TR diss
1847

Low-level Image Processing Architectures

Compared For Some
Non-linear Recursive Neighbourhood Operations

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft, op gezag van de Rector Magnificus, prof. drs. P.A. Schenck, in het openbaar te verdedigen ten overstaan van een commissie aangewezen door het College van Dekanen, op dinsdag 9 oktober 1990 te 14.00 uur.

door

Erwin Ronald Komen

geboren te Utrecht,
natuurkundig ingenieur

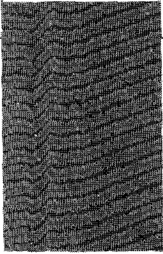
Dit proefschrift is goedgekeurd door de promotor
prof. dr. I.T.Young.

Dr. R.P.W.Duin heeft als toegevoegd promotor in hoge mate bijgedragen aan het tot-stand komen van het proefschrift.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Komen, Erwin Ronald

Low-level image processing architectures : compared for
some non-linear recursive neighbourhood operations /
Erwin Ronald Komen. - [S.l. : s.n.]. - Ill.
Thesis Delft. - With ref. - With summary in Dutch.
ISBN 90-9003713-6
SISO 520.6 UDC 519.72(043.3)
Subject headings: image processing / computer architectures /
algorithms.



Summary

A method is developed for the comparison of computer architectures for image processing which have been designed to work for low-level operations (i.e. image to image transformations). On the basis of features found in existing and published architectures, a division is made between three principally different groups: square processor arrays, linear processor arrays and pipelines. Low-level image processing is split into several categories of operations, distinguished by the amount and type of parallelism that these operations can handle. The comparison between the architecture groups is done for most of the operation groups.

From these groups, the global and object operations are time-consuming because they require much data to be fetched from large distances for the calculation of an output pixel value. A lot of these global and object operations, however, can be expressed by the recursive neighbourhood operations. These are more local oriented, and thus seem to be better suited for the investigated architectures. The development of a theoretical framework for recursive neighbourhood operations is initiated. The usage of these operations for image processing is demonstrated, and it is shown how they can be implemented on the architecture groups. The performance of the architecture groups is then investigated for six non-linear recursive neighbourhood operations.

Concerning the performance of the investigated architecture groups for low-level image processing in general, our results indicate that the linear processor array offers the best overall speed/efficiency combination.

Table of contents

Summary **i**

Table of contents **iii**

List of Symbols **vii**

Comparison of architectures vii

Recursive Neighourhood Operations vii

Mathematical conventions viii

1 Introduction **1**

2 Image processing architectures **5**

2.1 Architecture classification features 5

2.1.1 The stream-taxonomy 5

2.1.2 Local autonomy 7

2.1.3 Shared memory 7

2.1.4 Parallelism taxonomies 8

2.1.5 Connectivity features 9

2.1.6 The Erlangen classification scheme 9

2.1.7 Classification based on the general purpose 10

2.1.8 Classification based on topology 10

2.1.9 Classification feature selection 11

2.2 Grouping of architectures 13

2.2.1 Cellular logic grouping 13

2.2.2 Lougheeds taxonomy 13

2.2.3 Resulting architecture grouping 14

2.3 Principles of the architecture groups 14

2.3.1 Square Processor Array 15

2.3.2 Linear Processor Array 17

2.3.3 Pipelined Processors 19

2.4 Existing low-level image processing architectures 20

2.4.1 The Illinois pattern recognition computer (ILLIAC III) 21

2.4.2 The Massively Parallel Processor (MPP) 21

2.4.3 The Distributed Array Processor (DAP) 21

2.4.4 The UCL Cellular Logic Image Processor-4 (CLIP4) 22

2.4.5 The Geometric Arithmetic Parallel Processor (GAPP) 22

2.4.6 The Contents Addressable Array Parallel Processor (CAAPP) 23

2.4.7 The AIS-5000 from Applied Intelligent Systems 24

2.4.8 The PICAP3 24

2.4.9 The Cyto-High Speed System 25

2.4.10 The Delft Image Processor (DIP) 26

2.4.11 The Cellular Logic Operations chip (CLO) 26

2.5 Proposed and novel low-level image processing architectures 27
2.5.1 The BASE 27
2.5.2 The GEC Rectangular Image and Data computer (GRID) 27
2.5.3 Mesh with Multiple Broadcast 28
2.5.4 Polymorphic Torus Architecture 28
2.5.5 The Digital Optical Cellular Image Processor (DOCIP) 28
2.5.6 Scan Line Array Processor 28
2.5.7 Cellular Logic Image Processor - 7 29
2.5.8 SYMPATI-2 29
2.5.9 The Morphic Image Transform Engine (MITE) 30
2.5.10 The Cellular Logic Processing Element-VLSI (CLPE) 30
2.5.11 The Pipelined Image Processing Engine (PIPE) 31
2.5.12 The Pyramid Architecture for Parallel Image Analysis (PAPIA) 31
2.5.13 The NEC IMage Pipelined Processor 32
2.6 Evaluation on the use of the classification features 33

3 The road to a good comparison 35

3.1 Different approaches to a good comparison 35
3.1.1 Task analysis 35
3.1.2 Comparison on the basis of operations 37
3.2 Performance criteria for the comparison 41
3.2.1 Processing time 41
3.2.2 The number of processing elements 42
3.2.3 The size of the data 42
3.2.4 Cost of the architecture 42
3.2.5 Data I/O 42
3.2.6 Flexibility in image size 43
3.2.7 Flexibility in neighbourhoodsize and shape 43
3.2.8 Programmability 44
3.3 Figures of merit 44
3.4 Suggestion for a good comparison method 46

4 Comparing image processing architectures 49

4.1 Overhead 49
4.2 Data input 50
4.3 Flexibility in image size 51
4.4 Flexibility in neighbourhood size and shape 52
4.5 Programmability 53
4.6 Point operations 54
4.6.1 The use of grey value PEs 54
4.6.2 The use of bit serial PEs 55
4.6.3 Conclusion 57
4.7 Local neighbourhood operations 58
4.7.1 Processing time for the SPA 59
4.7.2 Processing time for the LPA 60
4.7.3 Processing time for the PL 60
4.7.4 Verification of the theory 62
4.7.5 Comparison of SPA, LPA and PL for LNOs 64
4.8 Object operations 71

4.9 Recursive neighbourhood operations 72
4.10 Global operations 72
4.11 Geometric Operations 73
4.12 Statistical Scalar Operations 75
4.13 Statistical Vector Operations 75
4.14 Conclusion 76

5 Non-linear recursive neighbourhood operations 79

5.1 Mathematical formulation for low-level image processing operations 80
5.1.1 Definitions 81
5.1.2 Spatial versus temporal recursion 82
5.1.3 Seeing an RNO as an eigenfunction problem 83
5.1.4 Edge handling 83
5.1.5 Constraint specification 84
5.2 The relation between RNOs, local, object and global operations 84
5.2.1 The relation between RNOs and LNOs 85
5.2.2 The relation between RNOs and GlOs 86
5.2.3 The relation between RNOs and OOs 89
5.2.4 Conclusions 97
5.3 Updating methods 97
5.3.1 Deterministic updating methods 98
5.3.2 Data dependent updating methods 100
5.3.3 Stochastic updating techniques 102
5.3.4 Updating methods revisited 103
5.4 Theoretical considerations on fixed points from RNOs 104
5.4.1 Fixed point stability and uniqueness 104
5.4.2 Cyclicity and RNOs 105
5.4.3 Criteria for RNOs concerning the number of fixed points 106
5.4.4 The number of fixed points for some operations 113
5.5 Classes of RNOs and object operations 114
5.5.1 Linear versus non-linear RNOs 114
5.5.2 Cyclical versus non-cyclical RNOs 115
5.5.3 Morphological RNOs 115
5.5.4 Recursive order statistics filters 116
5.5.5 Recursive stack filters 117
5.5.6 Relation between non-linear RNO classes 118
5.6 Conclusions 119

6 The use of RNOs for image processing 121

6.1 The model problem 121
6.2 Recursive morphological RNOs 122
6.3 Recursive stack filters 122
6.3.1 Recursive median 123
6.3.2 Unique median roots 124
6.4 The smallest enclosing regular polygon 125
6.5 Distance transforms 126
6.5.1 Normal distance transform 126
6.5.2 Signed Euclidean distance transform 127

6.5.3 Constraint distance transform 127
6.5.4 Grey weighted distance transform 128
6.6 Other non-linear RNOs 129
6.6.1 Dithering algorithms 129
6.6.2 Envelope estimation 130
6.6.3 Labelling algorithms 130
6.7 Conclusions 131
7 Implementation of RNOs on the architecture groups 133
7.1 Deterministic updating methods 133
7.1.1 Single processor 133
7.1.2 Pipelined processors 135
7.1.3 Linear Processor Array 135
7.1.4 Square Processor array 135
7.2 Data dependent updating methods 135
7.3 Stochastic updating methods 136
7.4 Combinations 138
7.5 Conclusions 139
8 Experiments with RNOs 141
8.1 Characteristics of the updating method implementations 141
8.2 Definition of the RNOs and their input images for the experiments 142
8.2.1 Object selection 143
8.2.2 Smallest enclosing regular polygon 143
8.2.3 Normal distance transform 144
8.2.4 Grey weighted distance transform 145
8.2.5 Median and minimum median root 146
8.2.6 Poisson equation 147
8.3 Performance of the updating methods for the RNOs 147
8.3.1 Deterministic and data dependent updating methods 148
8.3.2 Asynchronous updating method 149
8.3.3 The combined data dependent and deterministic updating 152
8.3.4 The efficiency of updating methods revisited 155
8.4 Performance of the architecture groups for the RNOs 156
8.5 Conclusions 159
9 Conclusions 161
9.1 Evaluation of results 161
9.2 Perspective 163
10 Bibliography 165
Acknowledgements I
Samenvatting III
Curriculum Vitae V
Appendix VII
A.1 Data VII
Index XI
Author's Index XVII

List of Symbols

Comparison of architectures

Algorithm length	l
Clock cycle time	t_{clock}
average	$t_{average}$
host	t_{Host}
I/O	$t_{I/O}$
maximum variation	$t_{variable}$
required	$t_c^{required}$
total	t_{point}
Connectivity	
neighourhood	C_N
recursive neighbour	C_R
Efficiency	E
Image width or height	N
Number of	
bits per pixel	B
clock cycles	n
processors	P
Overhead	
instruction	O_{instr}
I/O	$O_{I/O}$
programming	O_{progr}
scanning	O_{scan}
Parallelism	
neighbourhood	P_N
operator	P_O
pixel-bit	P_P or b
recursive neighbour	P_R
spatial	P_S
Polygon angle of a regular polygon with P sides	α_P
Processing time	t
Quality factor	Q
Relative Variable Processing Time	RVPT

Recursive Neighourhood Operations

Cardinality of S	$\#S$
Charge density (coulomb/m^3)	$\rho_{\vec{p}}$
Coefficient set used in LNOs and RNOs	c
Coordinates	x,y
Image	
input	X
output	Y
Path	π
Pathlength	π_l
Permittivity of vacuum	ε_0
Pointset	S
Position in image	
absolute	$\vec{p}$
relative	$\vec{q}$
Potential field (volt)	V
Random value uniformly distributed between [-1,1]	ρ
Rank (using an IMF)	r
Set of pos. integers excluding 0	N^+
Updating Effort	E_u
Update fraction	Q

Mathematical conventions

Absolute value of $X_{\vec{p}}$	$\lvert X_{\vec{p}} \rvert$
Average	$\overline{Q}$
Closest integer to the real value A	
higher	$\lceil A \rceil$
lower	$\lfloor A \rfloor$
Erosion operator	
binary	Θ
grey scale	Θ_g
Laplacian	∇^2
Length of vector $\vec{p}$	$\lVert \vec{p} \rVert$
Logical inversion of B	$\neg B$
Opening of $(\ldots)$ by S	$(\ldots)_S$
Separation brackets	
$(\ldots)$, $\{\ldots\}$, $[\ldots]$	
Set brackets	$\{\ldots\}$
Shortcuts	
$\vec{q}_1 \in S \wedge \vec{q}_2 \in S$	$\vec{q}_{1,2} \in S$
Value of $Y_{\vec{p}}$ on the kth update	$Y_{\vec{p}}^{(k)}$

Chapter

1 Introduction

The demands for high speed image processing capabilities keep rising: larger images have to be processed (satellite data of 4096*4096 and 3D images of about 256^3 pixels), and results have to be delivered faster to facilitate robot vision and interactive image processing (Lindskog 1988; Preston 1989). An increase in image processing performance on the other hand, may open new applications. The demands can not simply be met by an increase in clock frequency of one single general purpose processor. Therefore, the speed of image processing has to be increased by efficiently using more processors in parallel.

A distinction has to be made between the various layers in image processing: low-level, intermediate-level and high-level (Danielsson and Levialdi1981). The difference in these levels can be described by the difference in the data types handled at each level. *Low-level* image processing operations transform images into other images. This group consists of operations such as: noise filtering, edge detection, and geometric transformations. The images in this level may be coded without loss of data, i.e. with run coding (also called pxy tables) or chain coding etc. *Intermediate-level* image processing operations transform images into some sort of symbolic description. Examples of such operations are: graph building, region analysis and for instance the Ronse & Devijver method to construct a structural description of a scene out of runcoded images (Ronse and Devijver 1984). *High-level* image processing operates on the symbolic descriptions to do such tasks as: recognition, decision and reasoning.

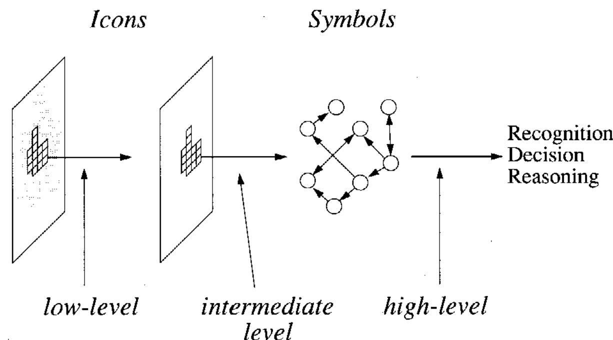


Figure 1.1 Different levels in image processing operations.

Other terminology can also be used to distinguish the three levels in image processing. The low, intermediate and high-level are then called the *iconic*, *iconic to symbolic*, and *symbolic* layer (Fountain 1986). This terminology clearly identifies the different data representations in these levels.

Because of the evident difference between the functionality of the data structures which are handled in the various image processing levels, two basically different approaches are taken in the design of image processing architectures. The *first* approach tries to find one

architecture which is optimal for all three levels of image processing, i.e. to all the functional descriptions in these levels. In the *second* approach, architectures are designed which can optimally process the data structure within one level. Optimized interfaces between the levels can also be designed.

A disadvantage of the second approach is the necessity of the optimal functioning interfaces between the levels. The first approach on the other hand doesn't allow for fine tuning towards the functional descriptions used in the different levels. We prefer the second approach towards image processing architecture optimization, because we feel that the fine tuning of the architectures towards the data types they have to handle (i.e. functional descriptions) will bear fruit in the better overall performance of an image processing system.

Specifically, the goal of this research is the following:

"Investigation of the performance of architectures for doing low-level image processing with the intention to use the results for a further improvement of the architectures".

To compare existing and suggested architectures, our strategy starts in Chapter 2 with gaining insight into the differences between these architectures. Features are therefore identified, on whose basis the architecture can be classified. Three architecture *groups*, which have some common and some differing architectural features are then selected for a closer study. The emphasis of these three architecture groups is on *local neighbourhood* processing because images themselves are strongly correlated in local regions. A strategy for the comparison between the members of the chosen architecture groups is derived in Chapter 3. The comparison could either be done by measuring the performance of the architecture groups on some selected low-level image processing *tasks* or on *operations*. We will focus on the operations analysis approach. The low-level image processing operations are first discerned by identifying a set of operation classification features, e.g. the types of parallelism in the operation etc. Second, operation *groups* are formed on the basis of corresponding and differing operation classification features. The architecture groups are then compared as to performance on processing members of the operation groups in Chapter 4. The operation and architecture classification features will also interact to some extent with the performance comparison.

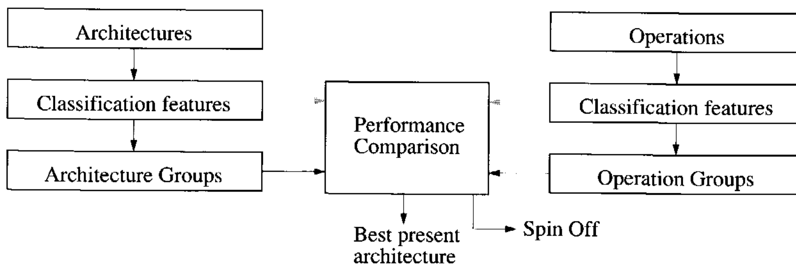


Figure 1.2 The comparison strategy.

At this point it is noted that comparisons for *object* and *global* operations as to performance for *low-level* image processing architectures have been found infrequently in the literature (see Section 4.8 and Section 4.10). These operations require the collection of data over larger areas than those usually found in local neighbourhood operations. Two of the uninvestigated operation groups (i.e. object and global operations) contain a number of

members which can be built up by a third group called Recursive Neighbourhood Operations[1] (RNOs in short). These operations are by their nature locally in a spatial sense. Because of the apparent possibilities offered by these operations, the use of the selected architecture groups for performing RNOs is also investigated in this thesis.

This investigation starts in Chapter 5 by giving a clear mathematical formulation for RNOs as well as the other operation groups defined in Chapter 3. It is then shown that some object and global operations can be derived from RNOs. A distinction is made between several classes of RNOs. Because the sequence in which image points are calculated for RNOs is correlated to the speed with which the operations can be done, different *updating methods* are introduced. One of the problems with RNOs is, that not all of the operations in this group can be applied to images in such a way that a so called invariant *fixed point* or *root* image results. In fact, an RNO may have none, one or more than one fixed point image(s). A theory is therefore developed which can be used to find out if an RNO has fixed point images by looking at the structure of the operation and the distribution of the neighbourhood points used in the RNO. Furthermore, some classes of RNOs are distinguished.

Chapter 6 relates the theoretically formulated RNOs to their practical usage in image processing. Several examples illustrate this.

The performance of the architecture groups on calculating RNOs depends on the updating methods with which they work. It is therefore shown in Chapter 7 what updating methods can be implemented using the architectures from the selected groups. Experiments are presented in Chapter 8 to find out what the performance is of updating methods for a specific set of RNOs. For every RNO to be tested, two different input images are used in the experiment; one for an "average" case and one for a "difficult" case. This not only serves to find out the difference between difficult and average case performance, but also serves to find out if the rank order of the updating methods depends on the images used. The performance of the architecture groups for calculating RNOs is measured for the investigated RNOs and in relation to the updating methods which the architecture can perform. Conclusions for the efficiency and speed with which members of the selected architecture group can calculate RNOs are drawn on the basis of these measurements.

The overall conclusions on the performance of low-level image processing architectures are given in Chapter 9. This chapter also includes recommendations for future research and lists open issues.

1. An RNO can for the moment be considered as a local neighbourhood operation, iteratively and/or sequentially applied on an image, possibly until no pixel values change.

Chapter

2 Image processing architectures

The comparison between low-level image processing architectures requires insight into their principles. Architectures are distinguished on the basis of *features*. It is discussed in Section 2.1 which selection of features is sufficient to describe present and future architectures. The number of architectures that can be described using these features is very large. A good comparison is therefore better done between a small set of well defined *architecture groups*. The ordering of architectures into groups is discussed in Section 2.2, and is used throughout this thesis.

An introduction into the way architectures from these chosen groups work is then given in Section 2.3. An overview of some existing architectures and of some interesting proposals in Section 2.4 and Section 2.5 serve as illustrations to the low level image processing architectures which are the subject of this thesis. Some of these architectures are already referred to in the first paragraphs of this chapter. The chapter ends with a review in Section 2.6 on the use of the classification features and groups for existing and novel architectures.

2.1 Architecture classification features

The classification features on whose basis low-level image processing architectures are distinguished should fulfil the following criteria:

- *Sufficient*: they cover at least all *existing* low level image processing architectures.
- *Low-level*: they emphasize low-level image processing, as opposed to image processing in general. Low-level image processing consists of those image processing operations which take one or more *images* as input, and produce an *image* as output (Danielsson and Levialdi1981; Weems et al. 1989b).
- *Significant*: they represent a meaningful difference between architectures.

Some contemporary classification schemes for general computer architectures are discussed by Händler (Händler 1977). Although his emphasis is not on low-level image processing, a lot can be learned from him. Some traditional classification features mentioned by him and others, and the drawbacks to using them for the classification of low level image processing architectures will be discussed in the coming sections. This will result in a set of features on the basis of which low-level image processing architectures can then be described.

2.1.1 The stream-taxonomy

Flynn categorizes computer architectures according to the "stream taxonomy" (Flynn 1972). By noting the magnitude (in space or time) of interactions of the instruction and data streams, four categories are defined by him:

- Single Instruction stream, Single Data stream (SISD)
- Single Instruction stream, Multiple Data stream (SIMD)

- Multiple Instruction stream, Single Data stream (MISD)
- Multiple Instruction stream, Multiple Data stream (MIMD)

The flow of the instruction streams between a *Control Unit* (CU) and a *Processing Element* (PE) and the data streams between PEs are shown in Figure 2.1. There are several proposals for extension of the stream model, to incorporate the specifics of new parallel architectures. One of them is the multiple SIMD (MSIMD) type (Cantoni 1986). This is used in connection with a pyramid, where each level of the pyramid works in SIMD mode, but each level can be individually programmed and controlled. A novel, experimental architecture named OPSILA, is able to switch between normal SIMD mode and a so-called SPMD mode (Duclos et al. 1988). This is defined as a *single program multiple datastream* mode, where each processor executes a copy of the same program, on a part of the data.

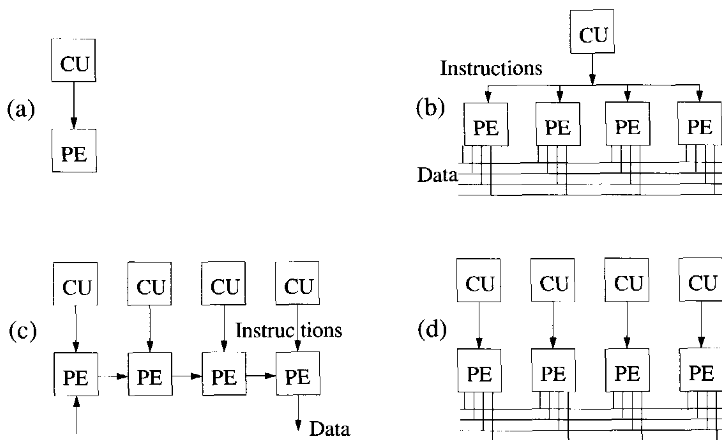


Figure 2.1 The classical stream taxonomy: (a) SISD, (b) SIMD, (c) MISD and (d) MIMD.

The use of the Flynn taxonomy for the classification of low-level image processing architectures gives some problems:

1. *Topology*. The way in which PEs exchange their data (topology) is not given. In image processing, however, a distinction is made between mesh connected PE configurations (see the CLIP and MPP architectures, Section 2.4) and, for example, N-cube connected ones (Connection Machine, Section 2.4).
2. *Data I/O*. The problem of I/O is left independent of the architecture. In image processing, however, I/O bandwidth considerations can be used to make a distinction between architectures (Duin and Jonker 1986).
3. *Autonomy*. It is argued that for image processing, there can be many stages of autonomy between the strict SIMD and the strict MIMD modes (Fountain 1987). Low level image processing itself is usually in SIMD mode.
4. *MISD*. Some people disagree about the use of the MISD classification for pipelined image processors (Gerritsen 1982; Lindskog 1988).

The first limitation can be overcome by classifying the architecture topologies. The I/O problem is treated in very few classification schemes. It will be taken into account for the comparison of architectures in Chapter 4. For the third point, those suggestions of Fountain

which represent existing image processing architectures are adopted. Concerning the fourth point, we refer to the author of the stream classification model. According to Flynn's definition the MISD class incorporates the use of *"...multiple instruction streams on a single sequence of data..."* (Flynn 1972). A pipeline is a set of PEs through which one datastream is fed and processed. In our opinion each pipeline element may be individually programmed and controlled, so that we will consider pipelines as belonging to the MISD class.

2.1.2 Local autonomy

Between the strictly centralized control of SIMD computers and the strictly decentralized control of MIMD computers increasing levels of local autonomy are possible (Fountain 1988b). The processing elements of an SIMD computer, for instance, can use their local data for local control purposes. The following types of local autonomy for SIMD and MISD computers are suggested by Fountain and others (Lindskog 1988; Cypher and Sanz 1989):

- *Local activity control.* A mask bit in a processing element indicates whether the PE may "join" the calculations or not.
- *Local data addressing control.* The address of the source and/or destination data address can be locally determined. This is usually done by allowing local data from a PE to serve as index in a global provided base address.
- *Local function control.* The local data (or part of it) is used to select the function which is executed in a PE.
- *Local connectivity control.* The way in which a PE connects to its neighbouring PEs can be locally controlled.
- *Local algorithm control.* In this level of autonomy, each PE can be loaded with a different program, but the sequencing is still global. Pipelines always have local algorithm control.
- *Local sequencing control.* On top of the local algorithm control, the sequencing of the programs in a PE can also be locally done. The PEs are no longer synchronously connected, so that handshaking is necessary.
- *Local partitioning control.* The PEs are given the capability to partition their programs (parts of them) over other PEs.

As will be clear later on in this chapter, when existing and novel architectures are described, there is indeed a trend in equipping low-level image processing architectures with local autonomy. Local autonomy will therefore be regarded as one of the classification features with which architectures are described.

2.1.3 Shared memory

A feature used in describing parallel architectures is whether or not the machine has a single shared memory (Cypher and Sanz 1989). In a shared memory computer, all of the processors read from and write to a single common memory. The NEC IMPP dataflow image processor can, when used in combination with the MAGIC chip, be regarded as a parallel low-level image processing architecture with one single shared memory (see Section 2.5.13; Dekker et al. 1987). Other low-level image processing architectures don't have shared memory to our knowledge, so that this feature does not represent a meaningful dif-

ference, and will not be used in the classification.

2.1.4 Parallelism taxonomies

Image processing architectures can also be distinguished by looking at the parallelisms involved. In 1977 the introduction of a *degree of parallelism* based on the number of bits that can be processed in one clock cycle was proposed (Feng 1977). In 1981 it was argued, that there are actually four orthogonal dimensions in parallelism (Danielsson and Levialdi1981). These dimensions, extended with two others (recursive neighbourhood and image parallelism), are listed here:

- *Operation.* Operations are executed in parallel, as is the case in a pipeline or with some kinds of pyramids.
- *Spatial.* Several points in the destination image are calculated in parallel, as is the case in linear and square processor arrays.
- *Neighbourhood.* This type of parallelism says how many pixels of the local neighbourhood of a source image can *concurrently* (within one clock-cycle) be used for the calculation of one destination pixel. Pipelines - and sometimes processor arrays - have a neighbourhood parallelism which is greater than one. If an architecture has possible access to its four nearest neighbours (i.e. it can access each of these neighbours within one clock-cycle), but can only get their values one at a time, than the neighbourhood parallelism is 1 according to our definition.
- *Recursive neighbourhood.* This parallelism says how many destination pixel values which have been calculated in the current updating pass[1] across the image can *concurrently* (that is, within one clock-cycle) be used for the calculation of one destination pixel.
- *Pixel-bit.* Some or all of the bits which make up the source pixels are used concurrently in the calculation of one destination pixel. This type of parallelism is used by single-processor general purpose computers, but also in some pipelines. Processors which do *not* have this type of parallelism are called bit-serial.
- *Image.* Different images are used in parallel. Most image processing architectures allow direct treatment of dyadic operations. Sometimes more than two input images can be treated in parallel (for instance the AIS-5000, see Section 2.4.7).

Danielsson and Levialdi suggest using the product of the operator, spatial, neighbourhood and pixel-bit parallelisms to define a *degree* of parallelism. Feng uses two of the six parallelisms as coordinates in a diagram. Händler describes how he classifies essentially different machines on the same Feng-coordinates, so that Feng's scheme is not sufficient. Lougheed uses the operator and spatial parallelism for the classification of image processing architectures (Lougheed 1987). Within spatial parallelism he distinguishes between architectures which treat *all* image points, a *subset* of them, or just *one* image point at once. In operator parallelism he distinguishes between pipelines which consist of one stage and pipelines consisting of many stages.

We propose to use five of the six types of parallelism (i.e. *operator*, *spatial*, *neighbourhood*, *recursive neighbourhood* and *pixel-bit*) as five separate features in the low-level image processing architecture classification. They identify meaningful differences between ar-

1. The notion of a 'current updating pass' is only defined for deterministic updating methods as described in Section 5.3.1.

chitectures, and none of the parallelisms is strange to low-level image processing. The discrimination between architectures provided by the feature *image parallelism* is not significant enough. Most of the low-level image processing architectures allow image processing on up to two input images.

2.1.5 Connectivity features

Although neighbourhood parallelism tells how many points of the local neighbourhood around the source pixel can *concurrently* be used, it does not say how much neighbourhood points can *in principle* be reached within one clock-cycle. The feature which describes this will be called *connectivity*, as it actually counts the number of neighbourhood lines coming into a processing element. We suggest the use of the following two connectivities:

- *Neighbourhood connectivity*. This type of connectivity indicates how many source pixels of the local neighbourhood can *in principle* be used for the calculation of one destination pixel.
- *Recursive neighbourhood connectivity*. How many destination pixel values which have been calculated in the current updating pass can *in principle* be used for the calculation of one destination pixel. The recursive neighbourhood connectivity actually counts the number of lines coming from the output of a processing element which are led back to its input.

The use of these connectivities helps to distinguish between architectures where the designers chose for fast accessibility to a few neighbourhood pixels concurrently or for more neighbourhood pixels sequentially. Connectivities are therefore closely related to their respective parallelisms.

2.1.6 The Erlangen classification scheme

An extension to the parallelism classification schemes is the inclusion of pipelining. This was done by Händler in his ECS (Erlangen Classification Scheme; Händler 1977). In this classification scheme a distinction is made between parallelism and pipelining. If the common denominator between the two is termed *concurrency*, then parallelism can be described as concurrency in *space*, whereas pipelining is concurrency in *time* (Kogge 1981). The spatial concurrency is with regard to image *data*, and the temporal concurrency is with respect to *instructions*. The ECS is based on the assumption that there are three processing levels, on which both parallelism and pipelining are possible. These levels and their corresponding parallelism and pipelining possibilities are:

- *Program Control Unit* (PCU). Each architecture can contain a number of PCUs, which work on different images in parallel (a form of *Image* parallelism).
  Pipelining on this level is called *macro-pipelining*. One PCU of the architecture can do a task on a data stream, from which the output is written into a memory. The other PCU performs another task, and takes its data from the memory, so that both PCUs are concurrently in time working on their subtasks of the whole problem (a form of *Operator* parallelism).
- *Arithmetic and Logical Unit* (ALU). A set of simple *processing elements* (PEs) governed by one PCU, and working on different data parts constitutes the (*spatial*) parallelism on this level.
  Pipelining here is the use of a number of *function* units which work concurrently on one instruction stream.

- *Elementary Logic Circuit* (ELC). The parallelism on this level is with regard to the number of bits per pixel that can be treated concurrently (*Pixel-bit* parallelism). Pipelining on the ELC level can for instance be done within the arithmetic part of the PE.

The ECS scheme results essentially in a 6-tuple with the three types of parallelism (i.e. image, spatial and pixel-bit) and the three types of pipelining (i.e. operator, function, instruction). We conclude that the emphasis of the ECS *as a whole* is not towards low-level image processing, so that it does not include important things such as Neighbourhood parallelism. Also, the pipelining on the ELC and ALU level is not of much importance to the distinction between low-level image processing architectures. The pipelining on the PCU level is already covered by the operator parallelism of the previous paragraph.

2.1.7 Classification based on the general purpose

Yet another approach was taken by classifying the architectures with two parameters: the number of processors versus the *"general purposeness"* (Cantoni and Levialdi 1983). The term "general purposeness" however, cannot be defined very accurately and objectively. Furthermore, in simply using the number of processors as one single parameter, no distinction is made between architectures having the same number of processors, which are combined in a completely different way. We will therefore not use either of these features in the classification of architectures.

2.1.8 Classification based on topology

An approach towards a comparison between "parallel architectures for perceptual tasks" has been given (Uhr 1988). Unfortunately, most of his results are only interesting for high-level image processing, i.e. for 'perceptual tasks'. In his comparison, the architectures are grouped according to their topology. The topology of an architecture determines how many steps are necessary to transport data from one PE to any other PE in the machine. The properties of interconnection schemes are analysed by Forshaw (Forshaw 1987). The topologies of Uhr, extended for completeness, are:

- *Stars*. Systems based on a bus which links a number of processors along with resources, including memories, I/O devices etc. (Figure 2.2a).
- *Lines*. This can be a pipeline, where the image data flows from processing element (PE) 1 to 2 etc. to PE L. This could also be a 1-dimensional array, where each PE works on own local data or data from neighbouring PEs (Figure 2.2b).
- *Polygons and Rings*. When the two ends of a line are linked, they form a polygon or ring (Figure 2.2c).
- *Trees*, and *Slightly Augmented Trees*. A tree linked multi-computer has one single root from which successive levels of processors fan out, possibly with a few extra links (Figure 2.2d).
- 2-Dimensional *Arrays / mesh*. Each PE is linked to its 4/6/8 2-dimensional nearest neighbours (Figure 2.2e).
- Binary *N-cubes*. This is an N-dimensional cube with 1 PE at every node (Figure 2.2f).

- *NlogN reconfiguring network-based systems.* N processing elements are linked using LogN banks of switches, allowing almost complete interconnection (Figure 2.2g).
- *Fully.* With full interconnection, every PE can directly obtain data from every other PE (Figure 2.2h).
- *Torus.* Just as a polygon is a line with connected ends, so is a torus a mesh where the right PEs are connected to the left and the top PEs connected to the bottom (Figure 2.2i).
- *Pyramid.* The PEs are organized in a stack of levels. At any level the PEs process directly neighbouring image points, to a father PE in the level above, and to son-PEs in the level below (Figure 2.2j).
- *Data dependent.* A data-dependent topology can be created using a dataflow image processing architecture or using local connectivity control (Dekker et al. 1987; Maresca et al. 1989).

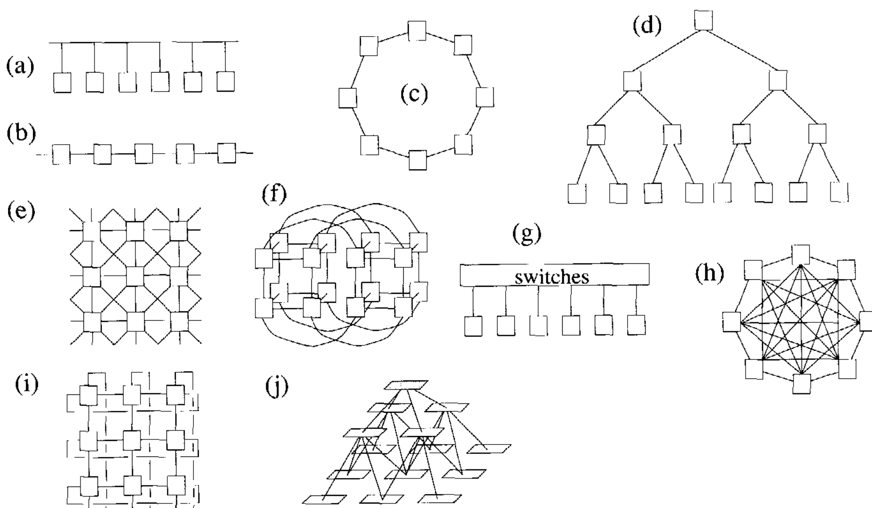


Figure 2.2 Topologies for image processing architectures: (a) star, (b) line, (c) polygon, (d) tree, (e) 8-mesh, (f) N-cube, (g) NlogN, (h) fully (i) torus and (j) pyramid

As topology is an important distinguishing factor for low-level image processing architectures, this will be one of the features on which architectures are classified. Figure 2.2 shows the topologies as discussed above.

2.1.9 Classification feature selection

Now that a broad set of classification features for the discrimination of low-level image processing architectures has been reviewed, a selection of them is taken on the basis of the criteria which were stated at the beginning of Section 2.1. The following features will be used, because they are sufficient to cover all existing architectures, they focus on low-level image processing, and they identify significant differences between the architectures:

1. *Instruction and data streams.* Flynn's classical model will be taken, extended with MSIMD and SPMD. For low-level image processing this will always be SIMD or / MISD, but the level of autonomy may differ.
2. *Autonomy level.* The local autonomy capabilities in a PE, if applicable, are named as follows:

- *Act.* Local activity control
- *Adr.* Local data control
- *Fie.* Local function control
- *Cnc.* Local neighbourhood connectivity control
- *Alg.* Local algorithm control
- *Seq.* Local program sequencing control
- *Prt.* Local program partitioning control

3. *Topology.* The topology in which PEs are connected to one another (see Section 2.1.5): star/bus, line, polygon, tree, pyramid, 4/6/8-mesh, N-cube, NlogN, fully.
4. *Operator parallelism* (written as P_O). The number of operations that are executed in parallel.
5. *Spatial parallelism* (written as P_S). The number of image points that are processed in parallel.
6. *Neighbourhood parallelism* (written as P_N). The number of neighbourhood source pixels that can concurrently be included in the calculation of a destination pixel.
7. *Neighbourhood connectivity* (written as C_N). The number of neighbourhood source pixels that can be reached within one clock cycle.
8. *Recursive neighbour parallelism* (written as P_R). The number of destination pixel values calculated in the current updating pass which can *concurrently* be used for the calculation of one destination pixel.
9. *Recursive neighbour connectivity* (written as C_R). The number of destination pixel values calculated in the current updating pass which can *in principle* be used for the calculation of one destination pixel.
10. *Pixel-bit parallelism* (written as P_P). The number of bits per pixel that can be processed in parallel.

The instruction/data stream feature is incorporated, as it helps to distinguish between the SIMD, MISD and MSIMD low-level image processing architectures. The level of local autonomy is especially helpful in discriminating novel processor arrays with extended capabilities. The topology is also included, as this gives information on the speed that data can be transferred from one point to another point in the image to be processed. Several types of parallelism are also included, as they show how an architecture can tackle the image processing operations (these are treated in Chapter 3). The two connectivities are included to discriminate between architectures which may have very fast access to a number of (recursive) neighbours sequentially. An architecture class is formed by using the ten classification features in a 10-tuple:

class = (Stream, autonomy, topology, P_O,P_S,P_N,C_N,P_R,C_R,P_P)

The classification features will be evaluated at the end of this chapter, after several ex-

isting and novel architectures are described by the 10-tuple, and the way these architectures work has been shown.

2.2 Grouping of architectures

The classification features which are selected in Section 2.1 are sufficient to distinguish between architectures for low-level image processing. If all of the possibilities they offer are considered, they in fact allow for the description of numerous classes. To offer a framework for the comparison between architectures in this thesis, *groups* of architectures will be created and described by the classification features. A group of architectures may have one or more *common* classification features, whereas other features do not matter within that group.

Two basically different suggestions for the grouping of architectures were found in the literature and will be treated. At the conclusion to this section, the groups which are chosen for this low-level image processing architecture comparison will be described.

2.2.1 Cellular logic grouping

Some comparisons have an emphasis on *cellular logic* operations, which take place using the elements of a 3*3 neighbourhood of a binary image (Fountain 1987; Duin and Komen 1989). The philosophy behind this strategy is, that cellular logic operations can be regarded as a basis of the low-level image processing architectures. The common factors in the architectures for these kind of operations (usually called *cellular logic architectures*) are the neighbourhood connectivity which is usually 3*3 (but at least higher than one), and the pixel bit parallelism which is always equal to one. Groups of cellular logic architectures are then made by looking at their topology (mesh, line or pyramid) and their operator parallelism (one or more). This leads to the following groups:

- *Square Processor Array* (SPA). Two-dimensional array of processing elements each working on part of the image data in the SIMD-mode. The neighbouring PEs can exchange data. How the data of a large image is divided up between PEs is not specified.
- *Linear Processor Array* (LPA). One-dimensional array of processing elements each working on part of the image data in the SIMD-mode.
- *Pipe Line* (PL). One-dimensional array of consecutive PEs which may each execute their own program, and through which the image is fed.
- *Pyramid* (PYR). A set of processor arrays stacked on top of one another with sizes going from $\sqrt{P}*\sqrt{P}$ in the bottom layer to 1*1 in the top layer.

These groups are all concerned with the cellular logic operations. However, the groups do not necessarily have to be restricted to a pixel bit parallelism of one, and a maximum neighbourhood parallelism of 3*3. Without changing the group names, they could easily also be used for architectures which are constructed to process grey-values (pixel bit parallelism is 8 or even greater). Their main emphasis is then to *local neighbourhood* processing.

2.2.2 Lougheeds taxonomy

A taxonomy of image architectures specifically for low-level vision was given by Lougheed, and consists of the following architecture *groups* (Lougheed 1987):

- *Full Array.* Square Processor Array, where the image is as large as the array
- *Parallel Subarray.* Square Processor Array which is smaller than the image it has to process
- *Raster Subarray.* PipeLine with only one stage[1]
- *Raster-PipeLine Subarray.* PipeLine with more stages.

Lougheed builds this grouping on the bases of the spatial and operator parallelisms. The first classification feature distinguishes between the processing of *all*, a *subset* or just *one* image point in parallel. The second feature distinguishes between the processing of *one* or *more* operations in parallel.

2.2.3 Resulting architecture grouping

Some of the suggested architecture groups are shown in Table 2.1. Lougheeds architecture grouping scheme distinguishes between an operation parallelism of *one* or *more*, whereas pipelined image processing architectures are nowadays always build with more than one stage. The distinction that he makes between full-size and undersized processor arrays is not convenient in our opinion. Given a processor array of size $\sqrt{P} \cdot \sqrt{P}$, there will always be an image of size $N \cdot N$, where $N > \sqrt{P}$, so that special measures will have to be taken to process such an image.

Table 2.1 Low level image processing architecture groups

	Streams	Autonomy	Topology	P_S	P_O	P_N	C_N	P_R	C_R	P_P
Square PA	SIMD	any	mesh	>1	1	1..9	4..8	0..9	0..8	1..32
Pyramid	SIMD	any	pyramid	>1	1	1..13	9..13	0..13	0..13	1..32
Linear PA	SIMD	any	line	>1	1	1..9	4..8	0..6	0..6	1..32
PipeLine	MISD	Alg	line	1	>1	9	9	0..4	0..4	fixed
Raster SubArray	SISD	Alg	mesh	1	1	9..	9..	0..4	0..4	fixed

The use of the cellular logic grouping scheme tends to a restriction to pixel bit parallelism, which is not desirable. The grouping into SPA, LPA, PL and PYR does not incorporate *all* possible architectures for low-level image processing, but has an emphasis on *local* neighbourhood processing. This fits quite well in the nature of low-level *image* processing operations. An image is a two dimensional data set where the data points have some kind of *spatial* relationship. The emphasis on local neighbourhood operations is therefore not unrealistic. The use of the pyramid architecture group is questionable. Only a few working pyramids are available, and it is still very doubtful whether pyramids offer any real improvement over processor arrays with extended capabilities (Prasanna and Dionisios1989). However, the distinction between SPA, LPA and PL is quite natural, and adapts to the literature on comparisons between low-level image processing architectures (Fountain 1987).

Therefore, in this thesis we will consider the architecture groups SPA, LPA and PL, with features as shown in Table 2.1. We suggest that future comparisons use more or different groupings, according to the available and forthcoming architectures at that time.

2.3 Principles of the architecture groups

As an introduction into the presentation of available architectures, the principles of the

1. A stage in a pipeline consists of those PEs which are operating on the same image point in the same clock-cycle.

architectures in the groups (SPA, LPA and PL) which are described using the classification features from Section 2.1.9 will now be treated in some more detail.

2.3.1 Square Processor Array

A Square Processor Array (SPA) is an array of processing elements (PEs) connected in a 2D grid in which each processor has access to the output values of its 'direct' neighbours (Figure 2.3). Depending upon the number of neighbours which it can directly access, a mesh can have a neighbourhood connectivity of 4, 6 or 8. Some SPAs allow switching between the so-called hexagonal (6-connected) tessellation and the 8-connected tessellation (i.e. the CLIP4, Section 2.4.4, and the BASE, Section 2.5.1). Note that the number of neighbours that can be reached directly (neighbourhood connectivity) is not always the same as the neighbourhood parallelism. Some arrays allow direct access to 8 neighbours, but only one or two at a time can be used in calculations (for instance the GRID, discussed in Section 2.5.2).

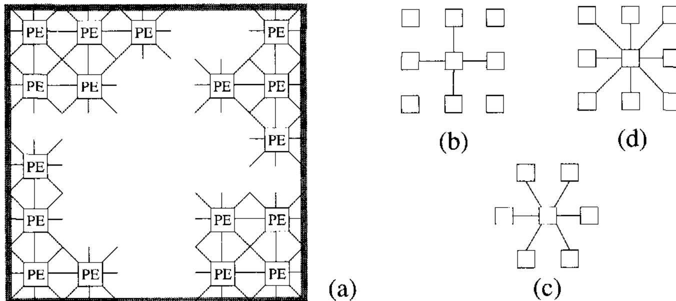


Figure 2.3 (a) Square Processor Array with the following neighbour connections: (b) 4-connected, (c) 6-connected and (c) 8-connected.

Each PE contains its own local memory to store the image pixel value corresponding to its position in the array. The basic form of a bit-serial PE for use in an SPA (8-connected) is shown in Figure 2.4.

The outputs of the 8 neighbouring PEs are fed into this processing element (i.e. the neighbourhood parallelism is 8), together with the value of the central pixel and the value of an additional image to allow for dyadic operations. The value of the carry register can also be used by the PE, so that bit-plane arithmetic is possible; stacks of bitplanes can thus be added, subtracted etc. This basic bit serial PE outputs a result bit into the local memory, and a carry out into the carry register. There is also a separate output shown to the 8 neighbours, so that the recursive neighbour parallelism is 8. If a large array is built, the PEs will in general be designed with less possibilities than the PEs of a small array. No SPAs with fully programmable PEs have come to our attention. An SPA equipped with the basic PE shown in Figure 2.4 would need to have a lookup table of $3 \cdot 2^{11}$ bits per PE (i.e. 11 bit input, 3 bit output). Although this is in principle possible for a small SPA, and may be done in the future, it is inconvenient, as the time it takes to load all lookup tables in the PEs is large with respect to the processing time.

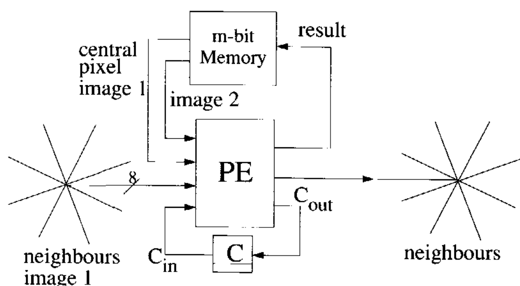


Figure 2.4 Basic bit-serial PE in an SPA

With a full-array, the image is as large as the SPA. Due to the large number of PEs needed for increasing image sizes (up to 4096 * 4096 for satellite images), this seems only suited for optical arrays such as the DOCIP, which will be treated later on (Huang et al. 1989). Available arrays have sizes from 8*8 to 128*128. A Processor Mapping Function (PMF) is used to distribute the image points over the PEs. A PMF shows in which memory plane *m* and in which PE at position (*x*,*y*) of the processor array the image point (*i*,*j*) is stored.

Table 2.2 Processor mapping functions for SPAs

Name	Function
Crinkle-mapping	$(x,y,m) = (i / k,\ j / k,\ i \bmod k + \{j \bmod k\} \cdot k)$ with $k = \lfloor N/\sqrt{P} \rfloor$
Window-mapping	$(x,y,m) = (i \bmod \sqrt{P},\ j \bmod \sqrt{P},\ \lfloor i/\sqrt{P} \rfloor + \lfloor j/\sqrt{P} \rfloor \cdot \lfloor N/\sqrt{P} \rfloor)$
Full-size	$(x,y,m) = (i,\ j,\ 0)$

For an SPA of *P* processors, processing an image of size *N*, the PMFs are listed in Table 2.2. With crinkle mapping, every PE contains a consecutive part of the image (see Figure 2.6a). This means that the points which are neighbours in the original image will in general not be neighbouring points in the crinkle-mapped image, as is shown in Figure 2.5. Crinkle-mapped images may therefore only be processed with a neighbourhood parallelism of 1. Because of the fact that sub-sampled versions of the image are stored, crinkle mapping may be used to do multi-scale image processing or to simulate an SIMD pyramid (Teeuw and Duin 1989).

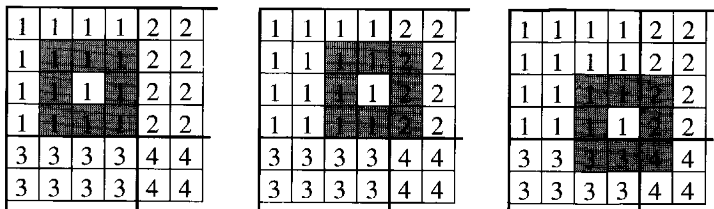


Figure 2.5 With crinkle-wise mapping neighbouring image pixels can be in the memory of up to 4 PEs.

Window mapping on the other hand, gives different problems. As can be seen from the PMF for window mapping, the SPA is loaded with image windows (pieces) of size $\sqrt{P}*\sqrt{P}$.

Although every window can be processed individually, hardware or software should provide the values of the neighbours which are across the window borders.

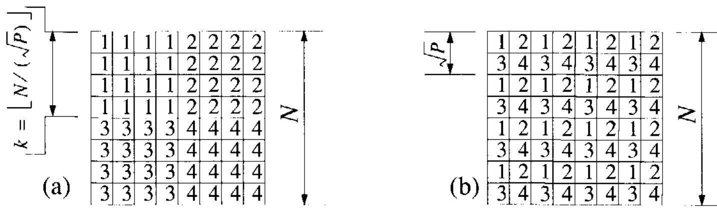


Figure 2.6 Mapping of an 8*8 image on 4 PEs of an SPA: (a) crinkle-wise, (b) window-wise.

Several methods exist to solve this 'edge-problem'. For instruction level processing (all image points are treated for one instruction then for the next etc.) the most promising methods appear to be the Edge Store Scanning (ESS) or the Half Scan Addressing (HSA) (Fountain 1987; Buurman and Duin 1988).

With *edge store scanning*, every window is processed only after an edge around the PA is filled with the values of the pixels which are neighbouring the window (see Figure 2.7a). Neighbourhood sizes should not exceed 3*3 in this method. Not many PAs are equipped with edge hardware to do this (Fountain 1987). A software implementation of this scanning technique is discussed in the next chapter.

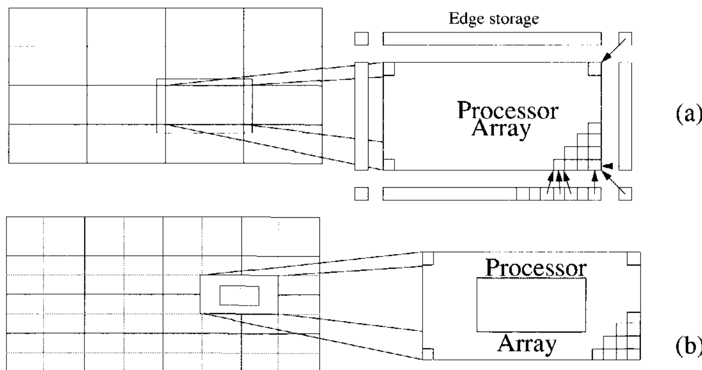


Figure 2.7 Hardware window mapping: (a) edge store scanning; (b) half scan addressing

With *half scan addressing*, the image is processed in four steps: for every window, the central part, the part connected to a window on the right, on the left and both are calculated (see Figure 2.7b). This is done by processing the normal image, the image shifted $\sqrt{P}/2$ to the left, to the top and both. After processing, only the central part of the processor array (a quarter of the total) contains data that is used (the dotted area in Figure 2.7b). Neighbourhood sizes up to $\sqrt{P}/4*\sqrt{P}/4$ can be treated with this method. If the PA is equipped with a hardware facility to address quarter planes in memory, no actual shifting is necessary, and the overhead involved is slightly less than four.

2.3.2 Linear Processor Array

A linear processor array (LPA) has P processing elements (PEs) to process an $N*N$ image. If the image size N equals the number of available PEs P, then every PE processes one column. Otherwise, a processor mapping function (PMF) determines which image

point is processed by which PE. The two PMFs used for an SPA - crinkle mapping and window mapping - are also used with LPAs. The AIS-5000 uses window mapping, and the PICAP3 uses crinkle mapping (Wilson 1988; Lindskog 1988). A third PMF in use, is the helicoïdal mapping for the SYMPATI-2 (Juvin et al. 1988). This mapping makes it possible to scan the array both horizontally as well as vertically across the image. The PMFs for these mappings are shown in Table 2.3.

Table 2.3 Processor mapping functions for LPAs

Name	Function
Crinkle-mapping	$(x,m) = (i / \lfloor N/P \rfloor, j \bullet \lfloor N/P \rfloor + i \bmod \lfloor N/P \rfloor)$
Window-mapping	$(x,m) = (i \bmod P, j + i \bullet \lfloor N/P \rfloor)$
Helicoïdal mapping	$(x,m) = (\{i+j\} \bmod P, i \bullet \lfloor N/P \rfloor + \lfloor j/P \rfloor)$
Full-size	$(x,m) = (i,j)$

To demonstrate the different PMFs for an LPA, Figure 2.8 shows how an image of size 8*8 is stored in 4 PEs.

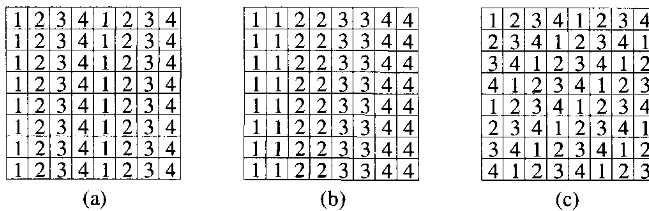


*Figure 2.8 Mapping of an 8*8 image on 4 PEs of an LPA, (a) window-wise, (b) crinkle-wise and (c) helicoïdal*

From the scanning point of view, PEs with a neighbourhood parallelism greater than one (the AIS for instance has a neighbourhood parallelism of five) will preferably use window-mapping, while other PEs may use crinkle mapping when scanning is only needed in one direction, or helicoïdal mapping when scanning should be possible in both horizontal and vertical directions. When an LPA uses window-mapping, hardware or software should provide for the values of the neighbours which are across the window-borders. An LPA has a strong advantage here over an SPA, as a simple hardware scheme allows a scanning technique which does not give any overhead (Wilson 1989a)

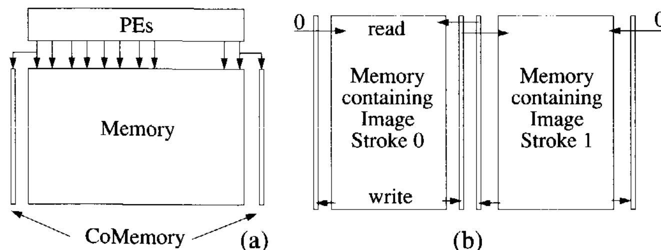


Figure 2.9 Hardware scanning without overhead on an LPA (see text).

Consider an image that is horizontally divided into two 'strokes' due to the fact that the number of processors P is half the image size N. Every time an image line is written into

memory (it may come from the PEs or from an input device), the two line edge points are also written into a co-memory (see Figure 2.9). Note that every memory line has two co-memory cells: one left and one right. Now every time that a memory line is read from memory, the edge memory cells are read from the required co-memories (Figure 2.9b). The co-memory address for reading or writing can be very easily calculated. Note that no overhead is involved in this scanning scheme.

The recursive neighbourhood parallelism of an LPA with a neighbourhood connectivity of $3*N$ cannot be higher than three. Only three pixelvalues which have been calculated for previous lines in a destination image are available for the calculation of the values of all pixels in the current destination image line.

2.3.3 Pipelined Processors

In a PipeLine (PL) of processors, there are P PEs performing a number of operations in parallel, and working on a sequence of pixels from an $N*N$ image. Image data comes from a memory or from an input device like a frame grabber which gives a sequential flow of pixels. This data is fed into the first PE, processed by it, and the result is fed into the next PE and so on (see Figure 2.10).

The result after P PEs can be displayed, or stored again in memory. The speed of the PEs should be equal for synchronisation, but the PEs themselves do not have to be the same.

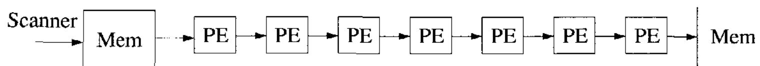


Figure 2.10 Example of a pipeline

When a PL incorporates different PEs, it should be reconfigurable, so that the optimal path through the PEs can be found. If a PL consists of identical PEs, these should be programmable, such that a set of operations can be performed in the right order. Research has been done to find out what a general purpose bit-serial PE for low-level image processing should look like (Duin and Komen 1989). By studying the nature of low-level image processing operations (they are treated in Chapter 3), it is noted that they can in principle be build up by bit-serial local neighbourhood operations. These operations can be performed by PEs with the following connections:

- two image inputs, so that dyadic operations can be done
- the local neighbourhood of one of the image inputs, so that local neighbourhood operations can be done directly, and larger neighbourhoods (up to global operations) can be treated by using the local neighbourhood operations as a basis
- a carry in- and output, so that grey value images can be processed
- at least one image output to be fed into the next PE of the pipeline

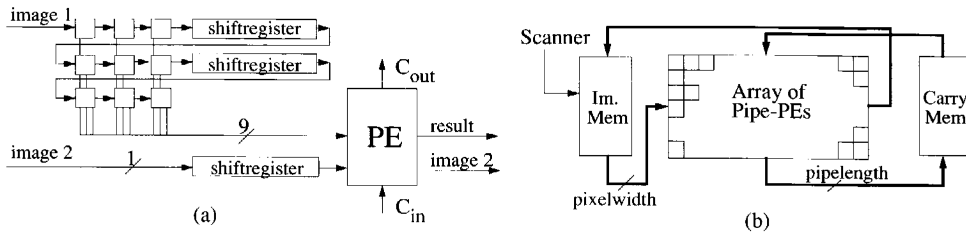


Figure 2.11 Pipeline Processing Element (a) as part of a folded PipeLine (b).

The basic bit serial PE is shown in Figure 2.11a. Two image bitstreams are led through shiftregisters. A local neighbourhood of points is tapped off the shift registers for image 1, and fed into the actual PE. Other inputs to the PE are one bitstream (sufficiently delayed) for a second input image, so that dyadic image operations can be performed, and a carry input from a PE which processes a lower significant bitstream. The PE should be general programmable, and deliver a result bitstream, the original image2 bitstream, and a carry-out for a more significant bitstream. If the PE (like the basic PE in Figure 2.4) is programmed through a lookup table of $3 \cdot 2^{11}$, then the general programmability is indeed fulfilled.

A PE of a PipeLine can have up to four recursive neighbours, if the resulting bitstream is fed back through shift registers and the processing of a pixel does not take more than one clock-cycle (otherwise three recursive neighbours could be available). The PEs may be combined to extend the operation parallelism as in Figure 2.10, by putting one after the other, but also as in Figure 2.11b, to extend the pixel parallelism. Such an array of pipelined PEs may process longer algorithms by using frame recirculation, and a carry memory may be used to store intermediate carry results, so that it is even possible to do high precision image processing at the cost of decreased speed (Jonker et al. 1989).

2.4 Existing low-level image processing architectures

Many existing architectures have been described in the available overview literature (Preston 1983; Fountain 1983; Preston and Duff 1984; Reeves 1984; Duin and Komen 1989). On the basis of the classification features developed here, an overview of existing architectures is given in Table 2.4 (for explanation of the symbols see Section 2.1.9).

Table 2.4 Overview of existing low-level IP architectures

Name:	Streams:	Autonomy:	Topology:	P_O:	P_S:	P_N:	C_N:	P_R:	C_R:	P_P:	Group
ILLIAC III	SIMD	no	mesh	1	32*32	8	8	0	0	1	SPA
DAP-610	SIMD	no	mesh	1	64*64	1	4	0	0	1	SPA
MPP	SIMD	no	mesh	1	128*128	1	4	0	0	1	SPA
CLIP4	SIMD	no	mesh	1	128*128	8	8	8	8	1	SPA
GAPP	SIMD	Act	mesh	1	24*24	2	4	0	0	1	SPA
CAAPP	SIMD	Act, Cnc	mesh	1	512*512	2	4	0	0	1	SPA
AIS-5000	SIMD	no	line	1	1024	5	$3N$	1	1	1	LPA
PICAP3	SIMD	Act Adr Fie	line	1	2	1	$3N$	1	4	32 FP	LPA
WARP	MIMD	all	line	10	1	1	1	1	1	32 FP	PL
Cyto-HSS	MISD	Alg	line	88	1	9	9	0	0	1 or 8	PL
DIP	MISD	no	line	5	1	9/1	9/9	4/0	4/0	1/18 FP	PL
CLO-VLSI	SIMD	Alg	line	12	1	9	9	0	0	1	PL

N = one dimension of the image being processed.

Some of the existing low-level image processing architectures will now be looked at, with the purpose to show how the systems work, what the structure of the PEs is, and how the parallelisms and connectivities are reached.

2.4.1 The Illinois pattern recognition computer (ILLIAC III)

One of the first mesh connected square processor arrays intended for visual recognition was the ILLIAC III (McCormick 1963). Large images (up to 1024*4096) were divided into scans of 32*32 pixels. The scans were then processed sequentially by the 32*32 PE processor array in a raster scan order from the bottom-left to the top-right one. Each PE (called stalactite at that time) was connected to its 8 nearest neighbour and to itself, resulting in a neighbourhood parallelism of 9. The functionality of the PE was limited, but included functions such as the Boolean or of the neighbours and a programmable threshold of the number of neighbours with value one (neighbourhood percentile filter).

2.4.2 The Massively Parallel Processor (MPP)

The MPP was designed by NASA (Fung 1977) with the primary function of analysing LANDSAT satellite data. The system was redesigned by Goodyear Aerospace (Batcher 1980). The PE array consists of a 128 * 128 matrix, so that it has a spatial parallelism of 128*128. The only directly implemented global feature extraction operation is the OR function. The simplified PE is shown in Figure 2.12.

The emphasis of this design is on fast arithmetic computation rather than binary neighbourhood operations. The N/S/W/E selector enables a bit plane to be shifted in one of four cardinal directions in one instruction, so that the neighbourhood parallelism is 1.

Figure 2.12 Simplified MPP PE organization

The Boolean processor implements all 16 possible Boolean functions between the P register and the value on the data bus; in this case P is an accumulator. For arithmetic operations a dynamically reconfigurable, variable length shift register with a maximum length of 30 bits, and a full adder are available. This organization is faster than the BASE/CLIP4 (see below) schemes especially where multiplication is involved; in this case the shift register is used for circulating the partial product. This organisation also influences the instruction overhead as defined in Chapter 4.

2.4.3 The Distributed Array Processor (DAP)

The DAP is a mesh connected processor array of 32*32 or 64*64, which can be used for image processing although it was not primarily intended for it (Reddaway 1973; Hunt 1981). Like the MPP, the DAP is a 4-connected mesh, where just one of the neighbours can be read at a time, so that it has a neighbourhood parallelism of 1. Just as with the MPP, the PE of the DAP is build around a full adder, but no internal shift register and no

boolean function unit is available. The DAP has row and column highways connected to the PEs, so that it can handle vector operations more easily. At the moment VLSI chips are available which contain 64 DAP PEs.

2.4.4 The UCL Cellular Logic Image Processor-4 (CLIP4)

The CLIP4-UCL is a Processor Array which was originally developed in the University College of London (Duff 1982). It was commercially produced by Stonefield Plc, UK, for a few years. The CLIP4-Delft (with 32*64 PEs) is one of these, see(Buurman and Duin 1988). The CLIP4-Delft has been used in experiments described in Chapter 4, Section 4.7.4. The LSI chips used in the CLIP4 contain 4*2 PEs (about 3000 transistors) and are made in an NMOS process.

To perform local and global operations, the processors (residing at the same position as the pixels) are each connected to their 8 nearest neighbour-processors. The logical **or** from a selectable number of the 8 neighbour processors can serve as input for a PE. Hexagonal neighbourhood connections are also available

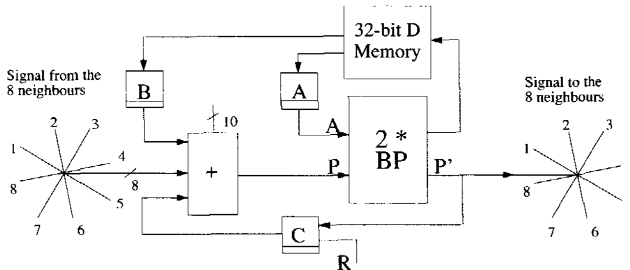


Figure 2.13 Simplified Processing Element

The CLIP4 PE contains an A, B and C register, and also 32 Data-registers for data storage. The A and B registers can be loaded with data from the D registers, or with external data. Data from the A register can be put into a D register or an external memory. At the PEs heart are two Boolean Processors (BP's). One serves to obtain a correct *propagation signal* (P).

After this is done, the second BP calculates from A and P a value which is stored in memory. The P-signal is a logical **or** from the B-register, the C-register and the outputsignal of the 8 neighbours. Any of these 10 signals can be in/excluded from the **or**ring.

It is clear from Figure 2.13, that the neighbourhood parallelism and connectivity of the CLIP4 is 8. If the CLIP chooses to use the propagation signal to and from its neighbours, then the neighbourhood parallelism and connectivity is exchanged for *recursive* neighbour parallelism and connectivity. At that moment there are eight combinatorial paths going out from the Boolean processor of one PE through its eight neighbouring PEs (processing happens on the fly) back into itself. The process is asynchronous, and ends when it is detected that the propagation signals do not change any more on all PEs. This feature of the CLIP is called 'global propagation', as it allows information from one end of the processor array to be propagated across the whole array asynchronously.

2.4.5 The Geometric Arithmetic Parallel Processor (GAPP)

The GAPP is a commercially available chip with which commercial systems can be built, see (Davis and Thomas 1984). Each of the PEs contains an ALU and 128 bits of

RAM, as well as bidirectional communication lines that connect the cell to its neighbours on the north, south, east, and west. In addition, a separate I/O communication bus allows data to be input from the south end of the array and output to the north without interfering with computation within the ALU.

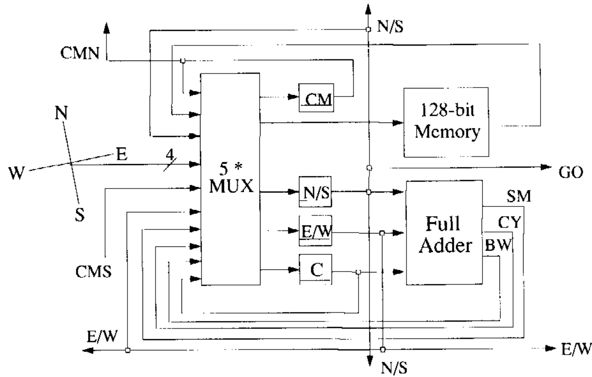


Figure 2.14 GAPP processing element

The chip consists of 72 Processors in an array of 6*12. It is manufactured with a 3-µm double-layer metal CMOS process (1 µm would crowd 512 cells onto an equal sized chip). Each processor element contains separate lines that link the cell to its neighbours and to the outside world. In addition to the North South (N/S) and East West (E/W) lines that pass data between cells, are the CM South input (CMS) and CM North output (CMN). There is also a global output (GO). Each of the four latches of the PE (CM, N/S, E/W and C - the C register) accepts data from up to eight possible sources, depending upon the setting of the control lines. The neighbourhood parallelism is 2, because only two neighbours are at the same time available for processing by the full adder circuit.

Working from a truth table, the array performs additions and subtractions. The summing output of the single bit ALU, SM, goes directly to the RAM and may also be fed in parallel to any of the four registers. The Carry and Borrow outputs (CY and BW) are open to the C register. A truth table is used as well to fulfil single and dual input logic functions on data in the N/S and E/W latches.

2.4.6 The Contents Addressable Array Parallel Processor (CAAPP)

The CAAPP is a square processor array forming the bottom of a pyramidal architecture called the Image Understanding Architecture (Weems et al. 1989b). Above the CAAPP, which performs the low-level image processing, is a layer of mesh-connected Digital Signal Processing chips (DSPs) for intermediate level image processing. On top a layer of LISP processors is situated for the high-level image processing. The CAAPP is four-connected, but can only use 2 neighbours at the same time, so that it has a neighbourhood parallelism of 2. The VLSI efforts of the CAAPP designers have resulted in a 2-micron CMOS chip containing 64 PEs (120000 transistors). For communication to the upper layer, the CAAPP has some associative capabilities:

1. An array wide logical OR output.
2. A local Some/None signal derived from a subarray is fed to one of the DSPs residing in the middle layer of the Image Understanding Architecture.

3. A count of all responding cells.
4. A count of an 8*8 subarray for the DSP from the middle layer attached to this subarray.

There are also some global features built within the processor array, such as broadcasting, and the use of the coterie network. The coterie network gives each PE local connectivity autonomy. This makes it possible to separate the array into isolated regions, so that the global response is now restricted to that region. The use of this local autonomy has been demonstrated for some operations and seems to be very promising.

2.4.7 The AIS-5000 from Applied Intelligent Systems

The AIS-5000 is a commercially available Linear Processor Array, made by Applied Intelligent Systems (Wilson 1985; Schmitt and Wilson 1988). The bit-serial PEs are built as 8 in one gate array, and 16 gate array's per VME board (128 PEs per board). Up to 8 boards (1024 PEs) can be used in one system. The 8 PEs of one gate-array are tightly coupled to a the memory in a 32K * 8 bit RAM. Every PE can access the data in its neighbouring PEs. The PEs on the ends of the array communicate only to one neighbouring PE on their left or right. The other neighbour connection is always a zero.

The gate array chip is also equipped with an I/O section to get data in and out of the parallel memory. This I/O section consists of three separate channels of 7 MHz, which operate independently of, and asynchronously with the PEs on the gate array. The I/O section and the PE section of the gate array share the line to memory. Management of the channels is under program control.

The individual PE of the AIS-5000 is a general-purpose bit-serial processor. It can be programmed using the C-language. There are three types of operations that the PEs perform: Boolean, neighbourhood, and arithmetic operations.

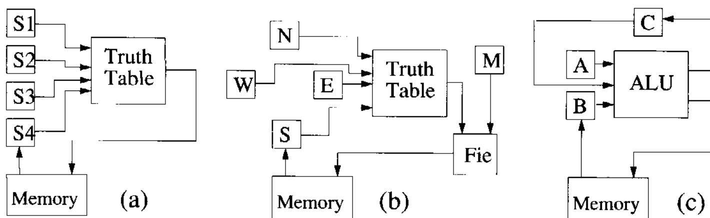


Figure 2.15 Functional use of the AIS-5000 PE: (a) boolean-, (b) neighbourhood- and (c) arithmetic-function

Any Boolean function between 1, 2, 3 or 4 input variables can be programmed using a fully programmable truth table (Figure 2.15a). For neighbourhood operations, any Boolean function between the 4 nearest neighbours can be programmed in a truth table, and the result of this can then be combined with the central pixel using a programmable function *Fie* (Figure 2.15b). The PEs can perform arithmetic by making use of their carry registers. In arithmetic operations, a sum bit and a carry bit are generated for two source bits and a carry bit input. The carry bit is stored in the carry register and used as input to the PE on the next cycle (Figure 2.15c).

2.4.8 The PICAP3

The PICAP3 is an SIMD linear processor array working with 32-bit floating point numbers (Lindskog 1988). Unlike other linear processor arrays, it uses crinkle mapping to di-

vide the image points over the available PEs. The global system is shown in Figure 2.16. A high speed (40 MB/s) I/O bus is used to input and output data to and from the array.

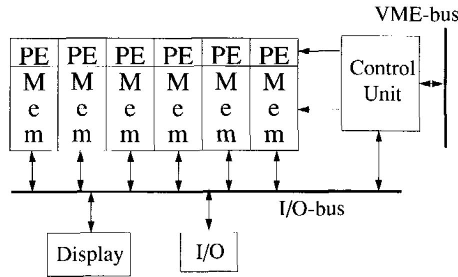


Figure 2.16 PICAP3 linear processor array system

Every processor is connected to its left and right neighbour, and the leftmost PE is connected to the rightmost, so that data can be circulated. The PEs have several types of local autonomy:

1. Local activity control - a bit which says whether this PE joins in the calculation or not
2. Local memory indexing (i.e. local addressing control) - used in operations such as histogramming.
3. Local shift count control (i.e. local function control) - to speed up floating point normalisation operations.

2.4.9 The Cyto-High Speed System

The ERIM (Environmental Research Institute of Michigan, Ann Arbor) Cyto-HSS system is a follow up of the earlier Cytocomputer (Lougheed and McCubbrey 1980a; ;Lougheed 1987). It is a frame-recirculating pipeline with a processing element suited for binary (morphologic) image processing, as well as for grey-value (max/min morphologic) processing. Each Neighbourhood Processing Stage is a single circuit board. Images are fed into the PEs through an image memory. In addition to the processing done by the PEs, point operations between images can be done using the image combiner. Provisions are made for auxiliary processors.

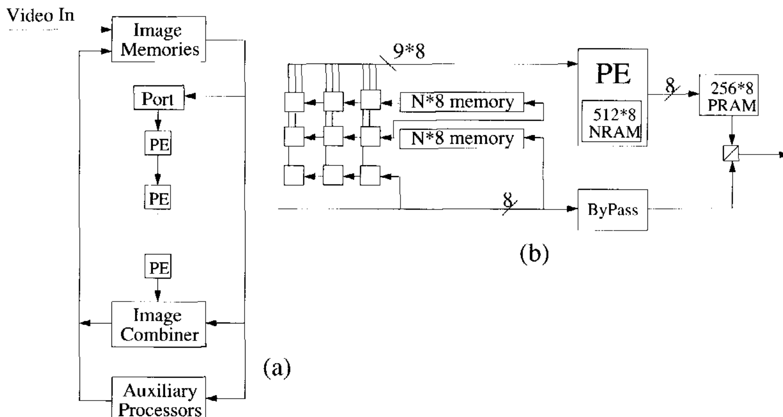


Figure 2.17 Cyto-HSS pipeline (a) and Processing Element (b)

The processing stage accommodates pixel widths up to 8 bits, to allow for storing intermediate results of image operations. Processing proceeds in three steps (Figure 2.17b): First a mapping of the nine neighbourhood pixels $P_0 ... P_8$ into a 9-bit Active Neighbour Vector (ANV) is performed. Second a mapping from the ANV (9 bit environment) and the central pixel P_0(8-bit pixel) into a Neighbourhood Transformed Centre (NTC) pixel value is performed, using a 2^9*8-bit lookup table (the Neighbourhood or NRAM) and a selector under program control. Thirdly the NTC pixel value is mapped into a Resultant Centre (RC) pixel value. This is done by using the NTC as an index into a 256*8-bit table (the Point by Point RAM or PRAM) of programmed RC pixel values. The obtained RC value is the stage's output. This final mapping is arbitrary and thereby performs all point-by-point functions of 8-bits, i.e., bit plane shuffling, ANDing, etc.

Each stage can also be passed without processing using the bypass circuit (this effectively reduces the pipeline length).

2.4.10 The Delft Image Processor (DIP)

The DIP was a programmatically reconfigurable pipeline of different PEs (Gerritsen and Aardema 1981; Gerritsen 1982). It is intended both for cellular logic as well as grey value image processing. For cellular logic operations it contains downloadable lookup tables to implement any function on the source pixels of a 3*3 neighbourhood (neighbourhood parallelism is 9) and four delayed destination pixels (recursive neighbourhood parallelism and connectivity is 4). Dyadic grey value operations are possible using two floating point ALU PEs and a floating point multiplier PE. Integer pixels are optionally translated to 18 bit floating point numbers using conversion tables in the data input and output streams of the pipeline. The neighbourhood of the grey value source pixels can be scanned using two programmable neighbourhood data buffers. These data buffers allow the use of neighbourhoods (with corresponding coëfficients) up to 16*16.

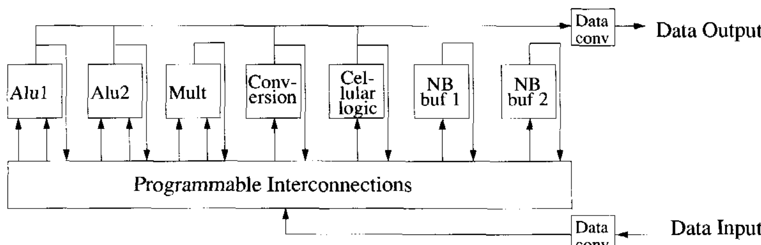


Figure 2.18 The Delft Image Processor datapaths (simplified)

Although the DIP is now extinct, it had at least two exceptional features. First, the DIP was to our knowledge the first pipelined image processing architectures in which there was a facility to use neighbours recursively. Second, the flexibility in the neighbourhood size has, to our knowledge, not yet been equalled by modern image processing pipelines.

2.4.11 The Cellular Logic Operations chip (CLO)

The CLO chip is a VLSI-design project of the John Hopkins University Applied Physics Laboratory (Jenkins and Lee 1987). This bit-serial pipeline processing element (serving as coprocessor) was designed with the intention to do research on cellular automata. A pipeline of 10 stages has been manufactured on a printed circuit board and attached to an IBM PC. Each PE is equipped with the necessary shift registers to form a 3*3 environ-

ment, and a LookUpTable which can be downloaded with one Boolean neighbourhood function. Additionally, the total number of ones remaining in an image is continuously counted, and can be read from a register. This can be used for image analysis.

2.5 Proposed and novel low-level image processing architectures

Some interesting low-level image processing architectures which have not yet come past the proposal phase or which are not yet completely operational are listed in Table 2.5.

Table 2.5 Overview of proposed low-level IP architectures

Name:	Streams:	Autonomy:	Topology:	P_O:	P_S :*	P_N:	C_N:	P_R:	C_R:	P_P:	Group:
BASE	SIMD	no	mesh	1	8*8	8	8	8	8	1	SPA
GRID	SIMD	no	mesh	1	64*64	1	8	0	0	1	SPA
MMB	SIMD	no	mesh	1	na	na	na	na	na	1	SPA
PTA	SIMD	Con	torus	1	na	na	na	na	na	1	SPA
DOCIP	SIMD	no	mesh	1	full	any	any	0	0	1	SPA
SLAP	SIMD	Adr	line	1	1024	1	3*N*	1	1	8..20	LPA
CLIP7	SIMD	Adr,Act	line	1	256	1	3*N*	1	1	8..16	LPA
SYMPATI-2	SIMD	no	line	1	256	1	3*N*	1	3	8	LPA
PIPE	MISD	Fie,Pr	line	>1	1	9	9	1	1	1	PL
MITE	MISD	Pr	line	>1	1	9	9	1	1	1	PL
CLPE-VLSI	SIMD	Pr	line	>1	1	9	9	4	4	1	PL
PAPIA	MSIMD	no	pyramid	1	na	5	13	0	0	1	PYR
IMPP	SIMD	Adr	data	>1	>1	1	1	1	1	16	-

* intended size; *N* = size of the image being processed; na = Not Applicable

In this paragraph we will have a closer look at some of the architectures shown in Table 2.5. The reader should be aware of the fact that there seems to be a certain positive correlation between how hypothetical an architecture is and how interesting it looks. As stated in Section 2.1 we do not take into account architectures which are clearly not aimed at low-level image processing such as the Associative String Processor (Lea 1988) or Transputer based systems (PIPS 1990).

2.5.1 The BASE

The BASE system is developed in a small prototype form at Purdue University, (Reeves 1980a). The BASE PE consists of three main components: a Boolean processor which can implement any three-input Boolean function, a 1-bit wide local memory and a near neighbour function processor (NNF). The BASE architecture looks very much like the CLIP4, in that it also allows the switching between a neighbourhood parallelism of eight or a recursive neighbourhood parallelism of eight.

2.5.2 The GEC Rectangular Image and Data computer (GRID)

The GRID is developed by a company named GEC for image, signal and numerical processing applications (Pass 1985). A testchip containing 8*8 PEs is under design. By using routing through the four physical links to its neighbours, a neighbourhood connectivity of eight is reached. The neighbourhood parallelism however, is one. It also contains row and column buses, which can be used for several purposes. One purpose is to read or write data to specific PEs. Another purpose is in histogramming the image data. The GRID controller is constructed so that the image data can be mapped crinkle wise in its local memory.

2.5.3 Mesh with Multiple Broadcast

The MMB (mesh with multiple broadcast) architecture is an idea to enhance the normal square processor array with more global communication (asynchronous) possibilities (Prasanna and Dionisios1989). In addition to the normal 4/8 connected mesh pattern within an SPA, it is proposed to provide each row and each column with row- and column-buses. It has been shown that an MMB architecture performs as well as a pyramid architecture (Prasanna and Dionisios1989).

2.5.4 Polymorphic Torus Architecture

The PTA (polymorphic torus architecture) is another enhancement idea for the mesh-connected SPA (Maresca et al. 1989). Fast communication channels (asynchronous) between non-neighbouring PEs without extra wiring complexity can be constructed by equipping every PE with a switch that allows the coupling of all 4 source directions (N,S,E,W) with all 4 destination directions (N,S,E,W). This facility is called *connection autonomy*, and is similar to the *switching autonomy* of the CAAPP. It allows the embedding of several interconnection patterns, including: mesh, multiple bus, tree, ring and even data-dependent patterns.

2.5.5 The Digital Optical Cellular Image Processor (DOCIP)

The DOCIPs are suggested as an all optical implementation of a mesh or hypercube connected SPA (Huang et al. 1989). Images are obtained optically, fed to an optical gate array, through an interconnection unit and again to the gate array. The interconnection unit is an optical hologram, which can implement any space invariant interconnection pattern. From the article of Huang it is not completely clear how the DOCIPs can be implemented. However, if architectures such as DOCIP were to be constructed in the future, it would be possible to process images with a full-size SPA, and with desired neighbourhood connectivity.

2.5.6 Scan Line Array Processor

The SLAP is a linear processor array which is being designed to work at video rate (Fisher and Highnam 1985). A VLSI-design project for a 4-PE chip incorporating 10000 transistors is being prepared (Fisher et al. 1987).

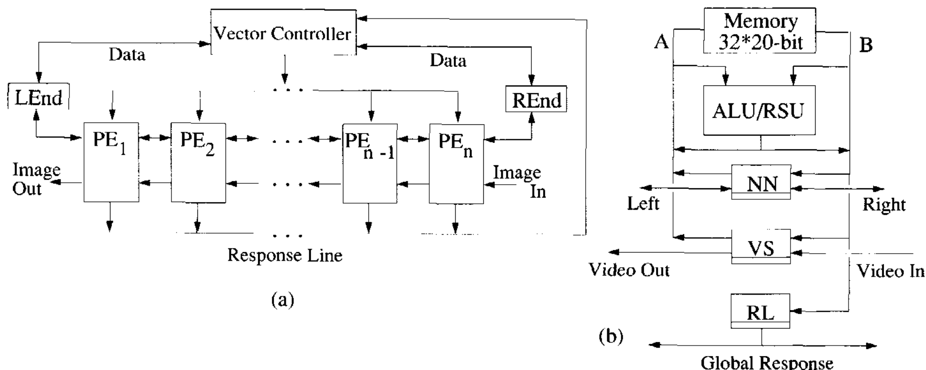


Figure 2.19 Scan Line Array Processor architecture (a) and processing element (b)

The SLAP processing element (Figure 2.19b) is built around a 20-bit read bus and a similar write bus (A and B). Input into one of the 32 20-bit registers can be obtained from

the left or right nearest neighbour (via the NN register) or from the video input (using the VS register, which works normally on a multiple of the clock-speed, but is slowed down for reading out). A register from the 32 register 'file' can be output into the NN, VS or RL register. The latter translates it into a 1-bit global response (this is used by the vector controller in the total system). The A and B buses are also connected to the ALU and RSU, which together implement several operations such as multiplication, division and logical shifting.

The system as a whole (Figure 2.19a) works in SIMD mode, but has the possibility of local addressing to perform data-dependent computations (i.e. histogramming etc.) more efficiently than is possible in a bit-serial processor. The so-called 'real-time' performance can be met by shifting in a line of video-input (raster scan data I/O) concurrently with processing. When it is completely in place, the line is stored in one of the registers in the register file. The pixel-bit parallelism of the SLAP is 8 externally, but 20 internally, as it accepts 8-bit image data, but processes it within the PE using 20 bits. The neighbourhood parallelism is 1, as it is capable of loading one of its left/right neighbours into the NN register.

Unlike the SYMPATI-2 (see below), the SLAP is at the moment not able to scan with a small LPA over a wider image. Every PE has to be fixed to one image column.

2.5.7 Cellular Logic Image Processor - 7

The CLIP7 is a linear processor array designed to work with 256 PEs (Fountain et al. 1988a). Each PE actually comprises two CLIP7 chips, where one is used for the address calculation and the other for the data processing part. Through the use of edge registers, the PEs are 8-connected. The processor mapping function can be defined through the use of the address calculation part of the PE. This allows the implementation of different forms of connection models on top of the physical CLIP7 array.

2.5.8 SYMPATI-2

The SYMPATI-2 project is a collaborative project for a grey-value linear processor array between the French CERFIA laboratory and the CEA/DEIN research center (Juvin et al. 1988). The array connections and PE-layout is shown in Figure 2.20. The processor mapping function (PMF) from the pixels to the PEs may be helicoïdal, which enables this linear processor array to scan both horizontally as well as vertically (this is of special interest for the recursive neighbourhood operations treated in Chapter 5). Address calculations for the helicoïdal PMF are done in parallel with the normal pixel processing of the PE (Figure 2.20b). The LPA is also capable of handling other PMFs through the user-supplied addresses via the index register. Addressing of the memory is either helicoïdal or through the index register, as indicated with the switch.

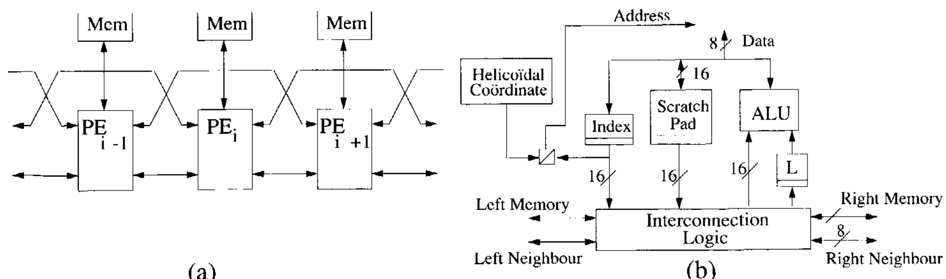


Figure 2.20 SYMPATI-2: (a) processor array slice, (b) and PE layout.

The connection between neighbouring PEs (Figure 2.20a) is twofold. A PE can get the contents of neighbouring scratchpad registers as well as the contents of neighbouring memory contents. The memory access may even be done at a distance of 2 PEs and the register access at a distance of 3 PEs within one clock-cycle. The neighbourhood parallelism as well as the recursive neighbourhood parallelism are 1.

The ALU implements several Boolean and arithmetic functions (including multiplication but excluding division). A chip has been made by AT&T in 1,2um CMOS holding 4 PEs (about 90000 transistors). The design of a whole SYMPATI-2 system is still going on. It is not clear at this moment, how the data I/O will be realised. The spatial parallelism of the SYMPATI-2 is at the moment restricted to 128 times the image size N.

2.5.9 The Morphic Image Transform Engine (MITE)

The MITE is a pipeline developed by IBM with has an interconnection pattern that can be programmed at configuration time (Kimmel et al. 1985). The processing of data takes place at processing time. The neighbourhood transformation of the PE is a function of ten inputs, nine from the neighbourhood, and one equal to the neighbourhood transformation from the previous "iteration" (compare this 1 recursive bit with the 4 recursive bits from the CLPE). Since the look up table LUT - RAM is a 1024 by 1 bit memory, any function of ten bits can be realized. The delay's can be individually programmed for each PE at configuration time to accommodate two-dimensional images widths between 8 and 16384 pixels.

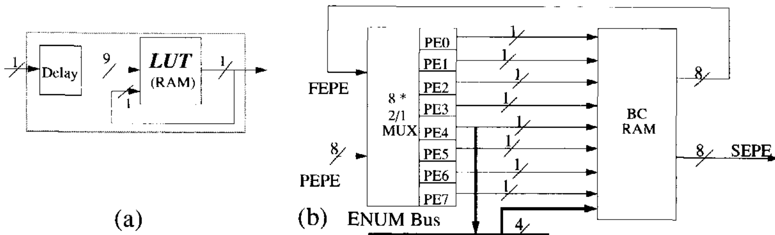


Figure 2.21 (a) MITE Processing Element. (b) MITE 8 PE group (PEG)

The basic structural element of MITE is the PE-Group (PEG), consisting of 8 PEs with reconfiguration capability. The PEG contains a programmable Boolean Combiner (BC) that provides sixteen connection points. The 8 PE outputs connect to eight of the address inputs of a BC-RAM which has sixteen bits per word. Programming the RAM can allow the sixteen outputs to be sixteen independent Boolean functions of the eight PE inputs. The ENUM bus is used to receive results from PEs which do not belong to the PEG or to offer results of PEs to other PEGs.

2.5.10 The Cellular Logic Processing Element-VLSI (CLPE)

The CLPE is a VLSI design for pipelined cellular logic operations, (Jonker and Duin 1985). Four input bitstreams *in1...in4* can be mixed using a selector and two Boolean Function Generators (a BFG is a switching device, which can be programmed with any of the possible functions between boolean inputs). An internal shift register (programmable up to 512 cycles, but this can be extended externally, to allow for larger images) delays one bitstream, so that a 3*3 neighbourhood is available to the processing heart of the CLPE. Instead of calculating the neighbourhood function with a LookUp Table as in the CLO, a

Writable Logic Array is used (a WLA is the writable version of a PLA). This requires less hardware. The WLA is also fed with four recursive neighbours so that results are immediately re-used in the calculation of new pixels. This means that the recursive neighbour parallelism is 4. It is also fed with the delayed mask bitstream to allow parts of the image to be masked off. The central pixel of the 3*3 neighbourhood is also available at the output.

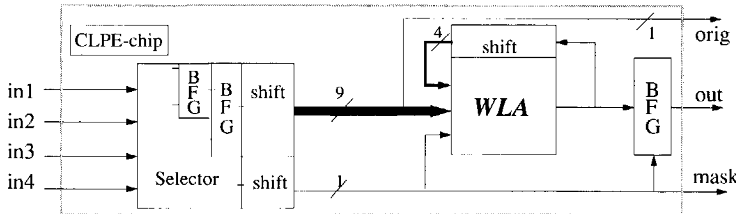


Figure 2.22 Cellular Logic Processor VLSI structure

Many basic operations can be directly programmed in the WLA and there is an extra provision in the chip for 3*3 threshold operations.

2.5.11 The Pipelined Image Processing Engine (PIPE)

A pipeline with some interesting features was proposed in 1985 (Kent et al. 1985). The PIPE consists of a sequence of identical PEs, working under the MISD mode. Each PE has three inputs and three outputs. One output goes to the next stage in the pipe and it is also fed back to the same stage so that recursive neighbourhood operations can be done. The second output goes to the previous stage, so that both forward and backward flow are possible. The inputs come from thc previous stage, the next stage and the same stage. There are also some separate paths to allow image transfer between stages at a distance greater than one. On top of the fact that each pipeline stage can be programmed, it is also possible to let the PE function depend on the position of the source pixel. PIPE also offers the possibility to change resolution from stage to stage, so that multiresolution processing can be done.

2.5.12 The Pyramid Architecture for Parallel Image Analysis (PAPIA)

The PAPIA stems from a national project between a number of Italian universities to evaluate, suggest and build a parallel architecture to perform fast image processing (Cantoni and Levialdi 1987).

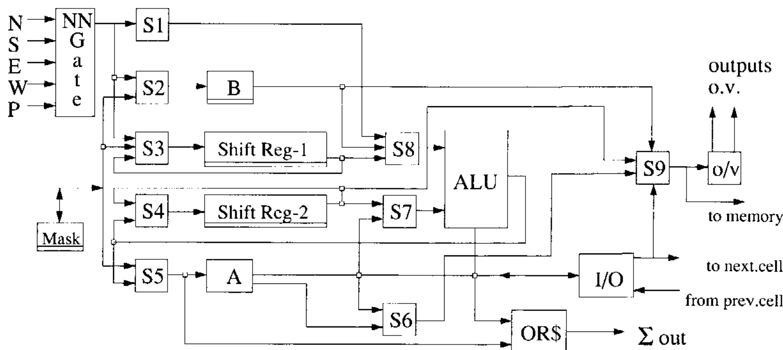


Figure 2.23 PAPIA Processing Element.

The pyramid is of the multi SIMD-mode type with a fixed interconnection scheme,

where each processor has one father above, four neighbours in plane, and 4 sons below. Every plane can run with an individual program in SIMD-mode. There are two different operation modes: a horizontal one in which each PE communicates inside the plane with its brothers (4 horizontal neighbours are enabled), and a vertical one in which a PE has extra plane communication with its father and its sons. The processing and the data loading/unloading may overlap. The PE not only contains registers A and B (and two shift registers *Sr1* and *Sr2* for arithmetic operations), but also a carry register C inside the ALU which is composed of a Boolean processor, a full adder, and a comparator between the two shift registers. The length of *Sr1* and *Sr2* is a maximum of 32, but can be programmed, depending on the number of bits per pixel.

2.5.13 The NEC IMage Pipelined Processor

A digital signal processor (DSP) based on the 'data flow' concept and introduced by NEC (the μPD7281 or 'IMPP') has been reported in the literature (Iwashita et al. 1986). All the architectures which have been treated so far operate in the 'control flow' concept, where the sequence of operations is defined by the programmer. In the data flow concept, the sequence of operations is defined by the data dependencies of the operations. Data flow programs can be represented by flow graphs, where the nodes represent the operations and the branches form the virtual paths along which the data flows (Davis and Keller 1982). Data flowing on a branch is represented by a 'token'. If the necessary input tokens for a specific programmed operation are available, an output token is produced. For a more detailed description of the data flow concept as applied to the IMPP we refer to the literature (Iwashita et al. 1986).

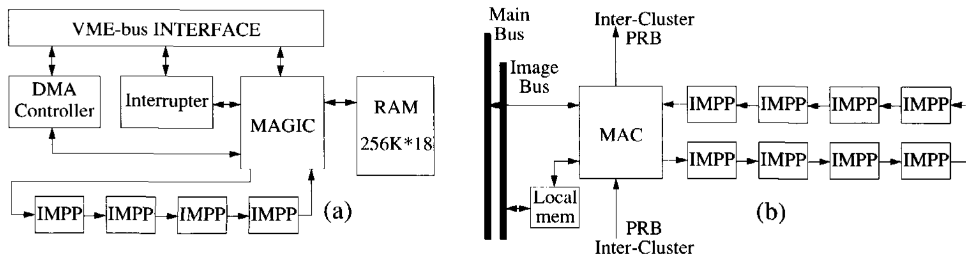


Figure 2.24 IMPP data flow systems: (a) Delft-VME data flow system, (b) TIP-4 part.

In one project four IMPP chips were implemented on a VME board together with one MAGIC chip which was used for memory and bus interfaces (Figure 2.24a; Dekker et al. 1987; Jonker et al. 1990).

Another image processing system that uses IMPP chips is the TIP-4 (Fujita et al. 1990). A prototype TIP-4 is built from 64 PEs which consist of 8 'clusters'. Each cluster (see Figure 2.24b) consists of 8 PEs connected by a PRB (pipeline ring bus) and sharing 512 words of local memory. Each cluster is interfaced by a MAC chip (Memory Access Controller).

The data flow concept seems to be promising in that it allows massive parallelism to be combined with the flexibility required to efficiently implement data dependent image processing operations (such as the recursive neighbourhood operations which are discussed in Chapter 5). In the literature it is noted that "the data flow concept is not very adequate to describe problems where the data does not flow, i.e. several processes that operate on one big data structure like a histogram" (Groen et al. 1988).

2.6 Evaluation on the use of the classification features

The aim of this chapter was to gain insight into the principles of the existing and novel low-level image processing architectures by looking at features to classify them. This led to the selection of a set of classification features in Section 2.1, and the selection of a set of architecture groups described by these features in Section 2.2. In further chapters of this thesis, the comparison will be continued between the chosen architecture groups.

After an introduction into representative members of these architecture groups, the operation of a large number of existing and novel architectures was explained, and their classification features were compiled in Table 2.4 and Table 2.5. The purpose of this section is to review the use of the classification features, and evaluate them against the criteria which were stated in Section 2.1.

It was noted that the difference in the SIMD and MISD stream taxonomies can also be seen from the operator and spatial parallelisms. All SIMD machines have an operator parallelism of one (only one operation is executed concurrently), and a spatial parallelism greater than 1. All MISD machines have an operation parallelism of greater than 1, and a spatial parallelism of 1 (only one destination pixel point is calculated concurrently). However, this is not necessary, as operation and spatial parallelism are not mutual exclusive. This is demonstrated by MSIMD architectures such as the pyramid or the use of a pipeline of processor arrays (Schmitt and Wilson 1989).

The use of the level of autonomy feature reveals that there is a trend in low-level image processing architectures towards the use of local activity, local addressing and local connectivity control. The local connectivity control on an SPA makes it possible to transfer data over distances beyond the physical connections between PEs. It also allows the simulation of other topologies than mesh, while still using a mesh. The local addressing is observed only in LPAs, and there it is effectively used for operations such as histogramming.

The use of the topology feature in the description of architectures shows that even in proposed machines the *mesh* and *line* topologies are still favourite. Few other topologies are offered for low-level image processing. A reason for this might be the philosophy for low-level image processing that any low-level operation can be built from *local* neighbourhood operations. This is due to the spatial correlation in the two-dimensional image processing data. Topologies other than line, mesh, torus or pyramid leave the *local* neighbourhoods.

There also seems to be some negative correlation between the pixel bit parallelism and the (recursive) neighbour parallelism. Bit serial architectures may have some (possibly recursive) neighbour parallelism, but grey value architectures usually don't (the Diff3 is an exception; Graham and Norgren 1980). This stems from the fact that the cost of the ALU in a PE in terms of chip space is proportional to the total number of bits offered concurrently to the ALU. A bit serial PE for instance may offer 9 neighbourhood bits and one or two extra data bits to its ALU. It thus has a neighbourhood parallelism of 9 and offers 11 bits to its ALU. A grey value PE may offer two times eight bits to its ALU. Then it has a neighbourhood parallelism of 1, but offers 16 bits to its ALU. The Cyto-HSS does accept 9 neighbours of 8 bits into its PE, but first operations are performed on the neighbours to bring them back from 8 to 1 bit, and the remaining 9 bits are offered to a lookup table.

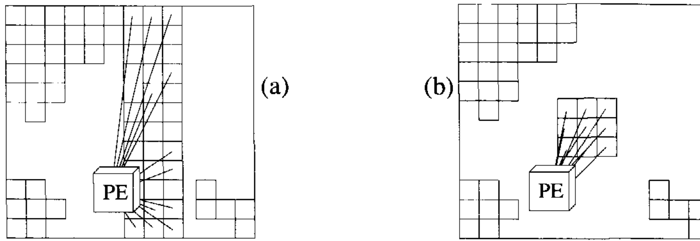


Figure 2.25 The number of neighbours reachable by a PE: (a) in an LPA, (b) in an SPA or PL.

The use of the neighbourhood connectivity reveals a basic difference between LPAs on one hand and SPAs and PLs on the other hand. The first group may have neighbourhood connectivities of order *N*, whereas the latter group generally has neighbourhood connectivities of order 1 (Figure 2.25). This is due to the fact that the LPA has one complete row/column/diagonal stored in the memory associated to one PE. If an SPA stores the image data in a crinkle wise manner, its neighbourhood connectivity order also gets larger.

A recursive neighbourhood parallelism of greater than four is only offered by processor arrays such as CLIP4, using their global propagation feature. The bit serial PEs can become combinatorial transparent, so that all paths going out of a PE can come back to it through neighbouring PEs. This is illustrated in Figure 2.26.

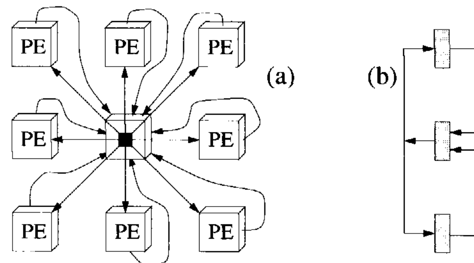


Figure 2.26 Maximum reachable recursive neighbourhood parallelism in CLIP4 via physical combinatorial paths through the nearest neighbours: (a) top view; (b) vertical cross section of the paths from the central pixel through the upper and lower neighbours.

For non bit serial PEs, and PEs which only work in a clocked way, the maximum reachable recursive neighbourhood parallelism and connectivity for a 3*3 neighbourhood in a PL is four, and in an LPA it is three (see Section 2.3).

It can be concluded, both from the description of the existing and novel architectures, and also from Table 2.4 and Table 2.5, that the classification features which are used indeed show meaningful differences between image processing architectures, while still focusing on the low-level aspects.

The selection of some *groups* of image processing architectures (described by the features) is a choice which allows a simplification of the comparison between architectures. This specific grouping choice (SPA, LPA and PL) can be viewed as a consequence of the special structure of images. These multi dimensional data sets have a spatial relationship between neighbouring data points. This is reflected in image processing architectures which have an emphasis towards local neighbourhood processing.

Other groupings may be chosen in future comparisons, but they should also be chosen in agreement with the nature of the multi-dimensional signals which they have to process.

Chapter

3 The road to a good comparison

Before an actual comparison can be made between architectures for image processing, the comparison should be defined in a meaningful way. A lot has been written about this in literature, and the discussion on this topic is not yet closed.

The classification features which are derived in the previous chapter already give some insight into the differences between architectures. However, a good comparison should also be based on the performance of the different architectures. To measure the performance, computers can be benchmarked for low-level image processing. The goal of benchmarking a computer architecture was defined by Duff. His definition is (Duff 1986):

> *"The purpose of benchmarking a computer architecture is to establish a figure of merit for the architecture with a view to justifying a particular design strategy."*

Benchmarking should be tuned towards reaching that goal. The design strategies have been treated in the previous chapter. What a good figure of merit is, and how it ought to be established is the topic of this chapter.

3.1 Different approaches to a good comparison

Architectures for low-level image processing are compared on the basis of their performance for this kind of processing. Two different approaches by which this comparison is done could be distinguished:

1. *Task analysis.* Architectures have to perform a low-level image processing *task* (a typical case), and are compared on the basis of their performance.
2. *Operations analysis.* The comparison is now done on the basis of the architectural performance for specific low-level image processing *operations.*

The *tasks* or *operations* should be chosen such that they represent low-level image processing. How this can be done, what figure of merit or performance criterion is used, and which parameters should be taken into account is discussed below.

3.1.1 Task analysis

3.1.1.1 Choice of the tasks

As indicated above, the *choice* of the typical case should be such that it represents low-level image processing. This goal can not be reached by taking only one task. The problem which arises in the choice of the typical cases is: "What sort of tasks are representative for low-level image processing". Depending on the field of application in which image processing is used, this will differ. There have been suggestions for typical image processing tasks in the literature, but they can not always be called low-level. For clarity, we propose the following definition of a low-level image processing task:

"A low-level image processing task is a task which in principle can be performed using a series of exclusively low-level image processing operations".

The definition of low-level image processing operations will be given more precisely in Chapter 5, Section 5.1. They comprise the following: Pointwise operations, local neighbourhood operations, recursive neighbourhood operations, object operations, global operations, geometric operations and statistical operations (although the last operation group is actually part of low-level image *analysis*).

Table 3.1 shows image processing tasks which were defined at the Tanque Verde Benchmark Suite and the DARPA workshop (Rosenfeld 1987; Weems et al. 1989a). It is also indicated in the table which tasks are low-level according to our definition and which are not. The table also shows tasks which can not easily be classified as belonging to low-level image processing (indicated by a dash).

Table 3.1 Image processing tasks from some benchmarks

Tanque Verde task:	low-level:	DARPA task:	low level:
Edge finding	yes	11*11 Gaussian conv.ff of 512*512*8 bit image	yes
Line finding	yes	Detection of zero crossing in a diff. of Gaus. im.	-
Corner finding	yes	Construct and output border pixel list	-
Noise removal	yes	Label connected components in a binary image	yes
Generalized Abingdon cross	yes	Hough transform of a binary image	yes
Segmentation	-	Convex hull of 1000 points in a 2D R (real space)	-
Line parameter extraction	no	Voronoi diagram of 1000 points in 2D R	no
Deblurring	yes	Minimal spanning tree across 1000 points in 2D R	no
Classification	no	Visibility of vertices for 1000 triangles in 3D R	no
Printed circuit inspection	no	Minimum cost path through a weighted graph of	
Stereo image matching	no	1000 nodes of order 100	no
Camera motion estimation	no	Find all isomorphisms of a 100 node graph in a	
Shape identification	no	1000 node graph	no

A ranking between the tasks shown in Table 3.1 or any other tasks defined in the literature on the basis of importance in image processing has not been done.

Some other difficulties in the comparison of low-level image processing architectures are shown in the next sections.

3.1.1.2 Task description

The task which is done by the low-level image processing architecture can be described on different levels of detail. It is argued by Preston, that typical cases for image processing should be taken (Preston 1981). Duff states that it should be described how the task must be solved (Duff 1986). Uhr softens this, by supplying a so called preferred method (Uhr 1988). The "benchmarker" is required to program according to the preferred method, but he may also give an alternative to it. If his machine is not capable of performing the preferred method, then it should explicitly be stated why.

According to Duff, a good task definition should also supply the test data on several formats (Duff 1986).

3.1.1.3 Problems with data and architecture size

Special architectures generally have a restriction of resolution or precision. Processor arrays are available at the moment with sizes up to about 512*512. Image data which is equal to or smaller than this array size may be processed in a straightforward way, but larger data have to be processed by some kind of scanning technique. Such a facility may

not always be available for the architecture, so that performance can not be measured for tasks which work with data too large for the array. By defining tasks which go beyond the restrictions imposed by the architecture, comparisons may not be fair.

We think, that if special architectures have restriction on resolution or precision, facilities should be made to process the required precision or resolution. If tasks are realistic low-level image processing problems, it is unrealistic for a low-level image processing architecture not to be able to do such a task. Therefore, the performance of the architecture should be measured, even in the case of a mismatch between data and architecture size.

3.1.1.4 Programming

The use of the programming time per task was suggested and criticized in the literature (Preston 1981; Duff 1986). The programming time depends on the programming language, the architecture used, but also on the programmer cleverness. This programmer dependency was found to be a special drawback. For the purpose of comparing *architectures*, only their influence on the programming language and programming ease should be considered.

There is a trade-off between software reliability and the speed of the software, because fast straight-forward routines can be written specifically for benchmarking, while leaving out the code that checks for special or erroneous cases (Duff 1986). Do the programs in the comparison have to be reliable, or just fast?

This problem can be solved by requiring that comparisons should be done solely with routines from standard libraries belonging to the machines investigated. It is expected, that these routines are written so that they perform their task as fast as possible, while still capable of treating errors in a sensible way.

3.1.1.5 Data Input / Output

It is questionable whether the time needed to load or unload (image) data should be taken into account for the calculation of the processing time. There are many ways to input data into a machine:

- Frame-grabber at video speed providing raster scan data input
- Line scanner providing one line (row or column) of data at a scanner dependent speed
- There are sensors which give 2D data in parallel
- Memory input, limited by the memory access times
- Disk-input, limited by the disc-access times

If the task which is programmed on an image processor requires data to be input in any way, then the time for the fastest available method of inputting data should be taken into account for the total processing time. A processor array would presumably get its data from a 2D parallel sensor, but a pipeline would benefit from using a raster scan device.

Data input times will not have to be taken into account, if a certain low-level image processing task requires its input image to be taken from local memory. Data output can take place to a display, or to memory. If data output is part of the task, it should also be incorporated into it.

3.1.2 Comparison on the basis of operations

3.1.2.1 Low level image processing operations

The comparison between low-level image processing architectures takes place on the basis of groups of operations. The grouping of operations can be done in a number of dif-

ferent ways:

- On the basis of semantics (i.e. noise filtering, edge detection, segmentation etc.)
- on the basis of the way in which they are calculated (i.e. how many source pixel points are necessary for the calculation of one destination pixel etc.)

For the purpose of comparing architectures, the grouping on the basis of the way in which operations are calculated is better suited. Architectures are usually designed to work for several calculation methods.

A good classification of the available operations for low-level image processing is therefore based upon the following points (the operations are formally introduced in Chapter 5, Section 5.1):

- *Spatial parallelism.* Depending on the number of input (or output) pixels necessary for the calculation of one output pixel, there can be spatial point, local neighbourhood, object, recursive neighbourhood or global operations. The number of input pixels for the calculation of one output pixel is one for spatial point operations, a fixed number of neighbours for local neighbourhood operations, a data dependent number of object pixels for object operations, and is the image size for global operations. Recursive neighbourhood operations also require a set of output pixel neighbours for the calculation of a resultant pixel (see Chapter 5).
- *Pixel parallelism.* Depending on the number of bits in a pixel necessary for the calculation of one bit in the output pixel, there can be pixel point, recursive (propagate) or global operations. The number of input pixel bits for the calculation of one output pixel bit is one for pixel point operations (such as the Boolean *or* between two images), and all for pixel global operations (such as the pixel wise multiplication of two images). Pixel propagation operations (such as the addition of two images) require one bit from the input pixel and one lower or higher significant bit from the output pixel for the calculation of a resultant pixel bit (for a full explanation of these operations, see Chapter 4, Section 4.6).
- *Image parallelism.* The number of necessary input images determines whether low level image processing operations are monadic, dyadic etc. With dyadic operations there may be a difference in the spatial parallelism of the two input images. The operation may require, for example, a local neighbourhood from one image and a point from another. The number of output images is generally one, so this is not often used as an operation distinguishing feature.
- *Output data structure.* Although low level image processing operations always have an image as output entity, they may have to use some low level image analysis operations for the parametrization of the operation. Contrast stretching for instance is a spatial point operation which first requires the calculation of a histogram. The output structure of a low level image analysis operation may be a single number (pixel count) or an array (histogram).
- *Isotropy.* If the calculation of an output pixel value is done using input pixels whose position relative to the output pixel's position is not the same for the whole image, then the operation is anisotropic. An example is the set of geometric transformations, for example, rotation and warping (but not translation).

On the basis of the classification points which are described above, Table 3.2 lists sev-

eral low-level image processing operations. The number of possible combinations of the classification points which are defined above is 5•3•2•3•2 = 180. Many of these combinations may not occur. The combinations are ordered into eight different groups, which are also indicated in Table 3.2.

Table 3.2 Operation taxonomy and grouping

Operation:	Parallelism			Output entity	Isotropic	Group
	Spatial	Pixel	Image			
Boolean NOT	point	point	monadic	image	yes	PO
Increment	point	propagate	monadic	image	yes	PO
Square root	point	global	monadic	image	yes	PO
Boolean AND	point	point	dyadic	image	yes	PO
Image Addition	point	propagate	dyadic	image	yes	PO
Image Multiply	point	global	dyadic	image	yes	PO
Translation	Point	point	monadic	image	yes	PO
Boolean percentile	Local	point	monadic	image	yes	LNO
Uniform filter	Local	propagate	monadic	image	yes	LNO
Convolution	Local	global	monadic	image	yes	LNO
Distance transform	Object	propagate	monadic	image	yes	OO
Convex hull	Object	point	monadic	image	yes	OO
Median root	Recursive	global	monadic	image	yes	RNO
Walsh Hadamar Transform	Global	point	monadic	image	yes	GIO
Fourier Transform	Global	global	monadic	image	yes	GIO
Pixel count	Global	propagate	monadic	scalar	yes	SSO
Histogram	Global	propagate	monadic	array	yes	SVO
Rotation	Local	propagate	monadic	image	no	GeO

The abbreviations for the groups are defined with respect to the classification points as follows:

- *Point Operation (PO).* All isotropic spatial point operations with image output
- *Local Neighbourhood Operation (LNO).* All isotropic spatial neighbourhood operations with image output
- *Object Operation (OO).* All isotropic spatial object operations with image output
- *Recursive Neighbourhood Operation (RNO).* All isotropic spatial recursive operations with image output
- *Global Operation (GlO).* All isotropic spatial global operations with image output
- *Geometric Operation (GeO).* All anisotropic operations with image output
- *Statistical Scalar Operation (SSO).* All isotropic operations with scalar output (where the scalar output may be any type of number, even an address)
- *Statistical Vector Operation (SVO).* All isotropic operations with vector output

Grouping is done because, in the comparison between low-level image processing architectures, it is sometimes relevant to refer to broader operation classes than those defined by the five operation classification features. Also some of the group names and definitions relate to the generally accepted names in literature (Levialdi 1988).

A mathematical definition for low-level image processing operation groups will be given in Chapter 5, Section 5.1.

3.1.2.2 Image processing operation ranking

There is no generally accepted ranking of image processing operations as to their importance. The ranking of operations cannot be objective, due to a number of difficulties.

- The application area more or less determines what operations are required.
- The availability (or lack) of operations determines what other operations are needed; operations are therefore not uncorrelated.
- The architecture will also determine what kind of operations are available and which are time consuming.

The first difficulty can be addressed by dividing image processing into a number of application areas and giving an operation ranking for each. The other two problems cannot be solved by ranking specific image processing operations, but by ranking on more general grounds. For example, noise filtering or edge detection may be ranked, so that it may differ from system to system how these low-level image processing problems are translated into operations. Noise filtering could be done with a gaussian or median filter, and may even be done using Fourier transforms. All depending on what is most effective, and what can be done within the available time.

3.1.2.3 Theoretical comparison

In a theoretical comparison architectures are compared on the basis of a theory which describes some facets of that architecture. It is noted by one author, and we agree, that "performance guessing will give too optimistic results" (Duff 1986). Another author is strongly in favour of building up mathematical descriptions of architectures, and doing comparisons on the basis of theoretical considerations, and checking the results for some cases (Uhr 1988). Advantages of doing comparisons with some kind of mathematical architecture description are:

- It takes less money to compare descriptions instead of hardware.
- A good, yet not too complicated description of an architecture may give more insight in the advantages and bottle-necks of the architecture.

To summarize, describing an architecture in a careful way is not the same as guessing.

3.1.2.4 Individual instructions

Two factors in the comparison of architectures are the number of clock-cycles per instruction and the power of each instruction. An instruction is executed by a processing element. The highest instruction power is reached by a processor which takes one clock-cycle for any image processing operation. This can, for example, be done by filling a lookup table with the results of an operation. Then there are processors that take one or a few clock-cycles for commonly used instructions and more for those which are less frequently used. In the comparisons therefore the instruction power should be taken into account. One way to do this is by defining the *instruction overhead* as the quotient between the average instruction time and the clock-cycle time.

The *power* of each individual instruction can be seen from the number of clock cycles needed to perform the instruction. This can simply be a design choice, but it can also be related to the architecture itself. An example of the last is found in the full-size processor arrays. As these arrays consist of many processing elements, designers generally give them limited functionality per clock-cycle. Any function can often be done within a number of clock-cycles.

It is argued that in research comparisons, these kinds of technology dependencies should be removed (Duff 1986). We agree with this in the case that individual instructions differ only due to technology dependencies. Instruction lengths can also differ due to architectural differences, e.g. an ALU versus a lookup table. As we are comparing architectures, instruction differences should be taken into account in those cases.

3.2 Performance criteria for the comparison

The measurement of an architecture's performance may incorporate a number of quantitative and qualitative items:

1. Processing time
2. Number of processing elements
3. Size of the data
4. Costs of the architecture in terms of chip area, number of transistors, or plain dollars
5. Data I/O
6. Flexibility in image size
7. Flexibility in neighbourhood size and shape
8. Programmability

The problems associated with these performance measures will be discussed in the next sections. It is also of interest to see if it is possible to define one or more figures of merit from the qualitative performance items.

3.2.1 Processing time

There is much discussion on how the processing time should be taken into account in a fair way. On one hand this discussion is related to individual instruction considerations. On the other hand, there is the difference between the processing time per task and per low-level image processing operation. Preston suggests to use the processing time of a task (a typical case) in a comparison (Preston 1981). Duff considers a comparison based on the processing time of a task as "naive" (Duff 1986). He lists a number of problems which influence the processing time of a task, but which are not dependent on the architecture. A programmer may for example choose a clever algorithm, and program it in a highly optimized way without any error checking. As discussed earlier in this chapter, this problem can be overcome by allowing only library routines to be used. A library routine is expected to contain enough error checking on one hand and be fast on the other hand to be considered acceptable in the context of performance measurement.

Good usage of library routines (for task analysis) is encouraged by supplying 'carefully specified' preknowledge of an image processing task. A library may, for example, contain more versions of an iterating image processing routine, where one iterates until an end condition is met, and another iterates a specific number of times. The first is robust and slow, whereas the latter is fast but requires special attention to choose the number of iterations correctly. In this case the fast routine may only be used, if the specified preknowledge allows it. The necessity of supplying preknowledge of an image processing task in a careful way can be illustrated by the well-known Abingdon Cross benchmark (Preston 1989). Although "...participants [of the benchmark] have been permitted to use the following a priori

knowledge: (1) mean value of the noise, (2) signal-to-noise ratio, and (3) size of the cross relative to the size of the image.", most of the participants of the benchmark also used the information that the shape is rectangular, and in a certain orientation and position within the image. Carefully supplied preknowledge should be complete in the sense that there is no ambiguity in implied preknowledge.

3.2.2 The number of processing elements

The number of all PEs belonging to the investigated architecture should be used in the comparison. This, in combination with the processing time, will make it clear how efficient the implementation of the image processing algorithm is on the investigated architecture.

3.2.3 The size of the data

In determining the performance of an architecture the amount of data which is processed must be taken into account. With the discussion on the figures-of-merit it will become clear, that there is no general agreement on *which* data size to use in a comparison: the number of pixels or the number of lines in an image. We propose to consider the total amount of data in comparisons. This means that not only the number of pixels should be considered, but also their depth (i.e. the number of bits per pixel).

3.2.4 Cost of the architecture

Should the price in dollars of an architecture be considered? This can only be the case for architectures where the price is representative for the architecture. This is in general not the case. The price of a product is determined by more things than the complexity of the product, e.g. the availability of other, competitive products. It is also not fair to determine the price of architectures which are designed at a university. Sometimes these architectures are never sold, so that no price is available. If they are commercialized, however, the development cost of the architecture will not (completely) be taken into account when the price is determined.

Instead of the real cost in dollars, Uhr suggests a scheme to quantify the cost for a system in a more objective way (Uhr 1988). First of all, the architecture is to be described in terms of *basic resources*, e.g. processors, memories, controllers, switches, wires etc. Second, these basic resources are to be described in terms of *primitive devices*, e.g. number of transistors or chip area. Third, costs might be associated with these resources. By measuring the chip costs only on the basis of the *number* of basic resources, the architecture complexity is not completely described. One other factor is the regularity in the components used for the architecture. The scheme can be reasonably objective by demanding that it is used only in the comparison of architectures which have a large number of equal PEs. In that case it offers a clear and objective approach for the quantification of architecture cost/complexity. However, the calculation of the costs for the different architectures requires deep insight into the resources they use.

3.2.5 Data I/O

If a low-level image processing architecture is fast in itself, but has no possibilities for fast connections to and from the outside world, then it is not that useful. An example of this is found in the Transputer[1] (INMOS 1988). Although the speed of the (floating point)

1. An example of an image processing system that uses the Transputer as PE is found in the Varsytec VMTM. This system integrates four Transputer chips.

PE is comparable to the 68000 microprocessors, the data communication to neighbouring PEs goes through serial links. These links operate an order in magnitude slower than the data access (i.e. 10-20 Mbits/second as opposed to 26 Mbyte/second). The time that the data I/O takes can be 'measured' by stating what types of I/O the architecture can accept. These types are:

- *Raster scan* (order N^2). Commercially available camera's and frame grabbers provide the image data at video speed (25 or 30 frames per second) on a pixel by pixel basis, starting with the top left, and ending with the bottom right. If an image is loaded from host memory into the (parallel) computer memory, it is generally also provided in the same raster scan order. The speed is then limited by the memory access time. Raster scan image data input can also be read directly from a disk, so that the speed is limited by the disk-access time. Note that the required bandwidth[1] between the sensor and the architecture is only one pixel wide.
- *Column* or *row parallel* (order N). Although not widely spread, there are line scanners available which provide the data of one complete image line in parallel (Forchheimer and Ödmark 1983). Row parallel data input is also achieved when raster scan camera's are equipped with a line buffer, which is filled sequentially, but can be read out in parallel. Both the sensor as well as the architecture need to have an I/O bandwidth of order N pixels wide for this type of data I/O. In practice this means, that the sensor should either be close to the processing hardware, or the two are integrated in one housing.
- *Image parallel* (order 1). There are some 2D sensors available which can be read out in parallel (Garda et al. 1988). Image parallel data input can also be achieved by loading a complete image through shift registers, and then providing all the data of the shift registers in parallel. The consequences of using this type of data I/O for the bandwidth which is required between the sensor and the architecture are even larger than for column parallel data I/O. The bandwidth between sensor and architecture should be of order N^2. This effectively means, that the sensor should be integrated with the architecture to achieve this type of data I/O.

When determining the type of data I/O which an architecture can handle, one should look at the possibilities offered at the input/output side of the architecture, and not so much at the speed with which images are offered. The latter speed is often a problem in the design of image I/O devices, and seldom an architectural one.

3.2.6 Flexibility in image size

Some architectures are very well suited for images of a specific length, width or pixeldepth. The flexibility in image size will have to indicate whether an architecture can easily adapt to a different image length, width or pixel size. A single pipeline of bit-serial processing elements, for example, will have trouble in performing grey-value operations.

3.2.7 Flexibility in neighbourhoodsize and shape

For local neighbourhood operations and also for recursive neighbourhood operations flexibility in the size and shape of the neighbourhood is of importance. Some architectures may be very fast in processing one specific neighbourhood (such as a 3*3 neighbourhood), but much slower when other neighbourhoods have to be processed (such as a 4*2 neigh-

1. With bandwidth is meant: the amount of data that can concurrently be transported and read by the architecture.

bourhood). Some neighbourhood operations perform at their best (qualitatively) with circular or octagon shaped neighbourhoods (Verbeek et al. 1988). An investigation on the effect of the neighbourhood shape on the qualitative performance of an edge filter by Verbeek et al. showed that a circular shape of the neighbourhood outperforms square and diamond shapes, even if a full circle is approximated by an open circle or by an octagon. The easy handling of such shapes requires great flexibility.

3.2.8 Programmability

At this point it is not appropriate to compare architectures on the basis of what kind of high, intermediate or low level programming language is available for them. Instead, those aspects which are architecture dependent and which influence the possibilities for a programming language for this machine should be shown. This prevents measuring the input effort which has been made for a specific architecture. It may also be argued that the input effort in turn depends on the possibilities that the architecture offers.

Programming languages offered for image processing architectures fall into three or sometimes even four levels of increasing language-designer-complexity and increasing user-friendliness:

- *Micro assembler*. The machine language instructions which are produced from the assembler language are sometimes encoded and/or interpreted by a micro-program within the image processing machine. Some machines have facilities to change this micro program. At this level all the data and instruction paths can directly be reached.
- *Assembler*. At this level all or most of the facilities offered by the architecture can be directly programmed. The assembler for a machine reflects much of the machine architecture itself. An assembler usually lacks the possibilities to use structured programming. However, this is essential for the lifetime of programs (Sommerville 1982). At this level, there is a tremendous difference between assemblers of the different architecture groups.
- *System programming language*. Interfaces to the facilities of the architecture are offered through some functions. The hard machine level is therefore hidden from the user. At this level, there are enough possibilities to write programs in a way that they are maintainable. However, the systems programmer should have a clear knowledge of the architecture. Languages at this level are compiled to the assembler level, and then executed by the (parallel) architecture.
- *High level programming or command language*. Languages are offered which can be compiled and/or interpreted, where no knowledge is necessary of the underlying image processing architecture. Any of the investigated architectures can be run with the same language at this level.

With regard to the programmability, it is noted that no distinction can be made between architectures for their influence on high level languages, because any high level language can (by definition) work on top of any architecture. It is clear that system programming languages and assemblers differ tremendously between architectures.

3.3 Figures of merit

Many figures of merit exist to measure the performance of image processing architec-

tures. In the previous section it was noted that the image size, the number of PEs and the processing time should all be taken into account in comparing architectures. In the description of the figures of merit, we use the following conventions:

- N = the number of rows or columns in the image (the total image size is $N*N$),
- P = number of PEs,
- t = processing time,
- n = the number of clock-cycles to perform an algorithm,
- l = length of the algorithm to be performed.

The Abingdon Cross benchmark uses the *quality factor Q*, which is defined as one dimension of the image size divided by the time required to perform the benchmark task (Preston 1986; Preston 1989):

$$Q = \frac{N}{t} \tag{3.1}$$

Preston chooses only one dimension, because he argues that the operations that can be used in the Abingdon Cross are intrinsically of order N for an $N*N$ image.

The number of *pixel point operations per second* is defined as:

$$\text{pixops} = \frac{1}{\text{time_to_perform_one_pixel_point_operation}} = \frac{l \cdot N^2}{t} \tag{3.2}$$

This is used in some benchmarks to distinguish between architectures and cellular automata (Preston 1983; Preston and Duff 1984). The figure is usually seen as a constant for one machine, but sometimes it is not. It may depend on the image size or on the algorithm length. It is certainly a function of the number of PEs. The number of pixel operations per second could be used to determine whether an architecture is capable of performing a certain image processing task in time.

In theoretical comparisons yet another figure is sometimes used, *the required clock-cycle time for the processing of images at video speed* (Lougheed and McCubbrey 1980b; Gerritsen 1982). This is of course a function of the algorithm length, number of processors available, the architecture and the image size. It can be calculated by taking the frame time available for video speed (1/25th or 1/30st second) and dividing it by the required number of clock-cycles to do the algorithm:

$$t_c^{\text{required}} = \frac{\text{frame_time}}{n} \tag{3.3}$$

The *efficiency* is defined as the amount of 'work' to be done per PE divided by the number of clock cycles which where taken to do that work (Komen and Duin 1989). The amount of work is given by the number of points to be processed multiplied by the length l of the algorithm which is performed. The efficiency is expressed as follows:

$$E = \frac{lN^2}{nP} \tag{3.4}$$

The efficiency thus gives the number of pixel operations per clock cycle per PE. Just as is the case with the number of pixel operations per second, the efficiency may vary with image size and algorithm length. The efficiency measure may be used to compare different architectures for one image processing problem independent of the number of processors they use. It may also be used to look at the efficiency of one architecture for an image

processing problem at different image sizes.

3.4 Suggestion for a good comparison method

It can be concluded from the previous sections, that the road to a good comparison is not straightforward, but that there are many side roads also leading to (good) comparisons. The strategy that is taken in this thesis aims at comparing low-level image processing architectures on the basis of the sort of operations they should perform well. The next chapter will show that architectures have been compared to some groups of low-level image processing operations, but comparison for other groups have not yet been done.

It is expected that this strategy will lead to a better insight into the architectural demands of different low-level image processing operation groups.

An architecture in this research will therefore be classified as good, if it is good in performance for *different* operation groups.

Here is a short review of the points that will be and that will not be taken into account according to this Chapter and Chapter 2:

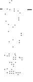
1. Architectures will be compared on the basis of the way they handle low-level image processing *operations* (cf. Section 3.1.2).
2. The operations are classified according to their
   - 2.1 *spatial parallelism,*
   - 2.2 *pixel parallelism,*
   - 2.3 *image parallelism,*
   - 2.4 *output entity type*, and their
   - 2.5 *isotropeness*, as is shown in Table 3.1
3. The groups of operations that will be considered are:
   - 3.1 Point Operations (PO)
   - 3.2 Local Neighbourhood Operations (LNO)
   - 3.3 Object Operations (OO)
   - 3.4 Recursive Neighbourhood Operations (RNO)
   - 3.5 Global Operations (GlO)
   - 3.6 Geometric Operations (GeO)
   - 3.7 Statistical Scalar Operations (SSO)
   - 3.8 Statistical Vector Operations (SVO)
4. No ranking between image processing operations or groups of operations will be used
5. The comparison is done between the following groups of low-level image processing architectures (cf. Chapter 2):
   - 5.1 Square Processor Array (SPA)
   - 5.2 Linear Processor Array (LPA)
   - 5.3 Pipe Line (PL)
6. The comparison will be theoretical, but the theory should have been checked with practice for some cases
7. The performance of the architecture will be measured in terms of

7.1 *efficiency* where there is a difference in image size, number of processors or algorithm length
7.2 *processing time* where the image size, number of PEs and algorithm length is the same
7.3 flexibility in *image size*
7.4 flexibility in *neighbourhood size*
7.5 possibilities of the *data I/O*
7.6 its *programmability*
7.7 the *costs of the architecture* will not be taken into account

For clarity in the rest of this thesis, the strategy taken to compare low-level image processing architectures can now be outlined as follows:

"Look at the speed, efficiency, flexibility (concerning image size, neighbourhood size and data I/O), and programmability of different architecture groups (SPA, LPA and PL) and for different operation groups (PO, LNO, OO, RNO, GlO, GeO, SSO and SVO)"

In Chapter 4 a start is made with the comparison according to this outline.



Chapter

4 Comparing image processing architectures

As a consequence of the suggested comparison method in Chapter 3, a start will be made with the actual comparison between the SPA (square processor array), LPA (linear processor array) and PL (pipeline) *architecture* groups. Some overhead measures used in the comparison are defined in Section 4.1. The architectures are then compared quantitatively on the basis of *data I/O* (in Section 4.2), *flexibility* (concerning image sizes in Section 4.3 and neighbourhood size/shape in Section 4.4) and *programmability* (in Section 4.4). This is followed by a quantitative comparison for the different *operation* groups as defined in Chapter 3. The comparison is done on the basis of *efficiency* and *speed*. The comparison is then closed with some preliminary conclusions in Section 4.14.

4.1 Overhead

To characterise the performance difference between architectures overhead measures are introduced. In general the overhead to do 'something' is seen as a dimensionless figure which results from dividing the time which is actually taken to do 'something' by the time which is ideally taken to do the same 'something'. Overhead therefore indicates the efficiency with which a task is done. We distinguish *instruction overhead* and *scanning overhead* and discuss each of these separately.

The *instruction overhead* (denoted by O_{instr}) relates to the time to do a 'basic' low-level image processing *operation* (such as erosion, image addition etc.). In this thesis we will assume that the ideal time to do any basic low-level image processing operation is the clock-cycle time. This choice of the ideal time is not unrealistic, because there exist PEs which can indeed do basic low-level image processing operations in one clock-cycle (i.e. the lookup-table PEs in Section 4.6). Note that with our definition the instruction overhead is expressed as the *number of clock cycles* to do a basic low-level image processing operation. For some architectures the instruction overhead is the same for all the operations that it can execute (i.e. for a pipeline with identical PEs). Other architectures, however, have a large instruction overhead for operations which are seldom used, and a low instruction overhead for operations which are frequently used (e.g. the CLIP4 PE). Note that in the latter case, the instruction overhead is not a constant, but depends on the instruction which is used. For architectures which consist of a software layer which is built on top of a hardware layer, the instruction overhead incorporates the time spent in the software layer. Note that for such architectures the instruction overhead may be different if the same hardware is used, but a different software layer on top of it (such is the case in Section 4.7.4).

The *scanning overhead* (denoted by O_{scan}) shows the efficiency with which an SPA or LPA can handle images of sizes larger than the number of PEs it has, or the efficiency with which a PL can handle programs which are longer than the number of PEs it has. For processor arrays the scanning overhead depends on the amount of information which is needed from other scans, to correctly perform an operation in one scan (e.g. for a neigh-

bourhood operation the points at the scan boundaries require the values of points in the neighbouring scan(s)). In the ideal situation no pixel values are required from other scans (e.g. for point operations). The scanning overhead is calculated as the time required to perform an operation on an image, divided by the time to perform an 'equivalent' (in the sense of processing time) point operation on the same image. Depending on the scanning method which is used, the scanning overhead may be a constant or it may depend on the number of scans, the scan size, etc.

4.2 Data input

As was concluded in Section 3.1.1.5, the time required for data input should be taken into account if a low level image processing task or operation requires the image to be taken from some kind of input device. For low level image processing architectures, at least three different forms of data I/O are used: raster scan, column/row parallel and image parallel (see Section 3.2.5).

The SPA can in principle handle all three forms of data I/O. Several existing SPAs only allow raster scan data I/O, others only column or row parallel data I/O. Only novel SPA architectures are constructed with image parallel data I/O in mind (for example the DOCIP). The CLIP4-Delft (see Section 2.4.4) uses raster scan data I/O. The PEs in the array are fed serially with the data of one bitplane the size of the array. To load the bitplanes of a grey value image, *corner turning* is necessary (Batcher 1980; Schmitt and Wilson 1989).

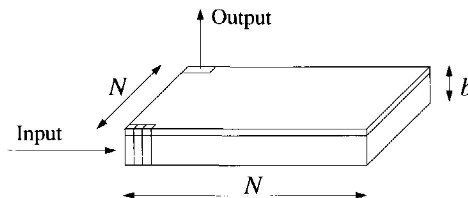


Figure 4.1 A grey value image is read in pixel by pixel, and read out bitplane by bitplane.

The data is received from the raster scan input device serially in the form of b bit pixels, as shown in Figure 4.1. However, the data has to be stored in the parallel memory of the SPA as a bitplane. Special hardware is available in the video input of the CLIP4-Delft which does this at the same speed that the video data enters the machine.

The CLIP4-UCL (see Section 2.4.4) uses column parallel data I/O (Duff 1982). This can be seen as follows. The $N*N*b$ image is first loaded into P shift registers of length P and depth b, without interrupting the SPA. Then the SPA is interrupted, so that each bitplane can be loaded in the PEs of the SPA in P shifts. The reason for calling this column parallel data I/O is, that the last phase of the I/O is done in a column parallel way. However, the P shift registers first have to be loaded with image data. This can be done column parallel when a line scan camera is used, but when a raster scan camera is used, it will happen serially.

The MPP uses a form of overlapped column parallel data I/O (Batcher 1980). The pixels of one column are shifted into a buffer at one edge of the array. The contents of this buffer is shifted in the SPA concurrently with the processing. After P shifts, all the col-

umns are at their correct position. At that moment, the processing is interrupted for one instruction cycle and the image data is read into the local memory of the PEs. The fact, that the shifting-in of image columns is overlapped with processing, is significant only when sequences of images have to be processed.

Data I/O for LPAs can in principle be done raster-scan wise, but this is not customary. Instead, row parallel data I/O overlapped with processing is used. The AIS-5000 is a bit serial LPA (see Chapter 2), which allows a line of image data to be shifted through fast I/O registers of the PEs concurrently with processing. After one line is entered, the processing is interrupted for b cycles to load all the pixels of a b-bit image line into the local memory of the LPA. Grey value LPAs such as the SLAP allow one line of grey value pixels to be read in just one clock cycle (Fisher et al. 1987). Similar to bit serial LPAs, the line is first shifted into a buffer serially and then transferred to the LPA memory in parallel.

The PL only uses raster scan data I/O. In a pipeline, the data I/O is usually *overlapped* with processing. This depends on the speed of the specific PL. Sometimes it is necessary to buffer the data I/O through a line or image buffer. The speed of the data input is then the speed with which the buffer can be read and is thus independent of the actual input device.

Evaluating the data I/O possibilities for the different architecture groups, we draw the following conclusions:

- *Possible speed.* Only a full sized SPA allows image parallel data I/O in principle, which is theoretically the fastest of all three I/O methods. Both SPA and LPA allow the next fastest column/row parallel data I/O. The slowest data I/O can be done by all three architecture groups.
- *Actual speed.* The data I/O method which is actually encountered in an SPA is the column parallel way. Existing SPAs do not have image parallel I/O. Actual implementations of the LPA and the PL are indeed encountered with row parallel and raster scan data I/O respectively. The SPA and LPA therefore offer the fastest *actual* data I/O possibilities.
- *I/O devices.* The speeds offered at the output of I/O devices are sometimes less than the speeds allowed at the input side of low-level image processing architectures, because the necessity to read image data into the architecture is larger than the necessity to display images (in object identification and position calculation for robot vision for example, no displaying is required). The problem is, that it is expensive to increase the bandwidth of I/O devices.

4.3 Flexibility in image size

An SPA which is small with respect to generally used image sizes can handle images of different sizes because of its relative smallness. Unfortunately, this flexibility in image size is at the cost of overhead due to the scanning of a small array over a large image (see Section 4.1). If SPAs are designed to operate on images which are smaller than the SPA, no provisions are needed for scanning. Such SPAs, however, do not offer much flexibility with respect to image size.

The bit-depth of an image is no problem for a bit serial SPA. The bitplanes of one grey value image can be stored in local memory. Grey value operations on images have to be broken down into binary operations on the different bitplanes. Only for pixel global grey value operations (such as pixel by pixel multiplication, or taking the square root of every

pixel, see also Table 3.2) is the bit serial SPA less efficient than an SPA which contains grey value ALUs. However, the bit serial SPA allows the efficient use of any pixel size image. An eight-bit grey value PE is, for example, less efficient in performing pixel point or pixel propagate operations on a four-bit image, than four bit-serial PEs.

The LPA can handle wider images by processing strokes, at no extra cost (see Section 2.3.2). It can handle any height image due to its one dimensional structure. The difference between bit-serial and grey-value LPAs is the same as between bit-serial and grey-value SPAs (see above). The bit-serial LPA is more efficient in performing pixel point or pixel propagate operations on any pixel depth image. Pixel global operations are done more efficiently on a grey-value LPA than on a bit-serial LPA. There are also LPAs which work with floating point numbers (e.g. the PICAP3).

The PL has the greatest flexibility in image size. Images with any width or height can be processed, provided that the line buffers necessary to construct the correct neighbourhood can be programmed for any desired length. The PL may have great problems in varying image depth. Pipelined architectures are usually constructed for a specific pixel size. Pipelines which are constructed with b bitstreams of bit serial PEs may handle larger pixel sizes by using carry recirculation. The carries of the least significant b bits of all image points produced in one pass of the image through the PEs are stored in a memory. A next pass through the pipeline processes the most significant b bits of the image points, with the buffered carries as input (see Section 2.3.3, see also Jonker et al. 1989). Although this is in principle possible, no pipeline is known at the moment which implements this pixel bit flexibility.

4.4 Flexibility in neighbourhood size and shape

Although an SPA is usually made for one neighbourhood size, it can handle any neighbourhood in principle, by shifting planes and then performing operations. Flexibility is thus gained at the cost of speed. An SPA which stores images crinkle-wise offers larger neighbourhood connectivity, and therefore larger flexibility in neighbourhood size and shape.

The LPAs offer greater flexibility towards neighbourhood size and shape in less time than the SPAs, which can be seen as follows. As was shown in the conclusion of Chapter 2, an LPA offers order N neighbourhood connectivity, because it can access any neighbour along one or more columns within one clock cycle. Extending the neighbourhood connectivity from three to even more columns is not as expensive in the number of required PE-PE connections as extending neighbourhood connectivity on SPAs, for example, from 3*3 to 5*5. The SYMPATI-2, for example, has a neighbourhood connectivity of $7{\bullet}N$, because it can access neighbours over a distance of 3 PEs within one clock cycle. Additional neighbours in horizontal direction will have to be fetched by a sequence of local shifts, as is done with the SPA.

The neighbourhood size for a given PL is usually fixed. Larger neighbourhoods or neighbourhoods with different shapes can be constructed by building them up from one-step shifts in the directions offered by the neighbourhood connections. Increasing the neighbourhood size from 3*3 to 5*5 also increases the latency[1] between two pipeline stages. Instead of two line buffers, four are needed, and the latency increases from $N+1+O_{instr}$

1. The pipeline latency is defined as the number of clock-cycles between two stages of a pipeline plus the instruction overhead.

to $2N+1+O_{instr}$ (where O_{instr} is the instruction overhead as defined in Section 4.1). It can be expected that even the instruction overhead will increase with this neighbourhood size enlargement. For 3*3 binary LNOs a lookup table from 2^9 bits is sufficient, but for a 5*5 binary LNO, a lookup table of 2^{25} bits would be required. Thus, it may be realistic to assume table lookup operations for binary 3*3 LNOs, but for larger environments ALUs will more likely be used. This in turn means, that the instruction overhead increases. Without changing the pipeline latency from a pipeline for 3*3 LNOs, the nine elements of the neighbourhood could be 'tapped' at different positions from the line buffers. Neighbourhood shape flexibility can thus be reached to some extent using little effort.

4.5 Programmability

Regarding the programmability of the three architecture groups under investigation, only those architectural items will be considered which have an influence on the *system* programming language (see Section 3.2.8). It is assumed that an architecture will have minimal (ideally none at all) influence on the progammability of a higher level programming language, because such a programming language should be architecture independent. The architectural consequences for low level programming languages such as assembler or micro assembler are not considered. It is assumed that the intermediate level system programming language hides the assembler level from the programmer.

The programming language is thereby mainly influenced by the possibilities of the individual PEs. LPAs or PLs are sometimes equipped with lookup tables which serve as the processing heart of their PEs. The programming languages of the machines which offer these possibilities should thereby also offer tools for constructing lookup tables in a user-friendly way.

Writing image processing routines for a pipelined machine requires that the pipeline length be taken into account. It is worth the effort in programming time to fit a routine in the number of operations offered by a pipeline. If routines extend beyond the pipeline length, then frame recirculation has to be done, which slows down the pipe considerably. There are also other factors which might prevent a pipeline algorithm from being executed in one pass through the pipeline. An example is that the type of one operation has to be chosen on the basis of a result from a previous operation, e.g. a histogram is made (which takes one complete pass through a pipeline), on the basis of which a threshold value for thresholding the image is selected.

In general it can be said, that the data size flexibility offered by a pipeline is at the expense of flexibility in the instruction sequences. In contrast, an SPA has large flexibility in the instruction sequence, but less in the data size.

When bit serial PEs are used in LPAs and SPAs, software should be written on the level of the systems[1] programming language to implement grey value image processing. Because high level languages are presumably written using the system programming language level, the user of a high level language is protected from the necessity to know the structure of the underlying machine in too much detail.

1. Writing such software in lower levels may be advantageous for the speed, but not for the maintainability of the software.

4.6 Point operations

Efficiency was defined in Section 3.3, Equation (3.4) as the processing time for one PE to produce one pixel. If P equal processors are available and they can be arranged as either SPA, LPA or PL, then their efficiency for binary point operations is about equal. In fact, their efficiency is also about equal for grey value point operations. The exact efficiencies can be calculated by using the formulas for the processing time for LNOs (see Equation (4.16)) with all scanning overhead (see Section 4.1) set to one (there is no scanning overhead for point operations), into Equation (3.4). The difference in performance over the point operations depends more on the type of PE used. To understand this, some more insight into the nature of grey value operations is necessary.

Grey value (spatial) point operations can be separated in pixel point, pixel propagation and pixel global operations, depending on the number of source pixel bits that are necessary for the calculation of one destination pixel bit. This is illustrated in Figure 4.2, and examples of grey-value pixel operations can be found in Table 3.2.

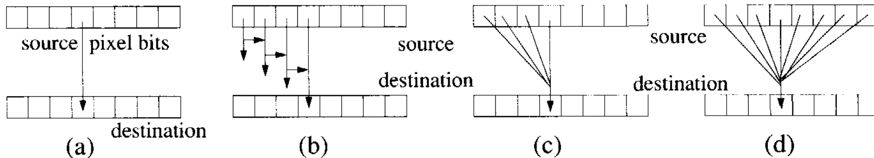


Figure 4.2 Types of grey value spatial point operations are distinguished by the number of source pixel bits necessary for the calculation of one destination pixel bit: (a) pixel point, (b) and (c) pixel propagation, (d) pixel global.

An example of the pixel point operation is the boolean inversion, where each destination bit is the inverse of a source bit (Figure 4.1a). Examples of pixel global operations are square root, trigonometric functions etc. (Figure 4.1d). Incrementing or decrementing all pixels in an image are examples of pixel propagation operations. Such operations can be done by calculating a destination bit from the source bit on the same position and a carry propagate bit from a position which is lower in significance (or sometimes higher). In this way results from operations on less significant bits propagate to more significant bits (Figure 4.1b). These operations can also be calculated in one step, by calculating one destination bit as a result of all source bits which have the same or lower (sometimes higher) significance (Figure 4.1c).

4.6.1 The use of grey value PEs

Several types of processing elements exist in low-level image processing architectures to support grey-value point operations. The fastest and most versatile implementation is the Point Operations Ram (Figure 4.3a). One or two grey-value source pixels (for monadic/dyadic operations) form an address into a random access memory device, which results in the output of the data at that address. The POR first has to be loaded with a specific point operation. If a precision less than or equal to eight bits is sufficient, a 64kbyte RAM device can be used. A double precision of 16 bits however, requires a 4Gbyte RAM device, which is unrealistic. The load-time for one point operation is proportional to the POR size, which makes it an unsuitable device for fast image processing architectures. For monadic image processing, PORs are used in the Cyto-HSS pipelined computer (Lougheed 1987). In such an application PORs are practical, as an 8-bit monadic POR only needs 256 entries, which

can be loaded quite fast.

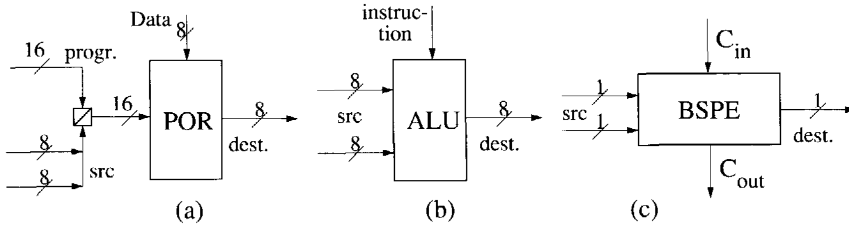


Figure 4.3 Types of PEs for grey value point processing: (a) Point Operations Ram, (b) Algorithmic and Logic Unit, (c) Bit Serial Processing Element

The second PE-type in use for grey-value point operations is an ALU (Figure 4.3b). Two source pixels of 8-bit (or higher, depending on the ALU) are presented to the PE, one of several available operations is executed. ALU-type PEs are used for example in the SYMPATI-2 (8/16 bit), the SLAP (20 bit) and the PICAP3 (32 bit floating point), which are all linear processor arrays. The Cyto-HSS also makes use of an ALU for dyadic grey value point operations called 'image combiner'. This ALU is incorporated in the pipeline as one specialized processor. The advantage of an ALU is that those operations between pixel points which are required most often can be quickly programmed and executed. Less used operations require a combination of available ALU functions.

4.6.2 The use of bit serial PEs

Another approach to the processing of grey-value point operations is to use a Bit Serial Processing Element (Figure 4.3c). Two (or even more) source pixel bits are offered to the BSPE together with a carry-in bit. This then results in one or more destination pixel bits and a carry-out bit. For the pixel point operations the carry is not used.

For pixel propagation operations in a processor array configuration (be it square or linear) the carry is usually stored in a carry-register (or carry memory). The grey-value point operation is performed one bitplane at a time, starting with the bitplane where the carry propagation begins. For LPAs or for scanning SPAs two modes of operation exist. In the first mode the complete grey-value point operation is carried out for one spatial position of the LPA/SPA. Intermediate carries can be stored in a carry register. In the second mode, one whole bitplane, for which the PA has to scan the image spatially, is processed, so that the carry has to be stored in a separate plane.

Pixel global operations can be performed by doing dyadic boolean calculations on the necessary source bitplanes, storing these intermediate results in memory, and so on, until the results can be placed in the respective destination bitplanes. Random bitplane access is required for these sort of operations.

Increasing the pixel parallelism by using more BSPEs in parallel is not trivial. Increasing the number of BSPEs per pixel is suggested for the Centipede (Wilson 1988; Schmitt and Wilson 1989). The Centipede is an LPA much like the AIS-5000 (see Chapter 2) but extended with a Multiply Accumulator Circuit (MAC) per PE. For grey-value operations, 8 PEs are grouped together, and the operation for all 8 bits of the grey-value pixel is done one bit at a time. The grouping of PEs in the centipede is also used for indirect addressing (8-bit address range) and transposing an image (per block of 8*8 bits). A study has been done on the construction of bit-parallel pipelines using BSPEs (Komen and Duin 1989).

Some of the results of this investigation will be discussed here.

If a number of parallel bitstreams are used and there is no provision for random access of the bits in a pixel, several problems are encountered. Pixel propagation operations could have the carry propagate from least to most significant, or the reversed way, so that provision should be made for the carry to propagate in either way.

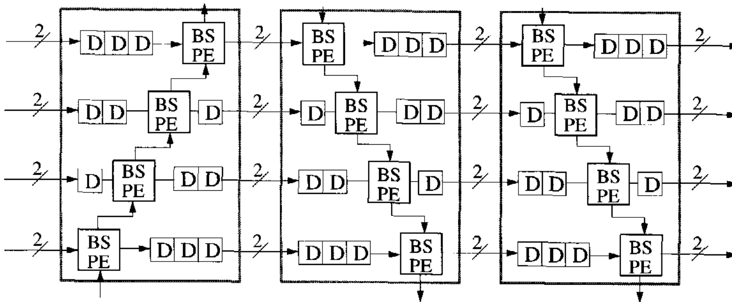


Figure 4.4 Bitstreams and carry direction specific delays in a 4-bit pipeline; the BSPEs have their own programmable delay circuitry.

To prevent a time-skewing between the bitstreams due to the carry propagation, a programmable amount of extra delays could be included between pipeline stages, to take care of the time-skewing (see Figure 4.4). A drawback of this is the increase in the latency (delay) between the pipeline stages. A pipeline which is fed raster scan wise, and which offers a 3*3 neighbourhood has a delay of $N+1+O_{instr}$ between any two stages. Here O_{instr} (i.e. the instruction overhead as defined in Section 4.1) is the number of clock cycles required to perform one operation within a stage. If a straight forward table lookup type BSPE is used, then the instruction time overhead is one. However, due to the extra delays which have to be inserted for the resolving of the time skewing between the bitstreams, the instruction time overhead is increased with b-1 clock cycles (b is the number of bits per pixel). Note, that the speed with which completely processed pixels exit this type of pipe is the same as with a single bitstream pipeline.

Alternatively, carries can be calculated asynchronously, so that they can ripple through all the BSPEs which are processing one pixel within one clock-cycle. This can be categorized as asynchronous carry look ahead. The operation of this type of carry calculation circuitry is explained in Figure 4.5. The principal consideration for this carry circuitry is, that a carry can be determined as a point operation in the spatial sense from up to two input bitstreams and from the more or less significant carry in. When a cellular logic pipeline element like the CLPE (see Chapter 2) is used, then the carry can be calculated before the bitstreams enter the CLPE. Figure 4.5a shows how two combinatorial circuits are used for the carry calculation. One is the Carry Select (CS) circuit, which is programmed with one bit to indicate that the carry flows upwards or downwards. The inputs of the CS are therefore the carries from the more and the less significant BSPEs. The output of the CS is the carry-in signal to the CLPE and to the second combinatorial circuit. This is called the Carry Function (CF) unit. Two of the image bitstreams together with the carry in signal enter the CF. Eight control lines allow any Boolean function between 3 input variables to be performed. The resulting carry output is fed to both the more and the less significant BSPEs. In Figure 4.5b it is shown, how the carries of one stage of a 4-bit pipeline flow between BSPEs.

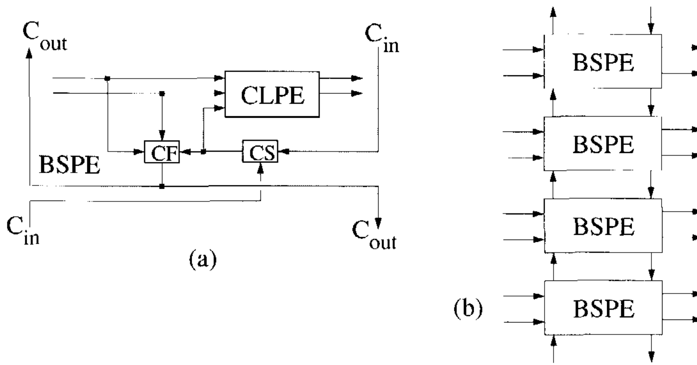


Figure 4.5 Asynchronous carry implementation, (a) in one BSPE containing a CLPE, and (b) connections for a 4-bit pipeline stage.

The pixel global operations require the use of all pixel bits for the calculation of one destination pixel bitstream. An example of such an operation is the multiplication of two pixels. If there is no interconnection network between two stages, then the individual bitstreams have to be directed towards the destination pixel bitstream through the carry circuitry, one position within a pixel per stage.

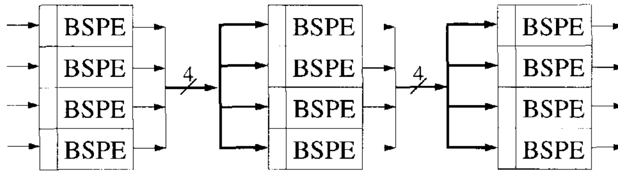


Figure 4.6 Interstage interconnection for one of the bitstreams per BSPE through a programmable "1 out of b" (b=4 here) selector per BSPE.

There exist other, more flexible facilities for the interconnection between the bitstreams of two stages. A very powerful interstage facility is offered by the *POR* (point operations RAM). Although such a device could be used for the programming of any point operation in the spatial sense, a $2b$ bits input, b bits output configuration or dyadic point operation requires 2^{2b} clock-cycles to load every combination of the input bits with the required output combination. Extension to more pixels with more bits requires the replacement of PORs with larger ones. The addition of a programmable *crossbar* between stages in a pipeline would enable every bitstream to have random access to a bitstream of the previous stage. However, the programming of these crossbars would take extra time and effort. A very simple and modular solution would be to equip every BSPE with a one-out-of-b selector as shown in Figure 4.6. One of the inputs of the BSPEs in one stage receives one of the outputs of all BSPEs in a previous stage. The input selector of each individual BSPE can be programmed such that a bitstream is taken from any of the BSPEs in the previous stage.

4.6.3 Conclusion

For *grey-value point operations* we can conclude that a grey-value PE (an ALU) is best with regard to processing time efficiency and programming effort. This points to the use of an LPA or PL, or a small scanning SPA, as large SPAs will not have enough space to incorporate grey-value PEs. Bit serial LPAs, extended with grey-value processing capabilities

as shown for the Centipede, can also be used.

At this point it should be realized, that a low level image processing architecture is usually made with an emphasis on local neighbourhood operations, i.e. connection in the spatial sense. In the design of low level image processing architectures, a trade off is usually made between neighbourhood connectivity/parallelism and pixel bit parallelism, in favour of the first.

4.7 Local neighbourhood operations

As has been done in previous approaches in the literature, formulas for the processing time of LNOs using an SPA, LPA or PL will be derived. In the comparison P processors are taken working on an $N*N$ image with an algorithm of length l local neighbourhood operations. A number of factors have to be taken into account when comparing image processing architectures for local neighbourhood operations.

- *Instruction length.* The 'basic instruction' length is defined as the number of clock-cycles to do an image processing operation (in this case an LNO). This will be different from architecture to architecture. Only with full table lookup capability is the basic instruction length equal to 1. In other cases, there is an overhead of O_{instr} due to longer instructions (see Section 4.1). For PEs which do not use a table lookup mechanism to perform operations, this overhead may depend on the actual instruction (i.e. a dilation may have a different instruction overhead than a convolution). The overhead is sometimes proportional to the spatial parallelism of the architecture (i.e. the number of PEs used). Full size SPAs will have large overhead factors, whereas scanning SPAs, LPAs and PLs may have more powerful PEs and have thereby less overhead.
- *Algorithm length.* The length of the algorithm may differ from architecture to architecture. If an SPA does not have access to all members of its neighbourhood in parallel, algorithms will be longer. Due to the nature of the pipeline, all available neighbourhood elements are at hand, but a pipeline may have difficulty in accessing bitplanes in a random access way, if no crossbar switches are used (see Section 4.6). Also, a pipe breaks up if conditional processing takes place, which results in longer algorithms. We will not take this into account for the quantification of the processing time.
- *Scanning.* The necessity to scan over data or instructions when only a relatively small set of processors P is available. For an SPA this means scanning P processors over an $N*N$ set of data, which introduces an overhead O_{SPAscan} of about four, when done using hardware half scan addressing (see Section 2.3.1, and Buurman and Duin 1988). For an LPA this means that strokes of P processors have to be done one after another resulting in an extra overhead of O_{LPAscan}. For a PL this means that the image of $N*N$ pixels has to be recirculated in a memory and processors have to be reprogrammed. This takes O_{PLscan} number of cycles.
- *Data I/O.* For real-time image processing, images have to be grabbed and fed into the parallel processor system for every l operations. This introduces different additive overheads for the architectures. If corner turning hardware (see Section 4.2) is used to send an image from a raster-scan device (camera) into an SPA, this may take a maximum of $2*N*N$ clock cycles (every part of the image has to be fed into the corner turning device, and can be read out only after it is stored). There are also SPAs which are capable of handling column parallel or image parallel I/O (Gerritsen 1982). A drawback of the

massively parallel data I/O is the width of the data paths required, i.e. both the sensor as well as the array should have sufficiently large data paths. We will nonetheless assume column parallel I/O for SPAs. The LPA can overlap the shift-in time of one line with processing, and then interrupt the array to store the line in parallel memory. This will take *N* interrupt cycles for an *N*N* image. Finally the PL needs at least *N*N* cycles for handling a complete image.

The time between fetching the first pixel and processing the last pixel for an algorithm of *l* steps with *P* processors on an *N*N* image will be derived for the SPA, LPA and PL.

4.7.1 Processing time for the SPA

It was suggested in 1980, that the time for a scanning SPA with *P* PEs to process an image of size *N*N* with an algorithm of *l* steps is (Lougheed and McCubbrey 1980b):

$$t_{\mathrm{SPA}} = \text{processing_time} + \text{data_I/O_time} = l\left(\frac{N^2}{P}\right)t_{\mathrm{clock}} + 2N^2 t_{IO} \qquad (4.1)$$

with: t_{clock}= 'effective processing element cycle time, including both instruction broadcast and execution times'.

Raster scan data I/O was assumed in this analysis, which is not always correct for SPAs, as is argued above. Furthermore, no overhead was assumed for the scanning of the array over a larger image. The formula was improved by Gerritsen in his thesis (Gerritsen 1982):

$$t_{\mathrm{SPA}} = \text{program_load_time} + \text{processing_time} + \text{data_I/O_time} = \text{i_size} \cdot l \cdot t_{\mathrm{Host}} + \left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 (R + l)\, t_{\mathrm{clock}} \qquad (4.2)$$

with: $R = 2 \cdot N^2 \cdot b$, $R = 2 \cdot N \cdot b$, or $R = 2 \cdot b$, depending on raster scan, column parallel or image parallel I/O for *b* bitplanes.

In this formula it is assumed that a program of *i_size* bytes will have to be downloaded by the host into the SPA. Most of the present SPA controllers are internally pipelined, so that the program loading is overlapped with program execution. Some SPAs, however, have some *instruction* overhead, which is not taken into account in any of the above two formulas. The formula of Gerritsen also presumes the use of a bit-serial processing element (BSPE). This may not be the case.

A good formula for the processing time of an SPA will take into account the overhead due to instructions longer than one clock cycle (see Section 4.1). Further, the overhead caused by scanning a small array over a large image has to be incorporated, where appropriate. In our formulas we will assume the use of a PE with any pixel-bit parallelism, that is capable of loading one pixel of image data in one clock-cycle. We therefore arrive at the following formula for the *number of clock cycles* n_{SPA}:

$$n_{\mathrm{SPA}} = \text{scans} \cdot \text{algorithm_length} + \text{data_I/O_time} = \left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 O_{\mathrm{SPAscan}} \cdot l \cdot O_{\mathrm{SPAinstr}} + R \cdot O_{\mathrm{I/O}} \qquad (4.3)$$

with: $R = N^2$ for raster scan data I/O, *N* for column/row parallel data I/O or 1 for image par-

allel data I/O,

$O_{\text{SPAscan}} = 1$, if the number of scans $\left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 \le 1$, and some fixed value otherwise,

$O_{\text{I/O}}$ is the number of clock cycles needed to fetch one pixel from the I/O device into the SPA,

and: $n_{\text{SPA}} = \frac{t_{\text{SPA}}}{t_{\text{clock}}}$.

4.7.2 Processing time for the LPA

There are several similarities between the processing time formulas for the SPA and the LPA. Both scan with a processor array over an image and initially it seems that they differ only in the relation between their horizontal and vertical sizes. However, the fact that an LPA is a *line* of processing elements opens the possibility to scan *without* overhead. This was shown in Chapter 2. This feature also allows column parallel data I/O. A line of image data can usually be shifted in the I/O parts of the PEs without interrupting the processing (see the description of SLAP and AIS-5000 in Chapter 2). The complete line can then be shifted in the PEs. Using some line buffers, this process can run completely in phase with a raster scan input device so that no special image data input devices are necessary. The formula for the LPA processing time in number of clock cycles becomes:

$$\begin{aligned} n_{\text{LPA}} &= \text{strokes} \cdot \text{rows} \cdot \text{algorithm_length} + \text{data_I/O_time} \\ &= \left\lceil \frac{N}{P} \right\rceil \cdot O_{\text{LPAscan}} \cdot N \cdot l \cdot O_{\text{LPAinstr}} + N \cdot O_{\text{I/O}} \end{aligned} \tag{4.4}$$

with: $O_{\text{LPAscan}} = 1$, if the number of scans $\left\lceil \frac{N}{P} \right\rceil \le 1$, and some fixed value otherwise.

If overlapped data I/O is used with a row parallel input device which uses the complete allowed data width, then the processing time is further reduced. After an initial start-up of a few lines (two for a 3*3 environment), the loading of the next image line is overlapped with the processing of another image line. The time for an LPA to process an algorithm of length l 3*3 neighbourhood operations then becomes:

$$n_{\text{LPA}} = \left\lceil \frac{N}{P} \right\rceil O_{\text{LPAscan}} \cdot N \cdot l \cdot O_{\text{LPAinstr}} + 3 \tag{4.5}$$

with: $R = 3$: two for the start-up of 3*3 neighbourhood processing, and one for the overlapped processing.

4.7.3 Processing time for the PL

An initial investigation into the processing time of one specific pipeline, the Cytocomputer, was published in 1980 (Lougheed and McCubbrey 1980b). Assuming raster scan data input the processing time through a pipeline takes at least $N*N$ clock-cycles to allow the complete image to be fed into the first pipeline stage. Stage buffering is used so that a whole neighbourhood of pixels can be offered concurrently to the PE. This is called the pipeline latency. These two factors are taken into account in the formula for the Cytocomputer processing time:

$$t_{\text{Cytocomputer}} = (l\,(N+2) + N^2)\,t_{\text{clock}} \tag{4.6}$$

As noticed by Gerritsen, this formula lacks instruction load time (Gerritsen 1982). A Cytocomputer needs to load P processing elements with the necessary instructions (lookup tables) so that they will be executed. The instruction loading for a processor array could in principle be overlapped with execution. Although this is not done for the Cytocomputer, newer pipelines exist for which instruction load overlapping is possible (Kraaijveld et al. 1986; Lougheed 1987). Adapting the processing time formula for the instruction load time (a total of 768 bits per PE) leads to the following:

$$t_{\text{Cytocomputer}} = 768 \cdot P \cdot (t_{\text{Host}} + (l(N+2) + N^2)\, t_{\text{clock}}) \tag{4.7}$$

The Cytocomputer is supposed to consist of as many PEs as necessary to process an algorithm of length l. However, no Cytocomputer has been built with length greater than 88, whereas algorithms can easily be found that are larger. If algorithms are larger than the pipelinelength, frame-recirculation must be done. A dual-ported memory is then used to store the image. While the input is read from one address, the intermediate result leaves the pipe after $P \bullet (N+2) \bullet t_{clock}$ seconds, and is then stored in the memory on the same (or a different) address. After all image points have been gone through the pipeline (i.e. $N^2 \bullet t_{clock}$ seconds), it can be reprogrammed (if necessary) and the data can be fed into the pipe again.

An approach to derive a reliable formula for the processing time of a pipeline in general takes into account the cost of frame recirculation for a pipeline (Duin and Jonker 1986; Duin and Jonker 1988). This approach also considers the possible pixel-bit parallelism b of a pipeline which processes B bit pixels. This leads to the following formula:

$$t_{\text{PL}} = \left(\left\lceil \frac{l}{P} \right\rceil N^2 \cdot \left\lceil \frac{B}{b} \right\rceil + O_{\text{progr}} P\right) t_{\text{clock}} \tag{4.8}$$

The program load time is indicated by an overhead factor O_{progr}. The pipeline latency is not taken into account in this formula.

In our approach we will neglect the program load time, because it is very small in comparison with normal image sizes, and because novel architectures allow for the overlapped program loading in a pipe (Jonker et al. 1988b). Just as with the SPA and the LPA, we will not consider the pixel widths, as these are already discussed in Section 4.2 for the point operation group. We will take into account the frame recirculation and allow for an overhead on top of this too (this is called the pipeline scanning overhead). The instruction overhead for a pipeline only increases the latency. This leads to the following formula:

$$\begin{aligned} n_{\text{PL}} &= \text{stages} \cdot \text{stage_delay} + \text{frame_recirculation_time} + \text{image_load_time} \\ &= \left(l(N+1+O_{\text{PLinstr}}) + \left\lfloor \frac{l}{P} \right\rfloor O_{\text{PLscan}} N^2 + N^2 \cdot O_{\text{I/O}}\right) \\ &= \left(l(N+1+O_{\text{PLinstr}}) + \left\lceil \frac{l}{P} \right\rceil O_{\text{PLscan}} N^2\right) \qquad \text{If:} \quad O_{\text{PLscan}} = O_{\text{I/O}} \end{aligned} \tag{4.9}$$

with: $O_{PLscan} = 1$ if the number of scans $\left\lceil \frac{l}{P} \right\rceil \le 1$.

4.7.4 Verification of the theory

As was discussed in Chapter 3, the theoretical approach is only worthwhile, if it can be experimentally verified. Complete extensive verification of the performance formulas for the SPA, LPA and PL is not possible for a number of reasons.

- *Varying number of processors.* Checking the formulas for one machine with varying number of PEs is not possible, as we do not have a computer within our reach for which the number of PEs can be changed.
- *Availability of architectures.* No working LPA or PL was available to test the formulas. An SPA is available, but lacks hardware scanning features.

Nonetheless, the SPA performance formula can be checked for varying image sizes and for varying algorithm lengths with the CLIP4-Delft. This particular CLIP4 can use two scanning techniques which are implemented in software: Half Scan Addressing (HSA) and Edge Store Scanning (ESS). Both techniques were discussed in Chapter 2. Before comparing the performance of the machine with the theoretical expected performance, two overhead factors need to be determined. The instruction overhead can be calculated from the behaviour of the SPA due to varying l but with $N \le \sqrt{P}$. In that case the complete image 'fits' in the array, so that no scanning has to be done. From Equation (4.3) the instruction overhead when no data I/O is done, can then be determined as follows:

$$O_{\text{SPAinstr}} = \frac{n_{\text{SPA}}}{l} \tag{4.10}$$

The instruction overhead has been calculated for one specific LNO (a 3*3 dilation) and the following algorithm lengths:

$l = \{1, 2, 4, 8, 16, \ldots, 2048\}$.

The measured processing times for these varying length algorithms are shown in Figure 4.7a. From the calculations using Equation (4.10) on the measured data, it follows that

$$\begin{aligned} O_{\text{CLIP4DelftInstrESS}} &\approx 3354 \ \text{clock_cycles_per_operation} \\ O_{\text{CLIP4DelftInstrHSA}} &\approx 14457 \ \text{clock_cycles_per_operation} \end{aligned} \tag{4.11}$$

The difference in the instruction overhead between ESS and HSA is due to the difference in the software (i.e. the number of CLIP4 and host instructions) used to implement these two scanning techniques, which is visible even if no actual scanning is done (see Section 4.1). If hardware scanning methods would have been used, then the *instruction* overhead for different scanning techniques would have been the same.

The *scanning* overhead is determined by timing an algorithm with fixed length l=32 for various number of scans $\lceil N/\sqrt{P} \rceil^2$, without doing data I/O (i.e. R=0). Via Equation (4.3), the scanning overhead is than calculated by:

$$O_{\text{SPAscan}} = \frac{n_{\text{SPA}}}{\left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 O_{\text{SPAinstr}}\, l} \tag{4.12}$$

The HSA technique has a scanning overhead which depends on the number of scans in the horizontal and vertical direction of the image. Its timing is given by Equation (4.13):

$$n_{\text{HSA}} = (S^2 + S(S-1)\,O_{\text{HVscan}} + (S-1)^2 O_{\text{Dscan}})\,O_{\text{SPAinstr}}\, l \tag{4.13}$$

with: $S = \left\lceil \frac{N}{\sqrt{P}} \right\rceil$ i.e. the number of scans in one direction.

Note, that if the number of scans is large enough, the timing formula for the SPA as derived in Equation (4.3) can be used. For the calculation of the scanning overhead, the number of scans was varied from 1 to 256. From the measured processing times and with

the help of Equation (4.12) and Equation (4.13) it appears that the scanning overheads are equal to:

$$\left.\begin{array}{l} O_{\text{HVscan}} \approx .74 \\ O_{\text{Dscan}} \approx 3.34 \end{array}\right\} \Rightarrow \begin{array}{l} O_{\text{CLIP4Delft ESS}} \approx 2.82 \\ O_{\text{CLIP4Delft HSA}} \approx 4.5 \qquad \text{for } (N \gg \sqrt{P}) \end{array} \tag{4.14}$$

The instruction and scanning overhead factors which have been derived from the data can now be used in Equation (4.3) and Equation (4.13), so that the following formulas are derived for the HSA and ESS processing times of the CLIP4-Delft SPA:

$$\begin{aligned} n_{\text{CLIP4Delft ESS}} &= \left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 \cdot 2.82 \cdot l \cdot 3354 + R \\ n_{\text{CLIP4Delft HSA}} &= S^2 + S(S-1) \cdot 0.74 + (S-1)^2 \cdot 3.38 \cdot l \cdot 14457 + R \\ &\approx \left(\left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 \cdot 4.5 \cdot l \cdot 14457 + R \right) \quad \text{for } (N \gg \sqrt{P}) \end{aligned} \tag{4.15}$$

With: $S = \left\lceil \frac{N}{\sqrt{P}} \right\rceil$, R=0 (i.e. no data input time) and $N > \sqrt{P}$.

Now that the scanning and instruction overhead factors have been determined, it is possible to check how well the measured values match with Equation (4.15) in which the overhead factors are filled in. In Figure 4.7a the experimental and the theoretical behaviour for N=32, P=1024 and l = {1, 2, 4, 8, 16, ..., 2048} is shown. Obviously, theory fits well with practice for this processor array and for both scanning techniques.

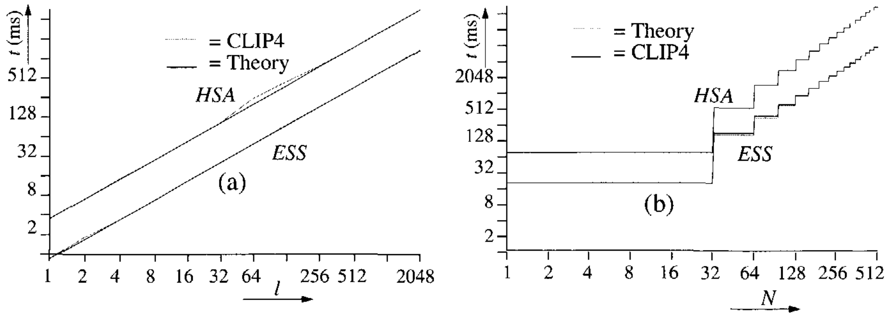


Figure 4.7 Expected and measured behaviour using Half Scan Addressing and Edge Store Scanning: (a) for an SPA with N=32, P=1024, (b) L=32 and P=1024.

The *instruction* overhead factors which were found are not representative for arrays having hardware scanning possibilities. Such arrays have a much lower instruction overhead. The measured values for the *scanning* overhead agree with an investigation on these overhead factors for SPAs which is reported in the literature (Buurman and Duin 1988).

4.7.5 Comparison of SPA, LPA and PL for LNOs

The comparison between the SPA, LPA and PL architecture for local neighbourhood operations (LNOs) will be based first on the number of clock cycles n to perform an algorithm of length l basic instructions using P processors for an N*N image. Second, the efficiency measure E as discussed in Chapter 3, Equation (3.4) will be used, i.e.:

$$E = \frac{lN^2}{nP}$$

This measure gives the number of pixel operations per second per PE. In the comparison between the SPA, LPA and PL for local neighbourhood operations, the following assumptions are made:

- All processing elements are of equal complexity and functionality
- The clock speed is the same for all architectures
- The minimum overhead factors as found in existing machines will be used for all three architecture groups
- No extra time for downloading programs is taken into account, as this is assumed to take place concurrently with processing
- The data I/O mechanisms which are generally found for the architecture groups will be taken. This means: raster scan data I/O for the PL and the LPA, and column parallel data I/O for the SPA.

Referring to the last assumption, it is interesting in some cases, to study the effect of different data I/O mechanisms for the SPA. Some SPAs still use the raster scan data I/O methods, and it is interesting to see the improvements that column parallel data I/O offer for those SPAs. Newer SPAs on the other hand, claim much improvement due to their image parallel data I/O. It is interesting to see when and if this claim holds.

The minimum overhead factors are given in Table 4.1. The scanning overhead for a processor array is about 4 for half scan addressing and about 2.5 for edge store scanning. More advanced scanning techniques may in principle even lead to a scanning overhead of 1 (i.e. no overhead). For the comparison, hardware implementations of the scanning techniques should be regarded, as these implementations have a very low instruction overhead, and are thus more realistic (see Section 4.1). Neither HSA nor ESS have been implemented for SPAs in hardware. In the comparison between the architectures for local neighbourhood operations, the HSA overhead factor of 4 will be presumed, and where applicable it will be shown what the influence of the optimal overhead factor (i.e. one) would be. This choice is made, because existing SPAs can quite easily be modified for HSA (Buurman and Duin 1988). Adaptation for ESS requires hardware implementation of edge stores, which is not at all straight forward. The AIS-5000 linear processor array scans without overhead, as is shown in Chapter 2. The overhead due to reprogramming a pipeline is negligible compared to the frame recirculation time, for both the Cyto-HSS and the CLPE. The instruction overhead is taken to be 1 everywhere, as it is in principle possible to equip PEs with a lookup table so that any instruction can be executed in one clock-cycle. This does, however, imply facilities to load the lookup table in one clock-cycle or to allow switching between several lookup tables.

Table 4.1 Overhead factors

	SPA	LPA	PL
Scanning	4 (HSA)	1 (AIS-5000)	1 (Cyto / CLPE)
Instruction	1 (ideal)	1 (ideal)	1 (ideal)

with: ideal = theoretical minimum

With the overhead factors from Table 4.1, the formulas for the processing times of the SPA, LPA and PL can be filled in. This leads to the following set of equations:

$$n_{SPA} = \left\lceil \frac{N}{\sqrt{P}} \right\rceil^2 \cdot O_{SPAscan} \cdot l + N \quad , \text{with } O_{SPAscan} = \begin{cases} N \le \sqrt{P} \rightarrow 1 \\ N > \sqrt{P} \rightarrow 4 \end{cases}$$

$$n_{LPA} = \left\lceil \frac{N}{P} \right\rceil \cdot N \cdot l + N \tag{4.16}$$

$$n_{PL} = l \cdot (N+2) + \left\lceil \frac{l}{P} \right\rceil \cdot N^2$$

Some basic conclusions can be drawn from Equation (4.16). First, the SPA will be slower than the LPA when $N > \sqrt{P}$, because of the scanning overhead of 4. When comparing the LPA and the PL, it can be seen from Equation (4.16) that the PL is *always* slower than the LPA due to its pipeline latency. Further analytical comparisons between the performance of the three architectures is very difficult because of the upwards rounding effects which must be taken into account.

A comparison can be made between the three architecture groups as to their performance for the calculation of an algorithm of a fixed number of l steps, a fixed number of P PEs, but varying image size N. The speed for a 32-step and a 256 step algorithm as a function of N is shown in Figure 4.8a and b.

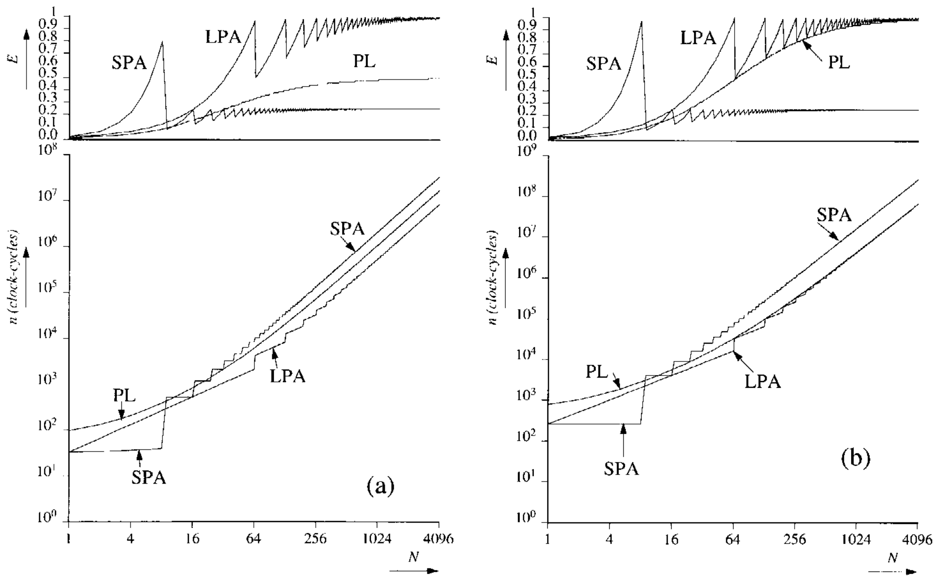


Figure 4.8 Speed in number of clock cycles n and efficiency E for an algorithm with varying image size N, but fixed number of processors P = 64: (a) l=32 steps and (b) l=256 steps.

Note that the efficiency of the SPA drastically decreases for images larger than the array size (size>8*8 for 64 processors) anymore. Notice the points where the performance graphs cross one another. At those points, one architecture wins over another architecture. Also note the behaviour of the LPA for increasing image sizes. Steps are taken at those points where the image does not fit into the LPA anymore, i.e. at multiples of 64 for N.

The efficiency for the three architecture groups is drawn above the cycle time graphs in Figure 4.8a and b. This measure indicates the number of pixel operations per PE per clock cycle. Obviously, the most efficient number will be one pixel operation per PE per clock cycle. This is only reached by the LPA for large image sizes. The PL can reach it, accord-

ing to Figure 4.8b, with large image sizes and large algorithms. The SPA never reaches an efficiency larger than 0.25 pixel operations per PE per clock cycle. This is due to the scanning overhead of the SPA, which limits the efficiency when the images no longer fit in the array.

One important point to notice is, that the transitions which indicate a change in the best performance for an architecture, are the same for the speed diagrams as for the efficiency diagrams.

The same step behaviour is seen for the PL if the algorithm length is increased while the image size and the number of processors is held constant. This is illustrated in Figure 4.9a. The PL has to do frame recirculation after the algorithm length l is more than the number of processors P. Notice that the efficiency of the LPA will reach a value of one if l is large enough, but the efficiency of the pipeline does not increase further than $1/(1+P/N)$ for large l.

The performance for a fixed image size and algorithm length, but with increasing number of processors is shown in Figure 4.9b. The number of clock cycles decreases quadratically (a straight line in the double log plot) if the number of PEs P is increased for the SPA. However, the performance of the LPA does not increase after the number of processors becomes more than the size of the image N. The PLs performance stays equal after the complete program of l=32 fits in the 32 PEs.

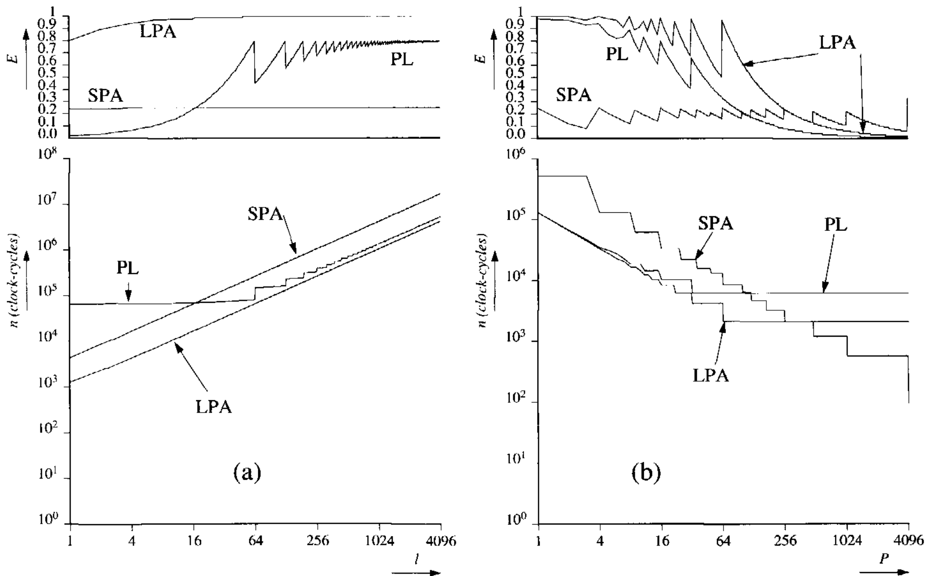


Figure 4.9 Speed in number of clock cycles n and efficiency: (a) image size N=256 and number of processors P=64 and (b) image size N=64 and algorithm length l=32.

Again an important point to notice is the fact, that the transitions from the area where one architecture performs best to an area where another architecture 'wins' are the same in the efficiency as in the speed diagrams. This can be explained as follows: the efficiency for the machines at any point in the (N,P,l) space is calculated by dividing the same factor $l \cdot N^2/P$ by n_{SPA}, n_{LPA} or n_{PL} at that point (see Equation (4.16) and Equation (3.4)). Because of the monotonic relation between n and $l \cdot N^2/P$, the ordering of the architectures will not change.

Investigating the performance in one-dimensional plots such as described above for all combinations of algorithm length, number of processors, and image size is not necessary. Instead of looking at all possible plots where the processing time is a function of N or l or P, *algorithmic phase-diagrams* can be made (Hockney 1987). The areas where a particular architecture is best are labelled with the name of that area. Also, areas are indicated with Greek letters where the difference between the ^2log of the difference in number of clock cycles n is less than 5%. In this way regions can be seen, where the performance difference between two of the three architecture types is negligible. We distinguish 'hard' and 'soft' phase transitions as follows. A 'hard' phase transition marks areas where one architecture is better than another, whereas a 'soft' phase transition marks areas where the ^2log of the number of clock cycles is less than 5%. Although these 'fuzzy' areas in the phase diagrams depend on one performance measure (the speed measured in number of clock cycles) the 'hard' phase transitions are independent of the performance measure (as explained above). Because of this, only phase diagrams for the processing time and not for the efficiency are shown.

The *first* case for which phase diagrams are drawn is shown in Figure 4.10. For a constant amount of processors, the influence of l and N is shown. To give an idea of the influence of the type of data I/O for the SPA, three separate diagrams are drawn.

The phase diagrams show that an SPA is better for images which fit in it, so that it won't have to scan. For larger image sizes, the LPA is always better. Only when the algorithm size become larger than the number of PEs the LPA and PL have processing times for which the ^2log's differ less than 5% (indicated by the fuzzy α area).

In Figure 4.10d, e and f, the comparison is done between the LPA, the PL and an SPA which has a scanning overhead of one. For as far as we know, there is no SPA which can reach such performance at this moment. However, if a scanning technique is developed in hardware or software which does not lead to overhead, then from Figure 4.10d, e and f it is clear that an SPA with column parallel data I/O is superior to both the LPA as well as the PL. Only for raster scan data I/O in the SPA (which is not too unrealistic with presently found bandwidths between sensors and architectures) is the LPA architecture better if the image size N is greater than the algorithm length l.

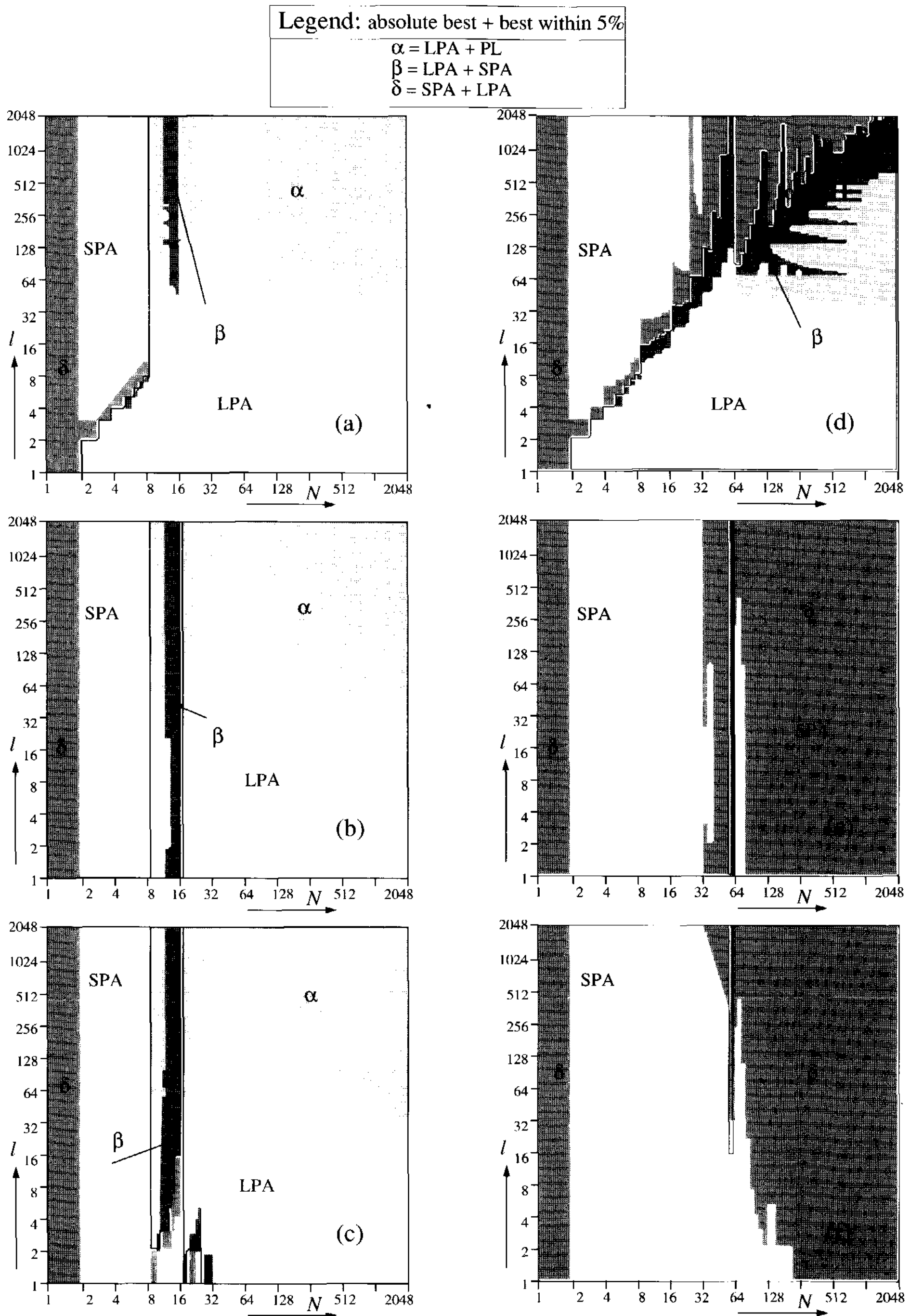
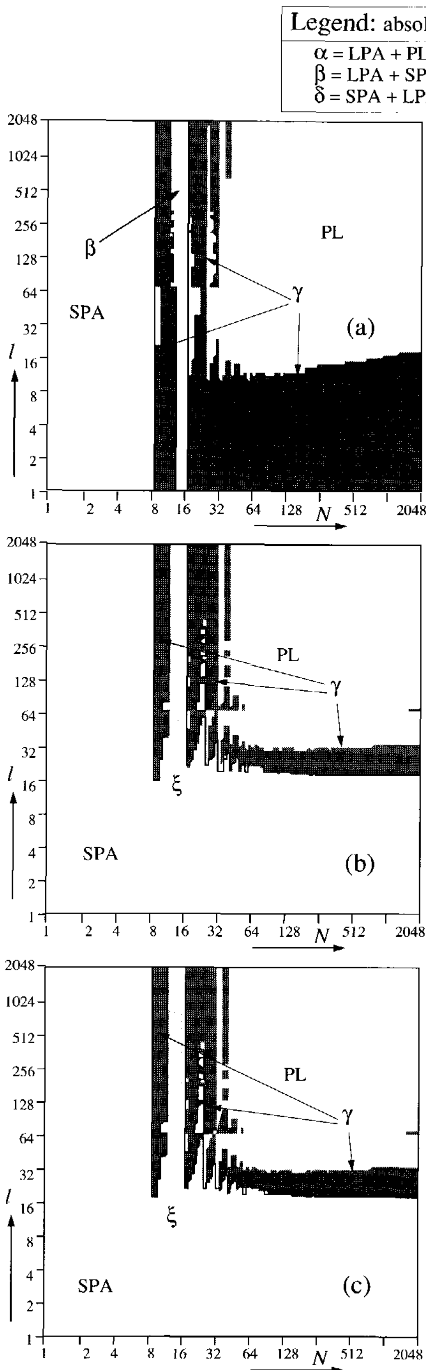


Figure 4.10 Comparison of an SPA, LPA and PL for 64 PEs, with different data I/O types for the SPA: (a) raster scan, (b) column parallel, (c) image parallel with scanning overhead 4.0, and (d), (e) and (f) for an SPA with scanning overhead equal to 1.0



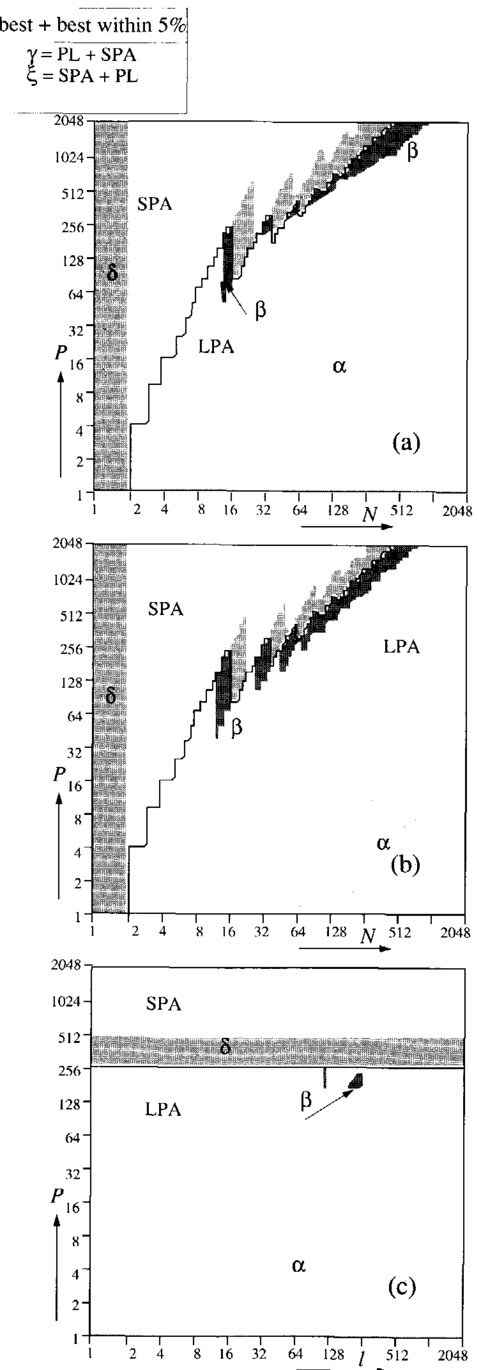


Figure 4.11 Comparison of an SPA and PL for 64 PEs, with different data I/O types for the SPA: (a) raster scan; (b) column parallel; (c) image parallel.

*Figure 4.12 Comparison of an SPA, LPA and a PL: (a) algorithm length of l=512 steps, (b) algorithm length of l=32 steps, (c) an image size of 64*64 PEs.*

From the comparison with a fixed number of 64 PEs, the borderline between the SPA and PL can also be drawn. This is shown in Figure 4.11a-c. This time there is no notable difference at all between the column parallel and image parallel data I/O for the SPA. In both of these I/O cases, the PL is better if the image is larger than 32*32 *and* the algorithm is larger than 32 steps, with a small fuzzy β area. So even if the SPA has to scan over the image ($N > 8$) it is still better for small local neighbourhood algorithms. If the SPA uses raster scan data I/O, the situation becomes worse. If it is smaller than the image (i.e. $N^2>P$) then the SPA is outperformed by the PL for almost all algorithm lengths. Small fuzzy areas β and γ indicate the area for which the processing times differ very little.

The *second* and *third* cases which will be considered now, are that of a varying number of processors P, and a constant algorithm length or a constant image size. Algorithms of length l = 512, and length l = 32 are considered, as well as a fixed image size of N=64. The constant algorithm length plots are not much influenced by the data I/O method for the SPA, as can be seen from the phase diagrams in Figure 4.12a and b.

From this second slice through the three dimensional phase space of the LNO comparison between the SPA, LPA, and PL, some new things can be learned. The phase transition between the SPA and LPA is the same for both lengths, although the fuzzy α area, where the PL is within 5% of the LPA (with respect to ${}^2log\ n$), changes. The phase transition reveals that the SPA performs better for large images, if more PEs are available. This can be explained on one hand by the fact that the SPA doesn't have to scan if $N^2 < P$. On the other hand it is clear, that if the image size N is smaller than the number of processors P in an LPA, then there are a number of processors in the LPA which do not take part in processing the image. That this process is virtually insensitive to algorithm length is revealed in the third plot, Figure 4.12c, for an image size of 64*64.

4.8 Object operations

As was shown in Chapter 3, object operations use all the pixels of an object in an input image for the calculation of a pixel value in the output image. Depending on the nature of the object operation, it can be calculated recursively or not. The comparison for those operations which can be calculated recursively starts in the next section, and is further studied in the rest of this thesis, starting from Chapter 5.

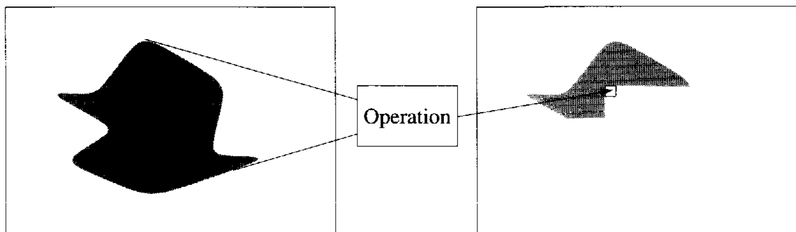


Figure 4.13 An object operation requires all object pixels to be fetched for the calculation of one destination pixel value.

It is not yet known whether there are (many) interesting object operations which can not be calculated using RNOs. However, such object operations, require that the architecture allows fetching the pixels of the object one by one, and performing calculations on them, as shown in Figure 4.13. The distances over which pixels have to be fetched are data

dependent. It is therefore assumed, that the performance of an architecture for calculating such object operations, increases with the possibilities it offers to transport pixels over data dependent distances larger than nearest neighbour, i.e. the neighbourhood connectivity.

The SPA, LPA and PL, all can fetch data over larger distances by using the local neighbourhood to do a sequence of small shifts. For the PL this is the only method. SPAs which offer local connectivity control, can form paths which go past the local connectivity. Data can then be passed along these paths. An SPA which is smaller than the image size, and uses crinkle-wise mapping for the image storage can also reach more neighbours in a faster way (see Section 2.3.1). An LPA has a neighbourhood connectivity which is of order N, because it allows pixelvalues to be fetched from anywhere in an image column within one clock cycle. Some LPAs are equipped with local addressing. This feature can be used to fetch pixels in a data dependent way over longer distances.

From this reasoning it must be concluded, that the LPAs offer the best possibilities for handling object operations which cannot be calculated using RNOs. SPAs with crinkle-wise data storage also offer good possibilities.

4.9 Recursive neighbourhood operations

This class incorporates several well known *object* and *global* operations, like the distance transform, object selection, and median root. No comparison has been found in the literature *specifically* for RNOs. However, from an image processing point-of-view it is more natural to use neighbourhood operations as a basis for the calculation of object and global operations. For image data it is assumed that there is a spatial dependency within the two dimensional data set (the image). Image processing architectures are therefore constructed with facilities offering easy accessibility of points in the neighbourhood. If such architectures can thereby be used for the calculation of object and global operations while still making maximum use of their neighbourhood facilities, then that should be explored in depth. With this in mind, the class of object operations and other RNOs will be specified in Chapter 5, and architectures will be compared on their performance for this specific class in Chapter 6 and 7.

4.10 Global operations

Traditionally, architectures for low-level image processing are built for point or local neighbourhood operations. This is due to the locally correlated nature of images themselves. Global operations can in principle be programmed on top of the local neighbourhood operations, but this is usually very inefficient. This was shown by Rosenfeld for the Hough transform on SPAs (Rosenfeld et al. 1988). The Hough transform *'requires global shifting and summarizing of data from all parts of the array'*, which cannot be done efficiently for SPAs. If an SPA is used which has the same size as the image (i.e. $N*N$), then $4*N*N$ local shifts are enough to allow every image point to 'visit' every PE. This is done by shifting the source image N steps to the right, and then the same source N steps to the left. After shifting one step downwards, the process is repeated, so that it is clear that $(2N) * (2N)$ shifts are required. If an SPA with torus topology is used, then $N*N$ shifts will suffice.

Not all global operations can be calculated by data offered in the raster scan order as done in the shifting scheme. In general, a global operation may require one pixel from one

position, and another pixel from a totally different position. An example of this is the discrete Fourier transform.

Recent new architectures are improving on their capabilities for transporting data across the array, so it is expected that global operations will improve on these novel SPAs. A study has been done on the hardware methods which are used to improve the connectivity over larger distances in an SPA (Fountain 1988c). Such methods are:

- *Row/Column buses.* Each PE is connected to a row and a column bus, which allows data to be transported in a horizontal or vertical direction over the array. This facility is used in the DAP, GRID and MMB (see Chapter 2).
- *Global propagation.* A recursive neighbourhood parallelism of eight (as in the CLIP4) allows a pixel to be broadcast from one position to any other position in very little time (depending on how fast a pixel can propagate through a PE).
- *N-cube connections.* The connection machine has mesh connectivity as well as hypercube connectivity between the PEs. The mesh connectivity is used for LNOs, and the hypercube connections can be used to pass data over longer distances.
- *Local connectivity autonomy.* Every PE can be equipped with a network which allows any PE input to be connected to any PE output. PEs can thus be bypassed, so that no time within a PE is lost. This is therefore a very interesting method. It is at this moment used in the coterie network on top of the CAAPP, and has been suggested for the PTA (see Chapter 2).

Although SPAs can be extended with global connectivity improvements, the resulting global operations can not be calculated with algorithms which are of the same order as array size, i.e. of order N^2.

Pipelines can only do global operations as a large sequence of local neighbourhood operations, and are therefore considered as unsuitable for global operations.

LPAs are much more suitable for global operations, as the transportation of data in one dimension occurs without extra cost. A 2-D FFT on an image can be done as a sequence of two 1-D FFTs in the appropriate direction (the vertical direction for the LPA), combined with image transpositions. The one dimensional FFTs can be efficiently implemented in an SIMD way on an LPA. The transpositions can efficiently be implemented if PEs are equipped with local addressing autonomy (Lindskog 1988). Therefore, grey value or floating point LPAs can, in this way, efficiently implement the FFT (Schmitt and Wilson 1988; Lindskog 1988). Splittable global operations in general are best performed by the LPA with enough local autonomy.

Some global operations, such as the median root filter, can be performed using RNOs. In that case PLs or SPAs can also execute them. The performance of the architectures for such RNO-implementable global operations will be derived in the rest of this thesis.

4.11 Geometric Operations

Geometric operations (e.g. warping, rotation, translation, scaling) belong to the low-level image processing operation class, because they map a source image into a destination image. It may be necessary to do geometric corrections to an image to compensate for the imaging technique. Also, time or space sequences of images may have to be aligned (geometrically corrected) before they can be combined in further processing.

The essence of geometric operations is, that they are anisotropic (except pure translation); i.e. the operation depends on the position in the image. This is opposed to the nature of low level image processing architectures, which are SIMD, and hence isotropically oriented.

Geometric operations have not yet been compared for low-level image processing architectures. Algorithms have previously been developed to perform some geometric corrections on SPAs (Clarke and Ip 1982). These algorithms are essentially of order N processing steps, whereas a full sized SPA has order $N*N$ PEs, so that an order of N^3 work has to be done (i.e. the number of processing steps for this algorithm executed with only one PE).

Rotating an image within an SPA can be done in two ways. In the first method every PE is loaded with the amount of horizontal and vertical shifts that its contents has to make to arrive at its destination (Strong 1982). For the calculation of these numbers, it is necessary to load every PE with the coordinates of the image point it holds. Some SPAs, like the MPP, have special facilities for this. The array is then shifted in horizontal and vertical directions, and every PE only joins the shifts, if its contents gets closer to its destination. Note that the PEs require local activity autonomy for this method. This algorithm is therefore of order N. The second method breaks the rotation up into horizontal and vertical skews. Rows and columns are then selected and skewed over the required distances. This algorithm is also of order N. The efficiency with which an SPA can do the skewing depends on the array size. If P processors are available in an SPA, then the amount of work that has to be done per scan is $P \cdot \sqrt{P}$. Because there are N^2/P scans in an image, the total amount of work is $N^2 \cdot \sqrt{P}$. This means, that the amount of work increases with the number of processors used, i.e. the efficiency decreases.

LPAs are themselves of a size which is order N. They therefore have less problems with the geometric operations. Rotation for example is done using skews in vertical direction through indirect data addressing. The horizontal skew is implemented by a transposition, a vertical skew and a transposition. Lindskog showed that with local addressing autonomy, a linear processor array can transpose an $N*N$ image in order $N*N(1+1/P)$ clock cycles (Lindskog 1988). The skewing algorithm on an LPA is of order N, which is the same order as the architecture itself. The amount of work to be done is therefore less than with an SPA. Translation and scaling can also be built up by a sequence of skew operations with transpositions in between.

A pipeline can also do geometric operations by building them up from skews and transpositions. They do not have the possibilities of an LPA, as they do not contain the image in a local memory. The problem with the pipeline is also, that it cannot be extended in the same way as the SPA; in an SPA facilities can be provided to move the data over longer distances. The PL works concurrently in the instruction sequence, so that data transfer over longer distances in one single step is not possible.

From the examples presented it is clear, that the LPA which is equipped with local addressing autonomy offers the best possibilities to handle geometric operations in an efficient way.

4.12 Statistical Scalar Operations

Low level image *processing* operations sometimes require parameters which have to be calculated using low level image *analysis*. Three such operations are generally used:

- *Bit count.* The number of bits in a binary image is counted.
- *Array zero test.* A test is done to see if all pixels in the array are zero or not.
- *Difference between two images.* For the efficient calculation of RNOs, there should be some fast way to check the difference between two calculated images. This will become clearer in Chapter 5, but it is noted here for reasons of completeness.

A survey of bit counting techniques for SPAs has been done in the past (Reeves 1980b). Reeves considers a 128*128 SPA, and suggests the construction of a network of counters to sum the values of the PEs. Three different bit-counting schemes are compared by him:

- *Full counter.* A counter with 16384 input bits and 15 output bits uses 8 clock cycles to do a bit count. However, the network consists of 627 chips which count 31 bits each.
- *Every-8 counter.* One out of eight PEs is connected to a 2048 bit input counter, so that eight steps are required to obtain the full count. The total counting operation takes 15 clock cycles, and uses 85 chips with 31 bit input chips.
- *Edge counter.* The 128 PEs of one edge are connected to a 128 input bit counting network. The total image content is counted in 128 steps. This scheme uses only 6 chips with 31 bit input, but takes in total 132 clock-cycles.

All three bit counting circuits which are proposed by Reeves work in a pipelined mode, so that the calculation of a *new* bit count can be started after one clock cycle.

Bit counting in a PL does not require special hardware. It can be done 'on the fly', while the image pixels ripple through a PE. The pixel bit count of an image in one stage of the pipeline is not immediately available to the next pipeline stage. This makes it almost impossible for a pipeline to do conditional processing without frame recirculation.

The array zero test in an SPA is generally done by a logical *or* of all PE output values. An LPA can use the same technique, but should logically *or* intermediate results line for line.

4.13 Statistical Vector Operations

The thresholding of an image is a low level image processing operation, which sometimes requires the calculation of a histogram to determine a good threshold value. Determining a grey value histogram is an example of an SVO (i.e. a statistical vector operation) used in almost every image processing system. A grey value histogram is a vector of length 2^b, which contains at position i the number of pixels with value i (each pixel has b bits).

In a bit serial SPA, the grey-value histogram can be calculated using the bit-counting hardware (if available). First, a grey value point operation binarises those pixels which have the grey value under current observation. Second, the number of bits in the binary result image of the previous step is determined using the bit-count hardware. These values are sent to the host of the SPA, where decisions will be taken on the basis of the histogram. If an SPA uses a bit counter which is internally pipelined (see previous paragraph), then the process of extracting one grey value level image and counting the bits could also be pipelined. This will reduce the amount of time needed to make a full histogram.

A bit serial LPA will find the histogram in the same way as the SPA. It will do separate bit counts on each grey value level of the input image. A grey value LPA (like the

SYMPATI-2 and the SLAP, see Chapter 2) will determine the histogram differently. First a histogram from the pixels stored in one PE (a column, row or diagonal of the image) is made in 2^b steps. Second, the partial histograms are combined in $2^b \cdot \log(P)$ steps, where P is the number of processors.

At present there are no strictly bit serial PLs (in the sense that the PL consists of only bit serial PEs) used for grey value processing, although a proposal has been done to use CLPE-VLSI for this purpose (Jonker et al. 1989). A *single* stream bit serial PL would have to determine one grey value level of the input image, and then count the bits in the resulting binary image. It will have to do frame recirculation for every level, so that this will take on the order of $2^b \cdot N^2$ steps. A bit serial PL with b bitstreams and equipped with a Point Operations Ram will be able to determine the histogram bins of b grey value levels in every pipeline stage. So if the PL is at least $2^b/b$ stages long, then the histogram can be calculated in one pipeline pass.

A grey value PL which is extended with local addressing autonomy can make a histogram 'on the fly', because all the pixels of the image flow through the PEs of the PL. One PE can then be used for the calculation of the histogram. After the complete image is passed by, the histogram can then be transferred to the host computer where decisions can be made on the basis of the histogram.

4.14 Conclusion

As a result of the points mentioned in this chapter, a theoretical comparison between the three architecture groups can be done. Table 4.2 indicates the performances of the SPA, LPA and PL architectures for the qualitative and quantitative points on which the comparison is based. This table is but a rough indication of the points which are made in the previous sections. The comparison is based on the following assumptions:

- *Number of PEs*. It is assumed that the number of PEs used in the SPA, LPA or PL is the same.
- *Power of each PE*. The power of the individual PEs used in the SPA, LPA and PL is the same. This assumption is in close connection with the previous one. If more PEs for the SPA are allowed, then the power of each PE will be designed to be less.

Table 4.2 Theoretical comparison

	SPA	LPA	PL
Data input speed	+/-	++	+
Image size flexibility	+/-	+	++
Pixel size flexibility	++	++	-
Neighbourhood size/shape	+/-	+	-
Programmability	+	+	-
Point operations	++	++	++
Local Neighbourhood Op.ns	+	+	+
Object operations	+/-	+	-/+
Recursive Nbhood Op.ns	?	?	?
Global operations	--	+/-	--
Geometric Operations	+/- (1)	+ (2)	-
Statistical Scalar Op.ns	+/-	+	++
Statistical Vector Op.ns	?	?	?

(1) Warping can only be done in some SPAs, which allow manipulation with their own address and have local activity autonomy.

(2) If equipped with local addressing autonomy

Meaning:

++	Very good
+	Good
+/-	In between
-	Bad
--	Very bad
?	Not yet known

The actually encountered data input speed is 'best' (i.e. fastest) for the LPA that uses row parallel data input and quite good for the PL due to its overlapping processing and I/O. Some of the SPAs today use column parallel data I/O, and some use raster scan data I/O. Hence the in-between score for SPAs.

Note that image size flexibility from the pipeline is traded for low programmability. On the other hand, the SPA and LPA trade enhanced programmability for low image size flexibility. This is in agreement with the difference in operator and spatial parallelism offered by these architectures. The image size flexibility of the SPA may be less if it does not have facilities to handle images larger than its array size.

Bit serial SPAs and LPAs offer the largest flexibility in pixel size. A bit serial PL needs to be equipped with a lot of special hardware to offer any flexibility in pixel depth. Also, processing multiple bit images requires the bit serial pipeline to recirculate through a frame buffer. When all three architecture types are equipped with the same grey value ALU, then their pixel size flexibility is the same.

Concerning the neighbourhood size and shape flexibility, the LPA scores best due to its large neighbourhood connectivity. An SPA which uses crinkle mapped image storage may also be reasonable. One pipeline which has actually implemented such feature is known from the literature (Duin et al. 1986).

Because of the fact that similar PEs are assumed in the comparison of Table 4.2, there is no difference in performance for the point and local neighbourhood operations. However, for object, global and geometric operations, data may have to be transported over longer distances at the highest possible speed - possibly in an anisotropic way. In such cases, the LPA performs best, followed by the SPA and PL. This is due to the large neighbourhood connectivity and the local addressing autonomy possibility of the LPA. At this point it may be argued, that the SPA can perform as good as the LPA when the images are stored crinkle-wise in the local memory of the SPA (see Chapter 2). The SPA would then also have extended neighbourhood connectivity. However, the gain in neighbourhood connectivity for the SPA with crinkle-wise mapping is accompanied by a loss in neighbourhood parallelism. This is because the neighbours of one image point may have been stored in the same PE for this mapping technique. These can not be fetched in parallel. The LPA does not loose neighbourhood parallelism due to the fact that its neighbourhood connectivity is large. SPAs can be enhanced with local connection autonomy, so that their performance increases. PLs cannot be enhanced in any way for these type of operations. They have their concurrency in the instruction stream, not in the data of one image.

The advantages of trading off pixel bit parallelism against spatial parallelism for an LPA are not at all trivial. The question is: is an LPA/SPA with 8-bit grey value PEs eight times more powerful than an LPA/SPA with the same number of bit serial PEs? From the point operations we know, this is true for point and propagation operations in the pixel sense. However, the pixel global operations (like multiplications) are done faster than the factor of eight increase in pixel parallelism would suggest. At this point one should realize, that the gain in pixel bit parallelism is usually at the cost of neighbourhood parallelism (maybe even neighbourhood connectivity). This means, that the grey value LPA/SPA will be less efficient in doing normal LNOs like erosions and dilations. Although equipped with less hardware, the bit serial LPA/SPA will be much faster for such operations. For the combination of pixel global and spatial local operations (i.e. convolutions) the grey value LPA/SPA again wins more than the pixel bit increase would suggest.

An important advantage for the PL and (in some cases) the LPA over the SPA can be noticed from the observation that the instructions loaded in the PEs of the first two architectures remain in the PEs for a longer time than is the case with the SPA. The longer the time that an instruction remains in a PE, the longer the time which can be used to load it, i.e. the more powerful (with respect to the instruction overhead) the PE which can be used. An instruction remains for $N*N$ clock cycles in the PE of a PL, it remains for $N \cdot \lceil N/P \rceil$ clock cycles in the PE of an LPA, and for $\lceil N/(\sqrt{P}) \rceil^2$ clock cycles in the PE of an SPA (when using hardware scanning). The LPA is only better than the SPA, if the SPA has more PEs than the LPA (this is the case for most existing LPAs and SPAs).

From the overview of the comparison which is done in this chapter it is clear, that the LPA performs better than or as well as the PL or SPA on all available points. However, not all points for which the architecture groups are to be compared are available. Specifically, the performance of the architecture groups for object and global operations which can be built up by recursive neighbourhood operations is not known. More comparisons between the architecture groups are also needed concerning their performance for global operations and object operations which can not be done using RNOs.

The next chapters will introduce the RNOs from a theoretical point of view, and compare the performance of the different architecture groups for their calculation.

Chapter

5 Non-linear recursive neighbourhood operations

In the previous chapter, a start was made on the comparison between some architecture groups and their performance for several operation groups. Of the operations which have not yet been compared, the recursive neighbourhood operations[1] appear to be of interest. These operations can be used as a way to calculate some well known object operations and global operations. Object and global operations need many pixels of the input image for the calculation of one output pixel, whereas RNOs (often repeatedly) need a local neighbourhood of output as well as input pixels to calculate one new output pixel. The spatial complexity of object and global operations is traded off against performing local operations repetitively. The use of local neighbourhoods for calculating object and global operations is attractive for two reasons. First, the architecture groups which we are investigating (i.e. the SPA, LPA and PL) perform better for operations with local neighbourhoods. We assume, that these architectures will perform better for RNOs then for equivalent OOs or GlOs. Experiments to gain insight into the performance of RNOs on the SPA, LPA and PL are presented in Chapter 8. Second, calculations in a local neighbourhood are appropriate due to the locally correlated nature of image data. This chapter introduces the theoretical background of RNOs. A restriction is made to *non-linear* RNOs for several reasons. First, a lot is already known about linear RNOs (Dudgeon and Mersereau 1984), whereas research on non-linear RNOs has lagged behind. Second, non-linear RNOs are of interest, as a lot of the object and some of the global operations in low-level image processing can be described by them.

As a point of reference to the reader with a background in linear RNOs, one inverse convolution type of RNO will be used as an example in this chapter.

Some frequently used image processing operations belong to the class of object operations: distance transform, medial axis transform, smallest enclosing regular polygon, convex hull, object selection and blob to point reduction. As an illustration of what can be done with object operations, some of them are shown in Figure 5.1.

The effect of the *distance transform* is shown in Figure 5.1b. A binary image containing objects is transformed into a grey value image, where each pixel value represents the minimum distance (according to a certain metric) from that point to the edge of the containing object (Rosenfeld and Pfalz 1968). There are many applications of the distance transform. Thresholding of a distance transform at a level L gives a binary image representing a version of the original image which is eroded over a distance of L. There are also methods in which a distance transform is used to construct a skeleton (Montenari 1968). This is an extension of the medial axis transform. The *medial axis transform* itself (see Figure 5.1c) is used to construct a medial axis[2] image out of a binary image containing objects (Blum 1967).

1. The term 'recursive neighbourhood operation' has been defined by Haralick (Haralick 1981).
2. The medial axis of an object may be unconnected, whereas the skeleton is always connected.

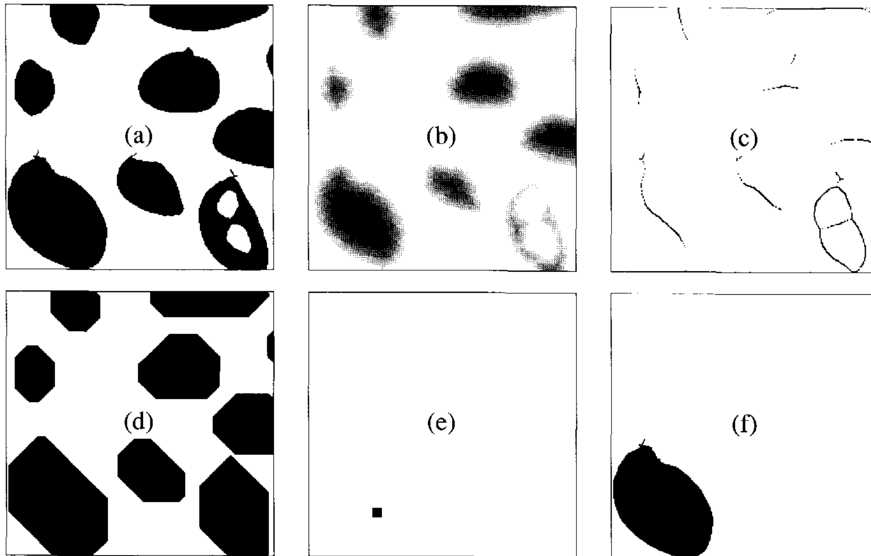


Figure 5.1 Object operations: (a) binary original, (b) distance transform, (c) medial axis transform, (d) smallest enclosing regular polygon (e) propagation seed and (f) object selection by propagating the seed into the original.

The *Smallest Enclosing Regular Polygon* produces the smallest regular polygon, where the number of corners is trivially related to the specified neighbourhood, which fits around the objects in the image (Figure 5.1d). This operation will be further treated in Section 5.2.3. The *object selection* fills the objects in an image from the seeds (Figure 5.1e) in another image. The process of filling an object is shown in Figure 5.1e-f, where Figure 5.1a is used as a mask.

The theoretical discussion of RNOs starts in Section 5.1 by giving a mathematical description of the RNOs in the light of other operation groups. The relation between RNOs and the local, object and global operations is illustrated in Section 5.2. Many RNOs can not be described completely without also specifying an order in which the image points are calculated. Such an order is described with an updating method. Known updating methods are discussed in Section 5.3. Then in Section 5.4 we begin the derivation of a theory which makes it possible to predict whether an RNO has none, one, or more-than-one fixed point[1] images. The RNOs are divided in classes and discussed in Section 5.5. Finally, some conclusions for RNO theory are summarized in Section 5.6.

5.1 Mathematical formulation for low-level image processing operations

In this section the low-level image processing operations from the operation groups derived in Chapter 4 are formulated in a mathematical way.

Definitions for the description of these operation groups are given in Section 5.1.1. To clarify the meaning of RNOs, Section 5.1.2 discusses the 'recursiveness' of them, and Section 5.1.3 relates RNOs to an eigenfunction problem. Section 5.1.4 discusses some possibil-

1. A *fixed point* or *root* image is an image which fulfils the RNO at every point at the same time. This is discussed further in Section 5.4.

itics to treat the images edge for the calculation of RNOs. Even more constraints, such as initial conditions and updating method, can be specified. When and how this can be done is discussed in Section 5.1.5.

5.1.1 Definitions

The definition of object and recursive neighbourhood operations is shown in Table 5.1, together with the definition of other low-level image processing operations. The consistency in definitions is apparent from this table. The operation categories which appear in Table 5.1 are also used in the low-level image processing operation taxonomy which is defined in Chapter 3.

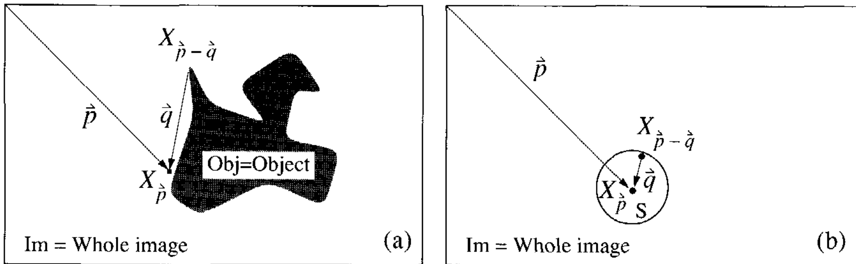


Figure 5.2 Use of symbols for the description of low-level image processing operations; (a) object operation and (b) local neighbourhood operations

The symbols used in the operation-definitions are explained with Figure 5.2. A point at position $\vec{p}$ in an image may have a set S of neighbourhood points connected to it. The value of a source image point at position $\vec{p}$ is denoted $X_{\vec{p}}$. The value of a point with relative position $\vec{q}$ towards $\vec{p}$ is denoted as $X_{\vec{p}-\vec{q}}$. Neighbourhoods can extend over the set S, to the object Obj until the whole image, denoted as Im.

To clarify the notation that is used, each of the formula definitions in Table 5.1 is translated into a plain English form.

Table 5.1 Definitions of image processing operations

Operation	Short	Definition
Point	PO	$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = f(X_{\vec{p}})$
Local Neighbourhood	LNO	$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = f_{\vec{q} \in S}(X_{\vec{p}-\vec{q}}, \vec{q})$
Object	OO	$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = f_{\vec{q} \in \text{Obj}}(X_{\vec{p}-\vec{q}}, \vec{q})$
Recursive Neighbourhood	RNO	$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = f_{q \in S}(Y_{\vec{p}-\vec{q}}, X_{\vec{p}-\vec{q}}, \vec{q})$
Global	GlO	$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = f_{\vec{q} \in \text{Im}}(X_{\vec{p}-\vec{q}}, \vec{q}, \vec{p})$
Geometric	GeO	$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = f_{\vec{q} \in S_{\vec{p}}}(X_{g(\vec{p})-\vec{q}}, \vec{q}, \vec{p})$
Statistical	SO	$Y = f_{p \in \text{Im}}(X_{\vec{p}}, \vec{p})$

Im = whole image; S = Structuring Element; Obj = object pixels
$X_{\vec{p}}$ = source image point; $Y_{\vec{p}}$ = destination image point.

- *Point Operation.* Every point Y at position $\vec{p}$ in the destination image is a function f of the point X in the source image with the same position $\vec{p}$.

- *Local Neighbourhood Operation.* Every point Y at position $\vec{p}$ in the destination image is a function f of all points with position $\vec{p}-\vec{q}$ (where $\vec{q}$ belongs to the structuring element S), and of the position $\vec{q}$ within the structuring element.
- *Object Operation.* Every point Y at position $\vec{p}$ in the destination image is a function f of all points with position $\vec{p}-\vec{q}$ (where $\vec{q}$ belongs to the object), and of the position $\vec{q}$ within the structuring element.
- *Recursive Neighbourhood Operation.* Every point Y at position $\vec{p}$ in the destination image is a function f of all points Y and X with position $\vec{p}-\vec{q}$ (where $\vec{q}$ belongs to the structuring element S), and of the position[1] $\vec{q}$ within the structuring element.
- *Global Operation.* Every point Y at position $\vec{p}$ in the destination image is a function f of all points X with position $\vec{p}-\vec{q}$ (where $\vec{q}$ belongs to the complete image), and of the absolute and relative positions $\vec{p}$ and $\vec{q}$.
- *Geometric Operation.* Every point Y at position $\vec{p}$ in the destination image is a function f of all points with position $g(\vec{p})-\vec{q}$ (where $\vec{q}$ belongs to the structuring element $S_{\vec{p}}$, which may depend on the position $\vec{p}$ in the image, and g is a function on the source coordinate), of the position $\vec{p}$ in the image, and of the position $\vec{q}$ within the structuring element.
- *Statistical Operation.* The value Y is a function f of all points X with position $\vec{p}$ (where $\vec{p}$ belongs to the complete image).

The relation between POs, LNOs, OOs and GlOs is clearly visible from Table 5.1. The operations are equal except for the number of source pixels that work together in the calculation of one destination pixel. This increases from one for the POs, to all elements of S for the LNOs, to all elements of the object for OOs, and finally to all image pixels for the GlOs.

5.1.2 Spatial versus temporal recursion

To avoid misunderstandings in the meaning of RNOs, we will explain its relation to recursion as used in linear systems theory. The adjective 'recursive' in the term recursive neighbourhood operation applies to 'neighbourhood', just as 'local' from LNO applies to 'neighbourhood'. Therefore, RNOs are recursive in the spatial sense, and not necessarily in the temporal sense. Haralick notes that this spatial recursiveness may be through the image memory, as he defines the RNOs as "those operators which use the same image memory for their input and output" (Haralick 1981).

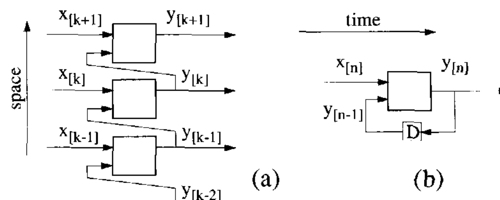


Figure 5.3 Recursive system: (a) spatial and (b) temporal.

The difference between temporal and spatial recursion is illustrated in Figure 5.3a and

1. The position of $\vec{q}=0$ in the Y image is called the *central pixel*, which does not necessarily belong to the neighbourhood point set S.

b, and will now be explained. Let us focus on a system where the output y at position k or time n is a function of the input x at position k or time n, and of the output at position k-1 or time n-1. This system can be described by:

$y_{[n]} = f(x_{[n]}, y_{[n-1]})$, for the temporal system, and

$y_{[k]} = f(x_{[k]}, y_{[k-1]})$, for the spatial system.

The temporal system needs a delay element D, so that the output signal $y_{[n]}$ can be calculated from the output signal at *time n-1*. In the spatial system, however, the output signal $y_{[k]}$ is calculated using the output signal at *position k-1*. A difference of great importance, which should be kept in mind when thinking about RNOs is the following: the order in which the output values for a temporal system are calculated is dictated by the sequential nature of time, whereas the output values of a spatial (recursive) system may be calculated in any desired spatial ordering.

Temporal and spatial recursion are essentially the same for a system where the output at time/position n/k depends on itself. Such a system is described as:

$y_{[n]} = f(x_{[n]}, y_{[n]})$, where n could be either position or time.

An example of a non-causal system with temporal recursion is the following:

$y_{[n]} = f(y_{[n-1]}, y_{[n+1]}, x_{[n]})$

The notion of causality is reserved for temporal systems, and can therefore not be used to describe spatial recursive systems of a form equivalent to the temporal recursive system:

$y_{[k]} = f(y_{[k-1]}, y_{[k+1]}, x_{[k]})$

For spatial recursive systems such as above, the notion of *cyclicity* is used. This will be discussed in Section 5.4.2.

5.1.3 Seeing an RNO as an eigenfunction problem

Concerning the definition of the RNO, a comparison can be made with the theory on eigenvalues and eigenvectors.

First, assume a linear system described by $A\vec{x} = \vec{y}$, where A is the system matrix, and $\vec{x}$ is the state vector. With regard to the eigenvalue problem, "a solution is sought of the form" $A\vec{x} = \lambda\vec{x}$, where $\vec{x}$ is called the eigenvector and λ the eigenvalue (e.g. Boyce and DiPrima 1965).

Second, assume a general eigenvalue/eigenfunction problem where "solutions of the form" $\psi(\vec{x}) = \vec{x}$ are being sought. Solutions of the general system $\psi(\ldots)$ give eigenvectors $\vec{x}$.

In the same way as the previous two examples, for RNOs we are "looking for a solution of the form" $Y = f(Y, X)$, where Y could be regarded as an eigenvector/eigenfunction of the RNO $f(\ldots)$. The fact that we are "looking for a solution of some form" implies:

- There may be no solution at all, exactly one, or more than one. Solutions for RNOs are called fixed points or roots, and the theory for them is treated in Section 5.4.
- How the solution is derived is not specified in the problem description, so that this may be chosen. This issue results in updating methods, discussed in Section 5.3.

5.1.4 Edge handling

As is the case of LNOs, the points at the image edge can gather their information across the edge in several different ways when RNOs are being calculated. Depending on the edge handling, an RNO may be stable or not in the fixed point sense (this stability is

further explained in Section 5.4.1). The following edge treatments are distinguished in this thesis:

- *Constant.* The edge is set to a constant value (Figure 5.4a).
- *Wrap.* Like in the torus architecture model from Chapter 3, the image points to the right edge connect to the image points on the left edge, and the points on the top edge connect to the points on the bottom edge. This effectively makes the image infinitely sized and periodic (Figure 5.4b).
- *Mirror.* The edge values are set at the same values as their 'corresponding' image points when mirroring in the line between the image and its edge (Figure 5.4c).

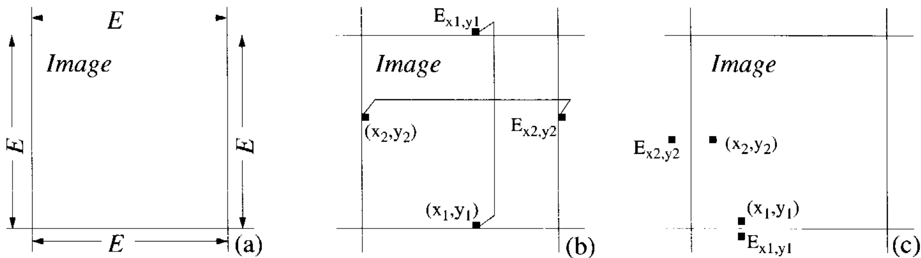


Figure 5.4 Edge treatments: (a) constant value of E, (b) wrapping, (c) mirroring.

The wrapping or mirroring of the image edge effectively makes the image data repeat itself in some way infinitely. In the case of at least one RNO (i.e. the median root, which will be treated later on in Chapter 5), it was found that image edge wrapping can result in an unstable RNO. However, for image processing purposes only stable RNOs in the fixed point sense are of interest.

Unless otherwise specified, the RNOs in this thesis will therefore use the *Constant* type of edge treatment.

5.1.5 Constraint specification

The description of an RNO may sometimes have to be accompanied by the specification of some other constraints:

- *Initial output image.* The initial state of the RNO output is specified with $Y^{(0)}_{\vec{p}} = \ldots$, and can be compared with the specification of an initial state vector for linear system theory. If nothing is specified, it is presumed that $Y^{(0)}_{\vec{p}} = 0$.
- *Updating method.* The result of some RNOs depends on the updating method used. Such RNOs should therefore always be specified together with the updating method for which they are meant. Updating methods are discussed in Section 5.3.

5.2 The relation between RNOs, local, object and global operations

It is not trivial from Table 5.1, to see how the RNOs relate to other operations. As LNOs and RNOs both work with local neighbourhoods, their relation will first be discussed. In the remaining two paragraphs it will be shown how some OOs and GlOs can be calculated using RNOs. This is done by showing the mathematical descriptions for the RNO and deriving its corresponding OO or GlO from them.

5.2.1 The relation between RNOs and LNOs

A local neighbourhood operation combines information from a source image into a destination image, whereas a recursive neighbourhood operation combines information from source and destination image into destination image iteratively until nothing changes.

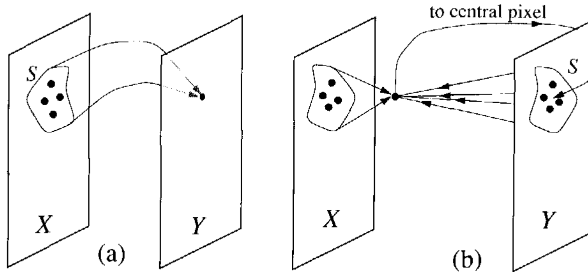


Figure 5.5 (a) Calculation of an LNO from a point set S of image X to a point in image Y, and (b) calculation of an RNO from a point in image X and a point set S in image Y to a point in image Y.

There are a number of correspondences, but also a number of essential differences between LNOs and RNOs. Both operations calculate a new value using a function which works on a local neighbourhood of points surrounding the source pixel value. The RNO, however, is applied with the view that the function is 'valid'[1] on all points in the image Y at the same time (see Figure 5.5). If this is not possible, the operation is still named an RNO, but it is not stable in the fixed point sense (see Section 5.4).

The aim of the RNO is to find a solution which is valid at all points at the same time. This has the following consequences:

- *Iteration.* The function f may be applied to a point in Y more than once, so that calculated pixel values are used in the calculation of a new pixel value.
- *Different solutions.* For some functions f and neighbourhoods S, the order in which points in Y are updated may influence the final result obtained (i.e. the RNO may have different *roots*[2]).
- *No solution.* For some functions f and neighbourhoods S, it may not be possible to find a final image Y for which f is valid on every pixel (i.e. the RNO has no root).

In one sense, an RNO may be seen as an iterated LNO (until a fixed point image is reached), and in another sense it may be seen as an LNO applied to the image points in a specific sequence (instead of being independent on that), *as well as* being iterated. The following example of an RNO will clarify this.

Suppose that there is an RNO which can be described as follows:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \max_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}})$$

with: $Y_{\vec{p}}^{(0)} = X_{\vec{p}}$; $(Y_{\vec{p}}, X_{\vec{p}}) \in \{0, 1\}$ and are one-dimensional, the edge is put to the constant value of zero, S = [illegible] = { (-1), (0), (1) }, so that the RNO is one-dimensional.

This description means, that starting from the original image X, the maximum is calcu-

1. An RNO is valid on a point $\vec{p}$, if indeed $Y_{\vec{p}}$ equals $f_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}, X_{\vec{p}-\vec{q}}, \vec{q})$.
2. For theory on *Roots* or *Fixed point images* see Section 5.4.

lated in a neighbourhood S of all image points, again and again, possibly until a solution is found. The order in which the calculation takes place is not specified. We will give an example of two possible orderings (i.e. updating methods). First, the values for all points are calculated simultaneously (called 'simultaneous updating'). Second the values for all points are calculated sequentially from left to right (called 'serial updating'). These 'updating methods' are further explained in Section 5.3.

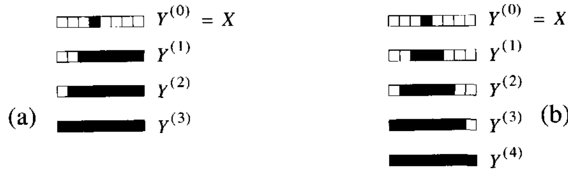


Figure 5.6 Demonstration of a one-dimensional RNO which calculates the maximum of three points: (a) sequential left to right updating, (b) simultaneous updating.

The RNO is demonstrated in Figure 5.6 for the two described updating methods. For both methods, the source 'image' X is generated in the first iteration $Y^{(0)}$. As can be seen in Figure 5.6, calculating the RNO in the prescribed orderings for the other iterations differs for the two methods. An important thing to notice is, that both methods are in principle repeated 'with the aim of finding an image that does not change anymore' (i.e. a fixed point or root). As explained in Section 5.1.3, the resultant 'image' Y could be called an eigenvector of the RNO.

5.2.2 The relation between RNOs and GlOs

The purpose of the examples given in this paragraph is to show the equivalence between some global operations and corresponding recursive neighbourhood operations.

5.2.2.1 Inverse convolution

In contrast to the other RNOs considered in this chapter, inverse convolution is linear. Showing how this linear RNO relates to a global operation serves the reader with a background in linear systems theory to better understand the theory for non-linear RNOs.

The inverse convolution filter is actually a filter with *infinite* size and can be programmed as an RNO. A convolution is an LNO (local neighbourhood operation), and can be written as:

$$(\forall \vec{p} \in \mathrm{Im})\, X_{\vec{p}} = \sum_{\vec{q} \in S^+} A_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} = \sum_{\vec{q} \in S} A_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} + A_{\vec{p}} \cdot c_{\vec{q}} \tag{5.1}$$

with: the neighbourhood S is the neighbourhood S^+ without the central pixel.

The coëfficients of the convolution mask are given by $c_{\vec{q}}$. Retrieving an image $Y = \hat{A}$ (i.e. an approximation of A) from a given convolved image X can in principle be done by rewriting the convolution Equation (5.1) into an RNO as given by Equation (5.2):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \varepsilon \cdot \left\{ X_{\vec{p}} - \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} \right\} \tag{5.2}$$

with: $\varepsilon = \dfrac{1}{c_0}$ $\quad c_0 \neq 0$, and $Y = \hat{A}$.

As will be shown, this RNO is equivalent to the following GlO (Global Operation):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = -1 \sum_{i=1}^{\infty} \left((-\varepsilon)^i \cdot \sum_{\vec{q} \in S^{i-1}} X_{\vec{p}-\vec{q}} \cdot c^i_{\vec{q}} \right) \tag{5.3}$$

with: $S^i = \{ \sum_{j=0}^{i} \vec{q}_j | (\vec{q}_j \in S) \}$, i.e. an i times expanded neighbourhood S.

By definition: $S^\infty = Im$, so that a neighbourhood will not expand past the image borders.

Theorem 5.1 Inverse convolution
The inverse convolution written as RNO according to Equation (5.2) is equivalent to the global operation according to Equation (5.3).

Proof 5.1 Inverse convolution
Equation (5.2) which describes the RNO can be worked out step by step. In the first step, the following holds:

$Y^{(1)}_{\vec{p}} = \varepsilon \cdot \left\{ X_{\vec{p}} - \sum_{\vec{q} \in S} Y^{(0)}_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} \right\}$, *and because* $Y^{(0)}_{\vec{p}} = 0$ *this is equivalent to:*

$Y^{(1)}_{\vec{p}} = \varepsilon \cdot X_{\vec{p}}$.

$Y^{(2)}$ *can be constructed from versions of* $Y^{(1)}$ *which are translated along vectors* $\vec{q}_1 \in S$, *which yields:*

$Y^{(2)}_{\vec{p}} = \varepsilon \cdot \left\{ X_{\vec{p}} - \sum_{\vec{q}_1 \in S} Y^{(1)}_{\vec{p}-\vec{q}_1} \cdot c_{\vec{q}_1} \right\}$, *and because* $Y^{(1)}_{\vec{p}} = \varepsilon \cdot X_{\vec{p}}$, *this is equivalent to:*

$Y^{(2)}_{\vec{p}} = \varepsilon \cdot X_{\vec{p}} - \varepsilon^2 \sum_{\vec{q}_1 \in S} X_{\vec{p}-\vec{q}_1} \cdot c_{\vec{q}_1}$. *The construction of* $Y^{(3)}$ *out of* $Y^{(2)}$ *along translations* $\vec{q}_2 \in S$ *yields the expression:*

$$Y^{(3)}_{\vec{p}} = \varepsilon \cdot X_{\vec{p}} - \varepsilon \sum_{\vec{q}_2 \in S} \left\{ \varepsilon \cdot X_{\vec{p}-\vec{q}_2} - \varepsilon^2 \sum_{\vec{q}_1 \in S} X_{\vec{p}-\vec{q}_1-\vec{q}_2} \cdot c_{\vec{q}_1} \right\} \cdot c_{\vec{q}_2},$$

which can be rewritten as:

$$Y^{(3)}_{\vec{p}} = \varepsilon \cdot X_{\vec{p}} - \varepsilon^2 \sum_{\vec{q}_2 \in S} X_{\vec{p}-\vec{q}_2} \cdot c_{\vec{q}_2} + \varepsilon^3 \sum_{\vec{q}_{1,2} \in S} X_{\vec{p}-\vec{q}_1-\vec{q}_2} \cdot c_{\vec{q}_1} \cdot c_{\vec{q}_2}.$$

Repeating the process as started in these first three steps ultimately yields:

$Y^{(\infty)}_{\vec{p}} = -1 \sum_{i=1}^{\infty} \left((-\varepsilon)^i \cdot \sum_{\vec{q}_j \in S} X_{\vec{p}-\Sigma\vec{q}_j} \cdot \prod_{j=1}^{i} c_{\vec{q}_j} \right)$. *Instead of using several vectors* $\vec{q}_j \in S$, *one vector* $\vec{q} \in S^i$ *can be taken such that* $S^i = \{ \sum_{j=0}^{i} \vec{q}_j | (\vec{q}_j \in S) \}$. *When the weighting coefficients are equal, i.e.* $(\forall i, j)\, c_{\vec{q}_i} = c_{\vec{q}_j}$, *then the global operation as defined in Equation (5.3) is obtained:*

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = -1 \sum_{i=1}^{\infty} \left((-\varepsilon)^i \cdot \sum_{\vec{q} \in S^{i-1}} X_{\vec{p}-\vec{q}} \cdot c^i_{\vec{q}} \right)$$

The operation could also be written in the form introduced in Section 5.1, by calculating what the coefficient values become for each position in the image.

Although this section only serves to show the relation between RNOs and GlOs, we would like to show when inverse convolution is stable in the bounded-input-bounded-output (BIBO) sense. From the GIO expression of the inverse convolution it can be seen that stability in the BIBO sense is assured if the following holds:

$$|-\varepsilon \cdot \#S \cdot c_{\vec{q}}| < 1 \tag{5.4}$$

with: $\#S$ is the number of points in the point set S.

Theorem 5.2 Inverse convolution stability

An inverse convolution as given by Equation (5.3) or Equation (5.2) is stable in the BIBO sense (i.e. $(\forall \vec{p} \in \mathrm{Im})\ (|X_{\vec{p}}| < M_x \Rightarrow |Y_{\vec{p}}| < M_y)$ *, with* M_x *and* M_y *a finite number) if Equation (5.4) holds.*

Proof 5.2 Inverse convolution stability

Looking at Equation (5.3), we realise that the output Y will be within bounds, if the factors with which the values of X are multiplied (i.e. ε and $c_{\vec{q}}$*) remain within bounds. The development of these factors is as follows.*

$\sum_{i=1}^{\infty} (-\varepsilon)^i \cdot \left\{ \sum_{\vec{q} \in S^{i-1}} c^i_{\vec{q}} \right\}$*. Now recall, that we chose* $(\forall i, j)\ c_{\vec{q}_i} = c_{\vec{q}_j}$*, and that there are #S points in the point set S, the summation becomes:* $\frac{-1}{\varepsilon c_{\vec{q}}} \cdot \sum_{i=0}^{\infty} (-\varepsilon \cdot \#S \cdot c_{\vec{q}})^i$*, which will only converge, if* $|-\varepsilon \cdot \#S \cdot c_{\vec{q}}| < 1$.

□

5.2.2.2 Relative extreme

Haralick defined an operation which on every output point calculates the value of the highest extreme that it can reach by a monotonic path in the input image (Haralick 1981). Haralick suggests that this relative extreme operation can be written as the following RNO (this is the relative maximum form; replace 'max' by 'min' and ≥ by ≤ for the relative minimum form):

$$(\forall \vec{p} \in \mathrm{Im})\ Y_{\vec{p}} = \max_{\vec{q} \in S} \{ Y_{\vec{p}-\vec{q}} | (X_{\vec{p}-\vec{q}} \geq X_{\vec{p}}) \} \tag{5.5}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$.

We state that this RNO can be rewritten as the following GIO:

$$Y_{\vec{p}} = \max_{\pi \in \mathrm{Im}} \{ X_{\vec{p}-\pi} \} \tag{5.6}$$

with: $\pi = \left\{ \vec{q}_1, \vec{q}_2, \ldots, \vec{q}_n | (\forall j \leq n) \left(X_{\vec{p} - \sum_{i=1}^{j} \vec{q}_i} \geq X_{\vec{p} - \sum_{i=1}^{j-1} \vec{q}_i} \right) \right\}$

Theorem 5.3 Relative maximum

The relative maximum operation as given in Equation (5.5) can be rewritten as the global operation given in Equation (5.6).

Proof 5.3 Relative maximum

In the first step, $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$ *is substituted into* $Y^{(1)}_{\vec{p}} = \max_{\vec{q}_1 \in S} \left\{ Y^{(0)}_{\vec{p}-\vec{q}_1} \middle| (X_{\vec{p}-\vec{q}_1} \geq X_{\vec{p}}) \right\}$.

so that: $Y^{(1)}_{\vec{p}} = \max_{\vec{q}_1 \in S} \left\{ X_{\vec{p}-\vec{q}_1} \middle| (X_{\vec{p}-\vec{q}_1} \geq X_{\vec{p}}) \right\}$ *. For the second step,* $Y^{(1)}$ *is shifted along all possible* $\vec{q}_2 \in S$*, so that:*

$$Y^{(2)}_{\vec{p}} = \max_{\vec{q}_2 \in S} \left\{ \max_{\vec{q}_1 \in S} \left\{ X_{\vec{p}-\vec{q}_1-\vec{q}_2} \middle| (X_{\vec{p}-\vec{q}_1-\vec{q}_2} \geq X_{\vec{p}-\vec{q}_2}) \right\} \middle| (X_{\vec{p}-\vec{q}_2} \geq X_{\vec{p}}) \right\}$$

$$= \max_{\vec{q}_{1,2} \in S} \left\{ X_{\vec{p}-\vec{q}_1-\vec{q}_2} \middle| (X_{\vec{p}-\vec{q}_1-\vec{q}_2} \geq X_{\vec{p}-\vec{q}_2} \geq X_{\vec{p}}) \right\}$$

The process of deriving $Y^{(2)}$ *from* $Y^{(1)}$ *can be generalized, so that in the limit the following holds:*

$$\lim_{k \to \infty} Y^{(k)}_{\vec{p}} = \max_{\{\vec{q}_i\} \in S} \left\{ X_{\vec{p}-\Sigma\vec{q}_i} \middle| (X_{\vec{p}-\Sigma\vec{q}_i} \geq \ldots \geq X_{\vec{p}}) \right\} .$$

This can be rewritten into a GlO using a slightly different notation:

$$Y_{\vec{p}} = \max_{\pi} \{X_{\vec{p}-\pi}\} \text{, where } \pi = \left\{ \vec{q}_1, \vec{q}_2, \ldots, \vec{q}_n \middle| (\forall j \leq n) \left(X_{\vec{p}-\sum_{i=1}^{j}\vec{q}_i} \geq X_{\vec{p}-\sum_{i-1}^{j-1}\vec{q}_i} \right) \right\}$$

Notice that this operation can be done in a finite number of steps, because the path from one image point to any other image point consists of a finite number of steps.

5.2.3 The relation between RNOs and OOs

In this paragraph we will show that there are some object operations which can be written as recursive neighbourhood operations.

5.2.3.1 Distance Transform

The distance transform is an operation which transforms a binary source image into a grey value image, where each point in the destination image has a value which is equal to the shortest distance from that point to the background (i.e. zero if it is a background point). The distances are calculated according to a certain metric. Such a metric *d(...)* should fulfil the following criteria (Rosenfeld and Pfalz 1968):

1. *Symmetric: d(a,b) = d(b,a)*
2. *Positive definite: d(a,b) = 0, iff a=b*
3. *Triangular: d(a,b) + d(b,c) ≥ d(a,c)*

Some of the generally used distances, together with their definitions are shown here:

$$d_1(\vec{a}, \vec{b}) = |a_x - b_x| + |a_y - b_y| \text{ (city-block distance)} \quad (5.7)$$

$$d_{5_7}(\vec{a}, \vec{b}) = 7 \cdot \min(|a_x - b_x|, |a_y - b_y|) + 5 \cdot ||a_x - b_x| - |a_y - b_y|| \quad (5.8)$$

$$d_{\text{Euclidean}}(\vec{a}, \vec{b}) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \quad (5.9)$$

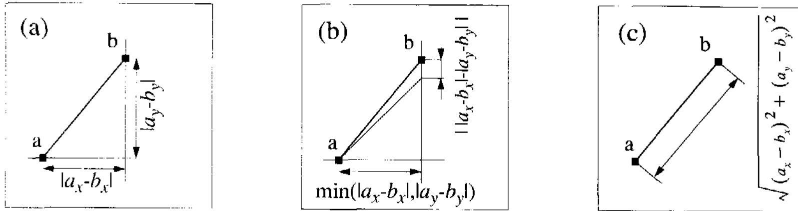


Figure 5.7 Illustration of distance measures: (a) city block, (b) 5_7 distance and (c) Euclidean.

It will now be shown, that the distance transform in OO-form can be written as:

$$(\forall \vec{p} \in \text{Im})\; Y_{\vec{p}} = \min_{\vec{p} - \vec{q} \notin \text{Obj}} d(\vec{p} - \vec{q}, \vec{p}) = \min_{\vec{p} - \vec{q} \notin \text{Obj}} \|\vec{q}\| \quad (5.10)$$

This can be explained by Figure 5.8. For every object point ($\vec{p} \in \text{Obj}$) the closest non-object point is searched for, and the distance from $\vec{p}$ to $\vec{p} - \vec{q}$ is stored in $Y_{\vec{p}}$ (see Figure 5.8). Every non-object point p has distance zero to its closest non-object point.

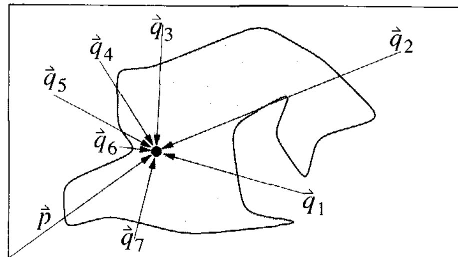


Figure 5.8 Search for the minimal distance from object point p to non-object points q_i, where q_6 is the best solution

We propose to write the distance transform as an RNO in the following way, which is derived from the available literature (Rosenfeld and Pfalz 1968; Haralick 1981):

$$(\forall \vec{p} \in \text{Im})\; Y_{\vec{p}} = \min_{\vec{q} \in S} (Y_{\vec{p} - \vec{q}} + d(\vec{p} - \vec{q}, \vec{p})) = \min_{\vec{q} \in S} (Y_{\vec{p} - \vec{q}} + \|\vec{q}\|) \quad (5.11)$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}} \cdot L$; L a value larger than the largest possible distance in the image, X a binary image, Y a grey value image and S a neighbourhood which includes the null-vector and is 'point symmetric' in the sense that if $\vec{q} \in S$, then also $-\vec{q} \in S$.

The distance transform as an RNO starts with an image filled with values L where there are object-points and with values 0 elsewhere. These values represent an initial distance. Then the RNO will take the minimum of the distance already stored on position p and the accumulated distances to the neighbours. This means that distances will be accumulated from points with initial 0-distance into objects with initial infinite distance.

The relation between the OO and RNO description of the distance transform, is stated as follows.

Theorem 5.4 Distance transform.

The distance transform as object operation in Equation (5.10) is equivalent to its corresponding recursive neighbourhood operation in Equation (5.11), iff the metric allows that

$$\left(\forall\vec{q} \in \sum_{\vec{q}_j \in S} \vec{q}_j\right)(\exists\{\vec{q}_i | \vec{q}_i \in S\})\sum\|\vec{q}_i\| = \|\vec{q}\|.$$

Proof 5.4 Distance transform.

The RNO-description of the distance transform $Y_{\vec{p}} = \min_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}} + \|\vec{q}\|)$ *can be rewritten by filling in the initial condition given in Equation (5.11), and working this out in consecutive iterations as follows:*

$$Y^{(0)}_{\vec{p}} = X_{\vec{p}} \cdot L$$

In the next iteration the $Y^{(1)}$ *is calculated from versions of the image* $Y^{(0)}$ *(i.e. on* $X \cdot L$*) which are shifted according to all vectors* $\vec{q}_1 \in S$*:*

$$Y^{(1)}_{\vec{p}} = \min_{\vec{q}_1 \in S}\{X_{\vec{p}-\vec{q}_1} \cdot L + \|\vec{q}_1\|\}$$

In the next step, $Y^{(2)}$ *is calculated from versions of* $Y^{(1)}$ *shifted according to all* $\vec{q}_2 \in S$ *as derived in the previous step:*

$$\begin{aligned} Y^{(2)}_{\vec{p}} &= \min_{\vec{q}_2 \in S}\{\min_{\vec{q}_1 \in S}(\{X_{\vec{p}-\vec{q}_1-\vec{q}_2} \cdot L + \|\vec{q}_1\|\} + \|\vec{q}_2\|)\} \\ &= \min_{\vec{q}_{1,2} \in S}\{X_{\vec{p}-\vec{q}_1-\vec{q}_2} \cdot L + \|\vec{q}_1\| + \|\vec{q}_2\|\} \end{aligned}$$

The process of deriving $Y^{(2)}$ *from* $Y^{(1)}$ *can be generalized, so that in the limit the following holds:*

$$\lim_{k \to \infty} Y^{(k)}_{\vec{p}} = Y_{\vec{p}} = \min_{\vec{q} \in S}\{X_{\vec{p}-\Sigma\vec{q}_i} \cdot L + \Sigma\|\vec{q}_i\|\} = \min_{\vec{p}-\vec{q} \notin \mathrm{Obj}} \|\vec{q}\|$$

In the final step, two things are done:

First, the vector $\sum\vec{q}_i$ *is chosen such, that* $\vec{p} - \sum\vec{q}_i$ *points to a background pixel, so that the value of* $X_{\vec{p}-\Sigma\vec{q}_i} \cdot L$ *is zero.*

Second, the summation of neighbourhood vectors $\vec{q}_i$ *is replaced by one* $\vec{q} = \sum\vec{q}_i$*, and the summation of individual lengths* $\|\vec{q}_i\|$ *by the length of the total vector* $\|\vec{q}\|$*. The latter can only be done, if the metric allows that the length of the vector* $\vec{q}$ *can be expressed as the summation of a number of individual lengths, i.e. if*

$$\left(\forall\vec{q} \in \sum_{\vec{q}_j \in S} \vec{q}_j\right)(\exists\{\vec{q}_i | \vec{q}_i \in S\})\sum\|\vec{q}_i\| = \|\vec{q}\|$$

As can be seen from this proof, the equality between the RNO-form and the OO-form of the distance transform is exact, if the distance measure used allows that,

$$\left(\forall\vec{q} \in \sum_{\vec{q}_j \in S} \vec{q}_j\right)(\exists\{\vec{q}_i | \vec{q}_i \in S\})\sum\|\vec{q}_i\| = \|\vec{q}\| \qquad (5.12)$$

We will now show, that Equation (5.12) holds for distances such as city-block, 5-7 and chessboard. Due to the associative quality of addition, to prove Equation (5.12) it suffices to prove that the following holds:

$$(\exists \vec{a}, \vec{b} \in S, \forall \vec{c} \in \vec{a} + \vec{b})\ (\|\vec{a}\| + \|\vec{b}\| = \|\vec{c}\|) \tag{5.13}$$

This will be proven only for the city-block metric, because the other metrics can be proven similarly.

Theorem 5.5 City-block metric

Equation (5.13) holds for the city block metric as defined in Equation (5.7), if used for the normal distance transform as defined in Equation (5.11).

Proof 5.5 City-block metric

Substituting the lengths of $\vec{a}$ and $\vec{b}$ as defined by the city-block metric yields the following condition which should be met:

$(|a_x| + |a_y|) + (|b_x| + |b_y|) = |c_x| + |c_y|$.

Using the information that $\vec{c} \in \vec{a} + \vec{b}$ shows that the following condition should be met:

$\{\exists \vec{a}, \vec{b} |\ ((|a_x| + |a_y|) + (|b_x| + |b_y|) = |a_x + b_x| + |a_y + b_y|)\}$

This condition can be fulfilled if we choose $\vec{a}, \vec{b}$ such that:

$(\text{sign}(a_x) = \text{sign}(b_x)) \wedge (\text{sign}(a_y) = \text{sign}(b_y))$.

For the Euclidian distance, Equation (5.13) does not hold, because of the fact that for a number of vectors $\vec{c}$ no $\vec{a}, \vec{b}$ can be found such that $\|\vec{a}\| + \|\vec{b}\| = \|\vec{c}\|$.

In this case, the distances which are accumulated with the RNO-form will be higher than the distances accumulated with the OO-form of the distance transform. When the distance transform OO 'requires' the Euclidian distance, then there is an RNO which will yield a result almost equal to the OO-form. Instead of calculating $\Sigma\|\vec{q}_i\|$ in the output image Y in the RNO, the vector-summation $\Sigma\vec{q}_i$ is updated in an image Z. A final pointwise operation calculates the distances from these vectors.

$$\begin{aligned}(\forall \vec{p} \in \text{Im})\ Z_{\vec{p}} &= \underset{\vec{q} \in S}{\text{argmin}}\ \|Z_{\vec{p}-\vec{q}} + \vec{q}\| \quad + Z_{\vec{p}} \\ Y_{\vec{p}} &= \|Z_{\vec{p}}\|\end{aligned} \tag{5.14}$$

with: $Z_{\vec{p}}^{(0)} = X_{\vec{p}} \cdot \begin{bmatrix} L \\ \bullet \\ L \end{bmatrix}$, and L a value larger than the largest possible distance in the image,

the image Z contains coordinates, Y contains distances (i.e. grey-values), and X is Boolean.

The *argmin* operation is defined as:

$$\underset{i \in B}{\text{argmin}}\quad x_i = \{i |\ (\forall (i, j \in B \wedge i \neq j))\ (x_i \leq x_j)\} \tag{5.15}$$

The RNO is initialised with coordinates Z reaching 'far enough' outside the image. Then in each step the distance $\|Z\|$ at position $\vec{p}$ decreases, if the length of the vector $\vec{q}$ pointing to the local neighbour added with the vector Z stored in the local neighbour at position $\vec{p} - \vec{q}$ is smaller than the length of the vector stored in Z at position $\vec{p}$. The smallest vector sum is stored in Z at position $\vec{p}$. The use of coordinates in the calculation of the distance transform was initially suggested in 1980 (Danielsson 1980).

5.2.3.2 Grey weighted distance transform (GWDT)

An analogy of the distance transform as applied to binary images with objects is the

grey weighted distance transform (Yokoi et al. 1981). This operation has two input images: a binary image B and a grey value image X. The result of the operation is a grey value image where a value in the output image represents the grey value weighted path from the corresponding point in the binary input image to the closest non-object point.

This operation can be described as follows. An output pixel value at position $\vec{p}$ is calculated by taking the path to a non-object point with minimal length, with regard to the weights found in X along the path. The 'paths' are described by a sequence of neighbourhood vectors $\vec{q}_i$. This can be described by the following OO:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \min_{\pi} \sum_i X_{\vec{p} - \vec{q}_i} \cdot \|\vec{q}_i\| \tag{5.16}$$

with: $\pi = \{\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_n | \left(\left(\vec{p} - \sum_{i=1}^{n} \vec{q}_i\right) \notin \mathrm{Obj}\right)\}$

The same operation can be expressed as an RNO as follows. An output pixel value at position $\vec{p}$ is calculated by taking the minimum of the distances stored in the neighbouring points, added with the weighted length to come to that neighbour. In this case the non-object pixels should be initialised with zero, and the object points with a 'large enough' value, so that the changes in calculated distances propagate from the edge points inwards. The GWDT-RNO is given in Equation (5.17):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \min_{\vec{q} \in S} (Y_{\vec{p} - \vec{q}} + X_{\vec{p} - \vec{q}} \cdot \|\vec{q}\|) \tag{5.17}$$

with: $Y_{\vec{p}}^{(0)} = B_{\vec{p}} \cdot L$, and L is a value larger than the largest weighted distance in the image, and the grey value coefficient image X fulfils $(\forall \vec{p} \in \mathrm{Im})\ (X_{\vec{p}} \geq 0)$.

We will now prove, that the RNO-description of the GWDT is indeed equivalent to the OO description, by working from the first into the latter.

Theorem 5.6 Grey weighted distance transform

The GWDT as described by the RNO in Equation (5.17) is equal to the GWDT as described by the OO in Equation (5.16).

Proof 5.6 Grey weighted distance transform

Taking the first step of the GWDT as described in Equation (5.17)

$$Y_{\vec{p}}^{(1)} = \min_{\vec{q}_1 \in S} \left\{ Y_{\vec{p} - \vec{q}_1}^{(0)} + X_{\vec{p} - \vec{q}_1} \cdot \|\vec{q}_1\| \right\}$$

, and filling in the initial condition

$Y_{\vec{p}}^{(0)} = B_{\vec{p}} \cdot L$, *yields the following:*

$$Y_{\vec{p}}^{(1)} = \min_{\vec{q}_1 \in S} \{ B_{\vec{p} - \vec{q}_1} \cdot L + X_{\vec{p} - \vec{q}_1} \cdot \|\vec{q}_1\| \}$$

For calculating Y^2 in the next step, shifted versions of $Y^{(1)}$ are required. This leads to:

$$Y_{\vec{p}}^{(2)} = \min_{\vec{q}_2 \in S} \left\{ \min_{\vec{q}_1 \in S} \left\{ (B_{\vec{p} - \vec{q}_1} \cdot L + X_{\vec{p} - \vec{q}_1} \cdot \|\vec{q}_1\|)_{\vec{q}_2} + X_{\vec{p} - \vec{q}_2} \cdot \|\vec{q}_2\| \right\} \right\}$$

$$= \min_{\vec{q}_{1,2} \in S} \{ B_{\vec{p} - \vec{q}_1 - \vec{q}_2} \cdot L + X_{\vec{p} - \vec{q}_1} \cdot \|\vec{q}_1\| + X_{\vec{p} - \vec{q}_2} \cdot \|\vec{q}_2\| \}$$

Generalizing the procedure to derive $Y^{(2)}$ from $Y^{(1)}$ leads to:

$$Y_{\vec{p}}^{(\infty)} = \min_{\vec{q}_i \in S} \left\{ B_{\vec{p} - \Sigma \vec{q}_i} \cdot L + \sum X_{\vec{p} - \vec{q}_i} \cdot \|\vec{q}_i\| \right\}$$

$$= \min_{\pi} \sum_i X_{\vec{p} - \vec{q}_i} \cdot \|\vec{q}_i\|$$

In the last step, the vectors $\{\vec{q}_i \in S\}$ *taken for the RNO are rewritten as the path*

$\pi = \left\{\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_n \middle| \left(\left(\vec{p} - \sum_{i=1}^{n} \vec{q}_i \right) \notin \text{Obj} \right) \right\}$, *so that* $B_{\vec{p} - \Sigma \vec{q}_i} \notin \text{Obj}$, *and hence*

$B_{\vec{p} - \Sigma \vec{q}_i} \cdot L = 0$

□

The choice of the binary image B in the calculation of the GWDT depends on the use of this operation (Verbeek and Verwer 1990).

5.2.3.3 Object Selection

The object selection operation takes a source image with a 'seed' in it. All pixels in the object, where the seed is, are set to one, and the rest of the pixels are set to zero. This definition of the object selection operation can be written mathematically as follows:

$$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = \bigcup_{\vec{p} - \vec{q} \in \text{Obj}} X_{\vec{p} - \vec{q}} \qquad (5.18)$$

An output pixel at position $\vec{p}$ will become one, if at least one of the pixels on the position of the object has a value of one (i.e. it is a seed).

The object selection as RNO requires that an image B is taken which contains all the objects. The output image Y is initially loaded with the seed image X. The output image is dilated under the condition that the expansion does not grow beyond the object borders. Object selection as RNO can therefore be written as:

$$(\forall \vec{p} \in \text{Im})\, Y_{\vec{p}} = \bigcup_{\vec{q} \in S} Y_{\vec{p} - \vec{q}} \wedge B_{\vec{p}} \qquad (5.19)$$

with: $Y_{\vec{p}}^{(0)} = X_{\vec{p}} \wedge B_{\vec{p}}$. The images B, Y and X are binary, and B contains all objects.

The equivalence between the RNO and OO representation of the object selection will now be stated and proven.

Theorem 5.7 Object selection

The object selection written as RNO according to Equation (5.19) is equivalent to the object selection written as OO according to Equation (5.18).

Proof 5.7 Object selection

The theorem will be proven by working the RNO description into the OO description. In the first step, the following holds according to Equation (5.19):

$Y_{\vec{p}}^{(1)} = \bigcup_{\vec{q} \in S} Y^{(0)}_{\vec{p} - \vec{q}} \wedge B_{\vec{p}}$, *where* $Y_{\vec{p}}^{(0)} = X_{\vec{p}}$, *so that* $Y_{\vec{p}}^{(1)} = \bigcup_{\vec{q} \in S} X_{\vec{p} - \vec{q}} \wedge B_{\vec{p}}$.

In the next step, $Y^{(2)}$ *is calculated by shifting* $Y^{(1)}$ *into directions* $\vec{q}_2 \in S$, *which results in orring the pixels over an expanded neighbourhood:*

$$Y_{\vec{p}}^{(2)} = \bigcup_{\vec{q}_{1,2} \in S} X_{\vec{p} - \vec{q}_1 - \vec{q}_2} \wedge B_{\vec{p}}$$

In following steps the neighbourhood over which points are orred is expanded even further:

$$Y_{\vec{p}}^{(\infty)} = \bigcup_{\vec{q}_i \in S} X_{\vec{p} - \Sigma\vec{q}_i} \wedge B_{\vec{p}} = \bigcup_{\vec{p} - \vec{q} \in \mathrm{Obj}} X_{\vec{p} - \vec{q}}$$

In the last step, the summation over all $\vec{q}_i \in S$ and the anding with B is substituted for one $\vec{q}$ which is chosen so that $\vec{p} - \vec{q} \in \mathrm{Obj}$. This yields the OO description.

5.2.3.4 Smallest Enclosing Regular Polygon

As the name indicates, the SERP operation tries to find the smallest regular (and thus convex) polygon with specific angles in a specific orientation that fits around the objects. When two or more polygons of objects overlap or touch, a polygon is taken which surrounds those objects. The number of sides (and their orientation) that the polygon will have depends on the number of different directions spanned by the vectors of the local neighbourhood set S, as will be shown in this section.

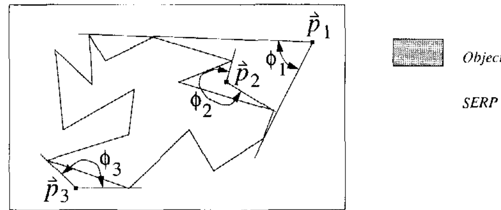


Figure 5.9 Determining if a point belongs to the SERP or not.

The SERP is calculated by checking recursively (for the RNO) or straight forward (for the OO) if a background pixel ought to belong to the polygon surrounding an object.

Assume that a polygon has P sides (e.g. P=8 for an octagon etc.). The angle α which the sides of a polygon make with one another is then given by:

$$\alpha = \frac{\left\lfloor \frac{P-1}{2} \right\rfloor}{P} \cdot 360° \tag{5.20}$$

Any background pixel observes the object with a certain angle (or the object pixels in the local neighbourhood). This is illustrated in Figure 5.9. Point $\vec{p}_i$ observes the object with *viewangle* ϕ_i. From the illustration it is clear, that if a point $\vec{p}_i$ has a corresponding viewangle $\phi_i \geq \alpha$, then it belongs to the SERP. For the SERP we restrict the angles which the 'arms' of the viewangle make and the viewangles themselves to be a multiple of $(360°)/P$.

The definition of the viewangle is given in Equation (5.21):

$$\phi_O(Y, \vec{p}, \vec{q}) = \max_{\vec{q}_{i,j} \in O} \left\{ \angle(\vec{q}_i, \vec{q}_j) \middle| \begin{pmatrix} Y_{\vec{p}}, Y_{\vec{p}-\vec{q}_{i+1}}, Y_{\vec{p}-\vec{q}_j} \notin O \wedge \\ Y_{\vec{p}-\vec{q}_i}, Y_{\vec{p}-\vec{q}_{j+1}} \in O \end{pmatrix} \right\} \tag{5.21}$$

with: O is an ordered set of vectors taken from S for RNOs, and from *Obj* for OOs,

$\angle(\vec{a}, \vec{c})$ = the angle made between vector $\vec{a}$ and vector $\vec{b}$ measured at the object side.

Note: If $Y_{\vec{p}} = 1$, then we define $\phi_O(Y, \vec{p}, \vec{q}) = 360°$.

The ordering of the sets S or *Obj* into O is done according to the following two criteria:

- The vectors of the set O have an increasing angle

- The vectors of the set O have angles which are a multiple of $\frac{360°}{P}$

For S this is defined as (the definition for deriving O out of Obj is analogously):

$$O = \left\{\vec{q} \mid (\vec{q}_i \in S) \wedge (\forall k < i)\ (\angle\vec{q}_i < \angle\vec{q}_k) \wedge \left(\angle\vec{q}_i = \left\lfloor 360°\frac{n}{P}\right\rfloor\right)\right\}.$$

Using the notion of the viewangle from Equation (5.21) and the polygon angle from Equation (5.20), the definition of the P-sided SERP from a source image X as an RNO is as follows:

$$(\forall\vec{p} \in \mathrm{Im})\ Y_{\vec{p}} = \begin{cases} \phi_S(Y, \vec{p}, \vec{q}) \geq \alpha_P \rightarrow 1 \\ \phi_S(Y, \vec{p}, \vec{q}) < \alpha_P \rightarrow 0 \end{cases} \tag{5.22}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$

The definition of the corresponding object operation is given in Equation (5.23):

$$(\forall\vec{p} \in \mathrm{Im})\ Y_{\vec{p}} = \begin{cases} \phi_{\mathrm{Obj}}(X, \vec{p}, \vec{q}) \geq \alpha_P \rightarrow 1 \\ \phi_{\mathrm{Obj}}(X, \vec{p}, \vec{q}) < \alpha_P \rightarrow 0 \end{cases} \tag{5.23}$$

Theorem 5.8 Smallest enclosing regular polygon

The SERP described as RNO according to Equation (5.22) is equivalent to the SERP written as OO according to Equation (5.23).

Proof 5.8 Smallest enclosing regular polygon

Due to the conditions which the SERP as we defined it should fulfil, rewriting the RNO into the corresponding OO starts as follows:

$$Y^{(1)}_{\vec{p}} = \begin{cases} \phi_S(Y^{(0)}, \vec{p}, \vec{q}) \geq \alpha_p \rightarrow 1 \\ \phi_S(Y^{(0)}, \vec{p}, \vec{q}) < \alpha_p \rightarrow 0 \end{cases}$$ *, and because* $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$*, it follows that:*

$$Y^{(1)}_{\vec{p}} = \begin{cases} \phi_S(X, \vec{p}, \vec{q}) \geq \alpha_p \rightarrow 1 \\ \phi_S(X, \vec{p}, \vec{q}) < \alpha_p \rightarrow 0 \end{cases}$$

In the next step, $Y^{(2)}$ *is calculated using* $Y^{(1)}$ *from the previous step, which yields:*

$$Y^{(2)}_{\vec{p}} = \begin{cases} \phi_{S^2}(X, \vec{p}, \vec{q}) \geq \alpha_p \rightarrow 1 \\ \phi_{S^2}(X, \vec{p}, \vec{q}) < \alpha_p \rightarrow 0 \end{cases}$$ *, with* $S^2 = \{\vec{q}_i + \vec{q}_j \mid (\vec{q}_{i,j} \in S)\}$ *being a 'twice extended neighbourhood'. The step from* $Y^{(1)}$ *to* $Y^{(2)}$ *can be taken because the maximum operation done in* $\phi_{S^2}(X, \vec{p}, \vec{q})$ *is distributive. Extending the neighbourhood further and further yields the OO description:*

$$Y^{\infty}_{\vec{p}} = \begin{cases} \phi_{S^\infty}(X, \vec{p}, \vec{q}) \geq \alpha_p \rightarrow 1 \\ \phi_{S^\infty}(X, \vec{p}, \vec{q}) < \alpha_p \rightarrow 0 \end{cases} \Leftrightarrow Y_{\vec{p}} = \begin{cases} \phi_{\mathrm{Obj}}(X, \vec{p}, \vec{q}) \geq \alpha_p \rightarrow 1 \\ \phi_{\mathrm{Obj}}(X, \vec{p}, \vec{q}) < \alpha_p \rightarrow 0 \end{cases}$$

□

We found that the actual calculation of the SERP in a 3*3 environment (i.e. for squares and octagons) is less cumbersome as shown above. For squares and octagons made with the vectors taken from a 3*3 neighbourhood, checking if the viewangle is larger than α is

similar (with a small difference) to checking if the number of object points in the neighbourhood is larger than or equal to half the total number of points neighbouring the central pixel:

$$(\phi_{3*3}(Y, \vec{p}, \vec{q}) \geq \alpha_p) = \left(\sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} \geq \left\lfloor \frac{\#S}{2} \right\rfloor\right) \tag{5.24}$$

One special case has to be considered. If the central pixel in the 3*3 neighbourhood belongs to the background, and the rest of the neighbourhood consists of more objects, then the pixel should also be made object pixel (i.e. it gets the value 1). This is done by the viewangle comparison, but not by simply counting the number of object pixels in the 3*3 neighbourhood. However, the special case can be identified by taking the *crossing number* into account (Rutovitz 1966; Preston 1983). The connectivity number in a 3*3 environment is defined as follows:

$$\mathrm{cnum}(Y_{\vec{p}}, S) = \sum_{\vec{q}_i \in S} \left|Y_{\vec{p}-\vec{q}_i} - Y_{\vec{p}-\vec{q}_{i+1}}\right| \tag{5.25}$$

with: $(\forall j > i)\ (\angle\vec{q}_i < \angle\vec{q}_j)$, i.e. the neighbourhood vectors are ordered to increasing angle.

This measure counts the number of transitions between object and non-object pixels (excluding the central pixel). If the neighbourhood only contains object pixels or only non-object pixels, then the crossing number is zero. Near the edge of an object, the crossing number will be two. Only if the special case as discussed above arises, does the crossing number become larger than two.

The formula for the square or octagon SERP in a 3*3 neighbourhood S is as follows:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = Y_{\vec{p}} \vee \left(\sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} > \frac{\#S}{2}\right) \vee (\mathrm{cnum}(Y_{\vec{p}}, S) > 2) \tag{5.26}$$

with: S is a convex point set, symmetric around $\vec{q} = 0$,

$\#S$ = the number of points in the set S

The SERP Equation (5.26) has been used in Chapter 6 as an example, and in Chapter 8 with the experiments.

5.2.4 Conclusions

A large group of object operations, together with some global operations can be seen as a subset of the RNO group. RNOs can also be seen as sequentially or repeatedly executed LNOs.

5.3 Updating methods

The order in which points are updated to calculate a specific RNO influences the speed of calculation and may result in different solutions for some RNOs. Many different updating methods have been developed. We propose dividing the updating methods into three groups:

- *Deterministic*. The order in which the points in an image are calculated is a deterministic function of the image coordinates.
- *Data dependent*. The order in which image points are calculated depends on the change that image points undergo.

- *Stochastic*. The order in which image points are calculated depends on a stochastic function of the image coordinates.

Combinations of these groups are also possible. The implementation of the updating methods into the architecture groups is discussed in Chapter 7. The efficiency of the updating methods as applied to some RNOs will be measured experimentally in Chapter 8. However, an attempt will be made in this section to predict the efficiency of updating methods to some extent. This is done by calculating the *updating fraction* Q. This is defined as the relative number of points in the neighbourhood S which have been updated prior to the central pixel but in the same pass. The higher the percentage of the neighbourhood that has been calculated *before* the central pixel, the better the performance. Like the different updating methods, the update percentage may also be deterministic, data dependent or stochastic. In this paragraph we will treat some updating methods from each of the groups.

5.3.1 Deterministic updating methods

The function of the image coordinates which determines the order in which the image points are calculated is called the Index Mapping Function (IMF) or order function (Dudgeon and Mersereau 1984; Manry and Aggarwal 1974). The IMF assigns a rank r to each image point (i,j):

$$r = I(i, j) \tag{5.27}$$

Points with the same rank are calculated at the same time. Points with higher rank are calculated after points with a lower rank. The updating fraction Q is now calculated by taking the percentage of points in the neighbourhood that has a rank value smaller than the value of the central pixel:

$$Q_{\text{deterministic}\vec{p}} = \frac{1}{\#S} \cdot \sum_{\vec{q} \in S} \{I(\vec{p}) \geq I(\vec{p} - \vec{q})\} \tag{5.28}$$

with: $\{I(\vec{p}) \geq I(\vec{p} - \vec{q})\} = \begin{cases} I(\vec{p}) \geq I(\vec{p} - \vec{q}) \rightarrow 1 \\ I(\vec{p}) < I(\vec{p} - \vec{q}) \rightarrow 0 \end{cases}$, and $\#S$ is the number of points in the neighbourhood set S.

As the updating fraction may differ from point to point, we also define the average updating fraction $\overline{Q}$ as follows:

$$\overline{Q} = \frac{1}{\#\text{Im}} \sum_{\vec{p} \in \text{Im}} Q_{\vec{p}} \tag{5.29}$$

with: #Im is the number of points in the image.

In Table 5.2 a number of index mapping functions are listed, with their respective names, and the average fraction of points $\overline{Q}$ in a 3*3 neighbourhood that have a rank value smaller then the central pixel.

In the simultaneous updating method, all destination points are calculated at the same time. Therefore, this updating method is equal to iteration. The raster scan updating method is accomplished by scanning the image from the top left to the bottom right point. This method has an updating fraction of 4/9 for a 3*3 neighbourhood. The position of these four pixels relative to the output pixel to be calculated does not change across the image. The relative position of the neighbourhood points from the updating fraction does vary for the meander, spiral and blob updating methods.

*Table 5.2 Index mapping functions (IMF) and average updating fraction ($\bar{Q}$) for 3*3 neighbourhood of deterministic updating methods*

IMF:	9*Q:	Name(s):
$r = 1$	0	Simultaneous ; Jacobi
$r = j$	3	Line serial
$r = i$	3	Column serial
$r = i + w \cdot j$	4	Serial ; Raster scan ; Gauss Seidel
$r = h \cdot i + j$	4	Vertical serial
$r = (i+j) \cdot \frac{(i+j+1)}{2} + i$	4	Diagonal serial
$r = (i+j) \bmod 2$	2	Chessboard ; Alternating
$r = (j \bmod 2) + 2 \cdot (i \bmod 2)$	4	8-connected alternating
see Figure 5.10	4	Right spiral updating
see Figure 5.10	4	Left spiral updating
$r = i \cdot (-1)^j + w \cdot (i + j \bmod 2)$	4	Meander updating
w = width; h = height		

Some of the IMFs can best be understood by considering the two-dimensional rank table. The left and right spiral updating are only defined in such a way, i.e. not through the use of a formula. Some two-dimensional tables belonging to IMFs for 64*64 images are shown in Figure 5.10.

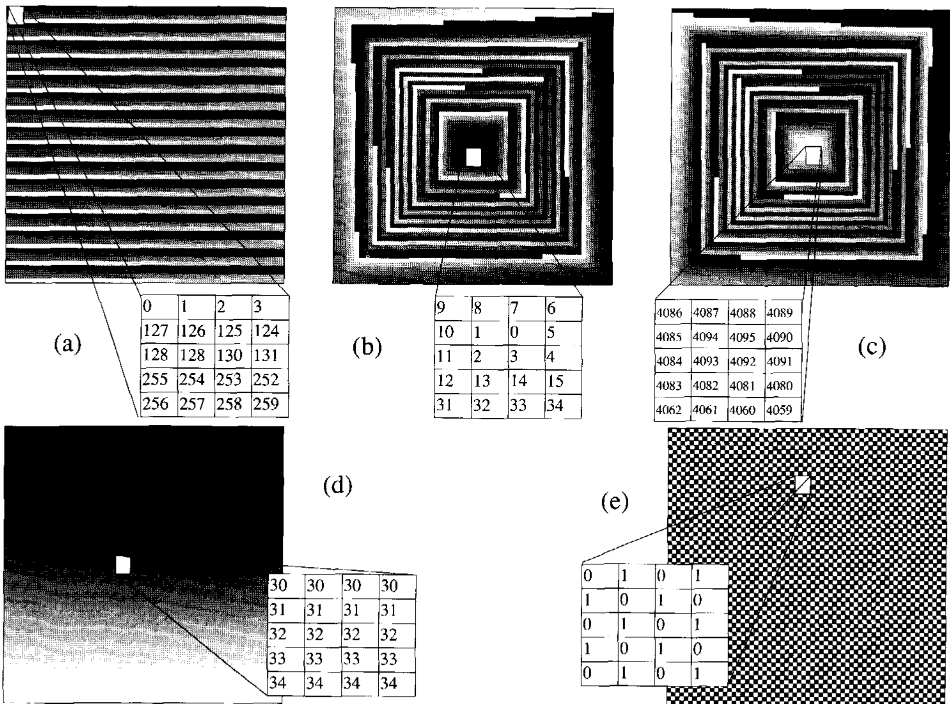


*Figure 5.10 Index Mapping Functions for a 64*64 image (the grey values represent the updating rank modulo 256): (a) meander updating, (b) left spiral updating, (c) right spiral updating, (d) line serial updating, and (e) 2-alternated updating.*

The deterministic updating methods can be extended with the successive over relaxation method (SOR), if the RNO is suitable (Hockney and Jesshope 1981). With the SOR extension, the new value of a destination pixel is a combination of the previous value at that position and the application of the RNO at that point. This is done the following way:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}}^{(k+1)} = (1-\omega) \cdot Y_{\vec{p}}^{(k-1)} + \omega \cdot f_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}^{(k)}, X_{\vec{p}}, c_{\vec{q}}) \qquad (5.30)$$

with: ω = the over relaxation factor, with a value between zero and one,

k = the time step at which the point is calculated.

The SOR methods have only been applied to linear RNOs, which are not the scope of this thesis. It is shown in the literature, that extending deterministic methods like serial and alternated updating with SOR gives very fast results for the solution of the discrete Poisson problem (Leveque and Trefethen 1988). An equivalent to the SOR extension for non-linear RNOs has not (yet) been found.

5.3.2 Data dependent updating methods

Some data dependent updating methods have been made for specific RNOs (e.g. skeletonization). We will focus on data dependent methods which are applicable either in general or to a well defined group of RNOs.

Data dependent methods differ in the (data-dependent) way that they cross the "precedence graph" (Chan 1976). Every destination pixel has a precedence graph associated with it. The graph consists of nodes associated with pixel positions and arcs between neighbouring pixel positions, so that it is completely dependent on the neighbourhood point set S. The top of the precedence graph is a node associated with the destination pixel position. Arcs go from this node to all those destination pixel values which have to be known before this destination pixel value can be calculated. An illustration of a precedence graph for a 1-dimensional image is given in Figure 5.11.

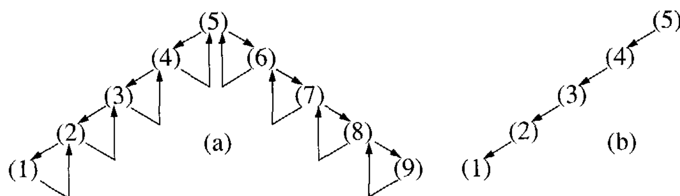


Figure 5.11 Precedence graph for point (5) of a one dimensional image with a neighbourhood consisting of (a) the left and right neighbour and (b) the left neighbour.

The precedence graphs can be traversed in two *top down* ways: depth first, or breadth first. To assure that all the destination pixels in an image are calculated, each precedence graph associated with a destination pixel is traversed separately. The data dependency is introduced in both methods by checking if the processing of the destination pixel changed the value of that pixel.

The depth first method was introduced by Piper in relation to distance transforms, and is called *recursive*[1] *updating* (Piper and Granum 1987). It can be programmed as shown by the pseudo code (conventions taken from Sommerville 1982) in Procedure 5.1. The RNO is solved by starting at the top of the precedence graph for every pixel, and going down to *all*

1. The term *recursive* here indicates programmatic recursion, whereas *recursive* in RNOs indicates spatial recursion.

its neighbours recursively if the processing of a pixel associated to a node changes that pixel value. Piper shows that changing the order in which the neighbours are processed can influence the speed of processing to some extent (Piper and Granum 1987).

Procedure 5.1 Recursive updating

```
for "all points in the image" do
  Evaluate("This Point")
od

PROC Evaluate("image point")
  if "point is not yet processed" then
     "process this point"
     if ("the value of the point has changed") then
        "call Evaluate for all neighbouring points"
     fi
  fi
CORP
```

Another way to walk the precedence graph is breadth first. This can be implemented using queues in which pointers to the pixels which have to be processed are stored. The pseudo code for such an algorithm is shown in Procedure 5.2.

Procedure 5.2 Queue updating

```
for "all points in the image" do
  if "point is not yet processed" then
     "put point on the queue"
  fi
  while ("there is a point on the queue") do
     "dequeue the point and process it"
     if "the value of the point has changed" then
        "enqueue all neighbour points of this point, if not already
         on the queue"
     fi
  od
od
```

A selection of points of the image are initially put on the queue. This is illustrated in Figure 5.12. Each point is "dequeued" and processed. If processing them changes anything, their neighbours have to be revisited, for which purpose the neighbours are again put on the queue. Note, that if a point is already on the queue, then it is not again enqueued. The process goes on until the queue is empty and all the image points are processed at least once. The speed of this queue processing technique depends to some extent on a clever initial choice of the queue.

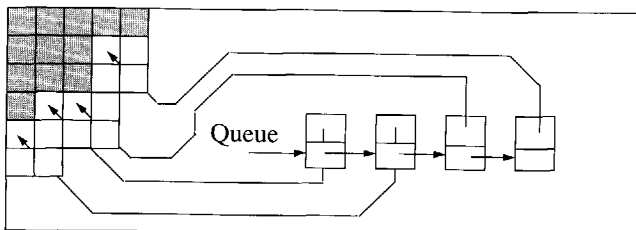


Figure 5.12 Processing an RNO according to the queue updating

A special version of the queue updating method has been demonstrated to be very efficient for binary operations such as propagation and skeletonization implemented on a single general purpose processor (van Vliet and Verwer 1987). For these operations, the contour of the binary input image is put initially in the queue. Successive processing of the queue has the effect that the processing propagates like a wavefront through the objects. This kind of processing is especially effective for monotonically increasing (or decreasing) binary RNOs (see Section 5.4), because pixels which are one (or zero for decreasing RNOs) do not have to be queued again. The processing time is thereby proportional to the number of object pixels.

Instead of the top-down methods, a bottom-up method can be used. For a specific class of RNOs, *bucket updating* is a highly efficient uniform cost method. Bucket updating is strongly inspired by the use of the A* algorithm in path finding (Verwer et al. 1989). The RNOs should fulfil the following two conditions:

- The value on an image point becomes monotonically higher (or monotonically lower) under successive applications of the RNO function.
- The points where the lowest (or highest) values of the output image will be must be known.

For an RNO which fulfils those two conditions, all the precedence graphs of all the destination pixels can be traversed in such a way that each destination point is visited a minimum number of times. The updating method proceeds as follows:

- First pointers to the points with lowest (or highest) values are stored in a bucket which is labelled with that value.
- Depending on whether the read or write formalism is used (Verwer et al. 1989):
  - *Read formalism:* One by one points are taken from this bucket, processed, and if they have changed a pointer to them is put in the bucket which corresponds to their output value.
  - *Write formalism:* A point is taken from this bucket, its neighbours are processed, and if a neighbour is changed 'enough', it is put in the bucket which corresponds to its new output value.
- When the points in the bucket which is labelled with the lowest output value are all processed, the bucket with the next highest output value is then processed.
- This process is repeated until all the image values have been evaluated.

As shown experimentally in Chapter 8, the bucket updating method is optimal for most members of this class of RNOs. For binary image processing bucket updating is (almost) equivalent to queue updating (the precedence graph is traversed using uniform cost, but for binary images the 'costs' are either 0 or 1).

5.3.3 Stochastic updating techniques

In these updating methods, the instant at which a destination pixel is updated depends on a stochastic process. Two examples of such updating methods are:

- *Asynchronous updating.* The method with which the CLIP4 calculates Boolean RNOs (see Chapter 2).
- *Poisson updating.* Each point is updated at random with a uniform distribution in time (Toffoli and Margolus 1987).

With the asynchronous updating all image point are calculated continuously, but the

time it takes to calculate a point differs in time (within a PE) and in space (between PEs). Due to the continuous updating it is not possible for asynchronous updating to define exactly what is meant by the updating fraction Q, so that this measure can not be used to indicate its performance. The time it takes to calculate the value of a point, given that all the neighbour values are known, can be written as:

$$t_{\text{point}} = t_{\text{average}} + t_{\text{variable}} \cdot \rho \tag{5.31}$$

with: ρ = random value uniformly distributed between -1 and 1.

For some RNOs the asynchronous updating will be advantageous due to the possibility that the time it takes to process a point so that it reaches a correct value may be less than average. This is for RNOs where the value of a point can already be known if only a few or just one neighbour has a particular value (e.g. for object selection). For RNOs where the value of a point can only be known if all its neighbours are correctly known, asynchronous updating will increase the required overall processing time. This is due to the fact that there is a probability for every point that it is calculated in a time slightly longer than average.

With Poisson updating, however, a point is updated at a random time t, which is uniformly distributed between t_1 and t_2. The probability that two points are updated concurrently is zero. The process of Poisson updating can be compared with assigning a unique rank (as used in deterministic updating) to every image point, so that the ranks are uniformly distributed between 1 and $N*N$ (i.e. the number of points in the image). In determining the updating fraction Q for Poisson updating, we cannot count deterministically how many points of the *neighbourhood* have been updated within the interval t_1 and t, but we can calculate the probabilities that the neighbourhood points were updated between the interval, so that:

$$Q_{\text{Poisson}} = \frac{1}{\#S} \cdot \sum_{\vec{q} \in S} \text{Prob}(I(\vec{p} - \vec{q}) < I(\vec{p})) \tag{5.32}$$

Due to the uniform distribution, $\text{Prob}(I(\vec{a}) < I(\vec{b})) = 0.5$, and $\text{Prob}(I(\vec{a}) < I(\vec{a})) = 0$ (for the central pixel). Thus the expected updating fraction Q for a 3*3 neighbourhood is 4/9. A characteristic of this method is, that updating takes place in a direction independent (i.e. spatially random) way.

5.3.4 Updating methods revisited

Three principally different updating method groups have been presented: deterministic, data dependent and stochastic. The performance of the deterministic methods may be predicted by looking at their updating fraction. A drawback of deterministic methods is that points which have already reached a steady value are still recalculated every iteration.

Data dependent updating methods can also be ordered as to their performance, but then on the basis of the efficiency with which they will traverse the precedence graph. Recursive updating is the worst, followed by queue updating and then by bucket updating. The latter is the best method known for the calculation of monotonically increasing (decreasing) RNOs with known positions of the lowest (highest) values. For other RNOs, queue updating should be used. This method does not recalculate most of the points which have reached a steady value. However, this also depends on the choice of the initial queue.

The asynchronous updating method may be advantageous only for some specific RNOs. The Poisson method will be advantageous for those RNOs requiring isotropic updating.

The implementation of the updating methods in several architectures and the associated performance for some real RNOs will be discussed in Chapter 7 and Chapter 8.

5.4 Theoretical considerations on fixed points from RNOs

When RNOs are used as OOs or GIOs, it is especially interesting to look at those for which the calculation of an RNO will finally converge to a solution whereby no point in the image will change anymore. Such a solution image is called a *fixed-point image* or a *root image*. The exact definition of a fixed-point is given in Definition 5.1.

Definition 5.1 A *fixed point* of an RNO is an image in which at all positions $\vec{p}$ the value $Y_{\vec{p}}$ will not change after the application of the RNO.

In this paragraph stability in the fixed point sense will be discussed in relation to other stability definitions. Attention will then be focused on the derivation of criteria with which the number of fixed points for an RNO can be determined. To this aim, the notion of "cyclicity" for the neighbourhoods used in RNOs is introduced (Otto 1984).

5.4.1 Fixed point stability and uniqueness

An RNO which has a fixed point image can be regarded as stable, as opposed to an unstable RNO, which does not have a fixed point image. To clarify the difference between this type of stability and other types, note the following alternative definitions of stability and their relation to fixed point stability:

- *Asymptotic stability.* With this kind of stability the value $Y_{\vec{p}}$ of every point at position $\vec{p}$ converges to a limit value $L_{\vec{p}}$ which it will not reach exactly (see Equation (5.33)). Although it is never reached a fixed point does exist (i.e. $L_{\vec{p}}$). This type of stability is therefore a subset of the fixed point stability.

$$\lim_{k \to \infty} Y_{\vec{p}}^{(k)} = L_{\vec{p}} \tag{5.33}$$

- *Bounded Input Bounded Output Stability (BIBO).* Stability in the BIBO sense means, that if the input values $X_{\vec{p}}$ of an image is bounded between two finite values, then the output values $Y_{\vec{p}}$ will also be bound between finite values. Output values which change periodically between two limits also fulfil this criterion, whereas such functions would be unstable in the fixed point sense.

For RNOs, only the fixed point stability/instability criterion is of interest. This criterion can be used to check if an RNO is finished or not. If there is no point in the image that changes after applying the RNO on it, then the RNO is finished.

In general, a distinction between RNOs can be made on the basis of the number of possible fixed points they have, given an arbitrary input image:

- *Zero fixed points.* This is called an 'unstable' RNO (in the fixed point sense it will never converge, although in the BIBO sense it may be stable).
- *One fixed point (or: update independent fixed point).* This will be called a 'unique' RNO, as it gives one unique solution. Unique RNOs are of particular interest as they allow the application of the fastest updating method in most cases (see the experimental results in Chapter 8).

- *More than one fixed point (or: update dependent fixed points):* This is called a 'non-unique' RNO. Some non-unique RNOs are of interest as well.

Only stable RNOs in the fixed point sense are of interest to low-level image processing. We will therefore try to derive some theorems which make it possible to distinguish between classes of RNOs which are stable and classes which are not.

5.4.2 Cyclicity and RNOs

For the derivation of the fixed point criteria, the notion of cyclicity has to be related to RNOs. The term cyclicity was first introduced by Otto in relation to instruction definitions for the CLIP4 processor array (Otto 1984). No formal definition was given, however. Depending on the structuring element S, an RNO is cyclical or non-cyclical. What is meant by a cyclical RNO is given in Definition 5.2.

Definition 5.2 An RNO is cyclical, if the calculation of a point in some way depends on its own value, i.e. $Y_{\vec{p}} = f_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}, X_{\vec{p}}, \vec{q}) = g(Y_{\vec{p}})$.

The cyclicity can be completely determined from the neighbourhood S of an RNO. If it is possible to put the vectors q of neighbourhood S in such an order, that the tail of the last vector reaches the head of the first, then the RNO has to be calculated using its own value, so that it is cyclical. From this observation, the following theorem is stated.

Theorem 5.9 Cyclicity

A structuring element S consisting of N points $\{q_1, \ldots, q_N\}$ used in an RNO causes cyclicity, iff for any $\{a_i \in N^+, i=1...N\}$, $\Sigma\, a_i \cdot q_i = 0$.

Proof 5.9 Cyclicity

From the definition of an RNO, it can be seen that a new point is calculated according to:

$$Y_{\vec{p}}^{(k+1)} = f_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}^{(k)}, X_{\vec{p}}, \vec{q})$$

In this iteration $Y_{\vec{p}}^{(k+1)}$ is dependent on any $Y_{\vec{p}-\vec{q}_i}$, therefore on $Y_{\vec{p}-\vec{q}_1}$, so that $Y_{\vec{p}}^{(k+1)} = g_1(Y_{\vec{p}-\vec{q}_1}^{(k)})$. However, $Y_{\vec{p}-\vec{q}_1}^{(k)}$ is on its turn dependent on $Y_{\vec{p}-\vec{q}_1-\vec{q}_2}$ .and when we go back n *iterations a whole 'path' $\{\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_n\}$ of dependencies is followed, so that:*

$$Y_{\vec{p}}^{(k+1)} = g_n(Y_{\vec{p}-\Sigma\vec{q}_i}^{(k-n)})_{i=1\ldots n}$$

This means that $Y_{\vec{p}}$ will be a function of itself (i.e. will be cyclical) if there exists a path $\{q_1 \ldots q_n\}$ for which $\Sigma\, q_i = 0$.

For a further discussion on cyclicity we define the 'minimal path length' π_l as follows:

$$\pi_l = \min_n \left(\{\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_n\} \middle| \left(\sum_{i=1}^{n} \vec{q}_i = 0 \right) \right) \tag{5.34}$$

*Table 5.3 Cyclical and non-cyclical 3*3 neighbourhoods*

	Non-cyclical:		Cyclical:					
Minimal path length:	0	0	1	1	2	2	3	4
Neighbourhood	□□□ □□□ ■■■	■■■ ■□□ □□□	□□□ □■□ □□□	■■■ ■■□ □□□	□■□ □□□ □■□	■□■ □□■ □□■	■□□ □□■ □■□	□■□ □□□ ■□■

Some examples of cyclical and non-cyclical neighbourhood point sets S are shown in Table 5.3. Note, that the 'central pixel' is in the middle of the 3*3 neighbourhoods, and that neighbourhood coordinates are taken relative to this central pixel. Depending on the minimum length of possible paths connecting a pixel with itself through its neighbours, we could speak of non, 1, 2, 3 and 4 cyclicity in a 3*3 neighbourhood. The 4-cyclical 3*3 neighbourhood for instance consists of the following point set:

$$S_{\text{4-cyclical 3*3}} = \{ (0,1), (1,-1), (-1,-1) \}$$

The fact that this neighbourhood point set is indeed cyclical can be seen by adding two times (0,1) with one time (1,-1) and one time (-1,-1), resulting in (0,0).

The notion of cyclicity for RNOs will be used further in the determination of the number of fixed points.

5.4.3 Criteria for RNOs concerning the number of fixed points

As was described in the previous paragraph, RNOs may have zero, one or more than one fixed point. The RNOs which have at least one fixed point are of interest to image processing, as image processing wants to transform one image into another well defined image. In this section conditions will be found to test whether an RNO has no fixed point(s). Some of the RNOs which have no fixed points are interesting for other purposes. One application is their usage in 'Modelling' (Toffoli and Margolus 1987). The so-called spin-flip RNO illustrates this:

$$(\forall \vec{p} \in \mathrm{Im})\; Y_{\vec{p}} = \begin{cases} \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} = 2 & \rightarrow \neg Y_{\vec{p}} \\ \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} \neq 2 & \rightarrow Y_{\vec{p}} \end{cases} \tag{5.35}$$

with: $Y_{\vec{p}}^{(0)}$ = 'a percentage of uniform distributed pixels with value one', Y is a binary image,

and the neighbourhood S is the following: (the dot indicates the central pixel).

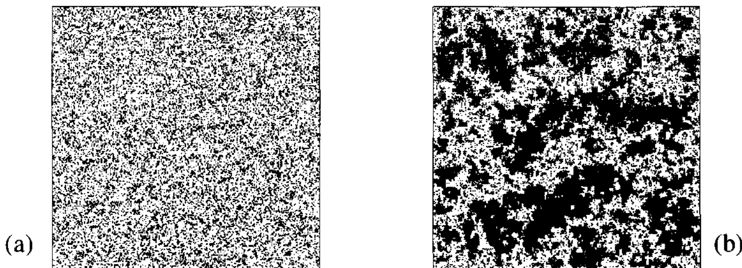


Figure 5.13 Spin flip example: (a) Initial 25% randomness (b) equilibrium after 1000 steps.

If exactly two points in the specified neighbourhood are equal to two, then the central pixel is inverted (i.e. 'flipped'). Otherwise the central pixel keeps its value. With an initial percentage enough above or below the 50% (i.e. 25 or 75%), the image will converge to a set of regions which from iteration to iteration keep on moving, but still stay 'regions'[1].

An intriguing question now is, whether it can be predicted if an RNO has or does not have a unique fixed point. A number of theorems are presented, which give some constraints on RNOs which are sufficient for it to have a (unique) fixed point. The theorems are preceded by a number of definitions.

First we define in Definition 5.3 what is meant with calculating an RNO using a deterministic updating method:

Definition 5.3 Deterministic updating

A deterministic updating method calculates and RNO according to:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}}^{(k+1)} = f_{\vec{q}_1 \in S_1, \vec{q}_2 \in S_2}(Y_{\vec{p}-\vec{q}_1}^{(k+1)}, Y_{\vec{p}-\vec{q}_2}^{(k)}, X_{\vec{p}-\vec{q}_1}, X_{\vec{p}-\vec{q}_2}, \vec{q}_1, \vec{q}_2),$$

where the complete neighbourhood S is divided into two disjoint sets S_1 and S_2 in the following way: $S_1 = \{\vec{q} \in S \mid (I(\vec{p}-\vec{q}) < I(\vec{p}))\}$ and $S_2 = \{\vec{q} \in S \mid (I(\vec{p}-\vec{q}) \geq I(\vec{p}))\}$.

The function $I(\ldots)$ is the index mapping function (IMF) as defined in Section 5.3.1, Equation (5.27). The IMF specifies the order in which the points $\vec{p}$ are updated. Within one pass (k) of the deterministic updating method, points with rank (r) are updated first, then points with rank (r+1) and so on.

For short we write a deterministic updating method as:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}}^{(k+1)} = f_{\vec{q}_{1,2} \in S_{1,2}}(Y_{\vec{p}-\vec{q}_{1,2}}^{(k+1,k)}, X_{\vec{p}-\vec{q}_{1,2}}, \vec{q}_{1,2}).$$

Note that dividing S as done in Definition 5.3 has as a consequence that: $S_1 \cap S_2 = \varnothing$, $S_1 \cup S_2 = S$ and also that $S_2 \neq \varnothing$. A a special case of deterministic updating methods, simultaneous updating has $S_1 = \varnothing$. This leads to Definition 5.4:

Definition 5.4 Simultaneous updating

The simultaneous updating method calculates an RNO according to:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}}^{(k+1)} = f_{\vec{q} \in S}(Y_{\vec{p}_1-\vec{q}}^{(k)}, X_{\vec{p}-\vec{q}}, \vec{q}).$$

For the fixed point theorems we also need to define the notion of monotonically increasing (decreasing) RNOs.

Definition 5.5 Monotonically increasing/decreasing

An RNO is monotonically increasing (decreasing), if for all points $\vec{p}$ in the image the value $Y_{\vec{p}}$ increases (decreases) monotonically after applying the RNO to those points, i.e.:

$$(\forall \vec{p} \in \mathrm{Im})\left(Y_{\vec{p}}^{(k+1)} \geq Y_{\vec{p}}^{(k)}\right)$$

$$\Leftrightarrow (\forall \vec{p} \in \mathrm{Im})\left(Y_{\vec{p}}^{(k+1)} \geq f_{\vec{q}_{1,2} \in S_{1,2}}(Y_{\vec{p}-\vec{q}_{1,2}}^{(k+1,k)}, X_{\vec{p}-\vec{q}_{1,2}}, \vec{q}_{1,2})\right)$$

In the following theorem we will show that the monotonically increasing (decreasing)

1. Not in the sense that pixels are connected, but in the sense that many pixels of the same "colour" cluster together.

property is sufficient for an RNO to guarantee that it has a fixed point.

Theorem 5.10 Fixed point

An RNO which is monotonically increasing (decreasing) and has an upper (lower) bound, has a (possibly updating method dependent) fixed point.

Proof 5.10 Fixed point

Because the RNO is monotonically increasing we know that for a deterministic updating method:

$$(\forall \vec{p} \in \mathrm{Im}) \left[Y_{\vec{p}}^{(k+1)} \geq Y_{\vec{p}}^{(k)} \right] \Leftrightarrow$$

$$(\forall \vec{p} \in \mathrm{Im}) \left(f_{\vec{q}_{1,2} \in S_{1,2}}(Y_{\vec{p}-\vec{q}_{1,2}}^{(k+1,k)}, X_{\vec{p}-\vec{q}_{1,2}}, \vec{q}_{1,2}) \geq f_{\vec{q}_{1,2} \in S_{1,2}}(Y_{\vec{p}-\vec{q}_{1,2}}^{(k,k-1)}, X_{\vec{p}-\vec{q}_{1,2}}, \vec{q}_{1,2}) \right)$$

Because of this and because of the existence of an upper bound L it follows that:

$$(\forall \vec{p} \in \mathrm{Im}) \left(\lim_{k \to \infty} \left\{ Y_{\vec{p}}^{(k+1)} - Y_{\vec{p}}^{(k)} \right\} = 0 \right)$$

which means that there is a fixed point.

□

Note that for real-valued RNOs/images the fixed point image may not be reached in finite time (see Section 5.4.1), although the fixed point image does exist. For integer-valued RNOs/images the fixed point which is meant by Theorem 5.10 is always reached in finite time.

Concerning the relations between neighbourhood we define a ranking order between them as follows:

Definition 5.6 Neighbourhood relations

Denote a neighbourhood of the image Y around a point $\vec{p}$ as $N_{\vec{p}} = \{ Y_{\vec{p}-\vec{q}} | (\vec{q} \in S) \}$.

A neighbourhood $N_{\vec{p}_1}$ is 'strictly higher' than a neighbourhood $N_{\vec{p}_2}$ iff:

$(\forall \vec{q} \in S)\ (Y_{\vec{p}_1 - \vec{q}} > Y_{\vec{p}_2 - \vec{q}})$.

A neighbourhood $N_{\vec{p}_1}$ is 'higher' than a neighbourhood $N_{\vec{p}_2}$ iff:

$(\forall \vec{q} \in S)\ (Y_{\vec{p}_1 - \vec{q}} \geq Y_{\vec{p}_2 - \vec{q}})$.

A neighbourhood $N_{\vec{p}_1}$ is equal to a neighbourhood $N_{\vec{p}_2}$ iff:

$(\forall \vec{q} \in S)\ (Y_{\vec{p}_1 - \vec{q}} = Y_{\vec{p}_2 - \vec{q}})$.

The 'stacking property' for an RNO is defined by Wendt, and we write it as follows:

*Definition 5.7 Stacking property (*Wendt et al. 1986*)*

An RNO has the stacking property, if when the RNO is applied to a neighbourhood $N_{\vec{p}_1}$ which is higher than an other neighbourhood $N_{\vec{p}_2}$ (with the same S), the result of neighbourhood $N_{\vec{p}_1}$ is higher than or equal to the result of neighbourhood $N_{\vec{p}_2}$, i.e.:

$$(\forall\vec{q}\in S)\ (Y_{\vec{p}_1-\vec{q}}\geq Y_{\vec{p}_2-\vec{q}})\Rightarrow f_{\vec{q}\in S}(Y_{\vec{p}_1-\vec{q}},X_{\vec{p}_1-\vec{q}},\vec{q})\geq f_{\vec{q}\in S}(Y_{\vec{p}_2-\vec{q}},X_{\vec{p}_2-\vec{q}},\vec{q})$$
$$\Leftrightarrow (N_{\vec{p}_1}\geq N_{\vec{p}_2})\Rightarrow (f(N_{\vec{p}_1},\ldots)\geq f(N_{\vec{p}_2},\ldots)) \tag{5.36}$$

We will show in the next theorem that having the stacking property in addition to the monotonically increasing (decreasing) property is sufficient to guarantee that an RNO has a unique fixed point (as defined in Section 5.4.1).

Theorem 5.11 Unique fixed point

An RNO which is monotonically increasing (decreasing) with an upper (lower) bound, and which has the stacking property, has a unique fixed point with respect to deterministic updating methods.

Proof 5.11 Unique fixed point

Because the RNO is monotonically increasing with an upper bound, we know from Theorem 5.10 that the RNO has a fixed point. It remains to be shown that RNOs which also fulfil the stacking property have a unique (i.e. updating method independent) fixed point. This will be proven by showing that the image resulting from an iteration of an arbitrary deterministic updating method (see Definition 5.3) is bounded above and below by iterations of the simultaneous updating method (see Definition 5.4). Images resulting from the arbitrary deterministic updating method will be denoted by $\overline{Y}$, *and images created with simultaneous updating by* Y.

(1) First we will show that the image which results from an iteration of an arbitrary deterministic updating method is always higher than or equal to an image which results from the same iteration using simultaneous updating.

By definition the initialisation of the simultaneous updating equals the initialisation of the arbitrary deterministic updating: $(\forall\vec{p}\in \mathrm{Im})\ Y_{\vec{p}}^{(0)}=\overline{Y}_{\vec{p}}^{(0)}$.

In calculating the image values for the first pass using both the methods, simultaneous updating uses neighbourhood values $N_{\vec{p}}^{(1)}=\left\{Y_{\vec{p}-\vec{q}}^{(0)}\middle|\ (\vec{q}\in S)\right\}$, *whereas the arbitrary deterministic updating method uses* $\overline{N}_{\vec{p}}^{(1)}=\left\{\overline{Y}_{\vec{p}-\vec{q}_1}^{(1)},\overline{Y}_{\vec{p}-\vec{q}_2}^{(0)}\middle|\ (\vec{q}_1\in S_1,\vec{q}_2\in S_2)\right\}$.

Because of the monotonically increasing property one can see that all over the image the neighbourhood points for the arbitrary deterministic updating method are larger than or equal to the neighbourhood points of the simultaneous updating:

$$(\forall\vec{p}\in \mathrm{Im})\left((\forall\vec{q}_{1,2}\in S_{1,2})\left(\overline{Y}_{\vec{p}-\vec{q}_1}^{(1)}\geq Y_{\vec{p}-\vec{q}_1}^{(0)},\ \overline{Y}_{\vec{p}-\vec{q}_2}^{(0)}=Y_{\vec{p}-\vec{q}_2}^{(0)}\right)\right)\Leftrightarrow$$
$$(\forall\vec{p}\in \mathrm{Im})\left(\overline{N}_{\vec{p}}^{(1)}\geq N_{\vec{p}}^{(1)}\right). \tag{i}$$

Using the stacking property on (i), the following is valid:

$$(\forall\vec{p}\in \mathrm{Im})\left(f(\overline{N}_{\vec{p}}^{(1)},\ldots)\geq f(N_{\vec{p}}^{(1)},\ldots)\right)\Leftrightarrow(\forall\vec{p}\in \mathrm{Im})\left(\overline{Y}_{\vec{p}}^{(1)}\geq Y_{\vec{p}}^{(1)}\right).$$

This can be repeated for other passes, so that: $(\forall\vec{p}\in \mathrm{Im})\left(\overline{Y}_{\vec{p}}^{(k)}\geq Y_{\vec{p}}^{(k)}\right)$, *which proves (1).*

(2) Second we show that for every image that results from an iteration of an arbitrary deterministic updating method a 'larger' image can be found which results from an iteration of the simultaneous updating.
For both updating methods, we start with the same initial image again:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}}^{(0)} = \overline{Y}_{\vec{p}}^{(0)}.$$

Now for both updating methods all points with $I(\vec{p}) = 0$ *are calculated. With the simultaneous updating* <u>*all*</u> *points have rank value 0, but for the arbitrary deterministic updating method only a* <u>*subset*</u> *of points has rank value 0. Those points which have rank value 0 in both methods receive the same value in the first calculation pass:*

$$\{\forall \vec{p} | I(\vec{p}) = 0\}\, Y_{\vec{p}}^{(1)} = \overline{Y}_{\vec{p}}^{(1)}$$

Next, the points with rank value 1 in the arbitrary deterministic updating method are calculated. The neighbourhoods $\overline{N}_{\vec{p}}^{(1)}$ *of the points* $\vec{p}$ *with* $I(\vec{p}) = 1$ *may consist of a mixture of* $\left\{\overline{Y}_{\vec{p}-\vec{q}_i}^{(0)}, \overline{Y}_{\vec{p}-\vec{q}_j}^{(1)}\right\}$, *and through the equivalencies we know that these are equal to:*

$$\{\forall \vec{p} | I(\vec{p}) = 1\}\, \overline{N}_{\vec{p}}^{(1)} = \left\{\overline{Y}_{\vec{p}-\vec{q}_i}^{(0)}, \overline{Y}_{\vec{p}-\vec{q}_j}^{(1)} \middle| \, (I(\vec{p}-\vec{q}_i) \geq I(\vec{p}), I(\vec{p}-\vec{q}_j) < I(\vec{p}))\right\}$$

$$= \left\{Y_{\vec{p}-\vec{q}_i}^{(0)}, Y_{\vec{p}-\vec{q}_j}^{(1)}\right\} \qquad (ii)$$

We also know that for the simultaneous updating method pass 2 only depends on pass 1:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}}^{(2)} = f_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}^{(1)}, X_{\vec{p}-\vec{q}}, \vec{q}) = f(N_{\vec{p}}^{(2)}, \ldots), \text{ with:}$$

$$N_{\vec{p}}^{(2)} = \left\{Y_{\vec{p}-\vec{q}}^{(1)} \middle| \, (\vec{q} \in S)\right\} \qquad (iii)$$

Because of the monotonically increasing property we know from (ii) and (iii) that:

$$\{\forall \vec{p} | I(\vec{p}) = 1\} \left(N_{\vec{p}}^{(2)} \geq \overline{N}_{\vec{p}}^{(1)}\right)$$

The stacking property guarantees that: $\{\forall \vec{p} | I(\vec{p}) = 1\} \left(f(N_{\vec{p}}^{(2)}, \ldots) \geq f(\overline{N}_{\vec{p}}^{(1)}, \ldots)\right) \Leftrightarrow$

$$\{\forall \vec{p} | I(\vec{p}) = 1\}\, Y_{\vec{p}}^{(2)} \geq \overline{Y}_{\vec{p}}^{(1)}.$$

For the arbitrary deterministic updating the neighbourhoods $\overline{N}_{\vec{p}}^{(1)}$ *of the points* $\vec{p}$ *with* $I(\vec{p}) = 2$ *may consist of a mixture of* $\left\{\overline{Y}_{\vec{p}-\vec{q}_i}^{(0)}, \overline{Y}_{\vec{p}-\vec{q}_j}^{(1)}\right\}$, *and through the equivalencies we know that these are equal to:*

$$\{\forall\vec{p}|\,I(\vec{p})=2\}\;\bar{N}_{\vec{p}}^{(1)} = \left\{\bar{Y}_{\vec{p}-\vec{q}_i}^{(0)}, \bar{Y}_{\vec{p}-\vec{q}_j}^{(1)}\,\middle|\,(I(\vec{p}-\vec{q}_i)\geq I(\vec{p}), I(\vec{p}-\vec{q}_j)<I(\vec{p}))\right\}$$

$$= \left\{Y_{\vec{p}-\vec{q}_i}^{(0)}, Y_{\vec{p}-\vec{q}_j}^{(1)}, \bar{Y}_{\vec{p}-\vec{q}_m}^{(1)}\,\middle|\,(I(\vec{p}-\vec{q}_i)\geq 2, I(\vec{p}-\vec{q}_j)=0, I(\vec{p}-\vec{q}_m)=1)\right\} \qquad (iv)$$

For $\bar{N}_{\vec{p}}^{(1)}$ *we consider an alternative neighbourhood* $\tilde{N}_{\vec{p}}^{(1)}$*, such that:*

$$\{\forall\vec{p}|\,I(\vec{p})=2\}\;\tilde{N}_{\vec{p}}^{(1)} = \left\{Y_{\vec{p}-\vec{q}_i}^{(0)}, Y_{\vec{p}-\vec{q}_j}^{(1)}, Y_{\vec{p}-\vec{q}_m}^{(2)}\right\} \qquad (v)$$

Also consider a neighbourhood for the simultaneous updating in pass three:

$$N_{\vec{p}}^{(3)} = \left\{Y_{\vec{p}-\vec{q}}^{(2)}\,\middle|\,(\vec{q}\in S)\right\} \qquad (vi)$$

Using (iv), (v) and (vi), the following relationship exists between the neighbourhoods: $\{\forall\vec{p}|\,I(\vec{p})=2\}\;N_{\vec{p}}^{(3)}\geq\tilde{N}_{\vec{p}}^{(1)}\geq\bar{N}_{\vec{p}}^{(1)}$*, which combined with the stacking property guarantees that:* $\{\forall\vec{p}|\,I(\vec{p})=2\}\left(f(N_{\vec{p}}^{(3)},\ldots)\geq f(\tilde{N}_{\vec{p}}^{(1)},\ldots)\geq f(\bar{N}_{\vec{p}}^{(1)},\ldots)\right)\Leftrightarrow$

$$\{\forall\vec{p}|\,I(\vec{p})=2\}\;Y_{\vec{p}}^{(3)}\geq\bar{Y}_{\vec{p}}^{(1)} \qquad (vii)$$

Generalizing the steps which led to (iii) and (vii) we obtain:

$(\forall\vec{p}\in \mathrm{Im})\left(Y_{\vec{p}}^{(1+I(\vec{p}))}\geq\bar{Y}_{\vec{p}}^{(1)}\right)$.

In fact, for further iterations of the generalized deterministic updating method we find that:

$(\forall\vec{p}\in \mathrm{Im})\left(Y_{\vec{p}}^{(k(1+I(\vec{p})))}\geq\bar{Y}_{\vec{p}}^{(k)}\right)$*, which proves (2).*

Because of Theorem 5.10 we know that in the limit case:

$(\forall\vec{p}\in \mathrm{Im})\left(\lim_{k\to\infty}\left(Y_{\vec{p}}^{(k+1)}-Y_{\vec{p}}^{(k)}\right)=0\right)$*, which implies that also:*

$(\forall\vec{p}\in \mathrm{Im})\left(\lim_{k\to\infty}\left(Y_{\vec{p}}^{(k(1+I(\vec{p})))}-Y_{\vec{p}}^{(k)}\right)=0\right)$,

so that using (1) and (2): $(\forall\vec{p}\in \mathrm{Im})\left(\lim_{k\to\infty}\bar{Y}_{\vec{p}}^{(k)}=\lim_{k\to\infty}Y_{\vec{p}}^{(k)}\right)$

The implication of Theorem 5.11 for binary images is quite interesting. According to Theorem 5.11, a binary RNO has a unique fixed point, if:

- It fulfils the stacking property; Wendt showed that this means that the RNO can be written as a 'positive Boolean function' (Wendt et al. 1986). A positive Boolean function can be written as a sum-of-products (where summation is Boolean *or*ring and multiplication is Boolean *and*ing) without any negations, and either
  - Transitions from $Y_{\vec{p}}^{(k)}$ to $Y_{\vec{p}}^{(k+1)}$ can only be 0->0, 0->1 or 1->1 and not 1->0, (i.e. monotonically increasing), or the reverse,

- Transitions from $Y_{\grave{p}}^{(k)}$ to $Y_{\grave{p}}^{(k+1)}$ can only be 1->1, 1->0 or 0->0 and not 0->1 (i.e. monotonically decreasing).

Another observation is, that RNOs which can mathematically be rewritten as an Object or Global Operation necessarily have to be stable in the fixed point sense. As the OO or GlO are unique mappings, so will their mathematical equivalencies also have to be unique and stable in the fixed point sense. This observation is formalised in Theorem 5.12:

Theorem 5.12 Fixed point related to OO or GlO
All RNOs which can - independent of the updating method - mathematically be rewritten as an OO or GlO, are unique and stable in the fixed point sense.

Proof 5.12 Fixed point related to OO or GlO
Any OO or GlO is a unique mapping with exactly one solution, so that a mathematically equivalent RNO is also unique and stable in the fixed point sense.

□

Given that cyclicity has been defined for an RNO in Section 5.4.2, it is possible to show that non-cyclical RNOs have but one fixed point to which they will converge.

Theorem 5.13 Fixed point related to cyclicity
All non-cyclical RNOs have exactly one fixed point.

Proof 5.13 Fixed point related to cyclicity
From the definition of a non-cyclical RNO it is clear that a whole image can be calculated in such a way that only values which are already valid (see Section 5.2.1) are used to determine the value of any image point[1]*. If all points of the destination image are calculated once in such a way, a new calculation of destination points will never change anything in the destination image.*

□

The latter theorem can be understood by realising that destination points which are calculated for the non-cyclical RNO only depend on points in the input image and on destination points which have already been calculated (i.e. not on 'invalid' destination points).

The three theorems concerning a unique fixed point (i.e. Theorem 5.11, Theorem 5.12 and Theorem 5.13) can be combined into one theorem concerning non-unique or unstable RNOs in the fixed point sense, by using the rule that $A + B + C \rightarrow D$ is logically equivalent to $\neg D \rightarrow \neg(A + B + C)$. This leads to Theorem 5.14.

Theorem 5.14 Fixed point combination
All RNOs which have zero or more than one fixed point at least fulfil the following conditions simultaneously:
(1) they are cyclical,
(2) they are either (2.1) non-monotonically, or (2.2) for increasing (decreasing) RNOs they do not have an upper (lower) bound, or (2.3) do not posses the stacking property,
(3) They can not be written as an OO or GlO.

We will give a few examples to illustrate that the conditions given in Theorem 5.10 until Theorem 5.13 are uncorrelated. First with respect to Theorem 5.10 the discrete Poisson RNO is not monotonically increasing or decreasing but does have a fixed point image (see Theorem 5.1). Concerning Theorem 5.11 we know that the SERP does not have the stacking property, yet does give a unique fixed point (see Section 5.4.4). Third, with respect to

1. i.e. the precedence graph (see Section 5.3.2) does not have loops.

Theorem 5.12, we have not succeeded in writing the maximum median root as a global or object operation, yet it does give a unique fixed point (see Theorem 5.15). Last, with respect to Theorem 5.13 the distance transform is a cyclical RNO that gives a unique fixed point (see Section 5.4.4).

The fixed point theory is not yet finished with this paragraph. The conditions formulated in Theorem 5.10 until Theorem 5.13 are sufficient for RNOs to have one (or more) fixed point(s). However, we do not know if these conditions cover *all* RNOs.

5.4.4 The number of fixed points for some operations

The theorems which were derived in the previous paragraph will now be demonstrated by some of the RNOs which were treated previously.

First note that the SERP has a unique fixed point, because it fulfils the conditions of Theorem 5.12, i.e. it has been proven in Theorem 5.8 that the SERP can be written as an OO.

Equivalently, the object selection, the distance transform and the grey weighted distance transform can be written as OOs (proven in Section 5.2.3), so that according to Theorem 5.12 they also have unique fixed points.

As another example we take the maximum median root defined later in Section 5.5.5, Equation (5.52).

Theorem 5.15 Maximum median root

The maximum median root as defined in Equation (5.52) has one unique fixed point.

Proof 5.15 Maximum median root

We will show that the maximum median root fulfils the three conditions of Theorem 5.11, i.e.: (1) monotonically increasing, (2) upper bound, (3) stacking property.

(1) Because of the definition for the maximum median root (see Equation (5.52)):

$$(\forall \vec{p} \in \mathrm{Im})\ Y_{\vec{p}}^{(k+1)} = \max(Y_{\vec{p}}^{(k)}, \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}^{(k)})),$$

we know that if $\mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}^{(k)}) < Y_{\vec{p}}^{(k)}$*, then* $Y_{\vec{p}}^{(k+1)} = Y_{\vec{p}}^{(k)}$*, and only if:*

$$\left(\mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}^{(k)}) > Y_{\vec{p}}^{(k)} \right) \Rightarrow \left(Y_{\vec{p}}^{(k+1)} = \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}^{(k)}) \right), \textit{ so that:}$$

$$(\forall k) \left(Y_{\vec{p}}^{(k+1)} \geq Y_{\vec{p}}^{(k)} \right)$$

(2) The maximum median has an upper bound because it becomes either equal to the old value, or equal to the median of the neighbourhood. The latter value is always lower than or equal to the largest value in the image. Therefore:

$$(\forall \vec{p} \in \mathrm{Im}) \left(Y_{\vec{p}}^{(k)} \leq \max_{\vec{w} \in \mathrm{Im}} Y_{\vec{w}}^{(0)} \right).$$

(3) Consider two neighbourhoods, such that: $(\forall \vec{q} \in S)\ (Y_{\vec{p}_1 - \vec{q}} \geq Y_{\vec{p}_2 - \vec{q}})$.

In the literature it is proved that (Wendt et al. 1986):

$$(\forall \vec{q} \in S)\ (\mathrm{med}_{\vec{q} \in S}((Y_{\vec{p}_1 - \vec{q}}) \geq \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}_2 - \vec{q}}))),$$

so that also: $(\forall \vec{q} \in S)\ (\max(Y_{\vec{p}_1}, \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}_1 - \vec{q}})) \geq \max(Y_{\vec{p}_2}, \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}_2 - \vec{q}})))$.

From the examples above it is clear, that the theorems which were derived are enough to conclude whether these type of RNOs will have one unique fixed point. We showed that the SERP, the distance transform, the object selection, the GWDT and the maximum median root have unique fixed points. The proves can be generalized to similar RNOs.

5.5 Classes of RNOs and object operations

The RNOs can be divided into classes, just like LNOs and other image processing operations. The aim of classifying RNOs is to find groups of RNOs with common properties which allow predictions to be made about their behaviour. Specifically, now that predictions can be made about the fixed point behaviour of individual RNOs, groups may be found which are stable, unstable, unique or non-unique in the fixed point sense.

5.5.1 Linear versus non-linear RNOs

The class of linear RNOs is only mentioned here for the reason of comparison and completeness. It will not be treated in-depth. One reason is that many authors have published results or are still working on this subject. It is felt that linear RNOs fall outside the scope of this thesis.

However, a study of some techniques used for linear RNOs can help us in developing techniques specifically for non-linear RNO-classes.

Definition 5.8 An RNO is linear if the value Y at position $\vec{p}$ is a linear combination of the values Y at positions $\vec{p}-\vec{q}$ (with $\vec{q}$ belonging to the neighbourhood S) and of the value of image X at position $\vec{p}-\vec{q}$.

A general linear RNO can thus be written as follows:

$$(\forall \vec{p} \in \mathrm{Im})\; Y_{\vec{p}} = \sum_{\vec{q} \in S} (X_{\vec{p}-\vec{q}} \cdot a_{\vec{q}}) + \sum_{\vec{q} \in S} (Y_{\vec{p}-\vec{q}} \cdot b_{\vec{q}}) \tag{5.37}$$

with: $a_{\vec{q}}$ and $b_{\vec{q}}$ are the coefficients which determine exactly what linear combination is taken.

A lot of special techniques have been developed for the investigation of non-cyclical linear RNOs and their stability in the BIBO sense (Dudgeon and Mersereau 1984). Cyclical linear RNOs arise from partial differential equations which are rewritten as difference equations. One example is the Poisson equation for a potential field V (Lorrain and Corson 1970):

$$\left(\nabla^2 V = -\frac{\rho}{\varepsilon_0}\right) \Leftrightarrow \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = -\frac{\rho}{\varepsilon_0} \quad \text{(for three dimensions)} \tag{5.38}$$

The Poisson equation can be rewritten as a difference equation, by approximating the second partial derivative as follows:

$$\frac{\partial^2 V}{\partial x^2} \cong V_{x-\delta x, y, z} - (2 \cdot V_{x,y,z}) + V_{x+\delta x, y, z}$$

This approximation in two dimensions leads to the following RNO for the potential $V(x,y)$:

$$V_{x,y} = \frac{1}{4} \cdot \left\{ V_{x-\delta x, y} + V_{x+\delta x, y} + V_{x, y-\delta y} + V_{x, y+\delta y} + \frac{\rho}{\varepsilon_0} \right\} \Leftrightarrow$$

$$V_{\vec{p}} = \frac{1}{4} \cdot \left\{ \frac{\rho_{\vec{p}}}{\varepsilon_0} + \sum_{\vec{q} \in S} V_{\vec{p}-\vec{q}} \right\} \tag{5.39}$$

The last formula uses our own notation for the discretisized Poisson equation in two di-

mensions. Note, that the charge density ρ could also be a function of the position $\vec{p}$. In Section 5.2.2 it has been shown that an inverse convolution like the discretisized Poisson equation can be rewritten as a global operation.

5.5.2 Cyclical versus non-cyclical RNOs

Another way to split the RNOs in two groups is by looking at their cyclicity, as defined in Section 5.4.2, Definition 5.2. Non-cyclical RNOs can be calculated in one pass over the image if the correct (deterministic) updating method is used. Linear non-cyclical RNOs are called 'fully recursive' in the literature (Woods and Lee 1983). The term 'fully recursive' does not indicate the use of a full recursive 3*3 neighbourhood, but indicates the use of a neighbourhood which is *almost* cyclical (i.e. adding one extra neighbour from the 3*3 environment to the neighbourhood point set S results in a cyclical neighbourhood).

The cyclical RNOs have to be calculated by iteration if a deterministic updating method is used. Some cyclical RNOs can be calculated in one pass over the image with the data dependent bucket updating. Cyclical RNOs may have any number of fixed points.

5.5.3 Morphological RNOs

Morphological operations are described extensively in the literature (Serra 1982; Sternberg 1986; Haralick et al. 1987; Maragos and Schafer 1987b; Maragos and Schafer 1987c). All morphological operations can in a way be thought of as a combination of erosions and/or dilations (Maragos 1987a). Recursive morphological operations are not known as such. However, some iterative morphological operations exist. One of them is the morphological skeleton, which is closely related to the medial axis transform (Maragos and Schafer 1986).

The definition of the morphological skeleton as object operation is in our notation:

$$Y_{\vec{p}} = \sum_{n=0}^{N} \left\{ X_{\vec{p}} \Theta S^n - (X_{\vec{p}} \Theta S^n)_S \right\} \tag{5.40}$$

with: Θ is erosion, S^n is the erosion of S with itself n times, $(...)_S$ is the opening of (...) by the set S, the '-' sign indicates Boolean *exclusive or*, and the summation Σ denotes Boolean *or*ring.

Maragos and Schafer show, that the morphological skeleton can also be calculated in an iterative way. Their algorithm leads to the construction of the following RNO:

$$(\forall \vec{p} \in \mathrm{Im}) \begin{cases} Z_{\vec{p}} = \left\{ Y_{\vec{p}} - (Y_{\vec{p}})_S \right\} + Z_{\vec{p}} \\ Y_{\vec{p}} = Y_{\vec{p}} \Theta S \end{cases} \tag{5.41}$$

with: $Y_{\vec{p}}^{(0)} = X_{\vec{p}}$, and $Z_{\vec{p}}^{(0)} = 0$

Why are morphological RNOs of interest? Because they can be built up as a Boolean *or*ring of hit-or-miss transforms (Maragos and Schafer 1987b), in the same way as linear operations are built up from an addition of multiplications. This makes them interesting from a mathematical viewpoint and also from an implementation viewpoint.

5.5.4 Recursive order statistics filters

The order statistics (OS) filters or rank order (RO) filters for grey value images are the same as threshold logic filters for binary images (Maragos and Schafer 1987c; Maragos 1989). Related to these filters is the Ξ-filter (Preston 1983). A two dimensional grey value image can be seen as a three dimensional logic space, where x and y are the spatial coordinates, and z is the pixel value coordinate. The operation of the Ξ-filters can be defined as taking the threshold of a convolution in one, two or three of the x,y,z dimensions (Preston 1983). Such an operation may be iterated a number of times. One iteration of a threshold function in the (x,y) plane can be denoted in Preston's notation as $\Xi_{xy}(T)^1$, and is calculated as follows (our notation):

$$\xi_L(x,y) = Y_{\vec{p}} = \begin{cases} \sum_{\vec{q} \in S} X_{\vec{p}-\vec{q}} > T & \rightarrow 1 \\ \sum_{\vec{q} \in S} X_{\vec{p}-\vec{q}} \leq T & \rightarrow 0 \end{cases} \tag{5.42}$$

with: X and Y binary images.

Because the convolution is calculated without weight factors, the Ξ-filters are called unweighted threshold logic filters. Weighted threshold logic filters for binary images have also been introduced (Wendt et al. 1986) and can be written in our notation as:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \begin{cases} \sum_{\vec{q} \in S} X_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} > T & \rightarrow 1 \\ \sum_{\vec{q} \in S} X_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} \leq T & \rightarrow 0 \end{cases} \tag{5.43}$$

with: X and Y binary images.

The equivalent for grey value images is the weighted rank order filter, where the weighting coefficients denote the number of times that a neighbourhood point is counted in the ranking (Wendt et al. 1986):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \mathrm{Rank}^{T}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}, c_{\vec{q}}) \tag{5.44}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$, Y and X grey value images,

$(Y_{\vec{p}-\vec{q}}, c_{\vec{q}})$ means: count $Y_{\vec{p}-\vec{q}}$ for $c_{\vec{q}}$ times in the ranking operation,

T is the ranking threshold.

At this point two properties of the threshold logic filters can be seen.

- *Convolution property.* Performing the convolution over the whole image and thresholding the resulting grey value image yields the same result as doing the convolution and thresholding per pixel (Maragos and Schafer 1987c).
- *Commuting with thresholding.* Performing a weighted threshold logic function in the *(x,y)* plane on threshold level i of a grey value image X yields the same result as convolving the original grey value image X with the same grey value kernel c_S, and taking the ith threshold level of it (Maragos and Schafer 1987c). In other words: the threshold logic filters commute with thresholding.

  Equivalent to commutation with thresholding for binary images is the *threshold decomposition* property for grey value images. Wendt shows that weighted rank order filters can be computed using threshold decomposition: threshold the image at grey

value levels, perform the weighted threshold logic operations (the equivalence of weighted rank ordering for binary images) on these levels, and add the resulting bit planes.

The relation between threshold logic filters and morphological filters lies in the fact that the first can be written as a union of erosions or an intersection of dilations (Maragos and Schafer 1987c). Van den Boomgaard proves that the weighted threshold logic filter can be written as the union of hit-or-miss transforms (van den Boomgaard 1989).

Recursive (weighted) threshold logic filters have been reported by Wilson, and are defined by us in the same way as their non recursive LNO equivalents (Wilson 1989a):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \begin{cases} \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} > T & \rightarrow 1 \\ \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} \cdot c_{\vec{q}} \leq T & \rightarrow 0 \end{cases} \tag{5.45}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$, and Y and X binary images.

With respect to fixed point stability, rank order functions are of interest. From the definition of the recursive rank order filters and the fixed point theory it can be seen, that these filters may give unstable or non-unique results in the fixed point sense (i.e. they are not necessarily monotonically increasing/decreasing). Rank order functions with positive weights can be slightly adapted to yield monotonically increasing RNOs with a well defined maximum value, i.e. RNOs which are stable and unique in the fixed point sense. Such a stable rank order function is defined as:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \max(Y_{\vec{p}}, \mathrm{Rank}^{T}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}, c_{\vec{q}})) \tag{5.46}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$, and Y and X binary images.

Examples of the use of recursive rank order filters will be given in Chapter 6.

5.5.5 Recursive stack filters

The class of stack filters has been defined by Wendt as the class of all filters that are defined for a finite window (i.e. the neighbourhood point set S is finite), and commute with thresholding (Wendt et al. 1986). If an operation $f(...)$ on a grey value image X commutes with thresholding, than the following equality holds:

$$(\forall \vec{p} \in \mathrm{Im})\ (Y_{\vec{p}})_t = f_{\vec{q} \in S}(X_{\vec{p}-\vec{q}}, \vec{q})_t = f_{\vec{q} \in S}((X_{\vec{p}-\vec{q}})_t, \vec{q}) \tag{5.47}$$

Performing the function f on the grey value neighbourhood of X_p and taking the threshold at t (indicated by the subscribed t) yields the same result as performing the function f on the thresholded version of the same neighbourhood.

Wendt shows, that stack filters always have the stacking property as defined in Definition 5.7. Stable and unique RNOs also need to be monotonically increasing (decreasing) according to Theorem 5.11. Stack filters fulfilling this 'unique fixed point' theorem can therefore be constructed the following way. Suppose that a normal stack filter is denoted by $\sigma(...)$, so that any stack filter RNO can be written as:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \sigma_{\vec{q} \in S}(X_{\vec{p}-\vec{q}}) \tag{5.48}$$

The corresponding unique and stable *increasing* recursive stack filter can then be writ-

ten as:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \max(Y_{\vec{p}}, \sigma_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}})) \tag{5.49}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$

The corresponding unique and stable *decreasing* recursive stack filter can be written as follows:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \min(Y_{\vec{p}}, \sigma_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}})) \tag{5.50}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$

These adapted recursive stack filters fulfil Theorem 5.11, so that they always give one single fixed point image, independent of the updating method used. The fact that they commute with thresholding means, that the behaviour of the binary filters can be translated to the behaviour of the grey-value filters. This can be compared with the superposition principle for linear RNOs. The class of recursive stack filters is therefore a very interesting class.

This section is concluded by showing an example of a non-unique stack filter which is transformed into two different stable stack filters. As an example of the stack filter we take the median root which is defined as follows (derived from Tyan 1981):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \operatorname*{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}}) \tag{5.51}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$, and the images X and Y may be binary or grey-value, and all sorts of updating methods are used, yielding (slightly) different results. The edge of the image is set to the constant value of zero.

From the median root, the minimum median root is defined as:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \min(Y_{\vec{p}}, \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}})) \tag{5.52}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$

The maximum median root is then defined as (compare Section 5.4.4):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \max(Y_{\vec{p}}, \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}})) \tag{5.53}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$

The qualitative results of the minimum and the maximum median root will be demonstrated in Chapter 6.

5.5.6 Relation between non-linear RNO classes

Dividing the RNOs into classes serves the investigation towards stable (and possible unique) RNOs. Linear RNOs can be built up from a linear combination of neighbourhood pixel values. A non-linear equivalent is found in morphologic operations. These can be built from a combination of erosions or dilations. Among the morphologic filters in general, the stack filters are interesting because they commute with thresholding. Hence, grey value stack filters can be build from binary stack filters.

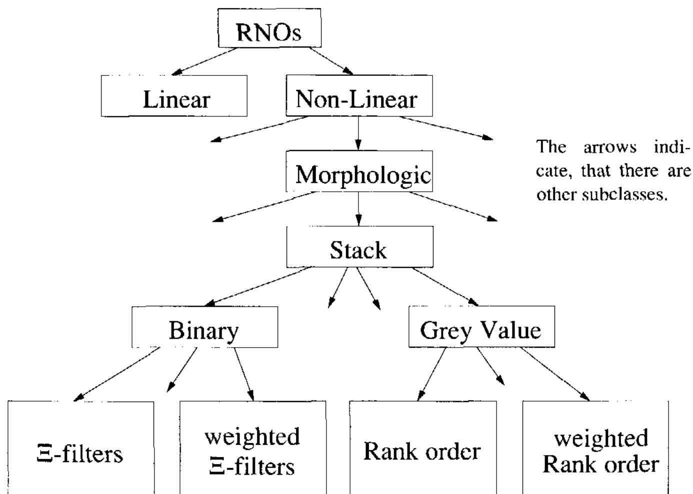


Figure 5.14 Relation between some classes of non-linear RNOs

Not all recursive stack filters are stable in the fixed point sense (the median root, for example, is unstable for some combinations of a neighbourhood point set S and; see Goles and Olivos 1981). There is a simple way to derive unique (and stable) stack filters (derived in Section 5.5.5). Examples from the classes of RNOs will be given in Chapter 6. The relation between stack filters, and some other classes of non-linear filters is given in Figure 5.14.

5.6 Conclusions

After motivating the use of RNOs by showing some operations for whose calculation they can be used, a mathematical notations for low-level image processing operations was introduced. Using this notation, it was shown how some object and global operations can be derived from their equivalent RNOs. The operations we investigate have to be calculated using some updating method. We defined and explored three types of updating methods: deterministic, data dependent and stochastic. For some limited class of RNOs, data dependent RNOs might be the most efficient. This will be further explored in Chapters 7 and 8. The stability and uniqueness of RNOs in the fixed point sense has been defined. Only stable (and possibly unique) RNOs in the fixed point sense are of interest in image processing. We explored some non-linear classes of RNOs with respect to their usefulness. A class of non-linear RNOs called recursive stack filters is shown which has the following qualities:

- *Stability*. Recursive stack filters do not necessarily have to be stable in the fixed point sense.
- *Unique*. Recursive stack filters can be reformed so that they have exactly one unique fixed point or root image.
- *Threshold commuting*. The fact, that recursive stack filters commute with thresholding, means that they can be calculated efficiently on the binary thresholds of the grey value image using cellular logic architectures.

- *Stackable.* The stacking property of these filters assures that the significance ranking order of the binary thresholds of the grey value image remains the same after applying the RNO.

An example of a recursive stack filter is the median root (Wendt et al. 1986). It is shown, that this filter can be made unique in the fixed point sense, with some slight adaptation.

In Chapter 6 some examples will be given of RNOs and their applicability in image processing. This is followed in Chapter 7 by a thorough investigation towards the implementation possibilities for updating methods on the three architecture groups SPA, LPA and PL which are being compared in this thesis. The results of some experiments will then be reported to show the efficiency of the updating methods for some specific RNOs in Chapter 8.

Chapter

6 The use of RNOs for image processing

Some important theoretical aspects of the recursive neighbourhood operations (RNOs) have been dealt with in Chapter 5. This chapter will use the notation, definitions and theorems from the previous chapter to explain the use of some non-linear RNOs for image processing purposes. The main goal of this chapter is to show the relation between the theory of RNOs and their usage in image processing.

One linear RNO, the 'model problem', will be used in Section 6.1 as a reference point to readers with a background in linear multi-dimensional signal processing operations. The rest of the examples are devoted to non-linear RNOs. As an example of morphological RNOs, the working of the morphological skeleton is illustrated in Section 6.2. The class of recursive stack filters is illustrated by the recursive median type filters in Section 6.3. This paragraph also illustrates the fact, that the recursive median is non-unique in the fixed point sense. A rough approximation to the convex hull, the smallest enclosing regular polygon, is discussed in Section 6.4. Several types of distance transforms are discussed in Section 6.5. Their usage is discussed and illustrated for the constrained distance transform. Other groups of non-linear RNOs, such as the dithering algorithms which can be used to convert grey-value images to binary images, are discussed in Section 6.6. Finally conclusions about the use of RNOs for image processing are discussed in Section 6.7.

6.1 The model problem

A common example of a linear RNO is the 'model-problem' (the discrete Poisson equation) which will be taken as an example here (Hockney and Jesshope 1981). The aim of the model problem is to retrieve a Laplace-filtered image. However, an image which has been filtered by a Laplacian is a second derivative of that image. Retrieving an original from a second derivative requires the knowledge of edge conditions. These are not known in general. The model problem is widely discussed in the literature, and it is well known, that the SOR updating technique is most suited to it (Leveque and Trefethen 1988). There are also direct methods which can be used to solve the discrete Poisson equation. These are usually one order in magnitude faster than the RNO solutions (Hockney and Jesshope 1981).

Recall the formula for the model problem as derived in Section 5.5.1, Equation (5.42):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \frac{1}{4} \cdot \left\{ -X_{\vec{p}} + \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} \right\}$$

with: $Y_{\vec{p}}^{(0)} = 0$,

the edge is mirrored (see Section 5.1.4),

Y = the potential field V, and $X_{\vec{p}}$ = the position dependent $-\rho_{\vec{p}}/\varepsilon_0$.

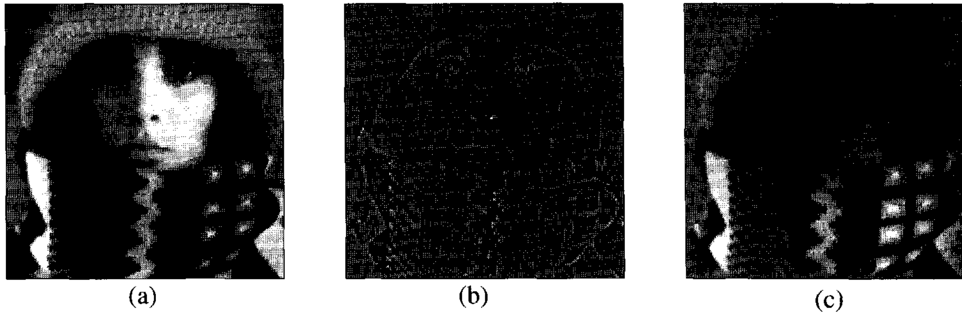


*Figure 6.1 Demonstration of the model problem for an 128*128 image: (a) original, (b) original filtered with a Laplacian, (c) approximate recovery with the inverse Laplacian.*

The performance of the inverse Laplacian is shown qualitatively in Figure 6.1. All three images have been 'contrast stretched' (i.e. in all three images the values 0 and 255 are found) after calculations, to improve the printing quality. Because the image in Figure 6.1c is constructed from Figure 6.1b without using knowledge of edge conditions, the recovery lacks the offset and shading component.

6.2 Recursive morphological RNOs

Morphological operations (see Section 5.5.3) have been described mathematically in the literature (Serra 1982; Haralick et al. 1987). The usage of morphological RNOs can be demonstrated by the morphological skeleton (defined in Equation (5.43); Maragos and Schafer 1986). We refer to their work for the qualitative analysis of this operation, which is closely related to the medial axis transform.

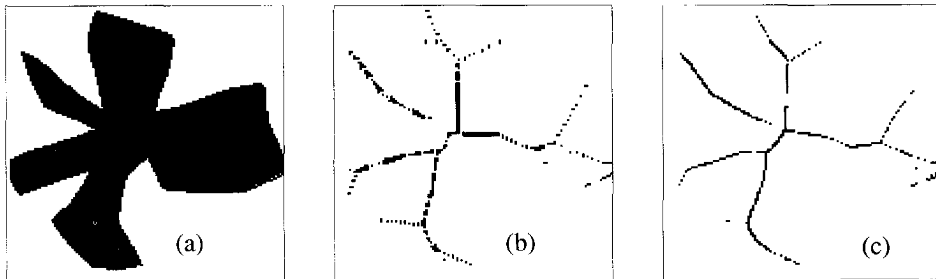


Figure 6.2 Illustration of a morphological RNO: (a) original, (b) the morphological skeleton operation, (c) the medial axis transform (for comparison).

An example of the morphological skeleton is given in Figure 6.2. Both the morphological skeleton as well as the medial axis transform were done using a neighbourhood consisting of all eight nearest neighbours. The effects of using other neighbourhood sizes and shapes is extensively discussed by Maragos and Schafer.

6.3 Recursive stack filters

As was discussed in Chapter 5, many local neighbourhood image processing operations can be categorized as stack filters (Wendt et al. 1986). Such filters commute with

thresholding, and do not disturb the ordering of grey levels. As shown in Section 5.5.5, recursive stack filters may be non-unique in the fixed point sense. Moreover, not all stack filters are stable in the fixed point sense. A simple rewriting of a stack filter, however, allows the implementation of a monotonically increasing or decreasing unique and stable version if required (as explained in Section 5.5.5).

6.3.1 Recursive median

Recursive median filters can be used for the shot noise reduction in images without loss of edge information (Arce and Crinon 1984). Another application stems from the observation that a recursively applied median filter on a binary image removes objects which are smaller in size and shape than a so called Smallest Surviving Object (Döhler 1989). The smallest surviving objects can be constructed from the vectors which span the set of neighbourhood points S.

Several different implementations of the recursive median are found in the literature. Their essential difference lies in the updating method used. Some methods use the simultaneous updating, whereas others use serial updating techniques. A two-dimensional recursive pseudo median is derived by doing row serial updating followed by column serial updating (Arce and Crinon 1984).

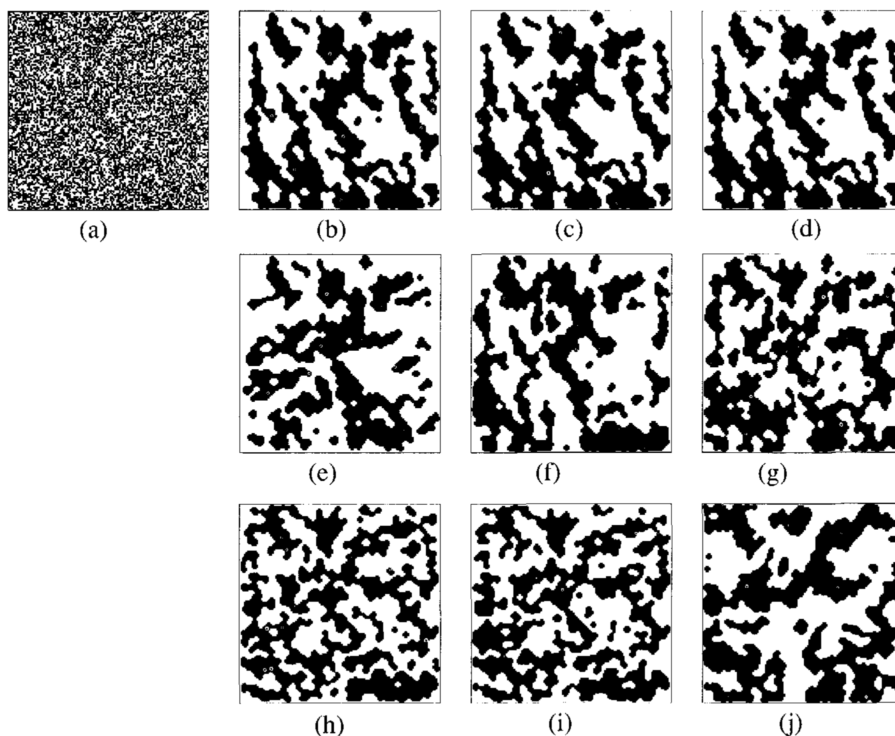


Figure 6.3 Effect of the binary median root RNO for different updating methods: (a) noisy binary original, (b) queue, (c) double serial, (d) single serial, (e) left spiral, (f) meander, (g) line serial, (h) simultaneous, (i) alternating, (j) right spiral updating.

The qualitative difference from the median root for some deterministic and one data-dependent updating method is illustrated in Figure 6.3 for a random binary input image.

All the images in Figure 6.3b-j are median roots[1] of the original Figure 6.3a. Some look quite similar, whereas others look completely different. The queue and spiral updating techniques, for example, give much 'smoother' results than the simultaneous and alternated updating.

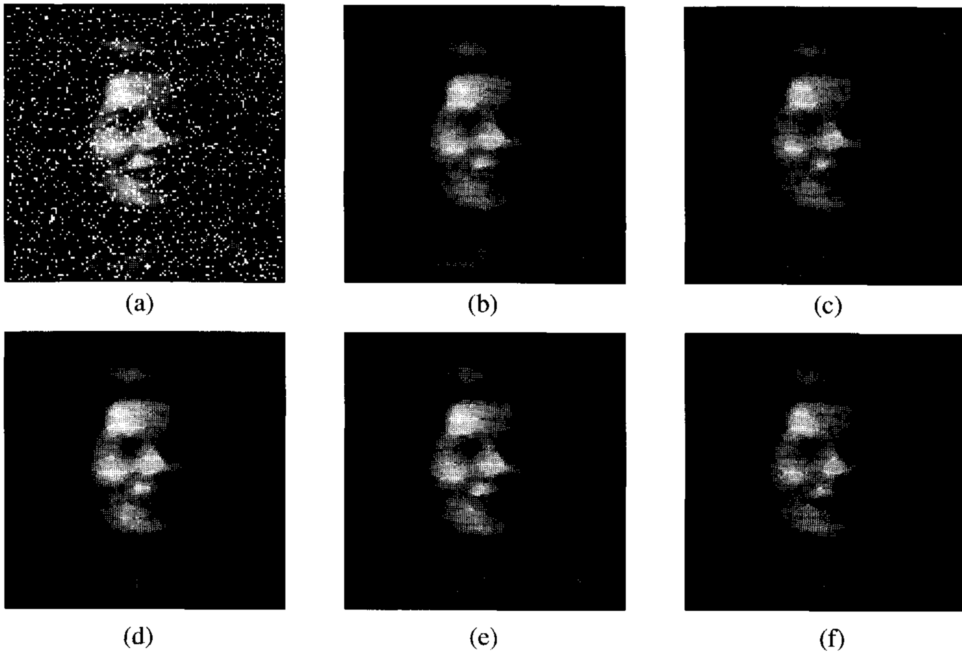


Figure 6.4 Median root RNO applied on a shot noise corrupted original (a) using the following updating methods: (b) queue, (c) serial, (d) spiral, (e) simultaneous, and (f) meander.

At this point it is useful to note, that the qualitative difference between applying the recursive median using different updating methods is less significant for "average case" images. An image to which the median root is normally applied consists of a normal grey value image corrupted with shot noise. The effect of applying the median root to such an average case image is shown in Figure 6.4.

6.3.2 Unique median roots

In Chapter 5, Section 5.5.5, we showed that a two-dimensional recursive maximum median and minimum median filter can be constructed which give unique results, i.e. they are independent of the updating method used. These filters also fall under the category described in Theorem 5.11, so that they can be calculated with the highly efficient bucket updating technique (see Chapter 8 for numerical results on this). The definition of the two-dimensional recursive median as defined in Equation (5.51) is:

$$(\forall \vec{p} \in \mathrm{Im})\ Y_{\vec{p}} = \underset{\vec{q} \in S}{\mathrm{med}}\ Y_{\vec{p}-\vec{q}}$$

with: $Y_{\vec{p}}^{(0)} = X_{\vec{p}}$,

The corresponding maximum and minimum recursive median filters can be constructed

1. Both terms *median root* as well as *recursive median* are used for the same type of operation.

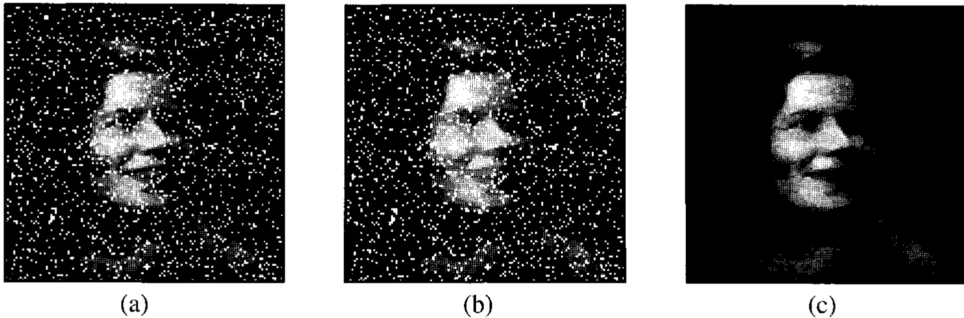
using the rule given in Section 5.5.5. This results in the following two equations:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \max(Y_{\vec{p}}, \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}})) \text{ (maximum median root, Eq(5.53))}$$

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \min(Y_{\vec{p}}, \mathrm{med}_{\vec{q} \in S}(Y_{\vec{p}-\vec{q}})) \text{ (minimum median root, Eq(5.52))}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}}$, in both cases.

The maximum median root will have a tendency to delete negative shot noise, whereas the minimum median root tends to delete positive shot noise. With this in mind, the effect of the maximum and minimum median root filters can be compared with respect to each other and to the 'normal' median root shown in Figure 6.4. This is illustrated in Figure 6.5. To our knowledge, these filters have not been introduced in the literature before.



(a) (b) (c)

Figure 6.5 Comparison of median root filters: (a) the original shot noise corrupted image, (b) the maximum median root and (c) the minimum median root.

The original image in Figure 6.5a is filtered through two different median root operations. The positive shotnoise in the original image is indeed not filtered away by the maximum median root, as shown in Figure 6.5b. However, the minimum median root applied to the original image filters this adequately (Figure 6.5c). The median root obtained with simultaneous updating preserves less detail, as can be seen in Figure 6.4e.

Open issues in the application of median root filters include the following:

- *Neighbourhood shape.* What is the qualitative effect of a different neighbourhood shape on the performance of the median filter?
- *Weights.* Can the introduction of weighting coefficients (as suggested in the literature: Wendt et al. 1986; Justusson 1981) further improve this filter, or even lead to different applications?
- *Relations.* The median root relates to morphology, in that the "median root of a finite [one-dimensional] signal is bounded by the open-closing and the close-opening" (Maragos and Schafer 1987c). The relation between the minimum and maximum two-dimensional median root operations and mathematical morphology still has to be determined.

6.4 The smallest enclosing regular polygon

The smallest enclosing regular polygon (SERP) is a unique RNO in the fixed point sense, which can be used to look for the smallest polygons in a specific orientation around objects. To our knowledge, it has not been described in the literature before. Its mathemati-

cal description is given in Section 5.2.3, which also shows that the SERP is actually an object operation. In Section 5.4.4 its uniqueness has been proved.

The SERP is not a stack filter, due to the presence of the crossing number[1] counter in the environment. The Boolean version of a stack filter can be described as a sum of neighbourhood element products. However, the crossing number operation can not be described as such.

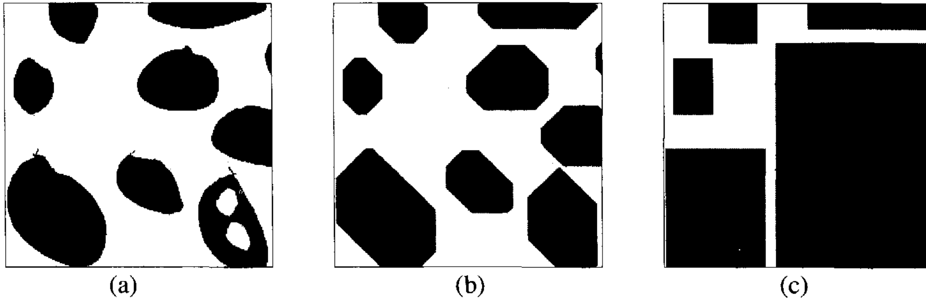


Figure 6.6 Smallest enclosing regular polygon as applied on an original image (a), using eight nearest neighbours (b) and using four nearest neighbours (c).

The usage of the SERP can best be demonstrated by Figure 6.6. For an image containing some objects, two different SERPs are taken. The first (Figure 6.6b) is computed with a neighbourhood with the eight nearest neighbours. This neighbourhood ensures the existence of polygonal angles of 135°, i.e. octagons. Taking only the four nearest neighbours (illustrated in Figure 6.6c) results in polygons with an angle of 90°, i.e. in squares. Because of the overlap occurring in the process of determining the smallest enclosing squares, the smallest enclosing square of the overlapping squares is calculated (see Section 5.2.3).

Because of the fact that the SERP is a unique RNO in the fixed point sense, it can be calculated fast using the queue updating technique (see the experiments in Chapter 8). It can therefore serve as a fast approximate alternative to more time consuming convex hull calculations.

6.5 Distance transforms

Several different distance transforms, with various applications, are described in the literature. The normal distance transform marks the distance from an object point to the closest background point. Several variations exist for the normal distance transform. First, the signed distance transform not only marks the distance to the nearest background point, but also stores a vector to that point. Second, some constraints may be introduced, so that the distance from an object point to the closest background point is measured via a path around the constraint. Thirdly, the medium through which the distances are measured may be inhomogenous, so that each distance step is weighted with a cost (additive or multiplicative). We will have a separate look at all three types of distance transforms.

6.5.1 Normal distance transform

The use of this distance transform is twofold. First, thresholding a distance transformed binary image at level g yields a binary image which is the original one eroded with a disc

1. The crossing number operation is defined in Chapter 5, Section 5.2.3.

with radius g. An erosion with radius g can also be executed "straight forward" on a binary image, i.e. by shifting the original and *and*ing the shifted versions with each other. Second, the distance transformed binary image can be used to construct the medial axis or a skeleton of the original image.

The normal distance transform can be viewed as a greyscale erosion, i.e. a greyscale morphological operation (Shih and Mitchel 1987). This can easily be seen from the definition of the distance transform (see Chapter 5, Section 5.2.3) and the definition of the greyscale erosion:

$$\begin{aligned}(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} &= \min_{\vec{q} \in S} \left(Y_{\vec{p}-\vec{q}} + \|\vec{q}\|\right) = \min_{\vec{q} \in S} \left(Y_{\vec{p}-\vec{q}} - c_{\vec{q}}\right) \\ &= (Y \Theta_g c)_{\vec{p}}\end{aligned} \tag{6.1}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}} \cdot L$, and L a value larger than the largest possible distance in the image.

c = the two-dimensional grey value function with the values $-\|\vec{q}\|$ on the appropriate positions (c is thus restricted by the size of S).

Θ_g = grey scale erosion as defined by mathematical morphology (Haralick et al. 1987).

Another use of the distance transform is that of determining the medial axis transform (MAT) of a binary object. Instead of calculating it directly, it can be done using the distance transform (Shih and Mitchel 1987). Contrary to the strictly morphological MAT, a skeleton can also be made using the distance transform as a start (Dorst 1986).

6.5.2 Signed Euclidean distance transform

The signed Euclidean distance transform stores both, the distance to the closest background point as well as the relative coordinates of that point (Danielsson 1980). In our notation it is written as (see Chapter 5, Equation (5.17)):

$$\begin{aligned}(\forall \vec{p} \in \mathrm{Im})\, Z_{\vec{p}} &= \operatorname*{argmin}_{\vec{q} \in S} \left\|Z_{\vec{p}-\vec{q}} + \vec{q}\right\| \quad + Z_{\vec{p}} \\ Y_{\vec{p}} &= \left\|Z_{\vec{p}}\right\|\end{aligned}$$

with: $Z^{(0)}_{\vec{p}} = X_{\vec{p}} \cdot \begin{bmatrix} L \\ \bullet \\ L \end{bmatrix}$, and L a value larger than the largest possible distance in the image,

the image Z contains coordinates, Y contains distances (i.e. grey-values), and X boolean.

The signed distance transform uses the Euclidean distance measure (defined in Section 5.2.3). Because of the fact that every destination image point has a *pointer* to the closest background pixel, it can serve as the basis of even more image processing operations than the normal distance transform. Algorithms are described for the detection of dominant points in a digital curve, curve smoothing, calculation of the Dirichlet tessellation and convex hull, using the signed Euclidean distance transform as a basis (Ye 1988).

6.5.3 Constraint distance transform

In path planning one may try to calculate the shortest path from a source point to a goal point given obstacles between them. For this end, the constraint distance transform can be used (Dorst and Verbeek 1986). Suppose that the original binary image X consists of all object points with the exception of one background point at the position of the goal. The binary constraint image C contains the obstacles as object points. The constraint distance transform as an RNO is then given by:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \begin{cases} C_{\vec{p}} = 1 & \rightarrow L \\ C_{\vec{p}} = 0 & \rightarrow \min\limits_{\vec{q} \in S} (Y_{\vec{p}-\vec{q}} + \|\vec{q}\|) \end{cases} \tag{6.2}$$

with: $Y^{(0)}_{\vec{p}} = X_{\vec{p}} \cdot L$, and L a value larger than the largest possible distance in the image.

The distances in the destination image remain infinite for the constraints and are calculated using the normal distance transform for the non-constraint points. After applying this operation, any point in the two-dimensional space gives the shortest distance with respect to the obstacles from that point to the goal point. From the start point, a path can be found by following the steepest descent. This is guaranteed to be optimal, because the distance transform is a monotone function, and therefore assures the absence of local extremes.

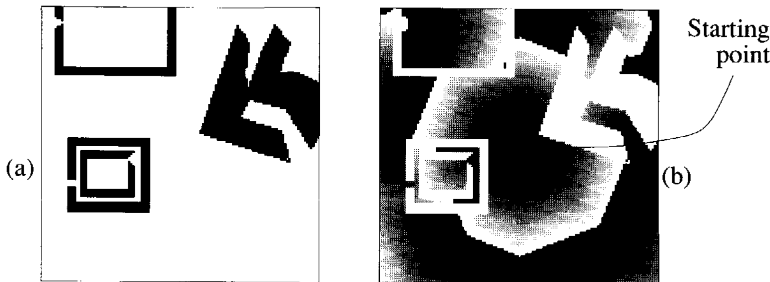


Figure 6.7 The constrained distance transform: (a) constraint image, (b) starting from the centre of the image using the constraints leads to grey value represented distance landscape (presented modulo 256).

The effect of the constrained distance transform is illustrated in Figure 6.7. Notice, that the distances are chosen such that black has a value of zero, and white a value of L (i.e. larger than the largest possible distance).

If only one path from a source to a goal point has to be calculated, then an A* algorithm can be used more efficiently than the constrained distance transform (Jonker et al.1988a).

6.5.4 Grey weighted distance transform

The GWDT has been introduced as an extension to the normal distance transform (Yokoi et al. 1981). Recently it has been discovered, that a solution to the Eikonal equation can be found in using the grey weighted distance transform (Verbeek and Verwer 1990). The Eikonal equation maps a source image Y^* into a destination image X in the following way:

$$(\forall \vec{p} \in \mathrm{Im})\, X_{\vec{p}} = \left|\mathrm{grad}\, Y^*_{\vec{p}}\right| = \min_{d\vec{q}} \left|Y^*_{\vec{p}-d\vec{q}} - Y^*_{\vec{p}}\right| \tag{6.3}$$

The inverse of this differential equation can be found by means of an integral:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \min_{\pi} \int_0^{\vec{q}_1} X_{\vec{p}-\vec{q}}\, d\vec{q} \tag{6.4}$$

with: $\pi = \left\{ \vec{q}_1 \middle| \left(\left(\vec{p} - \int_0^{\vec{q}_1} d\vec{q} \right) \notin \mathrm{Obj} \right) \right\}$

The discretisized version can be written as an RNO, and is given in Equation (5.20), which is repeated here:

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \min_{\vec{q} \in S} (Y_{\vec{p}-\vec{q}} + X_{\vec{p}-\vec{q}} \cdot \|\vec{q}\|)$$

with: $Y_{\vec{p}}^{(0)} = B_{\vec{p}} \cdot L$, and L is a value smaller than the largest weighted distance in the image.

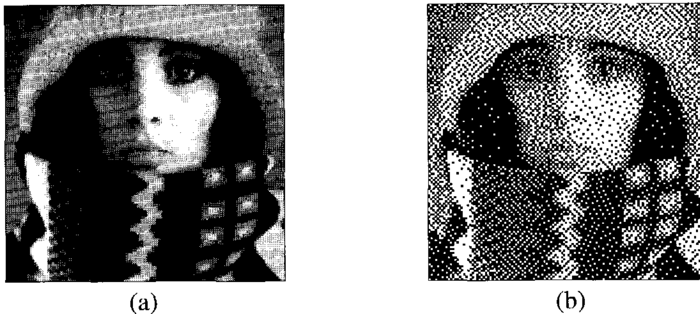
An example of the usage of the GWDT is the description of paths through inhomogenous media. For a complete description of the GWDT and its applications we refer to the literature (Verbeek and Verwer 1990).

6.6 Other non-linear RNOs

Besides the RNOs which have been discussed in this chapter, a number of others are in use in image processing. In this section some of them will be mentioned.

6.6.1 Dithering algorithms

These algorithms transform a grey value image into an image with less grey levels (ultimately a binary image) in such a way that the total intensity in a local neighbourhood is made proportional to the light intensity of the central point in that neighbourhood. Dithering methods can, for example, be used to binarise grey-value images for printing purposes.



(a) (b)

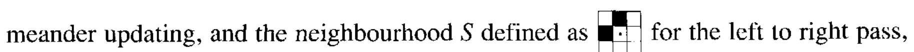
*Figure 6.8 Dithering example on a 128*128*8 bit image: (a) original grey value image, (b) binary image, dithered according to the Vossepoel technique.*

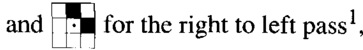
We will explain an example of such an algorithm taken from the literature (Vossepoel et al. 1979) and shown in Figure 6.8. The Vossepoel algorithm can be written as an RNO from the grey value source image X to the grey value destination image Y followed by a point operation from Y to a boolean image B in the following way:

$$(\forall \vec{p} \in \mathrm{Im}) \begin{cases} Y_{\vec{p}} = \left(X_{\vec{p}} + \frac{1}{\#S} \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}}\right) \bmod (2^{b-1}) \\ B_{\vec{p}} = \left\lfloor \left(X_{\vec{p}} + \frac{1}{\#S} \sum_{\vec{q} \in S} Y_{\vec{p}-\vec{q}}\right) / (2^{b-1}) \right\rfloor \end{cases} \quad (6.5)$$

with: $Y_{\vec{p}}^{(0)} = 0$,

the edge is set at a constant value (good results are obtained by putting the edge points at values which are uniform distributed between 0 and 2^{b-1}),

meander updating, and the neighbourhood S defined as  for the left to right pass,

and  for the right to left pass[1],

#S = the number of elements in the neighbourhood.

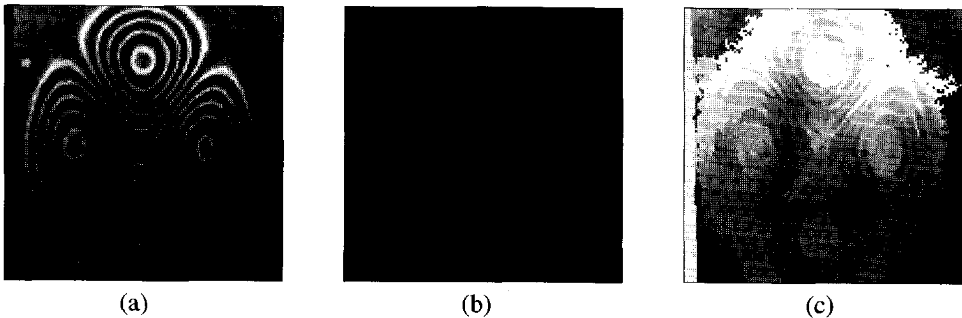
The Vossepoel dithering RNO uses a non-cyclical neighbourhood (see Chapter 5, Section 5.4.2), so that it gives a unique result at all times.

6.6.2 Envelope estimation

Haralick introduced an interesting operation which he calls the 'relative extreme operator' (Haralick 1981). He describes this operation as follows: "each pixel in the output image is filled with the value of the highest extreme which it can reach by a monotonic path". From this description it can be seen that the relative extreme operator is a global operation. As shown in Section 5.2.2, Equation (5.5) the relative maxima operator can be written as the following RNO (substitute the 'max'-operator for 'min' and the 'larger than' for 'smaller than' for the relative minimum):

$$(\forall \vec{p} \in \mathrm{Im})\, Y_{\vec{p}} = \max_{\vec{q} \in S} \{ Y_{\vec{p}-\vec{q}} \mid (X_{\vec{p}-\vec{q}} \geq X_{\vec{p}}) \}$$

with: $Y_{\vec{p}}^{(0)} = X_{\vec{p}}$.



(a) (b) (c)

*Figure 6.9 Relative extreme operation: (a) 128*128 original, (b) result of applying the 3*3 relative minimum, (c) result of applying the 3*3 relative maximum operation.*

As shown in Figure 6.9a-c, this operation can be used to estimate the envelope of a two-dimensional signal and to remove shading from an image. Haralick shows that it can be used to find the relative extremes (Haralick 1981).

6.6.3 Labelling algorithms

Another application of the RNOs is the labelling of binary images. This operation is actually an extension to the object selection operation. In a labelled image, every object is 'coloured' with a distinct grey value. The labelling operation maps a binary image into a grey value image. The simplest way of labelling an image is shown in Equation (6.6):

$$Y_{\vec{p}} = X_{\vec{p}} \wedge \min_{\vec{q} \in S} Y_{\vec{p}-\vec{q}} \tag{6.6}$$

with: $Y_{\vec{p}}^{(0)} = \mathrm{Unique}(\vec{p})$, where the *Unique(...)* function gives a unique number for each im-

1. The dot indicates the position of the central pixel.

age point, and is usually defined as:

$\text{Unique}(\vec{a}) = \vec{a}_x + \text{width} \cdot \vec{a}_y$, where *width* is the image width.

X is the binary source image containing the objects to be labelled.

The correctness of Equation (6.6) is stated and proven as follows.

Theorem 6.1 Labelling

The labelling algorithm as given in Equation (6.6) produces a grey value image Y in which each object from the original binary image X is represented by a unique number.

Proof 6.1 Labelling

By the definition of the Unique(...) function every $Y_{\vec{p}}^{(0)}$ *has a unique initial 'label' to it. In the first processing step,* $Y_{\vec{p}}^{(1)}$ *is produced according to* $Y_{\vec{p}}^{(1)} = X_{\vec{p}} \wedge \min_{\vec{q} \in S} Y_{\vec{p}}^{(0)}$ *, so that:*

$Y_{\vec{p}}^{(1)} = X_{\vec{p}} \wedge \min_{\vec{q} \in S} \text{Unique}(\vec{p} - \vec{q})$ *This process continues until the borders of the objects are reached (across the borders X becomes zero):*

$$Y_{\vec{p}}^{(\infty)} = X_{\vec{p}} \wedge \min_{\vec{q} \in \text{Obj}} \text{Unique}(\vec{p} - \vec{q})$$

The resultant minimum of a set of minimum numbers belonging to one object is one of the numbers within that object, and therefore unique between objects.

□

Other functions can also be used to construct a unique number per PE (Basille 1989). Depending on the function *Unique(...)* this labelling operation is monotonically decreasing with a lower bound. The fact that it is monotonically decreasing stems from the minimum operation, and the initialisation of the destination image Y at the specified value L. The lower bound results from the nature of the *Unique(...)* function. The function that we take here is positive, because $\vec{a}_x$, *width* and $\vec{a}_y$ are positive. Because of the above and according to Theorem 5.11, the labelling operation gives a unique result, and can be calculated with the bucket updating technique.

6.7 Conclusions

This chapter shows the relation between mathematically introduced RNOs and image processing applications. The emphasis is on non-linear RNOs. One linear RNO is presented as a reference to the user with a background in multi-dimensional linear system theory.

We conclude that the RNOs shown in this chapter cover some important aspects of image processing, but, at the moment, only to some extent. The skeleton construction (i.e. morphological skeleton) does give a skeleton from which the total object can be retrieved, but it is not necessarily connected. The shown RNO skeleton is not necessarily connected, but other RNO-like algorithms exist which do give a connected skeleton (Verwer et al. 1989). The noise filtering examples that were shown (i.e. the median root type filters) cover only part of the noise filtering problems. The smallest enclosing regular polygon that was shown is only a coarse approximation of the convex hull. It depends on the application whether this approximation suffices or the real convex hull is needed. The path finding problem is faster solved (on a single processor computer) using the A* algorithm than by calculation of the (constrained) distance transform (Jonker et al.1988a).

It is our hope, that the increase in the theoretical background of RNOs given in Chapter 5, will promote the number and quality of their application in image processing.

Chapter

7 Implementation of RNOs on the architecture groups

In the previous chapters it was found, that RNOs can be calculated in a number of ways, described by an updating method. Three types of updating methods are distinguished: deterministic, data dependent and stochastic. Combinations between these types are also possible. The efficiency with which updating methods can be implemented on the different architecture groups differs. For every architecture there is an updating method which is optimal in the sense that it fits well within the structure of the architecture. In this chapter it will be shown which updating methods can efficiently be programmed given an architecture group.

To compare the efficiency with which the SPA, LPA and PL can implement several updating methods, Section 7.1 discusses how the deterministic updating methods can be programmed on the different architecture groups. Section 7.2 and Section 7.3 show how the data dependent and the stochastic updating methods can be done with the different architecture groups. Combinations of updating methods are also possible and may even be desirable. This is investigated in Section 7.4. An evaluation of the efficiency with which updating methods can be *programmed* on the different architectures is then given in Section 7.5.

7.1 Deterministic updating methods

The implementation of deterministic updating methods will be discussed for all three low-level image processing architecture groups: the square processor array, linear processor array, and pipeline. However, for clarity the implementation will first be shown for an image processor consisting of a single processor.

7.1.1 Single processor

In a single general purpose image processor, any deterministic updating method can be programmed in a straight forward way. This is done with the help of the index mapping function (IMF; see Chapter 5). A one dimensional array is constructed which contains pointers to the image points. The image points are stored in the array in the order as indicated by their rank values in the IMF. Assuming that this IMF-array has been set-up, the program for a general purpose computer proceeds as shown in Procedure 7.1.

Points with the same ranking order value are evaluated, and their outcomes are put in a buffer. Next, all buffered values are assigned to the outputs of the PEs. This is done for all the points in the IMF array. The whole process is repeated until some kind of global error measure is low enough.

Procedure 7.1 Single Processor deterministic updating

```
PROC Single_Processor_IMF_Evaluate()
  REPEAT
     FOR "all points in the IMF-array" DO
        FOR "all points of the same rank" DO
           pe.buffer = "Evaluate the PE of present IMF point"
        OD
        FOR "all points of this last rank" DO
           "Adjust global error with pe.buffer and pe.output"
           pe.output = pe.buffer
        OD
     OD
  UNTIL "global error low enough"
CORP
```

with: pe.output = "output value of Processing Element", and
pe.buffer = "buffered value of Processing Element"

For Boolean or integer RNOs, the number of points from which the output value has changed between iterations can be taken as the global error measure. For floating point RNOs a different measure has to be taken. We have used two measures in our simulator:

- *Mean square difference.* The squared differences between PE output values of two iterations are added and divided by the number of image points.
- *Maximum absolute difference.* The differences between PE output values of two iterations are compared, and the maximum absolute one is taken.

As a slight variation to the normal deterministic methods, each of the updating methods can be speeded up by Successive Over Relaxation (SOR). Note that the SOR can only be applied to some specific RNOs (see Chapter 5). The discrete version of the Poisson equation is one of the few examples. Deterministic updating methods with SOR are programmed slightly differently from normal IMFs, as is shown in Procedure 7.2.

Procedure 7.2 Single Processor IMF SOR evaluation

```
PROC Single_Processor_SOR_Evaluate()
  ω = 1
  ρjac = 1 - ( π² / 2•image_size)
  REPEAT
     FOR "all points in the IMF-array" DO
        FOR "all points of the same rank" DO
           pe.buffer = "Evaluate the PE of present IMF point"
        OD
        FOR "all points of this last rank" DO
           "Adjust global error with pe.buffer and pe.output"
           pe.output = (ω • pe.buffer + (1-ω)•pe.output)
        OD
     OD
     ω=IF first cycle THEN (1 - ρ²jac/2)^-1 ELSE (1 - ρ²jac•ω/4)^-1
  UNTIL "global error low enough"
CORP
```

Through the use of the IMF-array, the implementation of the deterministic updating methods on a general purpose single processor is programmatically quite simple. Any deterministic updating method could in principle be implemented, by calculating its IMF-ar-

ray. For the sake of comparing updating methods, such an implementation has taken place. The experimental results for the updating methods are presented in Chapter 8.

Deterministic updating methods like serial updating can be implemented more *efficiently* by traversing the image in the correct order (raster scan wise for serial updating) and storing the evaluated results in the PE output immediately instead of through a buffer. The Procedure 7.1 serves to show how *any* deterministic updating method can be implemented, regardless of the efficiency.

7.1.2 Pipelined processors

Because of the shift registers used in pipelines, such an architecture is bound to a horizontal or vertical raster scan input. Therefore, the only deterministic updating methods which can be done with a pipeline are simultaneous and serial updating. Normally, pipelines only implement the simultaneous updating. Serial updating is only possible when they have the facility to recursively feedback their output, and process this output at the same speed with which pixels go through the pipe. Vertical or horizontal serial updating can then be done in a very natural way, by feeding the image points in the correct order through the pipe.

Pipelines which take their input directly or through a line buffer from a raster scan input device can only do the horizontal serial updating in the top down direction.

However, as is shown experimentally in Chapter 8, it is very advantageous to do 'double serial' scanning, i.e. alternatingly scanning downward and upward. Such a combination of upward and downward serial updating is only possible if the processed image is fed through a buffer image. This frame recirculation breaks the pipeline, because the image data has to be gathered in a buffer, and fed through the pipeline a next time, so that the pipeline can not be used to process incoming data.

7.1.3 Linear Processor Array

The most natural updating methods for the LPAs are simultaneous and line serial updating.

Any 'one dimensional' index mapping function could be done with an LPA using a program such as Procedure 7.1 for the single general purpose image processor.

With regard to existing LPAs, it can be said that they all use some form of line serial updating. The AIS-5000 uses row-serial top-down updating for some functions. Such a function is programmed on the micro-assembler level. The SYMPATI-2 can perform row- or column-serial updating in any direction due to its helicoidal processor mapping system.

7.1.4 Square Processor array

From the group of deterministic updating methods, the SPA most efficiently implements the simultaneous updating (i.e. iteration). Some SPAs have an activity bit per PE, so that methods like the alternating updating can also be performed. It is clear, however, that this method is less efficient in an SPA, because fewer PEs are actually used.

SPAs with a crinkle wise PMF may use alternated updating. A 32*32 SPA with a crinkle wise image storage facility will, for example, process a 64*64 image using 4 alternated updating.

7.2 Data dependent updating methods

The use of data dependent methods for the parallel, low-level image processing archi-

tecture groups which we have investigated, is unnatural. Data dependent updating methods process pixels on a sequence of positions which is not deterministic. Any architecture which implements such an updating technique should have the possibility of processing one or more points on any position. Such a possibility is generally not offered by the LPA, SPA or PL. Pixel positions are mapped to the LPA or SPA through the processor mapping function which is deterministic. It can not be made data dependent, because of the SIMD character of these architectures. However, LPAs or SPAs with local addressing autonomy might be capable of doing data dependent updating. This has not been investigated. It is one of the suggestions in Section 9.2, to investigate this.

An architecture which implements the bucket updating technique specifically for the distance transform has been proposed in the literature (Jonker et al.1988a). A data flow architecture is proposed, where a number of PEs work in parallel on pixels which reside in buckets whose values differ less than the lowest distance which can be taken by the distance function (i.e. less than 1 for the city block distance, and less than 5 for the 5-7-11 distance). After the pixels in these buckets are processed, a global counter is incremented, so that points with a higher distance can be processed. This process is repeated until all buckets are empty. This architecture does not have an emphasis on low-level image processing operations (only a subset of one low-level image processing operation group can be processed on it), so that it falls out of the scope of this thesis (see Section 2.1).

7.3 Stochastic updating methods

The use of a stochastic method like Poisson updating, where every PE has its own stochastic process to decide when it will calculate a new value, is contrary to the parallel SIMD nature found in SPAs, PLs and LPAs. Even if it were implemented with the use of local function autonomy (see Chapter 3), it is still obvious that such a method is very inefficient, because only one pixel is updated per time step.

Asynchronous updating, however, is found among low-level image processing architectures. It is used for example in the CLIP4 square processor array and it was suggested for use in the BASE and the SLAP (see Chapter 2). Pipelines are not suitable for any of the asynchronous updating methods because they do not possess spatial connections between PEs. Only one PE is performing one iteration of an RNO.

The performance of the asynchronous updating method has been studied as a function of the 'relative variation on the processing time per PE' (denoted as RVPT). Recall from Section 5.3.3, Equation (5.31) that the time it takes to calculate the value of a point, given that all the neighbour values are known, can be written as:

$$t_{point} = t_{average} + t_{variable} \cdot \rho,$$

with: ρ is a random value uniformly distributed between -1 and 1.

Using Equation (5.31) the RVPT is defined as follows:

$$RVPT = \frac{t_{variable}}{t_{average}} \tag{7.1}$$

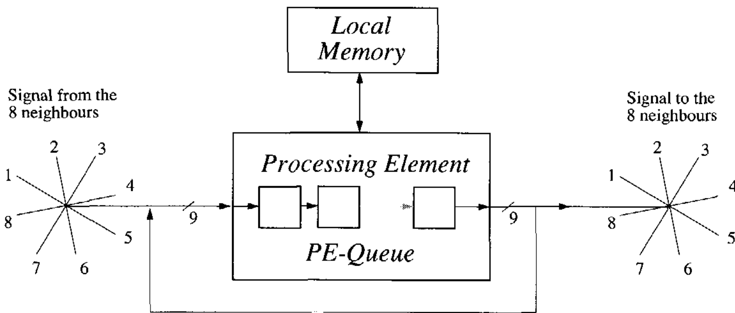


Figure 7.1 Simulation of a Processing Element with asynchronous updating.

The updating method has been implemented in a simulator. The simulation of asynchronous updating is done by representing a PE as a queue of calculated output values.

In discrete time steps of arbitrary precision, every PE that is connected to a neighbour that has changed, calculates a new value with the neighbour values of that moment. That value is stored in the PE-queue, together with a stochastic estimate of the time the result is available. Also in every time step, every PE-queue is checked for results which become available in that time step.

The algorithm in pseudocode is shown in Procedure 7.3. First, the RNO is calculated for all points in the image, and a time is calculated at which they will have to show their contents to their neighbours. The calculation of the time is in accordance with the formula in Chapter 5, where the variance over the processing time can be adjusted. The time and value are put in a queue within the PE structure. The size of this queue depends on the resolution of the time representation which is chosen.

After the initial filling of the PE queues, a global discrete clock start is started until the RNO is finished. At each time step, every PE is checked to see if the moment has come for it to output its data. If such a moment has come for a PE, its output is assigned to the present PE output. Neighbouring PEs are put in a recalculation buffer.

Procedure 7.3 Asynchronous updating simulation

```
FOR "all PE's" DO
  enqueue(new_ready_time(),PEevaluation() )
OD
FOR "time-clock runs until ready" DO
  FOR "all PE's" DO
     WHILE (present_time>=Next_queue_element.time) DO
        PE_output := dequeue().value
        FOR "all my neighbours" DO
           "put this neighbour in the recalculation buffer"
        OD
     OD
  OD
  FOR "all PE's in the recalculation buffer" DO
     enqueue(new_ready_time(),PEevaluation())
     IF (Last_queue_element.time<Previous_queue_element.time) THEN
        Last_queue_element.time:=Previous_queue_element.time
     FI
  OD
OD
```

After all the outcome times within the PEs have been checked, all PEs which are in the recalculation buffer are evaluated. These output values, together with newly calculated output times are enqueued within the PE structures. To prevent values which were calculated later from having an output time which is sooner than the last enqueued value, some correction may take place on the output time.

The process is repeated for every time step and finishes when the image does not change anymore.

7.4 Combinations

To improve the performance of scanning square or linear processor arrays, a combination of updating methods can be made to improve the overall performance for the processing of RNOs. Within the SPA or LPA one updating method is used and the array is moved over the image by means of a second updating method.

An example of a combination of two updating methods for the efficient processing of RNOs is seen in the operation of the CLIP4-Delft. This 64*32 bit serial CLIP4 SPA (see Chapter 2 for details of the CLIP4) uses asynchronous updating within its array. The experiments in Chapter 8 show that the most efficient updating method in general is a data-dependent one like bucket or queue updating. Therefore, it is decided also to use a data-dependent method on the scanning level of the processor arrays. For this purpose, queue updating is chosen. It is difficult to implement bucket updating on a scanning level, because this method requires individual pixels to be put in a bucket with their corresponding value, whereas the scan as a set of individual pixels. Instead of applying the queue updating to individual pixels, the queue is filled with pointers to image parts of 64*32*1 bit. If processing such a scan changes any of the points in the scan, then neighbouring scans (depending on the neighbourhood which is used) are put in the queue. The measurement of the overhead for performing LNOs using edge store scanning (ESS) on the CLIP4-Delft has been described in Section 4.7.4. The ESS technique is used in combination with queue updating for performing RNOs. The overhead for an RNO is now problem dependent, as it increases with the number of 'extra' scans that have to be processed. No hardware scanning techniques have been implemented for this SPA.

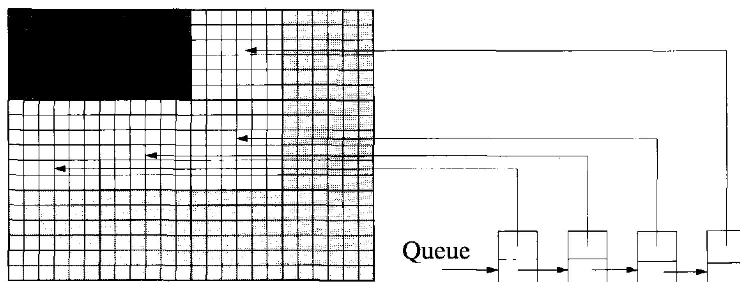


Figure 7.2 A combination of queue updating on the scan level, and another updating method on the PA level.

Other SPAs, which can perhaps only use a deterministic method such as simultaneous updating, could also benefit from using queue updating on the scan-level. An efficient implementation may be difficult, however, for SPAs which use a hardware scanning tech-

nique.

A combination of simultaneous updating on the array level and queue updating on the scan level was also found to be efficient for LPAs. This is also shown in Chapter 8.

7.5 Conclusions

It has been shown, that the deterministic, data dependent and stochastic updating methods can be very efficiently *implemented* on the single processor architecture (the *quantitative* efficiency of the updating methods is investigated in Chapter 8). The SPA, PL and LPA can only implement some of the deterministic updating methods. All architectures implement at least simultaneous updating. The LPA and the PL could be enhanced with line serial updating. The PL can do serial updating, but if no frame recirculation may be done, then this updating method may only be done in one direction across the image (i.e. raster scan wise). The LPA offers more flexibility, as it allows up and down line serial scanning. If the processor mapping function is helicoidal as in SYMPATI-2, then it also allows horizontal line serial scanning.

Data dependent updating methods can only be done on the 'scanning level' of the SPA and the LPA. The PL in principle doesn't allow such techniques. The processor arrays, however, allow a combination of simultaneous (or asynchronous) updating within the array and queue updating on the scanning level.

The Poisson updating method can only be implemented efficiently on the single processor architectures, as it is inherently not parallel. The asynchronous updating method is used by some bit serial processor arrays which allow data to be calculated in a combinatorial way across the array. A simulator has been written which implements this updating method.

The quantitative efficiency of the various updating techniques is experimentally investigated in the next chapter.

Chapter

8 Experiments with RNOs

The comparison between low level image processing architectures on the basis of operation analysis has been done to some extent in Chapter 4. One of the groups for which the analysis has not yet been done in literature is the recursive neighbourhood operation (RNO). Some theoretical background for RNOs has been established in Chapter 5. The performance of the RNOs for different architectures will be investigated in this chapter. The analysis is done in two parts. First the performance of deterministic, data dependent, stochastic, and combined updating methods as described in Chapter 5 is measured for a selected set of RNOs. The measurements are done using a simulator which can be programmed for the different updating methods. Second, the efficiency and speed are derived with which the different architecture groups can perform the RNOs, given their ability to perform the updating methods (see Chapter 7).

This chapter starts by showing the characteristics of the program that is used to implement the different updating methods. The RNOs for which the updating method performance is measured, and the images for which this is done are then discussed in Section 8.2. An overview of the resulting values for the performance of the updating methods is given in Section 8.3. The speed and efficiency values with which the different architecture groups can implement the investigated RNOs are then derived in Section 8.4. Finally, conclusions for the performance of RNOs in relation to the architecture groups are discussed in Section 8.5.

8.1 Characteristics of the updating method implementations

The updating method simulation toolset is written in the C programming language, and interfaced to TCL-Image as a special command group. TCL-Image is a combination of a command language (TCL) and specific image processing routines developed by the Delft Centre for Image Processing (CBD) and marketed by MultihouseTSI (Amsterdam). The toolset could just as well be interfaced to other image processing command languages, but TCL-Image was chosen because the roots of TCL-Image are in the Pattern Recognition group where the author has done the research for this thesis.

The specification of an RNO goes through the *SRset* command. This command allows the selection of an RNO (implemented in the "C"-language as a function), the neighbourhood (any combination of the points in a 3*3 neighbourhood), corresponding coefficients (integer), and an edge type. The last one specifies in which way the edge of the image has to be treated. There are three possibilities: constant, wrapping or mirroring (defined in Section 5.1.4). After specifying the RNO and its attributes through the *SRset* command, there are several possibilities to initialize the data on which the RNO will operate.

Processing of the RNO is started through the *SRpst* (process and store) command. The following updating methods can be specified with this command (see Chapter 7 for implementation details):

- *Deterministic:* simultaneous, alternating, single serial, double serial, spiral, meander, line serial, blob, and the combination of alternating and SOR. These methods are implemented using a set of previously loaded index mapping functions (IMFs), between which a choice is made. Some of these IMFs are also shown in Chapter 5, Figure 5.10.
- *Data dependent:* recursive, queue, bucket. With queue updating, a function should be specified which loads the queue with initial values. It will be specified in Section 8.2, which functions are taken for this purpose. Any of the IMFs can serve to specify in what order the image is scanned if the queue is empty before all the image points are visited.
- *Stochastic:* asynchronous. A queue with a maximum length of 25 elements is used inside each 'software PE' (see Section 5.3.3 for a full description of asynchronous updating, and Section 7.3 for implemental details). The mean number of time units t_{average} with which a PE can process a pixel should be specified, as well as the variable number of time units t_{variable} (according to Equation (5.31)). In our experiments, the mean value is taken to be 10 time units, and the variable value is uniformly varied from 0 to 9 time units in steps of 1. The variable processing time with respect to the mean PE processing time is defined in Section 7.3, Equation (7.1) as the *Relative Variable Processing Time*:

  $$\text{RVPT} = \frac{t_{\text{variable}}}{t_{\text{average}}}.$$
- *Combined*: a combination of simultaneous updating on the array level and queue updating on the scanning level. Scans with a size varying from 1*1 to $\sqrt{P} \cdot \sqrt{P}$ (the array has P processors) can be specified.

The measure used to determine whether an RNO is finished or not is the maximum absolute difference. For the deterministic updating methods the maximum absolute difference between two iterations of updating is taken. The queue updating method takes the maximum absolute difference between two passes through the queue. Note that the queue is initialized with a set of pointers to image points, and they are processed 'simultaneously'. This is done by buffering the calculated values between two passes through the queue.

For all the updating methods, the number of times that an image point is evaluated is measured. When the routine *SRpst* is entered, this number is initialized. When *SRpst* is finished, then the evaluation number is divided by the number of points in the image. This gives the *updating effort* E_u in number of evaluations per image point.

8.2 Definition of the RNOs and their input images for the experiments

To cover possible differences in binary and grey value RNOs, the updating method experiment is performed on a number of different operations:

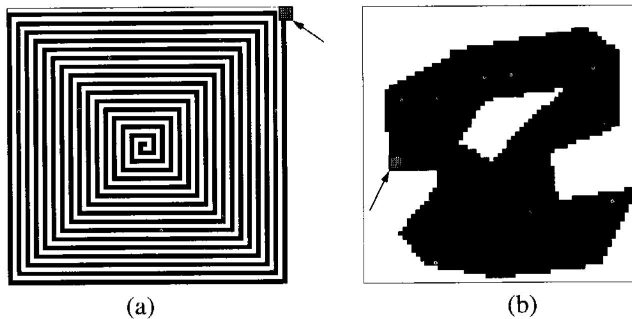
- *Binary to binary:* object selection, smallest enclosing regular polygon
- *Binary to grey value:* normal and grey weighted distance transform
- *Grey value to grey value:* median root and maximum median root
- *Floating point:* the (linear) discrete Poisson equation which serves as a reference for readers with a background in linear systems theory.

In the following sections *difficult* and *average* cases with respect to the investigated updating methods and the RNOs on which they work will be derived. The average case input image is an image which could more or less be expected to occur quite often as input to the RNO (i.e. it depends on the application area of the RNO). The difficult case is chosen with the mathematics of the operation in mind. We chose to use 'difficult' cases rather than 'worst' cases in our experiments, because the 'worst' case largely depends on the updating method so that it is sometimes impossible to define one 'overall' worst case per RNO.

All the images used for the experiments as described in this chapter have a size of 64*64 pixels. Where indicated they are binary or grey value. The qualitative results of applying an RNO on a difficult or average case input image will only be shown where they are not trivial.

8.2.1 Object selection

In this operation a binary seed image is propagated in a binary source, so that the object which covers the same position as the seed is selected (see also Section 5.2.3.3). Note that two input images are required; one for the seed and one with the objects. Depending upon the updating method used, the seed image, and the source image, the updating effort differs tremendously. A well-known difficult case for most of the data in dependent updating methods is filling a spiral-formed object from one seed pixel. The simultaneous updating will step only one point of the spiral at a time. The upwards and downwards serial updating will be able to do two corners of the spiral at a time.



(a) (b)

*Figure 8.1 Binary test images of size 64*64 for the object selection operation which starts from a seed indicated by the arrow: (a) difficult case and (b) average case*

For the analysis of the performance for an average case, it should be noted that the objects which have to be filled may be slightly irregular in shape and may contain one or two holes. With this in mind the object as shown in Figure 8.1 serves as average case. The filling is done starting at the indicated seed positions (only one seed pixel is taken in either case).

8.2.2 Smallest enclosing regular polygon

This operation tries to fit polygons around objects in the binary input image (a definition is given in Section 5.2.3.4). If two enclosing polygons touch, they are regarded as one new object, and a polygon is fitted around them. This is due to the recursive nature of the calculation of the SERP (smallest enclosing regular polygon). A difficult case will be the case where the most pixels have to be filled to get the SERP, yet the least pixels are initially one. We think that an image with a diagonal one pixel thick cross will serve as a good

example of a difficult case because the SERP operation will make all zero valued pixels in the image one (starting from the center), and only a few pixels are initially non-zero. This is shown in Figure 8.2c.

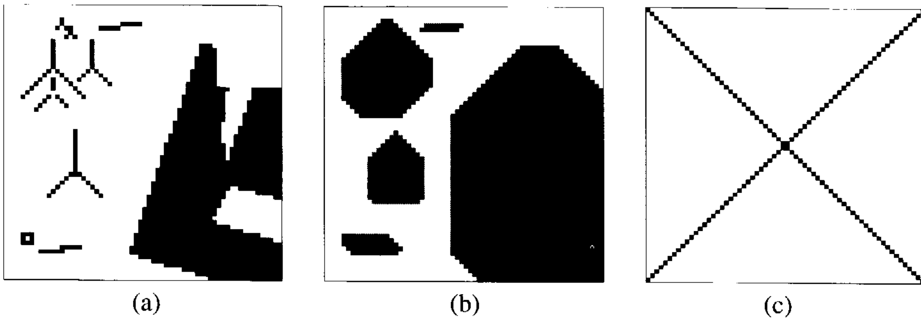


*Figure 8.2 Test images for the SERP; (a) 64*64 average case input, (b) average case result overlaid on the input, and (c) difficult case input image.*

The average case is an image with a few objects in it. Some may be misaligned according to the directions of the polygon. Some may have holes. Some of the object polygons may overlap. The test image which we have taken for the average case is shown in Figure 8.2a, and its resulting image is shown in Figure 8.2b.

8.2.3 Normal distance transform

This operation takes as input a binary image with objects and background. The output consists of a grey value image where the distances from every point to the closest background point are given (defined in Section 5.2.3.1). The distance measure used is the 5-7 distance, as defined in Section 5.2.3. A difficult case input image will be one in which the amount of updating to be done is maximal. This amount of updating is proportional to the number of object points and to the largest distance which occurs in the resulting distance image, but also depends on the updating method used.

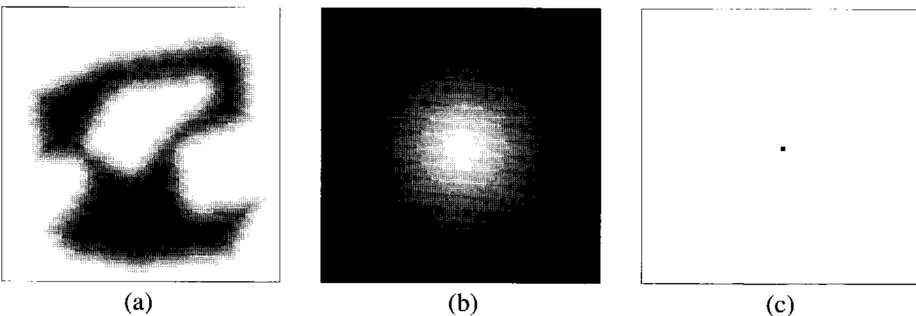


*Figure 8.3 The distance transform: (a) 64*64 average case result from original in Figure 8.1b and (b) difficult case result (c) original for difficult case.*

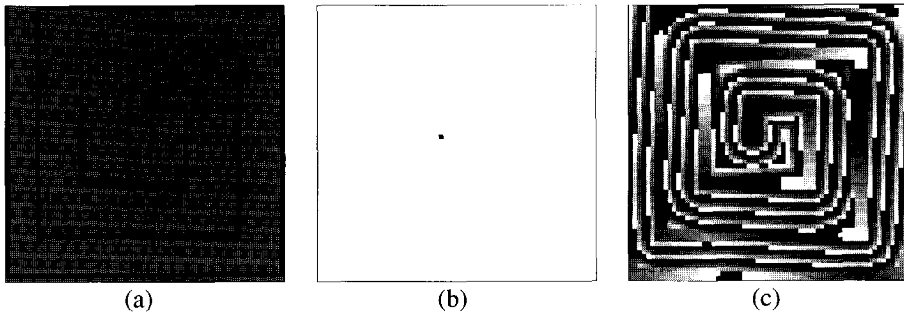
We propose to take an image that contains a very large object and very little background, so that the number of points without zero-distance is large. An image is taken with one background pixel and for the rest foreground points. If the position of the background pixel is taken in the upper left corner of the image, then the serial updating method will calculate the distance transform in one iteration through the image. If, however, the posi-

tion of the background point is taken in the lower right corner, then the speed of serial updating and simultaneous updating are the same. As an example of a difficult case, we choose the position of the background point in the middle of the image (see Figure 8.3c). The result of the normal distance transform is shown in Figure 8.3b.

On the average the distance transform is done on one or more objects with arbitrary shape or size. For the average case performance analysis we therefore take the same object as shown in Figure 8.1b, with the result shown in Figure 8.3a.

8.2.4 Grey weighted distance transform

The grey weighted distance transform (GWDT) takes two input images: a binary image with object and background, and a grey value image with the weight coefficients (for its definition see Section 5.2.3.2). As with the normal distance transform, the 5-7 distance is taken (see Section 5.2.3). Just as with the normal distance transform, the aim of the grey weighted distance transform is to calculate the distance from any image point to the closest background point. However, this is here done with respect to the weight coefficients stored in the grey value image.

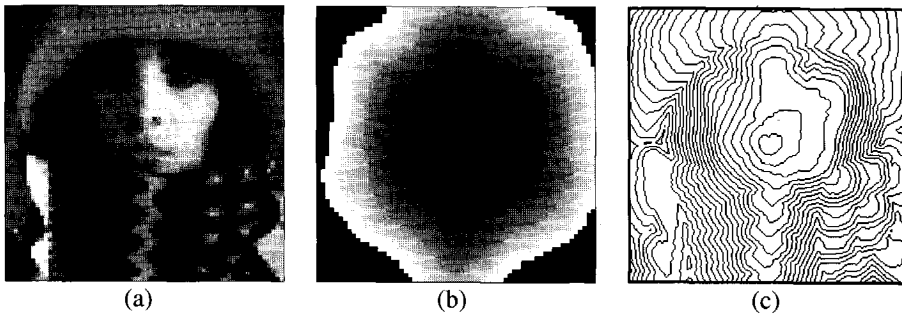


(a) (b) (c)

*Figure 8.4 Difficult case for the GWDT: (a) 64*64 grey weight input image, where black corresponds to the value 0.1 and white to the value 30, (b) binary object original, (c) resulting image where distances are shown modulo 256.*

For the difficult case grey value input image we propose to take a grey value spiral as shown in Figure 8.4a. The coefficients on the spiral are all assigned a very low value, whereas the other coefficients all have the same high value. The corresponding difficult case object image in Figure 8.4b consists of one background point at the center of the spiral, and all the other points belong to the foreground. When calculating the distance from any foreground pixel to the one background point, paths are forced to follow the spiral shape imposed by the grey value weight image (see Figure 8.4c).

The average case should be defined with respect to the 'normal' use of the operation. Although the GWDT can be used in many applications, for the determination of the average case we consider the coin application as described in the literature (Verbeek and Verwer 1990).



(a) (b) (c)

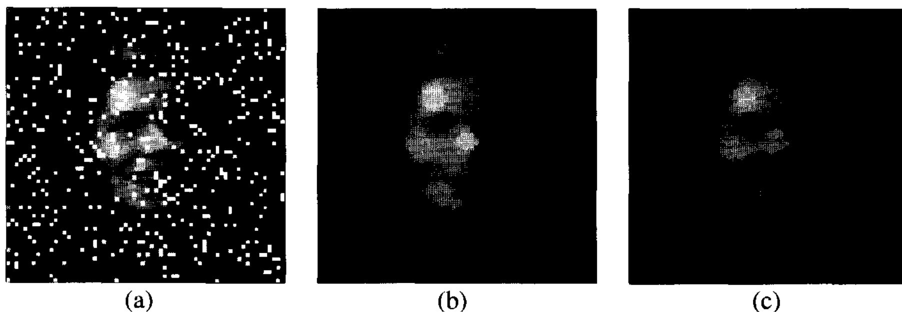
*Figure 8.5 Average case for the GWDT: (a) 64*64 original with grey weights, (b) result - modulo 256 - of applying the GWDT with object image as shown in Figure 8.2c, (c) contour plot of the GWDT applied on the 256*256 version of (a).*

The average case is therefore an existing grey value image (Figure 8.5a) that, when subtracted from 256 and normalized, will serve as weighing coefficients. As with the difficult case, the background point of the binary input image is chosen in the center of the image. Concerning the artistic consequences are of choosing different background areas (i.e. the points where the distances start propagating) we refer to the literature (Verbeek and Verwer 1990). The result of the GWDT as used in the experiments is shown in Figure 8.5b.

As shown by Verbeek and Verwer, a contour plot of the resulting grey weighted distance transform (Figure 8.5c) gives - when viewed from a certain distance - an impression of the original weight coefficient image.

8.2.5 Median and minimum median root

The median root and the minimum median root operation have been defined in Section 5.5.5. The application area of the recursive median filter(s) is (shot) noise reduction with preservation of the edges in grey value images. An average case input can therefore be found in a normal image, corrupted with shot noise. To this end we will take a standard image called *bnoise*, as shown in Figure 8.6a. Application of the median root results in the image shown in Figure 8.6b. A comparison of the qualitative results of the median root applied with different updating methods is shown in Section 6.3.1. The minimum median root gives a unique fixed point as shown in Figure 8.6c.

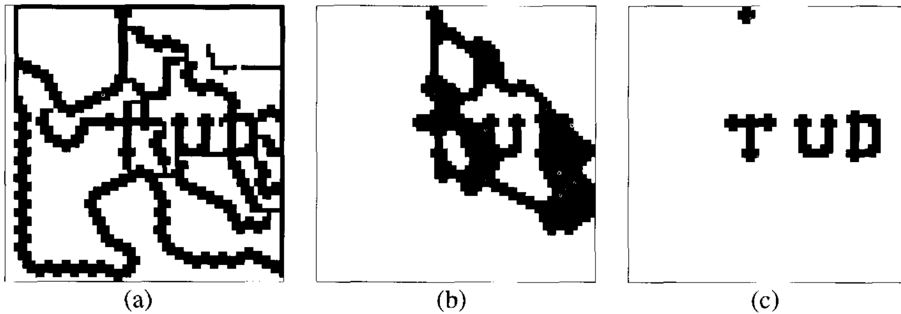


(a) (b) (c)

*Figure 8.6 Median and minimum median root average case: (a) original 64*64 image (b) result of applying the median root (c) result of applying the minimum median root.*

For a difficult case, it should be considered that the binary median root wipes away objects which are smaller than the so called smallest surviving objects. In a difficult case the median root will have to wipe away the most objects. To this end we constructed the image as shown in Figure 8.7a. This image contains several structures which are removed by the median root filter in a sequential way. The result of performing the median root using queue updating on the difficult case image is shown in Figure 8.7b. As shown in Chapter 6 other updating methods give different fixed point images. This should be kept in mind when comparing the performance of the different updating methods for the median root.

The minimum median root gives a unique fixed point image, as shown in Figure 8.7c.



(a) (b) (c)

*Figure 8.7 Difficult case for the median and minimum median root: (a) 64*64 original image, (b) result of applying the median root using queue updating, (c) result of applying the minimum median root (any updating method).*

8.2.6 Poisson equation

The Poisson equation can be used in image processing to (partly) recover an image which has been filtered by a Laplacian. Only one case is considered (see Figure 6.1). As an input to the Poisson equation a standard image which is filtered with a Laplacian is taken. This RNO uses floating point calculations, and is asymptotically stable in the fixed point sense. As a stop criterion for the Poisson equation, the order decay in the maximum error between the actual image and the best image possible with the floating point machine accuracy is taken. In our experiment, the maximum error should drop six orders of magnitude before the fixed point is reached.

An example of the qualitative performance of the Poisson equation is given in Chapter 6, Section 6.1.

8.3 Performance of the updating methods for the RNOs

In this section the experiments on the RNOs and their difficult and average case input images will be presented and discussed. The performance measure used is the updating effort E_u as defined in Section 8.1, i.e. the number of evaluations per image point. First the performance of the data dependent and the deterministic updating methods will be discussed. Second, the performance of the asynchronous updating method is measured for some of the problems. Finally, the experiments on the use of a combination of deterministic (simultaneous) updating within the square or linear processor array and data dependent (queue) updating on the scanning level are shown and discussed.

8.3.1 Deterministic and data dependent updating methods

The experimental results are presented in Table 8.1, which shows both the average and the difficult case performance in the number of evaluations per image point.

From this table it can be seen, that the two investigated data dependent methods are generally faster than the deterministic methods. The bucket updating is almost always faster than the queue updating. However, this method can only be applied to RNOs with a unique fixed point. The median root for instance, can not be calculated with it.

*Table 8.1 Comparison of deterministic and data dependent updating methods measured in updating effort (images are 64*64 pixels).*

	Object selection		SERP		Distance Transform		GWDT		Median Root		Minimum Med.Root	
	Av:	Diff:	Av:	Diff:	Av:	Diff:	Av:	Diff:	Av:	Diff:	Av:	Diff:
Data dependent:												
Queue (blob):	1.2	3.4	1.3	2.6	1.7	2.8	3.8	9.5	2.3	1.7	2.5	2.1
Bucket:	0.8	3.4	0.7	1.0	0.5	1.0	1.0	2.7	na	na	7.8	7.8
Deterministic:												
Double Serial:	3	34	23	31	4	4	14	10	13	126	15	128
Single Serial:	34	1010	32	32	13	33	42	280	12	89	14	92
Left Spiral:	22	2	37	32	13	33	53	37	15	101	16	101
Meander:	34	979	32	32	13	33	44	259	12	100	16	103
Line Serial:	41	1475	32	61	13	33	45	379	18	118	17	122
Simultaneous:	57	2018	32	61	14	33	53	501	26	184	24	187
Alternating(2):	41	1042	25	47	14	33	40	258	19	118	19	121
Alternating(4):	4	16	20	45	8	18	27	252	14	95	13	97
Right Spiral:	26	49	37	2	13	2	23	8	17	124	14	126

Av = Average case; Diff = Difficult case; na = not applicable

It is first noted that in some cases the difficult case was faster than the average case. This can be explained as follows.

- *Object selection*: The difficult case with left spiral updating is very fast because the image used in the difficult case is a left spiral.
- *Smallest enclosing regular polygon:* The difficult case with right spiral updating is very fast, because it allows the 'cross image' used for this difficult case to be filled in one pass.
- *Distance transform* and *grey weighted distance transform:* The difficult case with the right spiral updating method exactly follows the 'optimal updating path' to solve this problem.

An important point to notice is, that the ranking order between the deterministic updating methods depends on the following two points:

- The problem (i.e. the RNO)
- The difficult/average case.

It is obvious, that the simultaneous updating performs worst in all cases. Double serial updating seems to be quite fast for most of the problems

As a reference point to the reader, we also measured the performance of the updating methods on the discretisized Poisson equation. This was only done for one case, as defined in Section 8.2.6.

Table 8.2 Comparison of deterministic and data dependent updating methods for the Poisson problem in updating effort.

Data dependent:	
Queue (blob):	5569.9
Bucket:	na
Deterministic:	
Double Serial:	5483
Single Serial:	5477
Left Spiral:	5495
Meander:	5490
Line Serial:	8208
Simultaneous:	10924
Alternating(2):	5465
Right Spiral:	5595
SOR:	193

na = not applicable

An interesting result from this comparison (see Table 8.2) is that the deterministic updating methods now compete with the data dependent ones. The successive over relaxation (SOR) can only be applied to linear RNOs and from Table 8.2 it can be seen that it is the fastest updating method for the Poisson problem.

8.3.2 Asynchronous updating method

This method has been applied to only some of the problems. For each problem, the number of evaluations per pixel is measured as a function of the relative variation on the processing time (RVPT) per point. The RVPT is increased from 0 to 90% in steps of 10%. For every value, ten measurements are taken. The mean and the variance of these timings are shown in the Appendix, Section A.1, Table A.2 until Table A.7.

Figure 8.8 illustrates the performance of asynchronous updating, measured in number of clock cycles for the whole (asynchronous) processor array to perform the RNO. Every diagram shows the performance of the difficult case and the average case. For each case, a line is drawn through the mean measured number of clock cycles n, and a fuzzy area indicates where the maximum and minimum measured number of clock cycles are. The dark fuzzy area belongs to the difficult case, and the lighter fuzzy area to the average case.

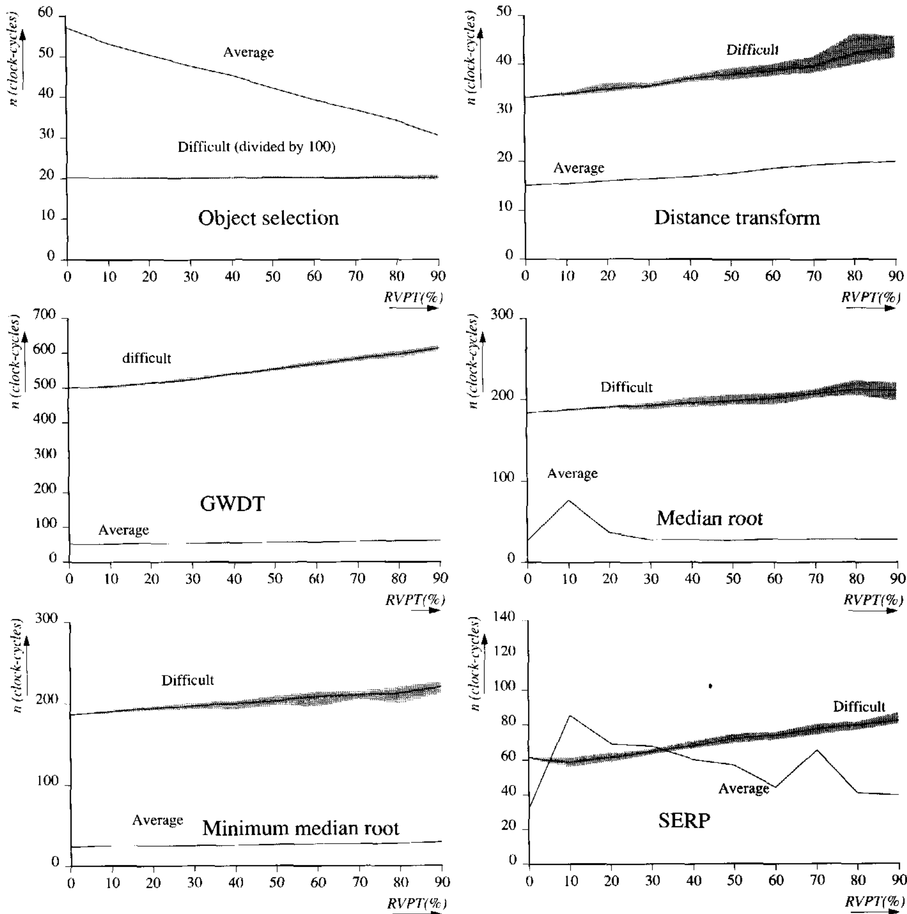


Figure 8.8 Asynchronous updating performance for the difficult and average cases of the indicated RNOs. For each RVPT 10 measures where taken. The lines indicate the mean of these 10 measures, and the fuzzy areas show where the other measures are.

For a good interpretation of the asynchronous updating behaviour which is shown in Figure 8.8 one should consider the following two effects which influence the speed with which a fixed point image is reached:

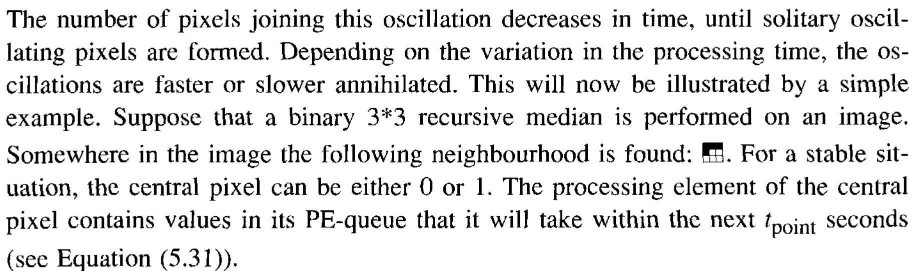
- *Speed-up.* Because of the variation in processing time per point, some points in the image are calculated *earlier* than they would have been if there was no variation on the processing time (i.e. for simultaneous updating). This is advantageous for neighbourhood operations where the final value of a point can be known if only a few neighbours take their correct value (i.e. for object selection).
- *Slow-down.* Points which have been calculated earlier than they would have been if there was no variation on the processing time have derived their pixel value from a neighbourhood which contains points which do not have their 'final' value (the value that they will have in the fixed point image). If more 'final' values become available, the number of pixel value 'corrections' increases, so that the total

processing time to reach a fixed point image increases. This effect is disadvantageous for all RNOs which require the correct values of all neighbours before a final value can be calculated on a point in the image.

Referring again to Figure 8.8 the following conclusions are drawn towards asynchronous updating:

- Only for *object selection* does the performance improve with increasing variation in the processing time per PE. This can be explained by the fact that there is no 'slow-down' effect because if a point becomes one, it will stay so without any need for correction.

- For the *distance transform*, the *grey weighted distance transform*, the *minimum median root* and the difficult case of the *smallest enclosing regular polygon* and of the *median root* the performance decreases with increasing variation on the processing time per PE. The 'speed-up' effect is completely annihilated by the 'slow-down' effect for increasing variation on the processing time, i.e. the number of necessary corrections of pixel values is always higher than the number of points calculated earlier than 'normal'.

- The average case of the *median root* behaves very badly for a relatively small variance in the processing time per PE, but behaves not too badly with increasing variation. For an explanation of this behaviour, one should consider that the median root is not monotonously decreasing like the minimum median root, but the median value may be higher or lower than the central pixel value. In addition to the 'normal' speed-up and slow-down effects, an 'oscillatory' slow-down effect which increases with decreasing variation of the processing time can be observed for some pixels. The existence of such oscillations can best be explained by considering a *binary* median root, and by looking at the *simulator* model in Figure 7.1. Clusters of pixels in a binary image are 'eaten away' at their edges, but expand again due to the 'feedback' (pixel values are continuously calculated and become available after different times) of ancient cluster pixels.
  The number of pixels joining this oscillation decreases in time, until solitary oscillating pixels are formed. Depending on the variation in the processing time, the oscillations are faster or slower annihilated. This will now be illustrated by a simple example. Suppose that a binary 3*3 recursive median is performed on an image. Somewhere in the image the following neighbourhood is found: . For a stable situation, the central pixel can be either 0 or 1. The processing element of the central pixel contains values in its PE-queue that it will take within the next t_{point} seconds (see Equation (5.31)).
  Due to 'historic' reasons, the 'future' values of the central pixel may be as shown in Figure 8.9a, i.e. the central pixel will become one at time t and zero again at time $t + \Delta t$. If the central pixel becomes one at time t, this will 'ripple' through the PE within $t_{average}(1 + \mathrm{RVPT} \cdot \rho_1)$, where ρ_1 is a random value, uniformly distributed between -1 and +1 (see Equation (5.31)).

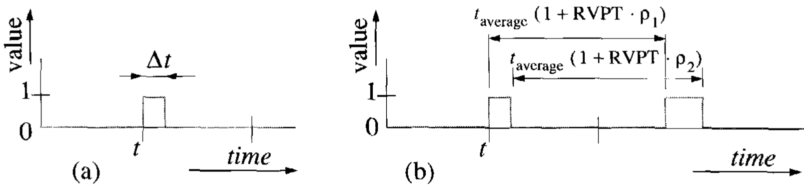


Figure 8.9 Behaviour of the values in a point which is updated using asynchronous updating: (a) values that the point will take within the next $t_{average}$ seconds, (b) calculation when the changes within the PE are rippled through and cause new changes.

The effect of the central pixel becoming zero at time $t + \Delta t$ will ripple through the PE within $t_{average}(1 + \mathrm{RVPT} \cdot \rho_2)$ (see Figure 8.9b). The oscillation will be annihilated if $t + \Delta t + t_{average}(1 + \mathrm{RVPT} \cdot \rho_2) \le t + t_{average}(1 + \mathrm{RVPT} \cdot \rho_1)$, i.e. if:

$$\frac{\Delta t}{t_{average} \cdot \mathrm{RVPT}} \le (\rho_2 - \rho_1) .$$

This will happen faster, if the RVPT is larger, i.e. with increasing RVPT, the oscillations will be annihilated faster.

At this point we raise the question whether this oscillatory effect stems from using the simulator as described in Chapter 7, or if this effect can also happen in a real asynchronous SPA? The form of the signals propagating through a PE in a real asynchronous SPA will be more smooth shaped than the digital representation in the simulator. Race and noise conditions within the physical PE not only influence the width of the signal within it but for very short signals also the height. If a signal becomes too short (i.e. it is almost caught up by another signal), than the height decreases, and it is not seen as a one any more. In that case annihilation has occurred.

- Another thing to notice is the behaviour of the average case for the *smallest enclosing regular polygon*. The average case problem is specified in Section 8.2.2. From this specification it can be seen, that the smallest enclosing regular polygons of some of the objects in the image overlap. The final solution of this operation therefore consists of a smallest enclosing regular polygon around those 'overlapping' objects. However, the stochastic nature of the asynchronous updating method makes it unpredictable *at what time* the objects overlap. For deterministic and data dependent updating methods, on the other hand, this point in time is well defined. The large variance in the performance for the smallest enclosing regular polygon average case can therefore be explained by the stochastic nature of the asynchronous updating method.

8.3.3 The combined data dependent and deterministic updating

For this updating method, an image is divided into scans (subimages). As explained in Section 7.4, the scans can be updated with the queue updating technique. Within a scan, the simultaneous updating (i.e. iteration) is used. The latter is done, so that the results of the combined updating method can directly be translated to performance for the LPA and SPA architecture groups (see Section 8.4).

Two series of scan sizes were taken: one which could in principle be done by the LPA, and one which could be done by an SPA. These series are shown in Table 8.3. Note, that the combined queue/simultaneous updating for a scansize of 1*1 PE is actually just queue

updating (on the pixel level) with a not-too-smart initial queue. Also, as we are working with 64*64 images, the scan size of 64*64 PEs for the SPA performs the same as normal simultaneous updating on the pixel level.

Table 8.3 Distribution of P PEs over scans for the LPA and SPA

P	LPA	SPA
1	1*1	1*1
2	2*1	2*1
4	4*1	2*2
8	8*1	4*2
16	16*1	4*4
32	32*1	8*4
64	64*1	8*8
128	128*1	16*8
256	256*1	16*16
512	512*1	32*16
1024	1024*1	32*32
2048	2048*1	64*32
4096	4096*1	64*64

Queue updating requires specification of how the queue is to be initialised (i.e. which points are put initially in the queue). We have chosen to put the first scan in the queue initially and start the combined queue/simultaneous updating. The image is then walked through raster scan wise, i.e. from the top left to the bottom right scan to check if any scans were not processed. If such a scan is encountered, then it is put in the queue, and the combined queue/simultaneous updating is performed again.

The numerical results are shown in Table A.1, Appendix A. The performance (in updating effort) of the combined updating method for the difficult and average cases of the six non-linear RNOs is illustrated in Figure 8.10a-f.

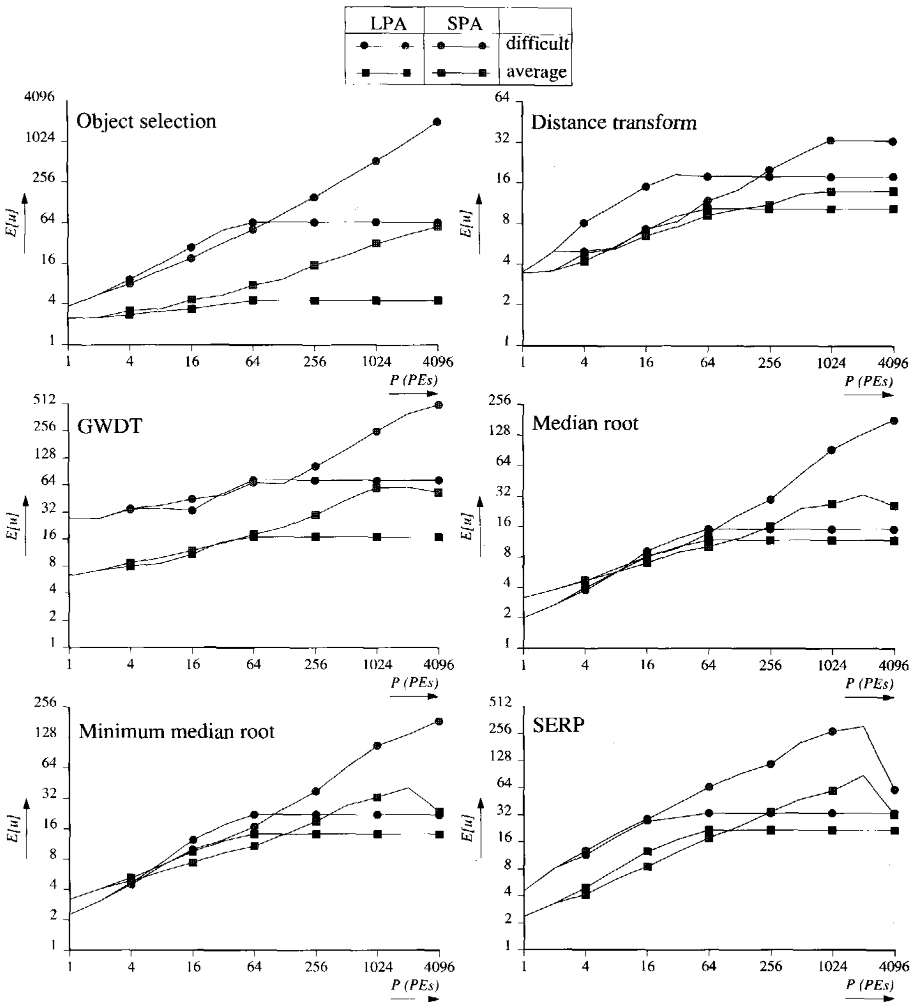


Figure 8.10 Updating effort of SPA and LPA for the combined deterministic and data dependent updating for the indicated RNOs (see Table 8.3 for the measure points).

From the comparison shown in Figure 8.10, the following can be learned:

- The number of evaluations per pixel increases with increasing parallelism for the SPA, but reaches a limit at P=64 for the LPA. The latter is caused by the fact that increasing the number of PEs above 64 for an LPA operating on a 64*64 image does not increase the updating effort. The decrease in evaluations per pixel for the SPA at the end of the diagrams (i.e. from P=2048) actually takes place, when there is no combined updating method anymore; at that point there is only simultaneous updating.
- For RNOs, the LPA takes the least number of evaluations per pixel for large number of available PEs. This means, that the LPA works more efficiently than the SPA for large number of PEs.

- For the object selection (average case) and the GWDT, the LPA performs best overall.
- The SPA performs most efficiently for a small number of PEs, in all other cases.

8.3.4 The efficiency of updating methods revisited

With regard to the efficiency of the updating methods (the reciprocal of the updating effort) used in the experiments, the following observations are made:

- *Difficult and average case.* The efficiency ranking order between the updating methods differs for the difficult and average case. For most of the RNOs these differences in ranking take place with one of the spiral updating methods (e.g. for the object selection the ranking order differs between the double serial and the left spiral updating). This can be explained by noting the correlation between the optimal updating path for these RNOs, and the updating path that the spiral methods follow. The differences in ranking order between the difficult and average case for the median root can be explained by the fact that the median root is a non-unique RNO in the fixed point sense, so that each updating method gives a different fixed point image.
- *Ranking order.* In Chapter 5, a suggestion for the ranking order of the deterministic updating methods was made on the basis of the recursive neighbour fraction Q. To compare this predicted ranking order we take the conclusions with the previous observation into account by leaving out the median root RNO and the two spiral updating methods. The ranking order of the other RNOs and updating methods (see Table 8.1) is 'almost' in correspondence with the one predicted by the updating fraction in Table 5.2. Simultaneous updating is slowest, followed by line serial updating and then the other updating techniques. For some cases line serial updating happens to be faster than, for example, alternated updating. This can not be explained by the updating fraction. We conclude therefore, that the updating fraction predicts some coarse ranking order between the updating methods, but is not capable of predicting the differences in ranking order related to the differences in RNOs and in the data with which the RNOs are processed.
- *Asynchronous updating.* As was already foreseen in Chapter 5, the performance of the asynchronous updating is either increasing or decreasing with increasing variance on the processing time per pixel. This depends on the character of the RNO, i.e. is one neighbour change enough to calculate a good and final value on a point (object selection), or are all points necessary (most of the other RNOs)?
- *Efficiency.* The best overall efficiency is gained with the data dependent updating methods. Between these two methods, the bucket updating is best.

Although the updating effort in number of evaluations per pixel of the combined updating method is an interesting measure for comparing the performance of other updating methods, it does not say how fast the methods are. This will be compared, however, in Section 8.4 for those updating methods which can be implemented by the investigated architecture groups.

8.4 Performance of the architecture groups for the RNOs

Using the results of Chapter 7, the relation between the efficiency of the updating methods and the architecture groups on which they are implemented can be made. First we will note which updating method is best with respect to which architecture:

- *Pipeline*. The best method for the pipeline is definitely the *serial* updating. The pipeline is unfortunately not able to implement any data dependent updating method.
- *Square Processor Array*. The most efficient updating method which can be implemented on this type of low level image processing architecture is the simultaneous one and sometimes asynchronous updating. If, however, the image is larger than the array, a data dependent updating method can best be used on the scanning level.
- *Linear Processor Array*. Just as with the SPA, the LPA can only implement simultaneous (or possibly asynchronous) updating on the array level. The LPA, however, always has to scan. On the scanning level, line serial updating can be used. Most efficient would be the use of a data dependent updating method.

The three architectures are now compared more specifically for the RNO test cases. For this goal, the total processing time in clock cycles as a function of the number of PEs is derived, for all three architectures. With respect to the formulas used to determine the processing time of the SPA, LPA and PL, reference is made to chapter 4, Equations (4.3), (4.5) and (4.9). These equations give the number of clock-cycles needed to process an LNO of l steps for an $N*N$ image using P PEs. These LNO formulas can be used for RNOs, if the algorithm length l is replaced by the number of passes across the image (for deterministic updating methods). In the examples of Section 8.3, the following settings were used:

- The image size $N = 64$
- The number of processors P varies from P=1 to P=4096 (see Table 8.1)
- The algorithm length l (equivalent to the number of passes across the image and also to the updating effort if deterministic updating methods are used) differs for the SPA, LPA and PL.

For the PL, l is taken to be the number of evaluations per pixel for single serial updating (see Table 8.1). The number of evaluations per pixel for double serial updating is taken from this table if frame recirculation is required (i.e. if $l > P$). This is formalised in Equation (8.1):

$$l_{\mathrm{PL}} = \begin{cases} E_{u\mathrm{DoubleSerial}} \le P & \rightarrow E_{u\mathrm{SingleSerial}} \\ E_{u\mathrm{DoubleSerial}} > P & \rightarrow E_{u\mathrm{DoubleSerial}} \end{cases} \tag{8.1}$$

For the SPA and the LPA, l is equivalent to the number of evaluations per pixel E_u for the combined deterministic and data dependent updating (see Figure 8.10):

$$l_{\mathrm{LPA}} = l_{\mathrm{SPA}} = E_u \tag{8.2}$$

With respect to the data I/O time, we consider two cases:

- *No data I/O*.
- *Data input*. The LPA and SPA use column parallel data input, and the PL uses raster scan data input.

First, the case is considered without data I/O. It is presumed, that the image on which the RNO has to be performed is already in the array memory (for the LPA and the SPA). The time is calculated from the moment the first pixel is processed, to the moment the first result pixel becomes available. The results of these experiments are visualized in Figure 8.11.

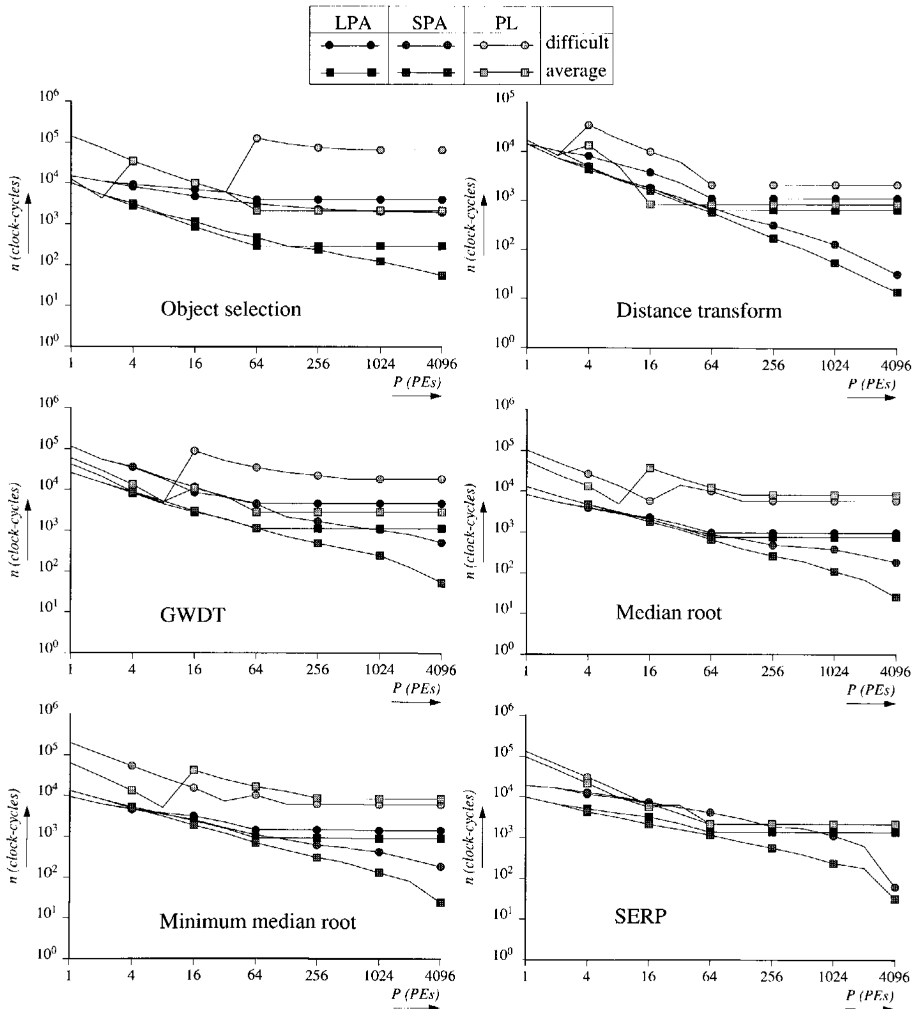


Figure 8.11 Comparison of processing time for PL, SPA and LPA without data input time, using combined deterministic and data dependent updating.

From the experiments it is clear, that in nearly all cases the full-sized (i.e. $P = N^2$) SPA is the fastest. At that point, it uses simultaneous updating. SPAs and LPAs which use the combined simultaneous/queue updating method are always faster than one single processor performing queue updating. In all cases the processing time for the SPA and the LPA monotonically decreases with an increasing number of PEs.

However, the PL architecture does not behave monotonically with increasing number of PEs. The non-monotonic behaviour occurs at the points where the pipeline has to start do-

ing frame recirculation. Also at this point the algorithm length is taken from the double serial updating technique instead of the single serial updating. Only frame-recirculating pipelines can do double serial updating.

The second case that will be considered is the one with data input. The results for these experiments are shown in Figure 8.12.

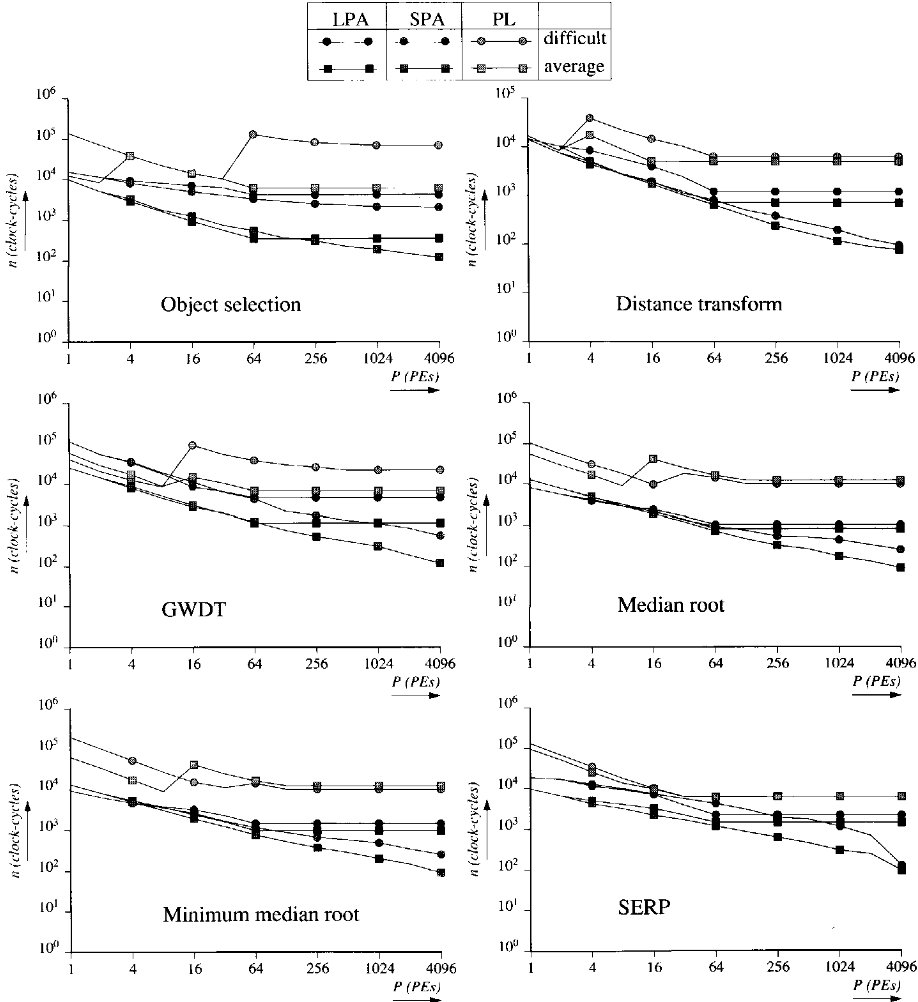


Figure 8.12 Comparison of processing time for PL, SPA and LPA including data input time, using combined deterministic and data dependent updating.

Due to the fact that the pipeline uses raster scan data I/O (i.e. this is of order N^2 in time), it performs definitely worst overall. The SPA performs best overall for all the other cases, except for the average case of object selection.

8.5 Conclusions

The performance in updating effort and speed for the SPA, LPA and PL architecture

groups has been investigated with respect to their use in processing RNOs. Although LPAs are generally more efficient than the SPAs, the latter are generally faster in performing RNOs.

The updating methods are divided into three classes: deterministic, data dependent and stochastic. With respect to the *deterministic* updating methods it is found that the simultaneous updating generally performs worst, and the double serial best in a number of cases. Although suggested in Chapter 5, ranking of the other deterministic updating methods cannot be done consistently between the six non-linear RNOs which were treated. For several combinations of RNO and updating method, the processing of a difficult case went faster than the processing of the average case.This can be explained by the fact that one updating method (the left spiral updating) happened to 'match' exactly with the 'optimal path' to process these RNOs.

The *data-dependent* updating methods perform one or more orders in magnitude better than the deterministic ones for most of the investigated RNOs. However, the parallelisms in the SPA, LPA and PL can not be used to implement the recursive, queue or bucket updating techniques. It is known from the literature, that special architectures can be built which do implement the parallelism in these methods efficiently, but the emphasis of such a machine is clearly not on low-level image processing in general, so that a treatment of such an architecture is beyond the scope of this thesis (Jonker et al.1988a).

Although the data-dependent updating methods cannot be directly implemented on the investigated architecture groups, a *combination* of a deterministic updating method on the array level and a data-dependent updating method on the scanning level can be done for the LPA and the PL. Simultaneous updating is chosen as a deterministic method, and queue updating as a data-dependent method. The image should be larger than the array. It is shown, that the updating effort with which these architectures perform the combined updating method monotonically decreases with increasing parallelism. For some cases, the updating effort of a relatively large scanning SPA using the combined updating method decreases even below the updating effort of a full-sized SPA using simultaneous updating.

From the *stochastic* methods, the asynchronous updating has been investigated. This method is, for example, used on the CLIP4 processor array (see Chapter 2). It can be used on an SPA which has unlatched connections between inputs and outputs. The effect of increasing the variance on the mean throughput time was shown to decrease the processing time only for the difficult case of object selection. Increasing the relative variation on the processing time generally leads to an increase in overall processing time. The behaviour of the average case for the median root is different, in that a peak in the overall processing time was observed for a 10% relative variation on the processing time, followed by a monotonic decay of the processing time for larger variation.

A performance comparison for increasing number of PEs has been done between all three architectures, using their respectively best updating methods. For the PL this is the serial (or double serial when frame recirculation has to be used) updating method. For the LPA and the SPA the combined updating methods were used. Whether or not data input time is taken into account, the SPA performs best in almost all cases. There are in general no large differences in performance between the LPA and the PL for the same number of PEs, if the width of the image is larger than the number of PEs. Only the SPA can benefit from a larger number of available PEs.

Chapter

9 Conclusions

The aim of this research was to compare low-level image processing architectures as to their efficiency and speed for low-level image processing. The strategy taken in this research has been as follows. After an investigation of the different possible techniques to compare image processing architectures, the operations analysis method was selected. For this method, low-level image processing architectures are divided into groups, and these architecture groups are compared for groups of low-level image processing operations. The comparison was based on quantitative as well as qualitative factors.

9.1 Evaluation of results

On the basis of architecture description parameters and, in correspondence with previous comparisons, three architecture groups are chosen for the comparison: the Square Processor Array (SPA), the Linear Processor Array (LPA) and the Pipeline (PL). These groups are shown to represent most available low-level image processing architectures.

Image processing operations are divided on the basis of operation features into eight groups. This division proves to be helpful, as it reveals a previous lack of comparison for some groups (i.e. object, recursive neighbourhood, global, geometric and statistical operations).

A comparison of the three architecture groups was performed on the basis of the quantitative performance measures (i.e. speed and efficiency) for a subset of the operation groups (i.e. point, local neighbourhood, object, global, geometric and statistical scalar operations) and on the basis of the qualitative factors flexibility, data I/O possibilities and programmability consequences. With the assumption of row parallel data I/O for the LPA and raster scan data I/O for the PL, the LPA is always better than the PL for the same number of PEs. The SPA can compete with the LPA if a relatively large number of PEs is available, or if the scanning overhead for the SPA is minimized to one.

No comparisons are published on the basis of the recursive neighbourhood operations (RNOs). However, it was shown in this thesis, that they can be a basis for the implementation of some interesting object and global operations. We showed the equivalence between several object/global operations on one hand, and corresponding RNOs on the other hand. Using an RNO instead of the equivalent OO or GlO is interesting from the viewpoint of image processing. Images are strongly locally correlated multidimensional signals (which also explains why the LNOs seem to constitute such a large part of available low-level image processing operations). Because of this, architectures for low-level image processing have been built which are also strongly locally oriented. However, object and global operations require the transport of information over larger than local distances, which conflicts with a locally oriented architecture. By implementing the object and global operations with their equivalent strongly locally oriented RNOs, the initial inefficiency of locally oriented low-level image processing architectures for these OOs and GlOs is removed. This

realisation led to the decision to further investigate RNOs.

Theoretical research into the nature of the RNOs in general leads to the introduction of stability and uniqueness in the fixed point sense. Depending on the updating method used to calculate an RNO, there may be zero, one, or more than one fixed point image to which the RNO converges. Some theorems have been introduced and proved, which can be used to find out if an RNO is stable and/or unique in the fixed point sense. The theory as it has been established so far, has been demonstrated to work on the RNOs which are presented in this thesis.

It has been shown how the fixed point theory can be used to convert recursive stack filters, which may be unstable or non-unique, into stable and unique RNOs, which are still stack filters.

It has been shown mathematically, that some RNOs can be rewritten as object or global operations.

An investigation on non-linear RNOs led to the discovery of a smallest enclosing polygon algorithm. This is an RNO which draws a polygon around the objects in an image, where the number of corners is determined by the size and shape of the local neighbourhood.

Research and experiments with updating methods reveal three major types: the deterministic, the data dependent, and the stochastic updating methods. Experiments revealed that the asynchronous updating technique behaves in a problem dependent way. On some RNOs it may give slight speed-ups for slight variations in the processor speed over the simultaneous updating technique. However, larger variations usually lead to slower processing.

It is shown experimentally, that the data dependent queue or bucket updating methods are superior in performance to the others examined. However, the implementation of these methods in the parallel architectures for low-level image processing contradicts their SIMD and parallel nature. As is pointed out in Chapter 7, SPAs and LPAs can only do simultaneous updating on the array level. For small SPAs or LPAs, a combination of the inefficient simultaneous updating method on the array level, and the efficient data dependent queue updating method on the scanning level was thought to be promising. If an architecture has fewer PEs than there are pixels in the image, then the combined data-dependent/deterministic updating is to be preferred above any other updating method. A PL using serial or double serial updating can only compete with the SPA or LPA using the combined data-dependent/deterministic updating if the number of PEs is relatively small, no data input is taken into account, and only for some RNOs. The experiments also revealed, that it is always faster to do any of the investigated RNOs using the deterministic simultaneous updating with a full-sized SPA, than trying to calculate it with the more efficient combined method using less PEs.

Concerning the performance of the investigated architecture groups for low-level image processing in general, our results indicate that the LPA seems to offer the best overall speed/efficiency combination. The SPA can only be competitive with the LPA if a better scanning technique for the SPA is developed, or if the number of available PEs is large. The PL is in general outperformed by the SPA or the LPA.

9.2 Perspective

The chosen comparison strategy reveals that low-level image processing architectures need to be compared more thoroughly for the following operation groups:

- those *Object Operations* which can not be implemented using RNOs (although we do not know examples of this group),
- the *Global Operations* which can not be implemented using RNOs (e.g. Fourier transform, Hough transform),
- *Geometric Operations,*
- *Statistical Vector Operations.*

Further research should be done to find out if other object and global operations can be rewritten as RNOs. If this is possible, then calculating them as an RNO on a low-level image processing architecture will probably be faster and more efficient than trying to calculate the corresponding object or global operations as such on the same architecture.

Although the scope of this thesis is on existing architectures for low-level image processing in general, special parallel architectures can be developed specifically for the calculation of RNOs. One mention has been found in the literature of an architecture that implements the bucket updating technique for a specific case (Jonker et al.1988a).

We suggest further research in the possibilities of the LPA equipped with local addressing autonomy to do an updating method for the RNOs which is more data dependent. Such a method may very well further improve the performance of the LPA over other architectures.

We also suggest investigating the performance improvements gained by forming a pipeline of LPAs, when more PEs than necessary for one LPA are available. Such an extension has already been proposed for the novel Centipede architecture (Schmitt and Wilson 1989).

The emphasis concerning RNOs has been on the non-linear ones in this thesis. The application of the theory and results for non-linear RNOs in other disciplines then low-level image processing is in principle possible, but has not yet been done.

Chapter

10 Bibliography

Arce GR, Crinon RJ (1984) Median filters: analysis for 2 dimensional recursively filtered signals. Proc. int. conf. on ASSP, pp.20.11.1 - 20.11.4

Basille JL (1989) Personal communication.

Batcher KE (1980) Design of a massively parallel processor. IEEE Trans. Comput. C-29(9):836-840

Blum H (1967) A transformation for extracting new descriptors of shape. In: Wathen-Dunn W (Ed) Models for the perception of speech and visual form, MIT Press, Cambridge, USA, pp.362-380.

Boomgaard R van den (1989) Threshold logic and mathematical morphology. In: Cantoni V, Cordella LP, Levialdi S, Sanniti di Baja G (eds) Progress in image analysis and processing. World Scientific, London, pp.111-118

Boyce WE, DiPrima RC (1965) Elementary differential equations and boundary value problems. John Wiley, New York.

Buurman J, Duin RPW (1988) Implementation and use of software scanning on a small CLIP4 processor array. In: Kittler J (ed) Lecture notes in computer science 301: Pattern Recognition, Springer-Verlag, London, pp.269-277

Cantoni V, Levialdi S (1983) Matching the task to an image processing architecture. Computer vision, graphics and image processing 22(2):301-309

Cantoni V (1986) Hierarchical sytstems: architectural features. In: Cantoni V, Levialdi S (eds) Pyramidal systems for computer vision, NATO ASI series, pp.21-39

Cantoni V, Levialdi S (1987) PAPIA: A case history. In: Uhr L (ed.) Parallel computer vision. Academic press, pp.3-13

Chan DSK (1976) A novel framework for the description of realization structures for 1-D and 2-D digital filters. IEEE Electronics and space convention record, pp.157(A-H).

Clarke KA, Ip HHS (1982) A parallel implementation of geometric transformations. Pattern recognition letters 1:51-53

Cypher R, Sanz JLC (1989) SIMD architectures and algorithms for image processing and computer vision. IEEE transactions on acoustics, speech, and signal processing, ASSP-37(12)2158-2174.

Danielsson PE (1980) Euclidean distance mapping. Computer Graphics and Image Processing 14:227-248

Danielsson PE, Levialdi S (1981) Computer architectures for pictorial information systems. IEEE Computer 14(11):53-67

Davis AL, Keller RM (1982) Data flow program graphs, IEEE Computer 15(2):26-41

Davis R, Thomas D (1984) Systolic array chip matches the pace of high speed processing. Electronic design,Oct:207-218

Dekker S, Jonker PP, Groen FCA(1987) Distance transforms with data flow techniques. Proceedings of the 6th Aachen symposium on signal theory, Springer Verlag, Berlin, Germany, pp.269-272.

Döhler HU (1989) Generation of root signals of two dimensional median filters. Signal processing 18:269-276

Dorst L (1986) Pseudo-Euclidean skeletons. Proc. of the 8th int. conference on pattern recognition, Paris, France, IEEE computer society press, pp.286-288

Dorst L, Verbeek PW (1986) The constrained distance transformation: a pseudo-Euclidean recursive implementation of the Lee algorithm.

Duclos P, Boeri F, Auguin M, Giraudon G (1988) Image processing on a SIMD/SPMD architecture: OPSILA, Proc. of 9ICPR, Rome, pp.430-433

Dudgeon DE, Mersereau RM (1984) Multidimensional digital signal processing. Prentice Hall, USA.

Duff MJB (1982) The CLIP4. In: Fu KS, Ichikawa T (eds) Special computer architectures for Pattern Recognition. CRC-Press, pp.65-86

Duff MJB (1986) How not to benchmark image processors. In: Uhr L, Levialdi S, Preston K, Duff MJB (eds) Evaluation of multicomputers for image processing, pp.3-12

Duin RPW, Haringa H, Zeelen R (1986) Fast percentile filtering. Pattern recognition letters 4:269-272

Duin RPW, Jonker PP (1986) Processor arrays versus pipelines for cellular logic image operations. In: Young IT et al.(eds) Signal Processing III:Theories and Applications, Elsevier Science Publishers, pp.1339-1342

Duin RPW, Jonker PP (1988) Processor arrays compared to pipelines for cellular image operations. In: Uhr L (ed) Multicomputer Vision. Academic Press, pp.151-169

Duin RPW, Komen ER (1989) Massively parallel architectures for cellular logic image processing. In: Cantoni V, Cordella LP, Levialdi S, Sanniti di Baja G (eds) Progress in image analysis and processing. World Scientific, London, pp.643-657.

Feng TY (1977) Parallel processors and processing. ACM computing surveys 9(1):-

Fisher AL, Highnam PT (1985) Real-time image processing on scan line array processors. IEEE Computer Society Workshop for Pattern Analysis and Image Database Management, Miami Beach, Florida:484-489

Fisher AL, Highnam PT, Rockoff TE (1987) Architecture of a VLSI SIMD processing element. Proc. IEEE int. conf. on Computer Design: VLSI computer process., oct 5-8, pp.324-327

Fitch JP, Coyle EJ, Gallagher NC Jr (1984) Median filtering by threshold decomposition. IEEE transactions on acoustics, speech, and signal processing, ASSP-32(6):1183-1188

Flynn MJ (1972) Some computer organizations and their effectiveness. IEEE transations on computers C-21(9):948-960

Forchheimer R, Ödmark A (1983) A single chip linear array picture processor. Proc. of SPIE, vol.397, pp.425-430

Forshaw MRB (1987) Array architectures for image processing: 1 Connection matrices, 2 Adjacency matrices. Reports 87/3 and 87/4, Image Processing Group, University college London, England.

Fountain TJ (1983) A survey of bit-serial array processor circuits. In: Duff MJB (ed) Computing structures for image processing, Academic Press, London, pp.1-14

Fountain TJ (1986) Array architectures for iconic and symbolic image processing. Proceedings of the eighth international conference on pattern recognition, Paris, France, pp.24-33

Fountain TJ (1987) Processor Arrays, Architecture and Applications. Academic Press, London.

Fountain TJ, Matthew KN, Duff MJB (1988a) The CLIP7A image processor. IEEE transactions on pattern analysis and machine intelligence PAMI 10(3):310-319

Fountain TJ (1988b) Introducing local autonomy to processor arrays. In: Freeman H (ed.) Machine vision: algorithms, architectures and systems. Academic press, pp.31-56

Fountain TJ (1988c) An analysis of methods for improving long-range connectivity in meshes. In: Kittler J (ed) Lecture notes in computer science 301: Pattern Recognition, Springer-Verlag, London, pp.259-268

Fujita Y, Iwashita M, Temma T (1990) A dataflow image processing system TIP-4. In: Cantoni V, Cordella LP, Levialdi S, Sanniti di Baja G (eds) Progress in image analysis and processing. World Scientific, London, pp.734-741

Fung LW (1977) A massively parallel processing computer. In: Kuck DJ et. al. (eds) High-Speed Computer and Algorithm Organization, Acadamic Press, New York, pp.203-204

Garda P, Reichart A, Rodriguez H, Devos F, Zavidovique B (1988) Yet another mesh array smart sensor? Proc. of 9ICPR, Rome, pp.863-865

Gerritsen FA, Aardema LG (1981) Design and use of DIP-1: a fast, flexible and dynamically microprogrammable pipelined image processor. Pattern recognition 14:319-330

Gerritsen FA (1982) Design and implementation of the Delft image processor DIP-1; dissertation, Pattern Recognition Group, Applied Physics Department, Delft University of Technology

Goles E, Olivos J (1981) Comportement periodique des fonctions a seuil binaires et applications. Discrete applied mathematics 3:93-105

Graham MD, Norgren PE (1980) The diff3 analyzer: a parallel/serial golay image processor. In: Onoe M, Preston K, Rosenfeld A (eds) Real-time medical image processing, pp 163-182

Groen FCA, Jonker PP, Duin RPW (1988) Hardware versus software implementations of fast image processing algorithms. Jain AK (ed) Real-Time Object Measurement and Classification, Springer-Verlag, Berlin, pp.73-91

Händler W (1977) The impact of classification schemes on computer architecture. Proceedings of the 1977 int. conf. on parallel proc., pp.7-15

Haralick RM (1981) Some neighborhood operators. In: Onoe M, Preston K Jr, Rosenfeld A (eds) Real-Time/Parallel Computers: Image Processing, Plenum Press, New York. pp 11-35

Haralick RM, Sternberg SR, Zhuang X (1987) Image analysis using mathematical morphology: Part 1, IEEE Pattern Analysys and Machine Intelligence PAMI-9,(4):532-550

Hockney RW, Jesshope CR (1981) Parallel Computers. Great Britain.

Hockney R (1987) Algorithmic phase diagrams. IEEE transactions on computer C-36(2):231-233

Huang KS, Jenkins BK, Sawchuk AA (1989) Binary image algebra and optical cellular logic processor design. Computer Vision, Graphics and Image Processing 45(3):295-345

Hunt DJ (1981) The ICL DAP and its application to image processing. In: Duff MJB, Levialdi S (eds) Languages and architectures for image processing, Academic Press, London, pp.275-282

INMOS (1988) Transputer technical notes, Prentice Hall, New York.

Iwashita M, Temma T, Mizoguchi M, Matsumoto K, Shuto M, Hanaki S (1986) A data-driven VLSI image processor (ImPP) -- a LSI version of TIP. In: Uhr L, Levialdi S, Preston K, Duff MJB (eds) Evaluation of multicomputers for image processing, pp.301-318

Jenkins RE, Gilbert Lee Jr D (1987) An application specific coprocessor for High-Speed Cellular Logic Operations. IEEE Micro, pp.63-70

Jonker PP, Duin RPW (1985) Considerations on a VLSI architecture for Cellular Logic Operations. Proceedings of the IEEE Computer Society workshop on Computer Architecture for Pattern Analysis and Image Database Management, Miami Beach, FL: 453-462

Jonker PP, Dekker ST, Verwer BJH (1988a) A hardware architecture for robot path planning. IAPR workshop on computer vision, special hardware and industrial applications, 12-14 oktober, Tokyo

Jonker PP, Nouta R, Kraaijveld MA, Schot CA (1988b) The realization of a VLSI Circuit for fast binary image processing using a new VLSI design system. In: Herrmann OE, Beijnum BJF van (eds) Lecture Notes of the Nelsis-Project. TU-Twente, pp62-83

Jonker PPJ, Komen ER, Duin RPW, Kraaijveld MA (1989) Cellular logic processing elements for real time robot vision. To be published.

Jonker PP, Zeppenfeldt F, Dekker ST, Duin RPW (1990) Image processing using data flow based digital signal processors. Workshop on parallel processing BARC, Bombay, India.

Justusson BI (1981) Median filtering: statistical properties. In: Huang TS (Ed) Two dimensional digital signal processing II: transforms and median filters, New York, Springer-Verlag

Juvin D, Basille JL, Essafi H, Latil JY(1988) Sympati 2, a 1.5 D processor array for image application. In: EUSIPCO Signal processing IV: Theories and applications, North-Holland, pp.311-314.

Kent EW, Schneier M, Lumia R (1985) PIPE - Pipelined image processor engine. Journal of parallel and distributed computing 2:50-78.

Kimmel MJ, Jaffe RS, Manderville JR, Lavin MA (1985) MITE: Morphic image transform engine, an architecture for reconfigurable pipelines of neighbourhood processes. IEEE Computer Society Workshop for Pattern Analysis and Image Database Management, Miami Beach, Florida:493-500

Kogge PM (1981) The Architecture of pipelined Computers. Hemisphere Publ.Corp.,USA

Komen ER, Duin RPW (1989) CLPE's for grey value processing. To be published.

Kraaijveld MA, Jonker PP, Nouta R, Duin RPW (1986) The VLSI realisation of a binary-image processor. EUSIPCO Signal processing III: Theories and applications, North-Holland, pp.1231-1234.

Lea RM (1988) ASP: A cost-effective parallel microcomputer. IEEE Micro pp:10-29

Leveque RJ, Trefethen LN (1988) Fourier analysis of the SOR iteration. IMA journal of numerical analysis 8:273-279

Levialdi S (1988) Computer architectures for Image analysis. Proc. of 9ICPR, Rome pp.1148-1158

Lindskog B (1988) PICAP3, An SIMD architecture for multi-dimensional signal processing; Dissertation No. 176. Dept.of El. Eng. Linköping University, Sweden

Lorrain P, Corson DR (1970) Electromagnetic fields and waves. Freeman and company, USA.

Lougheed RM, McCubbrey DL, Sternberg SR (1980a) Cytocomputers: Architectures for parallel image processing. Proceedings of the IEEE workshop picture data description and management, CA, pp. 281-286

Lougheed RM, McCubbrey DL (1980b) The cytocomputer: a practical pipelined image processor. Proc. of 7th annual international symposium on computer architecture, pp.271-277

Lougheed RM (1987) Advanced image-processing architectures for machine vision. Preprint from SPIE conference on image pattern recognition -- algorithm implementations, techniques and technology: Critical review of technology. (vol. 755), january.

Manry MT, Aggarwal JK (1974) Picture processing using one dimensional implementations of discrete planar filters. IEEE transactions on acoustics, speech and signal processing ASSP-22(3):164-173

Maragos P, Schafer RW (1986) Morphological skeleton representation and coding of binary images. IEEE transactions on acoustics, speech, and signal processing, ASSP-34(5):1228-1244

Maragos P (1987a) Tutorial on advances in morphological image processing and analysis. Optical engineering 26(7):623-632

Maragos P, Schafer RW (1987b) Morphological filters - Part I: Their set-theoretic analysis and relations to linear shift-invariant filters. IEEE transactions on acoustics, speech, and signal processing, ASSP-35(8):1153-1169

Maragos P, Schafer RW (1987c) Morphological filters - Part II: Their relations to median, order-statistic, and stack filters. IEEE transactions on acoustics, speech, and signal processing, ASSP-35(8):1170-1184

Maragos P (1989) A representation theory for morphological image and signal processing. IEEE transations on pattern analysis and machine intelligence PAMI-11(6):586-599

Maresca M, Hungwen L, Sheng MMC (1989) Parallel computer vision on polymorphic torus architecture. Machine vision and applications 2:215-230

McCormick BH (1963) The Illinois pattern recognition computer - ILLIAC III. IEEE transactions on electronic computers EC-12(6)791-813

Montenari U (1968) A method for obtaining skeletons using a quasi eucledian distance. Journal of the ACM 15(4):600-624

Otto GP (1984) Algorithms for image processing on the CLIP4 cellular array processor. Thesis, University College London, London, Great Britain.

Pass S (1985) The GRID parallel computer system. In: Kittler J, Duff MJB (eds) Image processing system architectures. Research studies press Ltd, Letchworth, pp.23-35

Piper J, Granum E (1987) Computing distance transformations in convex and non-convex domains. Pattern Recognition 20(6):599-615

PIPS (1990) Parallel Image Processing System. Pune University Campus, Ganeshkhind, Pune 411007, INDIA.

Prasanna-Kumar VK, Dionisios IR (1989) Image computations on meshes with multiple broadcast. IEEE transations on pattern analysis and machine intelligence PAMI-11(11):1194-1201

Preston K Jr (1981) Comparison of parallel processing machines: a proposal. In: Duff MJB, Levialdi S (eds) Languages and architectures for image processing, Academic Press, London, pp.305-324

Preston K Jr (1983) Cellular Logic Computers for Pattern Recognition. IEEE Computer 1:36-47

Preston K Jr, Duff MJB (1984) Modern cellular automata - theory and applications. Plenum press, New York.

Preston K Jr (1986) Benchmark results - the abingdon cross. In: Uhr L, Levialdi S, Preston K, Duff MJB (eds) Evaluation of multicomputers for image processing, pp.23-54

Preston K Jr (1989) The abingdon cross benchmark survey. IEEE Computer july:9-18

Reddaway SF (1973) DAP - a distributed array processor. 1st Annual sump. on computer architectures (IEEE ACM), Florida.

Reeves AP (1980a) A Systematically Designed Binary Array Processor. IEEE Transactions On Computers C-29(4):278-287

Reeves AP (1980b) On efficient global information extraction methods for parallel processors. Computer vision, graphics and image processing 14(2)159-169

Reeves AP (1984) Parallel computer architectures for image processing. Computer Vision, Graphics, and Image Processing 25:68-88

Ronse C, Devijver PA (1984) Connected components in binary images: the detection problem. Research study press Ltd.

Rosenfeld A, Pfalz JL (1968) Distance functions in digital pictures. Pattern Recognition 1(1):33-61

Rosenfeld AR (1987) A report on the DARPA image understanding architecture workshop. Proc. of the 1987 DARPA image understanding workshop, pp.298-302

Rosenfeld A, Ornelas J, Hung Y (1988) Hough transform algorithms for mesh-connected SIMD parallel processors. Computer vision, graphics and image processing 41:293-305

Rutovitz D (1966) Pattern recognition. Journal of the royal statistics society 129(A):504-530

Schmitt LA, Wilson SS (1988) The AIS-5000 Parallel processor. IEEE Transations on pattern analysis and machine intelligence PAMI-10(3):320-330

Schmitt LA, Wilson SS (1989) Architecture and Operation of the centipede parallel processor. Technical report #27.

Serra J (1982) Image analysis and mathematical morphology. London, Academic press.

Shih FYC, Mitchell OR (1987) Skeletonization and distance transformation by greyscale morphology. SPIE vol. 849 Automated inspection and High speed vision architectures. pp.80-86

Sommerville I (1982) Software engineering. Edison Wesley, London

Sternberg SR (1986) Grayscale morphology. Computer Vision, Graphics and Image Processing 35(9):333-355

Strong JP (1982) Basic image processing algorithms on the massively parallel processor. In: Uhr L (ed) Multicomputers and image processing algorithms and programs. Academic Press, pp.47-85

Teeuw WB, Duin RPW (1989) An algorithm for benchmarking a SIMD pyramid with the Abingdon Cross. Accepted by Pattern Recognition Letters.

Tyan SG (1981) "Median filtering: deterministic properties. In: Huang TS (Ed) Two dimensional digital signal processing II: transforms and median filters, New York, Springer-Verlag

Toffoli T, Margolus N (1987) Cellular automata machines: a new environment for modelling. MIT press.

Uhr L (1986) On benchmarks: dynamically improving experimental comparisons. In: Uhr L, Levialdi S, Preston K, Duff MJB (eds) Evaluation of multicomputers for image processing, pp.13-21

Uhr L (1988) The coordinated evaluation of parallel architectures for perceptual tasks. In: Uhr L (ed) Multicomputer Vision, Academic Press, pp.53-73

Verbeek PW, Vrooman HA, Vliet LJ van (1988) Low-level image processing by max/min filters. Signal Processing 15(3):249-258

Verbeek PW, Verwer BJH (1990) Shading from shape, the eikonal equation solved by grey-weighted distance transform. Submitted to Pattern Recognition Letters.

Verwer BJH (1988) Improved metrics in image processing applied to the Hilditch skeleton. Proceedings 9ICPR, Rome, pp.137-142

Verwer BJH, Verbeek PW, Dekker ST (1989) An efficient uniform cost algorithm applied to distance transforms. IEEE transations on pattern analysis and machine intelligence PAMI-11(4):425-429

Vliet LJ, Verwer BJH (1987) A contour processing method for fast binary neighbourhood operations. Pattern recognition letters 7:27-36

Vossepoel AM, Smeulders AWM, van den Broek K (1979) DIODA: delineation and feature extraction of microscopical objects. Computer programs in biomedicine 10:231-244

Weems C, Riseman E, Hanson A, Rosenfeld A (1989a) An integrated image understanding benchmark for parallel processors.

Weems CC, Levitan SP, Hanson AR, Riseman M, Shu DB, Nash JG (1989b) The image understanding architecture. International Journal of Computer Vision 2:251-282

Wendt PD, Coyle EJ, Gallagher NC Jr (1986) Stack filters. IEEE transactions on acoustics, speech, and signal processing, ASSP-34(4):898-911

Willson SJ (1989) Convergence of iterated median rules. Computer vision, graphics and image processing 47(1):105-110

Wilson SS (1985) The PIXIE-5000 - A systolic array processor. IEEE Computer Society Workshop for Pattern Analysis and Image Database Management, Miami Beach, Florida, pp.477-483

Wilson SS (1988) One dimensional SIMD architectures - the AIS-5000. In: Uhr L (eds) Multicomputer Vision, Academic Press, London, pp.131-149

Wilson SS (1989a) Personal communication.

Wilson SS (1989b) Vector morphology and iconic neural networks. Technical report #29. Accepted for IEEE Transactions on Systems, Man and Cybernetics.

Woods JW, Lee JH (1983) The fully recursive filter: a general 2-D recursive filter. IEEE transactions on acoustics, speech, and signal processing, ASSP-31(5):1327-1329

Ye QZ (1988) The signed euclidean distance transform and its applications. Proceedings of the 9th ICPR, IEEE CSP, pp 495-499

Yokoi S, Toriwaki JI, Fukumura T (1981) Theoretical considerations on a family of distance transformations and their applications. In: Onoe M, Preston K Jr, Rosenfeld A (eds) Real-Time/Parallel Computers: Image Processing, Plenum Press, New York. pp.73-94

Acknowledgements

At this point I would like to say some special words to those who made it possible that this thesis reached its final form.

First I thank the Lord for his twenty four hour support and for making me realise that "*The fear of the Lord is the beginning of knowledge*"[1], which, in my opinion, even includes image processing.

Second my warmest thanks and respect are expressed towards the project manager Bob Duin, who has always proven to ask the right question at the right time, and who has put me on many better tracks in the elaborate process of reading and commenting on this thesis.

Thirdly, professor Ted Young restored the vision for the whole project from time to time, and demonstrated the attitude of a good researcher to me. He is also thanked for his work in reading and commenting on this thesis.

Pieter Jonker is thanked for his discussions on the use of cellular logic processing elements for grey value processing and Hans Buurman for work with and discussions on the CLIP4 processor array. Ben Verwer is thanked for reviewing the thesis with regard to distance transforms and data-dependent updating techniques.

All the other people in the Pattern Recognition group are thanked for their stimulating discussions during the course of this work.

Liesbeth, my wife, is thanked for bearing my explanations of the tiniest details in this thesis, for her suggestions on paragraph ordering problems, and for her support in general. Irina, my first born daughter, is thanked for giving the necessary distractions while completing the thesis, and yet allowing her father to sleep without much difficulty.

This project was supported by the Foundation for Computer Science in the Netherlands SION with financial support from the Netherlands organisation for Scientific Research (NWO).

The text and figures of this thesis were made using FrameMaker 2.0 on a SUN workstation. The experiments presented in Chapter 8, and most of the images shown in this thesis have been made using the TCL-Image software package of the Delft Centre for Image Processing (CBD), marketed by Multihouse TSI (Amsterdam).

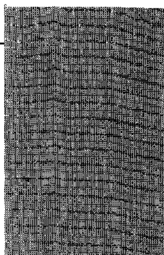
1. Proverbs 1:7, New International Version of the Bible.

Samenvatting

In dit proefschrift wordt een methode ontwikkeld om computerarchitecturen te vergelijken die ontworpen zijn voor "low-level" operaties (transformaties van beeld naar beeld) in de beeldverwerking. Op grond van kenmerken die in bestaande en gepubliceerde architecturen gevonden zijn wordt er een onderscheid gemaakt tussen drie principieel verschillende groepen: het vierkante processor array, het lineaire processor array en de pipeline. "Low-level" beeldverwerking wordt onderverdeeld in diverse categorieën operaties welke onderscheiden worden door de hoeveelheid en de soort van parallellisme die deze operaties aankunnen. De vergelijking tussen de architectuurgroepen wordt voor de meeste operatiegroepen uitgevoerd.

Van deze groepen kosten de globale en de object operaties veel tijd omdat zij voor de berekening van een beeldpunt in het resultaat beeld vereisen dat er veel data over een grote afstand wordt opgehaald. Veel van deze globale en object operaties kunnen echter geschreven worden als recursieve omgevingsoperaties. Deze zijn lokaler georiënteerd, en lijken daarom ook beter geschikt te zijn voor de onderzochte architecturen. Er is een begin gemaakt met de ontwikkeling van een theoretisch raamwerk voor de recursieve omgevingsoperaties. Het gebruik van deze operaties voor beeldbewerking wordt gedemonstreerd, en er wordt getoond hoe zij op de diverse architectuurgroepen kunnen worden geïmplementeerd. Uiteindelijk worden de prestaties van de architectuurgroepen onderzocht voor zes niet-lineaire recursieve omgevingsoperaties.

Voor wat betreft de prestaties van de onderzochte architecturen voor "low-level" beeldverwerking in het algemeen, wijzen onze resultaten er op dat het lineaire processor array de beste combinatie van snelheid en efficientie biedt.



Curriculum Vitae

Erwin R. Komen is born in Utrecht, the Netherlands, on September 8, 1960. He graduated in precision engineering at the Polytechnical Institution of Hilversum in 1981. A year military service then followed. Afterwards he gained the 'Ingenieur' degree in applied physics from the Delft University of Technology. Then a four year PhD project started with the Pattern Recognition Group of the Delft University of Technology. His research interests are in the comparison of architectures for low-level image processing, and in recursive neighbourhood operations theory and applications.

Erwin R. Komen is op 8 september 1960 te Utrecht geboren. In 1981 studeerde hij af in de fijnmechanische techniek aan de HTS te Hilversum. Daarna volgde een jaar militaire dienst. Vervolgens behaalde hij de ingenieurs graad in toegepaste natuurkunde bij de Technische Universiteit Delft. Toen startte een vier jaar durend promotie project bij de patroonherkenningsgroep van de Technische Universiteit Delft. Zijn onderzoeks belangstelling gaat uit naar de vergelijking van architecturen voor gebruik bij low-level beeldverwerking, en naar de theorie en de toepassing van recursieve omgevings operaties.

Appendix

A.1 Data

This section contains raw data of the combined deterministic/data-dependant and the asynchronous updating method experiments presented in Chapter 8. Table A.1 contains the data used for the determination of the updating effort comparison in Figure 8.10. Table A.2 until Table A.7 contain the results of the measurements for the performance of the asynchronous updating method, as presented in Figure 8.8.

Table A.1 Comparison of combined queue and simultaneous updating

	Object Selection		Distance Transf.		Min Median.		SERP		GWDT		Rec. Median	
	Diff.	Aver.	Diff.	Aver.	Diff.	Aver.	Diff.	Aver.	Diff.	Aver	Diff.	Aver
LPA:												
1*1	3.67	2.47	3.49	3.43	2.27	3.22	4.57	2.35	27.39	6.27	2.01	3.14
2*1	5.57	2.57	5.02	3.60	3.08	4.05	8.13	3.28	27.26	7.28	2.69	3.80
4*1	9.24	2.79	8.06	4.19	4.63	5.19	11.35	4.88	34.65	7.92	3.73	4.61
8*1	15.95	3.15	11.09	5.43	7.36	7.01	18.31	7.86	35.77	8.72	5.57	6.11
16*1	27.61	3.44	15.13	7.14	12.30	9.39	27.37	12.51	33.66	10.87	9.04	7.95
32*1	49.11	3.94	18.63	9.18	17.34	12.10	30.05	17.02	51.63	15.14	12.24	10.08
64*1	64.98	4.59	18.00	10.34	22.02	14.08	33.44	21.63	72.38	17.19	15.09	11.77
SPA:												
2*2	8.08	3.19	4.94	4.80	4.50	4.84	12.70	4.12	35.13	8.75	3.95	4.66
2*4	12.64	3.42	5.35	5.24	6.83	6.08	19.67	6.13	38.46	10.11	5.64	5.64
4*4	19.29	4.70	7.27	6.50	9.86	7.37	28.54	8.42	44.93	11.95	8.01	6.96
8*4	31.48	5.42	8.23	7.56	12.52	9.15	42.93	12.42	49.66	14.69	9.81	8.92
8*8	51.23	7.67	11.88	9.28	16.66	10.69	65.31	17.69	68.08	18.08	13.75	10.17
16*8	88.28	9.38	14.34	10.31	25.66	14.09	91.97	24.13	66.72	22.06	21.16	12.31
16*16	154.19	15.13	20.25	11.19	37.06	18.88	117.31	34.32	103.88	29.88	29.88	16.31
32*16	283.38	20.63	26.25	13.38	64.75	27.25	208.75	48.25	158.75	43.25	53.75	24.13
32*32	523.00	31.75	33.50	14.00	106.50	33.00	275.75	59.25	258.25	60.25	93.50	27.00
64*32	1025.50	43.50	33.50	14.00	137.00	40.50	309.00	87.50	396.00	61.50	133.50	33.50
64*64	2018.00	57.00	33.00	14.00	187.00	24.00	61.00	32.00	501.00	53.00	184.00	26.00

Table A.2 Performance of asynchronous updating for Object Selection: ten measurements per Relative Variable Processing Time for the processing speed in number of clock cycles.

RVPT(%:)	Average case									
0	57.1	57.1	57.1	57.1	57.1	57.1	57.1	57.1	57.1	57.1
10	53.4	53.1	52.6	53.3	52.8	53.4	53.8	52.7	53.1	53.6
20	51.2	49.8	50	50.7	51.5	49.8	50.7	50.2	49.9	50.8
30	47.5	48.5	49.1	46.9	47.7	47.4	47.5	46.7	47.3	48.3
40	44.1	47.1	44.2	44.5	45.3	46.6	45.9	44.3	45.8	44.9
50	42	41.7	40.9	42.6	43.9	44.6	41.6	42.2	42.2	41.8
60	40.4	39.7	38.6	39.9	39.1	38.7	38.2	39.1	38.3	40.7
70	37.2	37.3	36.4	36	37.7	35.3	40	35.8	34.8	38.4
80	34.4	31.8	35.4	34.8	34.1	34.2	35.7	33.3	36	32.8
90	30.4	29.4	30.5	29.5	29.5	31.8	28.7	33.6	30.8	31.5
RVPT(%):	Difficult case divided by 100									
0	20.181	20.181	20.181	20.181	20.181	20.181	20.181	20.181	20.181	20.181
10	20.162	20.178	20.157	20.215	20.157	20.127	20.187	20.13	20.237	20.113
20	20.13	20.24	20.14	20.185	20.159	20.168	20.209	20.149	20.081	20.107
30	20.159	20.18	20.158	20.207	20.058	20.113	20.179	20.145	20.118	20.209
40	20.147	20.259	20.204	20.229	20.252	20.241	20.22	20.042	20.288	19.961
50	20.036	20.224	20.12	20.094	20.14	20.184	20.312	20.504	20.21	20.414
60	20.127	20.20	20.144	19.911	20.11	20.287	20.016	20.194	19.929	20.266
70	20.197	19.882	20.129	20.125	20.454	20.227	20.459	20.018	20.113	19.974
80	19.959	20.681	20.28	19.999	20.124	19.875	19.931	20.032	19.953	20.231
90	20.649	19.821	20.234	20.024	20.429	20.349	20.307	20.15	19.964	20.884

Table A.3 Performance of asynchronous updating for the Distance Transform: ten measurements per Relative Variable Processing Time for the processing speed in number of clock cycles.

RVPT(%:)	Average case									
0	15.1	15.1	15.1	15.1	15.1	15.1	15.1	15.1	15.1	15.1
10	15.5	15.1	15.8	15	15.4	15.5	15.6	15.9	15.8	15.4
20	15.9	15.9	16.2	16.1	16	15.7	16.1	15.9	16.4	16.2
30	15.7	16.1	16.4	16.5	16.9	16.6	16.9	16.9	16.3	16.4
40	16.4	16.8	16.9	17	17	17.5	16.7	17.1	16.4	17
50	17.7	18	17.9	17.3	17.3	17	17.5	17.6	17.8	18
60	18.6	18	18.6	20.5	18.2	18.2	18.1	18.4	19.1	18.9
70	19.7	19.4	18.9	19.6	19.3	19.9	20.8	19.4	18.2	18.6
80	18.5	21.5	19.8	19.5	20.7	18.6	19.6	20.3	19.5	19.6
90	19	20.9	20.5	20.1	21.4	18.4	19.2	20.1	21	21.1
RVPT(%):	Difficult case									
0	33.1	33.1	33.1	33.1	33.1	33.1	33.1	33.1	33.1	33.1
10	33.7	33.5	33.9	34.1	34.3	33.7	34.3	33.6	33.8	33.7
20	36.2	34.2	34.7	34.4	34.5	34.8	34.7	34.9	35.2	34.6
30	35.4	35.4	35.2	35.1	35.7	36.1	35.6	35.3	35.4	35.8
40	36.4	36.7	37.4	36.6	37.4	37.6	37.2	37	37.4	36.6
50	37.3	37.8	39.2	38.5	38.4	37.2	37.8	36.9	37.9	37.4
60	37.9	37.6	40	38.8	38.8	39.2	38.9	39.1	37.9	38.4
70	38.4	39.5	40.1	39.5	39.2	40.3	39.3	40.5	38.4	41.6
80	40.2	43	43.1	41.5	42.3	40.6	44.2	40.9	46.3	39.9
90	43.8	43.9	43.9	46	43.7	42.1	44.7	43.6	43	41.5

Table A.4 Performance of asynchronous updating for the GWDT: ten measurements per Relative Variable Processing Time for the processing speed in number of clock cycles.

RVPT(%:)	Average case									
0	53.1	53.1	53.1	53.1	53.1	53.1	53.1	53.1	53.1	53.1
10	53.2	52.7	53.6	53.1	52.3	52.6	52.9	52.7	53.3	53.2
20	53.3	53.6	54.3	52.5	53.9	54.5	53.8	51.9	52.9	54.3
30	54.8	53.8	54.7	55	52	52.7	54	55	54.1	52.1
40	56	56	58.3	54.2	52.9	57.1	54.3	55.4	53.1	56.1
50	57.4	57.5	56.8	57.9	58.7	57	54.8	56.4	57.7	55.4
60	59.4	58	58.1	55.6	57.2	57.2	55.9	61.2	58.6	58.4
70	59.2	59.2	59.6	59.8	58.4	60.6	58	61.8	58.6	60.2
80	59.8	61.7	59.7	61.6	57.8	64	64.1	60.7	59.6	63.1
90	61.2	64.3	61	60.2	62.1	58.3	64.4	63.2	62.1	61.9
RVPT(%):	Difficult case									
0	501	501	501	501	501	501	501	501	501	501
10	506.5	506.7	500.9	504.8	504.6	503.5	506.4	504.3	505.7	506.2
20	515.8	514.5	516.8	511.6	516.1	513.3	515.6	515.4	514.1	515.6
30	525.9	525.8	523.9	522	524.5	528.1	521.2	525.5	528.9	531.2
40	538.6	537.1	537.5	543.3	543.7	543.3	538.7	539.2	543.7	536.8
50	553.8	555	552.9	550	554.5	556.1	552.7	558	558.4	556.5
60	566.3	569.4	577.7	565.6	563.1	563.2	566.7	565.6	573.3	573.6
70	580.5	580.6	592.1	589.6	584.7	588.8	585.4	584.2	578.8	577.1
80	605.9	590.9	593.1	593.8	596.5	600.4	603.5	595.3	585.9	595.3
90	607.6	613.2	617.9	613.6	619.7	621.6	615.4	614.1	611.1	611

Table A.5 Performance of asynchronous updating for Median Root: ten measurements per Relative Variable Processing Time for the processing speed in number of clock cycles.

RVPT(%:)	Average case									
0	26.1	26.1	26.1	26.1	26.1	26.1	26.1	26.1	26.1	26.1
10	62.9	72.5	59.6	63.1	78.1	91.3	69.8	106.3	87.6	70.3
20	36.1	26.6	41.2	34.3	30.6	33.5	45.1	38.2	36.8	42.3
30	28.2	28.7	25.8	25.7	26.7	26.8	32.1	28.9	28.4	27.5
40	26.7	27	30.3	29.1	24.4	26.8	29	30.7	29.9	26.3
50	23.7	22.5	30.4	29.5	28.9	22.8	28.3	25.1	27.7	27.5
60	30.1	23.9	24.5	27.7	33.6	30.5	31.1	23.7	30.7	30.6
70	27.2	30.8	29.6	31	24.4	27.7	26.2	31.1	30	25.7
80	32.2	32.4	25.1	25.5	24	31.5	34.7	26.4	27.6	27.8
90	25.1	28.3	25.8	27.2	34.3	30.5	28.9	32	30.1	25.2
RVPT(%):	Difficult case									
0	184	184	184	184	184	184	184	184	184	184
10	188.7	188.7	187.4	188	186.8	187.4	188.1	188.9	187.5	188.5
20	191.2	190.3	191.1	192.6	189.8	190.6	192.2	193	191.8	192.1
30	196.1	196.1	192.3	189.3	192.6	191.3	194.8	196.7	191.6	193.2
40	191.6	196.7	198.9	194.8	192.3	196.5	196.3	196.5	197.9	203.5
50	198.7	195.1	206	196.9	197.4	201.9	206.9	196.5	194.5	198.5
60	208.1	198	194.2	201.8	201.1	200.5	205.2	200.7	204.5	203.9
70	211.7	203.6	204.6	208.1	213.9	206.9	207.3	207.8	210.1	205.9
80	206.5	214.5	215.9	211.6	214.1	209.4	224.1	209.4	210.2	214.7
90	201.8	217.2	210.5	215.2	199.8	210.6	221.7	216.7	211.1	216.7

Table A.6 Performance of asynchronous updating for Minimum Median Root: ten measurements per Relative Variable Processing Time for the processing speed in number of clock cycles.

RVPT(%:)	Average case									
0	24.1	24.1	24.1	24.1	24.1	24.1	24.1	24.1	24.1	24.1
10	24.7	24.7	24.3	24.9	24.5	24.6	24.1	24.2	24.6	24.4
20	25.1	24.9	24.4	25	24.1	25.2	24.9	25.6	25.2	23.8
30	25.1	25.1	26.1	24.5	25.4	24.5	24.3	25.1	25.8	24.8
40	26.3	26.8	26	25.1	26.5	25.6	26.2	25.7	24.7	25.3
50	27	26.8	25.1	26.6	25.1	25	26.2	24.4	26	28
60	27.3	26	29	27	26.8	26	28.3	26	25.9	26.7
70	28.3	26.6	26.7	27	28.3	27	26.5	28	27.7	26.2
80	26.4	28.9	28.8	27.8	25.3	27.4	30	27.8	27.4	26
90	29.1	27.3	28.9	27.9	28.1	30.8	30.3	29.7	28.3	30
RVPT(%):	Difficult case									
0	187	187	187	187	187	187	187	187	187	187
10	191.9	189	189.8	191.3	190.5	188.8	190.2	190.2	191.1	190.6
20	195.5	193.3	193.2	192.2	196.8	193.6	195.7	195.1	195.4	192.9
30	199.4	196.4	198.2	201.3	193.6	196.4	196.2	196.1	198.8	198.8
40	202.8	197.7	198.1	201.1	202.5	201	193.3	202.5	198.6	203.2
50	197.6	198	203.3	201.1	205.4	207.6	201.3	203.4	203.5	209
60	207	213	196.5	208.6	214.5	208.8	214	208.6	206.1	214.8
70	207.4	208.7	215.2	215.3	205.4	212.8	213.1	209.1	208.6	208.2
80	221.4	210.1	218	219.8	207.8	208.3	219.5	213.9	200.6	211.8
90	222.9	212.7	214.8	227.1	219.2	226.2	220.1	212.4	220.8	226.1

Table A.7 Performance of asynchronous updating for SERP: ten measurements per Relative Variable Processing Time for the processing speed in number of clock cycles.

RVPT(%:)	Average case									
0	32.1	32.1	32.1	32.1	32.1	32.1	32.1	32.1	32.1	32.1
10	79.5	98.3	83.6	91.6	73.1	100.2	91.7	79.3	80	79.1
20	92.7	92.7	36	72	32.4	74.3	34	100.4	75.9	83.5
30	89.6	87.4	33.8	46.2	33.3	122	86.3	96.3	40.5	42
40	97.6	35.1	104.5	32.2	34.4	35.8	99.9	34.2	92.1	35.7
50	39.9	105.5	96.4	35.3	34.3	33.3	109.2	34.8	44	37.6
60	117.6	39.6	36	35.3	36.7	34.2	36.7	34.6	34.5	36.6
70	39.8	42.3	36	114.4	107	96.6	38.5	40.4	39.6	98.2
80	36.7	39.3	46.6	38.1	46.5	43.2	38.6	39.5	39.1	42.1
90	36.8	37.5	39.7	40	44.1	40.7	38.5	38.1	43.5	37.3
RVPT(%):	Difficult case									
0	61.1	61.1	61.1	61.1	61.1	61.1	61.1	61.1	61.1	61.1
10	61.1	56.1	60	61.3	59.6	58.6	56.4	57.5	58.5	60.2
20	61	62.6	61.7	60.7	59	64.2	61.6	60	62	61.3
30	64	64.7	63.7	63.5	65	65.3	64.7	64.9	66.2	65.8
40	68.6	67.8	68.3	67	70.9	66.8	68.7	67.9	67.7	70.2
50	72.6	71.3	73.4	71.4	72.6	72	72.1	72.6	74.8	69.7
60	73.6	73.7	75.2	74.3	71.6	75.3	74.6	76	74.2	75.1
70	78.5	77.2	75.9	77	78.3	77.6	74.5	79.8	78.1	80.5
80	79.6	81	79.8	80.5	81.9	78.7	79.4	81.2	81	77.4
90	81.2	82.1	81.7	81.9	80.7	87.1	81.9	82.5	83.1	82.1

Index

8-connected updating 99

A

AIS-5000 18, 24, 51, 60, 64, 135
algorithm length 58
algorithmic phase-diagram 67
alternated updating 99
ALU, see arithmetic and logical unit 55
analysis
  operations 2, 35, 161
  task 35
architecture
  cost 42
  existing low-level image processing 20
  group 2, 13, 34, 161
  proposed and novel low-level image processing 27
  size 36
architecture classification feature 2, 5
arithmetic and logical unit 9, 55
array 10
  full 14, 16, 28
array zero test 75
assembler 44, 53
associative capabilities 23
Associative String Processor 27
asynchronous 22, 28, 56
asynchronous updating 102, 138, 149
autonomy 6
average case 143

B

bandwidth 43, 51, 68
BASE 21, 27, 136
basic instruction 64
basic instruction length 58
benchmarking 35
BIBO, see stability
bit count 75
bit parallel pipeline 56
bit serial processing element 55
bitplane 15, 50
bit-plane arithmetic 15
bitstream 20, 56
blob to point reduction 79
bounded input bounded output stability 88, 104
breadth first 100
BSPE, see processing element
bucket updating 102, 136, 163

C

CAAPP 23, 28, 73
carry memory 20, 55
carry recirculation 52
cellular logic
  architectures 13
  grouping 13
  operations 13
Centipede 55
central pixel 15, 86, 97, 98, 106
chain coding 1
chessboard updating 99
CLIP4 21, 34, 50, 62, 73, 136, 138
CLIP7 29
CLO 26
CLPE 30, 56, 64, 76
column parallel data input 43
column serial updating 99
combined updating method 138, 152
co-memory 19
command language 44
commuting with thresholding 116
comparison
  low level image processing architectures 49 to 78
  strategy 35, 46
  theoretical 40
connectivity
  features 9
  neighbourhood 9, 15, 19, 22 to 34, 52, 58, 72, 77
  recursive neighbourhood 9, 22 to 34
constraint
  initial output image 84
  updating method 84
constraint distance transform 127
control flow 32
control unit 6
convex hull 79, 127
convolution 78
convolution property 116
corner turning 50, 58
counter 75
crinkle-mapping 16, 52, 72, 77, 135
crossbar 57
curve smoothing 127

cyclicity 83, 112, 115
Cytocomputer 25, 61
Cyto-HSS 25, 54, 64

D

DAP 21, 73
DARPA 36
data dependent updating method 97, 100, 135, 148
data flow 7, 11, 32
data I/O 6, 37, 42, 50, 58, 64, 77
  actual speed 51
  bandwidth 43
  column or row parallel 43
  devices 51
  image parallel 43
  overlapped 50, 60
  possible speed 51
  raster scan 43
data size 42
definition
  low-level image processing operations 81
degree of parallelism 8
depth first 100
deterministic updating method 97, 98, 107, 133 to 135, 148
diagonal serial updating 99
difficult case 143
digital signal processor 23, 32
DIP 26
Dirichlet tessalation 127
distance transform 72, 79, 89, 113, 126
  constraint 127
  grey weighted 128, 145
  normal 126, 144
  signed Euclidean 127
dithering algorithm 130
DOCIP 16, 28, 50
double serial updating 135
DSP, see digital signal processor
dyadic operation 8, 20, 54

E

ECS, see Erlangen classification scheme
edge
  constant 84
  mirror 84
  wrap 84
edge handling 83
edge store scanning 17, 62, 138
edge type 141
edge-problem 17
efficiency 45, 64, 66
elementary logic circuit 10
Erlangen classification scheme 9
error measure 134

F

feature
  architecture classification 2, 5
  operation classification 2
  selection of architecture classification 11
Feng-coordinates 8
FFT, see Fourier transform
figure of merit 44
  efficiency 45
  pixel operations per second 45
  quality factor 45
  required clock-cycle time for the processing of images at video speed 45
fixed point 80, 83, 85, 104, 108, 162
  stability 104
  theory 106 to 113
  uniqueness 104, 109
flexibility 77
  image size 43, 51
  instruction sequence 53
  neighbourhood size and shape 43, 52
floating point 52
Flynn taxonomy 5
Fourier transform 73
frame grabber 19, 43
frame recirculation 20, 53, 76, 135
full array 14
fully connected 11

G

GAPP 22
general purpose image processor 133
general purposeness 10
GeO, see geometric operation
geometric operation 36, 39, 73, 82
GlO, see global operation
global error measure 134
global operation 2, 36, 39, 72, 79, 82, 112
global propagation 22, 34, 73
graph building 1
grey value operation 51
grey value processing element 54
grey weighted distance transform 93, 113, 128, 145
greyscale erosion 127
GRID 27, 73
grouping
  architecture 2, 13, 161
  cellular logic 13
  Lougheeds taxonomy 13
  operation 2, 37, 39
GWDT, see grey weighted distance transform

H

half scan addressing 17, 62
high level image processing 1

high level programming language 44
histogram 76
hologram 28
Hough transform 72
hypercube 28, 73

I

iconic layer 1
iconic to symbolic layer 1
ILLIAC III 21
image analysis 36, 75
image combiner 55
image parallel data input 43
image parallelism 8, 38
image processing
  high level 1
  intermediate level 1
  low level 1, 35
  multiresolution 31
  multi-scale 16
IMF, see index mapping function
IMPP 32
index mapping function 98, 133
indirect addressing 55
instruction 40
instruction length 58
instruction level processing 17
instruction overhead 40, 49, 53, 56, 59, 78
instruction power 40
intermediate level image processing 1
inverse convolution 79, 86
isotropeness 38, 74
iteration 85

L

labelling 130
latency 52, 56, 61
layer
  iconic 1
  iconic to symbolic 1
  symbolic 1
left spiral updating 99
line 10, 33
line buffer 43
line serial updating 99
linear processor array 13, 17, 34, 49 to 78, 135
LISP 23
LNO, see local neighbourhood operation
local activity control 7, 25, 74
local addressing autonomy 7, 25, 74, 77, 136
local algorithm control 7
local autonomy 7, 33
  activity control 7, 25
  algorithm control 7
  connectivity control 7, 11, 24, 28, 72, 73
  data addressing control 7, 25, 74, 77, 136
  function control 7, 25
  local activity control 74
  partitioning control 7
  sequencing control 7
local connectivity control 7, 11, 24, 28, 72, 73
local function control 7, 25
local neighbourhood operation 36, 39, 53, 58 to 71, 77, 82
local neighbourhood processing 2
local partitioning control 7
local sequencing control 7
lookup table 15, 27, 33, 53, 64
Lougheeds taxonomy 13
low level image processing 1, 35
low level image processing operation 37
LPA, see linear processor array

M

macro pipelining 9
maximum median root 113, 118, 125
meander updating 99
medial axis transform 79, 122
median root 72, 73, 84, 88, 118, 120, 123, 146
mesh 10, 28, 33, 73
micro assembler 44, 53
MIMD 6
minimum median root 118, 125, 146
MISD 6, 33
MITE 30
MMB 28, 73
model problem 121
modelling 106
monotonically decreasing 107
monotonically increasing 107
morphological skeleton 115, 122
morphology 115, 122, 127
MPP 21, 50, 74
MSIMD 6, 31, 33
multiple bus 28
multiresolution image processing 31
multi-scale image processing 16

N

N-cube 10, 73
neighbourhood connectivity 9, 15, 19, 22 to 34, 52, 58, 72, 77
neighbourhood parallelism 8, 15, 21 to 34, 58, 77
neighbourhood size and shape flexibility 52
NlogN reconfiguring network-based system 11
non-linear operation 79
normal distance transform 126, 144

O

object operation 2, 36, 39, 71, 79, 82, 112

object selection 72, 79, 94, 113, 143
OO, see object operation
operation
  dyadic 8, 20, 54
  formulation of low-level image processing 80
  geometric 36, 39, 73, 82
  global 2, 36, 39, 72, 79, 82, 112
  grey value 51
  group 2
  local neighbourhood 36, 39, 53, 58 to 71, 77, 82
  low level image processing 37
  morphological 122
  non-linear 79
  object 2, 36, 39, 71, 79, 82, 112
  point 36, 39, 54 to 57, 77, 81
  ranking 40
  recursive neighbourhood 3, 36, 39, 71, 79 to 120
  recursive neighbourhood, see also RNO
  statistical 36, 82
  statistical scalar 39, 75
  statistical vector 39
operation classification feature 2
operation grouping 37, 39
operation parallelism 8, 20, 33
operations analysis 2, 35, 161
OPSILA 6
order function 98
order statistic filter 116
output entity 38
overhead 17, 40, 62, 64, 138
  instruction 49, 53, 56, 59, 78
  scanning 49, 58, 60
overlapped data I/O 50

P

PAPIA 31
parallel subarray 14
parallelism
  degree of 8
  image 8, 38
  neighbourhood 8, 15, 21 to 34, 58, 77
  operation 8, 20, 33
  pixel bit 58
  pixel-bit 8, 20, 29, 38, 78
  recursive neighbourhood 8, 19, 22 to 34
  spatial 8, 33, 38, 78
  taxonomy 8
PE, see processing element
performance 65
  criteria 41
phase diagram 67
phase transition 67
PICAP3 18, 24, 52, 55
PIPE 31
pipeline 7, 13, 19, 34, 49 to 78, 135
  bit parallel 56
  macro pipelining 9
  reconfigurable 19
  stage 14
pipeline ring bus 32
pixel bit parallelism 58
pixel operations per second 45
pixel-bit parallelism 8, 20, 29, 38, 78
PL, see pipeline
PMF, see processor mapping function
PO, see point operation
point operation 36, 39, 54 to 57, 77, 81
point operations ram 54, 57, 76
Poisson equation 114, 121, 147, 148
Poisson updating 102, 103, 136
polygon 10
POR, see point operations ram
positive Boolean function 111
precedence graph 100
preferred method 36
processing element 6, 42, 53, 64
  bit serial 15, 34, 55
  BSPE 55, 59
  grey value 33, 54
processing time 41
processor mapping function 16, 17, 29, 135
program control unit 9
programmability 44, 53
programming
  language 37
  time per task 37
propagation 54, 55
PTA 28, 73
PYR, see pyramid
pyramid 11, 13, 31, 33

Q

quality factor 45
queue updating 101, 138

R

rank order filter 116
raster scan data input 43
raster scan updating 99
raster subarray 14
raster-pipeline subarray 14
reasoning 1
recognition 1
reconfigurable pipeline 19
recursive median, see median root
recursive neighbour 20
recursive neighbourhood connectivity 9, 22 to 34
recursive neighbourhood operation 3, 36, 39, 71, 72
recursive neighbourhood parallelism 8, 19, 22 to 34
recursive updating 100
region analysis 1

relative variable processing time 142, 149
research
  goal 2
right spiral updating 99
ring 10, 28
RNO
  classes 114, 118
  cyclicity 105
  definition 82, 142
  experiments 141
  fixed point 104
  implementation 133 to 139
  morphological 122
  performance of architecture groups 156 to 158
  performance of updating methods 147 to 155
  relation to global operations 86
  relation to local neighbourhood operations 85
  relation to object operations 89
  usage in image processing 121 to 131
  with non-cyclical neighbourhood 130
RNO, see also recursive neighbourhood operation
root 3, 85
rotation 73, 74
row or column bus 73
row parallel data input 43
runcoding 1

S

scaling 73
scan 21, 63
scanning 58
  edge store 17
  half scan addressing 17
  hardware 18
scanning overhead 49, 58, 60
sensor 43
serial updating 135
SERP, see smallest enclosing regular polygon
shared memory 7
shotnoise 125
signed Euclidean distance transform 127
SIMD 5, 33, 73, 136, 162
simulator 137
simultaneous updating 99, 107, 135, 139
SISD 5
skew 74
SLAP 28, 60, 76, 136
slightly augmented tree 10
smallest enclosing regular polygon 79, 95, 113, 125, 143, 162
smallest surviving object 123
SOR, see successive over relaxation
SPA, see square processor array
spatial parallelism 8, 33, 38, 78
spin-flip 106
SPMD 6
square processor array 13, 15, 34, 49 to 78, 135
SSO, see statistical scalar operation
stability
  asymptotic 104
  bounded input bounded output 88, 104
  fixed point 104
stack filter 117, 122, 126
stacking property 108
stage 14
star 10
statistical operation 36, 82
statistical scalar operation 39, 75
statistical vector operation 39, 75
stochastic updating 102
stochastic updating method 98, 136
stream taxonomy 5, 33
stroke 18, 52, 58
subarray
  parallel 14
  raster 14
  raster-pipeline 14
successive over relaxation 100, 134, 149
SVO, see statistical vector operation
symbolic layer 1
SYMPATI-2 18, 29, 52, 55, 76, 135, 139
system programming language 44, 53

T

Tanque Verde Benchmark Suite 36
task
  choice 35
  description 36
task analysis 35
TCL-Image 141
tessellation
  8-connected 15
  Dirichlet 127
  hexagonal 15, 22
test data 36
theoretical comparison 40
threshold decomposition 116
threshold logic filter
  weighted 117
time-skewing 56
TIP-4 32
topology 6, 10
  array 10
  data dependent 11
  data-dependant 28
  fully connected 11
  hypercube 73
  line 33
  lines 10
  mesh 10, 28, 33, 73

multiple bus 28
N-cube 10
NlogN reconfiguring network-based system 11
polygon and ring 10
pyramid 11, 33
ring 28
slightly augmented tree 10
star 10
torus 11, 33, 72
tree 10, 28

torus 11, 33, 72
translation 73
transposition 55, 73
Transputer 27, 42
tree 10, 28
truth table 24
typical case 35

U

update fraction 98, 103
updating effort 142
updating method 3, 80, 97, 141, 162
8-connected alternating 99
alternated 99
asynchronous 102, 138, 149
bucket 102, 136, 163
chessboard 99
column serial 99
combined 138, 152
data dependent 97, 100, 135, 148
deterministic 97, 98, 107, 133 to 135, 148
diagonal serial 99
double serial 135
implementation 141
left spiral 99
line serial 99
meander 99
Poisson 102, 103, 136
qualitative comparison 123
queue 101, 138
raster scan 99
recursive 100
revisited 103
right spiral 99
serial 135
simultaneous 99, 107, 135, 139
stochastic 98, 102, 136
vertical serial 99

V

verification 62
vertical serial updating method 99
viewangle 95

W

warping 73, 77
window mapping 16

Author's Index

A

Arce and Crinon 1984 123

B

Basille 1989 131
Batcher 1980 21, 50
Blum 1967 79
Boyce and DiPrima 1965 83
Buurman and Duin 1988 17, 22, 58, 63, 64

C

Cantoni 1986 6
Cantoni and Levialdi 1983 10
Cantoni and Levialdi 1987 31
Chan 1976 100
Clarke and Ip 1982 74
Cypher and Sanz 1989 7

D

Danielsson 1980 92, 127
Danielsson and Levialdi1981 1, 5, 8
Davis and Keller 1982 32
Davis and Thomas 1984 22
Dekker et al. 1987 7, 11, 32
Döhler 1989 123
Dorst 1986 127
Dorst and Verbeek 1986 127
Duclos et al. 1988 6
Dudgeon and Mersereau 1984 79, 98, 114
Duff 1982 22, 50
Duff 1986 35, 36, 37, 40, 41
Duin and Jonker 1986 6, 61
Duin and Jonker 1988 61
Duin and Komen 1989 13, 19, 20
Duin et al. 1986 77

F

Feng 1977 8
Fisher and Highnam 1985 28
Fisher et al. 1987 28, 51
Flynn 1972 5, 7
Forchheimer and Ödmark 1983 43
Forshaw 1987 10
Fountain 1983 20
Fountain 1986 1
Fountain 1987 6, 13, 14, 17
Fountain 1988b 7
Fountain 1988c 73
Fountain et al. 1988a 29
Fujita et al. 1990 32
Fung 1977 21

G

Garda et al. 1988 43
Gerritsen 1982 6, 26, 45, 59, 61
Gerritsen and Aardema 1981 26
Goles and Olivos 1981 119
Graham and Norgren 1980 33
Groen et al. 1988 32

H

Händler 1977 5, 9
Haralick 1981 88, 90, 130
Haralick et al. 1987 115, 122, 127
Hockney 1987 67
Hockney and Jesshope 1981 100, 121
Huang et al. 1989 16, 28
Hunt 1981 21

I

INMOS 1988 42
Iwashita et al. 1986 32

J

Jenkins and Lee 1987 26
Jonker and Duin 1985 30
Jonker et al. 1988b 61
Jonker et al. 1989 20, 52, 76
Jonker et al. 1990 32
Jonker et al.1988a 128, 131, 136, 159, 163
Justusson 1981 125
Juvin et al. 1988 18, 29

K

Kent et al. 1985 31
Kimmel et al. 1985 30
Kogge 1981 9
Komen and Duin 1989 45, 56
Kraaijveld et al. 1986 61

L

Lea 1988 27
Leveque and Trefethen 1988 100, 121
Levialdi 1988 39
Lindskog 1988 1, 6, 7, 18, 24, 73, 74
Lorrain and Corson 1970 114
Lougheed 1987 8, 13, 25, 54, 61
Lougheed and McCubbrey 1980a 25

Lougheed and McCubbrey 1980b 45, 59, 60

M

Manry and Aggarwal 1974 98
Maragos 1987a 115
Maragos 1989 116
Maragos and Schafer 1986 122
Maragos and Schafer 1987b 115
Maragos and Schafer 1987c 115, 116, 117, 125
Maresca et al. 1989 11, 28
McCormick 1963 21
Montenari 1968 79

O

Otto 1984 104, 105

P

Pass 1985 27
Piper and Granum 1987 100, 101
PIPS 1990 27
Prasanna and Dionisios1989 14, 28
Preston 1981 36, 37, 41
Preston 1983 20, 45, 97, 116
Preston 1986 45
Preston 1989 1, 41, 45
Preston and Duff 1984 20, 45

R

Reddaway 1973 21
Reeves 1980a 27
Reeves 1980b 75
Reeves 1984 20, 27
Ronse and Devijver 1984 1
Rosenfeld 1987 36
Rosenfeld and Pfalz 1968 79, 89, 90
Rosenfeld et al. 1988 72
Rutovitz 1966 97

S

Schmitt and Wilson 1988 24, 73
Schmitt and Wilson 1989 33, 50, 55, 163
Serra 1982 115, 122
Shih and Mitchel 1987 127
Sommerville 1982 44, 100
Sternberg 1986 115
Strong 1982 74

T

Teeuw and Duin 1989 16
Toffoli and Margolus 1987 102, 106
Tyan 1981 118

U

Uhr 1988 10, 36, 40, 42

V

van den Boomgaard 1989 117
van Vliet and Verwer 1987 102
Verbeek and Verwer 1990 94, 128, 129, 145, 146
Verbeek et al. 1988 44
Verwer 1988 131
Verwer et al. 1989 102
Vossepoel et al. 1979 129

W

Weems et al. 1989a 36
Weems et al. 1989b 5, 23
Wendt et al. 1986 108, 111, 116, 117, 120, 122, 125
Wilson 1985 24
Wilson 1988 18, 55
Wilson 1989 117
Wilson 1989a 18
Woods and Lee 1983 115

Y

Ye 1988 127
Yokoi et al. 1981 93, 128