Model-based Application Development for Massively Parallel Embedded Systems

Jan W.M. Jacobs

Members of the dissertation committee:

prof. dr. ir.	G.J.M. Smit	University of Twente (promoter)
dr. ir.	J. Kuper	University of Twente (assistent-promoter)
prof. dr. ir.	Th. Krol	University of Twente
prof. dr. ir.	M. Aksit	University of Twente
prof. dr.	H. Corporaal	University of Eindhoven
ir.	P.G. Jansen	University of Twente
dr. ir.	P.J. Mosterman	The MathWorks, Inc.
prof. dr. ir.	A.J. Mouthaan	University of Twente (chairman and secretary)

CTIT Ph.D. thesis Series No. 08-132
Centre for Telematics and Information Technology (CTIT)
P.O. Box 217 - 7500 AE Enschede - The netherlands

Typeset with LaTeX.
Printed by Océ Technologies BV, Venlo, The Netherlands.

ISBN 978-90-365-2752-1
ISSN 1381-3617 (CTIT Ph.D.-thesis series no. 08-132)

MODEL-BASED APPLICATION DEVELOPMENT FOR MASSIVELY PARALLEL EMBEDDED SYSTEMS

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof. dr. W.H.M. Zijm,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, November 20, 2008 at 16.45

by

Johannes Wilhelmus Maria Jacobs

born on 30 April 1955,
in Kessel (LB), The Netherlands

This dissertation is approved by:

prof. dr. ir.	G.J.M. Smit	(promoter)	and
dr. ir.	J. Kuper	(assistant-promoter)	

Abstract

The development of embedded systems in information-rich contexts is governed by some intertwined trends. The increase of both volume of data to be processed and the related processing functionality feeds the growing complexity of applications. Independently, the processing hardware that is needed to process these applications, is becoming increasingly parallel and heterogeneous (many-core) because of performance and power problems. Furthermore, today's compiler technology is not able to translate sequential legacy code for multi-core or many-core systems in an efficient way.

This thesis addresses the problem of generating efficient code for a number of cores, that operate synchronously. Examples are Single Instruction Multiple Data (SIMD) and Very Long Instruction Word (VLIW) architectures. In this thesis we restrict ourselves to architectures that include a control processor that provides the instruction stream.

In practice the manufacturers of such many-core processors only provide a C-compiler that supports hardware intrinsic instructions. This situation usually requires manual adaptation of sequential code. Unfortunately, a first feedback of the implementation on the targeted parallel architecture only comes late in the development trajectory. Moreover, during implementation phases more engineers enter the project and this increases the risk of early errors proliferating to later phases. Although some parts of the system can be modelled in high level language(s) (e.g., MATLAB), the typical approach lacks a single integral and executable framework allowing for an immediate system-wide verification.

This thesis proposes an integral design methodology, named IRIS, for the development of firmware for many-core architectures.

The methodology is illustrated by three cases: a colour image processing pipeline for a printer, stochastic image quantisation, and data mining of dynamic document spaces. For the three cases the various development phases and the associated development roles result in mathematical models, that can be directly

transcribed in a functional language. The executable models are subsequently transformed into a series of implementation models, that converge to the targeted many-core implementation.

This thesis contains the following contributions:

First, all three cases showed that for an effective and efficient implementation of applications on a massively parallel processing architecture it is necessary to manually (re)model the problem in a suitable parallel representation.

Second, a semi-automatic and interactive development process is needed for mapping an application on a dedicated massively parallel processing core.

Third, the three cases demonstrate that a single architectural language – firmly based on mathematics – for all development phases, reduces development time and reduces the number of design errors.

Fourth, it is shown that the relevant extra-functional requirements can be handled by integrating them into the regular functional flow. As a consequence the architectural language should support *in situ* monitoring and visualisation of quantifiable extra-functional properties.

Fifth, in the development process small steps and immediate feedback are crucial as demonstrated by the various performed iterations (optimisations, correction of errors) and the involved design space explorations.

Sixth, it is shown that a development process having a phased approach works very well. This should subsequently include:

1. a familiarisation phase with respect to the problem and the target hardware architecture(s),
2. an incremental prototyping phase (hardware architecture independent), and
3. a transformational development phase (hardware architecture dependent),

which are performed in an iterative manner when needed.

Samenvatting

De ontwikkeling van embedded systemen in informatie-intensieve omgevingen wordt bepaald door enkele met elkaar verbonden trends. De groei van zowel volume van data alsook de gerelateerde processing funktionaliteit, voeden de groeiende komplexiteit van applikaties. Onafhankelijk daarvan ontwikkelt de voor de applikaties benodigde processing hardware zich vanwege prestatie en vermogen steeds meer in de richting van meerdere parallelle en heterogene cores. Daar komt nog bij dat de huidige compiler-technologie niet geschikt is om de bestaande sequentiële source code te vertalen in een efficiënte implementatie voor multi- of many-core systemen.

Dit proefschrift gaat over het probleem van de generatie van efficiënte code voor een aantal cores die synchroon samenwerken. Voorbeelden zijn Single Instruction Multiple Data (SIMD) en Very Long Instruction Word (VLIW) architekturen. In dit proefschrift beperken we ons tot architekturen die een besturingsprocessor hebben voor de benodigde instruktie-stroom.

In de praktijk leveren de producenten van dergelijke many-core processoren slechts een C-compiler die hardware-afhankelijke instrukties ondersteunt. Deze situatie vereist gewoonlijk een handmatige aanpassing van de sequentiële code. In deze aanpak is het echter pas op een laat tijdstip mogelijk om een eerste terugkoppeling te geven over de implementatie op de beoogde parallelle hardware architektuur. Bovendien neemt gedurende de implementatie-fase de instroom van engineers in het projekt toe en dit verhoogt het risiko van de proliferatie van vroege fouten naar latere fasen. Ofschoon delen van het systeem kunnen worden gemodelleerd in een hoog-nivo taal (b.v. MATLAB), ontbeert de typische aanpak toch een integraal en executeerbaar raamwerk dat een instantane systeem-brede verifikatie mogelijk maakt.

Dit proefschrift levert een integrale ontwerp-methodologie, genaamd IRIS, voor de ontwikkeling van firmware voor many-core architekturen.

De methodologie wordt geïllustreerd door drie praktijkvoorbeelden: een beeld-

verwerkingspipeline voor een kleurenprinter, stochastische beeldkwantisatie, en data mining van dynamische dokumentkollekties. De diverse ontwikkelfasen en hun overeenkomstige ontwikkelrollen resulteren voor al deze drie praktijk-voorbeelden in wiskundige modellen, die op hun beurt direkt kunnen worden overgezet in een funktionele taal. Deze uitvoerbare modellen worden achtereenvolgens omgezet in een reeks implementatie-modellen, die uiteindelijk konvergeren naar de beoogde many-core implementatie.

Het proefschrift bevat de volgende bijdragen:

Ten eerste, alle drie praktijkvoorbeelden laten zien dat het noodzakelijk is om het probleem met de hand te (her)modelleren in een geschikte parallelle representatie, ten einde een effektieve en efficiënte implementatie van de toepassing op een massief-parallel verwerkingsarchitektuur te verkrijgen.

Ten tweede, voor het afbeelden van een toepassing op een specifieke massief-parallelle verwerkingskern is een semi-automatisch en interactief ontwikkelproces nodig.

Ten derde, de drie praktijkvoorbeelden demonstreren dat voor alle ontwikkelfasen, een enkele architektuurtaal – direkt gebaseerd op de wiskunde – zowel de ontwikkeltijd alsook het aantal gemaakte ontwerpfouten terugbrengt.

Ten vierde wordt aangetoond dat de relevante extra-funktionele eisen kunnen worden afgehandeld door deze te integreren in de reguliere funktionele beschrijving. Een direkt gevolg hiervan is dat de architektuurtaal de *in situ* monitoring and visualisatie van meetbare extra-funktionele eigenschappen moet ondersteunen.

Ten vijfde, in het ontwikkelproces zijn het maken van klein stappen en instantane terugkoppeling van cruciaal belang zoals gedemonstreerd door de verscheidene uitgevoerde iteraties (optimalisaties, korrekties van fouten) en de betrokken verkenning van de ontwerpruimte.

Tenslotte wordt aangetoond dat een gefaseerde aanpak van het ontwikkelproces goed werkt. De fasen zijn achtereenvolgens:

1. een kennismakingsfase aangaande het probleem en de beoogde hardware architektu(u)r(en),
2. een incrementele prototyping fase (hardware-architektuur onafhankelijk), en
3. een transformationele ontwikkelfase (hardware-architektuur afhankelijk).

De fasen worden naar behoefte op een iteratieve wijze uitgevoerd.

Acknowledgements

Each end goes with a begin. The start of this enterprise was laid during my study at the Technische Hogeschool Eindhoven (TUE); it seemed to be quite a nice challenge to also become a "doctor" one time. Never give up your dreams!

During the development of microcode for the first commercial laserprinter of Océ in the mid 80s, unknowingly a first step was made in the selection of a theme (a methodology) for this PhD. This work was conducted together with Roger Hacking. Roger, thanks for your support.

One of the next steps was the selection of a suitable research group and professor, and mid 90s I met Thijs Krol at the University of Twente (UT). Although the meeting did not result in concrete plans, it led to the right scientific place and a free meal in the Bastille. Thijs, thanks for the good advice.

It was for Roelof Hamberg, that achieving a doctor's degree – as part of a liaison assignment with the UT – became a topic within Océ. Roelof, many thanks for your belief in me. I learned that communication of one's dreams is necessary before any strategic enterprise can start!

Now, almost at the end of this enterprise, I want to express my gratitude to Gerard Smit and Jan Kuper. Gerard's constant commitment, notably the conscientiously reviewing of texts (papers, thesis) leading to to-the-point criticism and – most of all – the encouraging way of coaching, made my PhD study both a challenging and a rewarding process. Jan showed me how to (re)model a problem in such a way that its mathematical description can be elegantly transcribed in a functional language. From him I also painfully learned never to bet with a mathematician. Gerard and Jan form a good complementary team in which global as well as more detailed concerns balance well.

The foundation of the work on IRIS are the three application cases. The cases could not have been conducted successfully without the assistance of Winston Bond (Aspex) and master students Rui Dai (University of Singapore) and Leroy van Engelen (UT). Winston, Rui and Leroy, thank you very much for your effort

and the many extra hours you have spent in analysing, designing and coding in Aspro-C and in the 'dreadful' J language! Roel Pouls, Samuel Driessen and Zoé Goey provided me real challenging problem cases, and gave constructive feedback on the quality of the work. Roel, thanks for introducing me into and guiding through the world of productive image processing for a colour printer and Field Programmable Gate Array (FPGA) based system design. Zoé, thank you very much for the attentive guidance through Markov Random Fields and the various critical remarks on the involved mathematical modeling. Samuel, your help in the world of natural language processing, data mining and knowledge discovery for news articles is very much appreciated. A lot of people supported the work on the cases in a direct way. I want to thank (in arbitrary order): Stuart Cornell (Aspex), Sebastian de Smet, Jos Nelissen, Rob Audenaerde, Harold van Garderen, Andras Zolnay and Anjo Anjewierden (VU) for their contribution.

A very special word of thanks goes to Klaas Kuin, for guiding me through the rough waters of writing a thesis. His typical way of giving constructive critism on one hand and offering opportunities for me to discover in the other, not only helped me finishing this work in time but also provided me with wise lessons for my future life. Klaas, thank you for your coaching.

Co-readers are appreciated, in particular when massive amounts of English-like (euphemism) texts are being generated. I want to thank the following people for proofreading and other kinds of support: Marco Krom, Lou Somers, Herman Driessen, Jos Kerkhoffs, Jack van der Elsen, Waldo Ruiterman, Aart van Meeteren, Jorrit Buurman, Kees-Jan Sonnenberg, Dion Slijp and Paul Verhelst.

You never work alone, and the following persons provided me with social context. First I want to express my thanks to Rob van den Tillaart for the many techno-philosophical and creative discussions (and U-memos) and Joost Meijer for the many exciting applied AI thoughts that were exchanged. You both provided me with an enjoyable research context. Thanks also to Juri Snijders, Eric Dortmans, Jan Beckers, Mechlin Pelders, Rokus Visser, Peter van den Bosch, Bart Verheijen, Matthijs Mullender, Dirk Schäfer, Rogier de Blok, Guus Muisers and Josse van der Plaat. Furthermore the advise of the young but experienced 'flying doctors' Jaap de Jong, Aico Troeman and Bart van As was very comforting. Thank you all for being my roommates in Venlo!

Once a week I visit Twente, where I am lucky to be part of a pleasant social matrix too. I want to thank Andre Kokkeler, Gerard Rauwerda, Bert Molenkamp, Hans Scholten, Paul Havinga, Berend-Jan van der Zwaag, Pascal Wolkotte, Philip Hölzenspies and all other staff-members, AIOs and Master students of the UT/EWI CAES group (too many to mention them all) for providing a challenging but at the same time comforting environment.

I also want to thank Marlous Weghorst, Nicole Baveld and Thelma Nordholt and their Venlo counterparts Petra van der Heijden and Bianca Meijers for all the secretarial work. You are the real motors in organisations, many thanks for your support.

This enterprise could not end successfully without the unconditional support

of my family. I want to thank my parents for their continuous support for my personal development. I want to thank my children, Marcia and Jorn, and Robert for their love and understanding, in particular the many times that my thoughts drifted away during the very scarce moments we were together. But most of all, I want to thank my wife Marja for all her love, understanding and patience with my peculiar way of being. Marja, you not only tolerated my frequent absence but also took over a lot of my domestic duties, despite your own full time job. Without you I would never have accomplished this work.

TABLE OF CONTENTS

List of Acronyms

AGU	Address-Generation Unit
ALU	Arithmetic and Logical Unit
AMD	Advanced Micro Devices
ANN	Artificial Neural Network
ANSI	American National Standards Institute
APL	A Programming language
ASIC	Application Specific Integrated Circuit
ASP	Associative String Processor
BBC	British Broadcasting Corporation
BMU	Best Matching Unit
CAM	Content Addressable Memory
CCU	Communication and Configuration Unit
CM	Configuration Manager
CMOS	Complementary MetalOxideSemiconductor
CMYK	Cyan, Magenta, Yellow, and blacK
CNAPS	Co-processing Node Architecture for Parallel Systems
CNN	Cable News Network
COCOMO	COnstruction COst MOdel
CSDF	Cyclo-Static DataFlow
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
dpi	dots per inch
DRAM	Dynamic Random Access Memory
DSL	Domain Specific Language
DSM	Domain Specific Modelling

DSP	Digital Signal Processor
DSRC	Domain Specific Reconfigurable Core
EBM	Energy-Based Model
EXT	EXTended memory
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
GB	Giga Byte
GHz	Giga Hertz
GOPS	Giga Operations Per Second
GPP	General Purpose Processor
HDL	Hardware Description Language
HP	Hewlett-Packard
HTML	HyperText Markup Language
HUT	Helsinki University of Technology
HVS	Human Visual System
IBM	International Business Machines
IDF	Inverse Document Frequency
IIR	Infinite Impulse Response
ILP	Instruction Level Parallelism
IO	Input / Output
IT	Information Technology
KB	Kilo Byte
KLOC	thousand Lines Of Code
KPN	Kahn Process Network
LFSR	Linear Feedback Shift Register
LOC	Lines Of Code
lsb	least significant bit(s)
LUT	Look Up Table
MB	Mega Byte
MBD	Model-Based Design
MC-SoC	Many-Core System-on-Chip
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MHz	Mega Hertz
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instructions Per Second
ML	Meta-Language
MMD	Modified Metropolis Dynamics

MMU	Memory Management Unit
MP3	MPEG-1 Audio Layer 3
MPEG	Moving Pictures Experts Group
MRF	Markov Random Field
msb	most significant bit(s)
MT	Memory Tile
NLP	Natural Language Processing
NML	Native Mapping Language
NoC	Network on Chip
PAC	Processing Array Cluster
PAE	Processing Array Element
PDA	Personal Digital Assistent
PDS	Primary Data Store
PDT	Primary Data Transfer
PE	Processing Element
PPA	Processing Part Array
ppm	pages per minute
QE	Quantisation Error
QoS	Quality of Service
RAM	Random Access Memory
RGB	Red Green Blue
RISC	Reduced Instruction Set Computer
RIP	Raster Image Processing
RSS	Really Simple Syndication
RTS	Run Time Support
SCM	Supervising Configuration Manager
SDS	Secondary Data Store
SDT	Secondary Data Transfer
SIMD	Single Instruction Multiple Data
SISO	Single Input Single Output
SF	Software Factory
SMP	Symmetric Multi-Processing
SoC	System-on-Chip
SOM	Self Organising Map
SPARC	Scalable Performance ARChitecture
SQL	Structured Query Language
SSE2	Streaming SIMD Extensions 2
SSM	Soft System Methodology
SRAM	Static Random Access Memory
SVG	Scalable Vector Graphics

TDS	Ternary Data Store
TDT	Ternary Data Transfer
TE	Topology Error
TF	Term Frequency
TLB	Translation Lookaside Buffer
TP	Tile Processor
UML	Unified Modeling Language
URL	Uniform Resource Locator
VHDL	Very-high-speed integrated circuits Hardware Description Language
VLIW	Very Long Instruction Word
WiMAX	Worldwide interoperability for Microwave Access
XOR	Exclusive OR
XP	Extreme Programming
XPP	eXtreme Processing Platform
XSLT	eXtensible Stylesheet Language Transformations

CHAPTER 1

Introduction

1.1 Introduction

This thesis is concerned with the development of embedded systems in information-rich contexts such as document processing for offices. Two intertwined trends play a role in the development of such systems. One is the unabatingly growing complexity of applications and the other the advance of powerful and often massively parallel embedded computer architectures. Combined, the trends cause a significant increase in the complexity of embedded systems and pose new challenges for the development of embedded software (*firmware*).

The goal of this chapter is to anchor firmware development for many-core[1] processors in tomorrow's document processing products and services. We do that by departing from a personal vision on document processing[2], that envisions tomorrow's computing demands. The trends on four computation-related aspects of this vision are mentioned and related to each other: content, hardware, software, and products & services. The latter links business to the first three aspects (Section 1.2.1). The mutual confrontation of these aspects motivates the importance of improved firmware development for embedded systems and cumulates in a problem description (Section 1.3). Finally, the structure of the thesis is given in Section 1.4.

[1] A core is an independent processing entity containing at least a control unit and one or more execution units.

[2] This personal vision is not the official vision of Océ.

1.2 Document processing in a changing world

The purpose of this section is to create a possible future scenario of document processing in the office in which trends in four aspects, *content*, *hardware*, *software*, and *products & services*, are related to each other.

1.2.1 Vision

Document processing follows the changing information flows in the world. One of the dominant media still is *paper*, but it is used differently nowadays. We like to use paper for a short time and then dispose it rather than use it for archiving purposes [105]. An alternative is a *digital medium*. Digital information can be processed, not only by humans, but also by intelligent software. The Semantic Web[3] for example, is an evolving extension of the World Wide Web in which the semantics of information and services on the web is defined, and which implements inference engine(s) and ontologies that cover the basic domains of human knowledge.

We have chosen for the *semantic copier* to play a central role in our vision. The semantic copier is a fictive extension of the basic copier (yellow parts), in Figure 1.1. The copier model is chosen because it is the most simple transformer that involves *input* → *processing* → *output* in a feedback loop that is being closed by a user. The goal of the semantic copier is to reduce the information burden of an office professional by processes with autonomous and proactive behaviour, based on knowledge of the context of the user (awareness). We will subsequently describe the semantic copier concept and the technologies that will be used to build it. At the left-hand side in Figure 1.1 the copier obtains its input, and after processing the output is generated (at the right-hand side). The vertical axis represents the projected developments over time. Possible emerging behaviour of such a copier includes summarisation (the act of preparing a summary), translation (e.g., Chinese → English), and even behaviours that support the decision making processes of professionals, as demonstrated in Apple's Knowledge Navigator[4]. Obviously these behaviours need – besides a thorough analysis and directed synthesis of the output – general world knowledge such as generally known objects as persons, buildings, and cities. At the left-hand side sensors feed the analysis, and symmetrically, at the right-hand side composed texts are output, for example, printed or articulated (speech) or in other ways. Note that the various parts of the semantic copier may be distributed to different locations and used at different points in time.

As an example we take the translation of an audio text and we will follow the stream of information from the origin (left) to the destination (right). The text

[3] Tim Berners-Lee et al: "The Semantic Web.", Scientific American, May 2001, `http://www.sciam.com/print_version.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21` .

[4] Apple Computer Inc.: "The Knowledge Navigator.", 1987, `video.google.com/videoplay?docid=-5144094928842683632`.

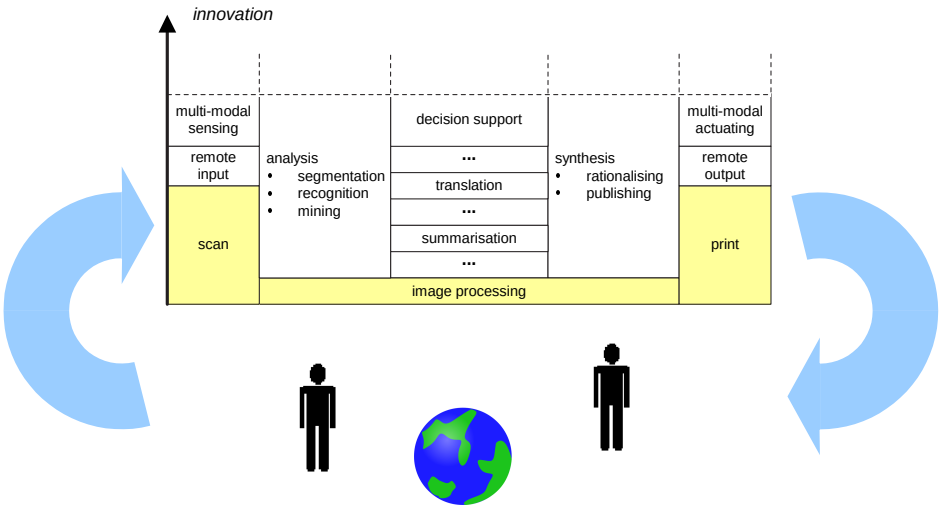


Figure 1.1: The *semantic copier* is a fictive but inspiring extension of the basic copier (yellow shaded parts). The ultimate goal is to realise the equivalent of a knowledgeable conversational agent as featured by Apple's Knowledge Navigator.

stream is entered via an audio channel, for instance in a MPEG-1 Audio Layer 3 (MP3) format, and is subsequently syntactically and semantically analysed. The translation involves world knowledge, that is for example needed for translation rules, dictionary lookups, and to resolve implicit references to generally known objects as persons, buildings, cities etc. Before the output is published (e.g., printed), it is composed according to the grammar of the destination language.

In our eyes a mixture of old and new technologies will be used to realise the required intelligence of this complex system. The older technologies such as image processing, natural language technology, and inference engines, process their input and are not directly influenced by the effect their output has on the outside world. However, new technologies better exploit the environment the system is in. Since the embedded system is situated in a physical environment, it is possible to set up a feedback loop in which the immediate user, the near environment, and even the world (internet) participate, see Figure 1.1. The system's output induces actions of a user or reactions in the (near) environment, and when those are fed back to the input side the system learns to adapt old behaviour or even learns to develop new behaviour.

In our opinion, developments in for example embodied intelligence [96] and co-evolution [71] show the way to this *emergent behaviour*. Emergent behaviour refers to the way complex systems and patterns arise out of a multiplicity of relatively simple interactions. It is behaviour that is not specified as such but emerges from a carefully set up optimisation process. The objective of these new approaches is

to write less and simpler code for setting up this optimisation process and train the behaviour rather than to program its functionality in an explicit manner. As a result the system obtains a smarter behaviour at lower development effort.

To allow for a good awareness of the environment, the semantic copier changes its physical appearance with respect to the basic copier. The point of service is not restricted to the location of the hardware anymore. The input and output are detached from the copier (today mostly positioned at a corridor or mail room) and are moved closer to the working desk. The same applies for the processing, that is integrated in the commodity IT infrastructure, leaving the bare scan and print unit in its familiar environment. Also at the processor level a behaviour-oriented approach is visible. For example Intel describes its "Recognition, Mining and Synthesis scenario"[5] a processing platform for the 2015 workload model. The platform supports a kind of sense-think-act behaviour: recognition (*what is*?), mining (*is this*?), and synthesis (*what if*?).

To conclude, in our vision the semantic copier is a system that realises intelligent behaviour in two complementary ways. First, it transforms its input data to the demanded value added output. Second, it is aware of the immediate context of the user – partly influenced by its own output – and can act autonomously on it, thereby realising desired behaviours e.g., adaptivity.

1.2.2 Trends

In the above vision four aspects, that play a role in building the semantic copier, can be identified. These four aspects are: content, hardware, software, and products & services. The purpose of this section is to describe the autonomous trends of these four aspects, and to prepare for their relations in embedded systems design (Section 1.2.3).

Content. According to Gulli [57] the indexable web (2005) is larger than 11 billion pages. Market research institute IDC estimates the 'digital universe' to be 161 billion gigabytes in 2006 and projects a six-fold increase by 2010[6]. The *usage*[7] of the web is still growing. An integral growth of 305% over the past 8 years is reported[8], and some 'less developed' continents (Africa, Middle East, Latin America) note already an average rate of more than 100% growth per year. To summarise, the amount of processable information is extremely large and is growing each day.

[5] Pradeep Dubey: "A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera.", 2005, `http://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf`.

[6] Frederick Lane: "IDC: World Created 161 Billion Gigs of Data in 2006.", 2007, `http://www.toptechnews.com/story.xhtml?story_id=01300000E3D0` .

[7] Internet World Stats defines usage by a person who has available access to an internet connection point and has the basic knowledge required to use web technology.

[8] Miniwatts marketing group: "Internet world stats: Usage and population statistics.", 2008, `http://www.internetworldstats.com/stats.htm`.

Information, that is acted upon, has shorter lifetimes. News is for example a typical information category that can influence decision makers directly. Most news items have a very short lifetime, but a few continue to be accessed well beyond their initial release. The average halftime of a news document is 36 hours[9]. For streaming information this can be reduced even further, down to the time of a single video frame (msec range).

A pressing problem is information overload. Information overload refers to the state of having too much information to make a decision on or remain informed about a topic. See also Chapter 6.

Hardware. For processor developments, Moore's law – originally formulated in 1965 [90] – still holds[10]. However, because of power dissipation, the single-core processor is replaced by a multi-core processor. To even further optimise the *computational efficiency* (performance-power dissipation ratio expressed in [MIPS/Watt]) heterogeneous System-on-Chips (SoCs), or many-cores, have been developed [63][39].

Besides a scalable processor and memory architecture, many-core SoCs also have a scalable communication bandwidth architecture [63]. For example the chip implementation of the IBM Cell employs multiple ring networks to interconnect the nine processors on the chip [74].

The size of transistors is decreasing, so does the cost per transistor. However, the manufacturing expenses per unit area has increased over time, since materials and energy expenditures per unit area have only increased with each successive integration technology. Large enough series keep the cost stable over time; in practice this means that the consumer gets 'more for the same price'.

Software. The major trend is that the complexity of software increases each year [64], and thus increases the existing *software crisis*. The software crisis was a term used in the early days of software engineering, before it was a well-established subject. The term was used to describe the impact of rapid increases in computer power and the complexity of the problems that could be tackled. In essence, it refers to the difficulty of writing correct, understandable, and verifiable computer programs.

Complexity emerges in many ways. We mention here the excessive development effort and the inherently weak performance of sequential processing. In particular for embedded systems the excessive development time has even more impact because of the many extra concerns that have to be dealt with. For ex-

[9] Z. Dezso et al: "Fifteen Minutes of Fame: The Dynamics of Information Access on the Web.", 2005, `http://www.citebase.org/abstract?id=oai:arXiv.org:physics/0505087`.

[10] Michael Kanellos: "Moore's law to roll on for another decade", 2003, `http://news.cnet.com/2100-1001-984051.html`.

ample in the automotive industry[11] the increasingly complex embedded systems have led to disappointment as cars are delivered to the market with software and electronic defects. Warranty costs are on the rise as brand perception suffers.

Traditional optimisation techniques are based on order of complexity reduction, that work for sequential processing but not for parallel processing. These techniques tend to introduce dependencies and new data structures that complicate parallelisation; programs are getting so immensely large that it is not feasible to unravel these dependencies.

Driven by the increasing performance demands, the transition of a single-core processor to parallel many-core SoCs, adds new problems. Compiler technology is not ready to translate sequential programs in multiple threads running on multiple cores [39]. Radical ideas are required to make many-core architectures a secure and robust base for productive software development since the existing solutions only shows successes in narrow application domains. This is the very reason why recently two groups of companies (AMD, HP, IBM etc. and Intel & Microsoft) sponsor parallel programming research at universities (Stanford and Berkley respectively)[12][13].

Products & Services. Products and services obey the general trends that need no further introduction. We only mention the demand for:

- increasing functionality (smarter, faster, better usable, etc.),
- shorter time to market, and
- cheaper services (or even free[14]).

1.2.3 Relating trends

The purpose of this section is to show that for an embedded system the various trends lead to a significant shift in the division between hardware and software. We will connect related autonomous trends from the different aspects and tag the connection with a matching or a non-matching (mismatch) relation, see Figure 1.2. The figure includes all four mentioned aspects with their trends enclosed in a rounded box. The tagged relations are shown, visualised by coloured ellipses:

[11] Stefan Gumbrich: "Embedded systems overhaul: It's time to tune up for the future of automotive.", IBM Business Consulting Services, 2004, `http://t1d.www-03.cacheibm.com/solutions/plm/doc/content/bin/g510_3987_embedded_systems_overhaul.pdf`.

[12] Advanced Micro Devices, Hewlett-Packard, IBM, Intel, NVidia and Sun Microsystems are funding Stanford's new Pervasive Parallelism Lab, and Intel and Microsoft officially announced their plan to research on parallel programming together with the University of California at Berkeley and University of Illinois at Urbana-Champaign.

[13] Rick Merritt: "Stanford kicks off parallel programming effort.", 2008, `http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=207403653` .

[14] Chris Anderson: "Free! Why $0.00 is the future of business.", 2008, `http://www.wired.com/techbiz/it/magazine/16-03/ff_free`.

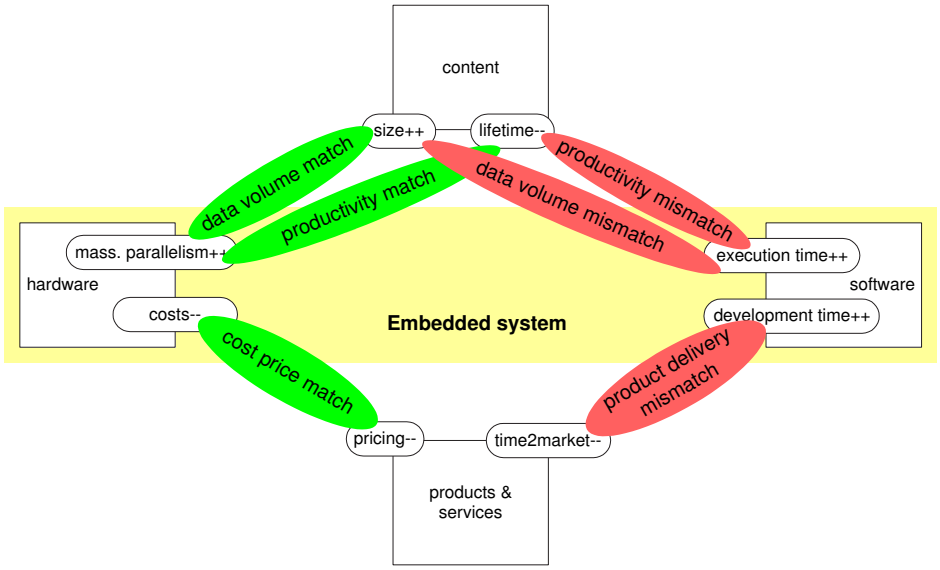


Figure 1.2: Relating trends in the four aspects: content, software, hardware and products & services, gives cause for a new partition in hardware/software co-design

green for a match, red for a mismatch. For example the increasing size of content matches with the parallel storage capacity of hardware.

Hardware. On the hardware side the following matches can be made, see Figure 1.2:

- the data volumes of content can be covered by distributing data over multiple processors,
- the update rate of information (reduced lifetime) can be handled by the massively parallel processing capacity and the high aggregated communication bandwidth of SoCs, and
- the demanded pricing reductions asked for by the market, are in line with the current developments in chip costs: more computational power for the same price.

Software. Software development still is immature in comparison to hardware development, where first-time-right[15] is the normal procedure. Software developments on average tend to be late, consume lots of engineering resources and have

[15] In digital hardware development the implementations are usually right the first time.

questionable quality, so it is a complex undertaking. This causes the following mismatches on the software side:

- the increasing amount of data (as indicated for example in Footnote (6) on page 4) cannot be handled adequately by current software practices while running on a General Purpose Processors (GPPs) [39],
- the software processing time (on GPPs) does not correspond to the update frequency of information, whether it concerns extensive on-line video processing (MB/sec), or off-line data mining (GB/h), and
- the increasing market pressure to deliver products faster than the previous version, demands reduced development times. This is in contrast to general software development practice.

Moreover, the following observations can be made:

- Compiler technology is not capable of generating parallel code from sequential legacy [39]. We need the option to code the parallelism manually.
- When moving towards very large number of processors, the current way of working requires more programmers than available [99].
- Most algorithms that require random access to data or take time greater than $O(N \cdot logN)$, for data size N, are not scalable to large data sets [99].

All relations at the hardware side do match and actually represent opportunities for solving problems. At the software side mismatches emerge, and those represent challenges for improving the development of embedded systems.

Summarising

The advances of heterogeneous multi-core chips in embedded systems design will also change the way software is written. This is independent of application domain, from a small multi-media Personal Digital Assistent (PDA) to blade-based racks in Amazon's compute server facilities; all have to run power-aware [39]. The above interrelation of hardware and software trends lead to the following conclusions:

- Traditional software development cannot cope with the identified trends: more data to process, shorter lifetime of information, and increasing market pressure to reduce time to market.
- Hardware, in particular the massively parallel many-core systems, enable new programming paradigms.
- The challenge is to find simple parallel processing schemes that reduce software complexity significantly.

1.3 Problem description

1.3.1 Problem description and thesis

Following the above mentioned line of reasoning it is beneficial to reconsider the traditional balance between hardware and software in embedded system design. Therefore, our approach is to break with the sequential coding tradition and apply parallelism to allow for new simple models. This requires support for modelling the problem in a parallel way such that it is suitable for a many-core hardware architecture, and human guidance for bridging the gap between the two in an orderly manner.

Today the *de facto* way applications are programmed on such dedicated systems is by manually adapting sequential code, which is mostly written in C. This adaptation involves the replacement of the time consuming sequential parts by parallel code. Most tooling is supplied by the manufacturer of the processor hardware and is, to no suprise and without exception, a C-compiler with *intrinsic instructions* (hardware dependent predefined functions), and occasionally, a simulator. This means that the design can only be validated at the end of the development cycle, when the code finally becomes available.

This leads to the following research thesis:
While most research on firmware[16] *development concentrates on automatic conversion of C-like descriptions to program applications for massively parallel processors, it is more productive to explicitly remodel the application in a parallel way by using a methodology based on a semi-automatic guidance through the whole firmware development process.*

1.3.2 Contribution

The research thesis leads to the following claims:

1. For an effective and efficient implementation on a massively parallel processing core it is necessary to manually (re)model the problem in a suitable parallel representation;
   Chapter 3, Section 4, and *Chapters 4, 5, 6, Sections 2, 3, 4.*

2. A semi-automatic and interactive development process is needed for mapping a task on a dedicated massively parallel processing core efficiently;
   Chapter 3, (Sub)sections 2, 5.1.

3. A single architectural language firmly based on mathematics for all development phases reduces development time and reduces the number of design errors;
   Chapter 3, Subsection 5.2.1, and *Chapters 4,5,6, Subsection 4.2.*

[16] Firmware is a computer program that is embedded in a hardware device, for example a microcontroller. The term "firmware" was originally used for micro-programs written for microsequencers such as AMD29xx.

4. Most of the relevant extra-functional requirements can be handled by integrating them into the regular functional flow; as a consequence the architectural language should support *in situ* monitoring and visualisation of quantifiable extra-functional properties;
   Chapter 3, Section 4, and *Chapters 4,5,6, Subsection 4.2.*

5. In the development process small steps and immediate feedback are crucial;
   Chapter 3, Section 4.

6. The development process should have a phased approach serving the various development roles, and should subsequently include:

   (a) a familiarisation phase with respect to the problem and the target hardware architecture(s),

   (b) an incremental prototyping phase (hardware architecture independent),

   (c) a transformational development phase (hardware architecture dependent),

   which are performed in a cyclic manner when needed (e.g., in case of design iterations);
   Chapter 3, Sections 6, 7, 8, and *Chapters 4, 5, 6, Sections 2, 3, 4.*

1.4 Thesis outline

The design methodology proposed in this thesis is shaped by evaluating three different case-studies, each with its own characteristics. The cases provide for a wide coverage of existing as well as new problem contexts and models. All three cases map on the same hardware architecture: a massively parallel processing array (Linedancer).

In the first case, a high volume image processing pipeline for a colour printer, is combined with a known model (FPGA implementation), see Chapter 5. Next, for image quantisation, a new model is developed that fits well on a parallel array, see Chapter 4. Finally, in the last case a new problem (mining and visualisation of a document space) is selected to extend and test the robustness of the methodology, see Chapter 6. The design methodology, called IRIS, is presented in Chapter 3, and includes an overview on state of the art methodologies. Because of the close interaction of hardware and software we have included a short overview of the state of the art on many-core systems, see Chapter 2, that also includes some details of the used Linedancer processor. In Chapter 7 we formulate the conclusions.

CHAPTER 2

State of the Art Massively Parallel Embedded Architectures

In this chapter we focus on many-core architectures for streaming applications. The many-core concept has a number of advantages: (1) depending on the requirements, cores can be (dynamically) switched on/off, (2) the many-core structure fits well to future process technologies, more cores will be available in advanced process technologies, but the complexity per core does not increase, (3) the many-core concept is fault tolerant, faulty cores can be discarded and (4) multiple cores can be configured in parallel. When processing and memory are combined in the cores, tasks can be executed efficiently on cores (locality of reference). There are a number of application domains that can be considered as streaming applications, for example colour image processing, data mining, multimedia processing, medical image processing, sensor processing (e.g., remote surveillance cameras), phased array radar systems and wireless baseband processing. In this chapter the key characteristics of streaming applications are highlighted, and the characteristics of the processing architectures to efficiently support these types of applications are addressed. We present an overview of some state-of-the-art embedded core architectures for streaming applications and select one as a target hardware architecture to be used in this thesis.

Major parts of this chapter have been accepted as a bookchapter for the CRC Book series [P9].

2.1 Introduction

This chapter addresses heterogenous and homogeneous many-core SoC platforms for streaming applications. In streaming applications, computations can be specified as a data flow graph with streams of data items (the edges) flowing between computation kernels (the nodes). Most signal processing applications can be naturally expressed in this modelling style [32]. Typical examples of streaming applications are: colour image processing (Chapter 5, Chapter 4), data mining (Chapter 6), multimedia processing (e.g., MPEG, MP3 coding/decoding), medical image processing, sensor processing (e.g., remote surveillance cameras), phased array radar systems and wireless baseband processing. In a heterogeneous many-core architecture, a core can either be: a bit-level reconfigurable unit (e.g., FPGA), a word-level reconfigurable unit, or a general-purpose programmable unit (DSP or microprocessor). We assume the cores of the SoC are interconnected by a reconfigurable Network on Chip (NoC). The programmability of the individual cores enables the system to be targeted at multiple application domains.

We take a holistic approach, which means that all aspects of systems design need to be addressed simultaneously in a systematic way [108]. We believe that this is key for an efficient overall solution, because an interesting optimization in a small corner of the design might lead to inefficiencies in the overall design. For example the design of the NoC should be coordinated with the design of the processing cores, and the design of the processing cores should be coordinated with the tile specific compilers. Eventually, there should be a tight fit between the application requirements and the SoC and NoC capabilities.

We first introduce streaming applications and many-core architectures in sections 2.1.1 and 2.1.2. After that we give a multi-dimensional classification of architectures for streaming applications in Section 2.2. For each category one or more sample architectures are presented (Section 2.3). We end this chapter with a conclusion and make a selection for the target hardware architecture to be used in this thesis, see Section 2.4.

2.1.1 Streaming Applications

The focus of this chapter is on many-core SoC architectures for streaming applications where we can assume that the data streams are semi-static and have a periodic behaviour. This means that for a long period of time subsequent data items of a stream follow the same route through the SoC. The common characteristics of typical streaming applications are:

- They are characterised by relatively simple local processing but a huge amount of data.
- Data arrives at nodes at a rather fixed rate, which causes periodic data transfers between successive processing blocks. The resulting communication bandwidth is application dependent and a large variety of communi-

cation bandwidth is required. The size of the data items and data rate is application dependent.

- The data flows through the successive processes in a pipelined fashion. Processes might work in parallel on parallel processors or can be time-multiplexed on one or more processors. Therefore, streaming applications show a predictable temporal and spatial behaviour.

- For our application domains, typically throughput guarantees (in data items per sec.) are required for the communication as well as for the processing. Sometimes also latency requirements are given.

- The life-time of a communication stream is semi-static, which means a stream is fixed for a relatively long time.

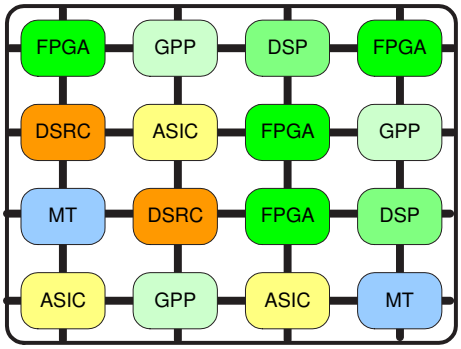
2.1.2 Many-core Architectures

Flexible and efficient SoCs can be realised by integrating hardware blocks (called tiles or cores) of different granularities into heterogeneous SoCs. In this chapter the term *core* is used for processor-like hardware blocks and the term *tile* is used for Application Specific Integrated Circuits (ASICs), fine-grained reconfigurable blocks and memory blocks. We assume that the interconnected building blocks can be heterogeneous (see Figure 2.1), for instance bit-level reconfigurable tiles (e.g., embedded FPGAs), word-level reconfigurable cores (e.g., Domain Specific Reconfigurable Cores), general-purpose programmable cores (e.g., DSPs and microprocessor cores) and memory blocks. From a systems point of view these architectures are heterogeneous multi-processor systems on a single chip. The programmability and reconfigurability of the architecture enables the system to be targeted at multiple application domains. Recently a number of many-core architectures have been proposed for the streaming application domain. Some examples will be discussed in Section 2.3.

A many-core approach has a number of advantages:

- It is a future-proof architecture, as the processing cores do not grow in complexity with technology. Instead, as technology scales, simply the number of cores on the chip grows.

- A many-core organization can contribute to the energy-efficiency of a SoC. The best energy savings can be obtained by simply switching off cores that are not used, which also helps in reducing the static power consumption. Furthermore, the processing of local data in small autonomous cores abides the locality of reference principle. Moreover, a core processor might be adaptive, it does not always have to run at full clock speed to achieve the required Quality of Service (QoS).



core	description
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
DSP	Digital Signal Processor
ASIC	Application Specific Integrated Circuit
DSRC	Domain Specific Reconfigurable Core
MT	Memory Tile

Figure 2.1: A heterogenous SoC template

- When one of the cores is discovered to be defect (either because of a manufacturing fault or discovered at operating-time by the built-in-diagnosis) this defective core can be switched-off and isolated from the rest of the design.
- A many-core approach also eases verification of an integrated circuit design, since the design of identical cores only has to be verified once. The design of a single core is relatively simple and therefore a lot of effort can be put in (area/power) optimizations on the physical level of integrated circuit design.
- The computational power of a many-core architecture scales linearly with the number of cores. The more cores there are on a chip, the more computations can be done in parallel (provided that the network capacity scales with the number of cores and there is sufficient parallelism in the application).
- Although cores operate together in a complex system, an individual tile operates quite autonomously. In a reconfigurable many-core architecture every processing core is configured independently. In fact, a core is a natural unit of partial reconfiguration. Unused cores can be configured for a new task, while at the same time other cores continue performing their tasks. That is to say, a many-core architecture can be reconfigured partly and dynamically.

Heterogeneous Many-Core SoC (MC-SoC)

The reason for heterogeneity in a SoC is that typically, some algorithms run more efficiently on bit-level reconfigurable architectures (e.g., pseudo random number generation), some on DSP-like architectures and some perform optimal on word-level reconfigurable platforms (e.g., FIR filters or FFT algorithms). We distinguish four processor types: *General Purpose Processor*, *fine-grained* reconfigurable hardware (e.g., FPGA), *coarse-grained* reconfigurable hardware and *ded-*

icated hardware (e.g., ASIC). The different tile processors in the SoC are interconnected by a Network-on-Chip (NoC). Both SoC and NoC are dynamically reconfigurable, which means that the programs running on the processing tiles as well as the communication channels are configured at run-time. The idea of heterogeneous processing elements is that one can match the granularity of the algorithms with the granularity of the hardware. Application designers or high-level compilers can choose the most efficient processing core for the type of processing needed for a given application task. Such an approach combines performance[1], flexibility and energy-efficiency. It supports high performance through massive parallelism, it matches the computational model of the algorithm with the granularity and capabilities of the processing entity, it can operate at minimum supply voltage and clock frequency and hence provides energy-efficiency and flexibility at the right granularity only when and where needed and desirable. A thorough understanding of the algorithm domain is crucial for the design of an (energy-) efficient reconfigurable architecture. The architecture should impose little overhead to execute the algorithms in its domain. Streaming applications form a good match with many-core architectures: the computation kernels can be mapped on cores and the streams to the NoC links. Inter-processor communication is in essence also overhead, as it does not contribute to the computation of an algorithm. Therefore, there needs to be a sound balance between computation and inter-processor communication. These are again motivations for a holistic approach.

Programmability

Design automation tools form the bridge between processing hardware and application software. Design tools are a crucial requirement for the viability of many-core platform chips. Such tools reduce the design cycle (i.e., cost and time-to-market) of new applications. The application programmer should be provided with a set of tools that on one side hides the architecture details but on the other side gives an efficient mapping of the applications onto the target architecture. However, high-level language compilers for domain specific streaming architectures are far more complex than compilers for general purpose superscalar architectures because of the data dependency analysis, instruction scheduling and allocation. Next to tooling for application development also tooling for functional verification and debugging is required for programming many-core architectures. In general, such tooling comprises:

- general Hardware Description Language (HDL) simulation software that provides full insight in the hardware state, but is relatively slow and not suited for software engineers,

[1] With performance we mean the number of operations per time unit, and this is reciprocal to execution time.

- dedicated simulation software that provides reasonable insight in the hardware state, performs better than general hardware simulation software and can be used by software engineers, and
- hardware prototyping boards that achieve great simulation speeds, but provide poor insight in hardware state and are not suited for software engineers.

By employing the tiled SoC approach, as proposed in Figure 2.1, various types of parallelism can be exploited. Depending on the core architecture, one or more levels of parallellism are supported.

- *Thread-Level Parallelism* is explicitely addressed by the many-core approach as different tiles can run different threads;
- *Data-Level Parallelism* is achieved by processing cores that employ parallelism in the data path;
- *Instruction-Level Parallelism* is addressed by processing cores when multiple data path instructions can be executed concurrently.

The programming of these kinds of streaming architectures is on one hand complex because of the variety in processors and parallelism but also complex because of the primitive state of the tooling. Furthermore, the *composability* issue[2] needs extra attention and restricts the design choices in hardware architecture as well as software [108]. The programmability of many-core architectures is an unsolved problem.

2.2 Classification

Different hardware architectures are available in the embedded systems domain to perform DSP functions and algorithms: *GPP*, *DSP*, *(re-)configurable hardware* and *application specific hardware* ASIC.

These hardware architectures have different characteristics in relation to *performance*, *flexibility* or *programmability* and *energy efficiency*. Figure 2.2 depicts the trade-off in flexibility and performance for different hardware architectures. Generally, more flexibility implies a less energy efficient solution.

Crucial for the fast and efficient realisation of a Many-Core System-on-Chip (MC-SoC) is the use of pre-designed modules, the so-called building blocks. In this section we will first classify these building blocks, second we classify the MC-SoCs that can be designed using these building blocks together with the interconnection structures between these blocks.

A basic classification of MC-SoC building blocks is given in Figure 2.3. The

[2] Composability is a desired property that relates to the mapping of multiple independent applications on the same platform with the condition that each application does not influence any other application.

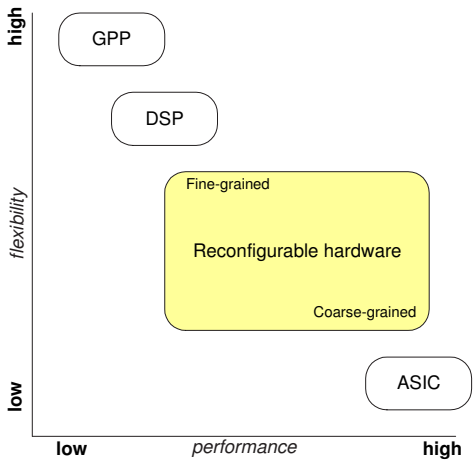


Figure 2.2: Flexibility versus performance trade-off for different hardware architectures

basic processing elements of an MC-SoC can be divided in run-time reconfigurable cores and fixed cores. The functionality of a run-time reconfigurable core is fixed for a relatively long period in relation to the clock frequency of the cores. Run-time reconfigurable cores can be subdivided into two classes: fine-grained reconfigurable cores and coarse-grained reconfigurable cores. Fine-grained reconfigurable cores are reconfigurable at bit-level (e.g., FPGA) while coarse-grained reconfigurable cores are reconfigurable at word-level (8 bit, 16 bit etc.). Two other essential building blocks are memory and I/O blocks. Reusing MC-SoCs building blocks to build larger systems increases the productivity of designers.

A classification of MC-SoCs is given in Figure 2.4. An MC-SoC basically consists of multiple building blocks connected by means of an interconnect. If an MC-SoC consists of multiple building blocks of a single type, the MC-SoC is referred to as *homogeneous*. The homogeneous MC-SoC architectures can be subdivided into Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD) and array architectures. Examples of these architectures will be given below. If multiple types of building blocks are used, the MC-SoC is called *heterogeneous*.

To interconnect the different building blocks, three basic classes can be identified: bus, Network-on-Chip and dedicated interconnects. A bus is shared between different processing cores and is a notorious cause of unpredictability. Unpredictability can be circumvented by a NoC [7]. Two types can be identified: packet-switched and circuit-switched. Besides the use of these more or less standardised communication structures, dedicated interconnects are still widely used. Some examples of different MC-SoC architectures are presented in Table 2.1.

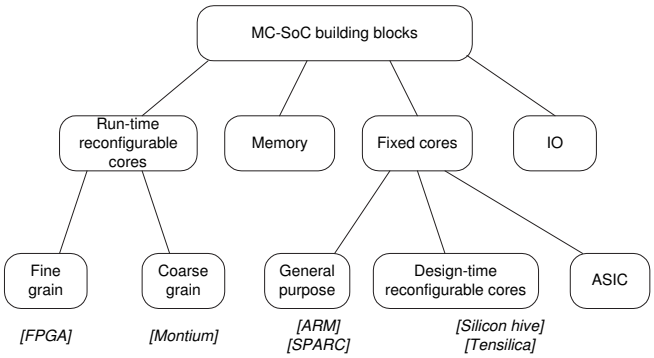


Figure 2.3: Classification of MC-SoC building blocks for streaming applications

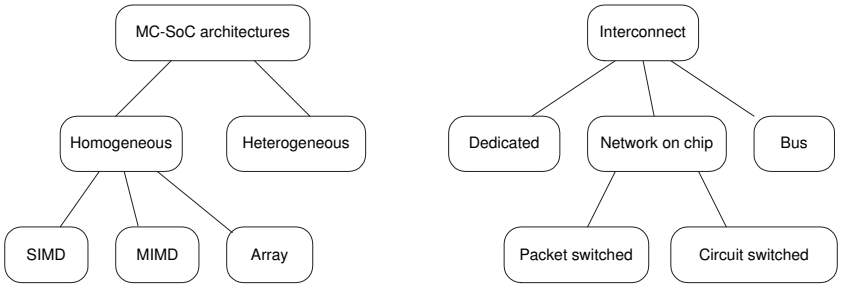


Figure 2.4: Classification of MC-SoC architectures and interconnect structures for streaming applications

In the following sections a few characteristic architectures will be presented in more detail.

2.3 Sample architectures

2.3.1 Montium Reconfigurable Processing Core

The Montium is an example of a coarse-grained reconfigurable processing core and targets the 16-bit DSP algorithm domain. The Montium architecture origins from research at the University of Twente [63][100]. The Montium processing core has been further developed by Recore Systems[5]. A single Montium process-

[3] Nvidia, "Nvidia G80, architecture and GPU analysis", 2007, `www.beyond3d.com/content/reviews/1`.

[4] Strictly speaking the Cell can be positioned as a heterogeneous processor, but because of the relative large number of SIMD cores it is categorised as homogeneous.

[5] Recore Systems, `www.recoresystems.com`.

Table 2.1: Examples of different MC-SoC architecture

Class		Example
Homogeneous	SIMD	Linedancer (see Section 2.3.4) Geforce G80[3] Xetal [42]
	MIMD	Tilera (see Section 2.3.3) Cell[4] [40] Intel Tflop processor [44]
	Array	PACT (see Section 2.3.2) ADDRESS [85]
Heterogeneous		Montium [63] Silicon Hive [24]

ing tile is depicted in Figure 2.5. At first glance the MONTIUM architecture bears a resemblance to a VLIW processor. However, the control structure of the MONTIUM is very different. The lower part of Figure 2.5 shows the Communication and Configuration Unit (CCU) and the upper part shows the coarse-grained reconfigurable MONTIUM Tile Processor (TP).

Communication and Configuration Unit The CCU (Communication and Configuration Unit) implements the network interface controller between the NoC and the MONTIUM TP. The definition of the network interface depends on the Network-on-Chip (NoC) technology that is used in a System-on-Chip (SoC) in which the MONTIUM processing tile is integrated [23]. The CCU enables the MONTIUM TP to run in *streaming* as well as in *block* mode. In *streaming* mode the CCU and the MONTIUM TP run in parallel. Hence, communication and computation overlap in time in *streaming* mode. In *block* mode the CCU first reads a block of data, then starts the MONTIUM TP, and finally after completion of the MONTIUM TP, the CCU sends the results to the next processing unit in the SoC (e.g., another MONTIUM processing tile or external memory).

Montium Tile Processor The Tile Processor (TP) is the computing part of the MONTIUM processing tile. The MONTIUM TP can be configured to implement particular DSP algorithms such as: all power of 2 FFTs upto 2048 points, non-power of 2 FFT upto FFT 1920, FIR filters, IIR filters, matrix vector multiplication, Discrete Cosine Transform (DCT) decoding, Viterbi decoders, Turbo (SISO) decoders. Figure 2.5 reveals that the hardware organisation of the MONTIUM TP is very regular. The five identical Arithmetic Logic Units (ALU1 through ALU5) in a tile can exploit data level parallellism to enhance performance. This type of parallelism demands a very high memory bandwidth, which is obtained by hav-

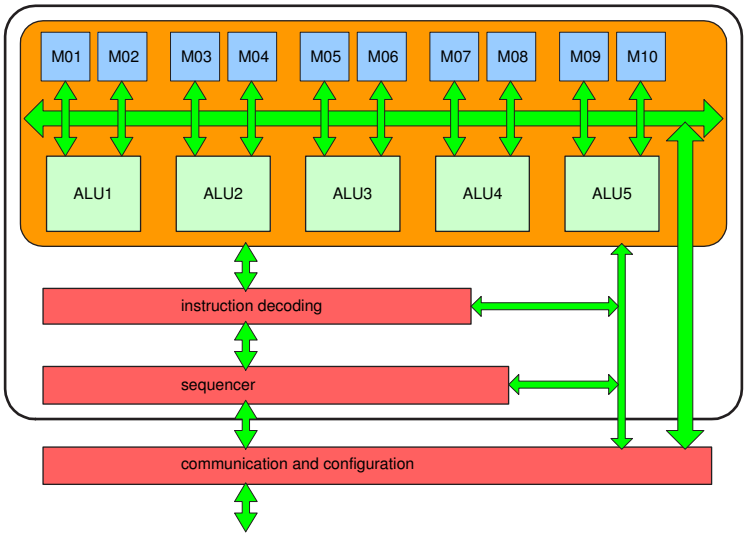


Figure 2.5: The Montium TP coarse-grained reconfigurable processing tile

ing 10 local memories (M01 through M10) in parallel. The small local memories are also motivated by the locality of reference principle. The data path has a width of 16-bits and the ALUs support both signed integer and signed fixed-point arithmetic. The ALU input registers provide an even more local level of storage. Locality of reference is one of the guiding principles applied to obtain energy efficiency in the Montium TP. A vertical segment that contains one ALU together with its associated input register files, a part of the interconnect and two local memories is called a Processing Part (PP). The five PPs together are called the Processing Part Arrays (PPAs).

A relatively simple sequencer controls the entire PPA. The sequencer selects configurable PPA instructions that are stored in the instruction decoding block of Figure 2.5. For (energy) efficiency it is imperative to minimise the control overhead. The PPA instructions, which comprise ALU, Address-Generation Unit (AGU), memory, register file and interconnect instructions, are determined by a DSP application designer at design time. All Montium TP instructions are scheduled at design time and arranged into a Montium sequencer programme. By statically scheduling the instructions as much as possible at design time, the Montium sequencer does not require any sophisticated control logic which minimises the control overhead of the reconfigurable architecture.

The Montium TP has no fixed instruction set, but the instructions are configured at configuration-time. During configuration of the Montium TP, the CCU writes the configuration data (i.e., instructions of the ALUs, memories and interconnects, etc.; sequencer and decoder instructions) in the configuration memory

of the MONTIUM TP. The size of the total configuration memory of the MONTIUM TP is about 2.9 KB. However, configuration sizes of DSP algorithms mapped on the MONTIUM TP are typically in the order of 1 KB. For example, a 64-point Fast Fourier Transform (FFT) has a configuration size of 946 bytes. The MONTIUM TP can be configured via the NoC interface by sending a configuration file containing configuration RAM addresses and data values to the CCU. The configuration memory of the MONTIUM TP is implemented as a 16-bit wide SRAM memory that can be written by the CCU. By only updating certain configuration locations of the configuration memory, the MONTIUM TP can be partially reconfigured. In the considered MONTIUM TP implementation, each local SRAM is 16-bit wide and has a depth of 1024 addresses, which results in a storage capacity of 2 KB per local memory. The total data memory inside the MONTIUM TP adds up to a size of 20 KB. A reconfigurable AGU accompanies each local memory in the PPA of the MONTIUM TP. It is also possible to use the local memory as a Look-up Table (LUT) for complicated functions that cannot be calculated using an ALU, such as sine or division (with one constant). The memory can be used in both integer or fixed-point LUT mode.

Design methodology

The MONTIUM development tools start with a high-level description of an application (in C / C++ or MATLAB) and translate this description to a MONTIUM TP configuration [58]. Applications can be implemented on the MONTIUM TP using an embedded C language, called MONTIUMC. The MONTIUM design methodology to map DSP applications on the MONTIUM TP is divided in three steps:

1. The high-level description of the DSP application is analysed and computationally intensive DSP kernels are identified;

2. The identified DSP kernels or parts of the DSP kernels are mapped on one or multiple MONTIUM TPs that are available in a SoC. The DSP operations are programmed on the MONTIUM TP using MONTIUMC;

3. Depending on the layout of the SoC in which the MONTIUM processing tiles are applied, the MONTIUM processing tiles are configured for a particular DSP kernel or part of the DSP kernel. Furthermore, the channels in the NoC between the processing tiles are configured.

2.3.2 PACT-XPP

The eXtreme Processing Platform (XPP) is an example of a homogeneous array structure. It is a run-time reconfigurable coarse-grained data processing architecture. The XPP provides parallel processing power for high bandwidth data

like for instance, video or audio processing. The XPP targets for streaming DSP applications in the multimedia and telecommunications domain[6] [9].

Architecture

The XPP architecture is based on a hierarchical array of coarse-grained, adaptive computing elements, called Processing Array Elements (PAEs). The PAE are clustered in Processing Array Clusters (PACs). All PAEs in the XPP architecture are connected through a packet-oriented communication network. Figure 2.6 shows the hierarchical structure of the XPP array and the PAEs clustered in a PAC.

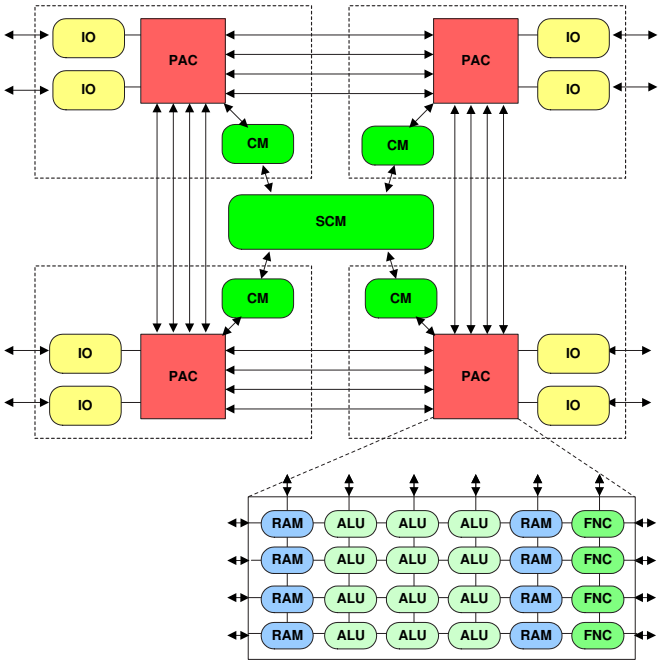


Figure 2.6: The structure of an XPP array composed of four PACs [9]

Different PAEs are identified in the XPP array: *ALU-PAE*, *RAM-PAE* and *FNC-PAE*. The ALU-PAE contains a multiplier and is used for DSP operations. The RAM-PAE contains a RAM to store data. The FNC-PAE is a sequential VLIW-like processor core. The FNC-PAEs are dedicated to the control flow and sequential sections of applications. Every PAC contains ALU-PAEs, RAM-PAEs and FNC-PAEs. The PAEs operate according to a data flow principle; a PAE starts processing data as soon as all required input packets are available.

[6] See also "PACT XPP Technologies", 2008, `www.pactxpp.com`.

If a packet cannot be processed, the pipeline stalls until the packet is received. So, it is possible to map a signal flow graph to the ALU-PAEs. Each PAC is controlled by a Configuration Manager (CM). The CM is responsible for writing configuration data into the configurable object of the PAC. Multi-PAC XPP arrays contain additional CMs for concurrent configuration data handling, arranged in a hierarchical tree of CMs. The top CM, called Supervising Configuration Manager (SCM), has an external interface that connects the supervising CM to an external configuration memory.

Design methodology

DSP algorithms are directly mapped onto the XPP array according to their data flow graphs. The flow graph nodes define the functionality and operations of the PAEs, whereas the edges define the connections between the PAEs. Basically, the XPP array is programmed using the Native Mapping Language (NML). In NML descriptions, the PAEs are explicitly allocated and the connections between the PAEs are specified. Optionally, the allocated PAEs are placed onto the XPP array. NML also includes statements to support configuration handling. Configuration handling is an explicit part of the application description.

A vectorising C compiler is available to translate C functions to NML modules. The vectorising compiler for the XPP array analyses the code for data dependencies, vectorises those code sections automatically and generates highly parallel code for the XPP array. The vectorising C compiler is typically used to program *regular* DSP operations which are mapped on *ALU-PAEs* and *RAM-PAEs* of the XPP array. Furthermore, a coarse-grained parallelisation into several FNC-PAE threads is very useful when *irregular* DSP operations exist in an application. This allows to even run irregular, control-dominated code in parallel on several FNC-PAEs. The FNC-PAE C compiler is similar to a conventional RISC compiler extended with VLIW features to take advantage of Instruction Level Parallelism (ILP) within the DSP algorithms.

2.3.3 Tilera

The Tile64[7] is an example of a homogeneous MIMD MC-SoC. It is based on the mesh architecture that was originally developed for the RAW machine [41]. The chip consists of a grid of processor tiles arranged in a network (see Figure 2.7), where each tile consists of a general purpose processor, a cache, and a non-blocking router that the tile uses to communicate with the other tiles on the chip.

Next to each processor there is a switch that connects the core to the iMesh on-chip network. The combination of a core and a switch form the basic building block of the Tilera Processor: the tile. Each core is a fully functional processor capable of running complete operating systems and off-the-shelf 'C' code. Each core is optimised to provide a high performance/power ratio, running at speeds

[7] Tilera Corporation, `www.tilera.com`.

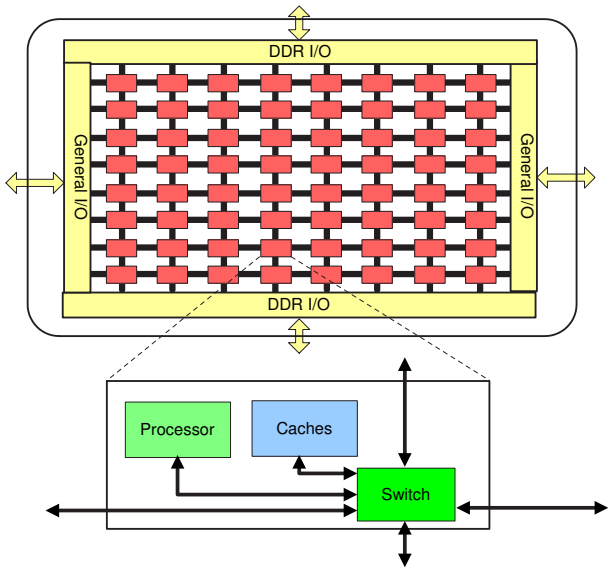


Figure 2.7: Tile64 Processor

between 600 MHz and 1 GHz, with power consumption as low as 170 mW in a typical application. Each core supports standard processor features such as:

- Full access to memory and I/O
- Virtual memory mapping and protection (MMU/TLB)
- Hierarchical cache with separate L1-I and L1-D
- Multi-level interrupt support
- Three-way VLIW pipeline to issue three instructions per cycle

The cache subsystem on each tile consists of a high-performance, two-level, non-blocking cache hierarchy. Each processor/tile has a split level 1 cache (L1 instruction and L1 data) and a level 2 cache, keeping the design, fast and power efficient. When there is a miss in the level 2 cache of a specific processor, the level 2 caches of the other processors are searched for the data before external memory is consulted. This way, a large level 3 cache is emulated.

This promotes on-chip access and avoids the bottleneck of off-chip global memory. Multi-core coherent caching allows a page of shared memory, cached on a specific tile, to be accessed via load/store references by other tiles. Since one tile effectively prefetches for the others, this technique can yield significant performance improvements.

To fully exploit the available compute power of large numbers of processors, a high-bandwidth, low-latency interconnect is essential. The network (iMesh) provides the high-speed data transfer needed to minimise system bottlenecks and to scale applications. iMesh consists of five distinct mesh networks: two networks are completely managed by hardware and are used to move data to and from the tiles and memory in the event of cache misses or DMA transfers. The three remaining networks are available for application use, enabling communication between cores and between cores and I/O devices. A number of high-level abstractions are supplied for accessing the hardware (e.g., socket-like streaming channels or message-passing interfaces.) The iMesh network enables communication without interrupting applications running on the tiles. It facilitates data transfer between tiles, contains all of the control and datapath for each of the network connections, and implements buffering and flow-control within all the networks.

Design methodology

The TILE64 Processor is programmable in ANSI C and C++. Tiles can be grouped into clusters to apply the appropriate amount of processing power to each application.

2.3.4 Linedancer

The Linedancer[8] is an *associative* processor and it is an example of a homogeneous MC-SoC. Associative processing is the property to execute only those processing elements where a certain value in their data register matches a value in the instruction [79]. Associative processing is built around an intelligent memory concept: Content Addressable Memory (CAM). Unlike standard computer memory (Random Access Memory or RAM) in which the user supplies a memory address and the RAM returns the data word stored at that address, a CAM is designed such that the user supplies a data word and the CAM searches its entire memory to see if that data word is stored anywhere in it. The CAM returns a tag list of zero or more storage addresses where the word was found. Each CAM line, that contains a word, can be seen as a Processor Element (PE) and each tag list element as a 1 bit condition register. Depending on this register, the control processor can either instruct the PEs to continue processing on the indicated subset, or to return the involved words subsequently for further processing.

In general the Linedancer belongs to the subclass of massively parallel SIMD architectures. This SIMD subclass is perfectly suited to support data parallelism, for example for signal, image and video processing, text retrieval, and large databases. The associative functions allow the processor to act as an intelligent memory (CAM), permitting high speed searching, and data dependent image processing operations (as median filters or object recognition/labeling). The so called

[8] Aspex Semiconductor: "Aspex Semiconductor Technology", 2008, `www.aspex-semi.com/q/technology.shtml`.

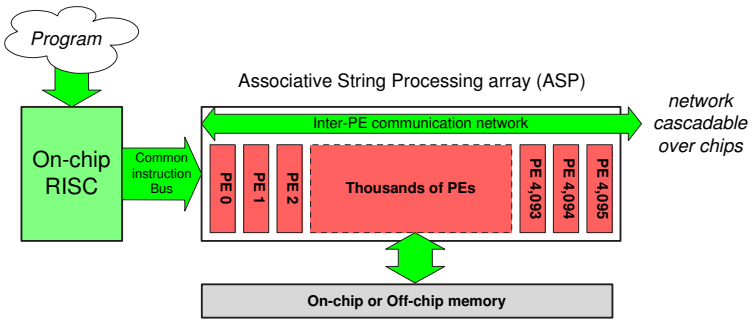


Figure 2.8: The scalable architecture of Linedancer

Associative String Processor (ASP) of the Linedancer, is designed around a very large number – up to 4,096 for the current Linedancer – of simple Processing Elements (PEs) arranged in a linear array, see Figure 2.8.

Application domains are diverse but have in common the simple processing of large amounts of data; from samples in 1D-streams to pixels in 2D or 3D data structures (e.g., images). Sample applications are: software defined radio (e.g., WiMAX), broadcast (Video compression), medical imaging (3D reconstruction), and in high-end printers – in particular for Raster Image Processing (RIP).

In the following sections the associative processor (ASP) and the Linedancer family are introduced. At the end we present the development toolchain and a short conclusion on the Linedancer application domain.

ASP architecture

Each PE has a 2-bit ALU, a 64 bit full associative memory array, a set of 8 user programmable bit-flags (ab_0 to ab_7), and 128 bit extended memory, see Figure 2.9 for a detailed view on the ASP architecture. The processors are connected in a 1-dimensional network, actually a 4K bit shift register, allowing data to be shared between PEs with minimum overhead. Linedancers can be cascaded by the indicated *left Link Port* (LLP) and *Right Link Port* (RLP). Two of the three indicated bit-lines are used for synchronous communication, the third one for asynchronous communication [106]. The ASP also has a separate bulk IO memory, the Primary Data Store (PDS), for high speed data input. The on-chip DMA engine automatically translates 2D and 3D images into the 1D array (and passed through via the PDS). The 1D architecture allows for linear scaling of performance, memory and communication, provided the application is expressed in a scalable manner. The Linedancer features a single Scalable Performance ARChitecture (SPARC) core for sequential processing and controlling the ASP, see Figure 2.10.

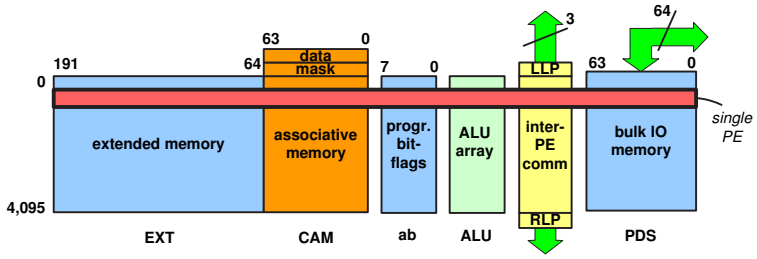


Figure 2.9: The architecture of Linedancer's Associative String Processor (ASP)

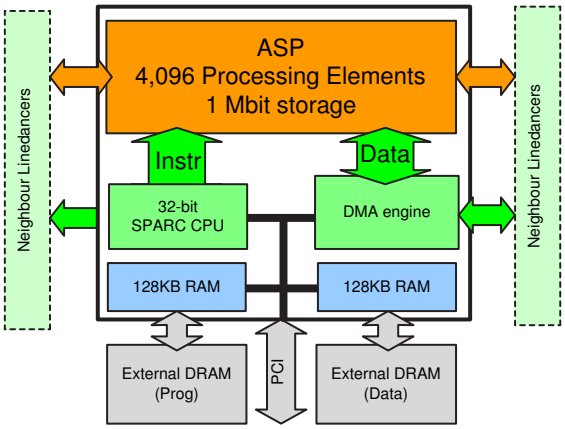


Figure 2.10: The Linedancer-P1 layout

Linedancer hardware architecture

The Linedancer processor family comes in two versions, the Linedancer-P1 and the Linedancer-HD. We will describe both briefly. See Appendix A for more details of the Linedancer.

Linedancer-P1. The Linedancer-P1 has a 32-bit SPARC core with 128KB internal program memory. System clock frequencies vary from 300, 350, 400 MHz. The Linedancer-P1 integrates an associative processor (ASP, with 4K PEs), a single SPARC core with a 4KB instruction cache, and a DMA controller capable of transferring 64-bit at 66 MHz over a PCI-interface, see Figure 2.10. It further hosts 128KB internal data memory. The chip consumes 3.5 W typical at 300 MHz.

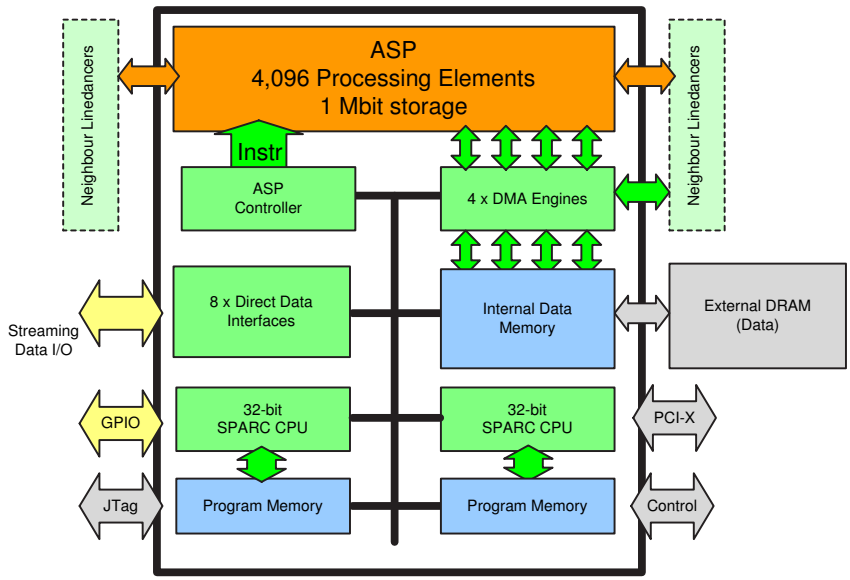


Figure 2.11: The Linedancer-HD layout

Linedancer-HD. The major differences of the Linedancer-HD compared to the P1 are:

- dual instead of single channel architecture: two 32-bit associative processors (2×2K PEs) and two SPARC cores,
- improved internal and external Direct Memory Access (DMA) and IO interfaces,
- improved inter-PE communication,
- improved DRAM interfacing, and
- slightly increased power consumption (4.5 W typical at 300 MHz).

The current Linedancers, the P1 and the HD, have been realised in 0.13μm CMOS process.

Design Methodology

The native software development environment for Linedancer consists of a compiler, linker and debugger. The Linedancer is programmed in C, with some parallel extensions to support the ASP processing array. The toolchain is based on the GNU compiler framework, with dedicated pre- and post-processing tools to compile and optimise the parallel extensions to C.

Associative SIMD processing adds an extra dimension to massively parallel processing, for example in searching/sorting and data dependent image processing. The Linedancer's 1D-architecture scales better than a 2D array often used in multi-ALU arrays as PACT's XPP or the Tilera's 2D multi core array. Because of the large size of the array, power consumption is relatively high compared to the Montium processor and prevents application into handheld devices.

2.4 Conclusion

In this chapter we addressed many-core architectures for streaming applications. Streaming DSP applications express computation as a data flow graph with streams of data items (the edges) flowing between computation kernels (the nodes). Typical examples of streaming applications are: printing related image processing, data mining, multimedia processing, medical image processing, sensor processing (e.g., remote surveillance cameras), phased array radar systems and wireless baseband processing. These application domains require flexible and energy-efficient architectures. This can be realised with a many-core architecture. The most important criteria for designing such a many-core architecture are: predictability and composability, energy-efficiency, programmability and dependability [108].

Two other important criteria are performance and flexibility. Different types of processing cores have been discussed, from ASICs, reconfigurable hardware, to DSPs and GPPs. ASICs have high performance but suffer from poor flexibility while DSPs and GPPs offer flexibility but modest performance. Reconfigurable hardware combines best of both worlds. These different processing cores are, together with memory- and I/O block, assembled into MC-SoCs. MC-SoCs can be classified into two groups: homogeneous and heterogeneous. In homogeneous MC-SoCs, multiple cores of a single type are combined where in a heterogeneous MC-SoC, multiple cores of different types are combined.

We also discussed four different architectures: the Montium, the PACT-XPP, the Tilera processor and the Aspex Linedancer. The Montium is a coarse-grain, run-time reconfigurable core. The PACT-XPP is an array processor where multiple ALUs are combined in a two-dimensional structure. The Tilera processor is an example of a homogeneous MIMD MC-SoC. The Aspex Linedancer is a homogeneous MC-SoC where a single instruction is executed by a large amount of simple processors simultaneously (SIMD).

The choice of a suitable target hardware architecture is mainly determined by the fit to the application domains: image processing (Chapter 5, Chapter 4) and data mining (Chapter 6). Because the involved image processing and neural network processing both demand for simple operations on a large number of data elements (i.e., pixels, neurons), we selected the Linedancer processor.

CHAPTER 3

The IRIS Firmware Design Methodology

Developing code for dedicated massively parallel hardware architectures is a tedious job. This is caused by the absence of both a coherent methodological framework and a hardware independent toolchain. Moreover, the inherently difficult nature of programming dedicated massively parallel embedded processors, complicates the matter. In this chapter we first present an inventory of influential methodologies for programming massively parallel architectures in streaming applications. Next we introduce a single framework called Iris *to generate code for such architectures.* Iris *is based on an incremental construction of executable representations, which converge to the final target language implementation in a semi-automated way. This chapter gives an overview of* IRIS *in a rather abstract way. In the next three chapters the methodology will be illustrated via case studies.*

3.1 Introduction

Embedded systems manufacturers nowadays are facing tough problems in developing high performance applications. The ever growing functionality of applications combined with new programmable many-core processors increase the development complexity. For this reason Patterson [39] states: "Although compatibility with old binaries and C programs are valuable to industry ... we welcome new programming models and new architectures if they simplify efficient programming of such highly parallel systems". Congruent to this we believe that parallelism cannot always be extracted automatically from sequential code: we need the pos-

Major parts of this chapter have been published in [P4].

sibility to specify the parallelism by the application programmer. In this chapter, we introduce a methodology that improves the programmer's efficiency for – but not limited to – *Single Instruction Multiple Data* (SIMD) architectures.

The *de facto* way applications are programmed on such dedicated architectures is by manually adapting sequential code, which is mostly written in C. This adaptation involves the replacement of the time critical sequential parts by parallel code. In Figure 3.1 the typical steps of a firmware development trajectory are depicted. The trajectory starts with an analysis phase that leads to the specification of the system. This specification is subsequently transformed in a software architecture and this marks the end of the analysis phase. The analysis phase is typically done by a single person (the architect) or a small team to support *conceptual integrity*, a quality property of an architecture [21]. After that the design of the various modules can start. Because the design of the modules, that make up the complete system, can be done in parallel, typically more developers are entering the developing team. The design phase will take relatively much effort to complete, see the effort per time unit in Figure 3.1. Both the analysis and design activities are informal and are for a part even text-based. Although some parts can be modelled in language(s) (e.g., MATLAB), the typical approach lacks a single integral and executable framework that allows for immediate system-wide verification. After compilation the application is executed for the very first time on the simulator (Figure 3.1) and thus will most probably reveal errors. Especially the errors caused by a poor analysis or high level design, cost a lot of effort to repair as reported by Boehm [17]. This percolation of early errors into (late) design phases prolongs the relatively high effort per time unit, even to redesign tasks (firmware maintenance).

Developers would like to have a compiler that is intelligent enough to generate code (from sequential source code) for SIMD architectures and that satisfies all requirements and constraints. However, such an approach is far from reality [39][86]. The reasons for the complexity of compiler construction for such architectures are manifold. To mention a few: SIMD architectures have a complex and deviating instruction set, have parallel (non-sequential) computing facilities, and have limited number of resources. The consequence is that parts of the code have to be written by hand in an assembler like language, and practically without assistance of a methodology or support of a tool.

Most tooling for SIMD architectures is supplied by the manufacturer of the processor hardware and is, to no suprise and without exception, a C-compiler supporting *intrinsic instructions* (hardware dependent predefined functions), and sometimes a simulator. This means that the design can only be validated at the end of the development cycle, when finally the code becomes available. This late validation constitutes a major problem in developing code for dedicated processors. Although the informal analysis and design are partly supported by formal modelling in some language(s), the absence of a single formal framework leaves a lot of space for errors. Two of the most popular languages for algorithm development are C and MATLAB. The language C, however, is notorious for its

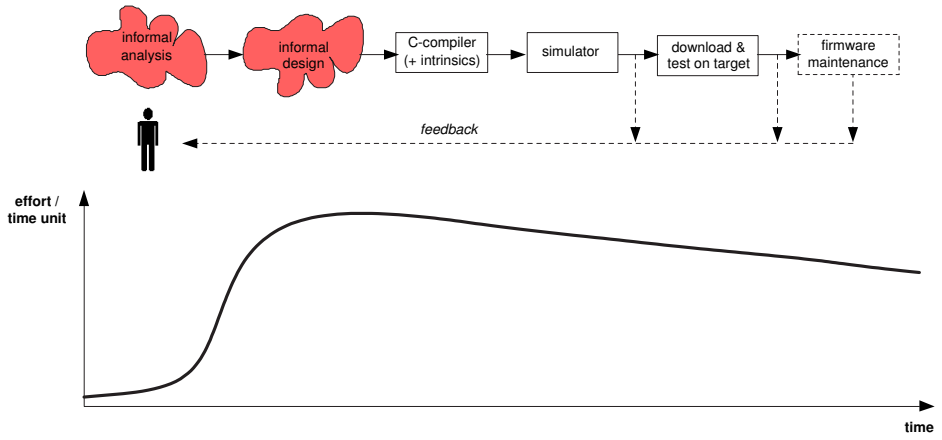


Figure 3.1: Estimated typical trajectory of current firmware development for SIMD architectures: a large integral development effort

poor representation of the problem (lack of abstraction) as well as parallelism, while MATLAB lacks the flexibility to code functionality efficiently (imperative programming model) and to incorporate implementation details (e.g., hardware concepts)[1]. As a consequence the missing parts are either performed by hand (rewriting code) or additional tooling is required.

Combined with the above mentioned specific problems of firmware development for SIMD architectures this section outlines the challenge to address. The main challenge is to design a *methodological framework for SIMD firmware development* that should at least fulfill the following requirements:
a. be an ***integral*** design method that supports firmware development for the whole trajectory (from problem-scouting till maintenance),
b. be ***interactive*** and be executable during the whole development process,
c. uses a **s**ingle language supporting multiple roles,
d. be ***incremental***, enabling further development given the current state of the design,
e. supports ***reuse*** to improve quality and efficiency, and
f. be domain ***independent***, that is, be applicable to multiple application domains.

In this chapter we propose a methodological framework, called IRIS, that satisfies these requirements. A more desired shape of effort per time unit curve – compared to the traditional approach as shown in Figure 3.1 – is depicted in Figure 3.2. The improvement allows for immediate feedback to the developer even

[1] MATLAB supports fixed point typing and direct compilation to C and HDL for a subset of the language though.

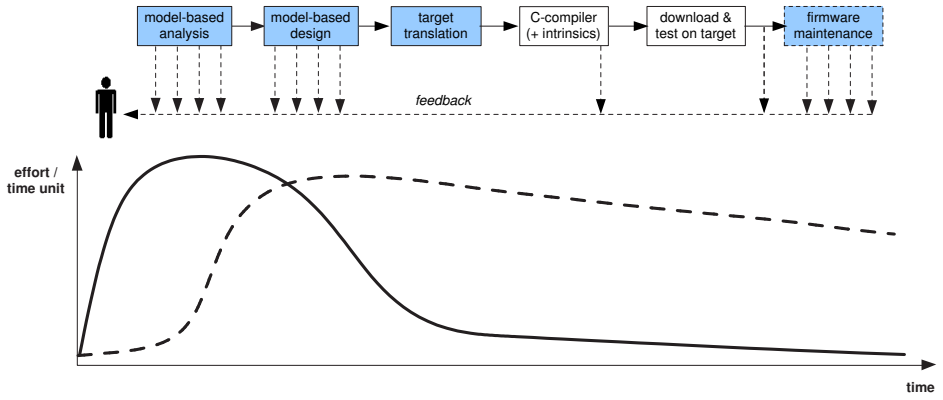


Figure 3.2: Desired improvement of the firmware development trajectory for SIMD architectures. The desired trajectory (solid line) results in a lower integral development effort.

without the need for (detailed) algorithm development or availability of hardware. In Figure 3.2 the target translation phase generates target source code semi-automatically from the model. The standard toolchain can then be used to compile and download the object code to the target hardware, as in the conventional trajectory in Figure 3.1. The effort graph in Figure 3.2 shows a significant lower integral development effort compared to the conventional approach. The IRIS framework is model-based, that is models are used to improve the development process in quality, cost and effort. Since textual specifications are exchanged by a model-based approach, the analysis phase is done more thoroughly (more quality) thus takes more effort to complete. The same applies to the design phase but here the 'overhead' is smaller because of the higher quality of the analysis phase. The real profit, however, is in the next phases because of this higher quality in the model code. The blue shaded development steps in Figure 3.2 are supported by IRIS.

As a prominent feature of IRIS we mention that during the complete design process the same language is employed, which supports executability of models during all phases. We call such a language an *architectural language*.

This chapter is structured in the following manner. We start with an inventory of the major developments in hardware/software co-design (Section 3.2), supplemented with model-based approaches (subsection 3.2.5). Then in Section 3.3 we come up with the description of an 'ideal' methodology for firmware development for massively parallel processors, and mention the problems to be solved. The departure points of IRIS, see Section 3.4, follow logically from these problems, and precede the introduction of IRIS in Section 3.5. In sections 3.6, 3.7, and 3.8 we introduce the various phases and subphases of IRIS. Because of the rather static

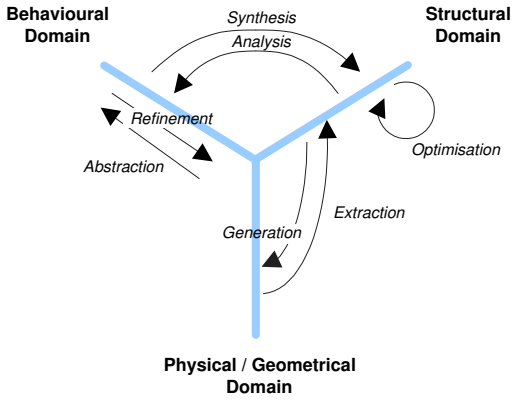


Figure 3.3: The Y-chart integrates the multi-stakeholder issues in the development of a hardware subsystem

description of these phases we will provide concrete guidelines for traversing the various (sub)phases and how to use the architectural language (Section 3.5.2). In the last two sections we summarise the observations (Section 3.9) and draw some conclusions (Section 3.10).

3.2 State of the Art Methodologies

This section presents relevant developments in the methodology for constructing firmware for embedded systems.

3.2.1 Multi-disciplinary aspect

An influential development approach for hardware/software co-design, the Y-chart [50][37], is based on concurrent elaboration on multiple domains (coupled to stakeholders) at different abstraction levels, see Figure 3.3. These domains, which in fact are different views for specifying a hardware system, are: behavioural (functional etc.), structural (hierarchy of interconnected components, computer architecture) and the physical/geometrical domain (physical placement in space and physical characteristics). It is potentially a very generic methodology, but it is mostly used for hardware development and not well suited for developing code for existing many-core programmable processing systems.

The development of embedded systems involves new types of stakeholders, that are not addressed in the Y-chart approach. One of these stakeholders is the customer or end-user of the embedded system. According to the Soft System Methodology (SSM) [26], the intended solution not only involves a *hard system*

(a technological system that can be engineered) but also a *soft system* (an organisation, a human being, etc.) that uses the system. SSM is particularly strong in preparing the foundation of a system development. In this respect it is useful in the search for the scope and demarcation of the hard system's boundary.

3.2.2 Iterative aspect

Claasen [27] puts emphasis on the iterative aspect in hardware/software co-design. The *extra-functional design properties*[2] as performance, power and resource consumption, are analysed by using post-mapping analysis tools, tools that can only be used after the complete design is finished. However, for interactive code development we need instant mapping analysis.

In addition to the common direction for functional development, in [82] an orthogonal direction, namely that of *design space exploration* is introduced. Design Space Exploration is a structured way of identification and evaluation of design alternatives, and the development of criteria. The ultimate choice, which is part of *decision recording*, starts off a next development cycle.

Agile methods such as Extreme Programming (XP) [10] try to reduce development time, typically from months to weeks, by reintroducing interactivity to the design process. These methods, however, mostly use an implementation language for the development roles. This leads to less readable and maintainable code in particular for the early phases. Recently [54] more emphasis is put on raising the level of abstraction by using new parallel languages instead of extending the traditionally used sequential languages (mostly C-based). However, these languages lack possibilities for the detailed control at elementary processor level, necessary for realising the tight constraints on extra-functional properties (e.g., power).

Platform based design [72] recognises the importance of both top-down and bottom-up development. The basic tenets for platform based design are: regarding design as a "meeting in the middle process" and the identification of precisely defined layers where the refinement and abstraction process takes place. No concrete proposals are given how to define such layers in practice. For Image Processing applications, Bagdanov [6] advocates the separation of development and implementation in large Object Oriented frameworks (Horus). He selected a functional language for the development purpose.

The general practice in designing a system is that a lot of design decisions are made implicitly and are not explored. In [15] a cybernetic (control by feedback) approach is described in which subsequent architectures are generated, a process that is directed by a set of (extra-functional) properties. These architectures evolve in the end to an architecture that is 'frozen' as a framework (the software architecture) for subsequent design and implementation. Although the process is described in a linear way, in practice the path taken can cycle a few times

[2] Extra-functional requirements refer to all non-functional requirements and constraints that are relevant for the realisation of an embedded system.

and already taken decisions have to be rolled back and redone. For software programmable many-core systems it would be possible to maintain these evolving architectures with respect to both the functionality and associated properties.

3.2.3 Software economics

From the software economics side it is known since long that two relevant issues influence the choice of a development methodology and in particular of the architectural language. First, the cost of reworking the software is much smaller (by factors up to 200) in earlier phases than later phases [17]. Second, the length of description (measured in thousand Lines Of Code (KLOC)) is a dominant factor in software development costs as described in the COnstruction COst MOdel (COCOMO) [16]. The shorter the description the better, giving credit to declarative languages (e.g., functional languages).

3.2.4 Many-core developments

The recent developments on many-core systems certainly will influence the methodology landscape. Patterson [39] formulated some interesting recommendations for developing software for many-core systems:

- Future programming models must be more human-centric compared to the conventional focus on hardware or applications.
- A programming model must allow the programmer to balance the competing goals of *productivity* and *implementation efficiency*. He foresees that for this balance the following two conflicting goals are important:
  - *Opacity* abstracts the underlying architecture.
  - *Visibility* makes the key elements of the underlying hardware visible to the programmer.
- For the multi/many-core processor domain it is difficult to innovate the current compiler technology to support parallelism[3]. Some tooling, for example search-based *autotuners*, can partly alleviate the problems [39]. Autotuners optimise a set of library kernels by generating many variants of a given kernel and benchmarking each variant by running them on the target platform.
- Add hardware support to increase robustness against programming faults.

Although most of the recommendations make sense for firmware development, the autotune technology is not ready yet for dedicated SIMD architectures. In this respect we follow Hwu [86] in the advice that the legacy (sequential) code base must be revised or redeveloped.

[3] For specific domains as vector processing, tools exists that extract parallelism out of legacy (Fortran) code, but for the general case it is extremely difficult [39].

3.2.5 Model-based design

According to [22] models provide abstractions of a physical system that allows engineers to reason about that system by ignoring extraneous details while focussing on relevant ones. Models are used in many ways: to predict system properties, to reason about specific properties when conditions are changed, and to communicate key system characteristics to various stakeholders. Today models are used in practically all methodologies. Model related design approaches come in different flavours, see Figure 3.4. Besides *Model-based design* and hardware/software co-design, three other model-based approaches are relevant to discuss: *Model-driven development*, *Stream oriented process models*, and *Mathematical models*. The IRIS methodology as presented in Section 3.5, is influenced by many of these models.

Model-based design. Model-based design originally emerged from the control engineering domain. In the early days analog control systems were commonly found in the industrial environment. Large process facilities started using electronic process controllers for regulating continuous variables such as temperature, pressure and flow rate. They started to use mathematical and visual methods for addressing the problems associated with designing complex control systems. Soon they started using models for system identification, for synthesising a controller, and for simulating the combination.

As is the case in other domains, the control engineers are under pressure to finish their projects within a tight schedule and at low cost. The traditional text based approach to designing embedded systems was not suitable anymore [83], and new approaches emerged. The most popular approach is Model-Based Design (MBD)[4] [49]. In MBD, a system model is at the center of the development process, from requirements development, through design, implementation, and testing. The model[5] is an executable specification that is continuously refined throughout the development process, and simulation can be done through model elaboration. When software and hardware implementation requirements are included, such as fixed-point and timing behaviour, one can automatically generate code for embedded deployment and create test benches for system verification, saving time and avoiding the introduction of hand-coding errors. A popular tool for MBD is Simulink[6] [68] and MATLAB [20] is often used as a modelling language.

To summarise, the following four aspects can be uniquely related to MBD [49]:

1. executable specification with models,

2. design with simulation,

[4] To avoid mixing up the category of flavour 'model-based design' with the concrete 'MBD' methodology, we consequently spell the flavour with lower case (see Figure 3.4).

[5] The model includes the relevant parts of the control software as well as the to-be-controlled system.

[6] Simulink is a graphical block diagramming language tool and a modelling language as well.

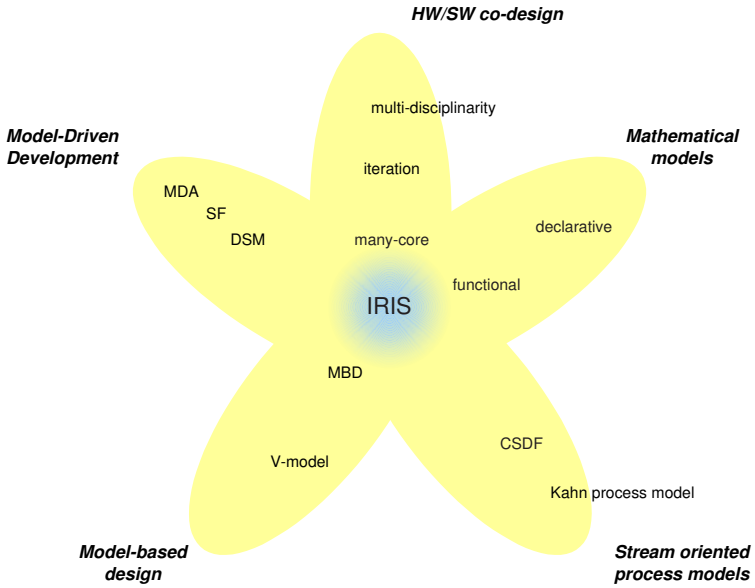


Figure 3.4: Model related design approaches come in flavours. Sample methodologies are indicated.

3. implementation with code generation,

4. continuous test and verification.

Model-Driven Development. Model-Driven Development (MDD) is a software engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software [60]. Its application areas include information systems in general but also embedded real-time control systems. The Object management Group, Inc. defines a particular realisation of MDD using the term *Model-Driven Architecture* (MDA) [75]. The MDA provides a systematic framework to understand, design, operate, and evolve all aspects of systems. MDA is based on modelling separately technology-independent and technology-dependent aspects of a system, by describing them in separate models. *Unified Modeling Language* (UML) [75] is a language for requirements analysis and system specification in MDA. UML is used in modeling event-based systems (object orientation) but it is not well suited to model data flows in streaming applications [19]. The latter propose a design flow which provides an automatic mapping to other models such as Simulink [68]. While platform independency is the primary goal of MDA, Microsoft's *Software Factory* (SF) on the other hand focusses on product line development [33]. It uses *Domain Specific Modelling* (DSM) [33] to narrow into the problem as well

as platform domains to facilitate for example code generation. For this purpose a so called *Domain Specific Language* (DSL) [33] is developed, that is especially created for this task. Although domain specific modelling can be useful for our purposes, the scale of deployment is too small for SF and DSL to be a serious contester.

Stream oriented methodologies. Stream oriented methodologies concentrate on the mapping of sequential pieces of code running in parallel on multiple cores, and the *Kahn Process Network* (KPN) is a popular model of computation [119]. An extension of the simple synchronous dataflow is the *Cyclo-Static DataFlow* (CSDF) paradigm, that supports algorithms with a cyclically changing behaviour, and still allows for static scheduling [12][123]. However, the methodologies lack the support for programming the often heterogeneous hardware architectures of the individual cores.

Mathematical models. Mathematical models are based on a mathematical formulation of a problem. A specific programming style, that fits naturally to these models is *declarative programming*. A program is 'declarative' if it describes what a certain program does, rather than how to do it. For example, HTML web pages are declarative because they describe what the page should contain (title, text, images) but not how to actually display the page on a computer screen.

According to a different definition, a program is 'declarative' if it is written in a purely *functional programming* language, logic programming language, or constraint programming language. The term 'declarative language' is sometimes used to describe all such programming languages as a group, and to contrast them against imperative languages.

Functional programming is a programming paradigm that treats computations as the evaluation of mathematical functions and avoids state and data updates. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state. Functional languages include Haskell [43][13], Erlang [3], Lisp [1], Scheme [109], ML [89], F# [88] and A Programming language (APL) [66]. Functional programming languages, especially purely functional ones, have largely been emphasised in academia rather than in commercial software development. However, notable functional programming languages used in industry and commercial applications include Erlang (concurrent applications), R [29] (statistics), Mathematica [114] (symbolic math), ML [89], J [113] and K (financial analysis)[7], and domain-specific programming languages such as XSLT [93].

[7] Dennis Shasha: "K as a Prototyping Language", 1998, `http://www.cs.nyu.edu/courses/fall02/G22.3033-007/kintro.html`.

3.3 Model-Based Design as a basis for IRIS

The purpose of this section is to discover what exactly is required for a firmware development methodology for many-core architectures. We start with an inventory of the gaps left by the hardware/software co-design developments, see Section 3.3.1. Since model-based design covers a lot of desired aspects we continue with emphasising the advantages of MBD. Before this is done relevant industrial requirements are listed, which are used to emphasise some aspects. In this way we are able to formulate what we can adopt from MBD, but also what problems remain to be solved (Section 3.3.2).

3.3.1 Extending Model-Based Design

In targeting embedded applications for a commercially available processor, the conventional hardware/software co-design methodologies have open problems. They are listed below.

1. Lack of one integrated coherent framework. The described methodologies are either very generic (Y-chart) or only address parts of the development trajectory. These multiple discrete and non-consecutive parts obstruct – from an integral point of view – a progressively modelled system.

2. Lack of interactive feedback on current functional design decisions (only post-mapping tools). There is no early design validation, no progressively expanded test cases, which consequently results in late system level verification. The same applies to the monitoring of extra-functional parameters that is decoupled from the development itself and also implemented via non-interactive post-mapping tools.

3. There is no support for evaluating design alternatives (design space exploration) in a single language environment, which is a necessity for progressive (evolutional) development.

4. No focus on reduction of time to market, no software generation (because of the one-sided focus on hardware development).

5. Traditionally hardware/software co-design covers only hardware designs. Software and in particular software economics for programmable processors has not been given the desired attention.

To conclude this section we mention a few relevant requirements for the development of a methodology from the side of industrial embedded systems development.

A. Effective methodologies for industrial applications have to reckon with the following success factors [11]: (1) they should be easy to learn, (2) easy to use, (3) short cycle time of the application to the model, and (4) should have reasonably accurate predictive power.

B. The reduction of time to market is a hot item [84], and tooling should support rapid prototyping and code generation. For example as a tool supplier, The MathWorks [83], recognises the importance of time to market and positions its products MATLAB and Simulink in the center of MBD[8].

C. Companies expect a significant increase in efficiency by using these tools (in a particular case even more than 200% is reported [28]). Furthermore an increase in the maturity level of developed functions is expected.

3.3.2 Remaining problems

In general most of the above mentioned hardware/software co-design remaining problems $1 \cdots 5$ are more or less covered by MBD. Also, from the list of requirements for industrial embedded system development, all three ($A \cdots C$) apply to MBD. The problems with MBD, however, are related to the specifics of the application domains: there are large differences between the control engineering domain and the domains covered in this thesis. These problems form the basis for the requirements for a new, to be created, methodology for many-core firmware development. The problems are:

a. The targeted application domain for MBD is mainly control engineering. Programming of many-core systems for our application domains, however, involves more intra-system modelling (software and hardware) than control engineering applications require.

b. No attention is given to hardware modelling in an early design stage (as is e.g., essential in power efficient architectures). Even the opposite is true: it is quite common that hardware enters the control system design at a late stage.

c. There are no specific design roles for trading functionality for (hardware dependent) extra-functional properties.

d. There is no explicit support for reuse (as a software engineering aspect). Reuse has a positive effect on quality of code and on code generation.

e. Many-core hardware is not considered as target in MBD (yet), until now only code generation exists for Digital Signal Processor (DSP) or for FPGA.

f. MBD has no built-in mechanisms for explicitly minimising the integral development time (for example by considering a risk based development process).

g. Problems in firmware development for heterogeneous many-cores need a mixture of function abstraction and data abstraction. The functional languages (such as Haskell or J) offer both kinds of abstraction. MATLAB [20] only offers data abstraction and is used in high level modelling (control engineering) and for regular parallel processing (DSP applications).

[8] These products use languages: M is the language of MATLAB and Block Diagrams that of Simulink

3.4 Starting points of IRIS

The following departure points, which are inspired by the previous sections, are taken for the IRIS methodology and earlier experience[9]:

a) Model-based. As introduced in Section 3.2.5 model-based design is a design process for complex, multi-disciplinary and multi-stakeholder problems. Mapping applications to many-core systems, the focus of this thesis, certainly falls in this category. In Section 3.2.5 and [49][53] the four basic aspects of the dominant methodology MBD are mentioned, which cover most of the requirements for IRIS. Therefore, MBD is selected as a point of departure.

Another aspect of MBD is the choice of programming model. Patterson [39] identified the need for new programming models if they simplify the programming of highly parallel many-core systems. New programming models use for instance statistical techniques (Energy-Based Model (EBM) [81]), or learning technologies like Artificial Neural Networks (ANNs) [77], can significantly reduce the amount of work involved (see Chapter 4 and 6 respectively). When new programming models are not applicable, and a sequential specification of an application is available, it is essential to (partially) remodel this specification, in order to better fit the target many-core hardware.

Unfortunately very often hardware is described in terms of a sequential programmer's view and this may hide the essential parallel features of the very same hardware. The explicit exposure of the parallelism of the hardware to the programmer is a first step in closing the semantic gap[10] between specification and hardware. An additional problem is the often poor state of the hardware documentation and programmer's guide. It is not uncommon that documentation contains errors and/or that the number of undocumented features is relatively large. Hardware models are needed to prepare the developer for the mapping task.

b) Early Feedback. Designing firmware is not a linear process. Although the major design flow is in the direction of the ultimate realisation, in practice lots of detours (caused by various reasons) have to be taken. For this purpose a feedback loop is used, see Figure 3.2. This loop is closed by the developer, who identifies the correct design *stage* to adapt, performs all necessary corrections and proceeds with the normal design flow. For the moment it is sufficient to know that a stage is a small unit of development time (see departure point e) and Section 3.5.1). An example for such a detour is the mapping of parts of the application to the cores. In (heterogeneous) many-core systems the optimal mapping of application threads on cores depends on the specific details of the

[9] In the timeframe 1982-2001 several excercises took place that can be interpreted as cases for a precursor of IRIS [P5][P6], and [122], [112].

[10] The semantic gap, in general, characterises the difference between two descriptions of an object by different linguistic representations. In our case an abstract but executable specification in an architectural language and an implementation description in a target hardware language for a particular SIMD processor.

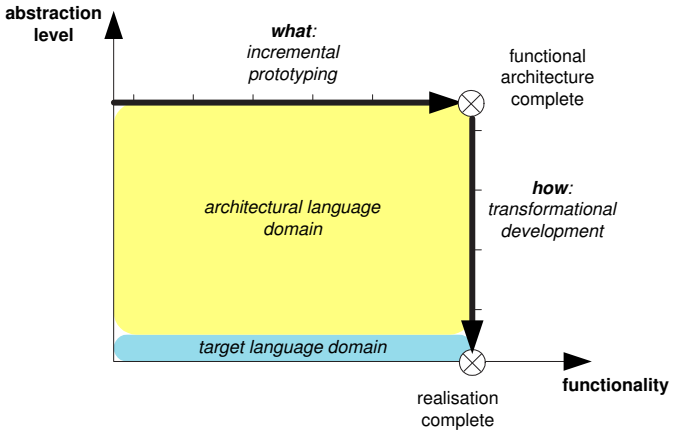


Figure 3.5: Incremental Prototyping and Transformational Development as design activities along the functionality and implementation design dimensions

selected cores and the algorithms used. Even the smallest change in an a priori partitioning can induce a feedback cycle through the various effected stages.

To minimise development effort it is essential that design flaws are detected as early as possible [17].

c) Feasibility and Risk analysis. Almost all projects are feasible – given unlimited resources and infinite time. However, the challenge in the development of realistic embedded systems is to deal with the scarcity of resources and pressing market windows. Therefore, it is recommended to evaluate the *feasibility* of a mapping as early as possible [98]. A lot of development effort can be avoided when an ill-conceived system or module is recognised in the early phases.

The technical feasibility is one of the most difficult areas to assess at early phases. Because in these phases objective, scope, functionality etc. are not yet clear, everything seems possible. Pressman [98] states: "It is essential that the process of analysis and definition be conducted in parallel with an assessment of technical feasibility". In this way early ideas of functionality (initial scope) may be evaluated by confronting these ideas with technology (tech probe).

The order of addressing or selecting a development step is important in reducing the risk of failure. The following development steps illustrate the importance of the order of: analysis→design→implementation.

Analysis. The design of a system is always preceded with an analysis of the problem to be solved (*why* before *what*). Overlooking a crucial requirement, for example missing a critical functional or attribute (non-functional) requirement is perhaps the greatest risk in requirement engineering [80]. Inadequate customer representation should be avoided and asks for a proactive

why-attitude.

Design. Functionality is determined by *what* the system should do and precedes its implementation (*how*) [14], see Figure 3.5. This figure describes the separation of two – often intertwined – concerns: the specification of the functionality and the development of the system (realisation). The required functionality of the system is developed during the incremental prototyping phase and leads to the *functional architecture*, still an abstract but executable specification. Next this specification is gradually expanded to target code in the transformational development phase. The strict separation of functional specification and implementation speeds up the development process. All elaborations are expressed in the architectural language except for the last stage: the translation to the target hardware programming language.

Implementation. The complex nature of programming many-core architectures and the tight constraints on system delivery, characterise the implementation as a risky process. Elaborating the various functions in the correct order for implementation (see the vertical trajectory in Figure 3.5) can mitigate the risk of a late delivery, or even worse: project failure. Often a Pareto analysis [98] is used for this purpose. Pareto analysis is the process of ranking opportunities to determine which of many potential opportunities should be pursued first. In order to minimise the propagation of errors to later phases we select the one with the largest risk. In this sense it can be a guide in the selection of the next function to work on. Functions are taken one at a time and transformed in a form that resembles their final implementation more, but not necessarily in the final form already. Such a transformation is done in a small development time interval, a so called *stage*, and corresponds to the succession of subspaces Q_i, that are mentioned in Section 3.2.2 and [15]. When a stage is finished a next function (may be the same) is selected for expansion by a new Pareto analysis. The succession of these subspaces eventually leads to a system that satisfies all requirements.

d) Monitoring extra-functional properties. Extra-functional properties are properties that characterise non-functional aspects of a system. These aspects can normally not be decomposed as easily as functional descriptions, but have to be 'measured' by post-mapping tools (see Section 3.2.2 [27]). Extra-functional properties can emerge at any moment in the development process as described in [15]. For some extra-functional properties, in particular the ones written down in the requirements, constraints are formulated that have to be respected. Others for example appear during implementation and are used for evaluating alternatives.

The functional model is coded in an architectural language with an appropriate level of detail that fits the current development phase. In order to monitor the extra-functional properties during development they should be modelled with a

similar level of detail. This could be realised by extending the functional model with extra-functional properties (e.g., timing[11]) or off-line processing for static analysis (e.g., software complexity measures). A typical course for such a property starts with a large variation in extra-functional properties in early stages and becomes more deterministic (small variation) in the end, satisfying all constraints (e.g., obeying required lower or upper bounds).

Once the relevant models – the functional as well as the extra-functional models – are in place, they can be evaluated to return quantitative results. This evaluation is done by a carefully orchestrated and gradually extended set of test cases or test suite[12], that gives the developer the assurance that the system-to-be will meet all relevant requirements for this stage. Not all models return quantitative results, for example first timing models and models for storage allocation might be better handled by qualitative considerations using a simple $space \times time$ scheme (in a 'textual' spread sheet). Typically the extra-functional models depart from qualitative models, which transform via probabilistic models to deterministic quantitative models.

One class of extra-functional properties that are monitored during this research is quality. Quality functions can be very diverse and may vary from for instance precision of computation to a perceptual optimisation criterion. A quite different extra-functional property is software complexity. Complexity measures can be obtained from the model descriptions during all phases of development. For example Nurminen [94] reports that simple Lines Of Code (LOC) metrics are useful in empirical algorithm comparison, which is suitable in design space exploration, see next paragraph.

e) Succession of subspaces and design space exploration. The development of a system can be viewed as the succession of subspaces $Q_i \subset Q$, where Q is the discrete and finite space of all possible digital systems. According to Boasson [15]: "Finding this succession is a difficult and not well-understood process", and he objects against the current practice that subspaces are implicitly defined every time a decision is taken. Clearly both an incremental development and an explicit design space exploration are needed.

To illustrate the relation between incremental development and design space exploration we first define a stage as a time interval, whose length represents the time it takes to develop the succession of $Q_{i-1} \rightarrow Q_i$. It is considered as the smallest unit of design[13]. Design space exploration describes the process of finding alternative strategies for realising the current stage. Because the winning alternative of such an exploration is coded directly into the model this also covers

[11] From a purely functional viewpoint timing is considered a non-functional property.

[12] The test suite, also known as validation suite, is a collection of test cases that are designed to show the existence of errors. In an indirect sense they intend to specify a behaviour by asserting relations between input and output.

[13] From experience we know that a time interval should last no more than a few hours.

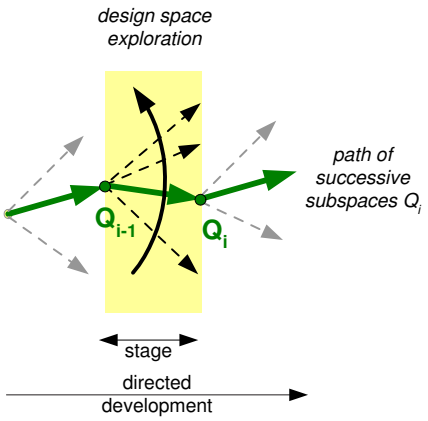


Figure 3.6: Relation of design space exploration and the succession of subspaces $Q_i \subset Q_{i-1} \subset Q$

the decision recording[14]. Figure 3.6 shows the relation of the design space exploration with respect to the succession of subspaces Q_i. Note that whereas this approach is described as a linear process, in practice the path taken is much more complicated, with decisions being revoked and other alternatives explored (cyclic development).

3.5 The IRIS methodology

Now that we have discussed the departure points, we can introduce IRIS gradually. First an overview of the methodology is given, followed by the architectural language that is used for the involved modelling.

3.5.1 Overview

The IRIS design methodology for deriving firmware for SIMD architectures does support different application domains and is strongly phased, see Figure 3.7[15]. In this manner IRIS supports the different development roles necessary for an effective and efficient development process. In our methodology we recognise three main phases: I) Familiarisation, II) Incremental prototyping, and III) Transformational development.

Familiarisation is specifically meant to vet both the problem and the target hardware that is intended to host the embedded system. Questions such as "*why* do we need an embedded system?", and "what is its scope?" are answered.

[14] Besides the generation of the target code one could also generate project history documentation.

[15] This figure is inspired by [104].

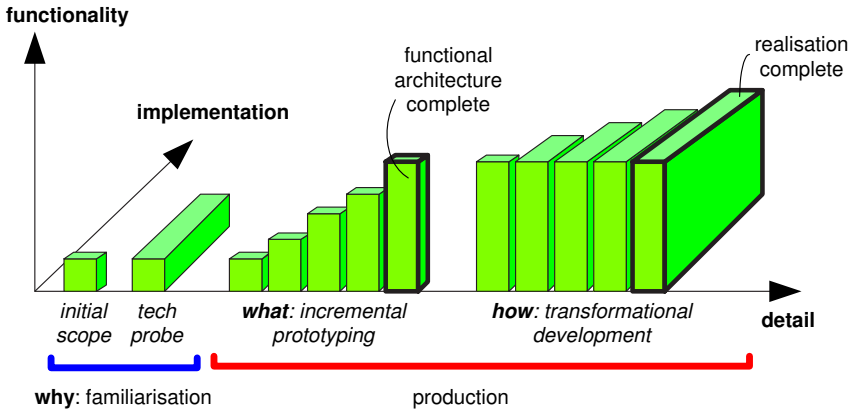


Figure 3.7: Design dimensions in IRIS

Incremental prototyping is concerned with *what* the system is supposed to do. It results in the complete functional architecture. Here the functionality is determined, strictly separated from the implementation. Finally, transformational development tackles the actual mapping – the *how* – to the target hardware.

As described in Section 3.4 the smallest unit of development is a stage, that should preferably be finished within a few hours. The above mentioned phases last longer and – in particular the transformational development phase – is relatively large in size. Beyond that, this phase incorporates various development roles and therefore groups stages in separate role-typical subphases.

Sequencing the stages is, in general, determined by the analysis - design - implementation flow. Particularly in embedded system design, however, deviations of this flow are quite common. These deviations can be initiated by – constantly monitored – extra-functional properties, of which the values move into a critical region. Figure 3.8 describes the global use-pattern of IRIS. In this figure the various phases are depicted in the normal design flow order (left to right), and this order shows the global development direction towards realisation. However, embedded system design has to respect constraints on (extra-functional) properties that cannot be foreseen. Insufficient performance of monitored extra-functional properties, such as exceeding an agreed memory budget, can lead to redesign. The tail of the arrows in the figure symbolise the probing of these properties, whereas the head of the arrows symbolise the earliest possible stage that has to be redesigned by the developer.

In Chapters 4, 5 and 6 the methodology will be illustrated via case studies.

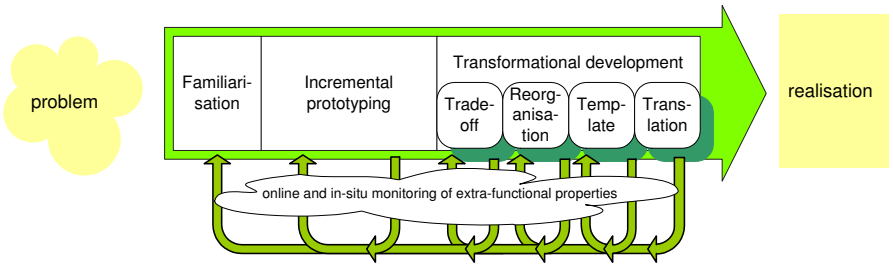


Figure 3.8: Global use-pattern of IRIS, ordered along its phases. Cyclic development is often induced by insufficient performance of monitored extra-functional properties.

3.5.2 Architectural language

First, we will state the requirements of and propose of a suitable architectural language. Second, we will go into how to use the architectural language and propose risk management as a guiding principle.

3.5.2.1 Requirements and proposed architectural language

One of the requirements of IRIS is the use of a single language for the modeling work in all phases. The language should satisfy a number of requirements. The language should be: (a) *flexible*, in the sense that it supports modelling of high level descriptions (close to mathematics) as well as implementation issues as data parallelism or even low level bit field assignments, (b) *compact*, since compactness of description is a virtue in reducing costs, (c) *executable*, to offer verifiability of work in all phases, (d) *interpretative*, in order to realise the needed interactivity (in, e.g., design space exploration), and (e) *general purpose*, to allow for creating auxiliary tooling, such as memory utilisation, performance monitoring or the automatic verification (by running the maintained test set).

In IRIS we propose a functional language (like Haskell [13] or J[16] [113]) as the architectural language because it fulfills the requirements mentioned above. We propose a single architectural language for all phases that supports multiple roles because of *ease of use*: (1) one single framework is better facilitated by a single language provided the different roles involved can be served adequately, (2) a language close to mathematics, can facilitate precise specifications as well as serve as a language that facilitates concise description of implementation details, (3) code refactoring (for example in the Template subphase) is hindered when interfaced over cross-language domains, and (4) one language to investigate *and* document suitable alternatives is more beneficial than using different languages.

[16] Jsoftware Inc.: "High-Performance Development Platform", 1990, `http://www.jsoftware.com/`.

The fact that various necessary development roles can be performed in an easy interactive manner, makes firmware development an enjoyable activity. Because Haskell is better known than J, we select Haskell to illustrate IRIS in this thesis, although all case experiments have been done with J. In this thesis we conveniently use a pseudo form of Haskell. We deviate for example from Haskell by allowing upper case names, by using a free form of case statement and taking some freedom in expressing conditional statements (syntactic sugaring).

3.5.2.2 How to use the architectural language

This section is concerned with the use of the architectural language. For IRIS this use is subjected to the minimisation of the development time. Not only the selection of the programming style but also the order of the elaboration of stages (succession of subspaces, Section 3.4) plays a role in a minimum development time.

The problem of developing a suitable implementation for the required system originates from the fact that the design space is immensely large. Even when the requirements were fully known and understood there is no clear trajectory that brings us from start to the complete implementation. In [15] a phased approach is described in which subsequent architectures are generated, a process that is directed by a set of (extra-functional) properties. These successive architectures evolve in the end to an architecture that is 'frozen' as a framework (the software architecture) for subsequent design and implementation. This pattern is also followed in IRIS and a crucial role is played by the architectural language. The main question to be answered for IRIS is: *How can the architectural language help in finding a suitable implementation?*

A suitable implementation includes functional as well as extra-functional properties. Not only properties that describe the state of the product but also the state of the development process itself. Timely delivery (time is a design-goal) and project costs within budget, are also important factors to cope with. Patterson [39] describes the problem of finding the optimal programming model as a balance of the competing goals of *productivity of design* and *implementation efficiency*, see Section 3.2.4. He concludes with a remark that the ability of the programmer to exploit the power of future many-cores is the real key to the success and implicitly indicates that maximising performance or minimising dissipated energy is of lesser importance. So with respect to the used architectural language an important trade-off has then to be made between productivity of design and implementation efficiency. This trade-off reflects in the two roles an architectural language should have (as mentioned in Section 3.5.2): it should support the modelling of high level descriptions as well as implementation issues. The first role can best be carried out by a functional programming style, the second role by an imperative programming style. Because of the pressure on development time, the choice for the cross-over point is important. Although the choice depends on a lot of factors (e.g., culture, education, situation) the cross-over point will

eventually be determined more by the strict deadline than personal preferences of the developer.

Another issue that is related to productivity of design is the order of selecting the stage to elaborate, and *Risk management* can be used for this purpose. In general, risk management is a structured approach to managing uncertainty through risk assessment, and developing strategies to manage it. In our case this means an assessment per stage including a Pareto analysis (Section 3.4) and the selection of the most risky next stage. This is the reason why we choose to base IRIS on reducing the risk that the system is not delivered on time or not even delivered at all.

3.6 Phase I: Familiarisation

We can now describe IRIS as depicted in Figure 3.7 in more detail in the following three sections. We start with the familiarisation phase. The goal of this phase is to come up with a provisionary demarcation of the system boundary and build some confidence on the feasibility with respect to the intended hardware. This corresponds to the design activities normally deployed between the behavioural and the structural domains in the Y-chart methodology [50] (described in Section 3). The physical domain is absent in our approach since we assume that the (many-core) hardware technology is already available.

We start with the vetting of both the problem (initial scope) and the intended candidate hardware architecture(s) (tech probing). In order to maximise the degree of freedom for system development an abstract 'mathematical' description is made of the formulated problem. At the same time models are made of the target hardware – partly based on the documentation and sample programs provided by the hardware supplier – to better understand its behaviour. This also involves opening up the parallel facilities of the target hardware, often hidden by the imposed C-programming model. Another issue concerns the documentation of the target architecture, which is not always in perfect condition. Driven by the need for clarity on the relevant parts, 'reverse engineering'- like activities lead to modelling that corrects relevant errors in the documentation, or even finds undocumented features.

Both activities use the architectural language, but in very different ways. The functional programming style is preferred to get a clear view on the context, and to model some interactions with the clear goal in mind to divide the concepts in two sets: system concepts, or environmental concepts. System concepts are directly related to the system being built, whereas environmental concepts are built to handle the interfacing to the outside world and for the system verification framework (test drivers, actual verification). The two sets implicitly define the system boundary demarcation.

In vetting the target hardware architecture, however, the functional as well as the imperative programming style are used. Familiarisation with the typical usage

of: instruction modes, parallelisation, data transport, inter-PE communication, memory hierarchy and sizes, is done by reading the manuals and following the – by the manufacturer provided – demo programs. Although the imperative programming style will be the preferred style, the global aspects can be handled better with the functional style.

Near the end of this phase, when sufficient confidence has been built up in both application and hardware architecture, a first feasibility study is conducted to evaluate the target hardware architecture(s) against the scoped problem. After a small risk assessment at least one feasible hardware architecture is identified and one is chosen provisionally. The final choice – including the setting of some parameters such as number of processors, clock frequency, size of memories – is done after the next phase. Actual code production consists of the following two phases: *incremental prototyping* and *transformational development*.

3.7 Phase II: Incremental Prototyping

The goal of this phase is to establish the specification of the system. This phase leads via a number of intermediate steps to a complete specification, the *Functional Architecture*. Furthermore, it leads to a *validation test set*, a baseline set used in the following phases (or in this phase in case of redesigns). Incremental prototyping (also known as evolutionary prototyping) is quite different from *throw away prototyping* [30]. The main goal when using incremental prototyping is to build a very robust prototype in a structured manner and constantly refine it (using design space exploration), whereas throw away prototyping is used for learning purposes only (and the end result is always discarded). In our approach the prototype is the specification.

The functional programming style in the architectural language is preferred for this task. Essential is to keep the functional description concise since this is the basis for the coming development. A typical activity that supports conciseness is *refactoring*[17]. Since refactoring is a purely functional development activity, the functional programming style is most adequate to use.

This functional specification is executable –as are all the intermediate steps–, is independent of the target hardware, and serves as a live description of the system. The functional architecture marks an important milestone in the customer-architect co-operation. At this point we know the desired functionality of the system and we can turn to the transformational development, which is hardware architecture dependent.

[17] In agile developments, refactoring a source code module means modifying without changing its external behavior, and is sometimes informally referred to as "cleaning it up".

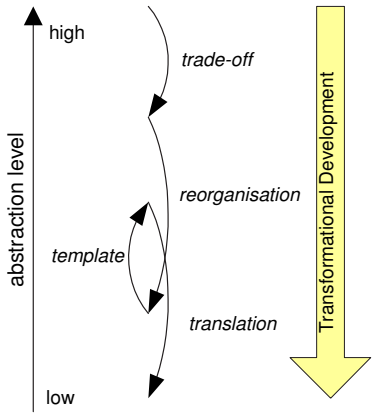


Figure 3.9: The layering of the Transformational Development phase

3.8 Phase III: Transformational Development

This phase consists of behaviour preserving transformations (except for the first subphase: the trade-off subphase), which progressively involves making design choices determined by the hardware architecture used, see Figure 3.7 (right part). The validation test set is progressively extended at the same pace as the functional decomposition. This allows intermediate checking against the current complete validation test set. The Transformational Development phase exhibits the following subphases: *Trade-off*, *Reorganisation*, *Template*, and *Translation* (see Figure 3.9). See also [120].

3.8.1 Trade-off Subphase

The goal of this subphase is to deliver a *golden reference*, which can be used for validation purposes for downstream transformations. Because of hardware limitations often concessions have to be made to the accuracy of computations, bit-width of variables, or even computation speed. Because of possible (mostly tiny) concessions made to the functionality, this subphase involves, besides architect and implementor, also the customer. Design space exploration has to address new aspects as implementation, deviations in functionality and quality, and the evaluation of the consequences these deviations induce. The various results of this exploration can be used to produce Pareto trade-off points to be used in e.g., run-time scheduling [111].

Regarding the choice of programming style preferably the functional style should be used. The architect should have knowledge of the target hardware but that does not necessarily imply the use of an imperative programming style. For

example limited precision by truncation of bit-fields can be described adequately in a functional style.

3.8.2 Reorganisation Subphase

The goal of this subphase is to rephrase the executable model in a top-down manner such that it is more geared towards the chosen hardware architecture. This and following subphases involve only behaviour preserving transformations. Because of the strong hardware coupling this subphase cannot always use the functional programming style, although it is the preferred style. We mention a few typical issues that are addressed in this phase[18].

Globalisation. The hierarchical structure of functions as well as data structures are adapted to better match the SIMD architecture. For most SIMD architectures this implies the flattening of both structures. Functions are gathered within a fewer number of functions to reduce/avoid function call overhead and variables are made global because the SIMD hardware typically does not support any form of advanced memory management.

As a result the variables have to become globally visible to all functions. To prepare for an effective allocation the variables are subjected to a variable naming convention (see Resource allocation). Globalisation is performed first since all following items depend on this choice for the allocation of variables.

Resource allocation. As is often with SIMD architectures, the memory of each Processing Element (PE) is limited. The static allocation of variables to memory fields is critical, IRIS supports the choice for location and width. This can vary between a simple spreadsheet containing a qualitative allocation overview (e.g., in the beginning) to a tool that warns for conflicting allocation attempts. Appropriate register naming conventions can serve this goal (call-by-name[19]). For example for a variable that is mapped to a bit-field the specification could look like `<name>_<start_position>_<length>`, so that position and length can be checked. If one of the following issues introduces a new variable, its allocation has to be validated again.

Resource sharing. Often the time-sharing of memory fields for different variables is a solution for lack of storage space. This implies processing in a strict order. A simple *space* × *time* overview can already be of help. For later phases IRIS supports a full emulation of the PE's memory, that allows for overlapping bit-fields and real detection of allocation faults. Full emulation is the only effective way to handle this. As mentioned in the introduction the

[18] Some of these issues have been inspired by the software washing machine concept [25].

[19] In the call-by-name evaluation strategy, the arguments to functions are not evaluated at all – rather, function arguments are substituted directly into the function body and evaluated when needed.

programmer takes the role of 'intelligent' compiler. Especially in resource sharing, where the *space* × *time* scheduling can be very complex.

Expansion. Expansion of model code is a way of closing the gap between the current model and final realisation in a step by step manner. This applies to computation as well as program flow. For example a square operation may not be available but can be expanded in a multiplication (expression reduction). Another example is the transformation of implicit iterations or recursive definitions by explicit loops or even the complete removal of a loop (loop unrolling). As a final example we mention the exchange of nested loops, turning the inside out, to better match with the processor's capabilities (see *transformation laws* in Chapter 4). New claims on resources during expansion of for instance expressions can potentially influence allocation and sharing of all bit-fields.

Expression optimisation. Expressions that have a relative large computation time are rephrased such that they perform better. For example a multiplication with a constant can be written as a repeated addition or as an optimised sequence of shifts, additions and subtractions [18]. Another example is looking for common subexpressions and by reusing stored computed intermediate results, for instance stored in a Look Up Table (LUT). This introduction of state influences allocation and sharing of bit-fields.

Tiling. Building cost efficient systems requires a sound balance between performance and cost. Since problems often require more storage than the target hardware can offer, repeated processing of *tiles* offers an economic solution. Tiles are – possibly overlapping – parts of a problem's data space, that fit in the target hardware system. Tiling requires the programmer to explicitly divide the storage requirements of the application in a static and dynamic part, which influences memory allocation and memory sharing of bit-fields.

Normalisation. To prepare the recognition of reusable code templates (see next section) we rewrite similar code fragments in a uniform way. For example loops, whether with constant or variable loop count, are presented in a limited number of ways[20]. This also will facilitate translation (Section 3.8.4) even when the fragment is not converted into a template.

Precomputation of constants. Some constants can be computed at compile time or during the initialisation phase and e.g., stored in a particular memory field administered per PE, or in a LUT (in case the target hardware supports this). Nonetheless constants consume storage thus need a check on allocation and sharing of bit-fields.

[20] The target hardware architecture often has a limited number of loop control instructions.

To conclude we remark that these issues can be addressed in many ways. We need to explore the design space many times in order to come up with suitable intermediate implementation models.

3.8.3 Template Subphase

The goal of this subphase is to identify reusable components that can reduce current and future work. This not only includes reusable macros for code fragments or even complete modules but also includes support for instruction coding and translation. As time progresses, *experience* translates into more powerful components/templates (bottom-up). Templates are intelligent pieces of interactive functionality that serves several roles. First of all, the functional behaviour of the involved SIMD processor instruction(s), *functional emulation*, should be properly modelled. All relevant effects and (perhaps undocumented) side effects of the hardware architecture should be modelled. Second, obeying the *calling conventions* for all relevant type of instructions should be enforced. For example, instructions can have restrictions on data width or type of memory fields. Finally, the syntax of the template-call should be rich enough to enable *automatic generation* of the target code for this call (facilitating the translation subphase). Both the calling convention and the translation support use a special calling convention for variables to express the allocation of memory to the variables. The same calling convention scheme as suggested in the Reorganisation phase (Figure 3.8.2) is used.

During the development phase, knowledge is being built up for the construction of a more advanced higher level compiler than the simple one-to-one translation (see transformation laws in Chapter 4). Successive modelling is needed to develop the not yet discovered transformation laws, so that the benefit of using the functional programming style is not frustrated too soon by hardware details.

The development direction is bottom-up, showing the abstraction of a code fragment (as a template instance) to the template. The combination of reorganisation and template subphases address the platform based issues [72], where successive refinements of 'specifications' meet with abstractions of potential implementations.

3.8.4 Translation Subphase

The goal of this subphase is to realise a smooth transition to the target hardware. This involves a fully automatic translation from the model of the design coded in the architectural language, following the template and all earlier subphases, into the native target language (mostly C+intrinsics) of the chosen hardware.

In the translation subphase, a functional programming style is preferred. For the simple one-to-one translation a compiler generator (e.g., the Yapp[21] parser

[21] Yapp (Yet Another Perl Parser) is a collection of modules that can generate parsers with the perl object oriented interface.

generator) is used or a dedicated function can be written. Template based translation, in which a simple template call is expanded into multiple target source code lines, is more effective since it can drastically reduce the size of the model to be translated.

3.9 Summary

In this section we summarise the IRIS methodology by structuring it in two different ways:

- By relating the described development phases (familiarisation, incremental prototyping, and transformational development) to the – up until now – implicitly mentioned development roles, which themselves will be explicitly described.
- By positioning IRIS as a hardware/software co-design activity and use the well known Y-chart as a template.

Development roles in the various phases. In the previous sections, where methodology is described in a phase-wise manner, the various development roles are implicitly described. Now, at this point, we can identify the roles and relate them explicitly to the phases in IRIS.

We identified five groups of development roles:

1. The *functionality* is prototyped and transformed into firmware that satisfies all requirements.
2. One of the most difficult roles in firmware development is the management of the *extra-functional properties*. This is caused by the lack of good decomposability of non-functional aspects in designs. The identified extra-functional properties in the described cases are: memory, (execution) time and quality. These properties are often constrained by requirements and therefore are part of the various feasibility tasks during the development phases.
3. *Reuse and translation* cover the desire to support manual translation by automatic tools. As experience is built up in implementation exercises during the various phases the knowledge of problem decomposition and (reusable) components can be turned into increasingly sophisticated translators / compilers.
4. *Design space exploration* gives foundation to the large number of design choices that have to be made. Decision recording documents the choices and provides a bases for potential rework when for instance the specifications change.

5. *Verification* is the role that gives legitimacy to the current development stage. If verification of the current stage fails then the stage or previous stages have to be reconsidered. The major risk is the absence of good fault coverage and is related to the fact that the absence of a fault can be difficult to show. A good remedy is to extend the test set with specific test cases for each developed stage. Note that at each stage the complete validation test set is executed. A subset of the validation test set is the golden reference: it only checks the functionality (which does not change after the trade-off subphase).

		Roles						
Phase (with case chapter)	Description	Functionality	Extra-functional properties (feasibility concerns) e.g., memory, time, quality			Reuse and translation	Design space exploration	Verification
			memory	time	quality			
2. Familiarisation	investigate hardware models, establish system boundary, provisionally choice of hardware	functionality ↔ environment,	1^{st} estimate feasibility: qualitative model (spreadsheet)		investigate quantitative models			
3. Incremental prototyping	establish the function, determine parameter value range	define (new) functional model	define quantitative model	define coarse quantitative model	define quantitative model		record	set up test environment, determine reference test set (→ validation test set)
4. Transformational development	global system considerations, choice of hardware		2^{nd} estimate feasibility: quantitative model (e.g., spreadsheet), scalability analysis				record	
4.1. Trade-off subphase	make concessions	functionality ↔ extra-functional properties	memory ↔ functionality	time ↔ functionality	quality ↔ functionality		record	check against reference test set, determine golden reference
4.2. Reorganisation subphase	memory and time allocations		globalisation, resource sharing, expression optimisation, tiling			normalisation	record	check against golden reference and the (extended) validation test set
4.3. Template subphase	reusability, calling conventions					identification of reusable components	record	check against golden reference and the (extended) validation test set
4.4. Translation subphase	build an automatic translator to target code					template translation	record (translator)	check target port against golden reference and the (extended) validation test set

Table 3.1: Specific development roles per phase

In Table 3.1 the roles are collected in columns, while the rows represent the phases. The left most column lists the main phases of IRIS with the corresponding section number of the three case chapters, and the second column contains a short description of related activities. The five roles are covered in the five last columns, and the extra-functional properties involve – for the in this thesis conducted cases – three concerns: memory, time, and quality. The normal flow of development starts with the familiarisation phase and processes them down to the translation subphase. Multiple stages (a stage is the smallest unit of development), fit in a phase, and a stage cannot be completed until all relevant roles have been processed. Rework breaks with the normal phase-wise order. Rework can for example be caused by customer-developer interaction, or enforced by a constraint on an extra-functional property that exceeds a limit. In this case a couple of stages (design decisions) will be rolled back, the correction will be performed and the work continues again from this position downwards. Note that the three cases only used three extra-functional constraints, however, there is no limit to the number of constraints. For example other constraints can be formulated on power consumption, code statistics for good programming practices, etc.

Stakeholder's concerns in the various phases. The 'product' of IRIS is not a hardware platform (which would favour the original Y-chart approach) but an optimised procedure to generate firmware for the selected problem domain and the selected hardware platform. One may view IRIS as a hardware/software co-design development framework. The *behavioural domain* of the Y-chart approach relates to incremental prototyping. The *structural domain* relates to the trade-off and reorganisation subphase. The familiarisation phase addresses both mentioned Y-chart domains in a broad sense and prepares for incremental prototyping and transformational development. Finally, the template and translation subphases can be viewed as on-the-job component generator and compiler framework respectively.

Therefore, the hardware realisation segment, also known as physical domain, is exchanged by a so called *firmware engineering* in the Y-chart, see Figure 3.10. The main problem in firmware engineering, as in software engineering, involves managing the complexity of the firmware. In IRIS this is handled by: the reuse of components, the support for instruction specific calling conventions, and the automatic translation (template and translation subphases).

3.10 Conclusions

IRIS can be characterised as a confidence-by-construction framework: it offers the application developer an incremental way of system construction, which converges to a target language implementation.

Interactivity and executability provide for early feedback, in particular on incorrect problem interpretation or design faults at the very moment in time that

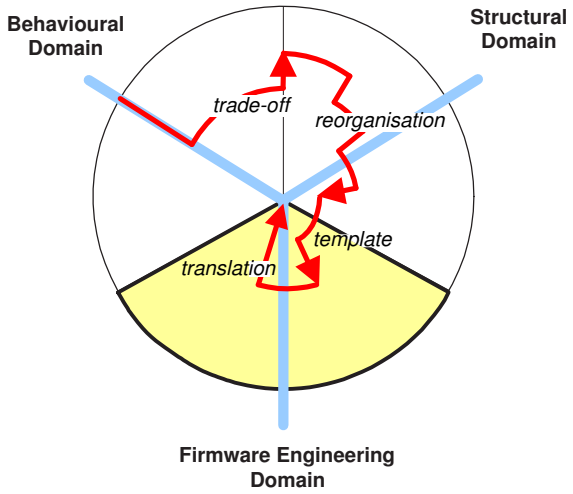


Figure 3.10: The four layers of the Transformational Development phase imposed on an adapted Y-chart. The Physical Domain has been exchanged by a Firmware Engineering Domain to better reflect a software dominant multi-stakeholder development process.

they appear. The functional model, as well as the models for the extra-functional properties (e.g., execution time), are used to trigger those faults. The models are driven by carefully selected validation test cases to obtain quantitative results, that subsequently can be compared with previously validated test cases.

In case of design changes, models of previous stages can serve as a solid base.

Decoupling the development language from the target hardware architecture language offers freedom of choice for migration to different target hardware architectures.

Design Space Exploration and the Decision Recording during development raises quality and takes less time because the evaluation of design alternatives can be done *in situ*.

All this is realised by using a single language based development framework for the entire trajectory, and in this way we lay a foundation for our integral IRIS framework. A functional language (such as Haskell or J) is a good option for such an architectural language. Because Haskell is better known than J, we use (a pseudo form of) Haskell to illustrate IRIS in this thesis.

CHAPTER 4

Case: Stochastic Image Quantisation

This chapter describes the results of the mapping process of stochastic image quantisation on a massively parallel processor. Stochastic image quantisation can be modelled in a parallel way. The parallel version on a dual Linedancer system is 128× *faster than the sequential implementation of the algorithm on a Pentium processor. Moreover, the parallel version has better scalable properties and offers easier control of quality improvement. The code of this case is developed with the proposed evolutionary development methodology.*

4.1 Introduction

A lot of low level image processing functions exhibit massive parallelism, for example *early vision* [69], a part of the Human Visual System (HVS) [56]. This part of the HVS comprises the low level and "hard-wired" processing performed on all pixels, and is highly parallel of structure. A key property of these systems is the simplicity of the involved processing. This thesis claims that applications that have functionality that exhibit massively parallelism should be (partly) remodelled in order to reduce the complexity of the system as well as to better exploit the potential of modern many-core processors. To support this claim we selected a simple inherently parallel processing algorithm in the context of *business graphics* and a many-core hardware architecture to implement it. This combination illustrates the potential of many-core processing for this application domain.

This case is included because it demonstrates both the power of simple parallel processing models, and the natural mapping to massively parallel hardware.

Major parts of this chapter have been published in [P3].

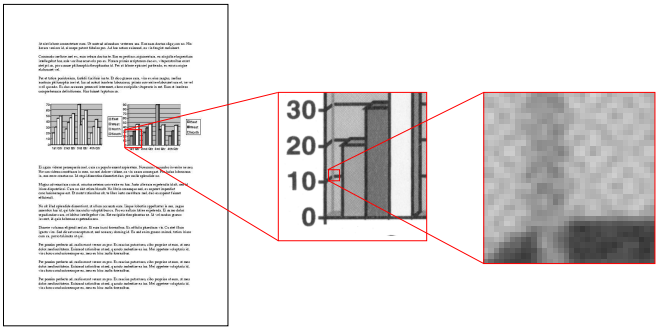


Figure 4.1: Typical office scan containing text and charts. Scanning introduces image degradation: the number of unique grey-values increases from 10 to 229.

Furthermore, the size of the case is small enough to demonstrate the complete trajectory of the development methodology.

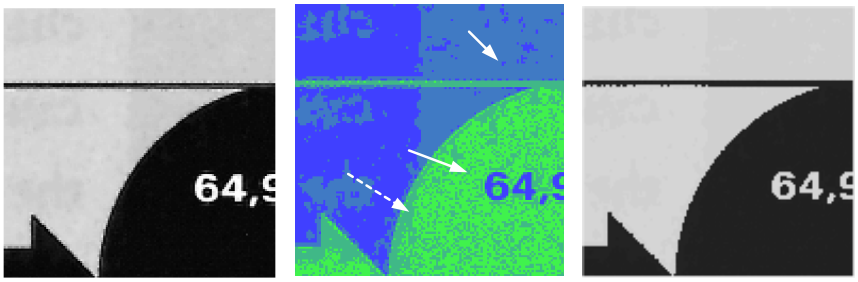
The structure of this chapter follows the proposed methodology. Section 4.2 covers the *familiarisation* phase and introduces various concepts of *(stochastic) image quantisation* and *simulated annealing*. In Section 4.3 the *incremental prototype phase*, a functional description of the system is given and the mapping to the target hardware is prepared. Next the *transformational development* phase – covering the implementation on a Linedancer (Section 2.3.4) – is described in Section 4.4, followed by the results in Section 4.5. Finally, conclusions and recommendations are given in Section 4.6.

4.2 Familiarisation

The goal of the familiarisation phase is to build confidence in the feasibility of realising a stochastic image quantisation on a Linedancer. For this purpose the design space is probed along the major design dimensions, involving the functionality and the intended hardware architecture. For the first dimension this includes the business graphics domain and image quantisation basics (Section 4.2.1) followed by stochastic image quantisation (Section 4.2.2). The final section describes the results of a first order feasibility study (Section 4.2.4).

4.2.1 Business Graphics and Image Quantisation.

Business graphics are characterised by large areas filled with a single colour. This type of information, such as presentation sheets and charts (Figure 4.1), is often scanned in an office environment. During scanning the image is sampled, which leads to distortion. One of the possible distortions is blurring, a kind of smearing, with the effect that new colours are introduced in a scan. For example in



(a) Original image, 256 grey-values

(b) Resulting false colour image, 4 grey-values, dashed arrow indicates ringing, the others indicate speckles

(c) Resulting image, 4 grey-values, ringing just visible

Figure 4.2: Example of state of the art quantisation algorithm

Figure 4.1, rightmost image, the checkerboard pattern is an unintended result of the scanning process and is caused by a raster in the original business graphics picture. A controlled reduction of the number of colours in such scans is essential for the image quality and can be useful as a first step in image compression. This process is called *colour quantisation*. Popular quantisation algorithms include *median cut* and *octree* algorithms [48]. These algorithms use a statistical approach: they determine the frequency of occurrences of each colour and try to assign quantised colours using only this (frequency) information.

Image quantisation is a process with many applications, that have a need for segmentation of an image. Examples are: to increase the quality of half-toning (for example Section 5.2.2.5), compression purposes [59], and improving the quality of scanned originals (this chapter).

Quantisation reduces the number of colours in an image by assigning pixels to a limited number of *classes*. The basic problem in this chapter, is to recover a limited set of colours from a scanned business graphics original such that the result resembles the intended original – as opposed to its scanned version – as faithfully as possible. For simplicity we restrict ourselves in this study to grey-value images since this does not alter the essence of both algorithm and mapping. Figure 4.2 shows the result of a state of the art quantisation algorithm. To observe quantisation artifacts, the quantised image is visualised in *false colours*, see Figure 4.2(b). A false-colour image is an image that depicts a subject in colours that differ from those a faithful full-colour photograph would show. We use false colouring to magnify the differences between the grey-value of pixels, that are almost equal, such that these differences are good visible for human perception. Note for example the ringing around edges and the various speckles

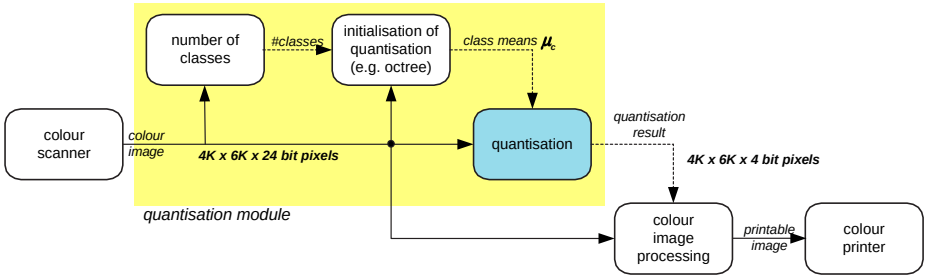


Figure 4.3: Context of the quantisation module

in Figure 4.2(b), showing the substructure in the light and dark parts barely visible in the grey-value representation as given by Figure 4.2(c).

In general image quantisation is a NP-hard problem [48]. Therefore, several heuristic approaches, which produce suboptimal results have been described. They can be divided into preclustering and postclustering quantisation schemes. In a preclustering scheme the colour space is divided into clusters of similar colours, depending on the distribution of colours in an image. For each cluster a representative is chosen (often the mean). Preclustering approaches, such as median cut or octree, are simple algorithms and have a reasonable quality. But for high end applications they cannot fulfill the increased demands for quality [48][69]. This has lead to the development of the so called postclustering approaches, which try to improve the quantisation by iteratively changing the quantisation means starting from an initial (pre)clustering [48]. For this purpose they use for example spatial or hierarchical relationships. As a result they offer better quality at the cost of increased computation complexity.

The context of the quantisation process is depicted by Figure 4.3. Here the algorithm starts from an initial (preclustering) quantisation, which is iteratively improved on. The initial quantisation needs a prespecified number of classes, which in turn is derived from the scanned image. The output of the quantisation module is used in the subsequent colour image procesing for a colour printer. Markov Random Field (MRF) and Modified Metropolis Dynamics (MMD) are examples of postclustering schemes, see Sections 4.2.2.1 and 4.2.2.3 respectively.

4.2.2 Stochastic Image Quantisation

The quality of quantisation can be further improved by including spatial (interpixel) relationships, since neigbouring pixels in business graphics often have similar grey-values. In this section we use an image processing model, MRF [69], known for its potential to reduce design complexity and its natural fit to the upcoming massively parallel embedded compute platforms (Section 4.2.2.1). Associated with MRF is *Simulated Annealing* [36], which is an efficient pseudo-stochastic

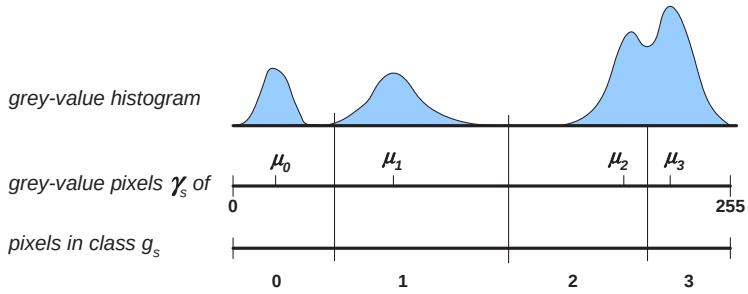


Figure 4.4: Estimation of classes with associated class means

procedure to solve combinatorial optimisation problems (Section 4.2.2.2). Present practice, however, makes such procedures unusable since they are far too inefficient when run on sequential machines. Therefore, we turn to massively parallel computing to implement a parallel version of MRF called MMD (Section 4.2.2.3).

4.2.2.1 Image models

First, we introduce some basic concepts, followed by two specific image models: *fidelity* and *regularity*. We conclude with a general image model based on the theory of MRF. The theory is described extensively in [69], the image model itself is taken from [110].

Fidelity Image Model. The scan process samples an original and returns a matrix of colours of pixels. In the context of this chapter we assume, without loss of generality, that all colours are grey-values. The matrix typically has a size of $w \times h = 5000 \times 7000$ pixels, whereas grey-values typically fall in the range 0..255. An example of a histogram for grey-values is shown in Figure 4.4. Image quantisation now proceeds by subdividing grey-values into classes. Let L be the number of classes, then in Figure 4.4 it is obvious that $L = 4$. Let s be a pixel, then γ_s denotes the grey-value of s, whereas g_s denotes the class to which s is assigned. The initial class assignments g_s^0 are determined by looking for each pixel which class fits best. But before doing this we need an estimate of the representative grey-value μ_c^0 per class, which can be derived for example by inspecting the mentioned histogram. A good initial class assignment g_s^0 is the class value that minimises the absolute difference between grey-value γ_s and its initial class representative μ_c^0:

$$g_s^0 = (c \mid argmin_c(\mid \mu_c^0 - \gamma_s \mid)), \tag{4.1}$$

where the function $argmin_c$ minimises its argument over all classes $c \in \{0 \cdots L-1\}$.

Let $\mathcal{S}$ be the matrix of all pixels s, then at any time, the mean of all grey-value pixels belonging to class c is μ_c:

$$\mu_c = mean\{\gamma_s \mid s \in \mathcal{S}, g_s \ = \ c\}, \tag{4.2}$$

where g_s is the class assignment for pixel s. The end effect for quantisation is that (the nearest integer to) μ_c is taken as the best grey-value for class c.

Stochastic image quantisation is an iterative process. On each iteration a new class c' is randomly chosen for a pixel s, and it is calculated whether taking $g_s = c'$ improves the quality of the quantisation result. This process is repeated long enough to allow for a sufficient sampling of the whole space of possible class assignments for all pixels.

One specific quality criterion of a pixel, given the class assignment function g, is the so-called fidelity. The full definition for fidelity $fid_g(s)$ is:

$$fid_g(s) = \ln(\sqrt{2\pi}\sigma_{g_s}) + \frac{(\gamma_s - \mu_{g_s})^2}{2{\sigma_{g_s}}^2}, \tag{4.3}$$

where σ_{g_s} is the standard deviation of class g where s is put in. Since the distribution parameters $(\mu_{g_s}, \sigma_{g_s})$ do not vary that much, the fidelity of a pixel s is mainly determined by the square of the difference between the actual grey-value γ_s of s and the associated grey-value μ_{g_s} of the class in which s is put.

For the complete image, containing values of fidelity $fid_g(s)$ for all pixels, the result is defined by the matrix:

$$Fid_g(\mathcal{S}) = [\![fid_g(s)]\!]_{s \in \mathcal{S}} \tag{4.4}$$

Regularity Image Model. Another desired property of business graphics is the occurence of large planes with a single colour or label. This property, called regularity, is optimised when the dissimilarity between neighbouring labels is minimised. That is, regularity indicates how well the grey-value of a pixel fits in its immediate surroundings.

Let $s = (i, j)$. Then we define

$$\mathcal{N}_s = \{(k, l) \mid \sqrt{(k-i)^2 + (l-j)^2} \leq R, (k, l) \neq (i, j)\}$$

as the *neighbourhood* $\mathcal{N}_s$ of pixel s. Thus, $\mathcal{N}_s$ contains all pixels within distance R from s, except s itself. See Figure 4.5 for a neighbourhood with radius $R = 2$, where the distance between two adjacent pixels is 1.

Let g_r be the label of a pixel in the neighbourhood of s. Then the regularity is defined by:

$$reg_g(s) = \mid \{r \in \mathcal{N}_s \mid g_s \neq g_r\} \mid \ - \ \mid \{r \in \mathcal{N}_s \mid g_s = g_r\} \mid \tag{4.5}$$

The lower $reg_g(s)$ is, the more uniform the neighbourhood is. The result for the complete image, containing values of regularity $reg_g(s)$ for all pixels, is thus defined by the matrix:

$$Reg_g(\mathcal{S}) = [\![reg_g(s)]\!]_{s \in \mathcal{S}} \tag{4.6}$$

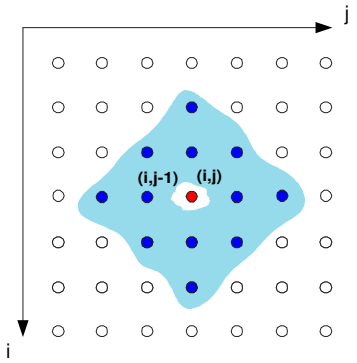


Figure 4.5: Pixels in a grid with neighbourhood. The pixels in the blue area are all neighbours of the central red coloured pixel $s = (i, j)$ within distance 2.

Markov Random Field. Both the regularity and fidelity components are combined to form a perceptual optimisation criterion. The basic form of this criterion is the weighted sum of both fidelity and regularity (for the whole image):

$$\begin{aligned} e_g(s) &= fid_g(s) + \beta \cdot reg_g(s) \\ E_g(\mathcal{S}) &= [\![e_g(s)]\!]_{s\in\mathcal{S}} \\ &= Fid_g(\mathcal{S}) + \beta \cdot Reg_g(\mathcal{S}) \end{aligned} \tag{4.7}$$

This matrix of weighted sums is denoted by the term *energy*, and originates from statistical physics modelling, and the term is consistently used in statistical optimisation, see [69][110]. 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.

A derived criterion is used in the general MRF image model where all elements of the matrix $E_g(\mathcal{S})$ are added and their sum is subsequently minimised. The energy of a quantised image g for the MRF approach is then given by the scalar:

$$\hat{E}_g(\mathcal{S}) = \sum_{s\in\mathcal{S}} e_g(s) \tag{4.8}$$

Minimising the energy $\hat{E}_g(\mathcal{S})$ over possible class assignments g_s per pixel, will raise the quality of the quantisation. As a consequence the *quality*[1] of an image is defined as the negation of $\hat{E}_g(\mathcal{S})$ (4.8):

$$\hat{Q}_g(\mathcal{S}) = -\hat{E}_g(\mathcal{S}) \tag{4.9}$$

[1] This operational definition of quality is not the same as the perceptual quality, however, 'has' the obligation to approximate it sufficiently.

The weight factor β is a positive model parameter for controlling – via the relative importance of $fid_g(s)$ versus $reg_g(s)$ – the homogeneity of the regions of the image.

The strong points of the MRF model for image quantisation as compared to traditional algorithms (such as median cut and octree) are: higher quality, simplicity and easier composability of different quality criteria. The weak points are: long computation times on sequential processors (high iteration count), and the difficulty in finding the optimal estimation of parameter β. The first results are promising but since the models are still in an experimental stage more experiments need to be performed before solid conclusions can be drawn. The scope of this thesis is not the model as such, but the mapping of the model to a parallel architecture.

4.2.2.2 Simulated Annealing

Finding *the* optimal label assignment g given a grey-value image γ is computationally difficult. However, reasonably good solutions can be found by *simulated annealing*, an efficient procedure for solving combinatorial optimisation problems [36]. Before describing the simulated annealing algorithm, we first introduce the concept of *state*, which is equivalent to the label or class assignment g of all pixels in an image, and is introduced to improve the readability. The algorithm repetitively changes the state g, computes $\hat{E}_g(\mathcal{S})$ in (4.8) based on the quantisation *state* g and updates the state along the way. The algorithm searches a state in which the weighted sum, or energy, is minimal. States which do decrease energy are always accepted (*deterministic acceptance*), but occasionally also slight increases are accepted in order to escape from local minima (*probabilistic acceptance*). In general the combination of MRF and simulated annealing is considered a powerful generic framework that can be used whenever an optimisation model can be constructed of a problem. See for example half-toning in [52], an even more complex application than quantisation. For our purposes, however, the main advantage of this approach is that the algorithm can easily be programmed to run in parallel for all pixels, as will be shown in Section 4.2.2.3.

The simulated annealing procedure is coded in Algorithm 4.1 [69]. Besides g another state, $\hat{g}$ is introduced, which is almost the same as g except for one single, randomly chosen, pixel which gets a new value. An essential variable in this algorithm is T or *temperature*, named after related concepts in physics [73]. In the simulation T is merely a control parameter that controls the randomness; it is not a true physical temperature. Together with the starting temperature T_0 the variables C (cooling factor) and n (number of iterations) determine the so called *annealing schedule* (line 2 of Algorithm 4.1) [36]. The involved variables must be chosen very carefully to ensure an effective but also efficient optimisation process. [110][69] report values for the tuple (T_0, C, n) between $(4, 0.95, 580)$–$(1, 0.95, 1)$ largely dependent on the type of problem. For some rare problem cases only a single iteration ($n = 1$) is sufficient. A desired property of this procedure is the

controlled and slow transition from a pseudo-stochastic ("high" temperature) to a deterministic phase ("low" temperature). This transition corresponds to the transition from a broad search for minima to the homing in on one – hopefully the global – minimum. The algorithm repetitively searches for quantisation g, which minimise an energy function E_g(4.8). First, it is initialised with a random quantisation state g. Then, the state is changed a little, to $\hat{g}$ (i.e., change the class of a single pixel), and accepted if this change decreases the energy E_g. If the energy does not decrease, the new state $\hat{g}$ is accepted with probability equal to the Boltzmann factor

$$e^{-\Delta E/T},$$

which constitutes a temperature dependent threshold (see [36]). The new state $\hat{g}$ is accepted under the condition

$$e^{-\frac{\Delta E}{T}} \leq \text{Random},$$

which is equivalent to

$$\Delta E \leq -T \cdot \ln(\text{Random}),$$

where the function Random is a pseudo random number generator, that draws random numbers from the interval $[0, 1)$, and where ln stands for the natural logarithm. The righthand side can be seen as an energy threshold for accepting energy increases and can be abbreviated by *Acceptance Threshold* A_{th}:

$$A_{th} = -T \cdot \ln(\text{Random}). \tag{4.10}$$

The involved comparison can be found in Algorithm 4.1, in line 7. The parameter

Algorithm 4.1 Simulated annealing

```
1: g ← initialisation state
2: for T ← T_0, T_0 · C, ..., T_0 · C^{n−1} do
3:    ĝ ← Randomly change the quantisation of a randomly chosen pixel s
4:    ΔE ← Ê(ĝ) − Ê(g)
5:    if ΔE ≤ 0 then {Deterministic acceptance}
6:       g ← ĝ
7:    else if ΔE ≤ −T · ln(Random) then {Probabilistic acceptance}
8:       g ← ĝ
9:    end if
10: end for
```

T determines via A_{th} to what extent energy increases are allowed. When T is high (almost) all proposed states $\hat{g}$ are accepted, which results in the visiting of very diverse states. At lower T values the algorithm only allows transitions lowering the energy, approximating a hill climbing algorithm [61]. The temperature is decreased during the procedure in order to converge to a final solution.

The starting temperature T_0 determines how well randomly different quantisation states are visited. It should be high enough to allow the visiting of a sufficiently well spread number of states (preferably uniform sampled) in the first stages of the algorithm, but if chosen too high, the algorithm needs too many steps to settle down [36]. The cooling factor $C \in (0, 1]$ determines the rate at which the temperature decreases. If the temperature is decreased too fast, the algorithm can get trapped in local minima. Because T is high in the beginning, the system is able to jump to states that do (not too excessively) increase the energy (line 7), allowing to escape from local minima. With T getting lower the system will behave more deterministically and fewer states that increase energy are accepted (lines 5 and 7). The procedure can start off with an arbitrary state. For the image quantisation case, the values of parameters such as the initial temperature T_0 and the cooling factor C are based on preliminary computational experience. Typical values for these parameters are: regularity weighting $\beta \in [1, 100]$, temperature $T_0 \in (0, 16]$, and the cooling factor $C \in [0.95, 1)$. The number of iterations required for obtaining reasonable results is extremely high ($100K$ and higher for a 64×64 pixel tile). The fact that per iteration, at most one single pixel changes its state (class) assignment, and that on average all pixels should be visited enough times, requires a high iteration count ($n \gg w \times h$). This causes the algorithm to be inadequate, even for small images.

4.2.2.3 Modified Metropolis Dynamics

Solving the MRF image model by simulated annealing, in order to obtain a solution for our image quantisation application, is not very useful. The reason for this is that MRF cannot be parallelised easily. This is caused by its single scalar energy $\hat{E}_g$ for the whole quantisation state, that involves a summation. Contrary to MRF, MMD strives for minimising a local energy $e_g(s)$ per pixel in parallel. When running on a parallel architecture, MMD can converge much faster because per iteration $w \cdot h$ trials are executed in parallel. Following [110], the standard deviation per class σ_c was found fairly constant over a variety of images, results in its elimination from the energy $e_g(s)$. Although the MRF is in the long run somewhat better in quality (i.e., lower energy), MMD offers a better "quantisation quality/compute time" ratio [110]. Figure 4.6 illustrates the convergence power of MMD compared to MRF. The local energy for the MMD approach is given by:

$$e_g(s) = fid_g(s) + \beta \cdot reg_g(s), \tag{4.11}$$

where $fid_g(s)$ is defined by

$$fid_g(s) = (\gamma_s - \mu_{g_s})^2 \tag{4.12}$$

Note that in the context of MMD we use a more simplified version for fidelity than given by (4.3). Minimising the energy $e_g(s)$ for all s will raise the quality of the quantisation. The fidelity term depends on the class means μ_{g_s}, which

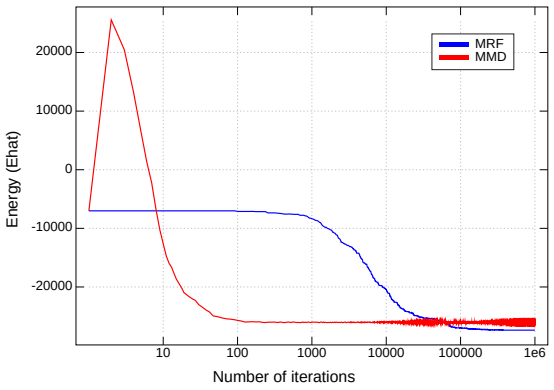


Figure 4.6: The energy decrease for MRF and MMD algorithms for up to $n = 1,000,000$ iterations

are constant, initialised by a previously executed module in the pipeline, see Figure 4.3. As in Section 4.2.2.1 the regularity term prefers neighbours having the same labels (4.5), and β is a positive model parameter controlling the homogeneity of the regions of the image. Some simplifications with respect to Algorithm 4.1 have been carried out. The statements in lines 5 and 7 can be combined into a single test. Furthermore, to avoid an elaborate computation of a logarithmic function, the comparison of ΔE with the acceptance threshold A_{th} (4.10) (in line 7 of Algorithm 4.1) is simplified to

$$\Delta e(s) = e_{\hat{g}}(s) - e_g(s) \leq T \cdot -\ln \alpha,$$

where α is a prespecified constant based on values found in literature [110][69]. See Algorithm 4.2 for the resulting algorithm. It should be noticed that Algorithm 4.1 as well as Algorithm 4.2 are taken from literature, and that they only serve as just one of the sources used in the familiarisation phase. In Section 4.3 an abstract model is made that gives the necessary freedom for the mapping to a massively parallel implementation.

The estimation of parameters such as α, β and initial temperature T_0 are crucial for obtaining a good qualitative result at a fairly low number of iterations. However, literature indicates that parameter estimation is not yet solved in a satisfactory way [70]. The values of these parameters in our case are indicated by literature [69][110] and further fine-tuned by preliminary computational experiments. Typical values for these parameters are in the same range as in Section 4.2.2.2 except for the number of iterations. The typical dynamic behaviour of MMD versus MRF is illustrated by Figure 4.6; in contrast to MRF, MMD settles around 100 iterations, independent of image size.

The complexity of the sequential implementation of MRF is $O(n \cdot w \cdot h)$. Here

Algorithm 4.2 Modified Metropolis Dynamics

```
1: g ← initialisation state
2: for T ← T_0, T_0 · C, ..., T_0 · C^{n−1} do
3:    ĝ ← randomly chosen quantisation state g of all pixels
4:    for all s ∈ S do {in parallel}
5:       Δe(s) ← e_ĝ(s) − e_g(s)
6:       if Δe(s) ≤ T · − ln α then {Acceptance check}
7:          g_s ← ĝ_s
8:       end if
9:    end for
10: end for
```

w and h stand for the width and height of an image, respectively and n for the number of iterations. The complexity of the parallel implementation of MMD is

$$O(\frac{n \cdot w \cdot h}{\#PEs}), \tag{4.13}$$

where $\#PEs$ stands for the number of Processing Elements.

4.2.3 Tiling

In this thesis feasibility estimations are made for the different cases described in Chapter 4, 5, and 6 (in several occasions per case). In particular the Chapters 4 and 5 these estimations involve *tiling*, a partioning process that arises from scarce memory resources. Tiling is required when the data volume of the problem does not fit the memory size of Linedancers in the system. For example, for the colour processing pipeline described in Chapter 5, forces repetitive processing of smaller parts of the bitmap. For some operations, in particular neighbourhood operations, require that tiles overlap. When the problem allows it, an optimal tile dimension may be chosen. For example, for a single Linedancer with 4,096 PEs this is 64×64 since that maximises the effective payload (minimises the number of reloads of same pixels), see Figure 4.7. For relatively large w and h the number of tiles n_t may be approximated by

$$n_t(o) = \frac{w \cdot h}{\#PEs - \eta(o)}, \tag{4.14}$$

where w and h represent the (2D) dimensions of the problem's data volume, #PEs the number of PEs (assuming one data unit per PE), and $\eta(o)$ the loss because of overlap o, both expressed in data units. For a single Linedancer this loss $\eta(o)$ is defined by the difference of all $4,096 = 64^2$ data units and the effective payload data (the inner $(64 - 2 \cdot o)^2$ pixels). This expression reduces to $\eta(o) = 64^2 - (64 - 2 \cdot o)^2 = 256 \cdot o - 4 \cdot o^2$. The overhead for 1 and 2 units overlap is 6%

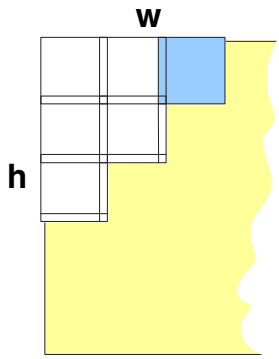


Figure 4.7: Overlapping square tiles

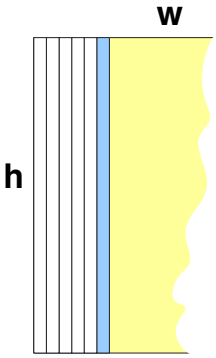


Figure 4.8: Non-overlapping strip tiles

and 12% respectively and is negligible for the purpose of a feasibility study, and hence $\eta(o)$ is removed from these estimations.

Also non-square tiles are supported, see for example Figure 4.8, where tiles do not overlap.

4.2.4 Feasibility

The purpose of this section is to obtain confidence about the feasibility of solving the problem by means of a system that is based on the intended hardware architecture. In this way early roadblocks can be identified, evaluated and their impact assessed in an early stage. This involves the appraisal of important extra-functional properties and determining the boundary of the system.

Demarcation of the system boundary. A system is not only determined by choosing the relevant interactions we want to consider, but also choosing the system boundary. In this way we know what is part of the system and what is part of the environment of the system.

Our search for the boundary is driven by risk considerations: the module(s) with highest risk for realising a viable system are selected as member(s) of the system. For this purpose we need to evaluate all relevant modules in the context, see the yellow shaded area in Figure 4.3. Of the related modules the computation of the number of classes and the initialisation for quantisation are not computationally intensive and do not pose a risk in feasibility terms. The quantisation module, however, should have significantly more quality than the pixel level quantisers. The associated computational demands are relatively high and this leads to confine the system to the quantisation module.

Feasibility: a first estimate. The goal of regular feasibility checks is to make the development process amenable to changes in the non-functional properties, even in the very beginning of development. In this way the development can be better controlled and potential roadblocks can be avoided. In our research we have chosen to describe just a single feasibility check, and we restrict ourselves to performance.

Nowadays medium range scanners are capable of processing documents at rates of over 60 pages per minute (ppm). We assume that image quantisation should at least keep up with this rate. Departing from the order estimate (4.13) we can derive the cycle budget belonging to the required throughput (1 sec), and next verify the feasibility of this budget.

The Linedancer-P1 has a clock frequency of $f_{P1} = 300$ MHz, so for a single scan (1 sec) this represents a budget of 300M clocks. The number of clocks to process the MMD algorithm should satisfy:

$$\frac{w \cdot h}{\#PEs} \cdot n \cdot C_{iter} \leq 300M, \tag{4.15}$$

where w, h are the dimensions of an A4 image in pixels, $\#PE$ represents the number of pixels a single Linedancer can host, and n is the number of iterations. Typical values for these parameters are $w = 5K, h = 7K, \#PEs = 4,096$, and $n = 100$ (see Section 4.2.2), resulting in a budget of $C_{iter} \leq 300$ clocks. This budget is reasonable for doing a few additions, subtractions and a squaring operation, but may be too tight for communicating pixels from a neighbourhood $\mathcal{N}_s$ to pixel s, as is required in the computation of the regularity in (4.5). At this point in time we know that this is a potential problem but that scalability of the technology can provide some design space.

4.3 Incremental Prototyping

This section derives the functional architecture of the stochastic image quantisation module as well as some implementation independent preparations. These preparations involve: a quality measure (for supporting the functional decomposition and evaluating implementation alternatives), the choice of the quantisation method and the estimation of the iteration count. All these activities take some time to develop and involve an extensive exploration of design space. We will follow the evolutionary development methodology – as proposed in Chapter 3 – closely. In this section the *incremental prototyping* template (Section 3.7) will be taken as a guide.

4.3.1 The algorithm

After the familiarisation phase (Figure 3.7), we turn to the stepwise creation of a complete functional model based on the mathematical model of the system as

given by the equations (4.2), (4.4), – (4.7), (4.11), (4.12). The MMD algorithm in Algorithm 4.2 is changed in an abstract and executable model. This model can be immediately transcribed in a functional language by defining the corresponding functions (where `s=(i,j)`):

```
mu c       = mean [ gamma s | s<-S; mem c (g s)]                (4.2)
fid g s    = (gamma s - mu (g s))^2                             (4.12)
Fid g S    = map (fid g) S                                      (4.4)
N (i,j)    = [ (k,l) | (k,l) <- S
                     ; sqrt((k-i)^2 + (l-j)^2) <= R
                     ; (k,l) <> (i,j)
                     ]
reg g s    = (length [r | r <- N s; g s <> g r]) -
             (length [r | r <- N s; g s == g r])                 (4.5)
Reg g S    = map (reg g) S                                      (4.6)
e g s      = (fid g s) + beta * (reg g s)                       (4.11)
E g S      = map (e g) S                                        (4.7)
```

Note that, for instance the grey-value γ_s is transcribed as `gamma s`, where `gamma` is a function, and `s` its argument. Thus, `gamma s` denotes the grey-value of pixel `s` and is generated by the scanning process.

Some further explanation of the notation: `[ e | ... ]` is notation for lists, close to mathematical notation for sets; `mem c x` is a standard function that checks whether `x` is a member of the list `c`; the function `length lst` calculates the length of the list `lst`. The environment `N(i,j)` of pixel `s=(i,j)` is parameterised by radius `R`. The function `map f S` applies the function `f` to all members of the list `S`. Note that this might be done for each pixel in parallel, provided that `mu (g s)` is available per pixel `s`.

The functional specification of the simulated annealing procedure is given by:

```
phi g T = \s -> if ((e gHat s - e g s) <= - T * log(alpha))
                then gHat s
                else g s
-- where gHat is a random g, chosen for every T
gn = fold phi g0 [T1,T2,... Tn]
-- where g0 is a suitable initialisation,
```

where the notation `\s -> f(s)` describes a function `f` without specifying its name (lambda expression), see [8]. For example the function $f(x) = x^2 + 1$ can be written as $f = \lambda x \cdot x^2 + 1$ or in functional code: `\x -> x^2 + 1`. The notation `fold` is a higher order function that realises an accumulation behaviour with another function. The expression `fold f value list` will successively execute the binary function `f` taking data elements from the `list` and starting or ending with `value`. It is typically used in recursion or iteration. The lines preceded by `--` are comment lines.

We now illustrate the first two iterations of the simulated annealing procedure. In the expression `fold phi g0 [T1,T2,... Tn]` the function `phi` changes the state assignment function `g0` into a new function `g1=phi(g0)` thereby con-

suming the first item (`T1`) of the list as a parameter. The next iteration generates `g2=phi(g1)=phi(phi(g0))` and consumes `T2`, and so on. After all iterations the resulting `gn` function is constructed, and applying this accumulated function to all pixels, by `map gn S`, will yield the end result.

We remark that this formulation of the model is just a first specification, but already at this stage it is executable. Thus, instant feedback is facilitated and consequences of this specification can be explored.

4.3.2 Quality function

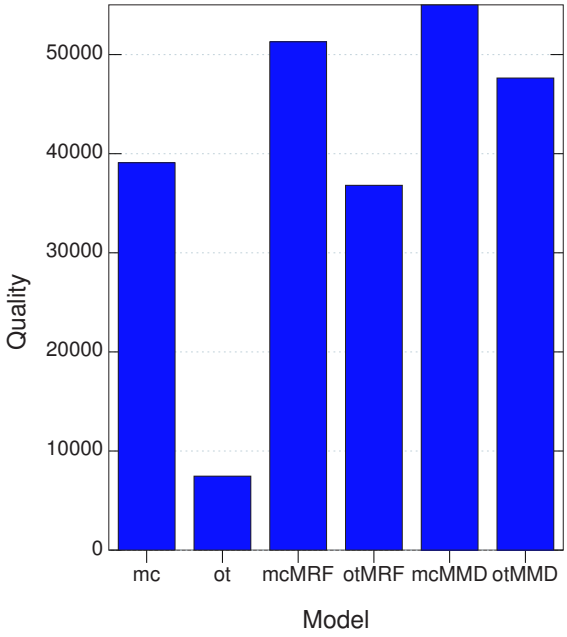
Although major decisions have been made, still some minor quality and productivity adaptations have to be carried through in order to satisfy feasibility, costs etc. Performance can be specified and measured in an unambiguous manner, but how about quality? Quality is a complex quantity to measure because it has objective (physical) aspects such as well as subjective aspects. In this study we restrict ourselves to objective quality measures and we chose the quality function $\hat{Q}_g(\mathcal{S})$ (4.9) as a guide in making the correct design choices. Note that the close relation with perceptual quality should be maintained and can lead to an adaptation of the quality function. By definition, higher values of $\hat{Q}_g$ correspond to lower values of $\hat{E}_g$ and represent "better" quantised images. Though not in the scope of this study, the quality function as well as the energy function can be extended to cover other quality aspects as macro-uniformity, contour-quality etc. The quality function is transcribed in functional code:

```
Qhat g S   = - sum E g S                                    (4.9)
```

4.3.3 Quantisation methods

Different quantisation methods give different results. To find the best combination of preprocessing (median cut, octree) and postprocessing (MRF, MMD), we tested different quantisation combinations with varying initialisations of class means μ_c. Several experiments were conducted on the image depicted in Figure 4.2(a). The models are executed for $L = 16$ quantisation classes by the image processing package *Paint Shop Pro*, which allows a minimal colour reduction of 4 bits ($L = 2^4 = 16$). The median cut and octree quantisation results are used as initialisations for the postprocessing quantisation module – implementing MRF as well as MMD processes, see Figure 4.9. The left two columns, median cut and octree, represent the results without the quality enhancing postprocessing. As expected MRF improves on quality, see next two columns. The right most measurements are conducted with MMD as postprocessing. Both MRF and MMD models are run for $100,000$ iterations.

As a result from these experiments the following conclusions may be drawn:



algorithm	description
mc	median cut
ot	octree
mcMRF	median cut, postprocessing by MRF
otMRF	octree, postprocessing by MRF
mcMMD	median cut, postprocessing by MMD
otMMD	octree, postprocessing by MMD

Figure 4.9: Comparison of various image quantisation algorithms for $L = 16$ classes, and $n = 100K$ iterations

- median cut is a better fit to this particular image (Figure 4.2) than octree for all models,
- $100K$ iterations is probably too small for MRF to outperform MMD (analogous to to Figure 4.6), and too small for octree & MRF to improve on solely median cut,
- both the MRF and MMD post-processing compensate the difference between median cut and octree,
- the median cut as preprocessing and MMD as postprocessing quantisations gives the best result for $100K$ iterations. However, it takes ± 100 ($\ll 100K$) iterations for MMD to settle, see Figure 4.6.

4.3.4 Iteration count

As can be seen in Figure 4.6 the iteration count plays an important role in raising the quality of quantisation. Typically the MMD reaches its minimum at ± 100 iterations or updates, independent on the size of the images. In quality terms MRF takes a long time to catch up but eventually outperforms MMD. For MRF the break even point is dependent on image size, typically $100K$ iterations for a 128×128 pixel image. Because of this dependency on image size, and the absence of good parallel models, the MMD model is preferred over MRF.

The following conclusions may be drawn:

- for MRF the settling iteration count for quality is dependent on image size,
- MMD converges very fast, typically 100 iterations and is in principle independent of image size (with sufficient number of PEs), and
- MRF has a slow convergence rate compared to MMD, typically $100K/(128 \times 128)[iterations/pixels^2]$.

4.4 Transformational Development

This phase is concerned with the mapping or implementation on the Linedancer. During implementation several concerns – some more general, others more specific – have to be considered. A number of them: timing, storage allocation, tiling, bit-width of variables (accuracy), and random number generation, are described below in detail. They all can potentially compromise the quality because they can trade quality for performance. The involved exploration of the design space is part of IRIS. The majority of the following sections utilise the derived quantitative quality measure $\hat{Q}_g$ for comparison. In this section the transformational development template, as given by Section 3.8, will be followed.

4.4.1 Global system considerations

A crucial property of IRIS is the continuous monitoring of functional and extra-functional properties with the goal to detect problems early and to guide the development in an appropriate direction. In this respect we already had a first impression of the feasibility at the end of the familiarisation phase (Section 4.2.4). In this section a more detailed but still global analysis is made on the timing and storage design space. Finally, relevant general system architecture issues are investigated because they can also influence the implementation process.

Global timing analysis. A global analysis at this point is merely a quick estimate of how fast a MMD algorithm can be computed. The performance target is ± 1 sec and we are interested in how many Linedancers would be needed to achieve this.

The time to process the MMD algorithm can – similar to Section 4.15 but with more detail – be described by:

$$T = \frac{w \cdot h}{\#PEs} \cdot n \cdot T_{iter},$$

where w, h are the dimensions of an A4 image in pixels, $\#PEs$ represents the number of pixels a single Linedancer can host, n is the number of iterations, and finally T_{iter}, which is the time for a Linedancer to process the algorithm for a single iteration. This time T_{iter} includes two energy computations (one for the current g and one for the proposed state $\hat{g}$, see Algorithm 4.2), a comparison and loop control. A single energy computation is estimated by summing the fidelity part (200 clock cycles for square operation on a bit oriented hardware architecture), the regularity part (some 1,000 cycles for mainly communication), and – for the remaining comparison and loop overhead – 100 cycles. We assume a maximum of $L = 16$ classes, so 4 bits of class data have to be communicated during the computation of the regularity. This sums up to 1300 cycles, and because the energy computation is performed twice (see Algorithm 4.2), this results in $T_{iter} = 2600$ cycles for a single iteration. For the whole image it takes $T = \frac{5K \cdot 7K}{4096} \cdot 100 \cdot 2K6 = 2.22G$ clock cycles, where K and G are short for 10^3 and 10^9 respectively. Finally the time needed for a single Linedancer-P1 to compute the MMD algorithm is estimated $T = \frac{\#cycles}{f_{LD}} = \frac{2.22Gcycles}{300MHz} = 7.4$ sec, where f_{LD} is the clock frequency of the Linedancer. Given the target of 1 sec this poses the challenge of bridging the gap. In Section 4.5 we will present the final outcome of the performance of the system.

Global storage allocation. The limited memory available per PE poses restrictions on the mapping of functionality and its associated data-structures. Therefore, first a qualitative analysis of the storage allocation of the MMD algorithm is conducted and its result is depicted in Table 4.1, see also [120]. The mapping related and relevant subsystems of the Linedancer are listed below (see also

Section 2.3.4). The columns 4 and 5 refer to memories located in the Linedancer. In case of the Secondary Data Store (SDS) the memory can be extended with off-chip devices. The last two columns are processing and communication subsystems. The relevant Linedancer subsystems are (see also Section 2.3.4):

- SDS, a relatively large memory, directly located in the address space of the SPARC.
- EXTended memory (EXT) and Content Addressable Memory (CAM), are two relatively small memories, of size 128 bit and 64 bit respectively. Both memories are directly located in the address space of the PE.
- Associative processor, that facilitates a parallel search and replace mechanism based on association. The mechanism ties the SPARC and CAM in a master-slave role.
- DMA, an autonomously operating channel that transfers data from and to the ASP concurrently with array computation. It connects the SDS (in the address space of the SPARC) with the PDS, which is located in the address space of all the PEs and accessible by the DMA controller, see Section 2.3.4.

The various variables in the algorithm are listed in the left-most column. Because several variables, which are specified at the C-source level imply lower level variables, an extra substructure layer is added (column 2). For example, the variable *seed* is a hidden variable, used by the pseudo-random number generator to produce a new state $\hat{g}$. A third column indicates the sort of dependency, that is useful in the final mapping to a memory subsystem of the Linedancer.

Scalability. Before conducting the mapping we should give some attention to desirable design properties such as scalability. Scalability is a design property that improves a system's key figure (as for example performance) linearly with respect to a particular design variable (for example the number of PEs). The *scalability* in the number of PEs is a property of the Linedancer's architecture. This allows for example for easy extendability of printing speed, or resolution or medium sizes or colour depth: once a suitable low-end solution exists, it can be scaled up along these dimensions without rewriting/recompiling the code.

For most fine grain SIMD systems the size of the local memory is limited. In order to be really scalable in the number of labels one must be able to retrieve the class means μ_{g_s} in an efficient way (without storing the L values of the μ_c per PE). The associative functionality of the Linedancer is suitable in providing lookup functionality for all PEs, thereby reducing claims on the limited local storage capacity. An analysis (see Section 4.4.2.2) shows that a LUT of 8 bit wide entries for the class means suffices. The required number of quantisation classes L

[2] Indication of the *sort* of dependency: the pixel (s), the number of classes (L) or not dependent of s or L.

MMD Algorithm: variable	variable substructure	dependency[2] $[s, L, none]$	Linedancer subsystems: Sparc SDS	PE CAM or EXT	Associative CAM-SDS	DMA SDS-PDS
$g, \hat{g}$	g	s		$\surd$		
	$\hat{g}$	s		$\surd$		
	$seed$	s		$\surd$		
T	T		$\surd$			
C	C		$\surd$			
n	n		$\surd$			
$\Delta e, e_g, e_{\hat{g}}$	Δe	s		$\surd$		
	e_g	s		$\surd$		
	$e_{\hat{g}}$	s		$\surd$		
	fid_g	s		$\surd$		
	reg_g	s		$\surd$		
μ	μ	L	$\surd$	$\surd$	$\surd$	
γ	γ	s		$\surd$		
β	β		$\surd$			
α	α		$\surd$			
complete tile (γ, g)			$\surd$			$\surd$

Table 4.1: Mapping of the various variables in the MMD algorithm on the memory and processing subsystems of the Linedancer

determines the depth of the LUT. The associative search and replace functionality of the Linedancer takes approximately 3 cycles per LUT-entry. For reasonably small LUTs, e.g., 16 entries, the processing overhead takes 16×3 cycles, which is small compared to the total processing time of a tile[3].

4.4.2 Trade-off subphase

The purpose of the Trade-off subphase is to absorb all concessions to the functional behaviour because of limitations of the hardware (as often is the case in embedded system design). In the following subsections, three examples are given: tiling (Section 4.4.2.1), precision of computation (Section 4.4.2.2), and random number generator (Section 4.4.2.3). During the illustration of this and other subphases we restrict ourselves – for this case – to the image given in Figure 4.2.

4.4.2.1 Tiling

Choosing a pixel-per-PE scheme means that a single Linedancer with 4,096 PEs can host *tiles* of upto 64×64 pixels. To process larger images we use tiling, that is, we divide the image in small chunks that fit in the Linedancer's ASP. Because of the neighbourhood operation (with neighbourhood $\mathcal{N}_s$), the tiles must overlap with half of the neighbourhood diameter. For similar reasons the border of the image is extended with the same number of pixels as this overlap.

For a single iteration this works fine. But when running each tile for all n iterations before proceeding to the next one, quality is compromised. This is mainly caused by pixels at a tile's boundary; these pixels are not influenced by neighbour pixels in adjacent tiles. In order to counter this loss of quality we have to provide for some form of inter-tile communication. For this purpose a multi-pass scheme is used, that is, each tile is executed r_{pt} (run per pass per tile) times, one after each other. During the r_{pt} iterations the data is kept in the ASP. When all iterations have completed the result of the tile is sent back from the ASP to the global memory. When all tiles have been processed a next pass over the tiles takes place, till all passes have been finished, see Figure 4.10. The total number of passes $\#p$ is bounded by iteration count n: $\#p = n/r_{pt}$. Each tile is executed r_{pt} times and, assuming that a single iteration takes T_{iter} number of cycles, r_{pt} iterations will last $T_{tile} = T_{iter} \cdot r_{pt}$ cycles.

Fetching tiles multiple times, in an overlapped fetch manner, effectively leads to inter-tile communication. This rather indirect way of inter-tile communication is shown in Figure 4.11. When for example tile 1 has finished its first pass after r_{pt} iterations, its intermediate state information (g_s) is written back to the Linedancer's global memory space. Since tile 2 overlaps with tile 1, a part of the just computed state is reused, and as an effect "exchanges" data across tiles.

[3] Search and replace of 16 entries will take approximately $48/8754 \approx 0.5\%$ of the total processing time, see Table 4.3.

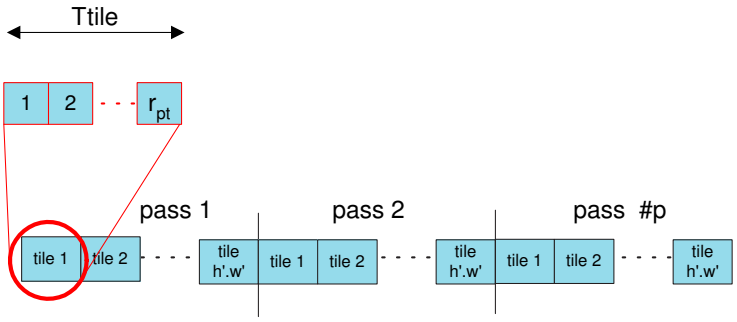


Figure 4.10: The multi-pass processing of tiles. Each tile is processed a number of runs per pass per tile (r_{pt}).

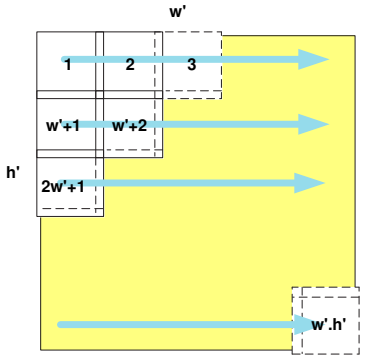


Figure 4.11: The tesselation of the bitmap in slightly overlapping tiles. Progression order in fetching tiles is free, here we show the scanline order.

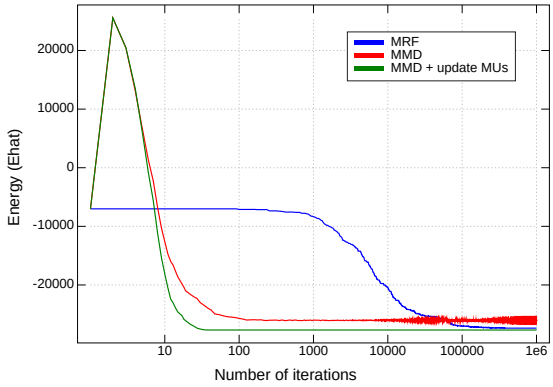


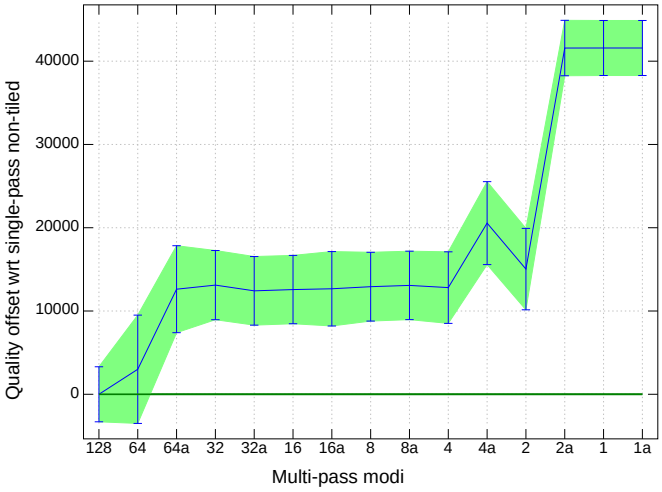
Figure 4.12: The energy decrease for MRF and MMD as in Figure 4.6 and an improved MMD algorithm for up to $n = 1,000,000$ iterations

Figure 4.13 contains the result of different multi-pass modi. Static processing parameters are tile-sizes 68×68 (tiles overlap 2 pixels wide on each side)[4], integral iteration count $n = 128$, and number of trials is 50. The number in the modi indicate the r_{pt}. Higher values of the result mean higher quality. The first modus, 128, is a tiled and single pass modus ($\#p = 1$) with a run of $r_{pt} = n(= 128)$, and is the norm for comparing all other modi. All other modi have also an "a" version, meaning that an adaptation of class means is computed after each completed pass. The adaptation of the class means is based on the current quantisation state g and renews the means $\mu_0 \cdots \mu_{L-1}$ of the L classes.

As can be observed the increase of the number of passes and the adaptation of the class means results in a quality improvement with respect to the non-tiled version but only in certain modi $(64a, 4a, 2a)$. No research has been conducted to verify if this behaviour can be generalised over multiple images, since it is not the focus of this research. The improvement in energy for the modus 2a is also shown in Figure 4.12 by the green curve.

Tiles need to be swapped out and in again to let the overlapped fetch (corresponding to the chosen neigbourhood size, Figure 4.5) effectively pass inter-tile information. Between the end of the swap out and the begin of the swap in of tiles, the class means μ_c are updated based on the actual distribution of classes over pixels. This can be done on the SPARC processor of the Linedancer. Since processing and communication of different tiles may be interleaved, this update can be completely hidden in the processing of the next tile, see Figure 4.20. To illustrate the improvement in quality by the adaptation of these class means, an example of a 4-pass quantisation is included, see Figure 4.14. The image size

[4] For the simulation of the multi-tile processing of this 128×128 image, a tile size of 68×68 is chosen (2 pixel overlap on both sides per dimension).



modus	description
128	tiled, single pass, 128 iterations
64	tiled, 2 passes, 64 iterations each
64a	tiled, 2 passes, 64 iterations each, adaptation of class means μ_c
...	...
1	tiled, 128 passes, 1 iteration each
1a	tiled, 128 passes, 1 iteration each, adaptation of class means μ_c

Figure 4.13: Comparison of different multi-pass modi for an image with size 128×128 and $L = 4$ classes

is 128×128 (4 tiles of 68×68 each) and the tiles are iterated 32 times per pass. A false coloured representation of a grey-value quantisation is given in Figure 4.14(a). The 4 classes 0, 1, 2, and 3 are indicated by colours red, blue, orange, and green respectively in Figure 4.14(b). The MMD process updates the class means after each pass (see Figure 4.14(c)), which has a direct effect on the measured quality Figure 4.14(e). Figure 4.14(d) shows the added result of MMD.

The following conclusions, based on the objective quality measure, may be drawn:

- the quality of a tiled approach is improved by trading runs per pass (r_{pt}) for number of passes ($\#p$),
- adapting the class means between the passes, conform the quantisation at that moment, in general improves quality in a global sense, and
- combining both strategies can yield a better result than a non-tiled single pass solution.

4.4.2.2 Precision of computation

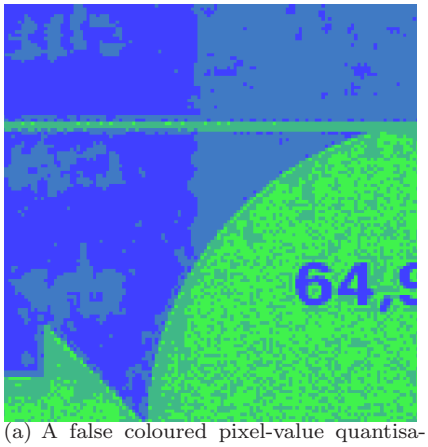
Two experiments have been conducted to illustrate accuracy analysis. The first is a sensitivity analysis of the class means μ_c, the other is a precision analysis of the variable *energy* $E_g(S)$.

Sensitivity Analysis. A sensitivity analysis is a simple way of measuring the sensitivity of small perturbations of a variable on a target function or property. Is the resulting perturbation of the target acceptably small then one may safely assume that the current accuracy is sufficient. A sensitivity analysis is often used as a fast way to verify whether a given accuracy is sufficient, or not, since it only involves a black box analysis.

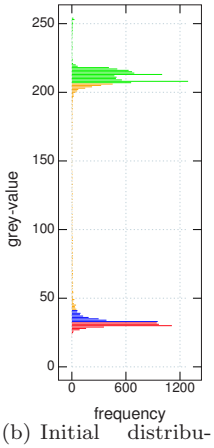
For reasoning about the necessary bit-width in a fixed point arithmetic scheme we use the following format $integer \bullet fraction$, where the numbers represent width of two bit fields: *integer* for the width of the integer part and *fraction* for the width of the fractional part (all measured in bits). The implied binary point is denoted by $\bullet$. For example $8 \bullet 0$ represents an integer byte, $8 \bullet 16$ represents a 24 bit number with 16 fractional bits. Also a negative number of fractional bits is allowed, for example $17 \bullet -3$ represents a 17 bit integer of which the 3 least significant bit(s) (lsb) are all 0. It can be concluded that only 14 bits are needed for storing this variable[5].

For the parameter μ it is expected that a precision of $8 \bullet 0$ is sufficient. In order to verify this, a sensitivity analysis is set up for all combinations of $\mu_g \pm 0.5$. For $L = 4$ classes in total 15 possible combinations with a $+0.5$ deviation and the same number of combinations with a -0.5 deviation are investigated. We

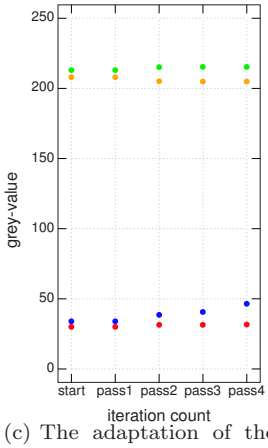
[5] This example is actually obtained from the accuracy analysis for energy $e_g(s)$, see Figure 4.16.



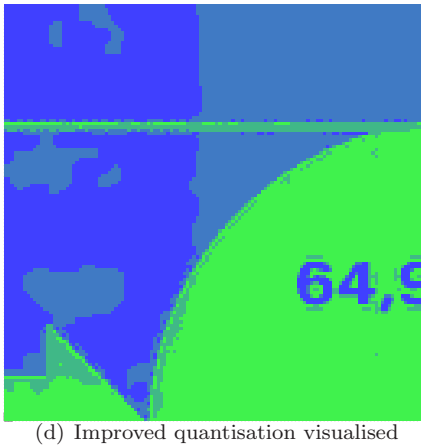
(a) A false coloured pixel-value quantisation



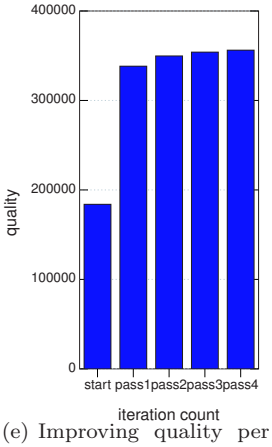
(b) Initial distribution of grey-values over classes



(c) The adaptation of the class means in a tiled multi-pass setting

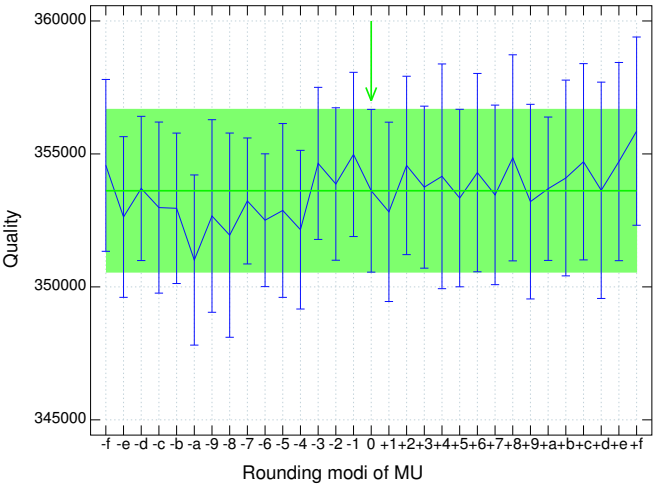


(d) Improved quantisation visualised



(e) Improving quality per pass

Figure 4.14: Intermediate stages of the quantisation process



rounding modus	description
-f	$\mu = (\mu_0 - 0.5, \mu_1 - 0.5, \mu_2 - 0.5, \mu_3 - 0.5)$
-e	$\mu = (\mu_0 - 0.5, \mu_1 - 0.5, \mu_2 - 0.5, \mu_3)$
...	...
0	$\mu = (\mu_0, \mu_1, \mu_2, \mu_3)$
1	$\mu = (\mu_0, \mu_1, \mu_2, \mu_3 + 0.5)$
...	...
+e	$\mu = (\mu_0 + 0.5, \mu_1 + 0.5, \mu_2 + 0.5, \mu_3)$
+f	$\mu = (\mu_0 + 0.5, \mu_1 + 0.5, \mu_2 + 0.5, \mu_3 + 0.5)$

Figure 4.15: The sensitivity of accuracy of μ to fractional deviations

restrict ourselves to deviations in the same direction (no combination of $+0.5$ and -0.5). Figure 4.15 shows the result of this experiment. Here all 31 possible combinations, including the reference (modus 0), are tried and their effect on quality is measured. The number of trials is 50. The combinations are identified by a single hexadecimal number, which enumerates for all combinations. The green line represents the reference representation (modus 0, green arrow). As can be seen from Figure 4.15 all the means of all modi stay within the standard deviation band of this reference. The combinations have almost no effect on the quality, which therefore renders a more detailed accuracy simulation as superfluous.

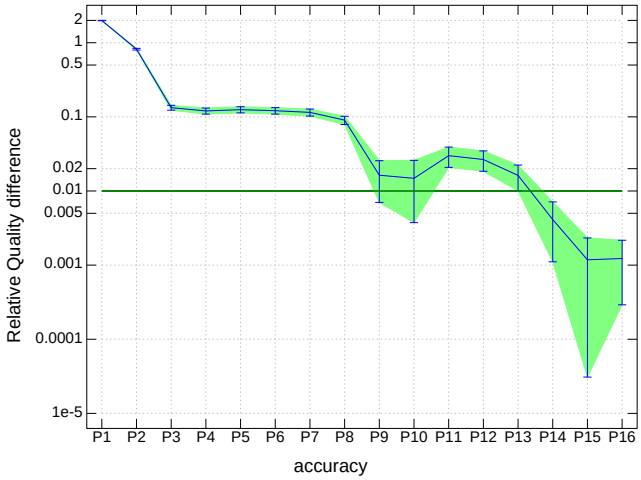
Accuracy analysis. Another type of analysis is involved to verify whether the width of bit-fields is sufficient, given a certain bound on the connected quality reduction. For the energy computation (4.11), the fidelity term $fid_g(s)$ takes the largest bit budget because of the squaring operation following a subtraction of two 8-bit values. Based on this widest field, needed for the comparison of $\Delta e(s) \leq T \cdot -\ln \alpha$, is dimensioned to a 20-bit number representation. This is sufficient for storing all intermediate results including energy, fidelity and regularity terms. The current implementation of the system is based on this 20-bit number representation. To illustrate the process of this kind of accuracy analysis we investigate a further reduction of bit-width for variable $e_g(s)$. Since we use an integer value for the weighting factor β, the energy may be safely represented by $17 \bullet 0$. The goal of this analysis is to derive the minimal width of this field (and if possible to even allow for less bits than 17). We estimated the relative quality difference $\delta\hat{Q}$ of using a representation with fewer bits by measuring the quality loss with respect to the exact 17 bit representation of *energy*. The relative quality difference is then defined by $\delta\hat{Q} = |\hat{Q}_{17}(g) - \hat{Q}_{trunc}(g)|/\hat{Q}_{17}(g)$, where $\hat{Q}_{17}(g)$ represents the full width quality, and $\hat{Q}_{trunc}(g)$ the quality when all $e_g(s)$ are truncated.

The truncation of fixed point numbers (with an integer and a fractional part) may be described by

$$\frac{\lfloor 2^{ACCUR} * x \rfloor}{2^{ACCUR}},$$

where x is a fixed point number and the accuracy parameter $ACCUR$ may take positive as well as negative integer values. Figure 4.16 contains the result for the various accuracy settings $P_{16} \cdots P_1$, where the subscript $i \in [1 \cdots 16]$ of P_i corresponds with the field width in bits. Starting from the omission of the lsb of $e_g(s)$, that is, P_{16}, the quality loss is far below the threshold of 1%. The number of trials for each selected width is 50. The 1% relative quality loss is indicated in Figure 4.16 by the green horizontal line. As can be seen, the successive removal of least significant bits ($P16, P15, P14$) do indeed increase this loss and passes the 1% threshold when more than 3 bits are truncated. So allowing for a 1% relative deviation on the quality, the variable *energy* can be coded in merely 14 bits.

A relative deviation of a design quantity is a simple way of expressing a quality



accuracy	description
P16	field is truncated to the 16 most significant bit(s) (msb), the lsb is lost
P15	field is truncated to the 15 msb, the 2 lsb are lost
...	...
P2	field is truncated to the 2 msb, the 15 lsb are lost
P1	field is truncated to the msb, the 16 lsb are lost

Figure 4.16: Quality loss as function of the accuracy of a variable precision representation of the width of the variable *energy*

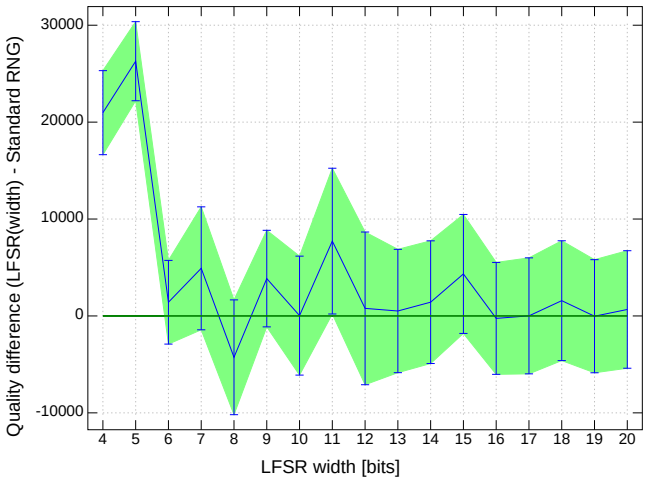


Figure 4.17: Effect of LFSR bit-width on quantisation quality, image size 128 × 128. Number of trials is 50.

deficiency in terms of resources on which one can make concessions.

4.4.2.3 Random Number generator

As a last topic for accuracy analysis we have selected a part of the algorithm of which, not the value gives precision problems, but rather the sequence of consecutive values. For every iteration a new state ($\hat{g}$) has to be generated in a random fashion. Pseudo random generators based on Linear Feedback Shift Register (LFSR) have low memory footprint and only need simple bit operations: *Exclusive OR (XOR)* and *Shift* [55]. A 10 bit LFSR with only two tap points can be described by: `\lfsr -> (tail lfsr) ++ [(lfsr!0)xor(lfsr!3)]`, a function representation in lambda notation, where the (list) argument `lfsr` is 'shifted' over 1 position to the left and the last element is exchanged by the boolean `xor` (not equal) of the 0^{th} (first) and 3^{th} element of the original list. This LFSR is capable of generating a pseudo random number sequence with cycle length $2^{10} - 1 = 1023$, which is sufficient as shown by Figure 4.17. Here the quality loss of a LFSR approximation with respect to a standard pseudo random number generator is measured. For our purpose a random number generator is used that is based on *GB_Flip*, an algorithm with a period length of at least $2^{55} - 1$, see [76].

Remarkably, the lower LFSR bit-widths have "good" quality. The explanation is the low cycle length $2^{W_{LFSR}} - 1$ for small bit-width $W_{LFSR} = 4$ or 5 compared to the size of the neighbourhood $\mathcal{N}_s$ (Figure 4.5). This causes a periodic effect in the random number sequence. Since the generation of new states, line 3 in

Figure 4.18: Regular patterns in quantised result because of small width (7 bit) of LFSR, for a 128×128 sized image

Algorithm 4.2, involves the taking of a few bits from this random sequence (only 2 bits in case $L = 4$), this periodic effect is amplified. The effect is a disturbance in the delicate fidelity-regularity balance, becoming more and more in favour of regularity with each following iteration. This occurs preferably in those areas, which contain pixels of two nearby classes (with almost same μ_g). In few iterations, for example $n = 7$ for $W_{LFSR} = 4$, the occasional fidelity mismatch is overcompensated by a large gain in regularity, forcing the emergence of large homogeneous areas. Even for wider LFSR bit-width the effect can be noticed, see Figure 4.18 for $W_{LFSR} = 7$.

The following conclusions may be drawn:

- for width ≤ 9 [bits] the quantised images exhibit annoying regular structures,
- for larger widths the change in quality is negligible,
- and for these widths the difference with respect to the higher quality random generator (GB_Flip) [76] is almost nil.

4.4.3 Reorganisation subphase

During the reorganisation subphase the model is expanded in a top-down manner, gradually changing the hardware independent description into a form dictated by the hardware architecture. To demonstrate this added value we mention a few issues in this phase.

Precomputation of Constants. Some constants can be computed during the initialisation and for example prestored in a particular memory field administered per PE, or in a Look Up Table (LUT) on the Lindancer's control processor. For example, in the definition of fidelity $fid_g(s)$ (4.12), the mean μ_c of a class c is a constant, and thus it is efficient to calculate it only once. We will assume that for every class c the mean μ_c will be stored in a lookup table on the Linedancer control processor (SPARC).

Transformational laws. For every pixel the *fidelity* has to be computed according to the definition

```
fid g s = (gamma s - mu c)^2   where c = g s.
```

We focus for now on the computation of the class means `mu c`, dependent on the class assignment `g s` per pixel `s` and for all classes `c`. The simplest way for a programmer of a more general parallel processor (e.g., MIMD) would be to let each PE do the lookup for its pixel. The procedure that every PE then has to execute is simple as well: just walk through the lookup table until you find your own class `(g s)=c`, and then lookup the corresponding class mean `mu c`. In terms of the architectural language this means that every PE executes a `fold` function (iteration) over the LUT. This may be expressed by saying that this `fold` function is `map`-ped to the set of all PEs. Thus, this simplistic approach would lead to a program that essentially looks like:

```
class_means = map ( x -> fold (f x) v0 lut ) class_assignments
              where f x v (y,w) = w   , if x=y
                                = v   , otherwise
```

The function `f` ensures that the initial value `v0` is updated with the correct value from `lut`. Note that the exact value of `v0` is irrelevant, since it is simply replaced by this value from `lut`. Summarising, this comes down to the parallel execution of an iteration over the entries of a LUT.

However, given the limitations in local computational capabilities and memory size of a SIMD architecture like the Linedancer, individual PEs cannot execute such an iterative process. The consequence is that the iteration has to be executed by the control processor, and the relevant data of each class have to be broadcasted to all PEs. Each PE only executes the above definition when its own class `g s` matches with the broadcasted current class index `c` of the LUT. Thus, the control processor performs a `fold` over the LUT, and `map`s the LUT data at each step to all associating PEs. In terms of the architectural language this pattern looks like (apart from some minor formal details):

```
class_means = fold (map f PEs) class_assignments lut
```

Again, the `fold`-function iterates over the lookup table `lut`, but now it is a "broadcast" function (`map f PEs`) that is iterated, that is, the function `f` (that updated a pixel with the correct value) is broadcasted to all pixels. This broadcast is done for each entry in `lut` and at each step the variable `class_assignments` is updated for the relevant part. Again, note that the exact value of `v0` is irrelevant, since the class assignments are replaced by the correct value from `lut`. Summarising, this comes down to the iteration over all entries of the LUT while performing the lookup for all PEs in parallel.

Without going into details we remark that there is a precise *law* that transforms the first specification into a second one. That is to say, this law transforms a straightforward specification that is very simple to design, into a more complex executable program. Such laws are important to guarantee correctness and therefore play an important role in the IRIS methodology. It is one of the advantages of a functional language as architectural language that such laws can be formulated precisely and proven formally.

A second application of the same law is discussed below.

Expression optimisation. In order to reduce execution time, each definition has to be checked for possibilities to optimise the computation. For example, definition (4.5) is straightforward and easy to specify, but the list of neighbours of each pixel has to be traversed *twice* in order to calculate the respective lengths. The following equivalent definition subtracts or adds 1 when the `g r` is equal or unequal (respectively) to `g s` and traverses the list of neighbours only once:

```
reg g s = sum [ if (g r == g s) (-1) (+1) |  r <- N s ]
```

According to the definition (4.11) the outcome of this expression has to be multiplied by the parameter `beta`.

One of the advantages of choosing a functional language as architectural language is that also at early stages in the design process the definitions are executable, thus quantitative experiments are facilitated. A simple experiment showed that the above definition of `reg` can be slightly optimised further (168 versus 180 cycles per pixel, given a 12 pixel neighbourhood $\mathcal{N}_s$ and $\beta \in \{1 \cdots 255\}$) by adding or subtracting this parameter `beta` directly. Thus, the definition in (4.11) can be replaced by the definition:

```
e g s = fid g s + sum [ if (g r == g s)(-beta)(+beta) | r <- N s ]
```

We remark that the equivalence of these definitions can be easily shown.

Transformational laws (2). Again, the above mentioned definition of `e` has to be broadcasted to all PEs. Note that determining the sum of a list requires an iteration over the list, that is, we have the same pattern as before: a `fold` inside a

`map`. Then clearly the same problem arises, such a specification is not executable on the Linedancer. However, we can apply the same law as before, leading to a `map` inside a `fold`, which *is* executable on the Linedancer.

4.4.4 Template subphase

The goal of the template subphase is to derive reusable components. This not only includes macros for code fragments or even complete modules but also includes support for context dependent instruction coding. As time progresses, *experience* translates into more powerful components (bottom-up). Templates are intelligent pieces of interactive functionality that serve three roles.

1. First of all, the functional behaviour of the involved Linedancer instruction(s), *Functional Emulation*, should be properly modelled.

2. Second, compliance with the *Calling Conventions* for all relevant instruction types should be enforced.

3. Finally, the syntax of the template-call should be rich enough to enable automatic generation of the target code for this call (*Facilitating Translation*).

This subphase, in fact, compresses the size of the description by rephrasing equivalent code fragments in such a way that reusable components emerge.

Both the calling convention and the allocation support use a special calling convention for variables to express the allocation of Linedancer memory to the variables. We express this in the variable name as `<name>_<start_position>_<length>` such that memory can be allocated based on these names (call by name).

For example the energy computation `e g s` is performed twice: once for the current state `g` and once for the alternate state `gHat`. Table 4.7 (on page 106) shows the allocation for the relevant variables. The two instances are given below:

```
energy_82_20 = e ( state_11_4 )
newEnergy_27_20 = e ( newState_65_4 ),
```

where `state_11_4` corresponds to `g` and `newState_65_4` corresponds to `gHat`.

Because of the above mentioned explicit allocation we can at this point also illustrate the handling of 'intelligent' compilation, as a typical activity in resource sharing, see Section 3.8.2. The variable `fid g s` (fidelity) shares the same field as `e g s` (energy). Therefore, we can allocate this variable by `fid_82_16` since $fid_g(s)$ can be represented by 16 bit (see Table 4.7).

4.4.5 Translation subphase

The goal of this subphase is to translate the model code into the native language of the target hardware (C enriched with hardware intrinsic instructions). The

translator takes a specially formatted description as input and translates this into Linedancer-C code that can be compiled using the Linedancer toolchain. The specific details of that language fall outside the scope of this paper, hence we restrict ourselves to pseudo code.

In many cases, the functional specifications can be translated straightforwardly into pseudo code. For example, the expression (`arr` stands for an array, `v0` for an initial value)

```
fold f v0 arr
```

translates into

```
a = v0;
forallSeq x in arr do
  a = f(a,x);
```

where the additional variable `a` plays the role of an accumulation variable which contains the required value after termination of the `for`-loop.

Clearly, in this case the `for`-loop goes through the list in a sequential way, as suggested by its name `forallSeq`. The parallel variant is expressed by

```
map f arr
```

and translated into

```
forallPar i in arr_indexes do
  arr[i] = f (arr[i]);
```

where `forallPar` suggests a parallel `for`.

Applying this to the definition of `e` as derived in Section 4.4.3 yields:

```
forallPar s in S do e_g[s] = fid_g[s];
forallSeq r in neighbourVectors do
  forallPar s in S do
    e_g[s] = e_g[s] + if g[s] == g[r] then (-beta) else (+beta);
```

In addition to this pseudo code, we again need the special naming convention for variables (as mentioned in the template subphase) to specify the bit-fields in Linedancer-C. Furthermore the array `neighbourVectors` – containing all indices `r` for the computation of `g[r]` – is needed. At last we mention the introduction of a temporary variable, that is administered on the SPARC, for the access of this index because the ASP cannot index the array itself.

The generated Linedancer-C code of a part of the pseudo code fragment above, is included below. The `forallSeq r` compound is mapped to the `for` statement, and the variable `r` is mapped to `n`. The evaluated index `[r]` is looked up in array `NBOURS`, stored in the mentioned temporary variable `temp`, and actually

represents the communication distance to the remote PE that hosts the state information `g[r]`. In this example bit-field (10,4) hosts state `g` for all PEs. See for further details Section A.1.

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

4.5 Results and Discussion

The resulting system is described and discussed in detail below. The results are structured in the following way:

- Timing
- Memory allocation
- Quality
- Methodology

Timing. This paragraph discusses the performance related results. Performance measurements are included and extrapolated to a full page to compare it with the given timing constraint. Suggestions for improving the performance are given.

Timing measurements. Table 4.2 summarises the timing results of three distinct implementations of image quantisation for $L = 16$ quantisation classes. Two of them implement the MMD scheme, one executed on a 2 GHz Pentium Xeon with 1 GB DRAM and one on the Linedancer system (consisting of 2 Linedancer chips). For comparison also a state of the art quantisation algorithm median cut [48] is given, which is part of the image processing package Netpbm[6]. Netpbm is an open source software package of graphics programs and a programming library. The MMD Linedancer implementation is 1.33 times faster than median cut running on the above mentioned Pentium processor (and 128× faster compared to MMD on a Pentium).

Extrapolation to full page processing. Figure 4.20 shows which of the Linedancer's processing units is involved in this process. In this figure a detailed timing chart is given for a specific case where the number of runs per tile per pass $r_{pt} = 2$ and for $L = 4$ classes. As explained before the tiles need to overlap

[6] Bryan Henderson: "About Netpbm", 2007, `http://netpbm.sourceforge.net/`.

# Pixels	Time (ms)		
	median cut on Pentium	MMD on Pentium	MMD on Linedancer
10000	31	517	5.95
40000	47	2070	23.1
160000	110	8420	80.3
640000	375	34600	280
2560000	1438	138000	1080

Table 4.2: Execution times of quantisation for $L = 16$ classes: median cut and the MMD version both on a Pentium, and MMD on a dual Linedancer system. All MMD processing is performed with $n = 100$ iterations.

in order to support the chosen neighbourhood system, see Figure 4.5. For example in our case we need a 2 pixel wide overlap on all 4 sides resulting in an effective tile size $t_{eff}^2 = 60 \times 60$, assuming that a single Linedancer processor can host 4,096 pixels. The whole bitmap of $w \times h$ pixels is covered by an integral number of $w' \times h'$ tiles, where w' is the number of tiles in horizontal direction and h' is the number of tiles in vertical direction. For an A4 image format ($W_{A4} = 210mm, H_{A4} = 297mm$) at $r = 24$ pixels/mm resolution, this means that $h' = \lceil \frac{H_{A4} \cdot r}{t_{eff}} \rceil = \lceil \frac{297 \cdot 24}{60} \rceil = 119$ and $w' = \lceil \frac{W_{A4} \cdot r}{t_{eff}} \rceil = \lceil \frac{210 \cdot 24}{60} \rceil = 84$ tiles respectively. This means that for an A4 sized image the number of tiles $h' \times w' = 119 \times 84 = 9996$. For the ASP a single iteration of a tile takes $T_{iter} = 8K8$ cycles as can be seen from Table 4.3. Since each tile is executed $r_{pt} = 2$ times this amounts to $T_{tile} = T_{iter} \cdot r_{pt} = 17K6$ cycles. Together with the load and dump time $T_{in} = T_{out} = 512$, the processing per tile totals to $T_{in} + T_{tile} + T_{out} = 18K6$ cycles. A concurrently running thread on the SPARC processes each intermediate result of a tile and maintains tallies as well as subtotals of the grey-values per class. This is estimated to 4 cycles per pixel and results in $4K \cdot 4 = 16K$ cycles, so just below the time of the tile processing by ASP and DMA controllers ($16K < 18K6$ cycles). The same thread also computes the update of the class means μ_c if needed. Since the frequencies and grey-value subtotals for all classes are already maintained per executed tile, this $T_{upd\mu}$ does not take much time (4 float divisions).

So the critical path for $r_{pt} \geq 2$ is formed by the tile processing by the ASP (T_{tile}) with both two input (T_{in}) and output (T_{out}) activities for tile data. The integral time T_{lat} (latency) is given by $T_{lat} = (T_{tile} + T_{in} + T_{out}) \cdot h' \cdot w' \cdot \#p + T_{upd\mu}$. For $r_{pt} = 2$ runs per tile uses $\#p = \frac{n}{r_{pt}} = \frac{100}{2} = 50$ passes. A single 300 MHz Linedancer-P1 will process an A4 in $T_{lat} = \frac{18K6 \cdot 9996 \cdot 50}{300MHz} \approx 31$ sec. This result is disappointing and should be improved.

Improving the performance. To give an idea how many cycles the different parts

Activity	# Cycles
Preparation	70
Processing one tile	
Calculate a new random labeling	44
For each label	**352**
$y - \mu_{g_s}$ (16×)	22
Square	164
For each neighbour	**3592**
Add or subtract β (avg) (12×)	299.3
Load y	16
For each label	**352**
$y - \mu_{g_s}$ (16×)	22
Square	164
For each neighbour	**3592**
Add or subtract β (avg) (12×)	299.3
Subtract energies, threshold values and update	70
Dump result	338
Total	**8754**

Table 4.3: Number of measured cycles used for each iteration per tile of the algorithm. Nesting indicates loops, bold numbers indicate accumulated results.

of the algorithm take, we measured the number of cycles taken for different stages of the algorithm. The results can be seen in Table 4.3. For the "For each neighbour"-parts, which take 3592 cycles each, 3200 (estimate based on a communication model, see Section A.1) are spent on communication. This is approximately 73% of a total of $\hat{T}_{iter} = 8754$ cycles for each iteration per tile. However, clever reuse in communicating the neighbourhood could reduce this overhead, provided some memory space is available for storing intermediate results. Then 55% of a reduced total of 5207 cycles is spent in communication, yielding a speedup of 1.7, see first row in Table 4.4.

A further improvement can be obtained by extending the Linedancer's synchronous inter-PE communication with a chordal ring [91], for example an extra link for each PE with distance 64, see Figure 4.19. This would yield a total speedup of 2.9 and would turn this realisation into a processing bound solution; only 21% of a reduced total of 2988 cycles is then spent in communication, see Table 4.4. The Linedancer-HD has implemented an extra chord with distance 32.

Other ways to improve the performance is to reduce the width of wide variables with marginal loss of quality (e.g., variable *energy* as shown by Figure 4.16) and/or increase the number of Linedancers. Table 4.5 reports

chords		$\hat{T}_{iter}$	T_{comm}	speedup
	1	5207	2852	1.7
2	1	4098	1743	2.1
4	1	3534	1179	2.5
8	1	3243	888	2.7
16	1	3097	742	2.8
32	1	3024	670	2.9
64	1	2988	633	2.9

Table 4.4: The speedup $\frac{T_{iter}}{\hat{T}_{iter}}$ for processing tiles when adding an extra chord to the default communication ring. The time spend in communication (T_{comm}) is included in $\hat{T}_{iter}$.

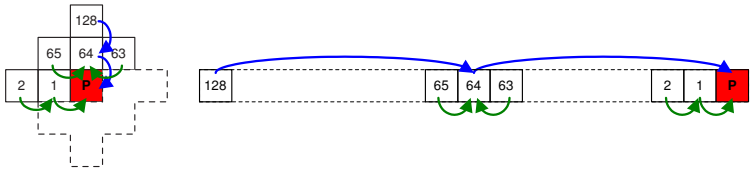


Figure 4.19: Effect of improving the connectivity of the PEs by a chordal ring with an extra chord with length 64

n	P1 time [sec]	HD time [sec]
1	18.2	10.7
2	9.1	5.3
4	4.6	2.7
8	2.3	1.3

Table 4.5: The estimated performance of stochastic image quantisation for the Linedancer P1 and HD processor with the mentioned speedups incorporated (n is the number of Linedancers)

the performance figures.

Summary. From a performance point of view this postprocessing step of the MMD algorithm, see Figure 4.3, represents a relatively large overhead. However, considering the scalable approach the Linedancer solution offers, and keeping in mind the increase in integration density at constant costs, the postprocessing by MMD is expected to obtain a reasonable quality/performance ratio in the future.

Memory Allocation. The final allocation is presented in two tables, one for fixed parameters used by the algorithm (Table 4.6) and the other for the variables used in the algorithm (Table 4.7). Table 4.6 lists all parameters used in Algorithm 4.2, complete with the domain they are defined on, and the chosen fixed point representation. Of the parameters, only grey-value γ_s is stored in the ASP (memory field specification 24(8) means bit positions 24-31). All other parameters are either broadcasted to the PEs for direct processing or kept on the SPARC (e.g., for flow control in case of iteration count n). Table 4.7 lists all the variables used in the MMD algorithm. From the variable dependencies (2^{nd} column), the range and subsequently the fixed point scheme and the memory field specification can be derived ($3^{rd}, 4^{th}$ and 5^{th} column respectively). The last column refers to the line numbers in Algorithm 4.2.

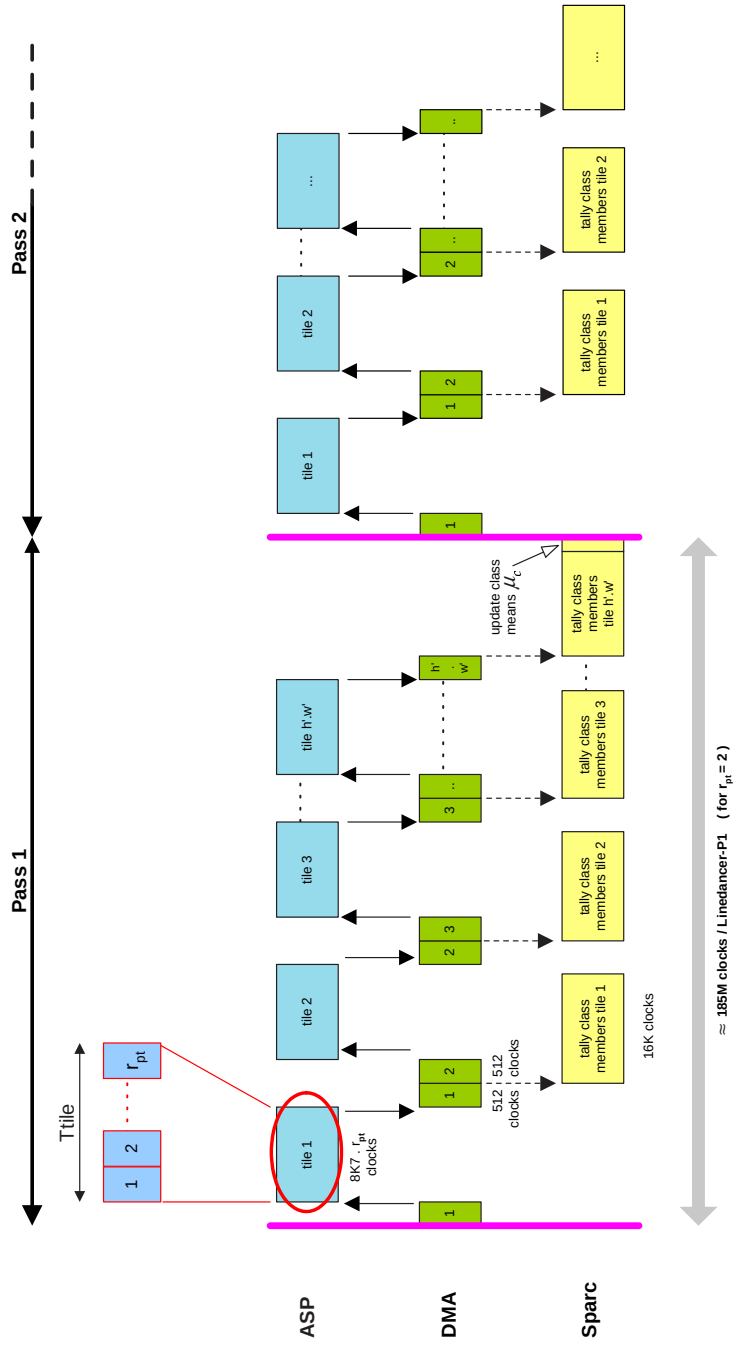


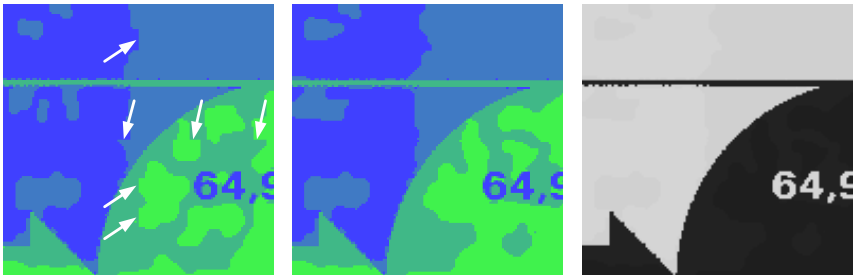
Figure 4.20: Detailed schedule of tasks over the Linedancer's processors

description	symbol	range	fixed point scheme	memory field	domain	host
grey-value	γ_s	[0..255]	8 • 0	γ 24(8)	array[0..w-1, 0..h-1]	ASP
class mean	μ_c	$[0, 256)$	8 • 0	broadcast	array[0..L-1]	SPARC
weight factor	β	$[1, 100]$	7 • 0	broadcast	fixed point int	SPARC
temperature	T	$(0, 16]$		global[7]	floating point	SPARC
acceptance threshold	α	$[0.01, 1)$		global[7]	floating point	SPARC
iteration count	n	[50..200]	8 • 0	global	fixed point int	SPARC

Table 4.6: Fixed point accuracy of parameters in the MMD algorithm

description	dependency	range	fixed point scheme	memory field	algo-rithm-line
quantisation state or class	$g_s, \hat{g}_s$	$[0..L-1]$	4 • 0	*state* 11(4), *newState* 65(4)	1,3,7
fidelity	$fid_g(s) = (\gamma_s - \mu_{g_s})^2$	$[0, 256^2)$	16 • 0	*energy* 82(20), *newEnergy* 27(20)	5
regularity	$reg_g(s) = \mid \mathcal{N}_s \mid - \; 2 \cdot \mid \{r \in \mathcal{N}_s \mid g_s = g_r\} \mid$	$[-12, +12]$	6 • 0	*accu* 15(12)	5
energy	$e_g(s) = fid_s + \beta \cdot reg_g(s)$	$[-12, 12 + 256^2]$	17 • 0	*energy* 82(20), *newEnergy* 27(20)	5
energy difference	$\Delta e(s) = e_{\hat{g}}(s) - e_g(s)$	$[-24, 24 + 2^{17}]$	18 • 0	*newEnergy* 27(20)	5
acceptance	$\Delta e(s) \leq -T \cdot ln\alpha$	$\{false, true\}$	19 • 0	*newEnergy* 27(20), *acceptBit* 10	6

Table 4.7: Fixed point accuracy of variables used in the MMD algorithm



(a) One pass, arrows indicate tiling artifacts (b) Two passes, no visible tiling artifacts (c) Resulting image

Figure 4.21: Quantisation by MMD on an image of 128×128 pixels, processed in chunks of 68×68 tiles

Quality. Some remarks can be made with respect to state of the art quantisation algorithms in terms of the improvement in quality. The ringing at the edges and the speckles have disappeared when comparing the image in Figure 4.2(b) and the image in Figure 4.21(a). But the redistribution of classes leads to larger areas with arbitrary borders. However, these larger areas can be handled more effectively than small areas by using techniques from object and image recognition [56]. Further study is needed to reduce these large areas.

Tiled processing of the image is necessary because the complete image is too large for just a few Linedancers. The fetching and processing of subsequent tiles – for single pass processing – is insufficient to compensate for tile border artifacts, see Figure 4.21(a)[8]. Therefore, each tile is processed multiple times, effectively allowing neighbouring tiles to better communicate their regularity information, see image in Figure 4.21(b). This can be done without performance degradation because on the Linedancer the dumping of the result of a previous tile and the loading of the next one can be completely hidden in the processing of the current tile. More detailed statements with respect to quality/cost are not appropriate here, because the used quality measure for quantised images is not adequate enough from an image quality perception point of view. Moreover, improving the quantisation was not the primary goal of this research.

Methodology and Architectural language issues. Regarding the used methodology the following remarks can be made.

The methodology was helpful in the exploration of image quantisation in gen-

[7] Parameters T and α are globals, but their combination in the acceptance threshold $A_{th} = -T \cdot \ln(\text{Random})$ is broadcasted to the ASP, line 8 Algorithm 4.2.

[8] The artifacts are "capped" structures, that are exactly positioned on the tile boundaries. Downward pointing arrows indicate the horizontal inter-tile boundary, the other arrows show the vertical boundary.

eral, and the modelling for Simulated Annealing, MRF and MMD in particular. After establishing the functionality, and the development of the code, the target code could be generated automatically. In particular for this case the monitoring and control of extra-functional properties was extended. For example the quantisation quality (tiling) and the resource consumption (accuracy or bit-width of various parameters and variables) are woven into the normal functional flow and can immediately signal the developer when values pass a predetermined limit. A coarse form of time modelling served our needs.

The derivation of the four layers within the transformational development process (Section 3.8), is the specific gain for methodology of this case. This case posed the correct conditions: the developers were unfamiliar with the problem domain of quantisation, with models as MRF and MMD, and were a bit experienced with Simulated Annealing and the Linedancer's architecture. In contrast to colour image processing for printing (Chapter 5) and mining of dynamic document spaces (Chapter 6), this case offered the optimal creative space to derive the mentioned layering.

Regarding the architectural language we can remark that an interpreter is a necessity for interactivity, which by itself is a feature welcomed among developers. Interactivity proved its value during prototyping of functionality and target hardware behaviour. The ease of setting up models, test cases and running them instantaneously returns control and joy to the developer. The transcription of the mathematical models as well as the hardware models to the architectural language can be done in an almost one-to-one manner. The architectural language environment is useful for generating instant graphics for the trade-off subphase.

4.6 Conclusions

Stochastic image quantisation can be modeled in a parallel way. Here a MMD model is implemented on a Linedancer massively parallel processor. The parallel version on a dual Linedancer is 128× faster than the sequential implementation of the parallel algorithm on a 2 GHz Pentium Xeon system. Moreover, the parallel version has better scalable properties and offers easier control for improvement of quality.

Careful engineering of the inter-PE communication could increase the speed by an extra factor of 1.7. When the processing array is extended with a chordal ring interconnection structure, with an extra chord connecting PEs at distance 32 or 64, then a total speedup of approximately 2.9 can be obtained.

Stochastic image quantisation proved to be a valuable testcase for extending the IRIS methodology for constructing the code that runs on the ASP (Linedancer's massively parallel core). By guiding the developer throughout the entire development cycle the complex coding, normally associated with such dedicated programmable processors, is turned into a manageable process. In particular the layering of the transformational development process is consolidated during this case.

CHAPTER 5

Case: Colour Image Processing

Today FPGAs form the dominant technology for implementing colour image processing pipelines for high volume colour printers. Although FPGA technology provides sufficient performance, it suffers from a tedious development process. In this chapter we show that massively parallel processing – for instance using the Linedancer (Section 2.3.4) – not only leads to a reduction in development time but also adds flexibility to the design.

For the image processing system we found that the Linedancer development is twice as fast as the FPGA development trajectory and that the Linedancer-HD is able to fulfill the processing requirements. The system was initially designed and implemented for an FPGA. The port of this application to a programmable processing environment is guided by the IRIS *evolutionary development methodology.*

5.1 Introduction

The market for colour printing is developing into the direction of high volume colour printing, where high quality and high speed are mandatory. In addition time-to-market is important; this can be directly translated into an increasing pressure on development cycles.

Until recently, only FPGA technology was able to fulfill the performance needs for the printer's colour image processing subsystem at a reasonable cost, but this technology has the disadvantage of a relatively long development trajectory. However, with the advance of programmable many-core devices such as

Major parts of this chapter have been published in [P1].

the Linedancer, the best of two worlds can be combined: the programmability similar to a GPP and the performance of an FPGA. Additionally, many of the image processing tasks in a printer show simple massively parallel processing suitable for SIMD processing. Moreover, this kind of processing allows for scalable design, that facilitates flexibility in an easy manner.

In this study we evaluate the IRIS methodology on an Océ specific problem case. A second motivation is to demonstrate that image processing contains easy to exploit massively parallelism.

This chapter starts off with an introduction of the used colour printing process, the colour image processing pipeline and a first global statement about the technical feasibility for a SIMD implementation (Section 5.2). The complete system is too large to present here. Therefore, we restrict ourselves to a difficult to parallelise part of the system namely *error-diffusion* (in module half-toning). The functional specification of error-diffusion is given in Section 5.3, immediately followed by its implementation on an associative processing array. Finally the results of the complete system, a comparison with an FPGA implementation, and conclusions are given.

5.2 Familiarisation

A rudimentary feasibility study of a programmable processor solution for a typical colour image processing pipeline is presented. First some relevant domain specific details are given.

5.2.1 Colour Printing Process

In general an image processing pipeline is heavily dependent on the used printing process. The particular printing process consists of 7 monochrome colour printing units, which deposit their colour image on an intermediate drum, see Figure 5.1. The intermediate drum accumulates these mono-colour images that are printed by the printing units. The image processing pipeline has to take into account the various offsets these units have with respect to each other. These mono colour images, or *separations*, need to be aligned with sufficient precision on the intermediate drum, a process called *registration*. After all 7 units have deposited their contribution to the intermediate drum the integral colour image is transferred to the receiving medium (in most cases paper). The used colours are abbreviated as follows: K (black), B(blue), R (red), G (green), C (cyan), M (magenta), and Y (yellow). Note that the printer colours R, G, B are similar but are in general not the same as the RGB colours of a colour scanner. A particular useful property during this registration process is the physical inability of the intermediate drum to stack multiple colours on the same spot. This allows for correcting slightly displaced colour separations (as exploited by the trapping module, see Section 5.2.2.4).

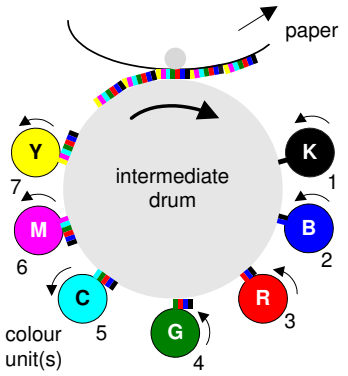


Figure 5.1: Colour printing process

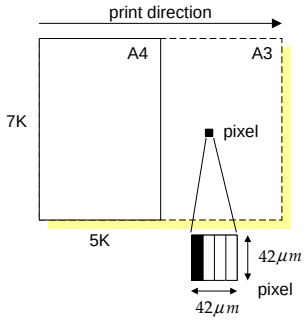


Figure 5.2: Orientation and sizes of the supported A4 and A3 formats with substructure of pixels in print direction only (A3 encloses A4)

The printing process considered in this chapter supports a maximum papersize of an A3 format, that is, it can print a landscape A3 and a portrait A4. The resolution r_h is 2400 dots per inch (dpi) in transport or print direction and $r_v =$ 600 dpi in the perpendicular direction, see Figure 5.2. This results in non-square subpixels measuring approximately $10\mu m \times 42\mu m$.

During the development of such a colour printer, major design decisions have been made conclusively, even before the development of the image processing subsystem starts. Typical starting points for such a subsystem cover *quality* issues such as resolution, process colour-scheme, medium type, and *productivity* issues as printing speed, medium size, and plexity[1]. The involved throughput can be quantified by computing the stream-productivity P, the data volume per time unit (in bytes/sec):

$$P = p \times w \times h \times r_h \times r_v \times c/t, \tag{5.1}$$

where p stands for plexity (1 for simplex, 2 for duplex), w and h for the medium width and height, r_h and r_v for resolution (in dpi), c for number of colour bytes, and t for the required time per page.

Key specifications for our specific colour printer are: speed 30 ppm or $t = 2$ sec page time, paper size A4 ($w = 210$ mm, $h = 297$ mm), resolution $r_h \times r_v =$ 2400×600 dpi$\approx 94 \times 24$ pixels/mm, the number of colour printing units $c = 7$, and simplex ($p = 1$). Taking (5.1), the stream-productivity or data rate is now 487.2 MB/sec. For high-end systems this could be up to 9 GB/sec.

[1] Plexity is a design quantity that indicates whether a sheet can be printed on just one side (simplex), or on both sides (duplex). Its value is either 1 or 2.

5.2.2 Colour Image Processing Pipeline

In this section we describe a simplified but functional colour image processing pipeline, that is composed out of some difficult to parallelise modules. First an overview is given of the pipeline, subsequently all pipeline modules are introduced.

5.2.2.1 Overview

Colour image processing deals with the transformation of an input colour image (in RGB colour space of the scanner) into an output colour image (expressed in the 7 process colours $K \cdots Y$) with good quality. This transformation consists of five modules, and each of them fulfills a specific task, see Figure 5.3. All blue hatched modules and the half-toning module are part of the system. The half-toning module is examined in more detail, see Section 5.3 and Section 5.4. Also shown is the amount of communication between the blocks, indicated as bits per pixel. The individual blocks are explained in the following sections.

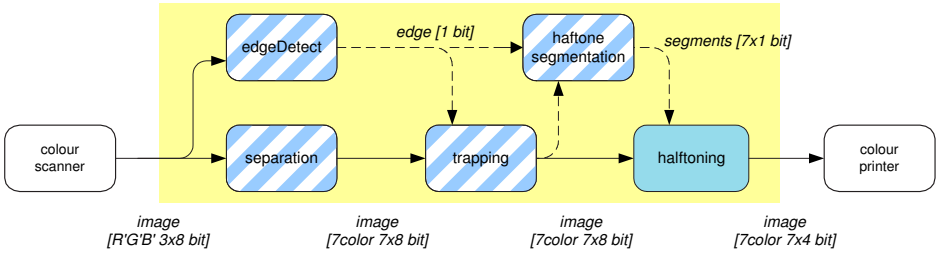


Figure 5.3: Simplified image processing pipeline

For our convenience we will use the notation (i, j) as well as s to designate a pixel. The pixel space $\mathcal{S}$ of an image is the set of all pixels s and is defined as the cartesian product $\mathcal{S} = \{\{0 \cdots H-1\} \times \{0 \cdots W-1\}\}$, where $H = r_v \cdot h$ and $W = r_h \cdot w$, and r_v, r_h are the horizontal and vertical resolution respectively.

5.2.2.2 Separation

In general the separation stage of an image processing pipeline, also known as colour space conversion, translates the output of the scanner image data into process colours that are available in the printer. Inkjet printers for example usually have 4 process colours, CMYK (short for cyan, magenta, yellow, and black). In that case separation involves a $3D \rightarrow 4D$ colour space conversion since a scanner represents each colour value in 3 RGB bytes.

Our specific printer on the other hand has 7 colours and the separation stage

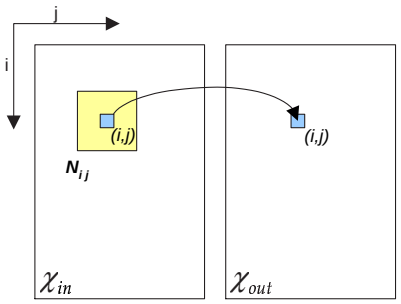


Figure 5.4: A neighbourhood operation needs besides the value of a pixel (i, j) also those in its neighbourhood $\mathcal{N}_{ij}$

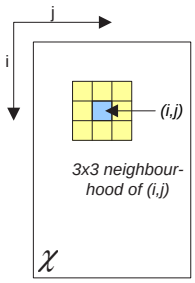


Figure 5.5: A 3×3 neighbourhood around a pixel (i, j)

has to translate scanner data R', G', B'[2] (3×8 bits per pixel) into 7 toner colours (7×8 bits per pixel) that are available in the printer: black (K), blue (B), red (R), green (G), cyan (C), magenta (M) and yellow (Y). The involved transformation can be described by:

$$(K, B, R, G, C, M, Y) = \vec{f}(R', G', B'), \tag{5.2}$$

where $K, B, R, G, C, M, Y, R', G', B' \in [0 \cdots 255]$.

Various algorithms exist for the separation task. However, high quality colour space conversion is a highly non-linear operation [67] and the best results – in the sense of perceptual quality – are not obtained with an algorithm but with a look-up table (LUT). The look-up table is a large LUT with $2^{3\times8=24}$ entries of 7×8 bit each, resulting in 940 Mbit in total.

5.2.2.3 Edge Detection

Edge detection is a neighbourhood operation that determines whether a pixel is an edge pixel or not. In general a neighbourhood operation needs the local environment of a pixel in order to determine its result, see Figure 5.4. This can be expressed in the form [56]:

$$\chi_{out}(i, j) = T_{\mathcal{N}_{ij}}\big(\chi_{in}(i, j)\big), \tag{5.3}$$

where $\chi_{in}(i, j)$ is the input image, $\chi_{out}(i, j)$ is the output image, and T is an operator on χ_{in} defined over a neighbourhood $\mathcal{N}_{ij}$ of pixel (i, j). A basic implementation of (5.3) is shown in Figure 5.5; typically the neighbourhood kernel is rectangular and centered around (i, j). The kernel is moved over the pixels in the image (2D convolution) to generate an output image [56]. Note that this does

[2] To avoid confusion with the 3 similar printing process colours R, G, B, the colour space quantities of the scanner are primed.

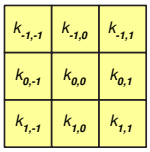


Figure 5.6: General 3×3 neighbourhood kernel

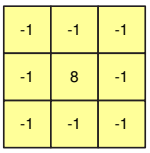


Figure 5.7: Simple edge detection kernel

not necessarily imply that pixels have to be processed in a sequential manner. The output $\chi_{out}(i,j)$ for a 2D 3×3 neighbourhood operation can be computed by multiplying the pixel and its neighbourhood with corresponding weights in a 3×3 matrix or *kernel*, and adding them together. The neighbourhood operation is defined on $0 < i < H - 1$ and $0 < j < W - 1$ by the following equation:

$$\chi_{out,K_{\mathcal{N}}}(i,j) = K_{\mathcal{N}} \otimes \chi_{in} = \sum_{k=-1}^{1} \sum_{l=-1}^{1} K_{\mathcal{N}}(k,l) \cdot \chi_{in}(i+k, j+l), \tag{5.4}$$

where $K_{\mathcal{N}}$ is the used kernel, χ_{in} is the colour plane (separation) and $\otimes$ stands for the neighbourhood operation. Examples of neighbourhood operations are noise removal, edge detection or error-diffusion. The kernel coefficients $k_{-1,-1} \cdots k_{1,1}$ (Figure 5.6) are tuned for the typical neighbourhood operation needed.

The purpose of the edge detection module is to assist other modules in making the right choices how to process individual pixels [56]. The functionality can be expressed as a neighbourhood operation performed on the R', G', B' colour values of a scanned image. Edge detection can be described by thresholding the per pixel summation of the absolute values of three neighbourhood operations – for all three colours $R'G'B'$ – with a 3×3 kernel:

$$edge(i,j) = threshold < \sum_{k \in \{R',G',B'\}} \Big| K_{3\times3,edge} \otimes \chi_k(i,j) \Big|, \tag{5.5}$$

where *threshold* is an experimentally determined constant, $K_{3\times3,edge}$ is an edge detection kernel, and $\chi_k(i,j)$ is the value of pixel s with colour $k \in \{R', G', B'\}$. See Figure 5.7 for an example of an edge detection kernel.

5.2.2.4 Trapping

The purpose of trapping is to reduce the visibility of a small misalignment of colour units (see Figure 5.1) and therefore enhancing the quality[3]. Figure 5.8 describes the potentially misalignment and the counter measure undertaken by trapping. Suppose our intention is to print a red square with a green border, as shown in

[3] Adobe Systems Inc.: "How to trap using Adobe trapping technologies", 2002, `http://www.adobe.com/products/extreme/pdfs/trapping.pdf`.

Figure 5.8(a) (topview). The red and green separations are aligned, resulting in a perfect print. Figure 5.8(b) however, shows a small misalignment and this results in a small white artifact between the green border and red square. Even a small unwanted white area of $\pm 20 \mu m$ between adjacent colours is disturbing for the human eye at a normal viewing distance. The most disturbing effect of a small mismatch is with text or lines on a uniform background printed on white paper, see for example the white gap in Figure 5.8(b).

Trapping now decreases the visibility of such misalignments by enforcing an overlap between the different colour separations. It does so by slightly dilating the image in the so called *background colour*. Trapping is a process operating on pairs of colour separations: a *foreground colour* that is, one out of $1 \cdots 6$ and a background colour that is, $2 \cdots 7$, where the numbers refer to colours in Figure 5.1. In our example in Figure 5.8 red is the foreground and green the background colour because green is printed later than red. There are $\binom{7}{2} = 21$ combinations which have all to be checked for a conditional dilation. The dilated background does not develop when separations perfectly match, making use of the physical inability in the printing process to stack a second colour on top of a first one (Section 5.2.1). However, in case of a misalignment the background colour develops in those areas that remained undeveloped by the foreground colour. Figure 5.8(c) shows the filling of the white gap by extending the background colour (green).

Before going into the functional details first a few observations. Misalignments are noticed at edges and especially then when the two involved edges have opposite directions or *gradients*, see Figure 5.8(b). Trapping operates on seven colour data from the separation result and is implemented in two steps: determine edge directions (gradients) in every colour plane, and expand areas of colour when an opposing edge is found in a colour that is printed earlier. The trapping conditions have to be checked for all combinations of colour separations.

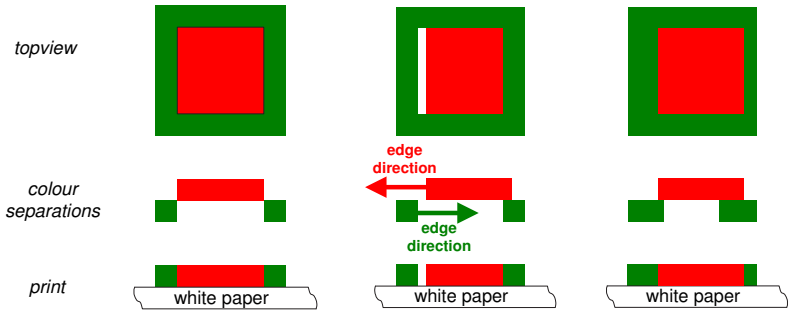
Before going into the trapping conditions we abbreviate a pixel (i, j) in a colour separation $\mathcal{A}$ by $(i, j)_{\mathcal{A}}$. A pixel $(i, j)_{\mathcal{A}}$ in colour separation $\mathcal{A}$ is said to trap pixels $(i, j)_{\mathcal{B}}$ in a colour separation $\mathcal{B}$ when all trapping conditions are fulfilled. We assume colour separation $\mathcal{A}$ is a background and $\mathcal{B}$ is foreground colour. The set of trapping-colours $\mathcal{A}$ is $\mathcal{A} = \{B, R, G, C, M, Y\}$ and the set of trapped-colours $\mathcal{B}$ is $\mathcal{B} = \{K, B, R, G, C, M\}$, since the first colour K cannot trap a colour and the last colour Y cannot be trapped. Before going into the conditions for dilation we define the scalar[4] edge gradient $\nabla_c(i, j)$ of colour separation $c \in \{1 \cdots 7\}$ for pixel (i, j):

$$\nabla_c(i, j) = \big(\, d \mid max_{d \in \{1 \cdots 8\}} \mid \chi_d - \chi_c \mid \big),$$

where $\nabla = d \in \{1 \cdots 8\}$ stands for the 8 possible edge directions, see Figure 5.9. Figure 5.10 illustrates the process. For all pixels and for all of the 21 combinations of separation planes, the following conditions are checked:

- pixels $(i, j)_{\mathcal{A}}$ and $(i, j)_{\mathcal{B}}$ are part of an edge, so $edge_{\mathcal{A}}(i, j) = edge_{\mathcal{B}}(i, j) = true$ (see (5.5)).

[4] For convenience the edge gradient is not represented as a 2D vector but as a scalar value.



(a) perfectly aligned registrations

(b) mis-registration with indicated edge directions

(c) mis-registration with trapping

Figure 5.8: Colour planes with opposite edge directions causes the trapping module to extent the colour that is printed later (in this case the green colour)

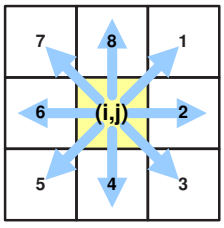


Figure 5.9: All possible directions $1 \cdots 8$ of edge gradients $\nabla(i, j)$ taken from the current pixel (i, j)

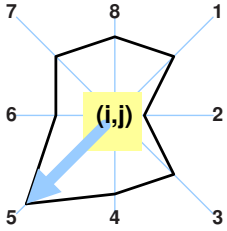


Figure 5.10: Sample radar plot of directional edges with largest gradient in direction 5

- pixels $(i,j)_{\mathcal{A}}$ and $(i,j)_{\mathcal{B}}$ have opposite edge gradients, so $\nabla_{\mathcal{A}}(i,j) = -\nabla_{\mathcal{B}}(i,j)$, where ∇_c is defined by the direction of the maximum absolute difference between the center pixel and pixels in the direct 3×3 neighbourhood in colour separation c, and where the negation of $\nabla_{\mathcal{B}}(i,j)$) is interpreted in modulo arithmetic (mod 8).
- the pixels obey the separation printing order as indicated by Figure 5.1 (e.g., a pixel with colour Y can trap a pixel with colour R but not the other way around).

To summarise, the pixel $(i,j)_{\mathcal{A}}$ is trapping the pixel $(i,j)_{\mathcal{B}}$ when the compound trapping condition $trap_{\mathcal{AB}}(i,j)$ is satisfied:

$$\begin{array}{rll} trap_{\mathcal{AB}}(i,j) = & edge_{\mathcal{A}}(i,j) & and \\ & edge_{\mathcal{B}}(i,j) & and \\ & (\nabla_{\mathcal{A}}(i,j) = -\nabla_{\mathcal{B}}(i,j)) & and \\ & (\mathcal{A}\ is\ printed\ after\ \mathcal{B}). & \end{array} \tag{5.6}$$

When a pixel's trapping condition $trap_{\mathcal{AB}}(i,j)$ is satisfied, then the colour value of the neighbouring pixel in $\mathcal{A}$, indicated by the largest gradient $(\nabla_{\mathcal{A}}(i,j))$, is copied to $(i,j)_{\mathcal{A}}$. This one pixel wide conditional dilation in separation $\mathcal{A}$ masks a possibly mis-registration of the pair $\mathcal{AB}$.

5.2.2.5 Half-tone Segmentation

Where half-toning (Section 5.2.2.6) actually transforms an 8 bit mono-colour image into a binary image, half-tone segmentation decides what specific technique for half-toning should be used. Half-tone segmentation is another neighbourhood operation, with many similarities to the edge detection stage (Section 5.2.2.3). The main difference is that it operates on each of the 7 colour planes as output by the trapping stage.

The half-tone segmentation may be described by thresholding the absolute value of the neighbourhood operation on the colour plane with a small 3×3 kernel:

$$segment(\mathcal{X}) = threshold(\mathcal{X}) < \left| K_{3\times3,segm} \otimes \mathcal{X} \right|, \tag{5.7}$$

where $\mathcal{X} = [\![\chi_{ij}]\!]_{s\in\mathcal{S}}$ is a full sized colour plane with $\mathcal{X}_{ij} = (K_{ij}, R_{ij}, \cdots Y_{ij})$ for all pixels (i,j), *segment* and *threshold* are equally sized pixel planes containing the segmentation results and the threshold respectively, and where the operator $<$ is performed component wise.

5.2.2.6 Half-toning

The purpose of half-toning is to render continuous tone information for a print engine, that has a lower tonal resolution[5] than the input bitmaps. 8-bit image

[5] Tonal resolution describes the level of detail, grey-value or colour, that can be realised per pixel by a printing process.

data can have 256 different values, but toner is binary – it is either printed or not printed [67]. Printers overcome this lower tonal resolution of the printing process by printing at a higher spatial resolution than the input bitmap. In our case, the printer uses in one dimension 4 ink dots per pixel and will print at 600×2400 dots per square inch for each of the 7 colours. These four ink dots allow for 5 quantisation levels within a single pixel: all subpixels off, and $1 \cdots 4$ subpixels on (see Figure 5.12). These 5 pixel "grey-values" realise 0%, 25%, 50%, 75% and 100% of the maximum value (100% coverage corresponds to value 255), see Figure 5.11. For the computation of one out of 5 grey-values we need to compare the value against 4 thresholds, which are 1/8, 3/8, 5/8 and 7/8 of the maximum value. Table 5.1 describes the halftoning process. This process can be represented

colour 'greyness'	condition	coverage
paper white	$0 \leq \chi_{ij} < 1/8$	0%
light	$1/8 \leq \chi_{ij} < 3/8$	25%
medium	$3/8 \leq \chi_{ij} < 5/8$	50%
dark	$5/8 \leq \chi_{ij} < 7/8$	75%
full saturation	$7/8 \leq \chi_{ij} < 1$	100%

Table 5.1: Thresholding a continuous grey-value to 5 discrete levels (quantisation levels)

as a function $f_{ht}(\chi_{ij})$. Although half-toning is not reversible in general, we do speak about the (pseudo) inverse halftoning f_{ht}^{-1}, and it returns the realised colour value χ'_{ij}.

Isolated subpixels – within a 4 subpixel slot – cannot not be printed in a physically stable way. Therefore, they are forced to join either the lower or the upper neighbour pixel. In particular cases, subpixels cluster together in the middle, see Figure 5.12. In case of a single subpixel, that is positioned in the middle (see in the figure on subpixel count=1 or count=2), no vertical clustering is possible. It is expected that a neighbouring pixel (left or right) has a subpixel count unequal to zero. A direction flag for edge pixels is computed, which indicates where half-toned subpixels should be positioned (low, up or middle). The computation simply uses the above and below pixel values.

For a good quality, half-toning depends on the pixel being an edge or not. Two specific half-toning techniques, dithering and error-diffusion, are discussed below. Each technique has its own strength: in general dithering half-tones smoothly varying information best, while error-diffusion performs better on edges. Because of the involved causality, error-diffusion poses the hardest problems for parallelisation. Therefore, it is selected for further analysis.

Dithering. Pixels in a smoothly varying neighbourhood are treated by dithering, a technique which optimises grey level quality at the expense of some spatial resolution [35]. For most pixels the dithering algorithm is used. For each

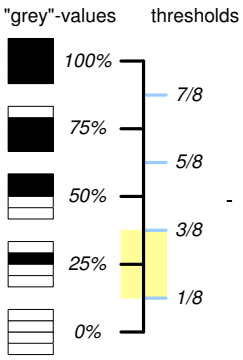


Figure 5.11: The 5 subpixel "grey-values" and the illustration of the thresholding process

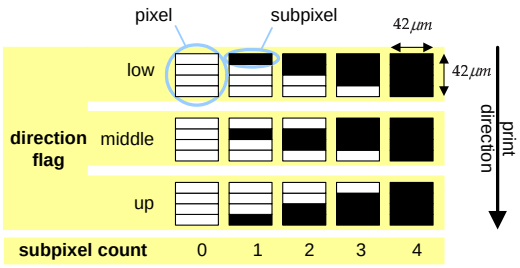


Figure 5.12: Position of the subpixels as a function of the direction flag and the subpixel count

of the 4 subpixel outputs there is a threshold. These thresholds are decided on beforehand by a delicate optimisation process that involves lots of perceptual quality tests. The subpixel is printed if the pixel value is above the threshold. A different set of thresholds (the dither kernel) is used for every pixel location and every colour component. This has the effect of spreading errors over a wide area and producing the correct perceived colour, on average. Thresholds are repeated according to some defined pattern.

Error-diffusion. Edges are treated specially in order to retain the sharpness or spatial resolution: error-diffusion spreads the error between the desired colour and the realised colour around to the pixels in the very close neighbourhood [47][35]. As with dithering, error-diffusion takes a monochrome or colour image and reduces the number of quantisation levels. A popular application of error-diffusion involves reducing the number of quantisation states to just two, which makes the image suitable for printing on binary printers. Essential is to exploit the high spatial resolution in an attempt to compensate for the lack of tonal resolution. The typical mechanism is as follows. Let s be an abbreviation for a pixel (i, j). First for each pixel s a quantisation state g_{ij} is derived from a colour pixel χ_{ij} by some half-tone function[6] f_{ht} (e.g., by thresholding) by

$$g_{ij} = f_{ht}(\chi_{ij} + \epsilon_{ij}),$$

where ϵ_{ij} represents the integral error from relevant neighbours for the current pixel s. Next, a new error ϵ'_{ij} is determined, which will be distrib-

[6] In general error-diffusion is considered a part of half-toning. Because we focus on error-diffusion we restrict the half-toning function f_{ht} to quantisation in the context of this thesis.

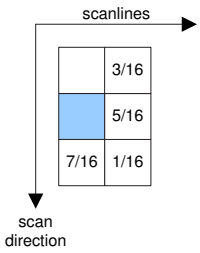


Figure 5.13: Floyd-Steinberg error-diffusion scheme $\mathcal{H}_{FS}$

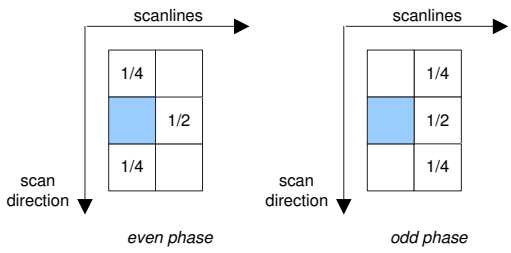


Figure 5.14: Used error-diffusion scheme, successive application of $\mathcal{H}_{even}$ and $\mathcal{H}_{odd}$

uted to some neighbouring pixels that have not been visited until now. The new error is computed by subtracting the realised monochrome colour-value $\chi'_{ij} = f_{ht}^{-1}(g_{ij})$ – the inverse halftoning of quantisation g_{ij} – from the intended monochrome colour-value χ_{ij}:

$$\epsilon'_{ij} = \chi_{ij} - f_{ht}^{-1}(g_{ij}).$$

Finally, the new error ϵ'_{ij} is distributed (and accumulated) to neighbouring pixels by some distribution kernel or diffusion mask $\mathcal{H}$:

$$\mathcal{E}'_{ij} = \mathcal{E}_{ij} + \mathcal{H}(\epsilon'_{ij}),$$

where $\mathcal{E}_{ij}$ and $\mathcal{E}'_{ij}$ are spatial error-distributions with the same size as $\mathcal{H}$. A popular error-distribution scheme $\mathcal{H}_{FS}$ is defined by Floyd and Steinberg in 1975 [47], see Figure 5.13. To ensure that all quantisation errors are diffused, $\mathcal{H}$ must satisfy the constraint $\sum_{s\in\mathcal{S}} h_{ij} = 1$, where h_{ij} represents the fraction of the error for pixel s within the set of pixels $\mathcal{S}$ for which the distribution kernel $\mathcal{H}$ is defined.

In our case, however, a different scheme is used. Every pixel propagates an error output to 3 neighbours and all pixels send an error component to the next line, see Figure 5.14 and Figure 5.15. As can be seen from this figure, even pixels send errors to the odd pixels on the same (vertical) line, and the even pixel on the next line, while odd pixels send errors to the odd and even pixels on the next line. Two sets of successively executed distribution schemes are used for this purpose, see Figure 5.14. This approach enforces an error propagation on scan line bases, realising adequate parallelism within each scanline. First doing the calculations for all even pixels (in parallel), then all odd pixels (in parallel) and working sequentially from the left of the page to the right.

We conclude the familiarisation phase with showing how a blue continuous tone original is rendered by the described error-diffusion algorithm, see Figure 5.16.

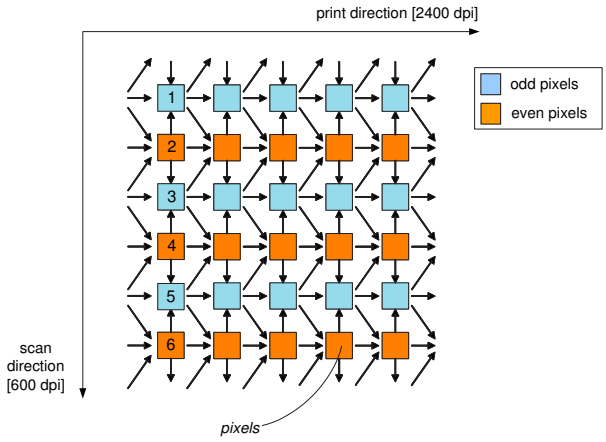
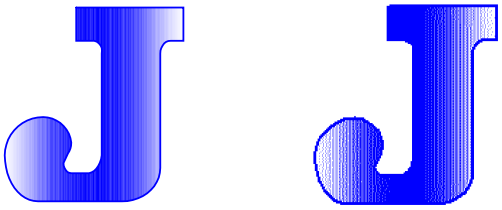


Figure 5.15: Half-toning error propagation, scanlines arranged vertically



(a) Original continuous tone image

(b) Error-diffused image

Figure 5.16: Rendering a continuous tone image with the described error-diffusion technique

This example is chosen on purpose because of the strict horizontal intensity gradation as depicted in Figure 5.16(a) and demonstrates the effect of its coarse approximation after half-toning (note the typical vertical striping).

5.2.3 Feasibility

The purpose of this section is to get confident about the scope of the system and the technical feasibility of realising the system with the target hardware architecture. In this way early roadblocks can be identified, evaluated and their impact assessed in an early stage.

Demarcation of the system boundary. To test the feasibility of a programmable processor solution a minimal system is composed, that covers the most

essential modules. These modules are taken from the project description of the FPGA implementation. All the blue shaded modules are within the scope of the system, see Figure 5.3, and the complete system will be evaluated in Section 5.5. For now the module half-toning will be considered in more detail during development.

Feasibility: a first estimate. To assess the full technical feasibility is difficult in this early stage of development. However, based on the provided C-algorithms of the five modules we can estimate the number of operations and subsequently obtain an estimate on timing. Table 5.2 contains the processing requirements for a sequential implementation of the colour printer. This result is obtained by profiling the sequential C algorithm. For example the separation, implemented as a (large) LUT only takes 3 cycles for a 24 bit Red Green Blue (RGB) lookup entry with a 7×8 bit KGRBCMY-result. Halftoning needs more time, per colour the half-toning takes 166 operations, so totaling up to 1162 cycles. For the whole A4 bitmap at a speed of 2 seconds per page we require 35 Mpixels $\times \frac{1518}{2} \approx 27$ Giga Operations Per Second (GOPS). A single Linedancer hosts 4,096 PEs, which all can process pixels in parallel. On the other hand, on the Linedancer processing is done at a 1 bit/clock pace: with the bit-width of variables between 3×8 or 7×8 bits this results in $\frac{27GOPS \times 24}{4{,}096} = 158$ to $\frac{27GOPS \times 56}{4{,}096} = 369$ Mclocks. The clock budget for this system (running at 400 MHz) is 400 MHz$\times$2sec$=$ 800 Mclocks. Hence, in first instance we estimate that a single Linedancer is sufficient for the processing at hand.

module	sequential operations per pixel
separation	3
edge detection	68
trapping	255
half-tone segmentation	30
half-toning	1162
total	1518

Table 5.2: All 35M colour pixels have to perform 1518 operations each, within a time slot of 2 sec

All modules, except separation, use neighbourhood computations. As in the image quantisation case, see Section 4.2.3, such computations need overlapped tiling for processing the large set of image data. For the edge detection, trapping and half-tone segmentation modules, the optimum tile dimensions for a single Linedancer, i.e., 64×64, can be used. This minimises the number of image reloads necessary for realising the overlap. Half-toning, however, needs a line based tile form for optimal processing, see Figure 5.17. So we first use in Pass 1

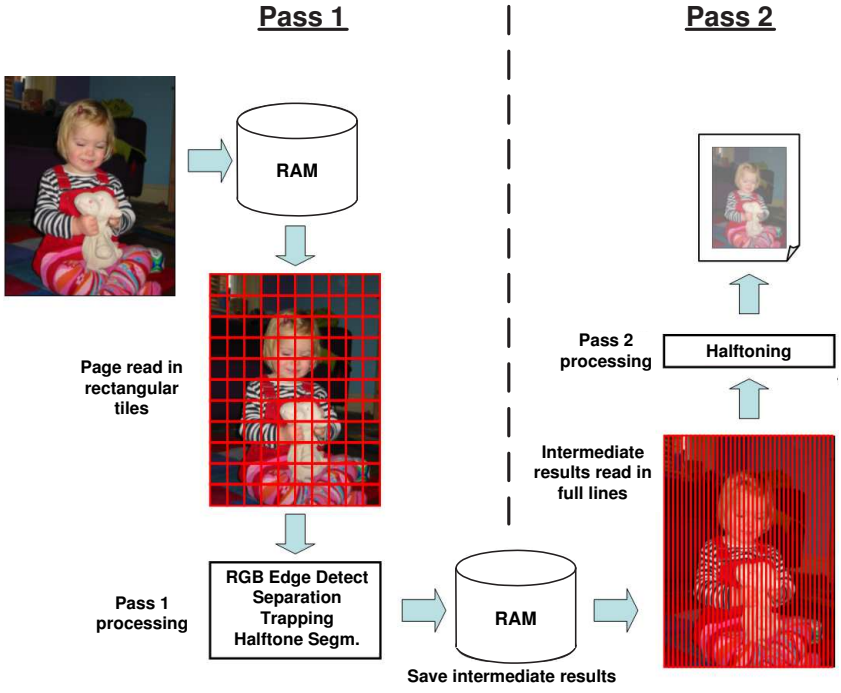


Figure 5.17: Overview of the 2 pass pipeline

64×64 tiles for edge detection, separation, trapping, and half-tone segmentation, and after that we use in Pass 2 line based tiles for half-toning.

5.3 Incremental Prototyping

We illustrate the design process by taking the error-diffusion algorithm because, in general, the class of these algorithms is hard to parallelise. The other algorithms can be parallelised in a rather straightforward manner.

5.3.1 Half-toning Algorithm: Error Diffusion

First, the half-toning algorithm is described. Then the algorithm is transformed (reduced) to an error-diffusion algorithm. Finally, the functional code for the 2D error-diffusion scheme is given. The used error-diffusion algorithm, that is

Algorithm 5.1 Error diffusion algorithm (part of half-toning)

```
 1: for all colour ∈ [0 ··· 6] do
 2:   err_{I,J} ← 0;                          // reset all errors for this colour plane
 3:   for all j ∈ J do {// for all scanlines}
 4:     for all even pixels i ∈ scanline j do
 5:       errSum_i ← χ_{i,j} + err_{i,j};                          // add error
 6:       subPix_i ← threshold(errSum_i);                          // threshold
 7:       dir_i ← dirSubPix(χ_{i,j−1}, χ_{i,j}, χ_{i,j+1});        // determine direction
 8:       valPix_i ← position(subPix_i, dir_i);  // position the required subpixel
 9:       error_i ← errSum_i − coverage(subPix_i);                 // compute error
10:       err_{i+1,j} ← err_{i+1,j} + error_i/4;                   // distribute errors
11:       err_{i,j+1} ← err_{i,j+1} + error_i/2;                   // for even pixels
12:       err_{i−1,j} ← err_{i−1,j} + error_i/4;
13:     end for
14:     for all odd pixels i ∈ scanline j do
15:       errSum_i ← χ_{i,j} + err_{i,j};                          // add error
16:       subPix_i ← threshold(errSum_i);                          // threshold
17:       dir_i ← dirSubPix(χ_{i,j−1}, χ_{i,j}, χ_{i,j+1});        // determine direction
18:       valPix_i ← position(subPix_i, dir_ij); // position the required subpixel
19:       error_i ← errSum_i − coverage(subPix_i);                 // compute error
20:       err_{i+1,j+1} ← err_{i+1,j+1} + error_i/4;               // distribute errors
21:       err_{i,j+1} ← err_{i,j+1} + error_i/2;                   // for odd pixels
22:       err_{i−1,j+1} ← err_{i−1,j+1} + error_i/4;
23:     end for
24:   end for
25: end for
```

lines	variable	function	description
5,15	$errSum_i$		$\chi_{i,j}$ is the to be half-toned colour value ('intention'), $err_{i,j}$ is the diffused error to pixel (i,j) ('social duty')
6,16	$subPix_i$	$threshold$	thresholding the intended colour value yields a quantisation $0\%, 25\% \cdots 100\%$, see Figure 5.11 and Table 5.1
7,17	dir_i	$dirSubPix$	the function $dirSubPix$ computes the direction flag as indicated by Figure 5.12
8,18	$valPix_i$	$position$	the function $position$ selects one of the 10 unique subpixel half-toning patterns (Figure 5.12)
9,19	$error_i$	$coverage$	the function $coverage$ computes the inverse half-toning $f_{ht}^{-1}(g_{ij})$, where $g_{ij} \in \{0,1,2,3,4\}$ as indicated by Figure 5.12, and the inverse half-toning by Figure 5.11

Table 5.3: Half-toning details

implemented before on an FPGA, uses a 2-phased approach. First all the even pixels of a pixel line are processed in parallel followed by the odd pixels, as can be seen in Algorithm 5.1. It should be noticed that Algorithm 5.1 is almost literally transcribed from the colour printer project documentation.

We now describe the algorithm in more detail. The algorithm sweeps for all colours sequentially through all scanlines, of which all even pixels (even phase) and all odd pixels (odd phase) are processed in parallel. The even phase is described by lines 5 through 12, the odd phase by lines 15-22. The even and odd phases are very similar, and only differ in the distribution of errors (lines 10-12 and 20-22). The processing steps that lead to the value of the error to be distributed, $error_i$, are described in Table 5.3.

We now focus on error-diffusion and present an adapted algorithm from which the half-toning is split off and the various functions ($threshold$, $dirSubPix$, $position$, $coverage$) are collapsed into a single function f, see Algorithm 5.2.

The next step is to translate the mathematical model into functional code. First of all, we would give a mathematical formulation of the above model, but as in the previous case (see Chapter 4) this formulation is an almost one-to-one copy of its functional code transcription. For this reason we skip the mathematical model and give the functional code formulation immediately. This formulation however, is a result of exploring the design space and takes some time to derive. For the sake of completeness the half-toning function f_{ht} can be transcribed in a completely separate Haskell function `g` and is added at the end of the functional code below.

Algorithm 5.2 Error diffusion algorithm (part of half-toning)

```
1:  for all colour ∈ [0 ··· 6] do
2:    err_{I,J} ← 0;                          // reset all errors for this colour plane
3:    for all j ∈ J do {// for all scanlines}
4:      for all even pixels i ∈ scanline j do
5:        error_{i,j} ← f(χ_{i,j} + err_{i,j});                    // compute error
6:        err_{i+1,j} ← err_{i+1,j} + error_{i,j}/4;               // distribute errors
7:        err_{i,j+1} ← error_{i,j}/2;                             // for even pixels
8:        err_{i−1,j} ← err_{i−1,j} + error_{i,j}/4;
9:      end for
10:     for all odd pixels i ∈ scanline j do
11:       error_{i,j} ← f(χ_{i,j} + err_{i,j});                    // compute error
12:       err_{i+1,j+1} ← err_{i+1,j+1} + error_{i,j}/4;           // distribute errors
13:       err_{i,j+1} ← error_{i,j}/2;                             // for odd pixels
14:       err_{i−1,j+1} ← err_{i−1,j+1} + error_{i,j}/4;
15:     end for
16:   end for
17: end for
```

```
error (i,j) = 0                              , if i = -1 \/ j = -1 \/ i = H
            = f (chi(i,j) + err(i,j))        , otherwise

err (i,j)   = error(i+1,j-1)/4 + error(i,j-1)/2 + error(i-1,j-1)/4  , if even j
            = error(i+1,j)/4   + error(i,j-1)/2 + error(i-1,j)/4    , if odd j

halftone2D  = g ( chi(i,j) + err(i,j) )
```

We remark that this specification in itself is very clean but it should be noted that it is computationally very inefficient. In the final hardware realisation this inefficiency will be removed. The clarity of a functional language description is shown by the one-to-one correspondence of a demand-driven dataflow representation in Figure 5.18 on the one hand, and the description in functional code for function `err` on the other. The imperative description of Algorithm 5.1 and Algorithm 5.2 is more difficult to understand because the values of the various variables (e.g., $err_{i,j}$) depend on its computation 'history', while this is not the case for a specification in a functional program. This is called *referential transparency*, and it means that the same name has the same values everywhere, without the need to know how the value was computed.

5.3.2 Implementation independent aspects

An FPGA design exists and the algorithms have already been developed. The quality of algorithms has been established and fixed and the Linedancer implementation should be 100% identical. Hence, there is no need for an image quality

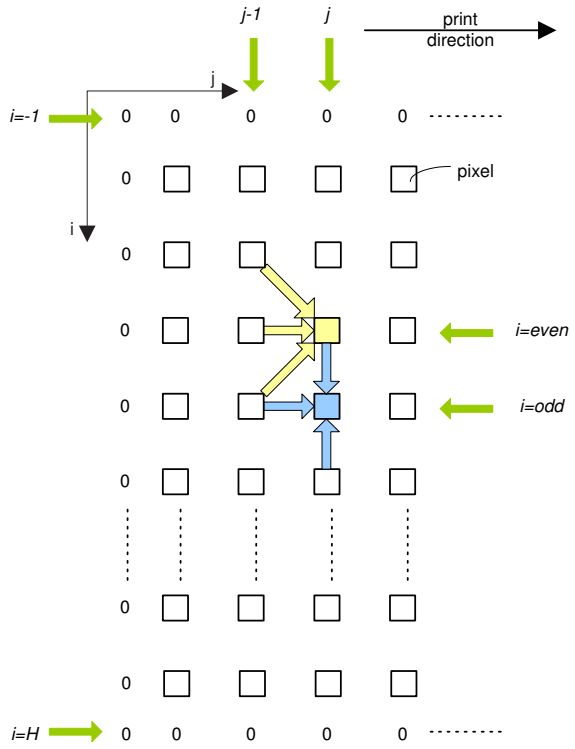


Figure 5.18: Demand-driven dataflow representation for the even pixels (yellow arrows) and for the odd pixels (blue arrows). The borders are extended with zeros to avoid a special treatment on the edges.

function to guide a trade-off subphase (which is consequently omitted). Models for other extra-functional properties are not useful because the functionality is known in all its details and the amount of dedicated Linedancer code is relatively small.

5.4 Transformational Development

In this phase the mapping to the Linedancer is addressed. We follow the standard template of the IRIS methodology except for the trade-off subphase which is omitted.

5.4.1 Global system considerations

In this section a slightly more elaborated analysis is made on the timing and storage design space compared to the feasibility study in Section 5.2.3. Finally, relevant general system architecture issues are considered because they can also influence the implementation process.

Global timing analysis. The processing of each pixel on the printed page is relatively simple, being mainly based on 3×3 neighbourhood kernels. However, the total image processing pipeline is a challenging task because of the volume of data involved, that has to be processed within two seconds.

Many of the tasks in the imaging pipeline can be implemented for many pixels in parallel. To get an improved timing estimate compared to the feasibility study, a more detailed analysis has been conducted. Algorithm details of the modules were paired with the target hardware architecture features. Table 5.4 is the result of this analysis. The table contains the performance estimates of each module for

module	processing tile	P1 clocks/ tile	HD clocks/ tile	number of tiles	P1 clocks/ page	HD clocks/ page
edge detection	ASP	5K	2K2	9K9	49M5	21M7
separation	DMA	4K-100K	6K5	9K9	39M6-990M	64M7
trapping	ASP	14K	7K3	9K9	138M6	72M3
half-tone segmentation	ASP	11K9	5K	9K9	116M8	49M5
half-tone	ASP	39K8	17K5	5K	199M	87M5
total					544M-1494M	295M

Table 5.4: Estimate of system performance for Linedancer-P1 and HD

the Linedancer-P1 as well as the Linedancer-HD. Different modules can require different tiling strategies, and as a consequence, require a different number of

Colour image processing

module	dependency on line order	Linedancer subsystems: Sparc SDS	PE CAM or EXT	Associative CAM-SDS	DMA SDS-PDS
edge detect			√		
separation		√			√
trapping			√		
half-tone segmentation			√		
half-toning	√	√	√	√	
loading and dumping of tiles		√			√

Table 5.5: Mapping of the various modules of colour image processing on the memory and processing subsystems of the Linedancer

tiles (5^{th} column), see Section 5.4.3.1 for more details. Also indicated is a first mapping to processing units given by the 2^{nd} column.

The following conclusions can be drawn:

- The Linedancer-P1 cannot meet the performance requirement (1494 Mclocks $>$ 600 Mclocks, $f_{P1} = 300$ MHz) in worst case. The HD can meet the requirement.

- The separation module is the bottleneck, because of the demanded table lookup functionality, that cannot be parallelised as easy as the other modules. The reason that this problem did not show up during the first feasibility estimate (Section 5.2.3) is because of the extremely low number of operations per pixel, that gave no cause for alarm.

- Edge detection and separation run in parallel at the same time because they both just need the scanner $R'G'B'$ and they are mapped to different processing units, see second column in Table 5.4. Pipelining is expected to reduce the integral processing time even further (Section 5.5).

Global storage allocation. An analysis of the storage allocation for the five modules has been conducted, see Table 5.5 for a global overview. The LUT used in the separation stage is kept in the SPARC's memory space SDS, see Section A.2. Because half-toning uses a separate pass its relatively large memory utilisation is manageable. For a description of the Linedancer subsystems we refer to Section 2.3.4. The format of these columns corresponds with those given in Section 4.4.1.

Scalability. Before conducting the mapping we should give some attention to desirable design properties as scalability. The colour image processing system should be scalable in performance, resolution, and page sizes.

Our system consists of five modules that all run in parallel or sequentially on the Linedancer. It uses a two pass pipelined scheme to process the colour images. Although the separation module uses the DMA controller in parallel to the processing array (PEs), it takes the most time in the first pass. Its performance scales almost linearly with the number of Linedancer chips. All remaining modules in pass 1: edge detection, trapping and half-tone segmentation, run on the ASP and scale easily with the number of Linedancers. The error-diffusion scheme cannot scale anymore in performance because for this specific printer the maximum parallelism is achieved. But when the resolution or the page sizes increase – meaning more tiles or lines to process – then scaling may be considered.

5.4.2 Trade-off subphase

Because the system should be an exact functional copy of an earlier FPGA based system, no trade-off on functionality is allowed. Therefore, this study does not apply.

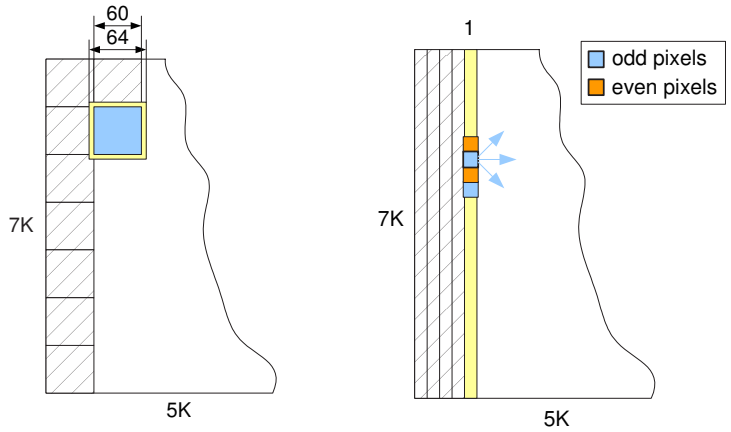
5.4.3 Reorganisation subphase

During the reorganisation subphase the model is, in general, expanded in a top-down manner, gradually changing the hardware independent description into a form dictated by the hardware architecture. Although in this case the algorithm was already optimised for a parallel (FPGA) hardware implementation, we still need to consider Linedancer specific mapping details. To demonstrate the added value of this subphase we have included an example: line-wide tiling.

5.4.3.1 Tiling

Because the size of the entire bitmap is much larger than the available storage space in the Linedancer we have to use bitmap partitioning or tiling. See Figure 5.19 for the two used tiling strategies in the two passes. Square tiling as illustrated in Figure 5.19(a), is used for edge detection, separation, trapping, and half-tone segmentation. Line tiling as illustrated in Figure 5.19(b) is used for half-toning. Half-toning can be done with a single line-wide tiling because all involved neighbourhood operations are already performed by half-tone segmentation.

It is not clear at this point how the 2D error propagation array in the mathematical model (Section 5.3.1), and in the functional code `err(i,j)` has to be transformed in a 1D variant. The 2D description is concise but its implementation is already inefficient for a sequential processor, let alone on a parallel hardware architecture. However, we can remove the deep recursive structure of `err(i,j)` by the addition of some state. In this manner we are able to transcribe the 2D



(a) In pass 1 a rectangular tiling scheme is preferred

(b) The chosen error-diffusion scheme enforces a line-wide tiling of the bitmap in pass 2

Figure 5.19: Different tiling strategies for the two passes

model into a more computationally efficient 1D model, but we lose conciseness of description. See the functional code below. This code needs some clarification.

First, the function `errorr` computes the error – using error function `f` – for pixel `i` in the current scanline for the even and the odd phase.

Second, the function `errE` distributes the errors during the even phase to the even as well as the odd pixels.

Third, the distribution of errors during the odd phase is handled by function `err1`.

Finally, the half-toning is defined by the recursive function `htscan`. The function produces a list of half-toning results `chi1`. The function consumes a first column of colour pixels `chi i` from the array of all pixels `chis` and maintains an error accumulator column `err`.

In both phases partial errors are generated for even and odd pixels. For example, computing `error` for odd pixels is delayed until the function `errE` has computed its partial result. Note that this functional description remains concise because explicit order is avoided as opposed to the imperative description (Algorithm 5.2).

Before the description in functional code is given we first clarify the notation:

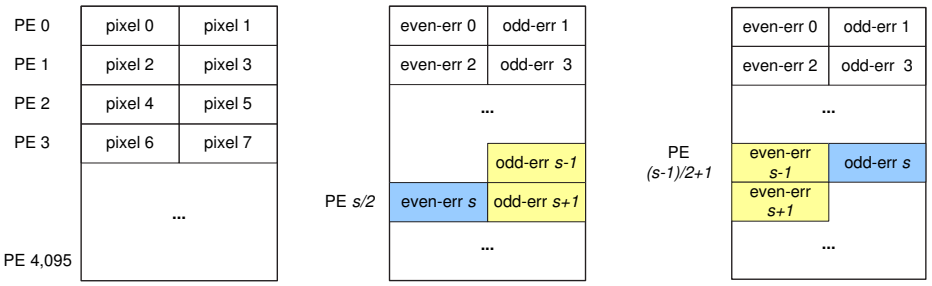
- `chis` is the matrix of colour-values pixels (χ), `chi` is a particular column of these values,
- `chi1` is the half-toned result (using the half-toning function `g`) taking column `chi` as input. The result of `htscan` is a list of these `chi1`-columns,

- `err1` is the new error-column. At every step `err1` is updated with the next `chi`-column.
- The function `even` returns a `1` (true) if its argument is even. Likewise for `odd`.
- The dimension of a column is `hh`, all values outside the interval `[0..hh]` are set to zero.
- The notation `( i -> 0 )` is short for the 'all zero' function.

```
odd  i  =  i \% 2 = 1
even i  =  i \% 2 = 0

htscan err  []       = []
htscan err (chi:chis) = chi1 : htscan err1 chis

  where

   chi1     = [ exp | i<- [0..hh-1]
                    ; exp := g (chi!i + if (even i) (err i) (errE i))
                    ]

    error i = 0                                          , if i<0 \/ i>=hh
            = f (chi!i + err  i)                         , if even i
            = f (chi!i + errE i)                         , if odd i

    errE i  = 0                                          , if i<0 \/ i>=hh
            = err i + error(i-1)/4 + error(i+1)/4        , if odd i
            = error(i)/2                                 , if even i

    err1 i  = 0                                          , if i<0 \/ i>=hh
            = errE i + error(i-1)/4 + error(i+1)/4       , if even i
            = error(i)/2                                 , if odd i


halftone1D   = htscan ( i->0 ) chis
```

In the above described functional code one can easily discern a symmetry between the even/odd pixel processing in the functions `err1` and `errE`. This will be exploited in the next section.

For optimal performance on a massively parallel hardware architecture it is essential to know of computations that are dependent on each other. Since the even and odd pixels have to be processed one after the other, they are allocated as pairs on a single PE, see Figure 5.20(a). In this way a single Linedancer (with 4K PEs) can host up to 8K pixels, enough for our purposes.



(a) Packing even and odd pixels per PE

(b) Even pixels accessing their odd pixel neighbours

(c) Odd pixels accessing their even pixel neighbours

Figure 5.20: Allocation of even and odd pixels and their access to neighbours

5.4.4 Template subphase

During the template subphase we are looking for common patterns, that can raise the quality of code and save on development effort. Although not directly visible from the algorithm in Algorithm 5.1, the even and odd error-distribution hosts a pattern that can be exploited for reuse. First, the quantitative distribution of errors $\frac{1}{4} : \frac{1}{2} : \frac{1}{4}$ is the same for both the even and odd pixels. Second, as can be seen from the functional code in the previous section, both error functions look very similar. We can rewrite the similar lines in `errE` and `err1` such that only a single function is involved and thus show the pattern that can be reused. We will now clarify the notation in the functional code.

First, the function `F` updates the error array `v`, always for one half, dependent on the phase `ph` of pixel `i` (either even or odd), and leaves the other half undisturbed.

```
F v ph i = v(i-1)/4 + error(i-1)/2 + error(i+1)/4        , if ph i
         = error(i)/2                                    , otherwise
```

Second, the 3^{rd} and the 4^{th} line in both `errE` and `err1` can be rephrased into two different calls to the same function `F`:

```
errE i   = 0                                             , if i<0 \/ i>=hh
         = F err odd i                                   , otherwise

err1 i   = 0                                             , if i<0 \/ i>=hh
         = F errE even i                                 , otherwise
```

Our ultimate goal is to obtain a description in the language of the target hardware architecture. This architecture, however, is 'imperative' in a way that it uses pieces of code that change a state in a pre-specified order. Bridging the gap between a functional description and an imperative one is too large to make in one step. Therefore we start with an intermediate step as prescribed by IRIS (Section 3.5). First we provide the imperative variant in a parallel pseudo-C language and explain the algorithm below.

```
odderror  = 0
evenerror = 0
for every line of pixels
    load evenpixel & oddpixel

    evenpixel += evenerror
    do half-toning to get evenoutput
    evenerror  = (evenoutput - evenpixel) / 2
    lower odderror  +=  evenerror / 2
    upper odderror  +=  evenerror / 2

    oddpixel  += odderror
    do half-toning to get oddoutput
    odderror   = (oddoutput - oddpixel) / 2
    lower evenerror +=  odderror / 2
    upper evenerror +=  odderror / 2
end for
```

At the start of the algorithm the 1D representation of all even and odd errors is initialised (in parallel) before looping over over all scan lines. The loop starts each time with the loading of the even and odd pixels (χ_{2k}, χ_{2k+1}). Then two compound blocks are being processed one after the other. The first one computes the errors of even pixels and distributes this error to even and odd (neighbouring) pixels. This corresponds to the function `errE` of the 1D functional code variant on page 135.

Next the process is repeated but now for the odd pixels. Note that the odd pixel values accumulate the odd errors changed by the previous compound block; this is effectuated by the separation of the error function `errE` and `err1` in the functional description on page 135.

From this context it is obvious that the distribution of even errors and the subsequent accumulation for the odd pixels is symmetrical with respect to both actions on odd and even pixels respectively, during the second part of the loop. We now give the imperative variant of function `F` on page 135. This template, called `distributeAndAccumulateError(X,Y)` uses two parameters (`X`,`Y`) that have the value `(X,Y)=(even,odd)` for the even phase. For the odd phase the value is `(odd,even)`.

Example

The template, which has at this point 2 parameters (X,Y), can now be defined by the following implication.

template body		generic template call

`Xpixel += Xerror`
`do half-toning to get Xoutput`
`Xerror = (Xoutput-Xpixel) / 2`
`lower Yerror += Xerror / 2`
`upper Yerror += Xerror / 2`
	⇒	`distributeAndAccumulateError(X,Y)`

End example

5.4.5 Translation subphase

The last step of developing code is the generation of a target hardware language description. Templates can also play a role in this step, because they can provide for supporting structures, that facilitate the process of writing code. This is especially true for deviating processing architectures that cannot be written in plain C.

To demonstrate this we show the translation of an instance of the *distributeAndAccumulateError*-template. In the translation text below, the substitutions are highlighted. Note that with respect to the previous template example, an extra parameter is introduced, that encodes the relative position (-1) of the lower neighbour of the even pixel. This is caused by the packing of a pair of even and odd pixels in a single PE.

Example

This example demonstrates the generation of Linedancer code from a template. The template computes the error-distribution from an even pixel perspective. The righthand side of the implication below, contains a large Linedancer specific `aop{...}` instruction sequence with eight instructions that are processed sequentially (for all PEs in parallel), see Section A.1. The example makes extensive use of bit-field logistics, see also Figure 5.21 for the memory field specifications. The first four describe the division of the even error by 2 (with sign extension, $\epsilon \in [-128 \cdots + 127]$) and store the result in the new even error, see Figure 5.20(b)[7]. The fifth instruction adds $\epsilon/4$ to the error of higher odd neighbour, administered on the same PE. The last three instructions deal with the lower odd neighbour and need extra communication directives (Get and Put) to effect the actual distribution.

[7] Note that the memory field `temp10A` is reused for the even as well as for the odd phase.

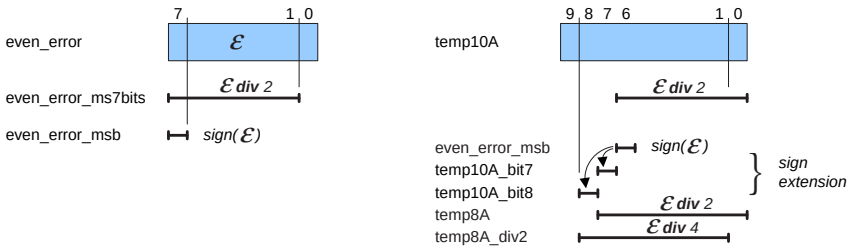


Figure 5.21: Memory field specifications used in the processing of even errors

```
distributeAndAccumulateError("even","odd",-1)
⇒
aop{
    // divide the even error value by 2 and sign extend
    RTS(Assign, Bitset, -, @{temp10A},      @{even_error_ms7bits},-),
    RTS(Assign, Bitset, -, @{temp10A_bit7}, @{even_error_msb}, -),
    RTS(Assign, Bitset, -, @{temp10A_bit8}, @{even_error_msb}, -),
    RTS(Assign, Bitset, -, @{even_error},    @{temp8A},  -),
    // add error/4 to neighbouring odd pixel on same PE
    RTS(Add, Int, -, @{odd_error}, @{odd_error}, @{temp8A_div2}),
    // add error/4 to neighbouring odd pixel on neighbouring PE
    RTS(Assign, Bitset, co{Get, -1}, @{temp8B}, @{odd_error}, -),
    RTS(Add, Int, -, @{temp8B}, @{temp8B}, @{temp8A_div2}),
    RTS(Assign, Bitset, co{Put, -1}, @{odd_error}, @{temp8B}, -),
};
```

The other instance of the template, called by `distributeAndAccumulateError("odd","even",+1)`, generates the code for the odd pixels, see Figure 5.20(c).

End example

5.5 Results and Discussion

In this section the results of the previously elaborated modules are combined in order to formulate a conclusion on the feasibility of the functionality and the timing of the Linedancer implementation. It serves as a basis for comparison with FPGA technology.

Timing. The system has been designed completely but partly implemented on a Linedancer-P1. Because the P1 cannot meet the performance we selected the Linedancer-HD as the target hardware. Pass 1, consisting of loading RGB, edge detection, separation, trapping and half-tone segmentation takes 16K5 cycles per tile. The effective area of a tile is $(64-4) \times (64-4) = 3600$ pixels because of

the 2 pixel wide overlap at each side. As already pointed out in Section 5.4.1 pipelining is a way of improving the throughput of a system, see Figure 5.22. We will now discuss the pipeline of pass 1. At first the *RGB* image data (for image frame $[n]$) is loaded into the Linedancer. After the RGB interleaving for optimised lookup[8] both edge detect and separation are executed in parallel. Trapping for iteration [n] cannot start before both edge detection and separation end because it needs both to finish first. Since separation takes significantly longer than edge detection, trapping on the current iteration is postponed, a pipeline stage is introduced and processing continues with trapping on the previous tile data ($[n-1]$). When separation finishes it stores the separation data for $[n]$ on the ASP, after trapping has finished iteration $[n-1]$ and before half-tone segmentation of $[n-1]$ commences. This data will be used for trapping the next iteration $[n]$. The result of half-tone segmentation is stored for each individual colour plane and can be reloaded efficiently for half-toning in pass 2. Note the jig-saw like packing of separate pipe stages as indicated by the pink line in Figure 5.22. Because separation runs in parallel with the other modules the amount of cycles per tile is $2K + 2K2 + 7K3 + 5K = 16K5$ cycles per tile.

[8] Interleaving the three RGB colour bytes such that the lookup index contains the higher colour bits $R_7G_7B_7$ in the most significant positions, followed by the next high bits $R_6G_6B_6$ etc. reduces the probability (and on average the time) of accessing a new DRAM page.

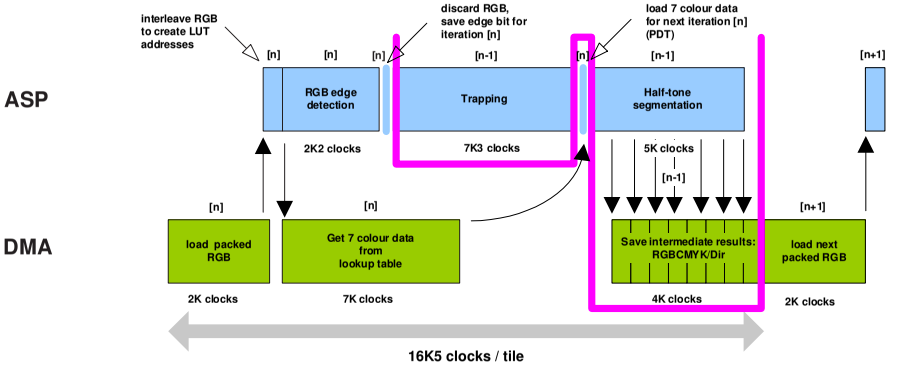


Figure 5.22: Pass 1 comprises separation, edge detect, trapping and half-tone segmentation, shown is the processing order for a single tile. The pink line separates pipeline stages.

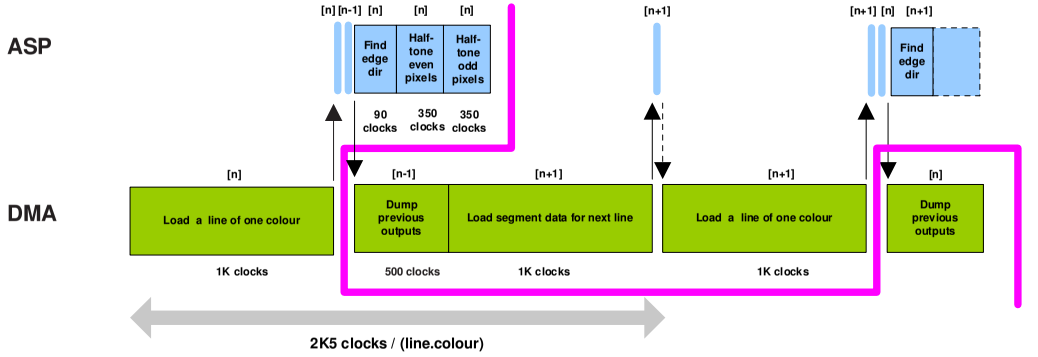


Figure 5.23: Pass 2 comprises half-toning, shown is the processing order for a single line and single colour. The pink line separates pipeline stages.

Figure 5.23 shows the pipelining of the second pass for a single line and for a single colour. At a certain point in time the current half-toning ($[n]$) on the ASP runs in parallel to the dumping of the previous halftoning results $[n+1]$ and the subsequent loading of the next segment data (a result from half-tone segmentation). After loading this segment data for iteration $[n+1]$ a new line of to be half-toned data is loaded ($[n+1]$). The ASP swaps this data in and dumps the half-tone results ($[n]$) using the DMA controller. Half-toning in pass 2 takes up 2K5 clocks per line per colour (see Table 5.6). The difference with respect to the estimates in Table 5.4 is that the execution time is now fixed by the design choices made during development. The major choices are: selection of the Linedancer processor (HD) and parallelisation of separation with edge detect and trapping. A further difference is the loading of the *RGB* tile data that could not be hidden in processing time. In this pass each line contains 7K pixels and can be fitted into one Linedancer by packing an odd/even pair of pixels into each processing element; 12% of the PEs remain unused.

ASP		DMA			
module	cycles / tile	module	cycles / tile	units	cycles / page
PASS 1					
edge detection trapping half-tone segmentation	2K2 7K3 5K	load tile separation	2K 6K5		
	16K5 cycles per tile			9K9	163M
PASS 2					
half-tone	17K5 cycles per line			5K	88M
TOTAL					251M

Table 5.6: Performance estimate for Linedancer-HD

The overall processing time is 251M cycles per A4 page, which is equivalent to 0.63 seconds per page, with a single Linedancer at 400 MHz. This is well below the required 2 seconds per page.

Illustrative for the power of massively parallel computing is the speedup compared to the sequential implementation. Although the processing capacity of each PE is much lower than a von Neumann processor, the number of processors working in parallel yield large speedups, see Table 5.7. Large speedups can be realised compared to the sequential case: the speedup in operations/sec can go up as far as ± 400, and up to ± 100 in execution time (based on the clock frequency ratio 1800 : 400 MHz).

[9] Based on single cycle operations.

module	sequential cycles[9] / pixel	parallel cycles / pixel	speedup
separation	3	6K5 / 3600 = 1.81	1.66
edge detection	68	2K2 / 3600 = 0.61	111
trapping	255	7K3 / 3600 = 2.03	126
half-tone segmentation	30	5K / 3600 = 1.39	21.6
half-tone	1162	17K5 / 7K = 2.5	465

Table 5.7: Measured speedup

description	symbol	range	fixed point scheme	memory field	domain	host
colour	*colour*	$[0 \cdots 6]$		global	int	SPARC
scanline pointer	l	$[0..max(h, w) - 1]$		global	int	SPARC

Table 5.8: Fixed point accuracy of parameters in the error-diffusion algorithm

Memory allocation for Error-diffusion. The final allocation is presented in two tables, one for fixed parameters used by the algorithm (Table 5.8) and the other for the variables used in the algorithm (Table 5.9).

The first table lists all parameters used in Algorithm 5.1, complete with the domain they are defined on, and the chosen fixed point representation. All parameters are hosted on the SPARC. The same generic description applies for the variables used in the error-diffusion algorithm, see Table 5.9, except for the reference to the algorithm in the last column.

description	dependency	range	fixed point scheme	memory field	algorithm-line
monochrome pixel value	$\chi_{i,j}$	$[0 \cdots 255]$	$8 \bullet 0$	*evenPixel* 64(8), *oddPixel* 72(8)	5,14
error input	$err_{i,j}$	$[-128 \cdots 127]$	$8 \bullet 0$	*evenError* 88(8), *oddError* 96(8)	5,14
direction information	dir_i	$\{0, 1, 2\}$	$4 \bullet 0$	*evenDir* 168(4), *oddDir* 172(4)	7,16
error accumulation	$errSum_i = \chi_{i,j} + err_{i,j}$	$[0 \cdots 255]$	$8 \bullet 0$	$half - tonePixel$ 0(8)	5,14
sub pixel	$subPix_i = threshold(errSum_{i,j})$	$[0 \cdots 15]$	$4 \bullet 0$	$half - toneSubpixels$ 8(4)	6,15
positioned pixel	$valPix_i = position(subPix_i, dir_i)$	$\{15, 14, 12, 8, 6, 4, 7, 3, 1, 0\}$	$4 \bullet 0$	$evenValPix$ 80(4), $oddValPix$ 84(4),	7,16
error output	$error_i = errSum_i - coverage(subPix_i)$	$[-128 \cdots 127]$	$8 \bullet 0$	`temp10A` 40(10)	8,17
distribute errors up	$err_{i+1,j} = err_{i+1,j} + error_i/4$ $err_{i+1,j+1} = err_{i+1,j+1} + error_i/4$	$[-128 \cdots 127]$	$8 \bullet 0$	*evenError* 88(8), *oddError* 96(8),	9,18
distribute errors right	$err_{i,j} = err_{i,j} + error_i/2$ $err_{i,j+1} = err_{i,j+1} + error_i/2$	$[-128 \cdots 127]$	$8 \bullet 0$	*evenError* 88(8), *oddError* 96(8),	10,19
distribute errors down	$err_{i-1,j} = err_{i-1,j} + error_i/4$ $err_{i-1,j+1} = err_{i-1,j+1} + error_i/4$	$[-128 \cdots 127]$	$8 \bullet 0$	*evenError* 88(8), *oddError* 96(8),	11,20

Table 5.9: Fixed point accuracy of variables used in the error-diffusion algorithm

Comparison with FPGA. The productivity benefits of using an associative SIMD processor approach to this problem rather than using FPGA technology are shown in Table 5.10. Since the design of the image processing pipeline is already done before, we here translate productivity by implementation effort. The ratio of implementation effort for an FPGA versus Linedancer is 100 man-days to 50 man-days (including coding, testing etc.), assuming a developer with domain and target hardware experience, see second column in Table 5.10. The FPGA estimate is based on the implementation effort of the existing system. The estimation for the Linedancer implementation is based on an extrapolation of the detailed design analysis of the existing system and on design data from the partly realised Linedancer based prototype. Aspects that play a role in the comparison between the FPGA and the Linedancer are: software programmability, the native support for data logistics (intelligent DMA), absence of hardware testing, and flexibility for handling redesigns.

So the development effort of the system can be reduced by a factor of two. This result is conform the experience of Aspex[10] in the various ports of FPGA based systems to the Linedancer technology.

technology	implementation effort [man days]	execution speed [ppm]
2 way SMP Intel 1.8 GHz Pentium Xeon[11]	10-20	2-30
FPGA Spartan2E[12]	100	30
Linedancer-HD	50	90

Table 5.10: Comparison of different technologies

Methodology and Architectural language issues. The system was initially designed and implemented for an FPGA. The goal of the port to a programmable processing environment is to test the applicability of these environments and for this purpose we select the most difficult to implement modules.

The port is – in the first design stages – guided by the proposed evolutionary development methodology. The methodology is helpful in understanding the problem domain. At the specification level the most critical modules are selected from a larger existing pipeline. The architectural language supports this selection and helped with making consistent interfaces.

The various modules are extended with coarse timing models for use in feasibility studies. Further assistance for remodelling the sequential code in a parallel way

[10] Aspex Semiconductor: "Aspex Semiconductor Technology", 2008, `www.aspex-semi.com/q/technology.shtml`.

[11] Numbers represent normal as well as the optimised case.

[12] The system with the current selection of five modules could be build using ± 5 Spartan XC2S400E devices.

is not necessary because the sofware has been set up for a parallel FPGA-based implementation already. Because expertise and legacy C- and VHDL-code is available the actual code design consists of porting the code directly to Linedancer-C. Test cases are set up, and the results are verified with the functional model, before and after the port. No concessions are made to the functionality (no trade-off), providing for a bit true golden reference.

5.6 Conclusions

An associative SIMD processor combines the speed of FPGAs with high-level software programmability and flexibility.

A design based on the 300 MHz Linedancer-P1 is capable of processing pages from 1.8 to 4.0 sec/page, depending on the colour distribution on the page.

A 400 MHz Linedancer-HD device is capable of implementing a colour image processing pipeline at a rate of 90 pages per minute, well above the required 30 pages per minute [ppm]. Large speedups per module can be realised compared to the sequential case: the speedup in operations/sec can go up as far as 400 (up to 100 in execution time).

A key issue in the design is how to partition the 35M pixels of a page into 4K chunks for processing. This apparently simple problem is complicated by the conflicting requirements of the various 3×3 kernel operations and the error propagation in half-toning.

Software defined systems enable fast developments. The development of code for a PC based solution is faster than for a programmable SIMD processor such as the Linedancer. But when real-time performance is critical, and the choice is between FPGA or Linedancer, than the use of the latter may reduce the design cycle by a factor of 2.

Because of the inherent scalable architecture the performance can scale with the number of processors with marginal changes in the code (e.g., delivering more productivity, more resolution, more colours).

IRIS supported the system development by helping with the specification of the five modules, by providing a (coarse) timing model, and helping with the verification of the correct integral function. Since this case was conducted first of the three cases, relatively extensive hardware modelling was conducted.

CHAPTER 6

Case: Mining Dynamic Document Spaces

Inspired by the success of Google, printer manufacturers are investigating – possibly paperless – document management services. One of these services is mapping dynamic document spaces, that is, improving the access to document spaces that are frequently updated (such as newsgroups), by a theme map. This process is computationally quite intensive. This chapter describes the development of this demanding part of mining dynamic document services on a massively parallel processor. A prototype has been built, which processes streams of information from subscribed newsgroups and transforms them into personalised theme maps. Although this technology does accelarate the training part compared to a general purpose processor implementation, its real benefits emerge with larger problem dimensions because of the scalable approach. The high level design stages are developed with the proposed evolutionary development methodology.

6.1 Introduction

We are living in a society that faces a deluge of information. For example, in 2005, web-based archives contained already over 11 billion indexable pages [57]. Moreover, the 'lifetime' of content becomes shorter and shorter. A related concept, the update frequency of information on the internet, is not even limited by relatively slow human interaction anymore, but by the response time of online sensors and data-bots that process them. So people are in need of tools to structure this vast amount of information and/or to inform users on new trends or remarkable events

Major parts of this chapter have been published in [P2].

in a timely manner. Google returns a (often too) long ranked list of hits, forcing the users to reformulate the query in order to obtain relevant answers. Users, however, would like to start from an overview – for instance using a geographic like map – and browse, instead of query [P7]. The user would then start with an overview and navigate to interesting subsets of the document space, eventually ending up with a short list of relevant hits. This also poses strict constraints on response times of the system: an update of the map should be made within seconds or otherwise the user loses interest. The problem is that real-time data mining is computationally very intensive. To solve this problem, a mapping of the training process to a massively parallel processing array is conducted.

The reason that this case is selected, is to probe future document business domains, to obtain experience with mapping computational intelligence technologies and to test the IRIS methodology (Chapter 3) for non-printing applications.

The study is presented in the IRIS framework structure. In Section 6.2 the reader is introduced to some relevant concepts such as: document map, data mining, Self Organising Map (SOM) technology (a neural network), and hardware architectures for SOM. Section 6.3 elaborates on the particular application and the subsequent chapters on implementation issues (Section 6.4) and results (Section 6.5). Finally, in Section 6.6 conclusions will be drawn.

6.2 Familiarisation

The goal of this phase is to gain confidence in the feasibility of realising an instant training based on the Linedancer processor Section 2.3.4. At first, we will introduce (dynamic) document maps as a way to negotiate the information overload (Section 6.2.1). Then relevant data mining technologies are introduced (6.2.2), followed by a more detailed description of a neural network training. The final section describes the first order feasibility (Section 6.2.4).

6.2.1 Information overload and Document maps

Information with a short life time, such as news, is preferably distributed in a digital manner. For this reason newspaper companies have their own online web-based versions of their printed publication. News agencies such as Cable News Network (CNN), Reuters International and others provide Really Simple Syndication (RSS) services that deliver personalised news instantaneously or at least at regular intervals within a one day timeframe. One way to master the information push to a user is to change the carrying medium from text to graphics (as inspired by the saying "a picture tells more than thousand words")[1].

[1] The presentation of the overview is done graphically; the content itself remains in textual form.

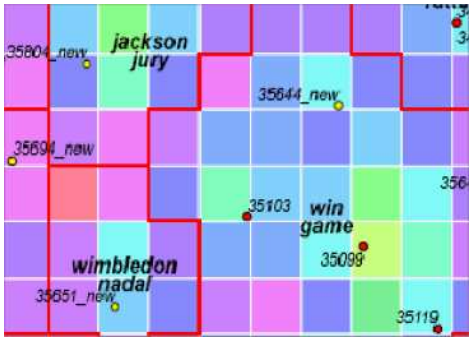


Figure 6.1: Part of an interactive map of newsgroup articles. Articles are grouped in themes (countries) and are linked to the original news articles.

Document Maps. The map metaphor has become popular in information visualisation [107][95][P7][87]. In [5] an entire text collection is presented to a user through a two-dimensional map, where each category in the map is associated to a set of documents. The closer two categories are in the map, the more similar the contents of their associated documents are. Figure 6.1 shows a part of a map in which categories are visualised as *countries* bordered by red lines and documents are visualised as *cities* by small red dots. Also, the closer the documents (cities) are on the map the more similar they are. Besides proximity of documents, the colour of the square patches that make up the map, also indicates whether neighbouring patches are similar or different. Once an interesting document is found on the map, the user can retrieve other similar documents by clicking on other documents in its vicinity[2].

It has been shown that humans have powerful visual recognition abilities, and spatial and visual representations are easier to learn, understand and communicate than textual information [115]. It is also known that the interactive cycle in a system should be designed in such a way that it exploits the extreme high bandwidth of the human visual channel [121]. These two observations are the foundation for our choice to select a pictorial representation – with a spatial ordering – for dynamic document maps.

Dynamic Document Map. In this paragraph we discuss the dynamic aspects that come with training and visualisation of news articles.

An interesting aspect of visualisation is that our recognition ability is much more effective than our recall ability [121][118]. Our ability to recognise information is particularly pronounced for pictorial information (e.g., the cognitive

[2] The shown partial example map covers the newsgroup British Broadcasting Corporation (BBC) News and BBC Sports in June 2005. It is built up by a grid of 16 by 32 square patches and each patch acts as a placeholder for a category of newsgroup articles.

spatial memory effect). A map representation of a data space could exploit this ability. Therefore, an important requirement is that the global structure of the map remains the same over time. For recurring visualisations the map is only useful if its global structure does not change significantly when new articles are incorporated. Only then the user will be able to quickly reorientate so he/she can identify the changes. We would like the system to behave in a predictable manner such that minor disturbances in input space result in minor disturbances in the document map.

In order to devise good solutions for a news visualisation system the following requirements should be satisfied. First, the system should support a single portal for all news sources, because of the ease of use. It should combine multiple sources, because this increases the reliability of the news. In order to fully utilise the visual bandwidth, the system creates a theme map, that should include the following properties:

- handling the similarity of documents (news articles) by proximity,
- providing abstraction by hierarchy (and allowing zoom and pan as navigation functions),
- providing identification of categories or documents by meaningful names, and
- supporting a view of the original document by linking its Uniform Resource Locator (URL).

This theme map should provide an accurate overview, especially respecting the topological ordering of the map. Finally it should provide instant response on new articles in the subscribed stream, because users expect rapid delivery of news.

6.2.2 Data mining technologies

Data mining is the science of extracting useful information from large data sets or databases [61]. For us, however, data mining involves the extraction of relevant data, and all the intermediate steps up to the presentation of the data to the user. A neural network is a technology that is often used in data mining. The technology is typically used for those problems that are difficult to model but have on the other hand lots of training data available. This is the reason why neural networks are popular in data mining applications. A special kind of neural network, that we use here, is the *Self Organising Map* (SOM), which is developed by Teuvo Kohonen at the Helsinki University of Technology (HUT) [77]. In this section we go into the relevant data mining technologies and focus in particular on the SOM neural network.

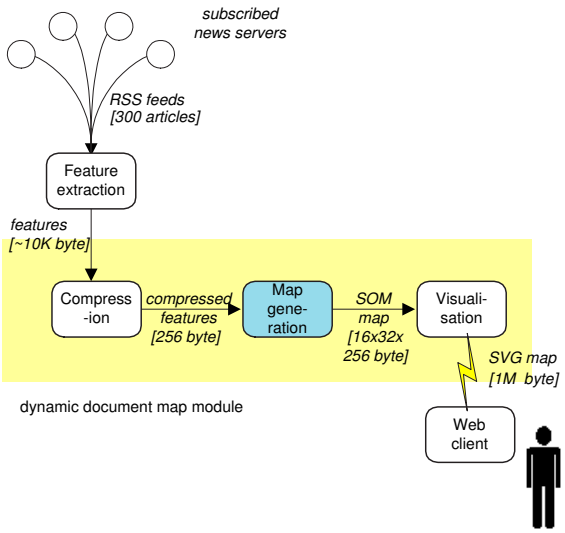


Figure 6.2: The data mining processing pipeline, the amount of data communicated between the modules is indicated. The update rate of the map depends on the update rate of the RSS feeds.

Data mining processing pipeline. The application that we address in this chapter is about giving an overview of a collection of personally subscribed newsgroups. The purpose of the involved data mining system is to transform the personal newsgroup feeds into a personal 2D map. In this way the user will have a quicker overview of the changes in his area of interest. The complete pipeline is described in Figure 6.2.

Feature extraction. In order to cluster newsgroups, all articles have to be expressed in a common notation. As in [95] we use *noun phrases* for this purpose, phrases that can serve as the subject or the object of a verb, and that give a better representation of the content of the article than just single words. These phrases are extracted from the *corpus* (a collection of documents) of newsgroup articles, and for this purpose we used a Natural Language Processing (NLP) tool named Sigmund, a Prolog project developed at the University of Amsterdam [2]. Each article is characterised by a set of noun phrases. To facilitate the comparison of articles we create a *vector space model* of the corpus [102]. First the set of all unique noun phrases for the entire corpus is expressed as a vector

$$\vec{p} = (p_0, p_1, \cdots p_{N-1}), \tag{6.1}$$

where each component p_i is a noun phrase. Next, each article s or *sample* is expressed in a numerical form, by a vector $\vec{x}_s$ of the same size N as $\vec{p}$,

to enable further processing. In the basic vector space model the articles are represented as real vectors in which each component corresponds to the frequency of occurrence of a particular noun phrase in the article, also known as *Term Frequency* (TF). Obviously one should provide the different noun phrases with weights such that its information content corresponds to their significance or power of discrimination. For example, a general word such as 'computer' in a computer science article collection, probably would be less discriminating than a specific word and hence, should receive a relative low weight for the entire collection. For the weighting the *Inverse Document Frequency* (IDF) schemes can be used (IDF is the inverse of the number of articles in which the noun phrase occurs) [102]. To summarise, the feature vector $\vec{x_s}$ describing the article s can be defined by

$$\vec{x_s} = (x_{s_0}, x_{s_1}, \cdots x_{s_{N-1}}), \tag{6.2}$$

where x_i is the weighted frequency of noun phrase i in article s [45]. Experiments show that individual articles, on average, have 10 to 20 unique noun phrases, whereas the whole collection typically has over $N = 10^4$ noun phrases (for ± 500 articles). These feature vectors are very sparse.

Compression. The number of features in a newsgroup collection can become very large, even with a modest number of articles. Without taking measures, the high computation time and storage requirement would prevent the realisation of a real-time system. Since these document spaces are very sparse, simple compression methods suffice and good results for quality as well as performance have been reported [95]. The document space compression reduces the original number of dimensions N typically by a few hundred times to a smaller number of dimensions N_c. As an example, the compressed articles in Figure 6.2 are represented by just $N_c = 256$ bytes. Although this compression is in principle irreversible, heuristics exist to recover information about the significance x_{s_i} of individual noun phrases (that is needed in visualisation [5]).

Map generation. The map module is responsible for creating a theme map in which the articles show topological ordering. A popular algorithm that generates such a map is SOM. This SOM-map is a 2D array of prototype vectors, that organise themselves towards the input samples. This self-organisation is expressed spatially: similar articles cluster together and different articles are positioned at a distance.

The algorithm iteratively compares input samples with these prototype vectors (with same dimension N_c as the samples) and adapts them in a particular way. After all samples have been processed, the process is repeated a fixed number of passes (called *epochs*). The involved four nested loops (dimension $N_c\times$ map dimensions $\times$ samples $\times$ epochs) determine *Map gen-*

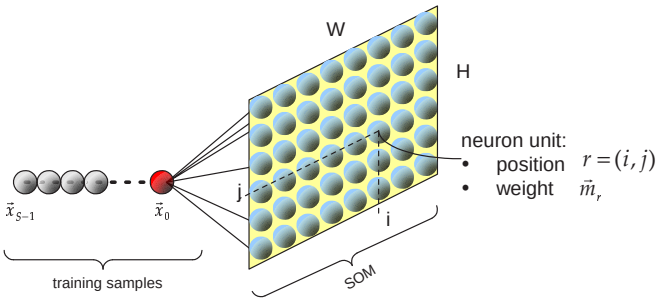


Figure 6.3: SOM network connectivity: the current training sample $\vec{x}_s$ is fully connected to all neuron units $r = (i, j)$ and thus can be compared directly with the associated weight vector $\vec{m}_r$

eration as the most time consuming module in the pipeline. The basic algorithm will be described in Section 6.2.3.

Visualisation. The final module prepares a vector graphics file on a server to allow for remote viewing by a light-weight web client. The used vector graphics format, Scalable Vector Graphics (SVG)[3], allows for operations such as zooming, panning and selection for viewing the article itself. The data for the graphics file is extracted from the neurons by standard techniques [116][117].

SOM basics. The *Self-Organising Map* (SOM) [77] is an artificial neural network model that has shown to be well-suited for mapping high-dimensional data into a two-dimensional representation space. In this paragraph we describe the technical details of the self-organisation process.

The SOM consists of an input layer that offers the training samples in parallel to a number of neuron units in the output layer, organised in a rectangular lattice with dimensions $W \times H$, see Figure 6.3. Every unit $r \in R = \{(0,0), \cdots (W-1, H-1)\}$ in this grid has a position (i, j) and is associated with a weight vector $\vec{m}_r \in \mathbb{R}^N$, or prototype vector, that at the end of the training will represent a cluster of similar articles. The weight vector $\vec{m}_r = (m_{r_0}, m_{r_1} \cdots, m_{r_{N-1}})$ is of the same dimensionality N as the training samples[4]. The weight vectors are hosted by the neuron units, see Figure 6.3. The weight vectors are initialised with random values before the training starts.

For the outline of the training process we will closely follow [34]. The training starts by selecting an training sample $\vec{x_s}$, which represents an article in the corpus.

[3] W3Schools: "Introduction into SVG", 2006, `http://www.w3schools.com/svg/svg{_}intro.asp[Online,accessed12/04/2006]`.

[4] Because of the focus on SOM we neglect the compression $N \rightarrow N_c$ for the moment.

The set of training samples is denoted by X and consists of S articles. Then the unit $r = (i, j)$ with the smallest distance between its associated weight vector $\vec{m}_r$ is selected as the Best Matching Unit (BMU) or *winner*[5]

$$r_w = argmin_r(\|\vec{x}_s - \vec{m}_r\|), \tag{6.3}$$

where the function $argmin_r$ minimises the argument over all unit locations $r \in R$, and where $||.||$ denotes the *Euclidean distance metric*. This distance metric (or 2-norm) of a vector $\vec{d}$ is defined by

$$\|\vec{d}\|_2 = \|(d_0, d_1, \cdots d_{N-1})\|_2 = \sqrt{\sum_{i=0}^{N-1} d_i^2}. \tag{6.4}$$

So the training sample $\vec{x}_s$ is at best represented by unit r_w. To increase the probability for this unit to be chosen as winner the next time the same input is selected again, the unit's weight vector $\vec{m}_r$ is slightly adjusted towards training sample $\vec{x}_s$. This gradual adaptation of the weight vector is controlled by the *learning rate* $\alpha(t) \in (0, 1]$, where t represents the number of training epochs, and it starts with $t = 0$ and ends for $t = T - 1$. The learning rate α is usually a decreasing function over time. Hence, weight vectors will be adapted stronger at the beginning of the training process. A rather low value of $\alpha(t)$ at the end of the training process leads to a fine-tuning phase. The training process resembles the simulated annealing procedure of Chapter 4.

To obtain a topological ordering of the map not only the weight vector of the winner r_w is adapted, but also the weight vectors of the units in its vicinity. As a result training samples similar to $\vec{x}_s$ are more likely to be represented in the region of the SOM where the winner is located. The *adaptation strength* $h_{r,r_w}(t)$ of neighbouring units is determined by their distance from the winning unit r_w on the map. This so called *neighbourhood function*, is also a decreasing function over time, and usually based on a Gaussian function

$$h_{r,r_w}(t) = e^{\frac{||r - r_w||^2}{2\sigma^2(t)}}, \tag{6.5}$$

where $r_w \in \mathbb{R}^2$ is the location of the winning unit on the lattice, r the location of a neighbouring unit, and *neighbourhood size* parameter $\sigma(t)$ controls the radius of effected neighbouring units. The adaptation strength for all neuron units r can conveniently be represented by the neighbourhood matrix

$$\Lambda_{r,r_w}(t) = [\![h_{r,r_w}(t)]\!]_{r \in R}.$$

It can be seen from (6.5) that units closer to the winner are adapted more than units that are farther away. A high value of $h_{r,r_w}(t)$ at the beginning of the

[5] If the samples $\vec{x}_s(t)$ are stochastic and have a continuous density function, the probability for having multiple minima in (6.3) is 0 [45]. With discrete-valued variables, however, multiple minima may occur; in such a case one of them is selected at random for the winner.

training process (small epoch number) leads to a global organisation of the weight vectors, that is, neighbouring units have similar weight vectors. By gradually decreasing the neighbourhood function during increasing epochs, the adaptations become more local.

Finally, we specify the adaptation of the neuron weights expressed in the already defined mathematical concepts. The weight vector $\vec{m}_r(t+1)$ of unit r is adapted by adding a portion $\alpha(t) \cdot h_{r,r_w}(t)$ of the vector difference $(\vec{x}_s - \vec{m}_r(t))$ to $\vec{m}_r(t)$, resulting in:

$$\vec{m}_r(t+1) = \vec{m}_r(t) + \alpha(t) \cdot h_{r,r_w}(t) \cdot \big(\vec{x}_s - \vec{m}_r(t)\big), \tag{6.6}$$

As a consequence of (6.6) the weight vector of the winner and the weight vector of the units in its vicinity are 'moved' towards the training sample. Hence, it is more likely that similar samples are mapped into this part of the map in successive training epochs. This actually gives SOM its auto-clustering ability.

Typical values for the parameters in the dynamic document mapping domain – for a small number of samples ($S < 500$) – are: $N_c = 315, W \times H = 32 \times 32$, and the amount of epochs 250. However, for larger document spaces, the number of units ($W \times H$) can exceed 1 million and can need more than 10,000 epochs of training.

6.2.3 SOM training

Neural network training, in general, takes a long time to compute. Since we expect that training is dominant in the performance of our system we now focus on the SOM training.

6.2.3.1 Algorithm

For this research we restricted ourselves to the SOM training because this is the most computationally intensive part. In the preamble to the specification of the algorithm we structure the training process, described in the previous section, in five steps. These five steps will compute the update for all neuron units r for a given sample $\vec{x}_s$ within an epoch:

forall r do

Step 1: determine the high dimensional distance of the sample $\vec{x}_s$ to all weight vectors $\vec{m}_r(t)$ of neurons r: $\delta_r = \|\vec{x}_s - \vec{m}_r(t))\|$

Step 2: determine the winning unit location: $r_w(t) = argmin_r(\delta_r)$, see (6.3)

Step 3: compute the 2D distance matrix of all units to the winning unit: $d_r = \|r - r_w(t)\|$

Step 4: compute the neighbourhood matrix: $h_{r,r_w}(t) = e^{\frac{d_r^2}{2\sigma^2(t)}}$, see (6.5)

Step 5: compute the update for the neurons: $\vec{m}_r(t+1) = \vec{m}_r(t) + \alpha(t) \cdot h_{r,r_w}(t) \cdot \big(\vec{x}_s - \vec{m}_r(t)\big)$, see (6.6)

The algorithm of the training process is given below (Algorithm 6.1), and is taken almost literally from [77]. In Section 6.3 an abstract model is presented of a part of the algorithm that provides the necessary freedom for the mapping to a massively parallel hardware architecture. The parameter settings for S, N, N_c, T, W, and H are application dependent and will be dicussed in the following sections.

Algorithm 6.1 SOM training algorithm

1: $\vec{m}_r \leftarrow$ random values taken from $(0, 1)$;
2: initialise α_T, σ_T;
3: $\alpha \leftarrow 1; f_\alpha \leftarrow \sqrt[T]{\alpha_T/\alpha_0}$;
4: $\sigma_0 \leftarrow max(W, H)/2; f_\sigma \leftarrow \sqrt[T]{\sigma_T/\sigma_0}$;
5: **for** $t \leftarrow 1 \cdots T$ **do** {for each epoch do}
6: **for** $s \leftarrow 0 \cdots S-1$ **do** {for each sample do}
7: **for** $r \leftarrow (0,0) \cdots (W-1, H-1)$ **do** {all neurons in parallel}
8: $\delta_r \leftarrow \|\vec{x}_s - \vec{m}_r\|_2$;
9: **end for**
10: $r_w \leftarrow argmin_r(\delta_r)$;
11: **for** $r \leftarrow (0,0) \cdots (W-1, H-1)$ **do** {all neurons in parallel}
12: $d_{r,r_w} \leftarrow \|r - r_w\|_2$;
13: $h_{r,r_w} \leftarrow exp(d^2_{r,r_w}/2\sigma^2)$;
14: $\vec{m}_r \leftarrow \vec{m}_r + \alpha \cdot h_{r,r_w} \cdot (\vec{x}_s - \vec{m}_r)$;
15: **end for**
16: **end for**
17: $\alpha \leftarrow \alpha * f_\alpha$;
18: $\sigma \leftarrow \sigma * f_\sigma$;
19: **end for**

To demonstrate the working of the SOM training a simple example is included, see Figure 6.4. In this example, the feature space is one-dimensional $N = 1$, the map is only ($W = 4 \times H = 4$) and initialised with values in range [0,255], and the current training sample is $\vec{x}_s = 198$. For convenience the learning rate is set to unity ($\alpha = 1$), and we take σ such that the neighbourhood attenuation $e^{-\frac{-d}{2\sigma^2}}$ satifies $e^{-\frac{-d}{2\sigma^2}} = 2^{-d}$.

6.2.3.2 Hardware mappings

SOM training is in general a computationally intensive step in data mining applications [77][36]. That is the reason why many hardware mappings for SOM have been described since its conception in 1982. Because of its inherent parallel structure also parallel implementations have been made. The most advanced ones have been written for SIMD architectures such as CNAPS, Hypercube, Connection Machine and MasPar, which, however, are expensive, voluminous and have extremely high power consumption [92][77][103]. Also other, more embed-

Initial neuron values $\vec{m}_r(t) =$

123	46	84	19
0	19	200	96
250	23	190	55
97	60	85	143

The values of the weight vectors of the 4×4 neuron map are given. A new sample, for example $\vec{x}_s$=198 (not shown), enters the training process.

Step 1,2 Distance matrix with winning unit $\delta_r =$

75	152	114	179
198	179	**2**	102
52	175	8	143
101	138	113	55

The high dimensional distance matrix $\delta_r = \|\vec{x}_s - \vec{m}_r\|$ is determined. The winning unit r_w (with smallest value=2) is indicated.

Step 3 2D distance $d_r =$

3	2	1	2
2	1	**0**	1
3	2	1	2
4	3	2	3

In order to construct the neighbourhood matrix (see next step: *neighbourhood*) first the 2D distance $d_{r,r_w} = \|r - r_w\|$ between the winning unit and all other units has to be determined. In this example the manhattan distance is used (1-norm). This norm can be written as $\|\Delta x, \Delta y\|_1 = |\Delta x| + |\Delta y|$ (6.10).

Step 4 Neighbourhood $h_{r,r_w}(t) =$

0	0.25	0.5	0.25
0.25	0.5	**1**	0.5
0	0.25	0.5	0.25
0	0	0.25	0

The neighbourhood matrix $\Lambda_{r,r_w}(t)$ determines how much the neighbourhood units of the winning unit will be adapted (besides the winning unit itself). To keep the example simple we assume matrix values of the form $2^{-d_{r,r_w}}$ for this epoch. Note that the neighbourhood distribution is dynamic over time.

Step 5 Update $\vec{m}_r(t+1) =$

123	84	141	63.75
49.5	108.5	**198**	147
250	66.75	194	90.75
97	60	113.25	143

The update for the weight vectors is computed according to (6.6). For the winning unit this is $\vec{m}_{r_w} = 200 + 1 \cdot 1 \cdot (198 - 200) = 198$, so adapting 100% to the sample. For the unit just below the winning unit this becomes $\vec{m}_r = 190 + 1 \cdot 0.5 \cdot (198 - 190) = 194$, and also adapts a bit towards the value of the winning unit.

Figure 6.4: Example of a simple SOM training (high dimensionality N=1) for a single epoch and for one training sample

ded parallel solutions have been devised for the Transputer [124] or an FPGA [97]. FPGA technology, however, exhibits rather long development cycles. A relative fast development is supported by a general purpose processor with special SIMD extensions [51], but is too costly to be a serious contender for embedded applications. Since the system should execute fast, should be compact and should not consume too much power it is decided to select an embedded processor. The Aspex's Linedancer fits the massive but simple processing required for neural network processing well. In order to map the SOM algorithm on the Linedancer in a performance optimal way the following observations from literature are relevant. It is shown in [97] and [77] that SOM is robust in the sense that it is somewhat flexible to:

1. lower precision, for instance exchanging a float by an 8 bit fixed point representation,

2. different distance metric, for instance the 1-norm (Manhattan distance), see Section 6.4.2.1 instead of the more common but computationally intensive 2-norm, and

3. choice of neighbouring function, for instance replacing the Gaussian neighbourhood function by a box function (e.g., see Figure 6.9).

In Section 6.4 we will elaborate on these issues.

6.2.4 Feasibility

The purpose of this section is to get confident about the scope of the system and the technical feasibility of realising the system with the target hardware architecture. In this way early roadblocks can be identified, and evaluated and their impact can be assessed in an early stage.

Demarcation of the system boundary. The dataflow diagram of the entire data mining system is described by Figure 6.2. From literature [77] we know the SOM training is computationally intensive. In contrast to the other modules, the SOM training is not only linear dependent on the number of training samples. Besides the number of training samples S, SOM training also depends (linearly) on the number of training epochs T, the dimensions of the map $W \times H$, and the dimensions of the high dimensional space N_c. This determines the SOM training as the most demanding module, having a complexity of

$$O\left(S \cdot T \cdot W \cdot H \cdot N_c\right).$$

For this reason the training module is selected for further investigation and constitutes the scope of this study.

Feasibility: a first estimate. Based on Algorithm 6.1 we can provide a first estimate of the timing aspects of the SOM training implementation. We will estimate the execution time on basis of a naive parallelisation of the sequential model: the total time becomes the sequential time divided by the number of processors.

We initially estimate values for the map dimensions: width $W = 32$, height $H = 16$ and the dimension of the compressed space $N_c = 256$. We start with a timing estimate of the inner loop, see lines $8, 10, 12 \cdots 14$ in Algorithm 6.1. After some initial experiments we estimate the number of clock cycles per operation. Lines 8 and 14 of the algorithm each take 3 operations per weight vector component (a subtraction, multiplication and a division), totaling to $W \cdot H \cdot N_c \cdot 3 = 32 \cdot 16 \cdot 256 \cdot 3 \approx 400K$ operations, see Table 6.1. The 2D distance calculated in line 12 consumes a little more per neuron (5 cycles) since this distance involves two components (in width and in height). This table not only summarises the sequential complexity but also includes concrete operation counts (in cycles per epoch per sample). The workload for a single sample and one epoch adds to approximately $C_{se} \approx 800K$ operations. For the average training session, we es-

algoritm line no	training step	sequential complexity order of operations	cycles per operation	number of operations
8	1. Distance in highD	$O(W \cdot H \cdot N_c)$	3	400K
10	2. Winner selection	$O(W \cdot H)$	1	700
12	3. Distance in 2D	$O(W \cdot H)$	5	2500
13	4. Determine neighbourhood	$O(W \cdot H)$	1	500
14	5. Update neurons	$O(W \cdot H \cdot N_c)$	3	400K
Total number of operations C_{se}				800K

Table 6.1: Base complexity, for comparison purposes and projected gain by parallelisation. The values in the last column contain estimates for a single sample per epoch with $W = 16$, $H = 32$, and $N_c = 256$, and are indicative for the performance.

timated that the number of samples S and the number of training epochs T are 300 and 250 respectively. For our experiments we use a dual Linedancer system with $2 \times 4K = 8K$ PEs. This system could potentially perform the training job in $\frac{S \cdot T \cdot C_{se}}{\#PEs} = 12$ M operations, *provided* that such a parallel scheme can be found. For a 300 MHz dual Linedancer system, and an average of ± 100 cycles per operation (single bit architecture), this would correspond to 4.1 sec processing time.

In the second column cycles are expressed in (big O) order notation. Conversion to concrete numbers of operations is straightforward; the distance computations (in lines 8 and 12), however, have to account for the subtraction, the

absolute value (for the 1-norm) and finally its accumulation over all components.

We conclude the feasibility study with the parallel order of complexity $O_{//}$,

$$O_{//} = O\left(\frac{S \cdot T \cdot W \cdot H \cdot N_c}{\#PEs}\right),$$

in which S is the number of samples, T the number of epochs, W and H are the width and the height of the map, and N_c is the dimension of the compressed document space. It is a challenge to determine how the work – symbolised by the numerator of this fraction – can be distributed over all PEs evenly.

6.3 Incremental prototyping

This section describes the incremental prototyping phase of the SOM training algorithm as well as some implementation independent choices. These choices involve a quality measure (for supporting the functional decomposition and evaluating implementation alternatives) and the choice of the training parameters. We will follow the evolutionary development methodology – as proposed in Chapter 3 – closely. In this section the *Incremental Prototyping* template (Section 3.7) will be taken as a guide.

6.3.1 The training algorithm

The SOM training algorithm (Algorithm 6.1) is taken as the functional specification. The specification in a functional language is derived in the following three steps: parameter estimations, intermediate mathematical model, and finally the functional specification of the SOM training.

Parameter estimations. From literature and preliminary experiments the following parameters were estimated:

1. the number of articles $S \leq 500$ (the number of visualised articles should not be too large),
2. the dimension of the (compressed) vector space $N_c = 315$ [5],
3. the map dimensions $W \geq 16$, and $H \geq 16$ (for good visualisation the map should be large enough to host at maximum 2 samples/neuron),
4. the number of training epochs $T = 250$,
5. the learning rate α varies from 1 to 10^{-4} and is reduced by $f_\alpha = \sqrt[T]{\frac{1}{10^{-4}}} \approx 0.964$ (see [101] for the adaptation scheme),

6. the neighbourhood size σ starts with half of the maximum size of the map, and for for example $W = H = 16$ this is 16/2=8. This is reduced by $f_\sigma = \sqrt[T]{\frac{1}{8}} \approx 0.992$ per epoch for $T = 250$.

Intermediate mathematical model. The original algorithm Algorithm 6.1 is transformed into an abstract mathematical model before it is transcribed in functional code. First we introduce some relevant definitions.
The function *argmin* (relates to (6.3)) determines the minimum value of a list with respect to a particular function value:

$$argmin\ f\ xs\ =\ (\ snd \cdot min \cdot zip\ )\ (\ (\ map\ \ f\ xs\ )\ ,\ xs\ )$$

The function *dist* computes the Euclidean distance between two vectors:

$$dist\ (\ \vec{x}\ ,\ \vec{y}\ )\ =\ \|\ \vec{x}\ -\ \vec{y}\ \|_2$$

The function δ determines the 'distance' between s and r based on the Euclidische distance between the vectors x_s and m_r:

$$\delta\ s\ r\ =\ dist\ (\ \vec{x_s}\ ,\ \vec{m_r}\ )$$

Before the intermediate model is given, first a few remarks are made:

- m_r represents a single N_c-dimensional weight vector,
- x_s represents a single N_c-dimensional sample, representing a newsgroup article,
- in the identifiers of the variables *xs* is the multiple of x, so it is a list of x-es (idem with *ms*, *rs*),
- in Algorithm 6.1, lines 12-14, the for-loop of r is formulated as a list comprehension, to better demarcate the scope of the various for-loops. This list comprehension generates a complete list of new m_r-values,
- in Algorithm 6.1, lines 7-10, are fomulated by a single line $r_w := argmin\ (\delta\ s)\ rs$,
- list comprehension uses $\leftarrow$; the same arrow is used in the variable assignment with the for-loops. However, for the assignment the copula := is used,

The intermediate model is depicted below:

$$
\begin{array}{l}
ms := init; \\
\alpha_T := init; \\
\sigma_T := init; \\
\alpha := 1; \qquad\qquad\qquad\qquad\qquad\qquad f_\alpha := \sqrt[T]{\frac{\alpha_T}{\alpha}} \\
\sigma := max(W,H)/2; \qquad\qquad\qquad f_\sigma := \sqrt[T]{\frac{\sigma_T}{\sigma}} \\
\textbf{for} \quad t \leftarrow [\,0 \ldots T{-}1\,]; \\
\qquad \textbf{for} \quad s \leftarrow [\,0 \ldots S{-}1\,]; \\
\qquad\qquad r_w := argmin\ (\delta\ s)\ rs; \\
\qquad\qquad ms := [\,m_r + \alpha.h_{r,r_w}.(x_s - m_r) \qquad\qquad | \ r \leftarrow rs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ;\ d_{r,r_w} := dist(r, r_w) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ;\ h_{r,r_w} := e^{d^2_{r,w}/2\sigma^2} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad]; \\
\qquad \alpha := f_\alpha.\alpha; \\
\qquad \sigma := f_\sigma.\sigma;
\end{array}
$$

Functional specification of SOM training. As described in Section 6.2.3 SOM training consists of successive rounds of adaptations (called epochs) of a rectangular array of neuron weight vectors `ms` (the set $\{\vec{m}_{0,0}, \vec{m}_{0,1}, \cdots \vec{m}_{W-1,H-1}\}$) by the set of samples `xs` (the set $\{\vec{x}_0, \vec{x}_1, \cdots \vec{x}_{S-1}\}$).

Before the descriptions in functional code are given, first a few remarks are made:

- the neurons are identified by `rs = [ [0,0],[0,1],... [H-1,W-1] ]`,
- for component-wise vector-vector operations as multiplication, subtraction and addition the following operators are used: `*^`, `-^`, and `+^` respectively,
- for convenience the scalar vector multiplication operation (for example the multiplication of the learning rate α) also uses `*^`,
- `norm2` computes the Euclidean norm.

The descriptions are given in a specific order: first the inner loop, then the next enclosing one and so on. The adaptation of all neurons `ms` by a single sample is described by:

```
Fsample ms s = [ ms!r +^ alpha *^ h *^ ( xs!s -^ ms!r )
               | r <- rs
               ; h = e^( dist(r,rw)^2 / sigma^2 )
               ]
 where
    delta    = \(s,r)   -> dist( xs!s, ms!r )
    dist     = \(v1,v2) -> norm2( v1 -^ v2 )
    rw       = argmin (delta s) rs
```

Within a single training epoch $t \in [0 \cdots T-1]$ all samples $\vec{x}_{0 \cdots S-1}$ (`xs`) have to be processed. This also involves the adaptation of the learning rate α (`alpha`) and neighbourhood size σ (`sigma`). The specification of a single epoch is given by:

```
update ms                 = fold Fsample ms xs
Fepoch (alpha, sigma, ms) = (f_alpha*alpha, f_sigma*sigma, update ms)
```

The entire training involves the iteration of `Fepoch` on all epochs `T`, where `T` is a constant specified at before hand. The higher order function `iter` below, takes a function `f`, a tuple `x`, an iteration count `n` as argument and simply applies `f` to `x` `n` times. Finally the training can be described by:

```
Ftrain (alpha, sigma, ms) = iter Fepoch (alpha, sigma, ms) T
where
    iter f x 0     = x
    iter f x (n+1) = iter f (f x) n
```

6.3.2 Quality functions

Now that the functional behaviour is in principle determined we can turn to the extra-functional constraints such as quality and performance. Performance can be specified and measured in an unambiguous manner, but how about quality? Quality is a complex (non-functional) property to measure because it has objective (physical) aspects as well as subjective aspects. In this study we restrict ourselves to objective quality measures that are used for comparison purposes only. The grounding or calibration of these measures is done via user experiments. We choose the following two measures [65]:

- The *Quantisation Error* (QE) measures how good the generalisation quality of a neural network is. This quality in fact tests how well the trained neural network has generalised from its input data. The smaller this value, the better the training. It is defined by the average distance of each sample $\vec{x}_s$ with its best matching unit (BMU). This distance can be written as $min_r \|\vec{m}_r - \vec{x}_s\|_n$, where n represents the norm. So, the average quantisation error QE taken over all samples is defined by

$$QE_n = \frac{\sum_{i=0}^{S-1} min_r \|\vec{m}_r - \vec{x}_s\|_n}{S}. \tag{6.7}$$

- The *Topology Error* (TE) is used to evaluate the topological quality and is in particular for the SOM a useful measure. It determines how often the BMU $r_w(s)$ of a sample s is not a neighbour of the *second best winning unit* $r_{2w}(s)$ of the same sample, where $r_{2w}(s) = argmin_{r'}(\|\vec{x}_s - \vec{m}_{r'}\|)$ minimises

the argument over all unit locations except for the winning neuron itself $r' \in R \backslash \{r_w\}$. This is an indication of how much the map is distorted at this location. The larger this number, the worse the topological quality is. Analogous to QE a topology error [65] is defined by:

$$\frac{\sum_{i=0}^{S-1} \begin{cases} 1, & \text{if } \|r_w(s) - r_{2w}(s)\|_n > 1, \\ 0, & \text{otherwise} \end{cases}}{S} \tag{6.8}$$

where n is the norm used.

The disadvantage of this measure is the indifference of small and large distances between the two locations. Therefore, the following measure is defined:

$$TE_n = \frac{\sum_{i=0}^{S-1} \|r_w(s) - r_{2w}(s)\|_n - 1}{S}, \tag{6.9}$$

which is more sensitive to larger topology errors.

The quality measures are computed for a training set as well as a representative test set. Upper bounds on these values, that correspond to sufficient perceptual quality, are not determined. However, in Section 6.4.2 we establish values for QE and TE that give reasonable quality.

6.3.3 Running experiments

In order to come up with a set of representatieve training samples, a model has been constructed of the input space, that is, the compressed article space. After inspection of the distribution of several component values of vectors $\vec{x}_s$, representing articles from a few different article collections, we modelled a mixed Gaussian distribution [36] for the generation of samples $\vec{x}_s$ in the compressed space. For the experiments that follow in the development phase, the same two representative article sets are used: one to train the SOM and the other to test (measure) the quality of the training. This is a standard procedure in the neural network domain to avoid *overtraining*[6] [62].

The results of the training as well as its quality test is presented in a standard format, that is, four vertically positioned graphs (see for an example Figure 6.5), where the quality measures are computed per epoch. The subfigures (c) and (d) in Figure 6.5 show the QE and TE respectively of the training set. For their computation Equations (6.7) and (6.9) are used and the samples $\vec{x}_s$ are taken from the training set. The subfigures (a) and (b) show the QE and TE respectively of the test set: for their computation (6.7) and (6.9) are used and the samples $\vec{x}_s$ are now taken from the test set. It can be observed from subfigures (c) and (d), that both QE and TE improve with more epochs. Furthermore it can be noticed

[6] Overtraining signals a fault condition in neural network training. In this situation the network "wears" in on the training set and does not generalise anymore from its input.

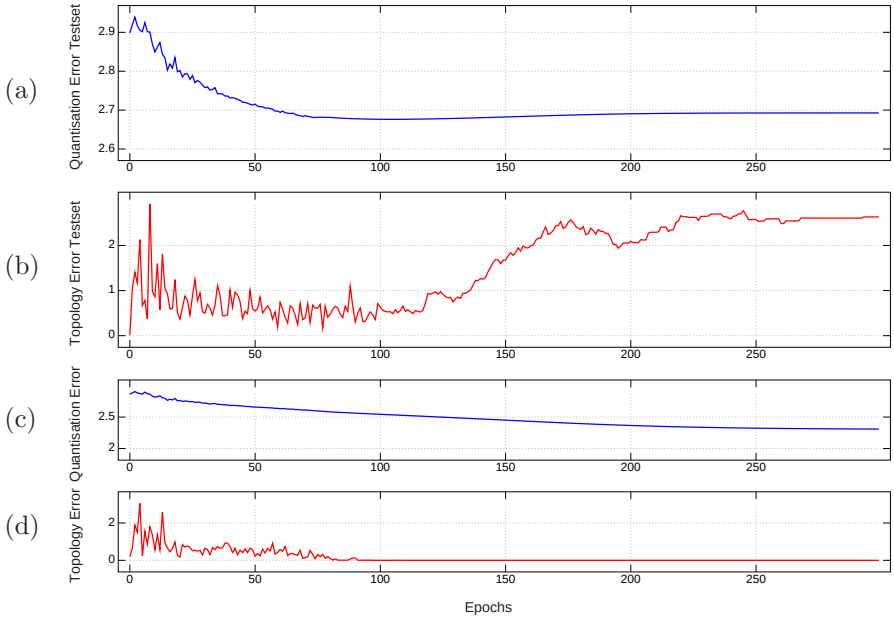


Figure 6.5: Two quality measures: Quantisation Error and the Topology Error

that the response of the test set, subfigures (a) and (b), return higher values of QE and TE than subfigure (c) and (d) because the test set was not used to train the SOM network. The most interesting phenomenon, however, is the exposure of overtraining in subfigure (b) (starting at epoch number $\pm$ 130) and – a bit less visible – in (a). So at this point the network generalises best for epochs between $100 - 160$. For our purpose the absolute values of QE and TE are not important, but the level of degradation or improvement is.

6.4 Transformational development

As mentioned before, we will map the SOM training scheme on the Linedancer to check our hypotheses regarding quality, performance, and development effort (methodology). During implementation several concerns – some more case specific than others – have to be considered. A number of them: storage allocation, dimension of feature space (N_c), the distance norms for the high as well as low dimensional space (1-norm, 2-norm, ∞-norm), approximation of the Gaussian neighbourhood function ($h_{r,r_w}(t)$) and bit-width of variables (accuracy), are described below. They all can potentially compromise the quality because they can trade quality for performance. These concerns require extensive exploration of the design space in order to come up with satisfactory results. The majority of the following sections utilise the quantitative quality measures QE and TE. In this section the *Transformational Development* template, as given by Section 3.8, will be followed.

6.4.1 Global system considerations

In this section a first analysis is made on the timing and storage design space given the problem requirements and available hardware architecture. Finally, relevant general system architecture issues are studied because they can also influence the implementation process. Because the timing analysis depends on the storage allocation, this will be addressed first.

Global storage allocation. The limited memory available per PE poses restrictions on the mapping of functionality and its associated data-structures. Therefore, an analysis of the storage allocation of the SOM training algorithm is conducted, see also [31]. The mapping related subsystems of the Linedancer are listed in Table 6.2 by the last four columns. The format of these columns corresponds with that given in Section 4.4.1.

[7] $\vec{r}_w$ is a 2D vector, and it is replicated during the training process to all neurons for computational efficiency of training step 3 and line 12 of Algorithm 6.1.

[8] For each epoch the 2D locations of the neurons in the grid are reloaded because of limited memory resources overhead and only takes little overhead ($1K/25K9 \approx 4\%$).

SOM Training Algorithm

				Linedancer subsystems			
item	variable	dimension	dependency $[N_c, \mathcal{N}, none]$	SPARC SDS	PE CAM or EXT	Associative CAM-SDS	DMA SDS-PDS
weight vector	$\vec{m}_r$	N_c	$N_c, \mathcal{N}$		$\surd$	$\surd$	
training sample	$\vec{x}_s$	N_c	N_c	$\surd$	$\surd$		
high dimensional distance	δ_r	1	$\mathcal{N}$		$\surd$		
location of	r_w	2	$\mathcal{N}^7$		$\surd$		
(winning) unit	$r = (i, j)$	2	$\mathcal{N}$		$\surd$		$\surd^8$
2D distance	d_{r,r_w}	1	$\mathcal{N}$		$\surd$		
neighbourhood matrix	h_{r,r_w}	1	$\mathcal{N}$		$\surd$		
learning rate, learning rate factor	α	1		$\surd$			
	f_α	1		$\surd$			
neighbourhood size and factor	σ	1		$\surd$			
	f_σ	1		$\surd$			
epoch number	t	1		$\surd$			
sample number	s	1		$\surd$			
loading of sample $\vec{x}_s$				$\surd$			$\surd$

Table 6.2: Mapping of the various variables of the SOM training algorithm on the memory and processing subsystems of the Linedancer

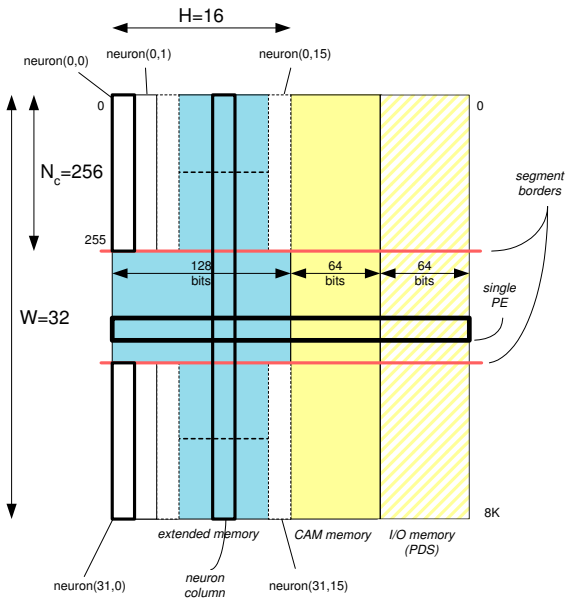


Figure 6.6: Vertical arrangement of neurons over PEs in extended memory (where as an example N_c=256), segment borders are indicated

The various variables in the algorithm are listed in the two left most columns. The third column indicates the dimension of the variables and the fourth column indicates inter-dependencies between variables and training parameters. For example, the weight vectors are related to the size of the high dimensional space (N_c) and the size of the neuron array ($\mathcal{N} = \{(0 \cdots W-1), (0 \cdots H-1)\}$). The list of these explicit inter-dependencies is useful in the final mapping to the memory subsystem of the Linedancer.

For a dual Linedancer system we have an 8K PE budget. Every PE is equipped with 128 bit EXT and 64 bit CAM, see Section 2.3.4. Because the computations concentrate around the neurons and because the size of the neuron map is fixed – as opposed to the number of samples – we have chosen to store the neurons in the array and the training samples in off-chip DRAM. To maximise the efficiency of computation, the N_c components of the neurons as well as the samples, are mapped one-to-one to N_c PEs. This corresponds to the vertical orientation of vectors as described in Figure 6.6. One set of the W=32 vertically organised neurons is called a neuron column.

Global timing analyses. For the above described storage allocation the following table is derived for the performance of the system. In this paragraph and in Table 6.3 we assume $N_c = 256$ and $H = 16$ (since all PEs work on W neurons in

parallel, W is omitted from the complexity estimation). The third column shows

algoritm line-no	training step	projected order of parallel operations	cycles per column	number of columns	total
8	1. Distance in highD	$O(H + log_2 N_c)$	2K2	16	35K2
10	2. Winner selection	constant	1K5	1	1K5
12	3. Distance in 2D	constant	62	1	62
13	4. Determine neighbourhood	$O(H)$	10	16	160
14	5. Update neurons	$O(H)$	186	16	3K0
total number of cycles C_{LD}					39K9

Table 6.3: Improved estimate of the training performance

the projected parallel complexity for a particular parallel architecture, which is parallel in $W \times N_c$ but sequential in H (see Figure 6.6). The additional $O(log_2 N_c)$ in training step 1 accounts for the time to compute a binary adding tree in parallel (for the 1-norm computation). The computation of a 1-norm of a vector starts with a component-wise subtraction of a vector, followed by an absolute value function. Subsequently the binary adding tree of height 8 is computed, and this takes 2K cycles. In particular the involved data communication near the root of the tree consumes much time. Moreover, since steps 1, 4 and 5 are dependent on the number of neuron columns H (see Figure 6.6), their contribution to the total number of cycles is dominant. As shown in Table 6.3 the total number of cycles $C_{LD} = 39K9$. The average job of $S = 300$ samples and $T = 250$ epochs needs $S \cdot T \cdot C_{LD} = 300 \cdot 250 \cdot 39K9 = 2.99$ G clocks and corresponds to 9.9 sec on a 300 MHz Linedancer. This time is too long for an interactive system. Step 1, the high dimensional distance computation, needs to be optimised because it dominates the training time significantly.

Scalability. Before conducting the mapping we should dedicate some attention to desirable design properties such as scalability. The data mining pipeline should be scalable in the dimensions of the map $W \times H$, the dimension of the compressed document space N_c, as well as the used precision.

Scalability is not limited to the current sizes of the map. Larger sizes of the map, for instance doubling both dimensions to $(2{\cdot}H){\times}(2{\cdot}W)$, can be accomodated by a $H \times (4 \cdot W)$ organisation of 4 times as many Linedancers.

Increasing the dimension of the high dimensional space to for instance $2 \cdot N_c$ can be realised by doubling the number of Linedancers. However, scalability in time is impeded by the computation of the adding tree in step 1, causing an extra

penalty of $O(log_2(2 \cdot N_c))$ cycles. This is a serious point of attention and will be covered in Section 6.4.2.3.

Increasing the precision of 8 to 16 bit (as described in [97]) can be done with marginal effort, but then the I/O memory (PDS) has to be used to supplement CAM as temporary storage. By doubling the number of Linedancers the implied decrease in performance can be repaired.

6.4.2 Trade-off subphase

The purpose of the Trade-off subphase is to absorb all concessions on the functional behaviour implied by limitations of the hardware. In the following subsections, four examples are given: dimension of the compressed feature space (Section 6.4.2.2), choice of the norm in vector distance comparisons (Section 6.4.2.3), simplification of the Gaussian neighbourhood function (Section 6.4.2.4) and finally, the accuracy of the various variables (Section 6.4.2.5). The order of addressing these trade-off issues has to be determined. We used (minimisation of) simulation time as the criterion to select the order, and since the dimension model takes most time, we first lockdown the dimensionality N_c of the compressed space. In this way all subsequent model simulations profit the most. Before starting with the dimension model we introduce some non-trivial vector norms.

6.4.2.1 Vector norms

Because the norm of a vector plays an important role in the mapping of a SOM training to a restrictive hardware architecture, it is important to analyse. We code the norm n of vector as a subscript like in $\|.\|_n$. The Euclidean distance metric ($n = 2$) is a standard used metric in neural network training. Other metrics, however, such as Manhattan or city-block distance ($n = 1$), and the maximum-norm ($n \to \infty$) are more amenable to hardware implementation. The 1-norm of a vector $\vec{d}$ is defined by

$$\|\vec{d}\|_1 = \|(d_0, d_1, \cdots d_{N-1})\|_1 = \sum_{i=0}^{N-1} |d_i|, \tag{6.10}$$

and does not need a square and square root[9] operations (unlike for the 2-norm), which are expensive operations on such hardware architectures. The maximum-norm is simple to compute by taking the maximum absolute component of a vector,

$$\|\vec{d}\|_\infty = \|(d_0, d_1, \cdots d_{N-1})\|_\infty = max_i(|d_i|), \tag{6.11}$$

where i is taken over all components $0 \cdots N-1$ of the vector. For convenience we will abbreviate the maximum-norm by ∞-norm from now on.

[9] For the computation of the winning neuron the square root operation can be omitted.

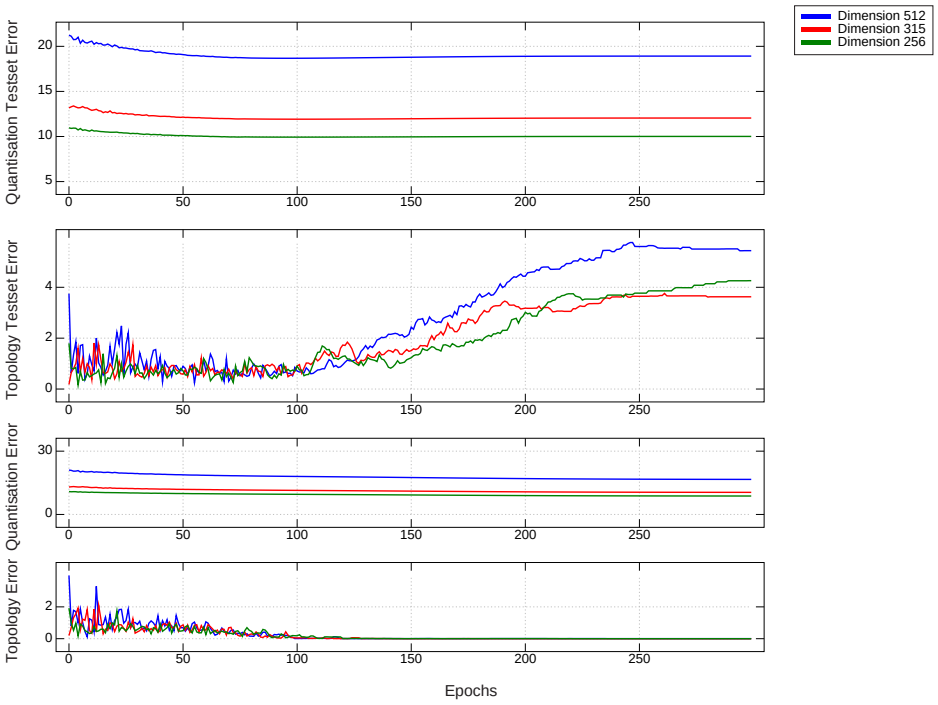


Figure 6.7: The choice of the size of the high dimensional space, the dimensionality N_c. Candidates are: 512, 315, or 256.

6.4.2.2 Dimension of the Feature space

The goal of this subsection is to establish the dimensionality (N_c) of the high dimensional space for the SOM trainer and as such also the output of the compression module, see Figure 6.2. In [4] it is motivated that 315 is a good compressed space dimension. However, for implementation reasons we would like to use a power of two because that fits the Linedancer array better. So the candidates for the dimensionality to investigate are 256 and 512, which will be compared to the value suggested by literature ($N_c = 315$). We do not pursue a small value of 128 because the advised values in literature are ≥ 315 [78][46][5].

A problem to solve is the definition of a good measure of the performance of these candidates. The difficulty now is, at this early moment in design time, that several other relevant parameters are not fixed yet. These parameters are:

- the number of epochs for the training,
- the norms used in the computation of the SOM training algorithm itself (in lines 8 and 12 in Figure 6.1), and

N_c	QE_{avg}	QE_{avg} / N_c
512	19	0.037
315	12	0.038
256	10	0.039

Table 6.4: Relation between the size of the feature space and the average quantisation error

- the norms used in the computation of the quality functions QE and TE.

A preliminary experiment is performed to find a first estimate for sufficiently good values of the mentioned parameters. As a result the Euclidean norm was used (as in [77]) for the distance computations in the training algorithm; in a later experiment this choice will be reconsidered.

For the moment the norm for the SOM training is not decided on yet; this choice is the subject of the next section. However, since we need a norm for the current dimension model, we selected the average of the three norms in order to avoid a bias to a particular norm. When the choice of the norms for training is fixed (see Section 6.4.2.3) we will reconsider the choice for the norms used in the computation of the quality measures QE and TE. So for now the topology error TE is computed by $TE_{avg} = \frac{1}{3} \cdot (TE_2 + TE_1 + TE_\infty)$. The average quantisation error QE_{avg} is computed by $QE_{avg} = \frac{1}{3} \cdot (QE_2 + QE_1 + QE_\infty)$.

The end result of the experiment, where the three candidate dimensions (256, 315, 512) for the dimensionality N_c are tested, is depicted in Figure 6.7. The quantisation error QE_{avg} is dominated by the 1-norm component which is linear in N_c, as confirmed by Table 6.4, which is derived from Figure 6.7. A better indication for quality is given by $\frac{QE_{avg}}{N_c}$. To have optimal behaviour and to avoid overtraining (in particular visible in the topology error on the test set in Figure 6.7), the number of epochs should be in the range 80-140 ($TE < 2$). In this range the difference between the three dimensions is neglectable. The conclusion is that for our application a value of 256 for the high dimensionality N_c is sufficient.

6.4.2.3 Choice for Norms

Now that the size of the high dimensional space is fixed, a next investigation concerns the norms for the high dimensional distance comparison as well as that of the low dimensional distance (2D). In the specification of the SOM algorithm the Euclidean norm is used, see the distance computations in lines 8 and 12 of Algorithm 6.1. The purpose of this subsection is to explore the design space for computationally "cheaper" alternatives such as the 1-norm or the ∞-norm.

Since the norms are subject of investigation, we perform three separate experiments for the computation of the high dimensional distance: one for the 2-norm,

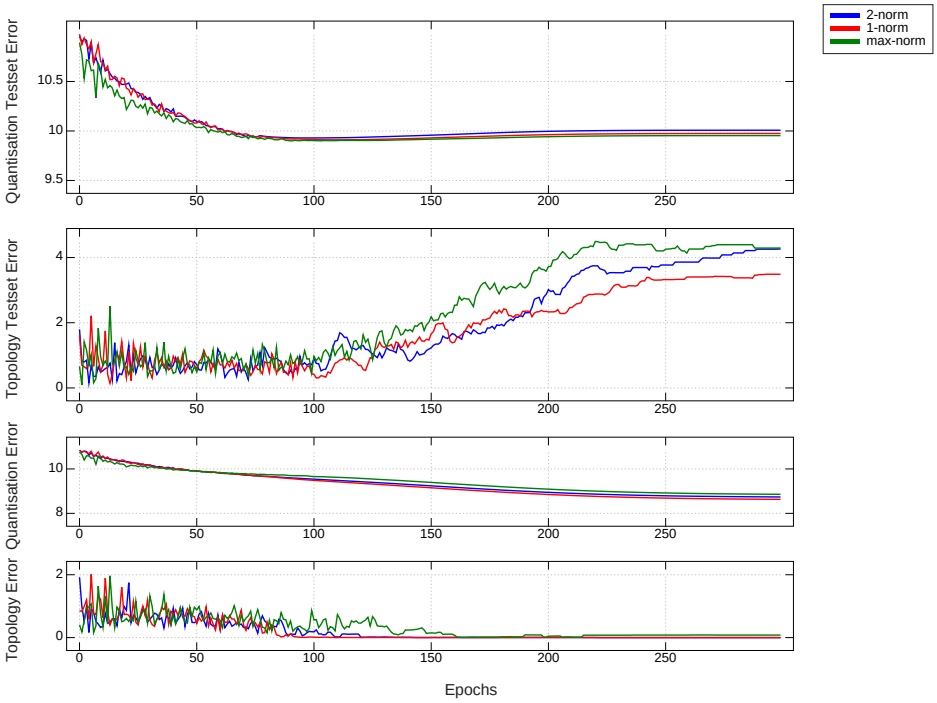


Figure 6.8: The choice of the high dimensional norm

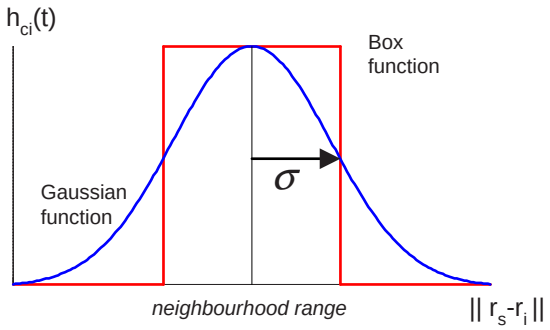


Figure 6.9: The approximation of a Gaussian neighbourhood by a box function

the 1-norm and the ∞-norm. For the quality measures we use the same functions, QE_{avg} and TE_{avg} as in Section 6.4.2.2. The size of the feature space is taken to be $N_c = 256$ and the number of epochs is taken sufficiently large for this experiment $T = 300$. See Figure 6.8 for the result. For now the optimum number of epochs is taken in the range of $80 < T < 150$ (for a low QE and reasonable $TE < 2$). Since the QE values practically are in the same range for all norms it is favourable to select the ∞-norm because the Linedancer can implement the ∞-norm with fewer clock cycles than the 1-norm. The computation for the 2D-distance will be done with the 1-norm since for short vectors it is faster than the ∞-norm.

The topology error on the test set in Figure 6.8 (the ∞-norm graph) has a higher slope than the topology error for Dimension($= N_c$) $= 256$ in the test set in Figure 6.7. To exclude that this effect is caused by a too few training epochs (under-training) it is therefore recommended to train more gradually and enlarge the number of epochs for the next trade-off issues.

6.4.2.4 Boxed neighbourhood

Another issue that touches on the quality-performance trade-off is the choice of the neighbourhood function. The SOM algorithm (Algorithm 6.1) uses a Gaussian function, but its computation on the Linedancer is expensive. The purpose of this subsection is to devise an acceptable alternative. Kohonen [77] suggests to use a box-function, see Figure 6.9.

The training algorithm is changed with respect to the neighbourhood function, the Gaussian $N(\mu, \sigma)$ is replaced by the box function $B(\mu, \sigma)$ with same values for μ and σ. For the experiment the size of the high dimensional space N_c is set to 256, the number of training epochs is increased to $T = 400$, and the norms are fixed to ∞-norm for the high dimensional distance and 1-norm for the 2D distance. The same applies to the quality measures $QE = QE_\infty$ and $TE = TE_1$ since this does not influence relative comparison with these measures. See Figure 6.10 for the result of the experiment.

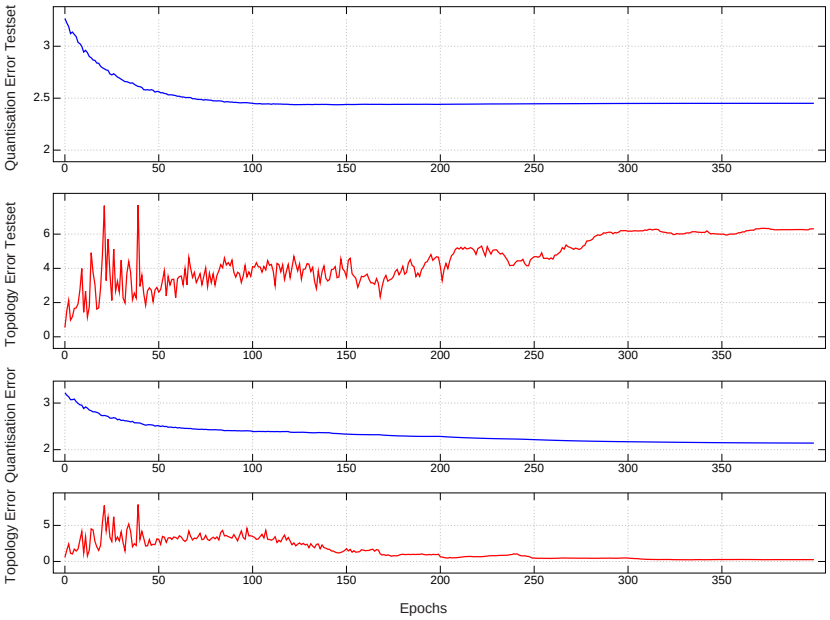


Figure 6.10: Result of the approximation of the Gaussian neighbourhood by a boxed neighbourhood

The experiment shows that the optimal range of epochs is dictated by the topology error on the test set. This range has moved up from $80 < T < 140$ to $150 < T < 250$ and the TE has increased from 2 $\rightarrow\approx$ 5. In the stable interval for QE and TE for $T > 300$ there is no indication that a larger number of epochs would improve the quality measures. From experiments we saw that the placement of similar articles close to each other is not critical for this increase of topology error. We therefore accept the replacement of the box function.

Since the neighbourhood function is centered around $\mu = r_w$ the function can be rewritten by

$$B(r_w, \sigma) = \begin{cases} 1, & \text{if } \|r - r_w\|_1 < \sigma \\ 0, & \text{otherwise.} \end{cases} \tag{6.12}$$

When a lot more articles have to be accomodated in the map it is recommended to approximate the Gaussian neighbourhood in a better way with a mixture of box functions, for example $\beta \cdot B(\mu, \sigma_1) + (1-\beta) \cdot B(\mu, \sigma_2)$, where each box function $B(\mu, \sigma)$ has unit area.

6.4.2.5 Precision

The topic of this subsection is on managing the memory resources of the two Linedancer chips that comprise the hardware system. This should be done in such a way that an optimal balance between quality (accuracy) and map size is obtained. For implementation reasons the N_c-dimensional neuron weights $\vec{m}_r$ have to be reduced from single precision floats to a fixed point representation. To reason about the influence of fixed point precision we use the format $integer \bullet fraction$, as introduced in Section 4.4.2.2.

In [97] it is reported that a precision of 8 or 16 bit for the representation of neuron components in $\vec{m}$ in a SOM training is adequate in some cases. Our hypothesis is that an 8 bit integer $0 \bullet 8$ is sufficient, which also applies for the learning rate $\alpha(t)$. The hypothesis also includes the neighbourhood size $\sigma(t)$, which is truncated to a 4 bit integer $4 \bullet 0$, see Table 6.7.

To test the hypothesis the code has been changed to accomodate for the reduction in accuracy. The memory allocation details are described in Table 6.8. The result of the test is shown in Figure 6.11. What can be observed clearly from this figure is the plateau behaviour (quantisation effects) of all QE and TE quality measures: $200 < T < 240$ for all quality measures, and $240 < T < 300$ for the topology error on the test set and the quantisation error. The optimal value for the $TE = 5$ in the epoch range $100 < T < 200$, whereas for $QE = 2.5$ an epoch range $110 < T < 240$ is best. The choice of the accuracy of the neuron weight $\vec{m}_r$ is directly coupled to the size of the neuron map, given a fixed number of Linedancer processors. For a $0 \bullet 16$ format a maximum map size of 16×16 is possible given the current Linedancer configuration. A $0 \bullet 8$ format allows for a doubling of the map size. From the maps that we created we found that the choice of 8 bit for the neuron weights $\vec{m}_r$ and learning rate α, and 4 bit for the neighbourhood size gave sufficient qualitative results. However, more extensive

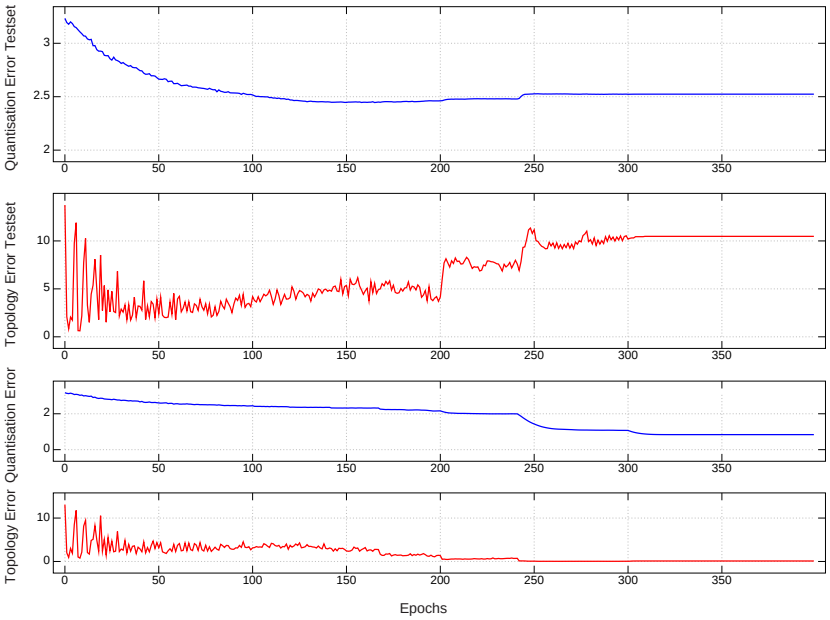


Figure 6.11: Result of a precision of 8 fractional bits for neuron weights and 8 integer bits for sigma

tests are needed with quantitative results before the training parameters can be fixed. For now the optimal number of training epochs is set to $T = 175$.

We did not investigate the effect of increasing the number of samples beyond 300, or the effect of increasing the precision to $0 \bullet 16$. However, the model can be adapted easily and the quantitative results can be correlated with the perceptual quality of the SOM-maps.

At the end of the trade-off subphase all concessions to quality (and implicitly also to the functionality) are made and a so called *golden reference* is determined. The golden reference is defined as the output of carefully constructed test cases and is used to build up confidence of a correct working of the system. From this point on the functionality of the models will be kept bit true. The quality measures QE and TE, fixed to QE_∞ and TE_1 respectively in Section 6.4.2.3, can subsequently be calibrated to a sufficiently perceptual quality. In this way these measures can be taken as a first baseline for quality in case of a design iteration.

6.4.3 Reorganisation subphase

The purpose of the reorganisation phase is to gradually expand the executable models such that they get "closer" to the target hardware architecture by each transformation step. Each individual transformation is behaviour preserving (bit-wise accurate).

As an example we selected an optimisation of the update step (line 14 of Algorithm 6.1)[10]. The computation of the update step is defined by (6.6) and is coded by the assignment:

$$\vec{m}_r = \vec{m}_r + \alpha \cdot h_{r,r_w} \cdot (\vec{x}_s - \vec{m}_r).$$

In the trade-off subphase (Section 3.8.1) the neighbourhood function h_{r,r_w} has been reduced to $B(r_w, \sigma)$ (6.12) and the update step simplifies to:

$$\vec{m}_r = \begin{cases} \vec{m}_r + \alpha \cdot (\vec{x}_s - \vec{m}_r), & \text{if } \|r - r_w\|_1 < \sigma\} \\ \vec{m}_r, & \text{otherwise.} \end{cases} \tag{6.13}$$

We will now estimate the execution time of performing the update step. Let ADD_{width}, SUB_{width} and MUL_{width} stand for the number of cycles for adding, subtracting and respectively multiplying two bit-fields that result in a bit-field with length *width*, as defined in Section A.1. Note also that an even field width is used for performance reasons, see same Section A.1. Then for each neuron – within the current neighbourhood range – this accounts for $T_{upd} = SUB_{10} + MUL_{8,10} + ADD_{10}$ clock cycles, where SUB_{10} corresponds to the computation of $\vec{x}_s - \vec{m}_r$, $MUL_{8,10}$ cycles corresponds to the scalar-vector multiplication by learning rate α, and ADD_{10} corresponds to the vector-vector addition with $\vec{m}_r$. According to Table A.1, a subtraction with a 10 bit result (SUB_{10}) takes

[10] A trivial optimisation, namely storing and reusing the common subexpression $\vec{x}_s - \vec{m}_r$ (steps 1 and 5 or lines 8 and 14 of Algorithm 6.1), is not feasible because of memory constraints.

12 cycles for $w = 8$ bit operands. The multiplication $MUL_{8,10}$ takes $(l_1 - 1) = 7$ (conditional) additions of $w = 10$ bit integers (see Section A.1) and $4 + l_1 + l_2$ cycles for initialisation, where $l_1 = 8$ and $l_2 = 10$. The multiplication accounts for $(l_1-1)\cdot(4+w)+(4+l_1+l_2) = 7\cdot(4+10)+(4+8+10) = 120$ cycles. The addition (ADD_{10}) takes, like the SUB_{10}, another 12 cycles. The total is $T_{upd} = 144$ cycles.

However, (6.6) can be rewritten as:

$$\vec{m}_r = \begin{cases} (1-\alpha)\cdot\vec{m}_r + \alpha\cdot\vec{x}_s, & \text{if } \|r - r_w\|_1 < \sigma \\ \vec{m}_r, & \text{otherwise.} \end{cases} \tag{6.14}$$

This is computed in time $T_{opt} = (MUL_{16} + \frac{MUL_{16}}{H} + ADD_{10})$, since the computation of $\alpha\cdot\vec{x}_s$ is constant for at least a row of $H = 16$ neurons and is computed once per sample. A MUL_{16} is computed in 112 cycles, according to Table A.1 ($=6\cdot w + w^2$ for $w = 8$). The other terms are 7 and 12 cycles, yielding a total of 131 cycles. To summarise: the original equation (6.6) takes $T_{upd} = 144$ cycles to compute, compared to $T_{opt} = 131$ (6.14). In this case the optimisation is not so large (relative improvement 9%).

The algorithm is transcribed as an adaptation of `Fsample` in page 162, see functional code below, where `norm1` computes the Manhattan norm (Section 6.4.2.1), and where the neighbourhood function is approximated by a box function.

```
Fsample' ms s = [
                   if ( norm1( r -^ rw ) < sigma )
                      ( ( 1 - alpha ) *^ ms!r +^ alpha *^ x!s )
                      ( ms!r )
                | r<-rs
                ],
```

where `-^`, `+^`, `*^` are the same vector operators as introduced on page 162, and with `rw` as on page 162.

6.4.4 Template subphase

The goal of the template subphase is to derive common parts of code or components. One relevant category of components is the handling of the Run Time Support (RTS) instructions that is used for more advanced mathematical functions, see Section A.1. Besides the computation of the result, also the setting of involved status registers has to be modelled in a bit true way for subsequent instructions. To show these aspects, as well as checking the calling conventions to decrease coding time, we included the following example.

Example

We selected the RTS vector subtraction to demonstrate the usefulness of checking the calling conventions and performing a full system emulation for the relevant instruction subset. The template is called with two operands

and one result bit-field. The calling conventions, dependencies and other constraints comprise:

1. only one of the operands can be in EXT memory,
2. the width of the result field should fit the computed result,
3. the restriction of even operand sizes because of performance reasons,
4. the RTS SUB function modifies register $ab1$, and
5. the result is computed *only* for those PEs that have register $ab0$ set (==1).

Because the template arguments are passed via a call-by-name mechanism (argument follows the variable naming convention `<name>_<start_position>_<length>`, see Section 3.8.2), the template can check constraints 1-3. Full system emulation handles, for example, constraints 4 and 5, and performs besides the subtraction also the update of flag $ab1$ for each PE except for those that have $ab0 == 0$.

End example

6.4.5 Translation subphase

The final step of developing code is the generation of a target hardware language description. Templates can also play a role in this step, because they can also provide additional information, that facilitates the translation process.

Example

To demonstrate the transformation of the model to the source code of the target hardware, we show the translation of an instance of the *subtractVector*-template. The example demonstrates the generation of Linedancer code from a template. The arguments of the template call are literally taken and substituted in the RTS frame, see Section A.1.

```
subtractVector(X0,X1,Xresult)
⇒
aop{
    // subtract X1 from X0 and store the result in Xresult
    RTS(Sub, Int, -, @{Xresult}, @{X1}, @{X0)
};
```

End example

6.5 Results and Discussion

In this section the results of the previously elaborated developments are discussed and combined in order to formulate a conclusion on the feasibility of the Linedancer implementation.

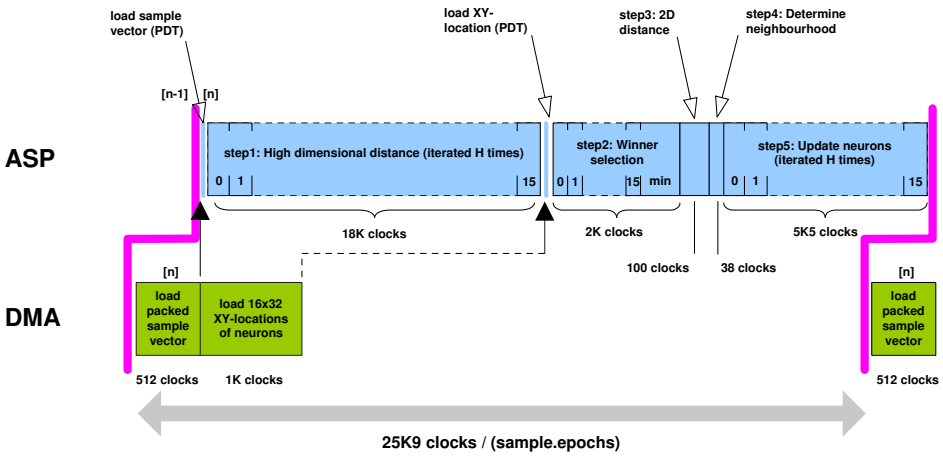


Figure 6.12: Timing diagram for the 5 processing steps in the inner loop, based on the ∞-norm variant

Timing. The timing diagram for the 5 steps 'in the inner loop' of Algorithm 6.1 (lines $8 \cdots 14$) is depicted by Figure 6.12. As can be observed the DMA controller activities for loading the next sample vector and the reloading of the 2D positions of the neurons, can be hidden in the processing by the ASP.

The performance measurements of the Linedancer are now compared with two – manually optimised – Intel Pentium 4 implementations, one with Streaming SIMD Extensions 2 (SSE2) instructions and one without SSE2 instructions. See Figure 6.13 and Table 6.5 for a detailed comparison. The timing results of Figure 6.13 are based on $W = 16$, $H = 32$ and $N_c = 256$ for the map dimensions, $S = 500$ for the number of samples and $T = 250$ for the number of epochs. Both Pentium implementations are based on estimates under ideal conditions, that is, using the most optimal SSE2 instructions and large enough caches to host all data. Since the SSE2 can operate on 4 single precision floats at a time, it is expected that the computation speeds up the sequential computation with a factor 4 for parallel operations (like step 5). Both Pentium versions use the 1-norm for computing the length of a vector. The two Linedancer results are measured cycles; the Pentium results are best estimates.

The SSE2 estimate is almost 5 times as fast as the sequential version, in particular step 1 has a relatively high speedup. The Linedancer versions however, are disappointing compared to the SSE2 implementation, in particular when considering the various concessions to the precision of computation. The performance of step 1 falls short for the 1-norm as well as for the ∞-norm. Especially for the ∞-norm, which was expected to take fewer cycles because there is no need to sum up all components as in the 1-norm. The reason for this poor performance is

Training step		Pentium estimates under ideal conditions (2 GHz)		Aspex Linedancer-P1 measurements (300 MHz)	
		sequential [cycles]	SSE2 [cycles]	1-norm [cycles]	∞-norm [cycles]
1. Distance in highD		393216	65024	43384	18028
2. Winner selection		768	768	2158	2158
3. Distance in 2D		2560	2560	100	100
4. Determine neighbourhood		512	512	38	38
5. Update neurons		393216	98304	5590	5536
total [cycles]		790272	167168	51270	25860
total time [sec] for a particular training job	estimated	49.4	10.4	21.4	10.8
	measured	125	75	21.1	10.7

Table 6.5: Comparison of training cycles per sample per epoch, and the total training time of $S = 500$ samples and $T = 250$ epochs

the relatively slow adding tree implementation for the 1-norm and slow maximum operator for the ∞-norm. At a local level, however, the Linedancers do speed up step 5 significantly and compensate amply for the lower clock frequency of the Linedancer ($f_{P4} : f_{LD} = 2.0$ GHz : 300 MHz).

Both Pentium implementations have also been run on a 2.0 GHz Pentium processor. Both versions are made by compiling the Algorithm 6.1 with the Intel C++ compiler (version 8.0), one with the SSE2 optimising option. The sequential Pentium implementation takes 125 sec to complete (for 500 samples, 250 epochs). The SSE2 version runs at a disappointing 75 sec; inspection of the generated assembly code revealed sub-optimal SSE2 code. Moreover, both versions suffer from L1 and sometimes L2 cache overflow; the SOM neural network and the samples already take 1MB of memory[11].

The main reason for the poor performance of the dual Linedancer system is the relatively high communication overhead in the inner loop. We collected for the most dominant part, the high dimensional distance (step 1), how many cycles were spent in communication and how many in computation, see Figure 6.14. This figure shows that the communication overhead dominates the computation cost.

When communication would improve such that processing and communication could be perfectly balanced then this would result in a 2.5× performance

[11] Cache overflow depends on the size of the caches. The L1 cache in the Pentium 4 processors is far less than 1MB in size; average value for L1 is 16 KB and for L2 512KB.

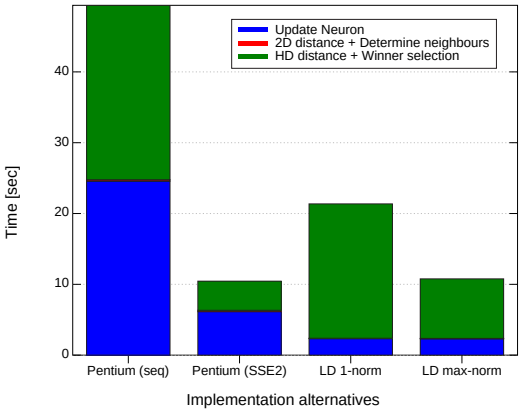


Figure 6.13: Comparison of implementation alternatives for SOM training

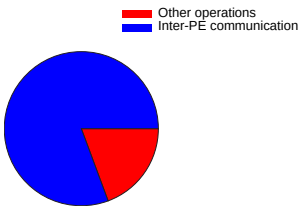


Figure 6.14: Distribution of communication and computation in High Dimensional Distance computation (∞-norm)

improvement for 1-norm (21.4 → 8.6 sec) and 2× for ∞-norm (10.8 → 5.3 sec). When the next generation Linedancer (HD, see Section 2.3.4) would be used then the performance could improve with a factor of 2.1 (10.8 → 5.2 sec)[12]. Since the performance specifications of the Linedancer-HD and the chip itself are not available no further estimates on HD timing can be given.

Concluded is that the inter-PE communication, in particular the asynchronous communication, of the Linedancer architecture should be improved significantly in order to be competitive even to non-embedded processors for this application.

Memory allocation. The final memory allocation for this mining case is presented in two tables, one for fixed parameters used by the algorithm (Table 6.7) and the other for the variables used in the algorithm (Table 6.8).

The first table lists all parameters used in Algorithm 6.1, complete with the domain they are defined on, and the chosen fixed point representation. Of the parameters, only the learning rate α is stored in the Linedancer; the other parameters are hosted on the SPARC. All other parameters are either used in immediate mode on the PEs for direct processing or kept on the SPARC (e.g., for flow control in case of n). Note that the Linedancer architecture has a preference for even field width, see Section A.1. Some variables are dimensioned a bit wider than necessary (e.g., r_w and r), because these are not involved in time critical sections of the code.

[12] The estimate is based on the following assumptions: the time for a *maximum*-operation is divided in half because of the two-channel architecture (see Section 2.3.4), the improved inter-PE communication network (chordal ring), and finally the higher clock frequency (400 : 300 MHz).

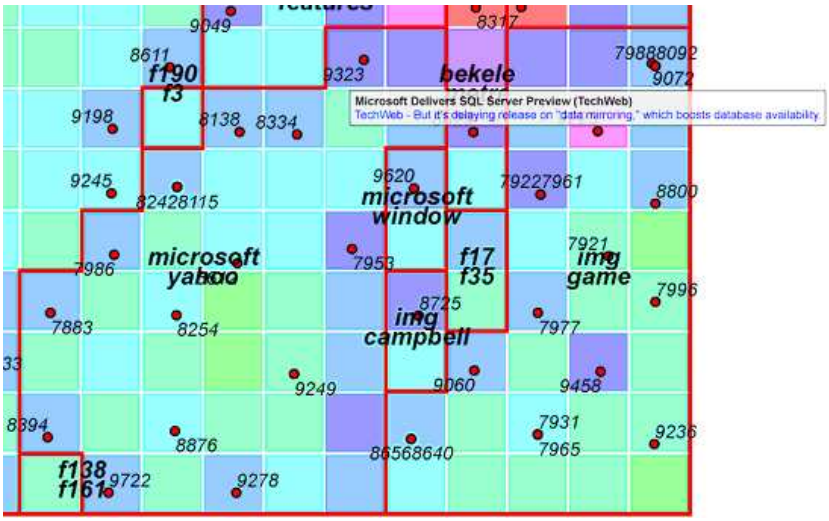


Figure 6.15: Part of a sample SVG-map (zoomed in)

The final allocation of variables to the local memory is more complex than for the other cases because the usage of the memory fields dynamically changes over time and is dependent of the location of the PE. For example, the computation of the high dimensional distance (step1, algorithm line 8) prefers a segmentation per neuron ($H == 32$ segments of 256 PEs each), whereas for the computation of the winning neuron (step 2, line 10) we prefer one single large segment.

Quality of the maps. No formal user tests are performed. The quality of the maps are informally discussed since the quality of the maps is not the focus of this study. The quality measures are only used for comparison purposes and to obtain an indication of the number of epochs T.

An example of a SVG map is shown in Figure 6.15. It has been created from newsgroups BBC News and BBC Sports in June 2005. Each square represents a neuron vector, containing $N_c = 256$ components, which is generalised from samples located in its neighbourhood. Similar neurons form a cluster, indicated by the colour of the neurons. This colour represent how much neighbouring neurons resemble the one in focus: e.g., the colour yellow and green indicate similarity whereas purple and pink indicate differences. Each cluster (or country) is bordered by red lines, to indicate the dissimilarity of two adjacent clusters. See for example the country "microsoft, yahoo" in the left part of the map in Figure 6.15. Table 6.6 describes some articles of the two largest countries.

Country name	ID	Title
microsoft, yahoo	9323	Microsoft Delivers SQL Server Preview (TechWeb)
	8138	Gates spotlights windows, Office updates (USATODAY.COM)
	8242	Microsoft Xbox 360 aims at Sony's hold on Japan (Reuters)
img, game	8656	Woods has open mind on Presidents Cup Pairings
	8640	Woods has open mind on Presidents Cup Pairings
	9458	Pierce returns to favorite venue for Fed Cup final

Table 6.6: Some articles in two countries of the map in Figure 6.15

description	symbol	range	fixed point scheme	memory field	domain	host
learning rate	α	$(0, 1]$	$0 \bullet 8$	immediate	fixed point	SPARC
neighbourhood size	σ	$[1, max(W, H)/2]$	$4 \bullet 0$	immediate	fixed point	SPARC
learning rate factor	f_α	$(0, 1]$		global	float	SPARC
neighbourhood size factor	f_σ	$(0, 1]$		global	float	SPARC
epoch number	t	$[1, 255]$		global	fixed point int	SPARC
training sample index	s	$[0..2^{32} - 1]$		global	fixed point int	SPARC

Table 6.7: Fixed point accuracy of parameters in the SOM training algorithm

description	dependency	range	fixed point scheme	memory field	algo-rithm-lineno
weight vector	$\vec{m}_r = (1 - \alpha) \cdot \vec{m}_r + \alpha \cdot \vec{x}_s$	$[0, 1)$	$0 \bullet 8$	EXT $64 - 191(8)$	14
training sample	$\vec{x}_s$	$[0, 1)$	$0 \bullet 8$	CAM $0(8)$	
high dimensional distance	$\delta_r = \|\vec{x}_s - \vec{m}_r\|_\infty$	$(-1, 1)$	$2 \bullet 8$	CAM $16(16)$	8
location winning unit	$r_w = argmin_r(\|\vec{x}_s - \vec{m}_r\|)$	$([0, W - 1], [0, H - 1])$	$8 \bullet 0$	CAM $48(8), 56(8)$	10
location of all units	r	$([0, W - 1], [0, H - 1])$	$8 \bullet 0$	CAM $16(8), 24(8)$	
low dimensional distance	$d_{r,r_w} = \|r - r_w\|_1$	$[0, 46]$	$8 \bullet 0$	CAM $16(8)$	12
neighbourhood matrix	$h_r = (d_{r,r_w} \leq \sigma) = Box(r_w, \sigma)$	$\{0, 1\}$	$8 \bullet 0$	CAM $16(8)$	13

Table 6.8: Fixed point accuracy of variables used in the SOM training algorithm

For the user interface an interaction functionality is supported. A mouse-over event on the name label of an article returns the title in a semi-transparent text balloon (tooltip), see Figure 6.15. Clicking on this label will open an internet browser window with the content of the whole article. A mouse-over event on a neuron will return information about its specialisation. SVG has native support for zooming and panning.

Methodology and Architectural language issues. The high level design stages are developed with the proposed evolutionary development methodology. The methodology contributed in the estimation of the value of training parameters as well as of their precision, justification of the various simplifications, the calibration of the two quality measures QE and TE, and the mapping of the variables used.

The various simplifications, conducted in the trade-off subphase, include the choice for the dimension of the feature space, choice for norms, boxed neighbourhood, and the precision of parameters and variables. Also behaviour preserving optimisations are guided by IRIS. Especially for the trade-off subphase the instant graphics capability of the architectural language environment proved to be very valuable.

The unique contribution of this case to IRIS as compared to the other cases are:

- providing for a non-printing application (extending IRIS beyond image processing),
- the dynamic segmentation of the Linedancer array (run time configurable),
- the handling of two quality functions,
- the extensive use of (hashing based) string handling functions needed for the various text handling in feature extraction (Figure 6.2), and, finally,
- the sparse matrix support for handling the very large dimensions of the uncompressed document space in feature extraction and compression (Figure 6.2).

The last two items are very convenient for modelling parts of the context of our system.

6.6 Conclusions

A dual Linedancer P1 is approximately 5 times faster than a sequential Pentium implementation in training a SOM neural network: 10.8 sec versus 49.4 sec (neglecting cache effects for the Pentium). Disappointing is that the dual Linedancer-P1 system is not faster than the Pentium SSE2 implementation, despite that the

SSE2 figures were estimated under optimum conditions (disregarding, e.g., cache misses).

Improving on inter-PE communication such that computation and communication are better balanced would slightly increase the performance of the Linedancer-P1 (factor of 2.5 for 1-norm and 2 for ∞-norm). The Linedancer-HD would increase the performance with a factor of 2.1 with respect to the P1 1-norm implementation.

It is recommended to improve the performance of the inter-PE communication, and in particular the asynchronous communication network. This would not only improve the performance, but also the effectivity of scalability to larger network dimensions using multiple Linedancers.

The case contributed to the IRIS methodology in several unique aspects, among which there are: the use of the run-time configurable segmentation, and for the architectural language the extensive use of string handling and sparse matrices in the modelling of the problem context.

CHAPTER 7

Evaluation and Conclusions

7.1 Introduction

This thesis describes IRIS, a firmware development methodology for synchronously coupled many-core architectures such as SIMD and VLIW processing cores. This chapter concludes the work of this thesis.

First the general conclusions are drawn (Section 7.2). Subsequently the claims, stated in the Chapter 1, are revisited, see Section 7.3. Finally, remaining problems are discussed and directions for future research are given (Section 7.4).

7.2 Conclusions

IRIS can be characterised as a confidence-by-construction framework: it offers an incremental approach to system construction, that gradually converges to a target language implementation. Interactivity and executability provide early feedback, in particular on incorrect problem interpretation or design faults. In case of design iterations, models of previous phases or iterations can serve as a solid base for continuation in an alternate design refinement. Decoupling the development language from the target hardware architecture language offers freedom of choice for either migration to different target hardware architectures or different problem domains. Design space exploration and the decision recording during development raises quality and takes less time because the evaluation of design alternatives can be done *in situ*. All this is realised by using a development framework, that is based on a single architectural language for the whole trajectory, and in this fashion the foundation for our integral IRIS framework is layed.

7.3 Claims

The validity of the claims about IRIS, raised in Chapter 1, is illustrated with examples.

1. For an effective and efficient implementation on a massively parallel processing core it is necessary to manually (re)model the problem in a suitable parallel representation.

   We showed for image quantisation that stochastic modelling leads to a simpler parallel algorithm, and that it performs efficiently when mapped onto a massively parallel hardware architecture (Sections 4.2, 4.3, 4.4).

2. A semi-automatic and interactive development process is needed for mapping a task on a dedicated massively parallel processing core efficiently.

   In the stochastic image quantisation case the effect of the width of variables (in bits) on quality – used in a Random Number Generator – can be quickly modelled by using IRIS without implementing it (Section 4.4.2.3). A simple executable specification, an instant display of a quality versus bit-width graph (Figure 4.17), and visualisation of results (Figure 4.18), suffices for this purpose.

3. A single architectural language firmly based on mathematics for all development phases reduces development time and reduces the number of design errors.

   The stochastic image quantisation case demonstrates the use of a single language through all phases. To mention a few phases: modelling the problem and its parallel implementation with simulated annealing (familiarisation, Section 4.2), a concise functional architecture and a quality function, both coded in a functional language (incremental prototype, Section 4.3), followed by an evaluation of various quality-performance trade-offs using instant graphics (Section 4.4.2) and finally the automatic translation to Linedancer code (Section 4.4.5).

4. Most of the relevant extra-functional requirements can be handled by integrating them into the regular functional flow; as a consequence the architectural language should support *in situ* monitoring and visualisation of quantifiable extra-functional properties.

   As an example the trade-off of quality versus performance in the data mining case, involved the comparison of a Gaussian neighbourhood with a box-shaped neighbourhood (Section 6.4.2.4, Figure 6.9). This resulted in an acceptable trade-off, by inspection of Figure 6.10 and relating it to sample maps.

5. In the development process small steps and immediate feedback are crucial.

   As an example we take the data mining case (Chapter 6, Section 6.4.2.3) where the computation of the 1-norm of a vector took too much time on the Linedancer. A less computationally intensive alternative norm, the ∞-norm, was quickly prototyped and had a small but acceptable quality loss.

6. The development process should have a phased approach serving the various development roles, and should subsequently include:

   (a) a familiarisation phase with respect to the problem and the target hardware architecture(s),

   (b) an incremental prototyping phase (hardware architecture independent),

   (c) a transformational development phase (hardware architecture dependent),

   which are performed in a cyclic manner when needed (e.g., in case of design iterations).

   An illustration of a phase-role interaction is a possible design iteration in the stochastic image quantisation case. During the behaviour preserving expansion of the functional model (reorganisation phase), an exceeded memory constraint can enforce reconsidering the precision of certain variables (trade-off phase). See Chapter 4, Subsection 4.4.2.2 and Figure 4.16 for the details.

7.4 Discussion and future research

In this section we address remaining problems and give direction for future research.

- The development framework IRIS is targeted to assist firmware development for many-core processing systems. To be more precise, the applicability of IRIS is restricted to the programming of cores that operate in a synchronous domain such as a control processor extended with SIMD cores (e.g., Linedancer Section 2.3.4) or a VLIW core (e.g., Avispa+[1] [24]). The Linedancer demonstrates that the constraint of the synchronous domain is not that strict, but the applicability of IRIS to Globally Asynchronous Locally Synchronous (GALS) [38] cores needs to be investigated in more detail.

- Processor manufacturers see a toolchain for firmware development as a an unavoidable obligation instead of a strategic opportunity to attract potential

[1] Silicon Hive designed the Avispa+, a VLIW-type processor.

users for their processor chips. The toolchains are without exception based on C with intrinsic functions or instructions. In this thesis the applications are specified in a functional language, which is more natural to specify parallelism. The question remains whether application programmers are willing to accept functional languages.

- In this thesis the functional specification is leading the development. However, the cases suggest that in the trade-offs between functional and extra-functional properties, print quality is just as important as functionality. It is not clear what the consequences are for IRIS if print quality is prime and functionality an outcome. Furthermore, the monitoring of extra-functional properties or constraints is now integrated in the functional flow. It is not clear whether this works for all extra-functional constraints (as for example development effort).

- Integration of the template and translation subphases into an interactive compiler framework is desirable, and therefore needs to be investigated.

- In this thesis we have shown that image processing functions and data mining can benefit from massively parallel compute models. The question is what other kind of advanced functionality around document processing is expected to profit from these kinds of hardware architectures; such as proposed in Intel's Recognition-Mining-Synthesis platform Section 1.2.1.

APPENDIX A

Relevant Linedancer Details

This appendix describes the relevant details of the used Linedancer processor family[1]. The programming aspects of the associative SIMD processor (ASP) and the communication hierarchy are addressed.

A.1 Relevant Linedancer instructions

This section describes various programming aspects of the Linedancer [106], that are used in the three cases, see Chapter 4, Chapter 5, and Chapter 6. In the following paragraphs we address: associative functions, high level library functions, inter-PE communication, and run-time segmentation. At the end the bit-field dependent instruction timing is described.

Associative functions. The associative functionality of the Linedancer allows for fast data dependent processing. For example a table-look-up-and-replace can be done efficiently on an associative array, see code fragment below. Multiple native Linedancer instructions are collected and specified in a special macro language (see the `aop{...}` macro block) to distinguish it from sequential C-code specifications. The search item (`key`) is looked up in parallel (`Tag`), and sets tag-register (`tr1`) for each PE that matches the search item (executed in the first cycle). The next instruction, `Activate`, actually activates the PEs that may participate in the subsequent operations. Finally, during cycle 3, the specified value (`value`) is written into the activated PEs in parallel. The total operation takes only 3 cycles, regardless of the number of data items.

[1] Aspex Semi-conductor, `http://www.aspex-semi.com/`.

```
aop{
    Tag(Binary, key, Byte, 1, tr1, -),      // cycle 1
    Activate(Matching, -, tr1, -),          // cycle 2
    Write(Binary, value, Byte, 0)           // cycle 3
}
```

RTS library functions. One relevant category of programming components is the *Run Time Support* (RTS) function-library, that is used for more advanced mathematical functions.

The RTS library provides high level instructions for manipulating multiple-bit vector and scalar operands. Multiple RTS-instructions are specified and collected in `aop{...}` macro blocks. A variety of instructions are provided to perform assignment, arithmetic, logical and relational operations. RTS Instructions may operate on a PE memory fields (on the ASP) or between a PE memory field and a scalar value (ASP and SPARC repectively). This so called *RTS frame* have six parameters:

```
aop{
    RTS (Function , DataType , CommsOpt , Result , Operand1 , Operand2 ),
}
```

where:

Function:	Specifies the function of the RTS instruction. For example arithmetic functions include addition (ADD), subtraction (SUB), and multiplication (MUL).
Datatype:	Specifies the type of data processed by the specified function. For example the datatype is used to distinguish between signed (INT) and unsigned subtraction (CARD).
CommsOpt:	The CommsOpt parameter specifies if the communication is local or remote. For example getting a bitfield from a remote PE is coded by `co{Get,disp}`, where `disp` specifies the communication displacement.
Result, Operand1, Operand2:	There are three result and operand parameter formats. First, memory field specifications of the form `@{Size, Address}` for a vector operand. Second, scalar operand specification (e.g., a variable hosted on the SPARC. Third, an operand may also take on a literal value.

RTS instructions implicitly use two particular activity bits (ab_0 and ab_1, two of the eight programmable bit-flags) for its operation. The RTS functions are governed by the following rules:

- if two vector operands are used then only one vector operand can come from extended memory (EXT),

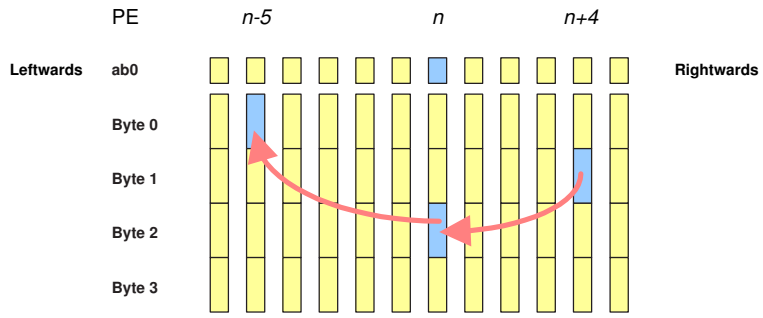


Figure A.1: Example of synchronous inter-PE communication

- only those PEs that have their activity bit ab_0 set participate, and
- activity bit ab_1 is modified.

It is beyond the scope of this thesis to describe further details. For more details we refer to [106].

Inter-PE communication. Communication between PEs is facilitated by a shift register, effectively 1 bit per clock cycle but for all PEs in parallel. The Linedancer-HD supports a chordal ring – similar to the MiMagic6 [91] – with a chord length of 32, meaning that long communication distances are reduced by a factor of 32 with respect to the Linedancer-P1. Longer distances or wider data fields increase the communication time. As is the case with the RTS functions, register ab_0 activates the PE that may participate. For example, in Figure A.1, PE n is allowed to receive a byte from its right neighbour on distance 4, see the first `RTS(Assign...)` instruction in the Linedancer-C code fragment below.

```
aop{
    RTS(Assign, Bitset, co{Get,4}, @{Byte2}, @{Byte1}, -),
    RTS(Assign, Bitset, co{Put,-5}, @{Byte0},  @{Byte2}, -)
}
```

The subsequent `Assign` instruction executes a byte transfer from the current PE to the 5^{th} PE left of the current activated PE. When all PEs would have their ab_0-register set, then all would participate in the parallel communication process. Effectively, moving byte1 to byte2 and byte2 to byte0 for all processing elements in parallel.

Segmentation. The string of PEs can be split into a number of independent segments and in fact makes the Linedancer run-time reconfigurable. Communication cannot take place between segments, therefore, when the string is split

operation	operand and result bit-width	estimated number of clocks
add	$w + w \rightarrow w + 2$	$4 + w$
subtract	$w - w \rightarrow w + 2$	$4 + w$
add/subtract	$w \pm w \rightarrow w + 2$	$4 + 2 \cdot w$
multiply	$w * w \rightarrow 2 \cdot w$	$\pm 6 \cdot w + w^2$
less or equal	$w \leq w \rightarrow (w + 2) \rightarrow 1$	$6 + w$
xor	$w \oplus w \rightarrow w$	$4 + w$
min/max (segment size 8K)	$\{w_0 \cdots w_{N-1}\} \rightarrow w$	$\pm 75 \cdot w$
min/max (segment size 4K)	$\{w_0 \cdots w_{N-1}\} \rightarrow w$	$\pm 54 \cdot w$
local assign	$w \rightarrow w$	$4 + w$
synchronous remote assign (P1)	$w \rightarrow w$	$22 + d \cdot w$
synchronous remote assign (HD)	$w \rightarrow w$	$\pm d \cdot w/32$
asynchronous remote assign	$w \rightarrow w$	$6 + 16 \cdot w$
PDT: (EXT,CAM)↔PDS	$w \rightarrow w$	$14 + 12 \cdot w$

Table A.1: Estimated performance of some native Linedancer-P1 instructions in number of clocks for fields with even bit-width. The parameters w and d are used for bit-width of variables and communication distance where appropriate.

each segment and their associated PEs can be considered as independent (smaller) SIMD engines driven by a common instruction bus. The Inter-PE communication network can only be split into segments that are multiples of sixteen PEs. Most of the time a segment of the maximum number of PE is chosen. In some case, for example in Chapter 6 we alternately use 256 and 8K.

Instruction timing. The Linedancer is in essence a bit-oriented architecture. This has its implications on the timing of basic functions as addition or multiplication. See for example Table A.1, that contains the estimated number of instruction cycles for even length bit-fields (w is even). This table shows the performance figures of some popular functions, some with two operands (dyadic) and some with a single operand (monadic). The dyadic functions include arithmetic and logical functions. The size of the operand bit-field is denoted by w, and for the dyadic functions the table only holds equal operand sizes. Since the Linedancer ALU actually is 2 bit wide, even length bit-fields are computed almost twice as fast as odd length bit-fields.

The monadic functions include special functions, such as maximum and minimum on a set of values (vector components), and assignment instructions which can use inter-PE communication. Two types of communication exists: synchronous (distance is denoted by d), and asynchronous for the longer distances.

In particular during feasibility and optimisation studies, an estimation of the time to compute arithmetic operations is often required. For convenience a second timing model is used that is parameterised by the length of the resulting bit-field.

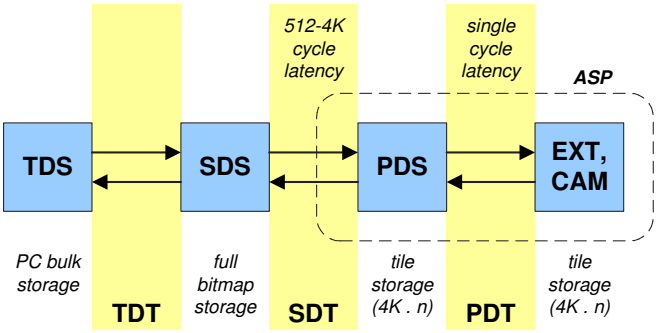


Figure A.2: The storage hierarchy of a Linedancer based system. Indicated are the four memory subsystem and the three involved communication processes (n is the number of Linedancers).

The reason for this second model is the opportunity to combine at least three concerns in a single way: the coding of the functionality, the memory resource consumption and the timing. In particular the better control on the memory consumption is the reason to base the (timing) model on the length of the result field. The model includes the operations add, subtract and multiply and their execution time is denoted by ADD_l, SUB_l, and MUL_l respectively, where l is the width of the result field (both operands are equally wide). For example, an addition with an l-bit result takes a time of $l+2$ cycles ($l = w+2$, see Table A.1).

Multiplications with not equal operand sizes are also used in the cases. The time for such a multiplication – denoted as MUL_{l_1,l_2} – is parameterised by two numbers, the length of multiplier l_1, and the length of multiplicand l_2. For example a multiply instruction of a $l_1 = 4$ bit variable by a $l_2 = 6$ bit variable can be computed by $l_1 - 1 = 3$ times a shift-add-pair on a 6 bit operand and would take $(l_1 - 1) \cdot (4 + w)$ cycles, where $w = 6$ (see Table A.1 for *add* timing). Additionally an initialisation is needed of a $l_1 + l_2 = 10$ bit result field that uses a local assign and takes an additional $4 + w$ ($w = 10$), see Table A.1 for *local assign* timing. The total time for a $l_1 + l_2 = 10$ bit-field multiplication result is around $3 \cdot (4 + 6) + (4 + 10) = 44$ cycles.

A.2 Storage hierarchy

Optimal performance not only requires balancing the computation and communication for the ASP, but also that of the complete data 'supply' chain. See Figure A.2 where the complete data supply chain is pictured. The three stores *Primary Data Store* (PDS), *Secondary Data Store* (SDS), and *Ternary Data Store* (TDS) are involved in three linked data communication processes: *Primary Data Transfer* (PDT), *Secondary Data Transfer* (SDT) and *Ternary Data*

storage element	transfer process	description
EXT, CAM	PDT: EXT, CAM↔PDS	Figure 2.9
PDS	SDT: PDS↔SDS	Figure 2.9
SDS	TDT: SDS↔TDS	Figure 2.10, Figure 2.11 (external data DRAM)
TDS		bulk-storage (e.g PC based)

Table A.2: Storage elements and their related communication processes

Transfer (TDT), see Table A.2. The SDT and the TDT are sequential DMA-like processes, while the PDT is a massively parallel communication process. At the start the local memory store (EXT, CAM) of the ASP is initialised with the data for the first job. Then, during computation by the ASP, the SDT communication process stores the previous result and loads new data into the PDS. After computation, the ASP usually performs a PDT communication process that swaps the content of a result vector in local memory with the content of the PDS in parallel for all PEs in a few cycles (see Table A.1 for *PDT* timing). When the SDS memory empties its data buffer for the final computation, a TDT process is triggered to request new data.

BIBLIOGRAPHY

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[2] A. Anjewierden, R. de Hoog, R. Brussee, and L. Efimova. Knowledge flows in weblogs. In *Proceedings of the 13th International Conference on Conceptual Structures (ICCS 2005)*, Kassel, Germany, 2005.

[3] Joe Armstrong. *Programming Erlang, Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] Arnulfo P. Azcarraga and Teddy N. Yap Jr. SOM-Based Methodology for Building Large Text Archives. In *DASFAA*, pages 66–73, 2001.

[5] Arnulfo P. Azcarraga and Jr. Teddy N. Yap. Extracting meaningful labels for WEBSOM text archives. In *Proceedings of the tenth international conference on Information and knowledge management (CIKM 2001)*, pages 41–48, New York, NY, USA, 2001. ACM Press.

[6] Andrew D. Bagdanov. *Style Characterisation of Machine Printed Texts*. PhD thesis, University of Amsterdam, May 2004.

[7] Arnab Banerjee, Pascal T. Wolkotte, Robert D. Mullins, Simon W. Moore, and Gerard J.M. Smit. An energy and performance exploration of network-on-chip architectures. *To appear in IEEE Transactions on Very Large Scale Integration (VLSI) Systems Special Section on Networks-on-Chip*, 2008.

[8] H. P. Barendregt. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[9] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP – A Self-Reconfigurable Data Processing Architecture. *Journal of Supercomputing*, 26(2):167–184, September 2003.

[10] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition, 2004.

[11] J.B.C. Beckers, W.P.M.H. Heemels, B.H.M. Bukkems, and G.J. Muller. Effective industrial modeling for high-tech systems: The Example of Happy Flow. In *The International Council on Systems Engineering (INCOSE)*, San Diego, 2007.

[12] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. In *IEEE Transactions on Signal Processing*, pages 397–408, 1996.

[13] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, 2nd edition, 1998.

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

[15] Maarten Boasson. Architecture and design: a case study. In *NATO Research & Technology Organisation: Building robust systems with fallible construction (IST-064/RWS-011)*, 2006.

[16] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Inc., 1981.

[17] Barry W. Boehm. Understanding and controlling software costs (invited paper). In *International Federation for Information Processing Congress (IFIP)*, pages 703–714, 1986.

[18] A.D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[19] L.B. Brisolara, M.F.S. Oliveira, R. Redin, L.C. Lamb, L. Carro, and F. Wagner. Using UML as Front-End for Heterogeneous Software Code Generation Strategies. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2008)*, pages 504–509, Munich, 2008.

[20] William W. Broenkow. *Introduction to Programming with MATLAB for Scientists and Engineers*. ML Books, The mathWorks, 2001.

[21] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.

[22] Alan Brown. Model Driven Architecture: Principles and practice. *Journal of Software and System Modeling*, 3(4):314–327, 2004.

[23] M.D. van de Burgwal, G.J.M. Smit, G.K. Rauwerda, and P.M. Heysters. Hydra: an Energy-efficient and Reconfigurable Network Interface. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'06)*, pages 171–177, Las Vegas, Nevada, USA, June 2006.

[24] G. Burns, P. Gruijters, J. Huisken, and A. van Wel. Reconfigurable accelerator enabling efficient sdr for low-cost consumer devices. In *SDR Technical Forum*, Orlando, Florida, November 2003.

[25] Francky Catthoor. The software washing machine. In *Proceedings of the 15th international symposium on System Synthesis (ISSS 2002)*, New York, NY, USA, 2002. ACM. (invited talk).

[26] Peter Checkland. *Systems Thinking, Systems Practice: Includes a 30-Year Retrospective*. John Wiley & Sons, September 1999.

[27] Theo A. C. M. Claasen. System on a Chip: Changing IC Design Today and in the Future. *IEEE Micro*, 23(3):20–26, 2003.

[28] Mirko Conrad and Heiko Dörr. Model-based development of in-vehicle software. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2006)*, pages 89–90, 2006.

[29] M.J. Crawley. *The R Book*. John Wiley, 2007.

[30] John Crinnion. *Evolutionary Systems Development, a practical guide to the use of prototyping within a structured systems methodology*. Hyperion Books, 1990.

[31] Rui Dai. Real time clustering and visualization of dynamic information using a massively parallel embedded processor. Master's thesis, University of Singapore, 2005.

[32] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Progammability and efficiency. *Queue*, 2(1):52–62, 2004.

[33] Ahmet Demir. Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development. In *Proceedings of Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software (MBD/MOMPES)*, pages 75–83, Berlin, 2006.

[34] Michael Dittenbach. The growing hierarchical self-organizing map: Uncovering hierarchical structure in data. Master's thesis, Technical University Vienna, 2000.

[35] Edward R. Dougherty. *Digital Image Processing Methods.* Marcel Dekker, Inc., New York, NY, USA, 1994.

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

[37] Petru Eles, Krzysztof Kuchcinski, and Zebo Peng. *System Synthesis with VHDL: A Transformational Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[38] Bevan Baas et al. Hardware and applications of asap: An asynchronous array of simple processors. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2006)*, August 2006.

[39] David A. Patterson et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[40] D.C. Pham et al. Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179–196, Jan. 2006.

[41] E. Waingold et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[42] Richard P. Kleihorst et al. Xetal: a low-power high-performance smart camera processor. *The IEEE International Symposium on Circuits and Systems (ISCAS 2001)*, 5:215–218 vol. 5, 2001.

[43] Simon Peyton Jones et al. Haskell 98 language and libraries: The revised report, 2002.

[44] S.R. Vangal et al. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan. 2008.

[45] T. Kohonen et al. Self organization of a massive document collection. In *IEEE Transactions on Neural Networks*, volume 11, pages 574 – 585. IEEE, 2000.

[46] Teuvo Kohonen et al. *Self organization of a massive text document collection*, pages 171–182. Elsevier, Amsterdam, 1999.

[47] R.W. Floyd and L. Steinberg. An adaptive algorithm for spatial gray scale. In *Society for Information Display, Symposium of Technical Papers*, pages 36–, 1975.

[48] Bernd Freisleben and Andreas Schrader. An evolutionary approach to color image quantization. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC 97)*, pages 459–464, Indianapolis, IN, USA, 1997.

[49] Jon Friedman. Matlab/simulink for automotive systems design. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2006)*, pages 87–88, 2006.

[50] D. D. Gajski, N. D. Dutt, Allen C-H. Wu, and Steve Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[51] C. García, Manuel Prieto, and Alberto D. Pascual-Montano. A speculative parallel algorithm for self-organizing maps. In *Proceedings of Parallel Computing 2005 (ParCo 2005)*, pages 615–622, 2005.

[52] Robert Geist, Robert Reynolds, and Darrell Suggs. A markovian framework for digital halftoning. *ACM Transactions on Graphics*, 12(2):136–159, 1993.

[53] Jason Ghidella and Jon Friedman. Model-Based Design Streamlines Development of Body Electronics Systems. *Automotive Electronics*, 05(6), 2005.

[54] Harry Goldstein. Winner: Cure for the multicore blues. *IEEE Spectrum*, 44(1), 2007.

[55] Solomon W. Golomb. *Shift Register Sequences*. Aegean Park Press, Laguna Hills, CA, USA, 1981.

[56] R. Gonzales and R. Woods. *Digital Image Processing*. Prentice Hall, 2002.

[57] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

[58] Yuanqing Guo. *Mapping Applications to a Coarse-Grained Reconfigurable Architecture*. PhD thesis, University of Twente, Enschede, The Netherlands, September 2006.

[59] P. Haffner, L. Bottou, P. Howard, and Y. Le Cun. Djvu: Analyzing and compressing scanned documents for internet distribution, 1999. `citeseer.ist.psu.edu/haffner99djvu.html`.

[60] B. Hailpern and P. Tarr. Model-driven development: the good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[61] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of data mining*. MIT Press, Cambridge, MA, USA, 2001.

[62] Simon Haykin. *Neural Networks, a comprehensive foundation*. Prentice-Hall International (UK) Ltd., London, England, 1999.

[63] Paul Heysters. *Coarse-Grained Reconfigurable Processors - flexibility meets efficiency*. PhD thesis, University of Twente, September 2004.

[64] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`.

[65] Arthur L. Hsu and Saman K. Halgamuge. Enhancement of topology preservation and hierarchical dynamic self-organising maps for data visualisation. *Int. J. Approx. Reasoning*, 32(2-3):259–279, 2003.

[66] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons Inc., 1962.

[67] H. Kang. *Color Technology for Electronic Imaging Devices.* SPIE-International Society for Optical Engine, 1997.

[68] Steven T. Karris. *Introduction to Simulink with Engineering Applications.* Orchard Publications, 2006.

[69] 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].

[70] Zoltan Kato and Ting-Chuen Pong. A markov random field image segmentation model using combined color and texture features. In *Proceedings of the 9th International Conference on Computer Analysis of Images and Patterns (CAIP 2001)*, pages 547–554, London, UK, 2001. Springer-Verlag.

[71] Stuart A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution.* Oxford University Press, May 1993.

[72] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE trans on Computer-Aided Design*, 19(12), December 2000.

[73] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[74] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[75] Anneke Kleppe. *MDA Explained, The Model Driven Architecture: Practice and Promise.* Addison-Wesley, 2003.

[76] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, 1993.

[77] T. Kohonen. *Self-Organizing Maps.* Springer, 1997.

[78] Teuvo Kohonen. Self-organization of very large document collections: State of the art. In L. Niklasson, M. Bodén, and T. Ziemke, editors, *Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN 1998)*, volume 1, pages 65–74, London, 1998. Springer.

[79] A. Krikelis and C.C. Weems. *Associative Processing and Processors*. IEEE Computer Society, 1997.

[80] Brian Lawrence, Karl E. Wiegers, and Christof Ebert. The top risks of requirements engineering. *IEEE Software*, 18(6):62–63, 2001.

[81] Yann LeCun, Sumit Chopra, Marc Aurelio Ranzato, and Fu-Jie Huang. Energy-based models in document recognition and computer vision. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR 2007)*, 2007. (keynote address).

[82] Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proceedings of the 1999 Workshop on Signal Processing Systems (SiPS 1999)*, pages 181–190, Taipei, Taiwan, October 1999.

[83] Prabhu S. M. and Mosterman P. J. Model-Based Design of a Power Window System: Modeling, Simulation and Validation. In *Proceedings of IMAC-XXII: A Conference on Structural Dynamics, Society for Experimental Mechanics*, Dearborn, 2004.

[84] Thate J. M., Kendrick L. E., and Nadarajah S. Caterpillar automatic code generation. In *Society of Automotive Engineers World Congress (SAE 2004)*, Detroit, 2004.

[85] Bingfeng Mei. *A coarse-grained reconfigurable architecture template and its compilation techniques*. PhD thesis, Katholieke Universiteit Leuven, January 2005.

[86] Wen mei Hwu et al. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual conference on Design Automation (DAC 2007)*, pages 754–759. IEEE, 2007.

[87] J. Meij, editor. *Dealing with the data flood. Mining data, text and multimedia*, chapter Introduction to Multidimensional Scaling. STT/Beweton, The Hague, The Netherlands, 2002.

[88] Microsoft. The F# Manual, 2008.

[89] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. The MIT press, 1997.

[90] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[91] NeoMagic Corporation. MiMagic 6+ enables exciting multimedia for feature phones (whitepaper). Website, 2008.

[92] Tomas Nordström and Bertil Svensson. Using and designing massively parallel computers for artificial neural neural networks. *Journal of Parallel and Distributed Computing*, 14(3):260–285, 1992.

[93] Dimitre Novatchev. Functional programming in XSLT using the FXSL library. In *Extreme Markup Languages*, 2003.

[94] Jukka K. Nurminen. Using software complexity measures to analyze algorithms – an experiment with the shortest-paths algorithms. In *Computers & Operations Research*, pages 1121–1134. Elsevier Science, 2003.

[95] Ivan Perelomov, Arnulfo P. Azcarraga, Jonathan Tan, and Tat Seng Chua. Using structured self-organizing maps in news integration websites. In *11th International World Wide Web Conference (WWW 2002)*, 2002.

[96] Rolf Pfeifer and Christian Scheier. *Understanding intelligence.* MIT Press, Cambridge, MA, USA, 2001.

[97] Christopher Pohl, Marc Franzmeier, Mario Porrmann, and Ulrich Rückert. gNBX reconfigurable hardware acceleration of self-organizing maps. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT 2004)*, pages 97–104, Brisbane, Australia, December 2004.

[98] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 6th edition, April 2004.

[99] Anand Rajaraman. The story behind google's crawler upgrade, 2008.

[100] G.K. Rauwerda, P.M. Heysters, and G.J.M. Smit. Towards software defined radios using coarse-grained reconfigurable hardware. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):3–13, Jan. 2008.

[101] H. Ritter and K. Schulten. Kohonen's self-organizing maps: Exploring their computational capabilities. In *Proceedings of the International Conference on Neural Networks (ICNN 1988)*, volume 1, pages 109–116, New York, 1988. (San Diego 1988), IEEE.

[102] Gerard Salton. *Automatic Text Processing – The Transformation, Analysis, and Retrieval of Information by Computer.* Addison–Wesley, 1989.

[103] Erich Schikuta and Claus Weidmann. Data parallel simulation of self-organizing maps on hypercube architectures. In *Proceedings of the Workshop on Self-Organizing Maps (WSOM 1997)*, pages 142–147, Espoo, Finland, 1997. Helsinki University of Technology, Neural Networks Research Centre.

[104] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 20 edition, 1994.

[105] Abigail J. Sellen and Richard H. R. Harper. *The Myth of the Paperless Office*. The MIT Press, 2001.

[106] Aspex Semiconductor. Programmers Reference Guide for Activate Release 2.3.0 (Document reference SP230-APRG.doc), 2005.

[107] A. Skupin. A cartographic approach to visualizing conference abstracts. In *IEEE Computer Graphics and Applications*, pages 50–58, 2002.

[108] G. J. M. Smit, A. B. J. Kokkeler, P. T. Wolkotte, and M. D. van de Burgwal. Multi-core architectures and streaming applications. In I. Mandoiu and A. Kennings, editors, *Proceedings of the Tenth International Workshop on System-Level Interconnect Prediction (SLIP 2008), Newcastle, UK*, pages 35–42, New York, NY, USA, April 2008. ACM.

[109] Gerald Jay Sussman and Guy L. Steel Jr. The first report on Scheme revisited. In *Higher-Order and Symbolic Computation*, page 399404, 1998.

[110] T. Sziranyi, Josiane Zerubia, Laszlo Czuni, David Geldreich, and Zoltan Kato. Image segmentation using Markov random field model in fully parallel cellular network architectures. *Real-Time Imaging*, 6:195–221, 2000.

[111] Radoslaw Szymanek, Francky Catthoor, and Krzysztof Kuchcinski. Time-energy design space exploration for multi-layer memory architectures. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10318, Washington, DC, USA, 2004. IEEE Computer Society.

[112] Jeroen Teitsma. Object classification in colour document images with associative array technology. Master's thesis, Twente University, 1999.

[113] D. Thomson. *J: The Natural Language for Analytic Computing*. Research Studies Pre, 2001.

[114] Michael Trott. *The Mathematica Guidebook: Programming*. Springer, 2004.

[115] Edward Tufte. *Envisioning Information*. Graphics Press USA, 1990.

[116] Alfred Ultsch. U*-matrix: a tool to visualize clusters in high dimensional data. Technical Report 36, Philipps-University Marburg, Germany, 2003.

[117] Alfred Ultsch and Dieter Korus. Automatic acquisition of symbolic knowledge from subsymbolic neural networks. In *Proceedings of the 3rd European Congress on Intelligent Techniques and Soft Computing (EUFIT 1995)*, volume I, pages 326–331, 1995.

[118] Anneke Jansen van de Vrie. Cartography/GIS for large document spaces: a case in visualisation and navigation. Master's thesis, Twente University, 2006.

[119] H.W. Van Dijk, H.J. Sips, and E.F. Deprettere. Context-aware process networks. In *IEEE International Conference on Application-specific systems, architectures, and processors (ASAP 2003)*, pages 6–16, The Hague, the Netherlands, December 2003.

[120] Leroy van Engelen. Massively parallel quantization implementation using simulated annealing. Master's thesis, Twente University, 2006.

[121] Colin Ware. *Information Visualization, Second Edition: Perception for Design (Interactive Technologies)*. Morgan Kaufmann, April 2004.

[122] Guido Westenberg. Evolutionary design, tools en toepassingen [dutch]. Master's thesis, Twente University, 1990.

[123] Maarten Wiggers, Marco Bekooij, and Gerard J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of the Design Automation Conference (DAC 2007)*, pages 658–663, 2007.

[124] C.-H. Wu, R. E. Hodges, and C. J. Wang. Parallelizing the self-organizing feature map on multiprocessor systems. *Parallel Computing*, 17(6-7):821–832, 1991.

LIST OF PUBLICATIONS

[P1] Jan W.M. Jacobs, W. Bond, R. Pouls, and Gerard J.M. Smit. High volume colour image processing with massively parallel embedded processors. In *Proceedings of Parallel Computing (ParCo 2005)*, pages 583–590, 2005.

[P2] Jan W. M. Jacobs, Rui Dai, and Gerard J. M. Smit. Mining dynamic document spaces with massively parallel embedded processors. In *Proceedings of the International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS)*, pages 69–78, 2006.

[P3] Jan W. M. Jacobs, Leroy van Engelen, Jan Kuper, and Gerard J. M. Smit. Image quantisation on a massively parallel embedded processor. In *Proceedings of the International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS)*, pages 139–148, 2007.

[P4] Jan W. M. Jacobs, Leroy van Engelen, Rui Dai, Jan Kuper, and Gerard J. M. Smit. IRIS: a firmware design methodology for SIMD architectures. In *Euromicro Conference on Digital System Design*, pages 609–617, 2008.

[P5] Jan W. M. Jacobs and Roger J. H. Hacking. An integrated microprogram development methodology based on APL. In *Proceedings of the APL 1987 Conference*, pages 323–328, 1987.

[P6] Jan W.M. Jacobs. Developing a raster detector system with the J array processing language. *Computing & Control Engineering Journal*, 13(6):299–304, December 2002.

[P7] Samuel Driessen, Jan W. M. Jacobs, and Wolf Huijsen. Combining query and visual search for knowledge mapping. In *Information Visualization (IV 2006)*, pages 216–224, 2006.

[P8] Wolf Huijsen, Samuel Driessen, and Jan W. M. Jacobs. Explicit conceptualizations for knowledge mapping. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS 2004)*, pages 231–236, 2004.

[P9] Gerard J. M. Smit, A. B. J. Kokkeler, G. K. Rauwerda, and Jan W. M. Jacobs. *"State of the Art Massively Parallel Embedded Architectures", submitted as bookchapter for Model-Based Design of Heterogeneous Systems, edited by Pieter Mosterman and Gabriela Nicolescu.* 2008.