' ; 'y_74_8'
67                  for_l. i. LABELS do.
68                                  SelectiveSubValue 'a_14_10' ; 'a_14_10' ; 'seed_0_4' ; l ; (l { MU) ; 'In
69                  end.
70                  Mul 'ne_24_20' ; 'a_14_10' ; 'a_14_10' ; 'Int'
71                  for_n. i. NBCOUNT do.
72                          FetchNeighbor 'a_14_10' ; 'state_10_4' ; (n { NBORS)
73  NB.}}
74                          NB. Alias a_14_4
75                          NB. a_14_10 is overwritten after this.
76                          a_14_4=: LABELS | a_14_10
77  NB.{{
78                          Xor 'a_14_4' ; 'seed_0_4' ; 'a_14_4'
79
80                          SelectiveAddSubValue 'ne_24_20' ; 'ne_24_20' ; 'a_14_4' ; 0 ; BETA ; 'Int'
81                  end.
82
```

```
83                  NB. Compare energies
84                  Sub 'ne_24_20' ; 'ne_24_20' ; 'e_82_20' ; 'Int'
85
86                  LEValue 'accept_6_1' ; 'ne_24_20' ; (t { ALPHAs) ; 'Int'
87                  SelectiveUpdate 'state_10_4' ; 'seed_0_4' ; 'accept_6_1' ; 1
88          end.
89          NB. Debugging assignment
90          AddValue 'debug_32_32' ; 'state_10_4' ; 0 ; 'Card'
91  NB.}}
92          NB. The Linedancer produces two times the same result.
93          debug_32_32=: ,~ debug_32_32
94  NB.{{
95          DumpFile 'state_10_4' ; '' ; 'ro{sdtRdv}'
96          DumpFile 'debug_32_32' ; 'd:\leroy\trunk\quantization\output.dump' ; 'ro{pdtRdv,sdtRdv
97  NB.}}
98          state_10_4
99  )
```

B.1.8 File: templates2.ijs

This file contains the J code implementing the code templates.

```
1   NB. ==========================================================
2   NB. Constraints checking
3
4   NB. Check RTS operation calling conventions.
5   NB. Returns (offset,size) pairs for <dst>, <src> and <src2>.
6   NB. Missing field can be given as empty string ''. This is
7   NB. returned as 0.
8   RtsCheck=: 3 : 0
9           'dst src1 src2'=. y.
10
11          NB. Get (offset,size) pair for src1, src2 and dst.
12          if. src1 -: '' do.
13                  us1=. 0
14                  nls1=. 0
15          else.
16                  us1=. Unpack src1
17                  nls1=. 1
18          end.
19          if. src2 -: '' do.
20                  us2=. 0
21                  nls2=. 0
22          else.
23                  us2=. Unpack src2
24                  nls2=. 1
25          end.
26          ud=.  Unpack dst
27
28          NB. Check for overlap
29          NB. Between src1 and dst
30          if. nls1 do.
31                  NB. When result and operand overlap, they must share the same
32                  NB. base address.
```

```
33                  if. us1 Overlap ud do. assert. {. ud = us1 end.
34          end.
35          NB. Between src2 and dst
36          if. nls2 do.
37                  NB. When result and operand overlap, they must share the same
38                  NB. base address.
39                  if. us2 Overlap ud do. assert. {. ud = us2 end.
40          end.
41
42          NB. Only one vector operand can come from EXT.
43          if. nls1 *. nls2 do.
44                  assert. 2 > +/ 64 < +/"1 us1 ,: us2
45          end.
46
47          ud ; us1 ; us2
48  )
49  NB. Get (offset,size) pair from register name
50  Unpack=: =&'_' ".;._1 ]
51  NB. Offset lhs lies between rhs.
52  Between=: ([ >: {.@]) *. ([ < +/@])
53  NB. Check if two (offset,size) pairs overlap.
54  Overlap=: (Between~ {.) +. (Between~ {.)~
55
56  NB. Check logical RTS operation calling conventions.
57  RtsLogicalCheck=: 3 : 0
58          'ud us1 us2'=. y.
59
60          NB. The size of the vector(s) Operand1 and/or Operand2 must always
61          NB. match the size of the Result.
62          if. -. us1 -: 0 do.     assert. 1 { ud = us1 end.
63          if. -. us2 -: 0 do. assert. 1 { ud = us2 end.
64
65          0
66  )
67
68  NB. ==============================================================
69  NB. Arithmetic operations
70
71  NB. Multiply field <src1> with field <src2> and put the result
72  NB. in <dst>.
73  NB.
74  NB. Activate(All,-,-,-),
75  NB. Write(Binary, 1, ab, 0),
76  NB. RTS(Mult, type, -, dst, src1, src2),
77  Mul=: 3 : 0
78          'dst src1 src2 type'=. y.
79          'ud us1 us2'=. RtsCheck dst ; src1 ; src2
80
81          NB. Result size must be equal to sum of size of operands.
82          assert. 1 { ud = +/ us1 ,: us2
83
84          NB. Result and multiplicand cannot be both in EXT.
85          m=. us1 Multiplicand us2
86          assert. 2 > +/ 64 < +/"1 ud ,: m
```

```
87
88         NB. Tests showed that we cannot do A_OUT <- A_IN * B, where A_IN and
89         NB. A_OUT have the same base address and size(A_OUT) = size(A_IN) +
90         NB. size(B).
91         NB. Basically this means that the result cannot have the same base
92         NB. address as any of the operands. RtsCheck already takes care of
93         NB. the overlap situation.
94         assert. {. ud ~: us1
95         assert. {. ud ~: us2
96
97         ". dst , '=: ' , src1 , ' * ' , src2
98 )
99 NB. Return the multiplicand of a multiplication as a pair (o,s).
100 Multiplicand=: 1&{ @: < { ,:
101
102 NB. Add value <delta> to field <src> and put the result in
103 NB. field <dst>.
104 NB.
105 NB. Activate(All,-,-,-),
106 NB. Write(Binary, 1, ab, 0),
107 NB. RTS(Add, type, -, dst, src, delta),
108 AddValue=: 3 : 0
109         'dst src delta type'=. y.
110         RtsCheck dst ; src ; ''
111         ". dst , '=: ', src , ' + delta'
112 )
113
114 NB. Subtract field <src2> from field <src1> and put the result
115 NB. in field <dst>.
116 NB.
117 NB. Activate(All,-,-,-),
118 NB. Write(Binary, 1, ab, 0),
119 NB. RTS(Sub, type, -, dst, src1, src2),
120 Sub=: 3 : 0
121         'dst src1 src2 type'=. y.
122         RtsCheck dst ; src1 ; src2
123         ". dst , '=: ' , src1 , ' - ' , src2
124 )
125
126 NB. Subtract value <delta> from field <src> and put the result
127 NB. in field <dst>.
128 NB.
129 NB. Activate(All,-,-,-),
130 NB. Write(Binary, 1, ab, 0),
131 NB. RTS(Sub, type, -, dst, src, delta),
132 SubValue=: 3 : 0
133         'dst src delta type'=. y.
134         RtsCheck dst ; src ; ''
135         ". dst , '=: ', src , ' - delta'
136 )
137
138 NB. Perform XOR on field <src1> and field <src2> and put the
139 NB. result in <dst>.
140 NB. This implementation does not actually perform a binary
```

```
141  NB. XOR, but only compares the values. This is enough for
142  NB. our uses.
143  NB.
144  NB. Activate(All,-,-,-),
145  NB. Write(Binary, 1, ab, 0),
146  NB. RTS(Xor, Bitset, -, dst, src1, src2),
147  Xor=: 3 : 0
148          'dst src1 src2'=. y.
149          'ud us1 us2'=. RtsCheck dst ; src1 ; src2
150          RtsLogicalCheck ud ; us1 ; us2
151
152          ". dst , '=: ' , src1 , ' ~: ' , src2
153  )
154
155  NB. Set field <dst> to 1 where field <src> <: value <val>,
156  NB. and to 0 everywhere else.
157  NB.
158  NB. Activate(All,-,-,-),
159  NB. Write(Binary, 1, ab, 0),
160  NB. RTS(le, type, -, dst, src, val),
161  LEValue=: 3 : 0
162          'dst src val type'=. y.
163          'ud us1 us2'=. RtsCheck dst ; src ; ''
164
165          NB. <dst> must have size 1.
166          assert. 1 = 1 { ud
167
168          ". dst , '=: ' , src , ' <: val'
169  )
170
171  NB. ==========================================================
172  NB. Assignment operations
173
174  NB. Assign the contents of field <src> to field <dst>.
175  NB.
176  NB. Activate(All,-,-,-),
177  NB. Write(Binary, 1, ab, 0),
178  NB. RTS(Assign, Bitset, -, dst, src, -),
179  AssignField=: 3 : 0
180          'dst src'=. y.
181          'ud us1 us2'=. RtsCheck dst ; src ; ''
182
183          NB. Check if size of src <= size of dst.
184          NB. This is for convenience and not imposed by the architecture.
185          assert. 1 { ud >: us1
186
187          ". dst , '=: ' , src
188  )
189
190  NB. Fetch field <src> of neighbor <nbor> and assign its value
191  NB. to field <dst>.
192  NB.
193  NB. Activate(All,-,-,-),
194  NB. Write(Binary, 1, ab, 0),
```

```
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
202 NB. ============================================================
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
218         ". 'eq=. ' , cond , ' = val'
219         neq=. -. eq
220         ". dst , '=: (eq * ' , src , ' - delta) + (neq * ' , src , ')'
221 )
222
223 NB. Subtract value <delta> from field <src> where field <cond>
224 NB. equals value <val> and add value <delta> to field <src>
225 NB. everywhere else.
226 NB. Put the result in field <dst>.
227 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
239         ". 'eq=. ' , cond , ' = val'
240         neq=. -. eq
241         ". dst , '=: (eq * ' , src , ' - delta) + (neq * ' , src , ' + delta)'
242 )
243
244 NB. Assign field <src> to field <dst> where field <cond> equals
245 NB. value <val>, keep the rest of field <dst> intact.
246 NB.
247 NB. Activate(All,-,-,-),
248 NB. Write(Binary, 0, ab, 0),
```

```
249  NB. Tag(Binary, val, cond, tr1, -),
250  NB. Activate(Matching, -, tr1, -),
251  NB. Write(Binary, 1, ab, 0),
252  NB. RTS(Assign, Bitset, -, dst, src, -),
253  SelectiveUpdate=: 3 : 0
254          'dst src cond val'=. y.
255          RtsCheck dst ; src ; ''
256
257          ". 'eq=. ' , cond , ' = val'
258          neq=. -. eq
259          ". dst , '=: (eq * ' , src , ') + (neq * ' , dst , ')'
260  )
261
262  NB. ============================================================
263  NB. Data loading / dumping
264
265  NB. Set field <dst> to the values of variable <src>.
266  NB.
267  NB. Activate(All,-,-,-),
268  NB. pdTransfer(Load, dst, -, flags),
269  LoadField=: 3 : 0
270          'dst src flags'=. y.
271          ". dst , '=: src'
272  )
273
274  NB. Set field <dst> to the contents of file <file>.
275  NB.
276  NB. Activate(All,-,-,-),
277  NB. pdTransfer(Load, dst, -, flags),
278  LoadFile=: 3 : 0
279          'dst file flags'=. y.
280          ". dst , '=: _. ". (1!:1) < jpath file'
281  )
282
283  NB. If <file> -: '', compare field <src> to the contents of
284  NB. file <file> and raise an assertion failure if there is
285  NB. a mitchmatch.
286  NB. Also assign the result of the comparison to the global
287  NB. variable DUMP.
288  NB.
289  NB. Activate(All,-,-,-),
290  NB. pdTransfer(Dump, src, -, flags),
291  DumpFile=: 3 : 0
292          'src file flags'=. y.
293
294          val=. ". src
295          if. -. file -: '' do.
296                  dat=. _. ". (1!:1) < jpath file
297                  DATA=: dat=. ($ val) $ dat
298                  DUMP=: chk=. val = dat
299                  assert. val -: dat
300          end.
301          val
302  )
```

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.

```

        /* 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.

```
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;
```

```
21
22          aop{
23                  /* Linedancer initialization */
24                  Segmentation(ioChannel, -),
25                  /* Load* */
26                  Activate(All,-,-,-),
27                  pdTransfer(Load, 5, 0, -, ro{pdtRdv,sdtRdv}),
28                  /* AssignField */
29                  Activate(All,-,-,-),
30                  Write(Binary, 1, ab, 0),
31                  RTS(Assign, Bitset, -, @{4,10}, @{4,0}, -),
32                  /* Load* */
33                  Activate(All,-,-,-),
34                  pdTransfer(Load, 8, 74, -, ro{pdtRdv}),
35          };
36          for (t = 0; t < RUNS; ++t) {
37                  aop{
38                          /* Xor */
39                          Activate(All,-,-,-),
40                          Write(Binary, 1, ab, 0),
41                          RTS(Xor, Bitset, -, @{1,64}, @{1,0}, @{1,2}),
42                          /* AssignField */
43                          Activate(All,-,-,-),
44                          Write(Binary, 1, ab, 0),
45                          RTS(Assign, Bitset, -, @{4,65}, @{4,1}, -),
46                          /* AssignField */
47                          Activate(All,-,-,-),
48                          Write(Binary, 1, ab, 0),
49                          RTS(Assign, Bitset, -, @{4,0}, @{4,65}, -),
50                          /* AssignField */
51                          Activate(All,-,-,-),
52                          Write(Binary, 1, ab, 0),
53                          RTS(Assign, Bitset, -, @{1,4}, @{1,64}, -),
54                          /* AssignField */
55                          Activate(All,-,-,-),
56                          Write(Binary, 1, ab, 0),
57                          RTS(Assign, Bitset, -, @{10,14}, @{8,74}, -),
58                  };
59                  for (l = 0; l < LABELS; ++l) {
60                          aop{
61                                  /* SelectiveSubValue */
62                                  Activate(All,-,-,-),
63                                  Write(Binary, 0, ab, 0),
64                                  Tag(Binary, l, 4, 10, tr1, -),
65                                  Activate(Matching, -, tr1, -),
66                                  Write(Binary, 1, ab, 0),
67                          };
68                          temp = MU[l];
69                          aop{
70                                  RTS(Sub, Int, -, @{10,14}, @{10,14}, temp),
71                          };
72                  }
73                  aop{
74                          /* Mul */
```

```
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) {
```

```
129                         aop{
130                                 /* FetchNeighbor */
131                                 Activate(All,-,-,-),
132                                 Write(Binary, 1, ab, 0),
133                         };
134                         temp = NBORS[n];
135                         aop{
136                                 RTS(Assign, Bitset, co{Get,temp}, @{10,14}, @{4,10}, -),
137                                 /* Xor */
138                                 Activate(All,-,-,-),
139                                 Write(Binary, 1, ab, 0),
140                                 RTS(Xor, Bitset, -, @{4,14}, @{4,0}, @{4,14}),
141                                 /* SelectiveAddSubValue */
142                                 Activate(All,-,-,-),
143                                 Write(Binary, 1, ab, 0),
144                                 Write(Binary, 0, ab, 1),
145                                 Tag(Binary, 0, 4, 14, tr1, -),
146                                 Activate(Matching, -, tr1, -),
147                                 Write(Binary, 1, ab, 1),
148                                 RTS(AddSub, Int, -, @{20,24}, @{20,24}, BETA),
149                         };
150                 }
151                 aop{
152                         /* Sub */
153                         Activate(All,-,-,-),
154                         Write(Binary, 1, ab, 0),
155                         RTS(Sub, Int, -, @{20,24}, @{20,24}, @{20,82}),
156                         /* LEValue */
157                         Activate(All,-,-,-),
158                         Write(Binary, 1, ab, 0),
159                 };
160                 temp = ALPHAs[t];
161                 aop{
162                         RTS(le, Int, -, @{1,6}, @{20,24}, temp),
163                         /* SelectiveUpdate */
164                         Activate(All,-,-,-),
165                         Write(Binary, 0, ab, 0),
166                         Tag(Binary, 1, 1, 6, tr1, -),
167                         Activate(Matching, -, tr1, -),
168                         Write(Binary, 1, ab, 0),
169                         RTS(Assign, Bitset, -, @{4,10}, @{4,0}, -),
170                 };
171         }
172         aop{
173                 /* AddValue */
174                 Activate(All,-,-,-),
175                 Write(Binary, 1, ab, 0),
176                 RTS(Add, Card, -, @{32,32}, @{4,10}, 0),
177                 /* Dump* */
178                 Activate(All,-,-,-),
179                 pdTransfer(Dump, 4, 10, -, ro{sdtRdv}),
180                 /* Dump* */
181                 Activate(All,-,-,-),
182                 pdTransfer(Dump, 32, 32, -, ro{pdtRdv,sdtRdv}),
```

```
183         };
184 }
```

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.

```
1  aop{InsertImports};
2
3  #sp_include "algorithm.tpl"
4
5  void
6  ismMain(void)
7  {
8          struct ism_parms ismparms;
9          mqId_t sccq, dsmq;
10         int i, ntiles;
11
12         setup_queues(&sccq, &dsmq);
13         mqGet(sccq, &ismparms, WAIT_FOREVER);
14         mqGet(dsmq, &ntiles, WAIT_FOREVER);
15
16         for (i = 0; i < ntiles; ++i) {
17                 algorithm(&ismparms);
18         }
19
20         thrExit(0);
21 }
22
```

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.

```
1  void
2  dsmMain(void)
3  {
4          mqId_t sccq, ismq;
5          uint32 dsmresult = 0;
6          int i, ntiles;
7          sdmcParameters_t sdmc;
8          struct tile_info ti;
9
10         setup_queues(&sccq, &ismq);
11         mqGet(sccq, &dsmparms, WAIT_FOREVER);
12
13         tile_init(&ti,
14             dsmparms.dp_width,
15             dsmparms.dp_height,
16             dsmparms.dp_border,
```

```
17              PDT_RDV | SDT_RDV | D8 | JUMP1 | PACK1);
18          ntiles = tile_count(&ti);
19
20          mqPut(ismq, &ntiles, WAIT_FOREVER);
21          for (i = 0; i < ntiles; ++i) {
22                  upload_seed(dsmparms.dp_block_seed);
23
24                  setup_tile_transfer(&ti, dsmparms.dp_block_in, PDS_LOAD, &sdmc);
25                  sdmcQueueTransfer(OWN_CHANNEL, &sdmc, NULL);
26
27                  setup_tile_transfer(&ti, dsmparms.dp_block_out, PDS_DUMP, &sdmc);
28                  sdmcQueueTransfer(OWN_CHANNEL, &sdmc, NULL);
29
30                  download_debug(dsmparms.dp_block_dbg);
31
32                  tile_next(&ti);
33          }
34
35          sdmcWait(OWN_CHANNEL);
36          mqPut(sccq, &dsmresult, WAIT_FOREVER);
37
38          thrExit(0);
39  }
```

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.

```
1   int
2   main(int argc, char **argv)
3   {
4           mqId_t dsmq[MAX_SDT_CHANNELS];
5           mqId_t ismq[MAX_SDT_CHANNELS];
6           struct ism_parms ismparms;
7           struct dsm_parms dsmparms;
8
9           ld_init();
10
11          attach_queues(dsmq, ismq);
12
13          load_config(&dsmparms, &ismparms);
14          parse_arguments(argc, argv, &dsmparms, &ismparms);
15
16          dsmparms.dp_block_seed = upload_seed();
17          upload_image(inputfile,
18              &dsmparms.dp_width,
19              &dsmparms.dp_height,
20              &dsmparms.dp_chsize,
21              &dsmparms.dp_block_in,
22              &dsmparms.dp_block_out);
23          dsmparms.dp_block_dbg = alloc_debug();
24
```

```
        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:
```

```
19                  return 'ab, ' + str(f.offset - 1000)
20          else:
21                  return str(f.size) + ', ' + str(f.offset)
22
23  # Return string representation for a field (offset, size) pair,
24  # in a RTS command context.
25  # If offset = 100x it is taken as a ab register with size x.
26  def RtsFieldStr(f):
27          if f.offset >= 1000:
28                  return '@{ab,' + str(f.offset - 1000) + '}'
29          else:
30                  return '@{' + str(f.size) + ',' + str(f.offset) + '}'
31
32  # Emit a line at the current indentation level.
33  def Emit(s):
34          global EmitIndent, Output
35
36          Output = Output + EmitIndent * '\t' + s + '\n'
37
38  # Emit the opening of the generated function.
39  def EmitOpen():
40          global FuncName, FuncType, FuncArgs, FuncDefs
41
42          Emit(FuncType)
43          Emit(FuncName + '(' + FuncArgs + ')')
44          Emit('{')
45          IndentOpen()
46          Emit(FuncDefs)
47          EmitAopOpen()
48          Emit('/* Linedancer initialization */')
49          Emit('Segmentation(ioChannel, -),')
50
51  # Emit the closing of the generated function.
52  def EmitClose():
53          EmitAopClose()
54          IndentClose()
55          Emit('}')
56
57  # Open an aop{ ... }; context if we are not already in one.
58  def EmitAopOpen():
59          global InAop
60
61          if not InAop:
62                  Emit('aop{')
63                  IndentOpen()
64                  InAop = True
65
66  # Close an aop{ ... }; context if we are already in one.
67  def EmitAopClose():
68          global InAop
69
70          if InAop:
71                  IndentClose()
72                  Emit('};')
```

```
73                  InAop = False
74
75  # Open a new indentation level.
76  def IndentOpen():
77          global EmitIndent
78
79          EmitIndent = EmitIndent + 1
80
81  # Close current indentation level.
82  def IndentClose():
83          global EmitIndent
84
85          EmitIndent = EmitIndent - 1
86
87  # Emit the opening of a FOR loop.
88  def EmitForOpen(loopvar, count):
89          EmitAopClose()
90          Emit('for (' + loopvar + ' = 0; ' + loopvar + ' < ' + count + '; ++' + loopvar + ') {'
91          IndentOpen()
92
93  # Emit the closing of a FOR loop.
94  def EmitForClose():
95          EmitAopClose()
96          IndentClose()
97          Emit('}')
98
99  # Get the final string representation of value <val> for use in Aop
100 # instructions.
101 # Assignment to a temporary variable for complicated values is emitted if
102 # needed.
103 def AopRep(val):
104         if val.complicated:
105                 v = ComplicatedTemp
106                 EmitAopClose()
107                 Emit(v + ' = ' + val.strrep + ';')
108         else:
109                 v = val.strrep
110         EmitAopOpen()
111         return v
112
113 # Emit a write of value <val> to field <f> for all PEs, taking into account the
114 # difference between CAM and EXT memory.
115 def EmitWriteAll(val, f):
116         v = EmitVal(val)
117         Emit('Activate(All, -, -, -),')
118         if f.offset >= 64:
119                 Emit('Write(Binary, 1, ab, 0),')
120                 Emit('RTS(Assign, Bitset, -, ' + RtsFieldStr(f) + ', ' + v + ', -),')
121         else:
122                 Emit('Write(Binary, ' + v + ', ' + FieldStr(f) + '),')
123
124 # Emit a write of value <val> to field <f> for only the PEs matching tr1,
125 # taking into account the difference between CAM and EXT memory.
126 def EmitWriteMatching(val, f):
```

```
127         v = EmitVal(val)
128         if f.offset >= 64:
129                 Emit("Activate(All, -, -, -),")
130                 Emit("Write(Binary, 0, ab, 0),")
131                 Emit("Activate(Matching, -, tr1, -),")
132                 Emit("Write(Binary, 1, ab, 0),")
133                 Emit("RTS(Assign, Bitset, -, " + RtsFieldStr(f) + ", " + v + ", -),")
134         else:
135                 Emit("Activate(Matching, -, tr1, -),")
136                 Emit("Write(Binary, " + v + ", " + FieldStr(f) + "),")
137 
138 ##
139 ## Emit functions for code templates.
140 ##
141 def EmitMul(dst, src1, src2, type):
142         EmitAopOpen()
143         Emit('/* Mul */')
144         Emit('Activate(All,-,-,-),')
145         Emit('Write(Binary, 1, ab, 0),')
146         Emit('RTS(Mult, ' + type              + \
147              ', -, '      + RtsFieldStr(dst)  + \
148              ', '         + RtsFieldStr(src1) + \
149              ', '         + RtsFieldStr(src2) + \
150              '),')
151 
152 def EmitAddValue(dst, src, delta, type):
153         EmitAopOpen()
154         Emit('/* AddValue */')
155         Emit('Activate(All,-,-,-),')
156         Emit('Write(Binary, 1, ab, 0),')
157         rep = AopRep(delta)
158         Emit('RTS(Add, ' + type             + \
159              ', -, '     + RtsFieldStr(dst) + \
160              ', '        + RtsFieldStr(src) + \
161              ', '        + rep              + \
162              '),')
163 
164 def EmitSub(dst, src1, src2, type):
165         EmitAopOpen()
166         Emit('/* Sub */')
167         Emit('Activate(All,-,-,-),')
168         Emit('Write(Binary, 1, ab, 0),')
169         Emit('RTS(Sub, ' + type              + \
170              ', -, '     + RtsFieldStr(dst)  + \
171              ', '        + RtsFieldStr(src1) + \
172              ', '        + RtsFieldStr(src2) + \
173              '),')
174 
175 def EmitSubValue(dst, src, delta, type):
176         EmitAopOpen()
177         Emit('/* SubValue */')
178         Emit('Activate(All,-,-,-),')
179         Emit('Write(Binary, 1, ab, 0),')
180         rep = AopRep(delta)
```

```
181         Emit('RTS(Sub, ' + type             + \
182              ', -, '     + RtsFieldStr(dst) + \
183              ', '        + RtsFieldStr(src) + \
184              ', '        + rep              + \
185              '),')
186
187 def EmitXor(dst, src1, src2):
188         EmitAopOpen()
189         Emit('/* Xor */')
190         Emit('Activate(All,-,-,-),')
191         Emit('Write(Binary, 1, ab, 0),')
192         Emit('RTS(Xor, Bitset'           + \
193              ', -, '  + RtsFieldStr(dst)  + \
194              ', '     + RtsFieldStr(src1) + \
195              ', '     + RtsFieldStr(src2) + \
196              '),')
197
198 def EmitLEValue(dst, src, val, type):
199         EmitAopOpen()
200         Emit('/* LEValue */')
201         Emit('Activate(All,-,-,-),')
202         Emit('Write(Binary, 1, ab, 0),')
203         rep = AopRep(val)
204         Emit('RTS(le, ' + type             + \
205              ', -, '    + RtsFieldStr(dst) + \
206              ', '       + RtsFieldStr(src) + \
207              ', '       + rep              + \
208              '),')
209
210 def EmitAssignField(dst, src):
211         EmitAopOpen()
212         Emit('/* AssignField */')
213         Emit('Activate(All,-,-,-),')
214         Emit('Write(Binary, 1, ab, 0),')
215         Emit('RTS(Assign, Bitset'      + \
216              ', -, ' + RtsFieldStr(dst) + \
217              ', '    + RtsFieldStr(src) + \
218              ', -),')
219
220 def EmitAssignValue(dst, val):
221         EmitAopOpen()
222         Emit('/* AssignField */')
223         Emit('Activate(All,-,-,-),')
224         Emit('Write(Binary, 1, ab, 0),')
225         rep = AopRep(val)
226         Emit('RTS(Assign, Bitset'      + \
227              ', -, ' + RtsFieldStr(dst) + \
228              ', '    + rep              + \
229              ', -),')
230
231 def EmitFetchNeighbor(dst, src, nbor):
232         EmitAopOpen()
233         Emit('/* FetchNeighbor */')
234         Emit('Activate(All,-,-,-),')
```

```
235         Emit('Write(Binary, 1, ab, 0),')
236         rep = AopRep(nbor)
237         Emit('RTS(Assign, Bitset, co{Get,' + rep + \
238              '}, ' + RtsFieldStr(dst)           + \
239              ', '  + RtsFieldStr(src)           + \
240              ', -),')
241
242 def EmitSelectiveSubValue(dst, src, cond, val, delta, type):
243         EmitAopOpen()
244         Emit('/* SelectiveSubValue */')
245         Emit('Activate(All,-,-,-),')
246         Emit('Write(Binary, 0, ab, 0),')
247         rep = AopRep(val)
248         Emit('Tag(Binary, ' + rep           + \
249              ', '          + FieldStr(cond) + \
250              ', tr1, -),')
251         Emit('Activate(Matching, -, tr1, -),')
252         Emit('Write(Binary, 1, ab, 0),')
253         rep = AopRep(delta)
254         Emit('RTS(Sub, ' + type            + \
255              ', -, '     + RtsFieldStr(dst) + \
256              ', '        + RtsFieldStr(src) + \
257              ', '        + rep              + \
258              '),')
259
260 def EmitSelectiveAddSubValue(dst, src, cond, val, delta, type):
261         EmitAopOpen()
262         Emit('/* SelectiveAddSubValue */')
263         Emit('Activate(All,-,-,-),')
264         Emit('Write(Binary, 1, ab, 0),')
265         Emit('Write(Binary, 0, ab, 1),')
266         rep = AopRep(val)
267         Emit('Tag(Binary, ' + rep + ', ' + FieldStr(cond) + ', tr1, -),')
268         Emit('Activate(Matching, -, tr1, -),')
269         Emit('Write(Binary, 1, ab, 1),')
270         rep = AopRep(delta)
271         Emit('RTS(AddSub, ' + type            + \
272              ', -, '        + RtsFieldStr(dst) + \
273              ', '           + RtsFieldStr(src) + \
274              ', '           + rep              + \
275              '),')
276
277 def EmitSelectiveUpdate(dst, src, cond, val):
278         EmitAopOpen()
279         Emit('/* SelectiveUpdate */')
280         Emit('Activate(All,-,-,-),')
281         Emit('Write(Binary, 0, ab, 0),')
282         rep = AopRep(val)
283         Emit('Tag(Binary, ' + rep           + \
284              ', '          + FieldStr(cond) + \
285              ', tr1, -),')
286         Emit('Activate(Matching, -, tr1, -),')
287         Emit('Write(Binary, 1, ab, 0),')
288         Emit('RTS(Assign, Bitset, -, ' + RtsFieldStr(dst) + \
```

```
289              ', '                        + RtsFieldStr(src) + \
290              ', -),')
291
292  def EmitLoad(dst, flags):
293          EmitAopOpen()
294          Emit('/* Load* */')
295          Emit('Activate(All,-,-,-),')
296          Emit('pdTransfer(Load, ' + FieldStr(dst) + \
297               ', -, '             + flags        + \
298               '),')
299
300  def EmitDump(src, flags):
301          EmitAopOpen()
302          Emit('/* Dump* */')
303          Emit('Activate(All,-,-,-),')
304          Emit('pdTransfer(Dump, ' + FieldStr(src) + \
305               ', -, '             + flags        + \
306               '),')
307  %%
308
309  ##
310  ## The parser
311  ##
312
313  parser Linedancer:
314          token END:      '$'
315          token NAME:     '[a-zA-Z][a-zA-Z0-9]*'
316          token NUM:      '[0-9]+'
317          token ID:       '[a-zA-Z_][a-zA-Z0-9_]*'
318          token STR:      '[^\']+'
319
320          ignore:         'NB\..*\n'
321          ignore:         '[ \n\t]+'
322
323          rule program:   {{ EmitOpen()  }} command* {{ EmitClose() }}
324                          END {{ return Output }}
325
326          rule command:   'Mul' field  {{ dst  = field }}
327                          ';'   field  {{ src1 = field }}
328                          ';'   field  {{ src2 = field }}
329                          ';'   string {{ type = string }}
330                          {{ EmitMul(dst, src1, src2, type) }}
331                  |       'AddValue' field  {{ dst   = field }}
332                          ';'        field  {{ src   = field }}
333                          ';'        expr   {{ delta = expr }}
334                          ';'        string {{ type  = string }}
335                          {{ EmitAddValue(dst, src, delta, type) }}
336                  |       'Sub' field  {{ dst  = field }}
337                          ';'   field  {{ src1 = field }}
338                          ';'   field  {{ src2 = field }}
339                          ';'   string {{ type = string }}
340                          {{ EmitSub(dst, src1, src2, type) }}
341                  |       'SubValue' field  {{ dst   = field }}
342                          ';'        field  {{ src   = field }}
```

```
343                      ';'       expr   {{ delta = expr }}
344                      ';'       string {{ type  = string }}
345                      {{ EmitSubValue(dst, src, delta, type) }}
346            |         'Xor' field {{ dst  = field }}
347                      ';'   field {{ src1 = field }}
348                      ';'   field {{ src2 = field }}
349                      {{ EmitXor(dst, src1, src2) }}
350            |         'LEValue' field  {{ dst  = field }}
351                      ';'       field  {{ src  = field }}
352                      ';'       expr   {{ val  = expr }}
353                      ';'       string {{ type = string }}
354                      {{ EmitLEValue(dst, src, val, type) }}
355            |         'AssignField' field {{ dst = field }}
356                      ';'           field {{ src = field }}
357                      {{ EmitAssignField(dst, src) }}
358            |         'AssignValue' field {{ dst = field }}
359                      ';'           expr  {{ val = expr }}
360                      {{ EmitAssignValue(dst, val) }}
361            |         'FetchNeighbor' field {{ dst  = field }}
362                      ';'             field {{ src  = field }}
363                      ';'             expr  {{ nbor = expr }}
364                      {{ EmitFetchNeighbor(dst, src, nbor) }}
365            |         'SelectiveSubValue' field  {{ dst   = field }}
366                      ';'                 field  {{ src   = field }}
367                      ';'                 field  {{ cond  = field }}
368                      ';'                 expr   {{ val   = expr }}
369                      ';'                 expr   {{ delta = expr }}
370                      ';'                 string {{ type  = string }}
371                      {{ EmitSelectiveSubValue(dst, src, cond, val, delta, type) }}
372            |         'SelectiveAddSubValue' field  {{ dst   = field }}
373                      ';'                    field  {{ src   = field }}
374                      ';'                    field  {{ cond  = field }}
375                      ';'                    expr   {{ val   = expr }}
376                      ';'                    expr   {{ delta = expr }}
377                      ';'                    string {{ type  = string }}
378                      {{ EmitSelectiveAddSubValue(dst, src, cond, val, delta, type) }}
379            |         'SelectiveUpdate' field {{ dst  = field }}
380                      ';'               field {{ src  = field }}
381                      ';'               field {{ cond = field }}
382                      ';'               expr  {{ val  = expr }}
383                      {{ EmitSelectiveUpdate(dst, src, cond, val) }}
384            |         'LoadField' field  {{ dst   = field }}
385                      ';'         expr
386                      ';'         string {{ flags = string }}
387                      {{ EmitLoad(dst, flags) }}
388            |         'LoadFile' field  {{ dst   = field }}
389                      ';'        string
390                      ';'        string {{ flags = string }}
391                      {{ EmitLoad(dst, flags) }}
392            |         'DumpFile' field  {{ src = field }}
393                      ';'        string
394                      ';'        string {{ flags = string }}
395                      {{ EmitDump(src, flags) }}
396            |         for_loop
```

```
397
398         rule for_loop:  'for_'
399                         ID {{ loopvar = ID }}
400                         '\.' 'i\.' expr
401                         'do\.' {{ EmitForOpen(loopvar, expr.strrep) }}
402                         command*
403                         'end.' {{ EmitForClose() }}
404
405         rule expr:      operand {{ val = operand }}
406                         [
407                             '{' ID
408                             {{ val = Value(True, ID + '[' + operand.strrep + ']') }}
409                         ] {{ return val }}
410                 |       '-' expr {{ return Value(True, '-' + operand.strrep) }}
411
412         rule operand:   NUM            {{ return Value(False, NUM) }}
413                 |       ID             {{ return Value(False, ID)  }}
414                 |       '\(' expr '\)' {{ return expr              }}
415
416         rule field:     "'" NAME '_'
417                         NUM {{ offset = int(NUM) }}
418                         '_'
419                         NUM {{ size = int(NUM) }}
420                         "'"
421                         {{ return Field(offset, size) }}
422
423         rule string:    "'"  {{ s = '' }}
424                         [ STR {{ s = STR }} ]
425                         "'"
426                         {{ return s }}
427
428 %%
```