To the Graduate Council:

I am submitting herewith a dissertation written by Hongtao Du entitled ”Efficient Image Processing in Resource-constrained Visual Sensor Networks.” I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Hairong Qi
Major Professor

Donald W. Bouldin
Co-Advisor

We have read this dissertation
and recommend its acceptance:

Gregory D. Peterson

Louis J. Gross

Accepted for the Council:

Linda Painter
Interim Dean of Graduate Studies

(Original signatures are on file with official student records.)

Efficient Image Processing in Resource-constrained Visual Sensor Networks

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Hongtao Du

December, 2006

Copyright ©2006 by Hongtao Du

All rights reserved.

Acknowledgments

I would like to extend my sincere gratitude and appreciation to all the individuals who made this dissertation possible.

First and foremost, I am grateful to my advisor Dr. Hairong Qi. Her guidance, understanding, patience, and encouragement through my graduate studies have allowed me to develop my skills as a researcher and an engineer. Without her support, this work would not have been possible. My thanks go out to my co-advisor Dr. Donald Bouldin for inspiring and encouraging me to pursue a career in computer engineering. Also I am highly indebted to Dr. Gregory Peterson and Dr. Louis Gross for serving as members of my committee and providing all the enlightening suggestions.

Special acknowledgment is given to Dr. Harold Szu at the Office of Naval Research and Dr. Wesley Snyder at the North Carolina State University for their valuable discussions and giving me the opportunities to work in so challenging but interesting projects. I would also like to thank all of the members of the Advanced Imaging & Collaborative Information Processing lab. We not only studied and worked well together, but relaxed and entertained as good friends and a team. I am thankful to all of my friends at the University of Tennessee and academic colleagues for everything they have done for me.

Finally, and most importantly, I sincerely thank my wife Xiaoling. Her endless support, quiet patience and unwavering love were undeniably the bedrock upon which the past six years of my life have been built. I thank my parents, my parents-in-law, my sister and brother for their faith in me, unconditional support, love, and encouragement through my life.

Publication History

This dissertation appears in part in the following academic journals and conferences.

- **H. Du**, H. Qi (2006). "A Reconfigurable FPGA System for Parallel independent Component Analysis", *EURASIP Journal on Embedded Systems*, Vol. 2006, pp.1-12, Article ID: 23025, Dec. 2006.

- **H. Du**, H. Qi, X. Wang (2006). "Comparative Study of VLSI Solutions to Independent Component Analysis", *IEEE Transactions on Industrial Electronics*, Vol. 53, No. 6, Dec. 2006.

- **H. Du**, H. Qi, X. Wang (2006). "A Parallel Independent Component Analysis Algorithm", *IEEE International Conference on Parallel and Distributed Systems*, Vol.1, pp. 151-160, Minnesota, July.

- **H. Du**, H. Qi, D. Bouldin (2006). "An Application-Oriented Virtual Microsensor Integration Platform", *IEEE International Conference on Networking, Sensing and Control*, pp. 874-879, Ft. Lauderdale, FL, April.

- H. Qi, L. Miao, **H. Du**, H. Szu (2005). "Fast Smart Blind Sources Separation in Mini-UAVs and its Firmware Implementation", *AIAA Infotech Aerospace Conference Advancing Contemporary Aerospace Technologies and Their Integration*, Virginia, October.

- **H. Du**, H. Qi, H. Szu (2005). "Synthesis of blind source separation algorithms on reconfigurable FPGA platforms", *SPIE Defense and Security Symposium*, Orlando, FL, April.

- **H. Du**, H. Qi (2004). "An FPGA Implementation of Parallel ICA for Dimensionality Reduction in Hyperspectral Images", *IEEE International Geoscience and Remote Sensing Symposium*, Vol. 5, pp. 3257 - 3260, Anchorage, Alaska, September.

- **H. Du**, H. Qi, G. Peterson (2004). "Parallel Independent Component Analysis and Its Hardware Implementation", *SPIE Defense and Security Symposium*, vol. 5439, pp. 74-83, Orlando, FL, April.

- X. Wang, H. Qi, S. Beck, **H. Du** (2004). "A progressive approach to distributed multiple target detection in sensor networks", *Sensor Network Operations*. Editor: S. Phoha, IEEE Press.

- **H. Du**, H. Qi, X. Wang, W. E. Snyder, R. Ramanath (2003). "Band selection using independent component analysis for hyperspectral image processing", *IEEE International Workshop on Applied Imagery Pattern Recognition (AIPR)*, pp. 93-98, Washington, D.C., October.

- **H. Du**, H. Qi, G. Peterson (2003). "Modeling mobile-agent-based collaborative processing in sensor networks using generalized stochastic petri nets", *IEEE International conference on System, Man and Cybernetics,*, vol. 1, pp. 563-568, Washington, D.C., October.

- X. Wang, H. Qi, **H. Du** (2003). "Distributed source number estimation for multiple target detection in sensor networks", *IEEE Workshop on Statistical Signal Processing*, pp. 395-398, St. Louis, MO, September.

Abstract

Visual sensor networks (VSNs) that employ content-rich 2-D images or image sequences as the basic media have been evolving rapidly in recent years. Besides the critical resource constraints that are already inherent in any micro-sensor networks, the development of VSNs also faces challenges from device design, image transmission, and onboard image processing, among which efficient onboard processing is the most difficult to tackle. The focus of this dissertation is to develop efficient image processing solutions from three aspects: to improve the time-consuming image processing algorithms using pipelined and parallel computing; to distribute the computation more effectively through novel function and image partitioning, clustering, and mapping approaches; and to implement these techniques on the virtual microsensor platform for fast onboard image processing. First, to show the efficiency of pipelined and parallel computing in algorithm improvement, we take independent component analysis (ICA) as an example and design a *parallel ICA (pICA)* method using the SPMD (Single Process Multiple Data) structure. Experimental results show that pICA accelerates the processing time by 2.4 to 5.7 times compared to the FastICA algorithm, which is the fastest existing software implementation of ICA. Secondly, in order to efficiently allocate image processing algorithms to microsensors in VSNs, we present a multi-weight operation level function model, a data dependency analysis, two resource-oriented function mapping algorithms, the *load attraction* and the *communication attraction*, with the Kernighan-Lin algorithm-based local refinements such that the execution of image processing algorithms can be closely coupled with available resources in a heterogeneous environment. A component clustering algorithm and a cyclic process model associated with the operation level function model are also proposed in order to provide appropriate granularity to the mapping process. Experimental results show that function models processed by the component clustering algorithm have the best mapping performance compared to other function models. The cyclic process modeling is very effective for complex image processing algorithms. The proposed load attraction and communication attraction mapping

algorithms respectively improve load variances and cut weights compared to existing mapping algorithms by 5 to 15 times, and both exhibit the closest performance to that of the optimal mapping. Finally, we present a *virtual microsensor platform* and implement the proposed techniques for application-specific microsensor design. While most existing microsensors are developed for general purposes, the microsensor design we propose is driven by specific applications and moves the reuse and reconfiguration features in hardware implementation to higher abstraction level. We develop an image processing intellectual property (IP) library and design four image processing IPs. Experimental results show that the performance our designs achieved is better than those of existing implementations, and the proposed virtual microsensor platform can efficiently integrate different image processing algorithms according to specific application requirements.

Contents

1 Introduction **1**

1.1 Microsensor 2

1.2 Distributed Sensor Networks 6

1.3 Visual Sensor Network 8

1.3.1 Algorithm Improvement using Pipelined and Parallel Structures 9

1.3.2 Algorithm and Data Partitioning and Mapping 9

1.3.3 Implementation of Fast Image Processing 9

1.4 Virtual Design Platform 10

1.5 Contributions 14

1.6 Document Organization 15

2 Partitioning in Pipelined and Parallel Computing for Image Processing **19**

2.1 Different Partitioning Approaches 20

2.2 Dependency Structure 23

2.2.1 Chain Structure 23

2.2.2 Tree Structure 27

2.3 Driving Force for Partitioning 30

2.3.1 Data 30

2.3.2 Function 35

2.4 Resource Constraints 38
2.4.1 Definitions and Formulations 39
2.4.2 Mapping in Homogeneous Environments 41
2.4.3 Multi-processor System 43
2.4.4 Hardware/software (HW/SW) Co-processing 47
2.4.5 VLSI 50
2.5 Summary 57

3 Modeling and Partitioning of Algorithms and Data for Image Processing 58
3.1 Multi-weight Operation-level Function Model 59
3.1.1 Model Setup 59
3.1.2 Component Clustering Algorithm 67
3.1.3 Cyclic Process Modeling 71
3.2 Data Dependency Analysis 74
3.2.1 Definitions 76
3.2.2 Independency 76
3.2.3 Uniform Dependency 77
3.2.4 Regional Dependency 79
3.2.5 Dependency in 3-D Images 82
3.2.6 Data Distribution Schemes 84
3.3 Resource Modeling 87
3.4 Mapping Procedure 89
3.4.1 Objective Functions 89
3.4.2 Resource-oriented Mapping in Heterogeneous Environments 91
3.5 Summary 100

4 Virtual Micro-Sensor Platform **101**

4.1 Virtual Microsensor Platform Structure 102

4.2 Image Processing IP Library 105

4.3 Algorithm Design and Improvement using Pipelined and Parallel Computing . 105

4.3.1 Contrast Stretching 106

4.3.2 Polynomial Approximation-based Geometric Correction 111

4.3.3 3×3 Filters 122

4.3.4 Parallel Independent Component Analysis 127

4.4 Implementation Challenges 142

4.5 Algorithm Implementation on Virtual Microsensor Platform 145

4.5.1 Contrast Stretching 145

4.5.2 Polynomial Approximation-based Geometric Correction 148

4.5.3 3×3 Filters 154

4.5.4 Parallel Independent Component Analysis 162

4.6 Microsensor Integration 164

4.7 VLSI Implementation 167

4.7.1 Implementation Procedure 169

4.7.2 Prototyping FPGA Platforms 170

4.7.3 SoC Platform 173

4.8 Summary 176

5 Experimental Results and Comparisons **181**

5.1 Experiments for Algorithm Improvement 182

5.1.1 Experiment Setup 183

5.1.2 Effect of the Number of Estimated Weight Vectors 186

5.1.3 Effect of the Number of Processors 187

5.1.4 Effect of the Data Size 189

5.1.5 Prediction Model Validation 189
5.2 Experiments for Function Clustering and Mapping 192
5.2.1 Component Clustering Algorithm Evaluation 196
5.2.2 Mapping in Heterogeneous Environment 199
5.2.3 Cyclic Process Modeling 226
5.3 IP Synthesis for Image Processing Library 234
5.3.1 Contrast Stretching 234
5.3.2 Polynomial Approximation-based Geometric Correction 240
5.3.3 3×3 Filter 242
5.3.4 pICA 246
5.4 Implementation of Microsensor Integration 249
5.5 Summary 252

6 Conclusions and Future Work **254**
6.1 Summary of Contributions 254
6.2 Directions of Future Work 257

Bibliography **259**

Vita **278**

List of Tables

2.1 Communication cost bound for data partitioning. 33

3.1 Basic component list. 60

4.1 Control point selection for the example image. 109
4.2 3×3 Filters. 124
4.3 Change of SoC design productivity. [108, 130] 174

5.1 MPI environment specification. 183
5.2 Comparison between the model prediction and experimental result. 191
5.3 Comparison of load variances of different mapping algorithms on the 7×7 filter. 204
5.4 Comparison of cut weights of different mapping algorithms on the 7×7 filter. . 205
5.5 Mean of load variances on different topologies of the resource graph. 206
5.6 STD of load variances on different topologies of the resource graph. 206
5.7 Mean of cut weights on different topologies of the resource graph. 208
5.8 STD of cut weights on different topologies of the resource graph. 208
5.9 Mean and STD of experiments on random computing resources. 210
5.10 Mean and STD of experiments on random communication resources. 212
5.11 Mean and STD of experiments on random component weights. 218
5.12 Mean and STD of experiments on random edge weights. 220

5.13 Overall processing time of mapping algorithms with and without the local refinements (us). 222
5.14 Performance comparison of FPGA implementations for contrast stretching. . . 239
5.15 Performance of Virtex II Pro implementations for blocks in polynomial approximation. 241
5.16 Performance of Virtex II Pro implementations for polynomial approximation. . 241
5.17 Design and device utilization. 242
5.18 Performance comparison of FPGA implementations. 243
5.19 Performance comparison of FPGA implementations for 3×3 filter. 245
5.20 Performance of Virtex II Pro implementations for blocks in the pICA algorithm. 247
5.21 Performance of Virtex II Pro implementations for the pICA algorithm. 248
5.22 Design and device utilization. 248
5.23 FPGAs used in comparison. 249
5.24 Performance of Virtex II Pro implementations for the integration design. 253

List of Figures

1.1 Sensor architecture [5]. . . . 3

1.2 The Motes from Crossbow Corporation. Left: MICA; Middle: MICA2; Right: MICA2DOT [46]. . . . 4

1.3 The Sensoria sGate platform [42]. . . . 5

1.4 The PC/104-based SensorView system [127]. . . . 5

1.5 Framework of partitioning. . . . 10

1.6 Microsensor design procedure. . . . 17

2.1 Partitioning categories. . . . 21

2.2 Pipelined computing structure. . . . 23

2.3 Layered graph for pipelined computing. . . . 24

2.4 Doubly weighted graph (DWG) for pipelined computing. . . . 26

2.5 Parallel computing. . . . 28

2.6 Directed dual graph (DDG) for parallel computing. . . . 29

2.7 Weighted DDG. . . . 29

2.8 Data partitioning schemes. . . . 33

2.9 Data partitioning example. (x, y, z are data with dependencies.) . . . 34

2.10 Original processing structure. . . . 36

2.11 The space-time-domain expansion. . . . 37

2.12 Directed acyclic graph (DAG). . . . 48

2.13 Control flow graph (CFG). 49
2.14 Hypergraph in circuit partitioning. 51
2.15 Multilevel bipartitioning [83]. 54

3.1 Basic operation components. 61
3.2 A multiplier & accumulator (MAC) module. 62
3.3 A 4-stage MAC module. 62
3.4 The hierarchical modeling structure. 62
3.5 Single-weight MAC module. 65
3.6 Dual-weight MAC module. 65
3.7 Single-weight 4-stage MAC module. 66
3.8 Dual-weight 4-stage MAC module. 66
3.9 Single-weight MAC module with edge weight. 66
3.10 A component clustering example for the 3×3 spatial filter. 71
3.11 Decomposition of cyclic process. 73
3.12 Cyclic process modeling. 75
3.13 3×3 spatial filter. 78
3.14 Uniform dependency. 78
3.15 Uniform dependency with weight. 79
3.16 Regional dependency with weight. 80
3.17 Distributing pixels with regional dependency. The number on each data indicates the assigned resource. 81
3.18 Different dependencies in the same region. 81
3.19 Hierarchical pixel clustering and partitioning. 81
3.20 Data dependency on image sequence or multispectral image. 83
3.21 Independency along the z axis. 83
3.22 Uniform dependencies exist along the z axis. 83

3.23 Regional dependencies exist both along z axis and on the same image. 84
3.24 Data distribution schemes. 85
3.25 Communication attraction mapping algorithm. 95

4.1 Virtual microsensor platform structure. 103
4.2 The original image. 107
4.3 Original image at red, green, blue channels and the corresponding histograms. . 107
4.4 Result image at red, green, blue channels and the corresponding histograms. . . 109
4.5 The result image of contrast stretching. 110
4.6 Traditional design. 110
4.7 The 4-pixel parallel design. 110
4.8 The 2-stage pipeline & 4-pixel parallel. 111
4.9 The transformation system between the distorted image and the corrected image. 113
4.10 Example of the polynomial approximation-based geometric correction. 115
4.11 The pipelined design of polynomial approximation-based geometric correction. 116
4.12 Structure of matrix multiplication. 117
4.13 Processing flow of matrix multiplication. 118
4.14 Structure of matrix inverse. 118
4.15 Processing flow of LU factorization. 120
4.16 Processing flow of LU inverse. 121
4.17 Structure of the polynomial approximation. 121
4.18 Effects of applying 3×3 filters. 125
4.19 Traditional design. 126
4.20 The parallel & pipelined design. 126
4.21 The parallel & pipelined design with partitioning. 127
4.22 Structure of the pICA algorithm. 135
4.23 Processing diagram of pICA on ten processors. 137

4.24 The IP design of the pICA algorithm. 138
4.25 Design of the one unit estimation process. 139
4.26 Design of the decorrelation process. 141
4.27 Internal decorrelation. 142
4.28 External decorrelation with multiple RCs in parallel. 143
4.29 Structure of the pICA. (Solid lines denote data exchange and configuration. Dotted lines indicate the virtual processing flow.) 143
4.30 Contrast stretching designs. 146
4.31 RTL schematics of contrast stretching (1). 147
4.32 RTL schematic of contrast stretching (2). (The 2-stage pipeline & 4-pixel parallel design) 149
4.33 Block of the 2-stage pipeline. 150
4.34 RTL schematic of the **W** formation block at top view. 151
4.35 RTL schematic of the **W** formation block. 152
4.36 RTL schematic of the matrix multiplication block. 152
4.37 RTL schematic of the matrix inverse block. 153
4.38 RTL schematic of the geometric correction design (Part 1). 154
4.39 RTL schematic of the geometric correction design (Part 2). 155
4.40 RTL schematic of the geometric correction design (Part 3). 156
4.41 Symbol of the geometric correction design. 156
4.42 Symbol of 3×3 filter designs. 158
4.43 RTL schematic of the traditional 3×3 filter design. 158
4.44 RTL schematic of the multiplication. 159
4.45 RTL schematic of the addition. 159
4.46 RTL schematic of the parallel & pipelined design (Part 1). 160
4.47 RTL schematic of the parallel & pipelined design (Part 2). 160

4.48 RTL schematic of the multiplier in parallel & pipelined design. 161
4.49 RTL schematic of the adder in parallel & pipelined design. 161
4.50 RTL schematic of the buffer in parallel & pipelined design. 161
4.51 RTL schematic of the parallel & pipelined design with partitioning. 162
4.52 RTL schematic of partition type 1 (V_1, V_5). 163
4.53 RTL schematic of partition type 2 (V_2, V_4). 163
4.54 RTL schematic of partition type 3 (V_3). 163
4.55 Symbols of function blocks in the pICA design. 164
4.56 RTL schematic of the pICA design. 165
4.57 Structure of the integration design. 166
4.58 Symbol of the integration design. 166
4.59 RTL schematic of the integration design. 168
4.60 Implementation procedure. 170
4.61 FPGA developments vs. Moore's Law. 171
4.62 The Pilchard platform embedded with Xilinx Virtex V1000E FPGA. 172
4.63 The Amirix platform embedded with Xilinx Virtex II Pro FPGA. 172
4.64 XUP development board. 177
4.65 Architecture of SoC design. 178
4.66 Symbol of PowerPC 405. 179
4.67 Internal structure of PowerPC 405. 180

5.1 MPI diagram of pICA. 184
5.2 (a) The AVIRIS hyperspectral image scene [103]. (b) Original 224-band spectrum curve 185
5.3 (a) The selected 50 spectral bands. (b) Spectrum curve plotted by the selected 50 bands. 186
5.4 Performance comparison between FastICA and pICA (10 processors). 187

5.5 Scalability evaluation of pICA for different number of processors. 188
5.6 Performance evaluation of pICA for different size of observation data set. . . . 190
5.7 Overall processing time comparison between model prediction and experimental results. 192
5.8 Framework of task partitioning. 193
5.9 Function models to implement contrast stretching. 197
5.10 Mapping result comparisons of two models for contrast stretching. 198
5.11 Mapping result comparison of two models for 3×3 filter. 198
5.12 Function models of the 7×7 filter. 200
5.13 Function models of the 7×7 filter with coarser granularities. 201
5.14 Mapping result comparison for the 7×7 filter models with different granularities. 202
5.15 Computing resources in heterogeneous environment. 203
5.16 Performances on random computing resources (10MHz increment). 209
5.17 Performances on random computing resources (1-1000MHz). 211
5.18 Performances on random communication resources (2MB/s increment). 213
5.19 Performances on random communication resources (1-200MB/s). 214
5.20 Performances on random component weight (increment of 10). 216
5.21 Performances on random component weights (1-1000). 217
5.22 Performances on random edge weight (increment of 2). 219
5.23 Performances on random edge weights (1-200). 221
5.24 Original mapping of pICA. 224
5.25 Mapping result of the load attraction algorithm without the local refinement on pICA. 225
5.26 Mapping result of the load attraction with the K-L local refinement on pICA. . 227
5.27 Performance comparison of different mapping algorithms. 228
5.28 Scalability of mapping algorithms on pICA. 229

5.29 Performance comparison for cyclic process modeling in pICA. 230
5.30 Scalability of number of processors. 232
5.31 Scalability of number of iterations. 233
5.32 Schematic of Virtex II Pro for contrast stretching (the traditional design). 235
5.33 Schematic of Virtex II Pro for contrast stretching (the 4-pixel parallel design). . 235
5.34 Schematic of Virtex II Pro for contrast stretching (the 2-stage pipeline & 4-pixel parallel design). 236
5.35 Layout on Virtex II Pro for contrast stretching (the traditional design). 236
5.36 Layout on Virtex II Pro for contrast stretching (the 4-pixel parallel design). . . 237
5.37 Layout on Virtex II Pro for contrast stretching (the 2-stage pipeline & 4-pixel parallel design). 237
5.38 Layout on Virtex II Pro for the polynomial approximation-based geometric correction. 240
5.39 Schematic of Virtex II Pro for traditional 3×3 filter. 243
5.40 Layout on Virtex II Pro for 3×3 filter. (the traditional design). 243
5.41 Layout on Virtex II Pro for 3×3 filter. (the parallel & pipeline design). 244
5.42 Layout on Virtex II Pro for 3×3 filter. (the Parallel & pipeline design with partitioning). 244
5.43 Layout on Virtex II Pro for the pICA algorithm. 246
5.44 Implementation procedure of the pICA algorithm on Pilchard board. 248
5.45 Performance comparisons [97, 123]. (a) Target FPGA capacity. (b) Frequencies of synthesized designs. (c) Size of observation data sets. 250
5.46 Original and result images of the integration design. 251
5.47 Schematic of Virtex II Pro for the integration design. 251
5.48 Layout on Virtex II Pro for the integration design. 252

Chapter 1

Introduction

Advances in sensor technology, MEMS fabrication, and wireless communication have inspired the development of small-size and low-cost microsensors that integrate sensing, processing, and communication capabilities together and form an autonomous entity. These advantages make it economically feasible to deploy large numbers of microsensors in a field of interest, where each sensor independently senses the environment and neighboring sensors can collaborate with each other in an energy-efficient manner. The sensors adjust their subsequent operations according to the remaining resources, re-organize upon changes such as sensor failure, sensor addition, and sensor movement, thereby establishing a *distributed sensor network (DSN)*. Although one sensor has only limited amounts of sensing and processing capabilities, through the collaboration, multiple sensors can accomplish complex tasks such as target tracking and environmental monitoring, enable real-time adaptations to environmental conditions and user requirements, and provide capabilities greater than the sum of individual ones [3]. The DSN possesses several features that have challenged many aspects of the commonly used system design and integration, including the real-time processing, the communication support for collaborative processing, and the limited resources on-board the microsensor. These new features call for a re-design of the overall microsensor structure. Unless stated explicitly, the words *sensor* and *microsensor*

will be used interchangeably in the following discussions.

1.1 Microsensor

The performance requirements of microsensors in size, power consumption, communication and processing capabilities bring extreme challenges to circuit design and system integration. Many approaches have been proposed in recent years. For examples, the SmartDust project at UC-Berkeley pushes the size of sensors to a new limit - a cubic millimeter, such that these sensors can float in the air like dust [114]. The WINS (Wireless Integrated Network Sensors) project at UCLA and the WSN (Wireless Sensing Network) project at Rockwell Science Center [120] integrate multi-modality sensing devices and a low-level signal processor on the microsensor, making it more intelligent and powerful. The Oceana Sensor Technologies (OST) Inc. has developed an ICHM (Intelligent Component Health Monitor) system, which is a smart, networked, open-architecture sensor infrastructure, for prognostics and health management of aircraft engines and industrial machines [104].

In general, a microsensor is a platform that combines sensing, data processing, wireless communication, and power components. It may also have additional application-dependent components such as a location finding system, power generator, mobilizer, etc. The general architecture of a sensor is illustrated in Fig. 1.1 [5]. The sensing unit usually includes different kinds of sensing modalities and analog-to-digital converters (ADC), through which the analog signals captured by the sensing devices are converted to digital data and then fed into the digital processing unit. The digital processing unit, which consists of processor(s) or function blocks and associated storage block(s), processes data locally and collaborates with other sensors to accomplish required tasks. The transceiver unit is used to communicate with other sensors or a base station in the network. The most important component on the sensor is the power unit, which provides the energy resources to all other components. Since there are some protocols and algorithms that require accurate information on the sensor location, a location finding sys-

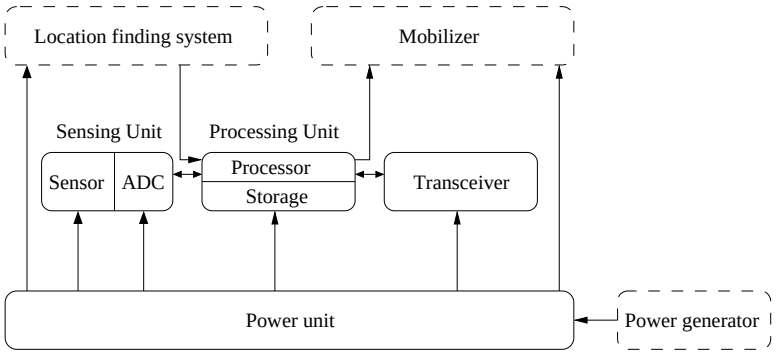


Figure 1.1: Sensor architecture [5].

tem is also commonly used. A mobilizer is useful if the sensor needs to be moved in some applications in order to carry out the assigned task.

So far, a couple of commercial companies are developing and deploying microsensor platforms for various applications. Some examples of existing platforms include the mote-based testbed from Crossbow [46], the sGate sensor platform from Sensoria [42], and the SensorView System from Ricciardi Technologies, Inc. (RTI) [121].

The motes include three generations of tiny, smart, wireless sensor platforms with sensing, processing, and communication capabilities. They are developed by UC Berkeley's research group on wireless sensors and commercialized by Crossbow Corporation. The second generation, MICA, and the third generation, MICA2 and MICA2DOT, are shown in Fig. 1.2. All of these platforms provide a plug-in sensor board with a processor running an event-driven TinyOS distributed software operating system and a two-way radio transceiver. Various sensing modalities are integrated on the platform. For example, the MICA sensor is equipped with optical, thermal, seismic/acceleration, acoustic, and magnetic sensors. In the third generation, The RH (relative humidity) and barometric pressure sensing are further added.

The Sensoria sGate development platform employs a dual-issue Hitachi SH-4 processor as both a real-time interface and a power-efficient RISC processor hosting 32-bit applications. In

Figure 1.2: The Motes from Crossbow Corporation. Left: MICA; Middle: MICA2; Right: MICA2DOT [46].

addition, a dual-mode RF modem system enables scalable wireless communications. On the sGate platform, the Linux operating system is used and up to four analog sensor inputs are supported. Common sensing modalities used on the sGate platforms include acoustic, seismic, and PIR (passive Infra-red) sensing. This sensor platform was adopted in the DARPA SensIT program [16] to detect, track, and classify moving military targets. Figure 1.3 shows the Sensoria sGate platform.

The SensorView system (shown in Fig. 1.4) is a chemical-biological point detection system infrastructure, created by RTI [121]. It can be effectively employed in biological warfare and enables flexible control of disparate detectors, collectors, identifiers, and triggers to deliver early-warning detection, identification and communication of biological warfare agents (BWAs) and events [127]. The SensorView system is developed based on the embedded PC/104 hardware components from Parvus where PC/104 is a standard for PC-compatible modules (circuit boards) that can be stacked together to create an embedded system. On top of PC/104, the system is able to embed PC architecture without having to use a bulky, less reliable motherboard- or backplane-based approach. The SensorView system integrates a variety of nuclear, biological and chemical sensors being deployed at various sensitive locations and performs the information acquisition task.

Figure 1.3: The Sensoria sGate platform [42].

Figure 1.4: The PC/104-based SensorView system [127].

1.2 Distributed Sensor Networks

A DSN is composed of a large number of sensors that are densely deployed either inside the phenomenon or very close to it [5]. Thus one important feature of the DSN is the collaboration capability among sensors, through which a complex task can be accomplished with the use of hundreds or even thousands of sensors. DSNs have many advantages over a single sensor, which can be outlined in four aspects: redundancy, complementarity, timeliness, and cost of information [99].

In DSNs, since one sensor can only sense of the environment with a certain degree of reliability, information fusion among multiple sensors is needed to provide redundant information on the environment, therefore reducing the uncertainty and promising better signal to noise ratio (SNR) [25].

The use of multiple sensing modalities on a single sensor platform can compensate the limitations of individual modalities and improve the overall sensing performance. Similarly, the complementary sensor network provides information on a large spatial area through the union of multiple small areas covered by individual sensors. In addition, DSNs may provide several aspects of the same phenomenon that can be used together to study a specific event, which is an impossible achievement with only a single sensor [26, 99].

By fusing the information among multiple sensors, the processing speed of a DSN for a given task is greater than, or at least equal to, that of a single sensor due to the parallel execution of the fusion algorithm [99], providing in-time response or decision making.

Benefiting from the current high level integration trend, the microsensor design tends to be smaller and cost less, which makes the implementation of DSNs economically feasible.

The unique features of DSNs also bring several technical challenges to the microsensor design that must be overcome before DSNs can be practically used in real applications. These challenges can be summarized as two issues: energy efficiency and flexibility.

The energy efficiency challenge is the most important issue in DSNs. Sensors are usually

supplied with batteries of limited power resources. In many applications, it is impossible to replace or recharge the battery from remote operations. Therefore, how to conserve energy and thus prolong the lifetime of the entire system plays a critical role in DSN designs.

On the other hand, sensors in DSNs are normally deployed to the field of interest in a dense and random fashion. This ad hoc deployment requires sensors to be able to flexibly communicate with each other and set up the network automatically. In the case of sensor failures due to lack of power, physical damage, or environmental interference, other sensors should be able to adaptively change the network topology and reorganize the available neighboring sensors to accomplish the task without affecting the performance of the entire network.

The advantages of DSNs over the traditional stand-alone sensor facilitate a wide range of applications in military, civilian, and environmental monitoring.

In military applications, the rapid deployment of numerous sensors makes the DSN a promising solution for command, control, communications, processing, intelligence, surveillance, reconnaissance, and monitoring tasks in military fields [39, 5].

In civilian applications, various implementations of DSNs are assisting people in improving their daily lives. For example, several DSNs have been practically used to monitor transportation patterns in urban areas [70, 91]. The Intelligent Transportation Project conducted by Muntz *et al.* is one example. The smart Kindergarten project conducted by Srivastava *et al.* [128] is another example that integrates the DSN technique, middleware design, and data management together to construct early childhood education environments. They monitor the learning process through portable badges and networked toys embedded with sensors that are connected to a control center through a wireless network.

In environmental monitoring applications, DSNs have been used to monitor the air [49], soil, and water [141]. DSNs are also used in ecosystem monitoring that aims at understanding the response of wild populations to habitats over time [34, 133].

1.3 Visual Sensor Network

To capture information from the physical environment, most sensors in DSN applications are equipped with different sensing modalities, including acoustic, seismic/acceleration, electromagnetic waves (such as optical, infra-red), magnetic fields, imaging, etc. If a sensor network employs content-rich vision-based visual sensors, it is specified as a Visual Sensor Network (VSN). VSNs have been deployed in various applications such as automatic tracking and intruder monitoring. The visual sensor refers to both the still and the video cameras that use a CCD or CMOS as the sensing device. Compared to the scalar sensor networks in which only 1-D signals are captured, the high volume of the collected images in VSNs as well as the necessary on-board content processing bring more challenges to this emerging field [76]. Besides the resource constraints and the limited computation and communication capabilities that are already inherent in any micro-sensor networks, the development of VSNs also faces challenges of its own, including those from the device design [30], the effective image transmission [106], and the efficient image processing [145], among which the fast on-board image processing is the most difficult to tackle. The goal of this dissertation work is to investigate and develop fast implementation approaches to image processing algorithms. We achieve this goal from two aspects: to improve image processing algorithms using pipelined and parallel computing; and to distribute the computation more effectively through the development of novel algorithms for image modeling, clustering, and mapping.

When we evaluate the improvement of image processing, "real-time" computation is sometimes used to measure the performance. In real-time processing, systems are subject to the operational deadlines from event to system responses [134]. The implementation is treated as failure if computations are not completed in the time period after the event but before the deadline relative to the event. In this dissertation work, we do not consider the operational deadline, but desire fast response and high performance in image processing applications.

1.3.1 Algorithm Improvement using Pipelined and Parallel Structures

Due to the increase complexity of image processing algorithms and the increase of image size caused by improved resolution, pipelined and parallel computing structures have been broadly used in the realization of fast image processing. Pipelined computing divides a sequential process into several stages of a chain structure, with each stage performing one sub-process of the entire computation. Parallel computing separates a process into several sub-processes in a tree structure according to the dependencies between each other, and allocates the independent sub-processes to multiple computing resources, with each of which performing the same or different processes at the same time.

1.3.2 Algorithm and Data Partitioning and Mapping

When applying pipelined and parallel computing to image processing, we need to consider several factors that have large impacts on design performance, e.g., the number and the type of resources used in the design (e.g., general-purpose processor, field programmable gate array (FPGA), application-specific integrated circuit (ASIC), etc.), the interconnection and communication mechanisms of resources, and the manner in which the processes are partitioned among these resources. In this dissertation work, we focus on algorithm and data partitioning. As the framework of partitioning shown in Fig. 1.5, image processing algorithms (tasks) or images (data) are first represented in a model with appropriate granularity through partitioning and clustering. This model is then mapped to homogeneous or heterogeneous resources.

1.3.3 Implementation of Fast Image Processing

For various image processing applications, three types of systems are available to implement pipelined and parallel computing, i.e., the multi-processor, hardware/software (HW/SW) co-processing, and very large-scale integrated circuit (VLSI) systems. The multi-processor system takes advantage of high speed connections between processors and utilizes multiple computing

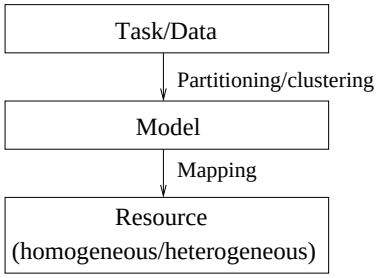


Figure 1.5: Framework of partitioning.

resources located at the same or distributed physically places. The sub-processes in pipelined or parallel computing structure are sent to individual resources, thus the computation burden is distributed from a single processor to multiple ones. The HW/SW co-processing performs the same way as the multi-processor system but uses the application-specific hardware as the co-processor to conduct time-consuming computations. In parallel to the emergence of various multi-processor and HW/SW co-processing solutions, the fast developing VLSI technology has also triggered the algorithm implementations directly on FPGA, ASIC, and system-on-a-chip (SoC). These microelectronic systems possess extensive parallel and pipelined computation capabilities. During the implementation stage, the sub-processes are synthesized with target technologies and connected with on-chip interconnections according to pipelined and parallel processing structures. The entire design is then downloaded on FPGAs or sent out for ASIC fabrication.

1.4 Virtual Design Platform

In the VLSI circuit and system domain, the virtual design platform is becoming popular as a result of the market demands for fast design and implementation. Both research institutions and industrial companies have concentrated their research on virtual platforms in recent years. The virtual platform provides various pre-qualified components to specific application designs

for fast integration and early testing, which is desired in application-driven microsensor design in VSNs. The available microsensors are mostly designed for general-purpose applications. In order to satisfy the resource limits and prolong lifetime in real environments, microsensors should include only necessary functions. We therefore move the reuse and reconfiguration features in hardware implementation and integration to the higher design level and present the virtual microsensor platform.

We classify the existing virtual platform designs into two categories: an embedded system oriented platform and a VLSI system oriented platform.

The objective of the embedded system platform is mostly to provide flexible HW/SW co-design environments that dynamically partition the system architecture. In 2002, Giusto *et al.* [66] from the Cadence Design Systems proposed a virtual integration platform for automotive electronics, in which both HW and SW components were included to constitute the overall model of the distributed functions and architectures. In 2003, Brini *et al.* [24] from STMicroelectronics presented a virtual platform that explored an optimized HW/SW partitioning on the architecture of the communication part for the VDSL modem. They used a function-architecture co-design approach (Y-Chart) that independently manage the developments of architectural and behavioral components, and then performed the HW/SW partitioning validated by performance simulations. This platform was integrated with VCC (Virtual Component Co-design) from Cadence, a system level environment for HW/SW co-design and intellectual property (IP) reuse. In order to decrease the design time, there is also a need for parallel HW/SW development and HW/SW co-verification that integrates both the hardware and the software components. For purpose of evaluating the methodologies of concurrent HW/SW design, Benini *et al.* [18] presented a so-called *virtual in-circuit emulation* HW/SW verification platform to validate the interaction between an existing core processor and some application-specific peripheral system. The idea was to co-simulate hardware blocks described in System C simulation environment and software programs running on actual prototyping board.

The VLSI system oriented platforms, as the other branch, set up reuse-based design environments in which system designs targeting FPGA or ASIC-based SoC can be developed and verified in a very efficient manner, therefore reducing the time-to-market. The virtual platforms in this category can be further divided into behavior, register transfer level (RTL), and system levels according to different abstractions in the synthesis procedure. In 1999, Ke and Truong [85] from Cadence Design Systems described an application-oriented platform-based design environment for design-for-testability (DFT). This integration platform consists of a foundation block, a virtual component (VC) library containing sets of hardware and software IP blocks, a baseline scalable architecture, and an integration environment for performance evaluation. In 2002, Givargis and Vahid [67] presented the *Platune* framework in which the designer could select appropriate architectural parameter values for a specific application in order to meet the performance requirement of low power consumption. This framework includes a collection of simulators for different structures such as CPU, cache, and peripherals. Coussy *et al.* [43] proposed a virtual IP core integration platform that provides multi-application system backbone for MPEG-2/JPEG2000 encoder. They claimed the best way to solve the problem of IP core reuse was a global methodology of integration, going from the system level performance analysis down to the synthesis step. Paulin *et al.* [112] from STMicroelectronics introduced the so-called *StepNP* system-level exploration platform to network processor design. Being a hardware architecture simulation platform, it consists of a high-level multiprocessor-architecture simulation model (including ARM v4 and PowerPC instruction-set architectures, and Standford DLX processor model using System C as modeling language), a network router application framework, and an SoC control, debugging, and analysis toolset.

As an RTL level example, Onishi *et al.* [107] proposed a tool called *VCores* (virtual cores) that uses 3-tier models: the method variable model, “Source”, consisting of a set of algorithms; the structure variable model, “Generic”, with the attribution of selected algorithms; and the “Type” model that specifies size and parameters of the specified design. The objective of the

VCores is to realize the variability in system-level design, since the abstraction level of conventional IPs are usually at the register transfer level (RTL).

System level platforms generally consider a target FPGA prototype as a backbone. In 2001, Xun *et al.* [142] developed an SoC prototyping platform that contains a microcomputer and a prototyping board on which two Altera Apex EQ20k FPGAs were embedded. The microcomputer and the prototyping board were connected through a parallel port with JTAG configuration. The overall system environment running on the microcomputer was modeled using System C. In 2003, Kearney *et al.* [86] constructed a virtual reconfigurable platform FPGA, implemented on Xilinx XCV1000E. Since the virtual platform removes the constraints of native FPGAs, it supports runtime resource allocation and enables the use of dynamic IP cores. In 2004, Bieger *et al.* [19] designed a top-down approach to implementing a real-time MP3 decoder on a reconfigurable SoC. This architecture integrates Atmel AT94K FPSLIC FPGA, an 8-bit RISC micro-controller core, and 36K Bytes SRAM within a single chip. Afridi *et al.* [1] proposed MEMS-based gas sensor VC that integrated the sensor and analog circuit into a digital shell. Since all interface connections are digital and compatible with standard CMOS technology, it could be easily integrated into SoC designs.

The advantages of the virtual design platform in efficiency, flexibility, and reusability are obvious. However, little work has been reported on its usage in microsensor design that desires the advantages of a virtual platform. For general microelectronic systems at different design levels, the use of a virtual platform is mostly for reconfiguration convenience and fast implementation purposes. But for the microsensor design, the use of virtual platform will be necessary for different applications to be implemented on resource constrained physical platforms. In this dissertation, we present the structure of an application-oriented virtual platform for microsensor design. The concept of the virtual microsensor platform is to provide a design environment so that a set of pre-qualified IPs can be selected with pre-defined metrics, and address earlier validation of various applications in regular and faulty conditions.

1.5 Contributions

In this dissertation, we focus on the algorithm improvement using pipelined and parallel structures, the development of modeling, clustering and mapping algorithms, and the implementation of these techniques on the virtual microsensor platform for fast on-board image processing in VSNs.

Parallel image processing. Many image processing algorithms require time-consuming computation due to either algorithm complexities or the use of large data sets. To solve this problem, we can employ available computing resources and conduct parallel processing. In this dissertation, we take independent component analysis (ICA) as an example and propose a *parallel ICA (pICA)* method using the SPMD (Single Process Multiple Data) structure. Experimental results on NASA AVIRIS 224-band hyperspectral images and a MPI environment with 10 computers show that pICA accelerated the overall processing time by 2.4 to 5.7 times compared to the FastICA algorithm, which is the existing fastest software ICA implementation. The performance comparisons conducted on Xilinx Virtex FPGA show that pICA can handle much larger data sets in various applications. We will present the pICA algorithm in Chapter 4.

Innovative clustering and mapping. Given a parallel image processing algorithm, it is necessary to partition the involved processes and images so as to efficiently allocate computations to specific computing resources. We first analyze the partitioning in pipelined and parallel computing for image processing from the computing dependency, the driving force, and the computing resource perspectives. Secondly, we propose a multi-weight operation-level function model in order to partition image processing algorithms in finer granularity. In this hierarchical model, an image processing algorithm is decomposed into basic arithmetic operations. These operations, represented as components, are then connected by edges according to the processing flow. Multiple weights are assigned to components and edges in respect to performance parameters. A component grouping algorithm and a cyclic process modeling are also proposed in order to provide function models with appropriate granularity to the mapping process. In

addition, we analyze data dependency respectively from the independency, uniform, and regional dependencies for 2-D and 3-D images. Finally, we propose the load attraction and the communication attraction mapping algorithms, and the K-L algorithm-based local refinement for function mapping in a heterogeneous environment. The proposed algorithms map functions to resources for different objectives, including balancing load, minimizing communication or overall processing cost. Experimental results and comparisons show that function models processed by the component clustering algorithm have the best mapping performance compared to other function models. The cyclic process modeling was very effective for complex image processing algorithms. In our case study, the proposed load attraction mapping algorithm improves load variances of other existing mapping algorithms including OLB, MET, Min-Min, Max-Min, and Duplex by 5 to 15 times, and is close to the optimal mapping. The proposed communication attraction mapping algorithm improves cut weights of other existing mapping approaches by 4.15% to 47.22%, and is also close to the optimal mapping. These will be described in detail in Chapter 3.

Virtual microsensor platform. In order to efficiently implement various image processing algorithms on VSNs with specific requirements, we develop the virtual microsensor platform in which a user can specify the design of the microsensor and quickly implement the design using the image processing IP library. In the image processing IP library, we develop several IP designs that employ pipelined and parallel computing structures for various image processing algorithms. Performance comparisons show that the performance our designs achieved is better than those of existing implementations. The development of the virtual microsensor platform will be presented in detail in chapter 4.

1.6 Document Organization

A complete microsensor design procedure for fast image processing includes the following steps: algorithm improvement using pipelined and parallel computing structures, algorithm and

data partitioning, IP development and system integration on the virtual microsensor platform, and implementation on a physical device. The overall structure of the system design is illustrated in Fig.1.6.

The dissertation is organized based on the microsensor design procedure:

Chapter 2 reviews existing partitioning, clustering, and mapping approaches in pipelined and parallel computing. The discussion is organized from aspects of the computing dependency, the driving force, and the computing resource.

Chapter 3 presents a hierarchical multi-weight operation-level function model that represents image processing algorithms using components and edges. A component grouping algorithm and the cyclic process modeling are also proposed in order to provide function models with appropriate granularity to the mapping process. Since the focus of this dissertation is image processing, the data dependency is also analyzed from aspects of independency, uniform, and regional dependencies. In addition, we present the load attraction, the communication attraction mapping algorithms, and their local refinements whose objectives are to balance computing loads, minimize communications, and minimize the overall processing cost, respectively.

Chapter 4 describes the structure of the virtual microsensor platform and presents its innovations, design issues, as well as the feasibility issue. During the development of the virtual microsensor platform, we focus on the digital processing section that consists of an image processing IP library. We design four image processing IPs, including the contrast stretching, the polynomial approximation-based geometric correction, the 3×3 filter, and the pICA algorithm. Since the algorithm improvement is closely related to the IP block development, the derivation and the performance model of the pICA algorithm are also presented in this chapter. We also present the implementation procedure from the virtual microsensor platform to the prototyping FPGAs and the final SoC in this chapter, and introduce the Pilchard and the Amirix boards as two FPGA prototyping platforms, the XUP board as the target SoC platform.

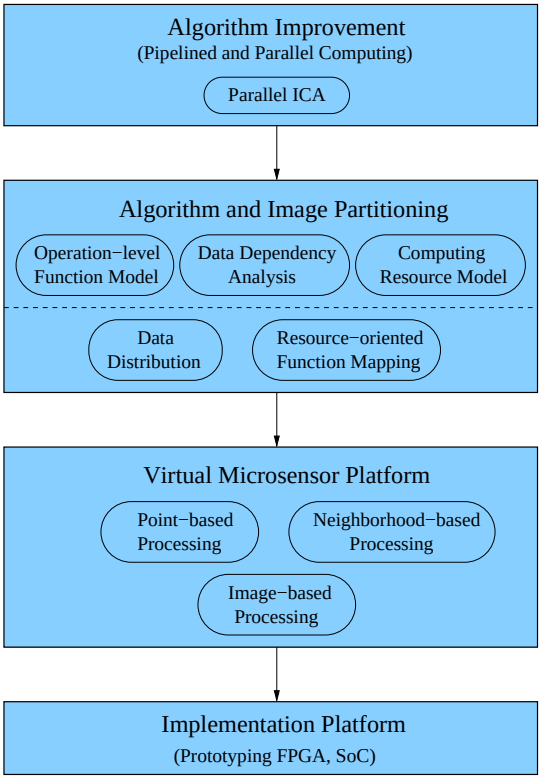


Figure 1.6: Microsensor design procedure.

Chapter 5 exhibits experimental results based on the theoretical analysis in Chapters 3 and 4 for algorithm improvement, function clustering and mapping, and IP block synthesis. Three comparison scenarios are specified, including that between the pICA algorithm and the FastICA algorithm, that between the proposed component clustering algorithm, the cyclic process modeling, the load attraction mapping algorithm, the communication attraction mapping algorithm, the local refinement and existing approaches, and that between the proposed IP designs in the image processing library and other existing designs.

Chapter 6 summarizes the breakthroughs of this work and discusses some potential developments in the future.

Chapter 2

Partitioning in Pipelined and Parallel Computing for Image Processing

As discussed in Chapter 1, three factors have large impacts on the performance of pipelined and parallel computing, i.e., the number and type of resources used, the interconnection and communication mechanisms between resources, and the manner in which the processes are partitioned and distributed to resources. Among these impact factors, the partitioning is the biggest challenge in the efficient implementation of image processing algorithms and requires appropriate designs according to resource constraints.

In VSNs, designs are tightly coupled with specific applications in order to make the best usage of the limited resources. Therefore, we review partitioning approaches from the application point of view, and emphasize the impact from the aspects of the dependency structure, the driving force, and the resource constraints. The dependency structure refers to the temporal relationship between processes in an application, which leads to pipelined computing in the chain structure and parallel computing in the tree structure. The driving force refers to the cause resulting in time-consuming computations in an application, which includes the effects from the data and the function. The resources that execute applications include the multi-processor, HW/SW

co-processing, and VLSI systems, whose constraints restrict the execution of the computation. In this chapter, we review some existing partitioning, clustering, and mapping approaches from these three perspectives of applications. Since we study partitioning from the application point of view, different clustering and mapping approaches will be mixed in individual sections.

2.1 Different Partitioning Approaches

Partitioning follows the divide-and-conquer design philosophy by decomposing problems into small partitions in a manner appropriate to the application [81]. Partitioning consists of the *spatial partitioning* and *temporal partitioning* [13]. Spatial partitioning evaluates the features of computing resources and assigns processes to the suitable resources according to individual resource specifications. Processes are divided and mapped with respect to computing resources in such a way that the overall performance corresponding to resource constraints such as the processing speed, the available power, and the capability of the utilization area is optimized. Temporal partitioning examines the level of concurrency in the design, and distributes an application to different computing resources such that processes are executed simultaneously on the resources.

An important precept following the mission statement of partitioning is that the objectives and the algorithms must fit applications, not the other way around [81]. Therefore, we study various existing partitioning approaches with respective to features of applications, specifically from the dependency structure, the driving force, and the resource constraint aspects, as demonstrated in Fig. 2.1.

The dependency structures are separated into the pipelined and the parallel structures, which may be respectively represented as chain and tree in computing. The chain-structured computation divides a sequential computing procedure into several ordered stages, each of which performs one process of the entire procedure. The assembly line broadly used in industry manufacture is one example. In tree-structured computation, independent processes can be sent to

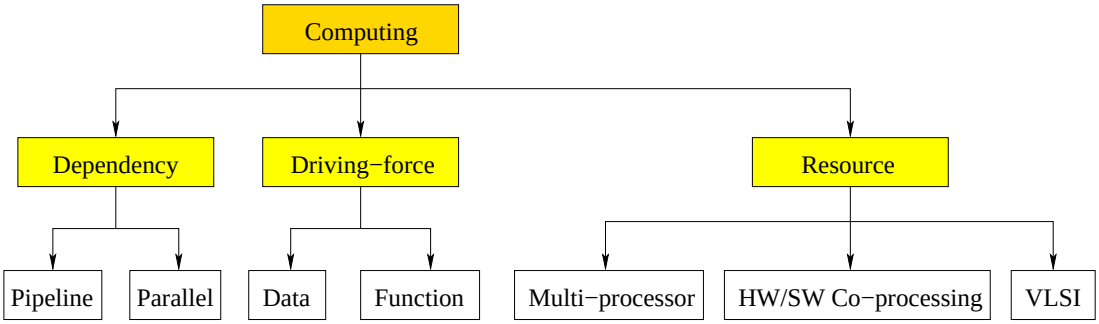


Figure 2.1: Partitioning categories.

multiple computing resources, which simultaneously perform the same or individual processes at the same or different levels with synchronization. In the matrix multiplication operation, for example, the matrix can be divided into several sub-matrices and multiplications are executed in parallel.

The driving force can come from two sources according to the partitioning criterion: data and function. Data partitioning considers how to divide one data set into several subsets for multiple computing resources and how to coordinate data flows along different directions such that appropriate data can be assigned to the suitable computing resource at the right time. Data partitioning is a popular topic in high volume data processing, such as those performed on gray-scale images, videos, multispectral and hyperspectral images. Function partitioning deals with how to perform different functions of one task on different computing resources at the same time. Function partitioning is broadly used in applications with complicated processing procedures such as edge detection.

The resource constraints reside in the system platforms that implement the application. We will study three implementation systems, including multi-processors, hardware/software (HW/SW) co-processing, and VLSI. The multi-processor system can be homogeneous or heterogeneous systems, and performs software implementations. The HW/SW co-processing system takes care of the hardware and the software designs at the same time. In the multi-processor and the HW/SW co-processing systems, both the computation time on individual tasks and the

communication time between each other are taken into account. The VLSI system performs hardware implementations, in which the communication time between processes may be ignored if the frequency is not very high.

Besides the dependency structure, driving force, and resource constraints, partitioning approaches can be analyzed in more detail by considering the following criteria: modeling graphs, granularity, partitioning strategy, and cost function [31]. The modeling graphs define the representation for the task and the computing resource. The granularity refers to the partitioning grain levels that could include process, subprogram, basic block, operation levels, or mixtures of them. The partitioning strategy consists of the manual/interactive partitioning and the automatic partitioning where the latter initializes a task using software components, hardware components, or both. The cost functions of partitioning approaches make use of different parameters from either the performance expectation or the resource constraints. Some methods only use the execution time and the communication time, while others consider the design area [53, 110] and the power consumption [28, 126] as well.

From the analysis point of view, the partition problem is similar to the network routing problem, since solutions of both intend to find the optimal or the shortest path between the start and the end points. Given a variety of criteria for optimality, finding an efficient partitioning is computationally equivalent to the NP (non-deterministic polynomial-time)-complete graph partitioning problem. The graph partitioning problem is to divide the vertices of a graph $G = (V, E)$ (where V is the set of nodes and E is the set of edges in the graph) into two or more subsets with equal size such that the number of edges between every two subsets is minimized. For an NP-complete problem, no known algorithm exists to solve it in polynomial time, but there is also no proof that no such algorithm exists. The relationship between the number of input parameters to the problem and the problem complexity is generally regarded as exponential.

2.2 Dependency Structure

From the dependency point of view, computation architectures can be modeled as a chain structure and a tree structure. Both serial computing and pipelined computing follow a chained structure, and both parallel computing and distributed computing are representatives of a tree structure. Dependencies between every two processes determine the overall pipeline structure, and independencies among individual processes determine the overall parallel structure. Hence, the discussion of partitioning approaches from the dependency point of view focuses on pipelined and parallel computing.

2.2.1 Chain Structure

As a representative of the chain structure, pipelined computing exploits temporal parallelism. According to the dependency on the chain, the task is divided into concurrently executing processes. At each stage, processes are executed on the computing resource using the input data and results are passed to the succeeding resource. Since each stage behaves like a filter, it is also called a *filter* or a *transformer* in pipelined computing. Figure 2.2 illustrates a chain structure. Note that resources are not necessarily connected in a chain structure. Detailed discussions of the computing resource will be left to Sec 2.4.

The biggest challenge in performing partitioning in pipelined computing is how to reduce the execution time of the bottleneck stage that is defined as the stage with the longest execution

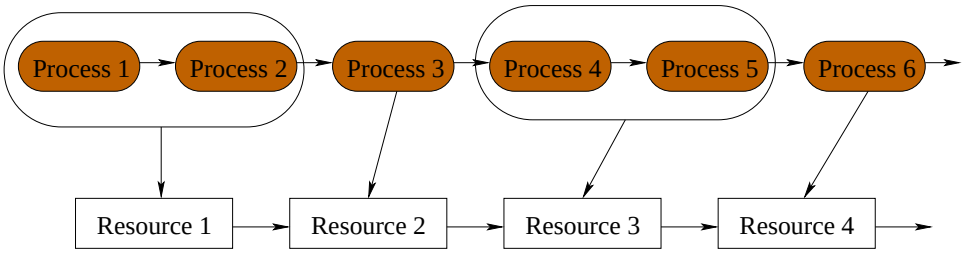


Figure 2.2: Pipelined computing structure.

time. In order to discuss partitioning approaches for pipelined computing, it is necessary to model the processes first. The layered graph, as shown in Fig. 2.3, has been a commonly used partitioning analysis tool in recent decades. Given m processes and n computing resources, node $< i, j >$ denotes a sub-chain in which processes i to j are executed on the same computing resource, and $1 \leq i \leq j \leq m$. There are totally $2m + \frac{(m+1)m}{2}(n-2)$ nodes in the model. A layer corresponds to one resource that may contain more than one sub-chain. Edge e that connects node $< i, j >$ and $< j+1, * >$ represents not only the dependency between two sub-chains but also the transmission from one resource to another. In the layered graph, nodes $< 1, * >$ in the first layer connect to the start node S, and $< *, m >$ in the last layer connect to the end node T. The weight assigned to a node refers to the execution time for the current sub-chain on current resource, and the weight assigned to an edge refers to the communication time between two resources required by the two connected sub-chains.

Based on the layered graph, Bokhari [21] proposed the minimum bottleneck path algorithm to solve the partition problem. For each node i, label $L(i)$ is defined to record the minimum execution time, over all paths ending at i. For each edge e, label $W(e)$ is marked as

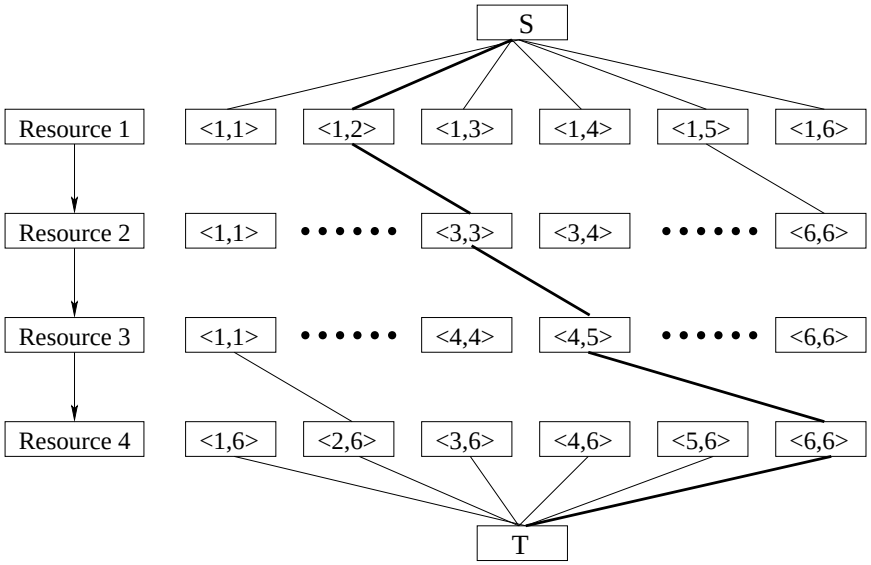


Figure 2.3: Layered graph for pipelined computing.

the weight (communication time) for connecting the node a at the higher layer and the node b at the lower layer. Initially, the nodes in the first layer are assigned zero weight, and all others are assigned with infinite weight. In order to mark the bottleneck process and identify the path with the minimum execution time, the algorithm then replaces the label on node b by $min\{L(b), max[W(e), L(a)]\}$. This procedure is repeated until all nodes are updated. Once the end node T is labeled, the path is then found by backward tracing. Since one node may have at most $m-1$ connections from the higher layer, the overall complexity of the minimum bottleneck path algorithm is $O(m^3 n)$.

According to the analysis on the layered graph, the partition problem can be obviously solved by more efficient approaches such as dynamic programming [71]. Let t_i^j denote the time required to run process i on resource j, c_i denote the amount of data to be transmitted from i to $i+1$, v_j denote the transmission speed between resources j and $j+1$. Suppose processes k to i are assigned to resource j, the total execution time is represented as t_{ki}^j and $t_{ki}^j = \sum_{l=k}^{i} t_l^j + \frac{c_i}{v_j}$. Suppose the first i processes are assigned to the first j resources, f_i^j is defined as the time consumed by the bottleneck resource, and p_i^j is defined as the pointer to the first process on the j^{th} resource. Hence, the overall objective of this algorithm is to minimize f_m^n. The dynamic programming on the partition problem consists of four steps: (1) initialize each sub-chain by t_{ki}^j, where $j \le k \le i \le (m-n+j)$ and $j = 1, 2, \cdots, n$; (2) initialize the first resource by $f_i^1 = t_{1i}^1$ and p_i^1 for $i = 1, 2, \cdots, m-n+1$; (3) calculate the Bellman equation $f_i^j = \min_{k=j,j+1,\cdots,i} \max\{f_{k-1}^{j-1}, t_{ki}^j\}$ for $i = j, j+1, \cdots, m-n+j$ and $j = 2, 3, \cdots, n$, and set $p_i^j = k$ where k is the smallest index holding the above equality; (4) assign processes $p_m^n, p_m^n + 1, \cdots, m$ to the source n, then set $m = p_m^n - 1$, decrease n by 1, and then repeat this step until all processes are assigned. From the preceding description we can obtain the complexity step by step. Step 1 requires $O(m^2 n)$ time since the execution time is progressively updated instead of being computed every time. Step 2 takes $O(m)$ time. Step 3 requires $O(m^2 n)$ time, where each f_i^j consumes $O(m)$ time and the total number of f is mn.

And Step 4 takes $O(n)$ time. Hence the time complexity of this approach is $O(m^2 n)$.

By examining the layered graph, we find that many edges are obviously unnecessary and can be ignored in path seeking. Hence, it is worth searching for more efficient modeling tools in order to further reduce the complexity. The doubly weighted graph (DWG) is one example. The notion of the doubly weighted graphs was defined by Lawler [93] for combinatorial optimization problems such as the shortest paths in networks with specified transit time. For the partition problem in pipelined computing, DWG is defined as $D =< N, E >$, where N denotes the set of nodes and E denotes the set of edges. In DWG, the node represents the sub-chain, the edge represents the dependency or the transmission between two resources, and the path P represents the processing flow. As shown in Fig. 2.4, each edge in DWG is assigned with a weight pair $< \sigma(e), \beta(e) >$, where $\sigma(e)$ is the edge weight, and $\beta(e)$ is the bottleneck weight [21]. Let $S(P) = \sum \sigma(e)$, $S(\oslash) = \infty$ be the sum of edge weights, and $B(P) = \max[\beta(e)]$, $B(\oslash) = 0$, be the bottleneck weight along the path P. Then, the optimal partitioning of the chain is equivalent to the searching for the optimal path between the start node S and the end node T, for which the sum-bottleneck (SB) weight that is defined as $max\{S(P), B(P)\}$ is minimal.

The optimal SB path algorithm is one approach for the previously described path finding [21]. Suppose a DWG has m nodes and n edges. The optimal SB algorithm first defines a

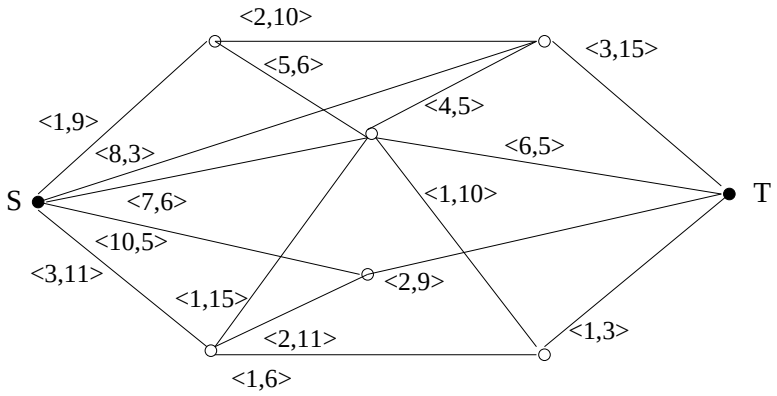


Figure 2.4: Doubly weighted graph (DWG) for pipelined computing.

bottleneck weight B_T in order to remove redundant edges with $\beta(e) > B_T$. If a path exists between S and T, then the shortest path weight S is returned. The Dijkstra's algorithm which takes $O(mlogm)$ time (m is the number of nodes) is used in the optimal SB algorithm to find the shortest path. This procedure is iterated with B_T decreasing until no path can be found between S and T. Finally, the sum weight S and the bottleneck weight B are plotted along the B_T axis. The sum weight S is a non-increasing curve, and the bottleneck weight B is a non-decreasing curve. The intersection of S and B is therefore the optimal SB path. Since the number of the distinct values B_T may have is less than the total number of edges, the complexity of this algorithm is $O(m^2logn)$.

2.2.2 Tree Structure

Parallel computing is best modeled in a tree structure. Parallel computing exploits spatial parallelism by utilizing several computing resources and executing multiple independent processes simultaneously, which are represented as branches in the tree structure. Processes on individual branches depend only on their parent processes. The independency between processes at the same layer permits the computing burden to be distributed from a single computing resource to multiple ones. Hence, the partition objective in parallel computing is to evenly distribute processes into functional subsets of the same workload. In the meanwhile, the communication time between subsets should be minimized. A typical partitioning is shown in Fig. 2.5, where each node represents one process. In most cases, the parent process at the top of the tree is assigned to a powerful computing resource, called the *master* or the *host*, and works as a coordinator in the entire task. The remaining processes on the branches and leaves of the tree are assigned to other computing resources, called the *slaves* or the *nodes*. Unless stated explicitly, the words *master* and *host*, *slave* and *node* will be used interchangeably in this chapter. In order to minimize the communication time along individual branches, if a process is assigned to one resource, all of its children and grandchildren processes are assigned to the same resource unless the resource

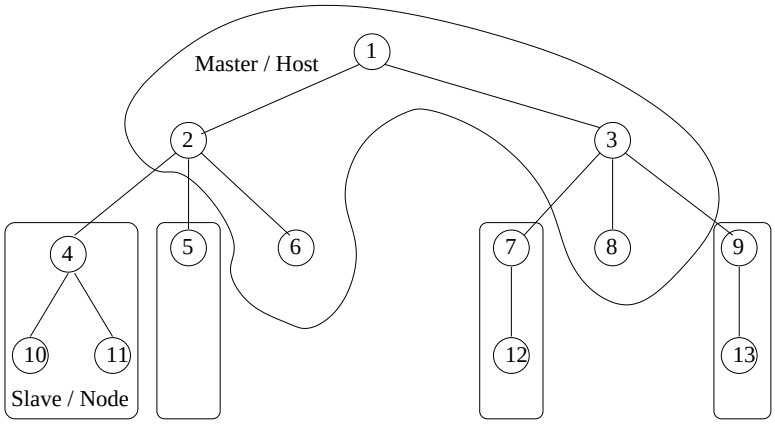


Figure 2.5: Parallel computing.

cannot handle all of them.

Similar to the chain structure, the partitioning of the tree structure also requires an appropriate graph to model the task at first. The directed dual graph (DDG) is a commonly used tool. Figure 2.6 demonstrates the partitioning on the tree previously shown in Fig. 2.5. The extra nodes, shown in square, are added to the tree such that the nodes representing processes are isolated from each other. The extra nodes are marked from the left to the right. The direction of the graph is from the lower numbered nodes to the higher numbered ones.

In order to apply the SB path algorithm to the partitioning of tree structured tasks, Bokhari [21] remodeled Fig. 2.6 into a weighted DDG, as shown in Fig. 2.7, where h_i, $i = 1, \cdots, 13$ denotes the processing time of node i, and the weight assignment follows the left-hand principle. Hansen and Lih [71] improved this minimum sum-bottleneck path algorithm with Dijkstra's algorithm, and used a heap structure to store the temporary labels, a linked list structure to store the graph. Suppose there are m nodes in the tree structure. This method reduces the time complexity from $O(m^2 logm)$ in the SB path algorithm to $O(logm)$.

In recent years, many algorithms exploit the partition problem in both parallel and pipeline parallelism at the same time, such as pipelined data parallel processing [89, 90].

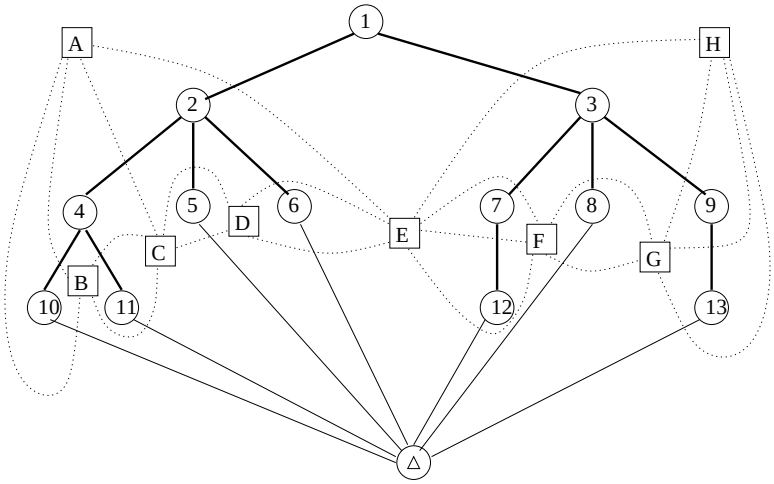


Figure 2.6: Directed dual graph (DDG) for parallel computing.

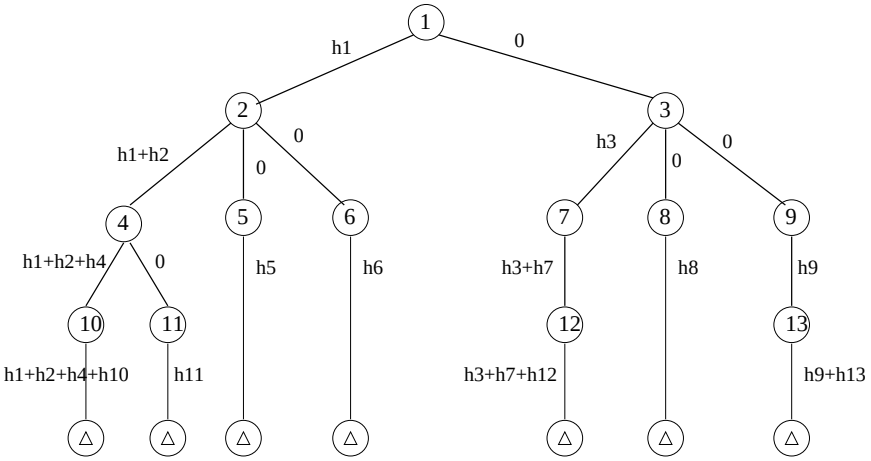


Figure 2.7: Weighted DDG.

By combining the chain structure and the tree structure we identify the following four cases [21]:

1. Chain-structured pipelined programs over chain-connected systems;
2. Multiple chain-structured parallel and pipelined programs over single-host multiple-satellite systems;
3. Multiple arbitrarily structured serial programs over a single-host multiple-satellite system;
4. Single-tree structured parallel programs over single-host multiple identical satellites systems.

Having analyzed the dependency structures, we move the discussion on partition problems to the other two perspectives: the critical driving force that directs the partitioning of the computation, and the resource constraint that restricts the execution of the computation.

2.3 Driving Force for Partitioning

In recent years, the increase in algorithm complexity for image processing and the increase of image size caused by improved resolution are the main causes of time-consuming computations in an application. These two factors put restrictions on the partitioning and clustering approaches that can be adopted.

2.3.1 Data

In many image processing applications, a data parallelism is appropriate to the algorithms that perform the same processes on large data sets. Parallel computing distributes data among multiple subroutines, each of which works on one subset of data only. Such applications have a common challenge, *data partitioning*.

From the autonomous point of view, many data-concerned partitioning algorithms do not include global synchronization. The arrivals of data blocks initiate and synchronize the processes in the task. The entire system in execution is modeled as a network of computing resources linked together by data streams. Such approaches are labeled as *data-driven*. As King [89] stated for the pipelined data-parallel algorithms, the most important characteristic of executions on multiple computing resources is the asynchronous large-grained data flow.

Since the data transmitted between computing resources are the focus of this category, the communication time consumed by data transfer is the first concern in data partitioning. Data transmitted between computing resources are partitioned into blocks, defined as the *data blocks*. The size of data block determines the granularity of the algorithm. The flows of data blocks, along with the computing resources, form the data stream [144].

In data partitioning, data dependency and granularity determine the overall performance. Before discussing them in detail, it is necessary to describe the data-related computation architectures and general partitioning schemes.

Based on the relationship between the instruction delivery mechanism and the data stream, data concerned computation architectures are classified into Single Instruction Flow Single Data Stream (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data Stream (MIMD), which is called *Flynn's taxonomy* [111]. SISD is the basic architecture for serial computing. An example of this architecture is the stand-alone computer with single processor that serially executes processes by itself without any assistance. MISD sets up a pipelined computing structure that resembles the assembly line used in car manufacture, in which data are progressively processed by consecutive instructions. SIMD builds up parallel computing environments with one master processor and multiple slave processors. Such environments have either shared or distributed memory structures. Popularly used parallel computing environments include the message passing interface (MPI) [131] and the parallel virtual machine (PVM) [65]. MIMD constructs a distributed computing envi-

ronment such as the distributed sensor network, where different data are respectively processed by individual instructions.

General data partitioning schemes can be identified as [45]: (a) scatter, (b) contiguous point, (c) contiguous row, and (d) block, as shown in Fig. 2.8. Crandall and Quinn [45] mathematically summarized the communication costs of these four schemes. The communication patterns considered in their work are in two forms: the successor/predecessor (S-P) pattern and the north/south/east/west (NSEW) pattern. The upper bounds of communication for each partitioning scheme in correspondence to the different communication patterns are listed in Table 2.1, where α is the message preparation latency, β is the transmission speed (Byte/s), p is the number of processors, n is the number of data, and d is the length of each data item to be transmitted. In the block partitioning, if the processors are not homogeneous or $\sqrt{p}$ is not an integer, we can use the binary recursive partitioning scheme and the general quadrilateral decomposition scheme, in which the largest numbers of communication are respectively $6p-4$ and $8p-4$.

The above partition schemes are very general. In order to further improve the performance, we need to consider the effects of the data dependency and the granularity. Given a large data set or image, a straightforward data partition scheme is to divide the data set or image into rectangle blocks, each of which is assigned to one computing resource, as shown in Fig. 2.9(a). But this scheme may result in a lot of communications between edge pixels. Taking a 3×3 filter in image processing as an example, if the calculation of y depends on the result of x, and z depends on those of x and y, then the calculation of one pixel requires two communications at every iteration. The overall delay caused by communication therefore decreases, or even dismisses, the speedup achieved by parallel or pipelined structures. Such conflict is originally caused by the data dependency between edge pixels that are assigned to different computing resources.

1	2	3	4	1	2	3	4
2	3	4	1	2	3	4	
3							
4							

(a) Scatter.

1	1	1	1	1	1	2	2
2	2	2	2	2	3	3	3
3	3	3	3	3	3	3	3
3	4	4	4	4	4	4	4
4	4	4	1	1	1	1	1
1	1	1	1				

(b) Contiguous point.

1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
1	1	1	1	1	1	1	1

(c) Contiguous row.

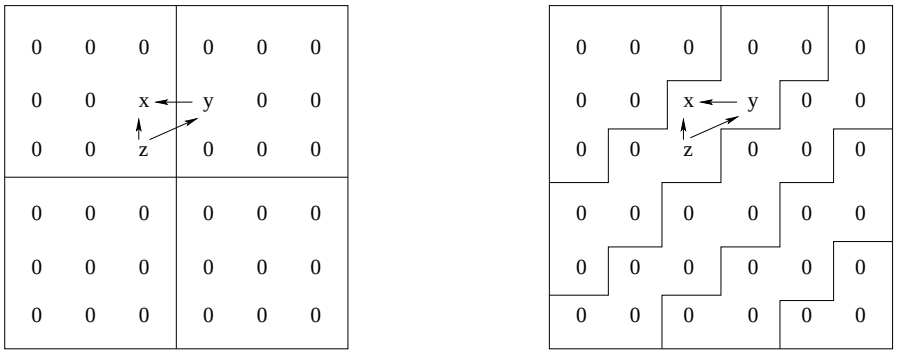
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
3	3	3	3	4	4	4	4
3	3	3	3	4	4	4	4
3	3	3	3	4	4	4	4

(d) Block.

Figure 2.8: Data partitioning schemes.

Table 2.1: Communication cost bound for data partitioning.

Partitioning Scheme	S-P	NSEW
Scatter	$2\alpha(p-1)+\beta n^2 d$	$4\alpha(p-1)+\beta n^2 d$
Contiguous point	$4\alpha+4\beta d(p-1)$	$6\alpha+4\beta d(p-1)+2\beta dnp$
Contiguous row	No communication	$2(\alpha+\beta dnp)$
Block (homogeneous)	$2(\alpha+\beta dn\sqrt{p})$	$4(\alpha+\beta dn\sqrt{p})$



(a) Straightforward partitioning. (b) Granularity control partitioning.

Figure 2.9: Data partitioning example. (x, y, z are data with dependencies.)

A simple solution to this problem is to partition the image along the reverse diagonal direction, as shown in Fig. 2.9(b). In this partition structure, all of x, y, and z are assigned to the same resource, thereby eliminating the communications. Even if the dependency spreads across the entire image, this partition structure still reduces the amount of overheads and the overall communication time. We will discuss this issue in more detail for multi-processor systems in sec. 2.4.3.

If a data dependency exists in the parallel structure, we can also conduct a transaction from the parallel structure to the pipelined structure, which is in fact a special case of space-time mapping. The grouping along a particular direction is called *projection*, which transforms an n-dimensional structure into an m-dimensional structure, and $n > m$. The direction of projection indicates the progress of time. The projected structure depicts the final spatial layout.

As the other consideration in data partitioning, the granularity is important to the balance between communication and computation. The computation/communication ratio can be controlled by adjusting the granularity. A large granularity, i.e., the size of the entire data block, reduces the degree of parallelism. Because large data blocks are allocated to individual processors, the time spent on computation increases and little overlapping exists between processors.

On the contrary, if the granularity is small, the degree of overlapping will increase, and the communication time related to the overhead will rise correspondingly. Therefore the selection of granularity presents a trade-off problem between computation and communication/overhead.

If granularity is small, we can apply clustering algorithms to images. The goal of clustering is to reduce the large amount of data by grouping in smaller items to blocks. Many off-shelf clustering algorithms are available in pattern recognition and data mining fields [79]. Commonly used clustering approaches in partitioning include hierarchical clustering, k-clustering, self organizing maps (SOM), etc. The hierarchical clustering [136, 2] first produces a hierarchy according to data dependencies. Then the clustering process starts from leaves to root in bottom-up sequence to satisfy certain clustering cost function such as minimizing variances between clusters. The goal of k-clustering or k-means algorithm [58, 68] is to minimize dependency in data between data in different clusters while maximizing dependency within each cluster. The k-clustering first initializes several clusters, and then assign dependent data to the same cluster. The SOM [58, 27] maps data into a k-dimensional space following some specific geometrical topology such as grids and rings. Data are initially placed at random, and then iteratively adjusted according to dependencies along the k-dimensional space.

In general, the improvement of data partitioning depends on the optimal selections of the granularity of data blocks and the efficient communication mechanisms, which consider data dependency and desire to reduce overhead and overlapped operations.

2.3.2 Function

Similar to the data parallelism, processes with function parallelism can be programmed by using multiple independent subroutines. In function partitioning, we evaluate the complexity of individual processes with computation and communication to decide the optimal assignments. Approaches in this category solve the partition problems with considerations of either the process or the computing resource. If the process is concerned, then the algorithm itself is first

analyzed. Processes in the algorithm are clustered to satisfy the objectives such as minimizing communication and balancing the computation. This is a common partitioning routine for homogeneous computing environments like VLSI systems.

If the computing resource is the most important, the first consideration is the computing capability of individual resources and the communication capability between each others [109]. Processes are then optimally assigned to computing resources. This method is generally used for implementations on heterogeneous computing environments. Bakshi and Gajshi in [12] described a system level approach for the pipelined implementation of HW/SW co-designs. They first mapped processes to the processor and pipelined the system specification. Under the assumption that the hardware resource can execute two tasks in parallel, they scheduled the processes in each stage using a list scheduling-based pipelined scheduler.

As another solution, the space-time-domain expansion is the most straight-forward partitioning approach. It sacrifices the processing time to meet the performance requirements. Figure 2.10 shows an example with the pipelined computing structure. When m and n are above

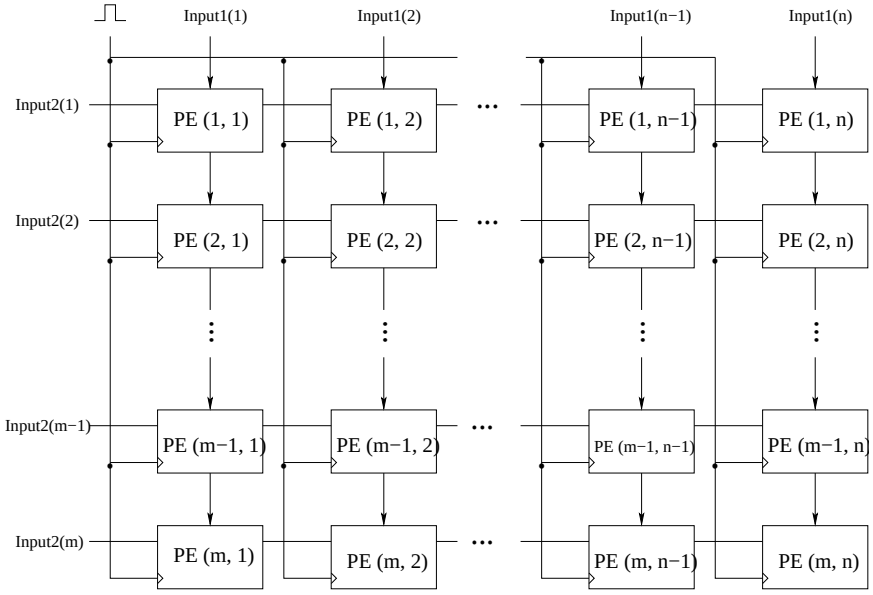
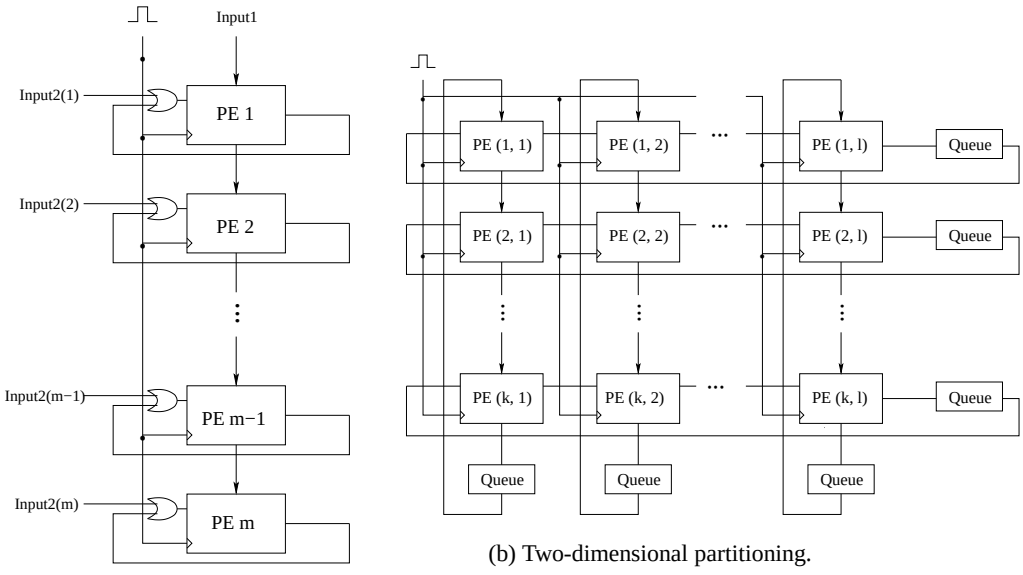


Figure 2.10: Original processing structure.

certain number depending on the complexity of the processing element (PE), the size of original processing is too large for computing resources such as FPGA to accommodate or execute without partitioning. The space-time-domain expansion solves this problem by fixing the processing size according to the available resources. Figure 2.11 demonstrates the one-dimensional partitioning and the two-dimensional partitioning of the original processing structure. The one-dimensional partitioning limits the processing size to one column at a time. The data are repeatedly fed to the system until the process finishes. This structure increases the time complexity by n, where n is the number of columns. The two-dimensional partitioning limits the processing size to a two-dimensional subset of the original processing. Similar to the one-dimensional partitioning, the two-dimensional partitioning increases the time complexity by $\lceil \frac{m \times n}{k \times l} \rceil$, where m and n are the number of rows and columns of the original data set, k and l are the number of rows and columns of the subset. Another drawback of the two-dimensional partitioning is the need for $(k + l)$ queues or buffers that store the intermediate results.



(a) One-dimensional partitioning.

(b) Two-dimensional partitioning.

Figure 2.11: The space-time-domain expansion.

The space-time-domain expansion structures have been broadly applied to various process-driven pipelined computing [125]. In 1989, Cheng *et al.* [37] studied the partition problem for an iterative digital picture comparison process. As the preceding discussion on the chain-structured system, pipelined computing is appropriate to this task. In their work, the authors first proposed a 2-D pipelined VLSI architecture that consists of $m \times n$ PEs. The data move from one PE to another at one time unit. Within the one time unit, $m \times n$ data are processed simultaneously, where m is the number of intensity levels of the input digital picture, and n is that of the reference digital picture. In other words, this architecture distributes the computation from single computing resource to an $m \times n$ VLSI matrix, and reduces the time complexity from $O(m^2 \times n)$ to $O(max(m, n))$. Since the processing size cannot meet the VLSI capacity constraint, both the one-dimensional and two-dimensional space-time-domain expansion partitionings are implemented. The time complexities correspondingly roll back a little to $O(m \times n)$ for the one-dimensional partitioning and $O(max(m, n) \times (\lceil \frac{m \times n}{k \times l} \rceil))$ for the two-dimensional partitioning.

2.4 Resource Constraints

In this section, the partition problem is to be discussed according to different computing environments on different implementation platforms, in which the mapping of function or data on resources is mostly concerned. We first formulate the objective functions and introduce some mapping methods for homogeneous systems, then study various mapping approaches for multi-processor, HW/SW co-processing, and VLSI systems in heterogeneous computing environments. In multi-processor system, software components are emphasized. In HW/SW co-processing system, both software and hardware components are co-designed. The VLSI system is for hardware implementation. The software view is modeled to characterize the nature of software processing: asynchronous, data processing per memory buffer basis, and sequential execution. The hardware view is modeled to characterize the nature of hardware processing:

synchronous, per token data processing, and parallel execution [24].

2.4.1 Definitions and Formulations

Suppose an image processing algorithm is expressed by the a function model F, where $F = (C, E)$ has $\|C\|$ components and $\|E\|$ edges.

Definition 1. *A partitioning $P^k = \{V_1, \cdots, V_i, \cdots, V_k\}$ on the operation-level model F is k disjoint clusters, where $V_i \subset C$ ($1 \leq i \leq k$) is a subset of C with size equal to or greater than one component, $V_1 \bigcup V_2 \bigcup \cdots \bigcup V_k = C$ and $V_i \cap V_j = \varnothing$, and k represents the number of participating computing resources.*

The *load weight* of the partition V_i is defined as

$$w_c(V_i) = \sum_{c \in V_i} w_c(c) \tag{2.1}$$

Definition 2. *Given a partitioning $P_{ij} = \{V_i, V_j\}$ between partitions V_i and V_j, the cut of P_{ij} denotes the set of edges that connect components in both partitions, i.e., $cut(P_{ij}) = \{e | e \cap V_i \neq \varnothing and e \cap V_j \neq \varnothing\}$. If edges in F are unweighted, the cut size, expressed as $|cut(P_{ij})|$, denotes the number of edges in $cut(P_{ij})$. For the partitioning P^k, the cut size is*

$$|cut(P^k)| = \sum_{i=1}^{k} \sum_{j=i+1}^{k} |cut(P_{ij})| \tag{2.2}$$

If edges are weighted, the cut weight, expressed as $w_e(P_{ij})$, represents the sum of edge weights in P_{ij}, i.e.,

$$w_e(P_{ij}) = \sum_{e \in P_{ij}} w_e(e) \tag{2.3}$$

The cut weight of P^k is then

$$w_e(P^k) = \sum_{i=1}^{k} \sum_{j=i+1}^{k} w_e(P_{ij}) \tag{2.4}$$

Commonly used objectives include minimizing (1) the load variance on individual partitions, (2) the communication between partitions, and (3) the overall processing cost, etc. In homogeneous environments, the variance of resources can be ignored. The objective of load balance is to find a partitioning $\hat{P^k}$ such that the variance of load weights on individual partitions is minimized. Given k partitions, the objective function is written as

$$\arg\min_{P^k}\{\sum_{i=1}^{k}[w_c(V_i) - E_c(V)]^2\} \tag{2.5}$$

where

$$E_c(V) = \frac{\sum_{j=1}^{k} w_c(V_j)}{k} \tag{2.6}$$

The objective of minimizing communication is to find a partitioning $\hat{P^k}$ such that the cut weight of k partitions is minimized. The objective function is written as

$$\arg\min_{P^k}\{w_e(P^k)\} \tag{2.7}$$

The objective of minimizing overall processing cost is to find a partitioning $\hat{P^k}$ such that the sum of computing and communication costs is minimized. The objective function is written as

$$\arg\min_{P^k}\{Cost_{comp} + Cost_{comm}\} \tag{2.8}$$

where $Cost_{comp}$ and $Cost_{comm}$ are respectively related to the computing and the communication resources such as processing time or power consumption.

2.4.2 Mapping in Homogeneous Environments

In homogeneous environments, all computing and communication resources are similar. We can ignore the variance of resources and directly partition the function model F given only the number of resources. In this case, the number of partitions k may simply be equal to the number of resources. Compared to that in heterogeneous environments, mapping in homogeneous environments focuses more on the function model partitioning which has been studied for decades. The partitioning approaches in homogeneous environments can be briefly classified into three categories: clustering, geometric, and move-based [23]. The clustering method, which is different from the clustering in function modeling, distributes components to individual resources such that each resource has a cluster of components. If an algorithm is modeled as a graph, we can then use the geometric information incorporated in the graph to partition algorithms. The move-based partitioning progressively evaluates the performance by switching components between partitions. We introduce two simple partitioning algorithms, linear [48] and scattered [20], as examples of the clustering-based partitioning; the spectral partitioning [147] as an example of the geometric-based partitioning; and the multi-level Kernighan-Lin (K-L) [87] partitioning as an example of the move-based partitioning. These three types of partitioning are global partitioning that are operated on the entire function model. The partitioning results can be improved by the local refinement. We use the K-L algorithm, which can be used as both global partitioning and local refinement, as an example and apply it to the partitioning results obtained by the first three approaches.

Global Partitioning Algorithms

Linear Scheme. Given n components in a random sequence and k resources. The linear method [48] assigns the first $\lceil \frac{n}{k} \rceil$ components to the first resource, and the next $\lceil \frac{n}{k} \rceil$ components to the second one, etc. This method may generate good performance in load balance and communication as the sequence of components more or less reflects the locality of components.

Scattered Scheme. Given n components in a random sequence and k resources. The scattered method [20] sequencely maps one component to each partition, then the $(k+1)^{th}$ component to the partition with the smallest load weight. This assignment procedure iterates until all components are mapped to partitions.

Spectral Scheme. The spectral scheme [147] uses geometric information to conduct bipartitioning as we previously defined. The adjacency matrix $A = (a_{ij})$ and the degree $deg(c_i)$ $(1 \leq i \leq n)$ are first derived from the function model F. The $n \times n$ *degree matrix* D is given by $d_{ii} = deg(c_i)$ with $d_{ij} = 0$ if $i \neq j$. Then the Laplacian matrix of F is defined as $Q = D - A$. We find the normalized n-dimensional eigenvectors $\mu_1, \mu_2, \cdots, \mu_n$ for Q with corresponding eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. It has been shown that the second eigenvector μ_2 gives the optimal partitioning result and the elements of μ_2 are approximated to 0 or 1, indicating the assignments to partitions.

Multi-level K-L Scheme. The K-L algorithm [87] can be used as both global partitioning and local refinement. However, the quality of the global partitioning generated by the K-L algorithm is not good enough. So the multi-level K-L algorithm [84] is developed as a global partitioning approach. The multi-level K-L algorithm generates smaller function model by clustering several neighbor components into one module, and creates a sequence of increasingly smaller models to approximate the original model. The K-L partitioning is conducted on the smallest function model. The result is then projected back to the upper level (larger model) and the K-L partitioning is conducted again. This process iterates until back to the original model.

Local Refinement

The local refinement improves the global partitioning results by evaluating every pair of partitions and conducting necessary reassignments. We introduce the K-L algorithm [87] as an example of the local refinement. This algorithm uses a pair-swap neighborhood structure and conducts a series of exchange between the pair of partitions. At the beginning, an initial solution

$\{V_1, V_2\}$ is given and each component within the two partitions is unlocked, meaning that it is free to swap any two components belonging to different partitions if needed. Suppose $c_i \in V_1$ and $c_j \in V_2$ are two unlocked components. If c_i and c_j have the highest gain in all pairs of unlocked components, where the gain is defined as $gain = w_c(\{V_1 + c_j - c_i, V_2 + c_i - c_j\}) - w_c(\{V_1, V_2\})$, they are swapped between the two partitions and the result serves as the initial solution to the next iteration. After a component is swapped, it becomes locked. The K-L local refinement iteratively swaps the pair of unlocked components with the highest gain until a swap does not improve upon the previous solution.

Based on the three categories of global partitioning and local refinement, numerous partitioning approaches have been proposed for the homogeneous environments. Most VSNs, however, form heterogeneous environments due to remaining energy on individual micro-sensors and distances between micro-sensors. Therefore, we will focus on the developments of resource-oriented mapping approaches in heterogeneous environment in the next chapter.

2.4.3 Multi-processor System

In multi-processor environments, approaches for partition problems deal with how to assign processes of an algorithm to individual processors in order to minimize the cost of executing this algorithm. The cost consists of (a) computing time of the processes on the processors to which they are assigned; and (b) communication time between processes if they are assigned to different processors. The former relates to the computational complexity of the algorithms, and the latter relates to the communication overhead that is associated with data transfer between processes.

When we examine a multi-processor system, the computing capability is always the first concern. In order to efficiently manage the available computing resources, most multi-processor systems, especially for parallel computing as we discussed in Sec 2.2.2, assign a powerful processor as master or host that schedules the entire task. The processing speed is determined

by the processor that takes the longest time to execute the assigned processes, which is also known as the *bottleneck processor* [21].

Compared to the computation concern, communication delay plays a more important role in mapping performance. The communication between processors is carried out through the message-passing mechanism, which requires extra overhead to manage messages during the data movement between processors. Therefore, partition approaches such as that demonstrated in Fig. 2.9(b) reduces the amount of overhead. Since an overhead in the message transformation is fixed regardless of the length of the message, more data output from one processor can be packed into one message. However, some communication work, such as allocating buffers for messages and setting up DMA channels, has to be performed by the CPU and cannot be overlapped with the computation.

Reference [90] proved that an optimal execution can be obtained when the communication (message startup delay) and computation (single-iteration execution time) are balanced.

From the practical point of view, a multi-processor system can be set up in either a homogeneous or a heterogeneous computing environment. A fundamental problem in both environments is the difficulty of optimally partitioning a task across computing resources, but the structure of the heterogeneous computing system is more complicated and thorough.

Heterogeneous computing (HC) is a computing paradigm that takes the advantages of the high-speed network to exploit available computing resources in order to provide feasible solutions. These resources include high performance computers in different types of parallelism [88]. However, the overheads involved in communication among the resources add extra delays. If not carefully analyzed, such scenario can possibly degrade the overall performance, instead of improving as expected.

For the heterogeneous multi-processor system, Banerjee *et al.* [14] presented a macro pipelining-based scheduling technique in which the partition problem is described as two steps, one is to determine the pipeline stages for the processes, and the other is to schedule each

pipeline stage in more detail with respect to the processors. At the first step, the Ratio Cut (RC) partitioning is used to find a global coarse solution.

The RC partitioning is a two-way partitioning procedure. Suppose A is the process set, $A1$ and $A2$ are two subsets, $|A1|$ and $|A2|$ are the size measure of the two partitions, and W is the weight sum of the edges involved between $A1$ and $A2$. The RC partitioning generates the partitioned blocks such that the ratio metric defined by

$$C = \frac{W}{|A1||A2|} \tag{2.9}$$

is minimized. The meanings of W, $|A1|$ and $|A2|$ vary according to different applications. For DSP processes, W refers to the total signal crossing bandwidth, and $|A1|$ and $|A2|$ refer to the computational complexities of the respective blocks. Given a process set A, the RC partitioning first evaluates W and initially partitions A into two subsets. Then another two-way partitioning is performed on the larger subset. In each of the succeeding iterations, the largest subset in A is further divided into two, and the partitioning continues until the size of the largest subset is acceptably small or the number of subsets is large enough.

Next, the processors in the heterogeneous environment are involved. Let τ_{ij} be the execution time of process i on processor j, $t(i,j)$ be the execution time of the subset T_i on processor j, where $t(i,j) = \sum_{k \in T_i} \tau_{kj}$. The precedence relationship between T_i and T_j in the partial order by $T_i > T_j$ indicates T_j depending on T_i, and $T_i \geq T_j$ indicates that either T_i precedes T_j or there is no ordering between T_i and T_j. Given the lists of the process subsets *TLIST* and the available processors *PeList*, the pipelined scheduling algorithm, which is based on the implicit enumeration of linear extensions of the partially ordered subsets, returns the best assignment. Each pipeline stage is then scheduled by any heuristic scheduling algorithm such as the Branch & Bound technique that reduces the solution space by rejecting partial solutions whose cost is greater than the minimum cost solution found at a given stage. Although the scheduling problem in general is NP-complete as we previously stated, the Branch & Bound can be executed as an

optimization procedure when the number of processes is small.

At the second step, the macro pipelining iteratively identifies the bottleneck pipeline stage and reduces its execution time by applying the architecture driven partitioning, the repartitioning, and the time axis relabeling techniques. The architecture driven partitioning reduces the execution time of the bottleneck stage by either increasing the parallelism through partitioning a task, or decreasing the interprocessor communication through merging tasks. If the architecture driven partitioning fails to reduce the execution time of the bottleneck stage, then the repartitioning and the time axis relabeling are applied to reduce it. The repartitioning minimizes the size of the bottleneck processes by repartitioning processes into a new stage and two other consecutive stages. If the execution time of the bottleneck stage is actually reduced by the rescheduling, the execution time of the entire schedule is then reduced by relabeling the time axis at the pipeline stages. In this case, a new bottleneck stage is identified after recalculating the execution time, followed by relabeling each stage.

Iqbal *et al.* [78] also intended to approximately solve the partitioning problem in parallel and pipelined programs over heterogeneous system by using a fully polynomial time approximation scheme. This scheme takes into account the heterogeneous nature of each subset in processing as well as the communication overheads of subsets residing on different types of processors.

As an extension of the heterogeneous computing system, the Grid computing environment has been becoming popular during the last decade. The Grid environment provides scalable processing power and storage space on demand by flexibly integrating resources distributed at different sites. Akiyama *et al.* [4] presented a dynamic data partitioning technique for real time pipelined image processing in the Grid computing environment. Since the resources involved in the pipeline may change for each request, it is necessary to detect the bottleneck stage dynamically, and then decide the data partitions. This method plots a line of latency with respect to the data partition size for each stage. By evaluating the intersections and iteratively updating the inspected region, the partition points is found such that the inspected region does not include

intersections.

As we observe in this section, various partitioning methods have been applied to the multi-processor systems and new ones are emerging. In recent years, progressive partitioning methods such as the multilevel partitioning have been very popular and applied to multi-processor and VLSI systems [117]. In general, multilevel partitioning first roughly divides the entire task by efficient algorithms so as to reduce the size of the inspecting region. The coarsened solution or graph is then repartitioned and refined by another partitioning approach. In addition, the initial cut edges dividing the entire task can be adjusted in order to obtain global optimal partition. We will discuss the multilevel partitioning in detail in Sec 2.4.5.

2.4.4 Hardware/software (HW/SW) Co-processing

The HW/SW co-processing system generally contains a single general-purpose processor like Pentium or PowerPC, a single hardware coprocessor like FPGA or ASIC, and a block of shared memory. From the design point of view, the HW/SW co-processing system consists of the hardware components and the software components. The hardware components refer to various RTL components such as adders, multipliers, ALUs, and registers; the software components refer to the general-purpose processors [36]. In HW/SW co-processing, only the communication time between the software component and the local memory is taken into account.

For different applications, HW/SW co-processing systems vary significantly in numbers and types of function modules and components. But two concerns are emphasized in common, the system modeling and the objective/cost function formulation. Various graphs have been used and developed in system/task modeling. Some good examples include directed acyclic graph (DAG) [35], control flow graph (CFG) [13], control data flow graph (CDFG) [129], variable independence graph (VIG) [146], etc. Another purpose of using graph in modeling is to directly apply well-developed solutions for the graph partitioning problem.

A DAG, as shown in Fig. 2.12, consists of three types of vertices, the primary inputs (PI),

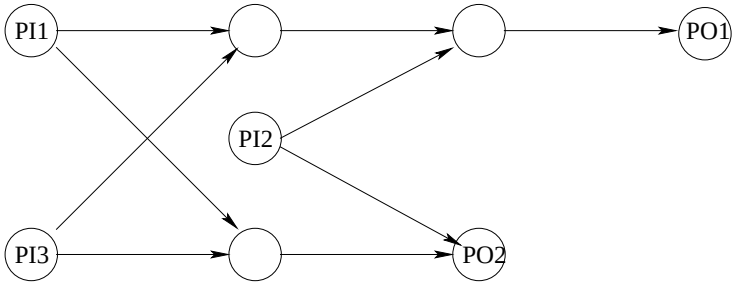


Figure 2.12: Directed acyclic graph (DAG).

the primary outputs (PO), and the internal nodes of the graph. PI and PO are nodes that have only outgoing edges and incoming edges, respectively. DAG can be used to model various systems. When we use a DAG to represent a combinational circuit, an internal node that has two incoming edges and one outgoing edge is associated with a Boolean function. A system of Boolean functions with specified variables as PI and functions as PO generates a Boolean network. Since the functionality of a Boolean network is specified by its primary output function set, two Boolean networks are equivalent if they have the same PO.

In CFG, the nodes represent the behaviors and the arcs represent the control dependencies. The input to a CFG can include behaviors, a hardware and software library, and clock constraint. Arcs in CFGs specify the control flow. Take a CFG with 6 nodes as an example, which is illustrated in Fig. 2.13, the behaviors 3 and 4 cannot start until the behavior 2 has been completed. Similarly, the behavior 5 can only start if both the behaviors 3 and 4 have been completed. Each behavior, for example, may contain a sequence of Very High Speed Integrated Circuit Hardware Description Language (VHDL) statements that represents computation done on variables.

As the other focus of the partitioning in HW/SW co-processing systems, the objective function is critical in order to satisfy the performance requirements for a sequential implementations. For the purpose of reducing the required time in partitioning, system designers sometimes choose some simple and well-known techniques such as 90-10 rule in real applications [129]. Since the most frequent few loops generally correspond to 90 percent of execution time

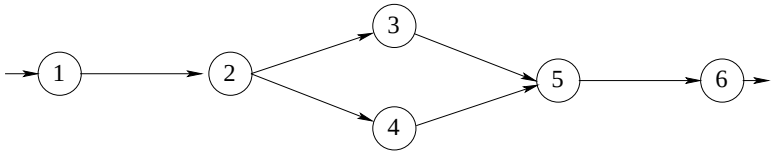


Figure 2.13: Control flow graph (CFG).

but only consist of a few lines of codes, it is better to assign these loops to the hardware partition for speedup purposes. For those regions of code accessing the same memory locations we should include them in the hardware partition as well as embedding the memory within the FPGA until the area constraint is violated.

Researchers also develop efficient but a little complicated objective functions in partitioning. Chatha and Vemuri [35] considered the timing and the area constraints in pipelined HW/SW implementation, and presented a synthesis tool. The objective of this tool is to minimize the time difference between the start of two consecutive iterations, defined as the initiation interval (II). The presented partitioner searches for the minimum initiation interval (MII) of the final design.

Castellano *et al.* [31] developed a DAG-based automatic partitioning tool, GACSYS, for pipelined HW/SW computing. This process-level partitioning tool especially considers the power consumption, together with the design area and the execution time, in the cost functions. The cost functions of power and area are respectively

$$COST_P = \alpha \frac{P - P_{min}}{Range(P)} + (1 - \alpha) \frac{T}{Range(T)} \tag{2.10}$$

and

$$COST_A = \alpha \frac{A - A_{min}}{Range(A)} + (1 - \alpha) \frac{T}{Range(T)} \tag{2.11}$$

where P, A, P_{min}, A_{min} represent the current and the minimum power consumption and design area, respectively. The weight parameter α, $0 \leq \alpha \leq 1$, is used to give priority to either power consumption or design area. The redundancy time T is expressed by the difference between the

maximum execution time T_E and the pipeline stage execution time T_{Fi} in each phase.

$$T = \sum_{i}^{Number of Phases} (T_E - T_{Fi}) \tag{2.12}$$

$$T_{Fi} = \sum_{step} max[max(T_{HW}), \sum_{process} T_{SW}] \tag{2.13}$$

where T_{HW} and T_{SW} denote the execution time of hardware and software processes. After specifying (a) the hardware and the software parameters including the system execution time, the design area, and the power consumption, and (b) resource constraints, the GACSYS generates the cost functions in the format of Eq.2.10 or 2.11. The optimization of the cost functions is then conducted based on the simulated annealing algorithm until a partitioning in the process level is finally found.

2.4.5 VLSI

For VLSI systems, the most important motivation for conducting partitioning is the large size of circuit designs. According to the general VLSI synthesis procedure, the partitioning approaches are mostly discussed in either the CAD simulation domain or the circuit design domain. In the CAD simulation domain, researchers are interested in how to partition one large design into multiple small segments and speed up the entire simulation procedure. In the circuit design domain, the partition problem is considered in the process scheduling that refers to as deciding the sequence or the order of tasks in a pipeline or parallel structure.

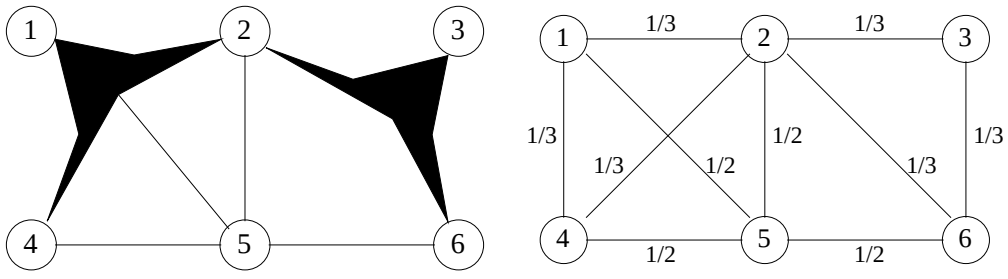
Before discussing the specific solutions for individual domains, let us first study the common methods that can be applied to both domains. In the VLSI application field, most approaches to partition problems also utilize various graph algorithms in order to directly apply and extend the off-the-shelf solutions. Milestone works of partitioning in VLSI applications have evolved from the bipartitioning, the multilevel bipartitioning, the k-way partitioning, to the multilevel k-way partitioning. In order to introduce these approaches, we first demonstrate

the popularly used *hypergraph* as the analysis tool.

The hypergraph has been used as a conventional tool in circuit modeling without altering the circuit design. The hypergraph is defined as $H = (V, E)$, where the vertices V represent the modules and E represent the hyperedges or nets (edges). As an example shown in Fig. 2.14(a), the solid black triangles denote the interconnection of a group of nodes. For the purpose of modeling, the hypergraph is sometimes transformed into the hyperedge-weighted hypergraph as shown in Fig. 2.14(b). The weight $1/m$ of each edge is given by the number of nodes the edge connects.

Suppose a hypergraph $H = (V, E)$ has n modules $V = (v_1, v_2, \cdots, v_n)$; a net $e \in E$ is defined to be a subset of V with size greater than 1. A *bipartitioning* $P = \{X, Y\}$ is a pair of disjoint clusters $X \bigcap Y = \emptyset$, i.e., subsets of V, and $X \bigcup Y = V$. The *cut* of a bipartitioning $P = \{X, Y\}$ is the number of nets which contains modules in both X and Y, i.e., $cut(P) = |\{e | e \bigcap X \neq \emptyset, e \bigcap Y \neq \emptyset\}|$. These nets or edges are called *cut edge*. Let $A(v)$ denote the area of $v \in V$ and $A(S) = \sum_{v \in S} A(v)$ denote the area of a subset $S \subseteq V$. Given a balance tolerance r, the *min-cut* bipartitioning problem seeks a solution $P = \{X, Y\}$ that minimizes $cut(P)$ subject to $\frac{A(V)(1-r)}{2} \leq A(X), A(Y) \leq \frac{A(V)(1+r)}{2}$ [6], i.e., the bipartitioning with minimum cut.

The bipartitioning approach was the iterative improvement based on the Kernighan-Lin



(a) Hypergraph.

(b) Hyperedge-weighted hypergraph.

Figure 2.14: Hypergraph in circuit partitioning.

(KL) heuristic algorithm [87], and later improved by Fiduccia and Mattheyses (FM) [61]. The FM algorithm proceeds a series of passes. A pass begins with some initial solution $\{X, Y\}$; modules are successively moved between X and Y until each module has been moved exactly once. Given a current solution $\{X', Y'\}$, the previously unmoved module $v \in X'$ (or Y') with the highest gain ($= cut(X' - v, Y' + v) - cut(X, Y)$) is moved from X' to Y'. After each pass, the best solution $\{X', Y'\}$ observed during the pass becomes the initial solution for a new pass. The iteration terminates when a pass does not improve upon the previous solution.

Many improvements have been developed based on KL and FM algorithms. For example, Wakabayashi *et al.* [132] presented an algorithm by combining the KL heuristic with the odd-even transposition sorting algorithm. The proposed algorithm runs on a linear array of $\frac{n}{2}$ processing units, and produces a partition in $O(n)$ computation time. In the FM algorithm, one potential problem is that many modules in the top bucket may potentially have the same gain. So Albert *et al.* [6] utilized a last-in first-out (LIFO) bucket scheme for storing module gains.

With the fast development of VLSI technologies, the design size, or equivalently, the problem size, has been increasing dramatically. One technique typically used to handle increasing problem size is clustering or coarsening. In a circuit, the modules are grouped into many small clusters, which form new nodes in a smaller and coarser graph. Iterative improvements are then performed on the clustered graphs. From practical point of view, iterative approaches possess many advantages. They are generally intuitive, easy to describe and implement, and relatively fast [95].

Based on the iterative approaches, the concept of multilevel partitioning is generated. The multilevel partitioning generally includes three steps [137]: (1) Cluster or coarsen the problem with efficient algorithms so as to reduce the size. (2) Apply a graph-domain partitioner on the graph and obtain a high quality initial solution. (3) Uncluster or uncoarsen the graph and apply a partitioning refinement algorithm in order to adjust the cut edge between partitions. Here, we mainly analyze the multilevel bipartitioning, since the multilevel k-way partitioning is a k-way

extension of the multilevel bipartitioning. The formal definitions of the multilevel bipartitioning are given as follows [6].

Definition 3. *A partitioning $P^k = \{C_1, C_2, \cdots, C_k\}$ of H_i induces the coarser graph $H_{i+1}(V_{i+1}, E_{i+1})$ with $V_{i+1} = \{C_1, C_2, \cdots, C_k\}$. For every $e \in E_i$, the net e^* is a member of E_{i+1}, where $e^* = \{C_h | e \bigcap C_h \neq \emptyset\}$, unless $|e^*| = 1$, i.e., e^* spans the set of clusters containing modules of e.*

In this definition, P^k represents a partitioning, H_i denotes the i^{th} level in the hierarchy, V_i and C_i respectively denote the set of modules and a single module, E_i and e respectively denote the set of nets and a single net.

Definition 4. *Suppose that H_{i+1} was induced from H_i by the partitioning $P^k = \{C_1, C_2, \cdots, C_k\}$. The projection of the bipartitioning solution $P_{i+1} = \{X_{i+1}, Y_{i+1}\}$ of H_{i+1} onto H_i is the solution $P_i = \{X_i, Y_i\}$ where $X_i = \{v \in V_i | \exists C_h \in P^k, v \in C_h, C_h \in X_{i+1}\}$ and $Y_i = \{v \in V_i | \exists C_h \in P^k, v \in C_h, C_h \in Y_{i+1}\}$. The process of projecting P_{i+1} to P_i is called uncoarsening.*

In the above definition, P_i represents the partitioning at the i^{th} level in the hierarchy, X_i and Y_i represent the two partitions at the i^{th} level.

Figure 2.15 illustrates the multilevel bipartitioning with five levels [83]. In a multilevel algorithm, a clustering of H_0 is used to induce the coarser graph H_1, then a clustering of H_1 induces H_2, etc., until the most coarsened graph H_m ($m = 4$ in the figure) is constructed. A bipartitioning solution $P_m = \{X_m, Y_m\}$ is found for H_m by using algorithms such as FM. This solution is then projected to $P_{m-1} = \{X_{m-1}, Y_{m-1}\}$. P_{m-1} is then refined by FM post-processing. In Fig. 2.15, the projected and refined solutions are respectively denoted by dotted and solid lines. The uncoarsening process continues until a refined partitioning of H_0 is obtained.

Next, we move the discussion from the bipartitioning to the k-way partitioning, and then the multilevel k-way partitioning. The k-way partitioning extends the bipartitioning from one direc-

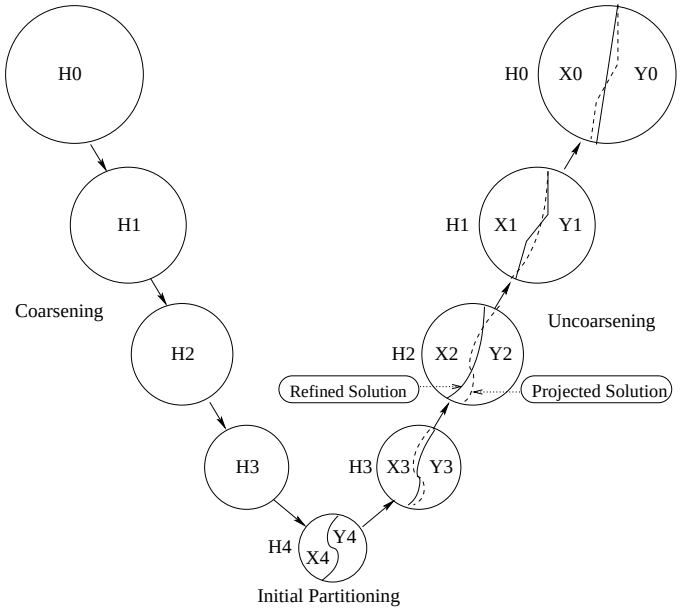


Figure 2.15: Multilevel bipartitioning [83].

tion to k directions. Given a hypergraph $H = (V, E)$ with weighted-vertices V and weighted-hyperedge E, a k-way partitioning of V assigns the vertices to k disjoint nonempty partitions [29]. The objective of the k-way partitioning is to minimize a given objective function $c(P^k)$, whose arguments are partitioning. A standard objective is the *cut size*, i.e., the number of hyperedges whose vertices are not all in a single partition. Other potential objectives include the ratio-cut, the scaled cost, the absorption cut, etc. In the k-way partitioning, some constraints are typically imposed on the solutions. For example, the total vertex weight in each partition may be limited (balance constraints), which results in an NP-hard formulation [29].

In the multilevel partitioning, the multilevel k-way partitioning is an extension of the multilevel bipartitioning. Karypis and Kumar [84, 83] developed an hMetis-Kway partitioner that obtains encouraging results in both quality and runtime. This partitioner first coarsens the hypergraph, followed by recursive bisections on the graph that generates k parts, and finally uncoarsens the hypergraph with refinement algorithms. Then, Alpert *et al.* [6] presented the ML multilevel algorithm based on the matching algorithm [83] that can control the speed of coars-

ening, hence the number of levels in the hierarchy. By allowing coarsening to proceed more slowly, this algorithm obtains more levels in the hierarchy and provides more opportunities to refine the current solution at the various levels. In addition, they adopted the CLIP algorithm [59] to the FM implementation. Since the CLIP algorithm breaks ties based on the adjacency of the most recently moved modules, the adjacent vertices in the ML multilevel algorithm are moved sequentially.

In the rest of this section, we discuss some solutions specifically to the CAD simulation domain and the circuit design domain. Most approaches for the CAD simulations inherit the partitioning objectives from the multi-processor system, that is, minimizing communication and keeping coarse granularity in partition. Frohlich *et al.* [63] developed the analytical partitioning method (APM) to improve the circuit simulator TITAN developed by Siemens. They applied the analytical placement method with the ratio cut objectives to circuit designs at a transistor level. The partitioning procedure first identifies the critical modules and relating nets. Then the circuit is modeled in a weighted hypergraph. The APM calculates the sum of such edge weights between modules and tries to minimize the cut weight. The cut procedure is fulfilled by a k-way RC partitioning, i.e., dividing a circuit into k partitions using ratio cut. The final partitioning results for the parallel simulation are subcircuits with a small number of interconnected nodes and with well-balanced sizes.

If we combine the hypergraph with other information, the partitioning results could be further improved. Wu *et al.* [137] considered the detailed logic functions in circuit rewiring and presented a multi-way partitioning method to improve the results obtained by the hypergraph partitioning. The idea of their method is to study the logic functions near the cut edges and replace the circuits with large cut size by equivalent circuits with less cut size. The proposed partitioner uses the graph-domain partitioner hMetis-Kway to obtain the initial partitions, and then iteratively couples the logic information into the optimization process.

In the circuit design domain, the standard formulation seeks a multi-way partitioning of a

vertex- and hyperedge-weighted directed hypergraph as we previously discussed [81]. However, many partitioning approaches originate from the practical point of view instead of the general system models. These approaches still focus on min-cut bipartitioning. For example, the FM algorithm is a widely used one by the physical design community due to its short run-times and ease of implementation.

On the other hand, the objective function is also formulated from practical points of views. With respect to the partition size constraints, the objective function can include the minimum hyperedge-cut, I/O-count or path-cut constraints, replication and retiming degrees of freedom, hierarchy awareness, etc. Correspondingly, common DSP tasks implemented on DSP ASIC are performed under the constraint of a specified execution time. When the time constraint is satisfied, the objective that maximizes the throughput rate of the system is then considered. For example, Jung and Lee [80] proposed a memory partitioning method that divides data into 4 modules, in which elements from individual modules alternate to each other both by row and by column. The purpose is to avoid the memory access conflict when implementing a 4-way pipelined processing architecture. This method was eventually applied to the three-step search block-matching algorithm (TSS BMA), one of the best algorithms for motion estimation in video coding standards such as H.261 and MPEG. Plosila *et al.* in [115] presented a pipelined on-chip bus architecture for globally asynchronous locally synchronous (GALS) system-on-chip (SoC) design. The GALS architecture divides a system into multiple clock domains, in each of which the attached modules compose a cluster and operate at the same speed internally. Modules in the same cluster are connected by a local bus or local point-to-point links. Individual clusters independently connect to the pipelined bus. Interconnected clusters operating at different speeds are linked to each other through either a self-timed interface based on asynchronous handshake signaling, or a separate clock domain with synchronization. The global bus is partitioned into asynchronously interacting segments such that clusters in different clock domains can access the bus simultaneously.

2.5 Summary

In this chapter, we reviewed some existing approaches to partitioning, clustering, and mapping in pipelined and parallel computing. Based on the application-specific design trend of VSNs, we categorized different approaches from three application aspects, i.e., the dependency structure, the driving force, and the resource constraints. From the review, we see that most of these approaches only focus on one or a subset of the application performances. In visual sensor networks, we need to consider all three application aspects in the partitioning procedure in order to make the best usage of the constrained resources. In next chapter, we will propose our modeling, clustering, and mapping approaches that reflect this principle.

Chapter 3

Modeling and Partitioning of Algorithms and Data for Image Processing

Nowadays, in order to obtain better performance, many image processing applications encounter both complicated arithmetic functions and large data sets that result in time-consuming calculations. As we reviewed in Chapter 2, a promising solution is to use pipelined and parallel structures, therefore distributing computing burdens from a single computing resource to multiple computing resources. But how to partition functions and input data sets so as to efficiently map them to the limited computing resources remains a big challenge.

In this chapter, we propose several modeling, partitioning, and mapping approaches for fast image processing. In order to analyze functions in image processing algorithms, we propose a hierarchical function model, referred to as the *multi-weight operation-level model*, and use it in the modeling procedure. In this model, a function is first decomposed into basic arithmetic operations, which are then connected according to the processing flow and form the pipelined and the parallel processing structures. This model includes a component clustering algorithm

that provides components in appropriate granularity to the mapping procedure. Meanwhile, the cyclic process modeling is discussed in the hierarchical function model. In this chapter, we also analyze the data dependencies in the input images, and categorize them into the independency, uniform, and regional dependencies. The data dependency instructs the data partitioning and distribution at the mapping stage. Finally, we present two resource-oriented mapping algorithms, the load attraction and the communication attraction, for function mapping in heterogeneous environments. The objectives of these two algorithms are to balance load and minimize communication, respectively. Both algorithms are combined with local refinement in order to satisfy the objective of minimizing the overall processing cost.

3.1 Multi-weight Operation-level Function Model

3.1.1 Model Setup

Given an image processing algorithm, we first use basic components to model functions of the algorithm.

Definition 5. *Basic components lc, nc, and $logc$ perform basic linear, nonlinear, and logic operations, respectively, if and only if the operations are directly implemented by the target synthesis technologies.*

The linear basic components consist of linear arithmetic operations such as addition and multiplication that are available in the instruction set or on hardware circuit. The nonlinear basic components include logarithm (LOG), exponential (EXP), etc., which are implemented by using respective lookup tables (LUTs). The logic basic components include comparator (CMP), buffer (BUF), etc., which are basic logic operations used to control the processing flow. If one component c is not directly implemented by the target synthesis technologies or it is necessary to decompose c into other components, then c is not treated as a basic component.

We initialize an expandable basic component list in Table 3.1. In the following illustrations,

Table 3.1: Basic component list.

Linear component		Nonlinear component		Logic component	
Operation	Component label	Operation	Component label	Operation	Component label
Addition	lc_1	Logarithm (LOG)	nc_1	Comparator (CMP)	$logc_1$
Multiplication	lc_2	Exponential (EXP)	nc_2	Buffer (BUF)	$logc_2$
		Sine (SIN)	nc_3		
		Cosine (COS)	nc_4		

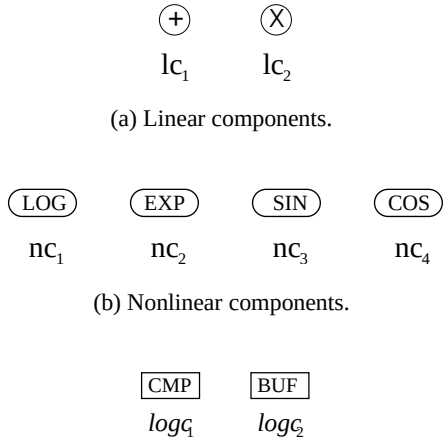
these basic components are expressed as nodes in circle, ellipse, and rectangle shapes as shown in Fig. 3.1. A set of these basic components forms a component set C. Hereinafter, we use c to denote a generic basic component unless otherwise specified.

Definition 6. *An edge e_{ij} represents a uni-directional connection from component c_i to component c_j, and that c_j is temporally executed after c_i.*

Note that if no edge exists between two components, they can be temporally executed in parallel.

Using components and edges, we can decompose an image processing algorithm into basic operations, and express the algorithm with the operation-level function model F, where $F = (C, E)$ has $\|C\|$ components and $\|E\|$ edges. Here, we use $\|.\|$ to denote the number of elements. Based on the operation-level function model, pipelined and parallel processing can be generated according to the existence of edges. That is, by clustering the components connected by edges into a module, we are able to model various sequential functions.

Definition 7. *A module $M = (C_M, E_M)$ is a group of $\|C_M\|$ components connected by $\|E_M\|$ edges, where $M \subset F$, $C_M \subset C$ and $E_M \subset E$. Suppose a module M contains x number of basic components c_i and y number of basic components c_j, the module is then expressed as $c_i^x c_j^y$.*



(a) Linear components.

(b) Nonlinear components.

(c) Logic components.

Figure 3.1: Basic operation components.

For example, we use one multiplier (lc_2) and one adder (lc_1) to model a multiplier & accumulator (MAC) module as shown in Fig. 3.2, and express the module as $lc_1 lc_2$. The first and the last components in the module are respectively marked on the left and the right edges of the module.

Then, modules like MAC can be used as element to generate more complicated modules. For example, by using the MAC module we can compose a 4-stage pipelined process, as illustrated in Fig. 3.3. We observe that the pipelined process has been grouped into a 4-stage MAC module with the expression of $lc_1^4 lc_2^4$.

By repeating the clustering process, we can model a function from the basic operation level to higher levels. The modeling procedure is therefore in a hierarchical structure with several levels. A 3-level structure example is shown in Fig. 3.4.

Using the hierarchical modeling structure, the software designer can implement the algorithm by examining only the upper module level without considering the detailed operations. The hardware designer needs to investigate the lower level but may understand the complexity of the algorithm from the upper level.

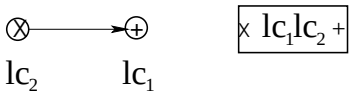


Figure 3.2: A multiplier & accumulator (MAC) module.

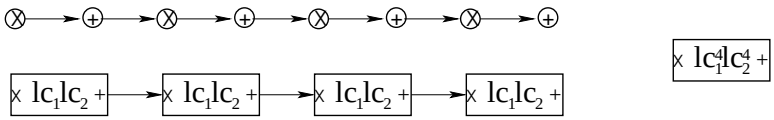


Figure 3.3: A 4-stage MAC module.

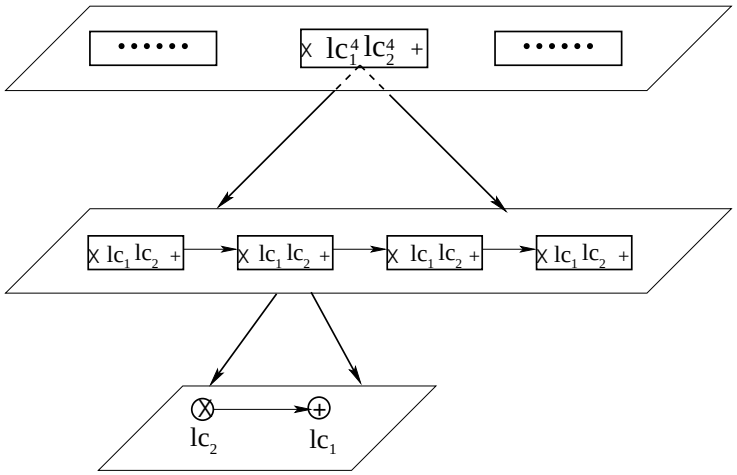


Figure 3.4: The hierarchical modeling structure.

In addition, the off-the-shelf intelligent properties (IPs) available at a specific level can be rapidly applied to the algorithm implementations.

With regard to complicated image processing algorithms, multiple computing resources are commonly used to speedup the overall process. In order to efficiently distribute processes among participating computing resources, function partitioning is required to guarantee that an expected overall speedup can be achieved. To formulate the objective function in terms of function partitioning, several criteria are taken from both the features of the implemented function and those of the computing resources. From this point of view, it is necessary to assign weights that represent the concerned performance parameter such as delay and power consumption to components.

Definition 8. *The weight $w_c(c) = (w_c^1(c), \cdots, w_c^m(c))$ denotes the performance parameter(s) of the component c when it is implemented using a target synthesis technology. m is the number of performance parameters.*

If only the most important performance parameter is used, i.e., a component is assigned with one weight, the model is called a *single-weight operation-level function model*.

In specific applications, different types of parameters may affect the overall performance and are required to be considered simultaneously. In this case, it is necessary to assign multiple weights to a component. If two types of performance parameters are used in the function modeling, then a component c is assigned with two weights, which is expressed as $(w_c^1(c), w_c^2(c))$. The model is called the *dual-weight operation-level function model*. In addition, parameter w_c^1 is supposed to play a more important role in affecting the performance than parameter w_c^2 does.

Similarly, if m ($m > 2$) types of performance parameters are considered at the same time in the function modeling, then a component c is assigned with multiple weights that are expressed as $(w_c^1(c), \cdots, w_c^m(c))$. The model is referred to as the *multi-weight operation-level function model*. The multi-weight operation-level model is very practical in VLSI implementation, where design performance is normally measured by power consumption, delay, and utilization

area (PDA).

When we group the weighted components into a module, the corresponding weight is then assigned to the module.

Definition 9. *The weight $w_c(M)$ of a module M is the weight sum of the composing components. In a single-weight operation model,*

$$w_c(M) = \sum_{i=1}^{\|C_M\|} w_c(c_i) \tag{3.1}$$

where $\|C_M\|$ is the number of composing components, c_i; in a dual-weight operation model,

$$w_c(M) = (w_c^1(M), w_c^2(M)) = (\sum_{i=1}^{\|C_M\|} w_c^1(c_i), \sum_{i=1}^{\|C_M\|} w_c^2(c_i)) \tag{3.2}$$

and in a multi-weight operation model,

$$w_c(M) = (w_c^1(M), \cdots, w_c^m(M)) = (\sum_{i=1}^{\|C_M\|} w_c^1(c_i), \cdots, \sum_{i=1}^{\|C_M\|} w_c^m(c_i)) \tag{3.3}$$

where m is the number of different types of performance parameters, and $m > 2$.

Since the execution time is critical in real-time image processing and the utilization area is mostly concerned in hardware implementation, we assume the delay is the most important performance parameter in the following discussion, then the utilization area, and finally the power consumption. The weight assignment is only demonstrated on the single-weight and the dual-weight operation models. The assignment procedure on the multi-weight operation model is similar. Let us take the single-weight MAC module as an example, as shown in Fig. 3.5. Because the weight (execution time) of the multiplier lc_2 and the adder lc_1 are $w_c(lc_2)$ and $w_c(lc_1)$, the weight of the MAC module is $w_c(lc_1lc_2) = w_c(lc_1) + w_c(lc_2)$.

The dual-weight MAC module is shown in Fig. 3.6.

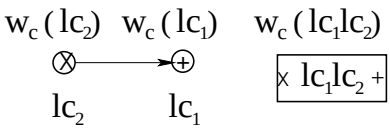


Figure 3.5: Single-weight MAC module.

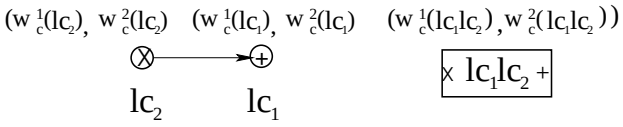


Figure 3.6: Dual-weight MAC module.

The dual-weight (execution time, utilization area) of the multiplier lc_2 and the adder lc_1 are $(w_c^1(lc_2), w_c^2(lc_2)$ and $(w_c^1(lc_1), w_c^2(lc_1)$, respectively. So the dual-weight of the MAC module is $(w_c^1(lc_1lc_2), w_c^2(lc_1lc_2)) = (w_c^1(lc_1) + w_c^1(lc_2), w_c^2(lc_1) + w_c^2(lc_2))$.

In the hierarchical modeling structure, the weight assignment procedure starts from the operation level to the module level, and then the higher module level. The single-weight and the dual-weight 4-stage MAC modules are respectively demonstrated in Figs. 3.7 and 3.8, where $w_c(lc_1^4lc_2^4) = 4w_c(lc_1lc_2) = 4w_c(lc_1) + 4w_c(lc_2)$ and $(w_c^1(lc_1^4lc_2^4), w_c^2(lc_1^4lc_2^4)) = (4w_c^1(lc_1lc_2), 4w_c^2(lc_1lc_2)) = (4w_c^1(lc_1) + 4w_c^1(lc_2), 4w_c^2(lc_1) + 4w_c^2(lc_2))$.

In the previous discussion of the function modeling, we have emphasized the features of the basic components but ignored the data communication between components. If the amount of data transferred between components is significant, then data communication should be incorporated as well in the function modeling. In order to consider both the component and the data communication, we define the weight of edge as follows.

Definition 10. *The weight $w_e(e_{ij})$ of an edge e_{ij} is the amount of data transfered from the output of the component c_i (the start point of the edge) to the input of the component c_j (the end point of the edge).*

Figure 3.9 shows the single-weight MAC module with edge weight as an example, where the multiplier (lc_2) is component c_1 and the adder (lc_1) is component c_2.

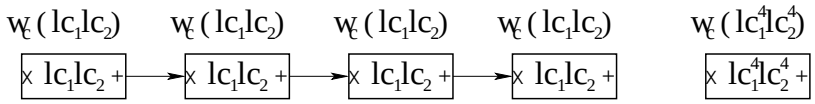


Figure 3.7: Single-weight 4-stage MAC module.

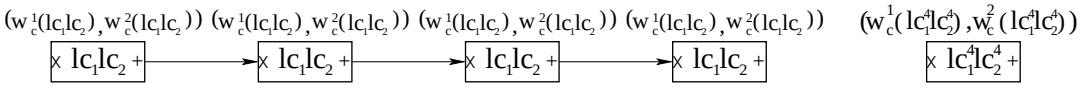


Figure 3.8: Dual-weight 4-stage MAC module.

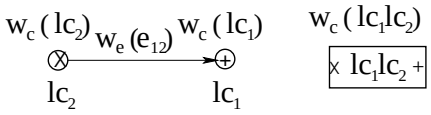


Figure 3.9: Single-weight MAC module with edge weight.

During the clustering procedure of components or modules, if a module M_i connects to a component c_j, then the edge weight is the amount of data transferred from the last component in module M_i to component c_j. If the module M_i connects to another module M_j, then the edge weight is the amount of data transferred from the last component in the module M_i to the first component in the module M_j.

By using the operation-level function model, we can express the n operations of an image processing algorithm in a component (or module) weight vector $\mathbf{W_c}$, where $\mathbf{W_c} = [w_c(c_1), \cdots, w_c(c_n)]$. The communication between components or modules are expressed as an $n \times n$ edge weight matrix $\mathbf{W_e}$, and

$$\begin{bmatrix} 0 & w_e(e_{12}) & \cdots & w_e(e_{1(n-1)}) & w_e(e_{1n}) \\ w_e(e_{21}) & 0 & \cdots & w_e(e_{2(n-1)}) & w_e(e_{2n}) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_e(e_{(n-1)1}) & w_e(e_{(n-1)2}) & \cdots & 0 & w_e(e_{(n-1)n}) \\ w_e(e_{n1}) & w_e(e_{n2}) & \cdots & w_e(e_{n(n-1)}) & 0 \end{bmatrix}$$

If no communication exists between c_i and c_j, then $w_e(e_{ij}) = 0$. The diagonal elements of the matrix, W_e, is assigned to 0, indicating that there is no communication on the component or the module itself.

3.1.2 Component Clustering Algorithm

At the partitioning stage, the local information of components and edges always has large effect on the quality of partitioning results. For example, the linear and the multi-level K-L partitioning algorithms that use the locality of components in most time perform better than the scattered algorithm that ignores the local information. In the proposed function model, the operations of an image processing algorithm are modeled as components or modules. It is necessary to take the local information into account and provide a model with appropriate granularity to the succeeding mapping process. Hence, we take advantage of the hierarchical structure in the proposed operation-level function model, and present a component clustering algorithm for weighted function models.

Similar to the granularity on data partitioning as we analyzed in Sec. 2.3.1, the granularity in function model significantly influences the partitioning result. In the operation-level function model shown in Fig. 3.4, granularity is determined by the weights of components and modules. Large modules always result in coarse granularity partitioning. Small components sometimes result in fine granularity partitioning depending on mapping algorithm. Theoretically fine granularity leads to better load balance, but induces more communications [72]. On the contrary, coarse granularity reduces communications but prohibits good load balance. The selection of granularity is a trade-off between load balance and communication. How to obtain an optimal partitioning granularity remains a challenge and varies for different applications and resources. The idea of the proposed component clustering algorithm is to incorporate the local information of components and edges when performing the partitioning. This is a heuristic approach but is very effective as will be shown in the performance evaluation in chapter 6.

In order to obtain coarser granularity that may result in a more efficient partitioning solution, we combine several components to produce a module. Most existing coarse partitioning algorithms [72, 73, 82] group several small weight components into a large one so as to reduce communications between components. Such clustering enables less communications, however, leads to potentially worse load balance since these small weight components may be very important in balancing loads in the next mapping procedure. Therefore, we group small weight and large weight components, instead of only small weight ones, into a module. Due to the reduction in the number of edges, the overall communication would decrease. Meanwhile, the load balance is not worsened, but improved since the component clustering algorithm pre-processes the function model using the local information of components and edges.

Suppose a function model F with n basic components to be clustered into k partitions. An $n \times n$ *adjacency matrix* $A = (a_{ij})$ is defined to represent the connections of F, where $a_{ij} = 1$ if components c_i and c_j have a connection, otherwise $a_{ij} = 0$. The *weighted degree* $deg(c_i)$ represents the communication on the component c_i, and is calculated by the summation of the amount of data sent in and out of c_i. It is expressed as

$$deg(c_i) = \sum_{j=1}^{n} w_e(e_{ij}) \times a_{ij} \tag{3.4}$$

We first identify four types of components: components with small weight and light communication; components with small weight and heavy communication; components with large weight and light communication; and components with large weight and heavy communication. The weight refers to the component weight, and the communication is measured by the weighted degree. The component clustering algorithm is to leave the extreme components alone and only cluster the rest. The component with small weight and light communication is useful in load balancing, so we leave this type of components untouched. The component with large weight and heavy communication may generate very large module that is not good for load balancing, so we also leave this type of components untouched. We are then left with the two so

called “hybrid” components, that is, components with large weight and light communication or components with small weight and heavy communication. We start the component clustering process from the starting components that receive data from external inputs. For component c_i and its neighbor component c_j, if

1. Weighted degree $deg(c_i) > deg(c_j)$;
2. Component weight $w_c(c_i) < w_c(c_j)$; and
3. Current module weight $[w_c(c_i) + \sum_{c \in M_{c_i}} w_c(c) + w_c(c_j)] \leq \bar{w}(p)$, where M_{c_i} is the set of components already grouped by c_i, $\bar{w}(k)$ is the average load weight on k partitions and

$$\bar{w}(k) = \frac{\sum_{i=1}^{n} w_c(c_i)}{k} \tag{3.5}$$

then we combine components c_i and c_j to a module. The first condition compares the weights of two components. The second condition compares the communications of two components. These two conditions reflect the two types of components we want to group. The third condition is to prevent the clustering process from generating too large weight module that may affect the load balance. Since the n components are to be divided into k partitions, we use the average load weight $\bar{w}(k)$ as a threshold. If c_i and c_j are combined, we lock c_j and check the next neighbor component of c_i. When a component is locked, it cannot combine with or be combined by other components in the rest clustering procedure. Until all neighbor components of c_i are checked, if c_i combines any of its neighbor components into a module, c_i is locked as well. The clustering procedure continues until all of the n components in function module F have been checked.

The component clustering algorithm for weighted function model is summarized in Algorithm 1.

Let us take the 3×3 spatial filter as an example and divide the components into 5 partitions. Figure 3.10 shows the original 3×3 filter structure and the component clustering result according to Algorithm 1.

Input: component weight vector $\mathbf{W_c}$; edge weight matrix $\mathbf{W_e}$; k predefined

Output: new component weight vector $\mathbf{W'_c}$; new edge weight matrix $\mathbf{W'_e}$

initialization;

generate adjacency matrix $\mathbf{G}$;

calculate the average weight sum $\bar{w}(k) = \frac{\sum_{i=1}^{n} w_c(c_i)}{k}$;

for *all components* **do**

  calculate the weighted degree $deg(c_i) = \sum_{j=1}^{n} w_e(e_{ij}) \times a_{ij}$;

end

for *all components* **do**

  if *current component* c_i *is unlocked* **then**

    for *all of its neighbor components,* c_j **do**

      if *current neighbor* c_j *is unlocked* **then**

        if $deg(c_i) > deg(c_j)$ & $w_c(c_i) < w_c(c_j)$ &

        $[w_c(c_i) + \sum_{c \in M_{c_i}} w_c(c) + w_c(c_j)] \leq \bar{w}(k)$ **then**

          group c_i and c_j;

          update new component weight vector $\mathbf{W'_c}$;

          update new edge weight matrix $\mathbf{W'_e}$;

          lock c_j;

        end

      end

    end

    if c_i *combines with any of its neighbors* **then**

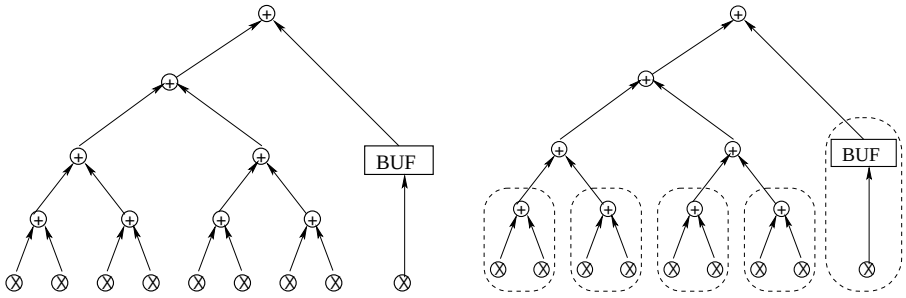
      lock c_i;

    end

  end

end

Algorithm 1: Component clustering algorithm.



(a) Original structure. (b) Component clustering result.

Figure 3.10: A component clustering example for the 3×3 spatial filter.

Suppose the weights of addition and multiplication components are respectively 1 and 2, and that of the buffer component is 1. Then $\bar{w} = 5.4$. During the component clustering procedure, the multiplication components at the bottom level do not combine any of their neighbors, since neither the condition 1 nor the condition 2 is satisfied. Each of the addition components at the second level combines two multiplication components into a module whose weight is 5. The buffer component combines its multiplication neighbor into a module with a weight of 4. The addition components at the third and the fourth level do not satisfy condition 1. Meanwhile, the addition components at the second level have been locked. Therefore, no further clustering is performed. We will show the effectiveness of the proposed component clustering algorithm on mapping in Chapter 6.

3.1.3 Cyclic Process Modeling

Some complicated image processing algorithms contains not only the acyclic processes as we have modeled in the operation-level function model, but also the cyclic processes that form closed loops on several operations.

In a cyclic process, the operations enclosed in the loop are iteratively executed until the stop condition is satisfied. The stop condition may be convergence or a pre-defined number of

iterations. If every component in the function model is a part of the cyclic process, the number of iterations does not affect the partitioning. Then we can ignore the stop condition and use one iteration to decide component weights and edge weights. Otherwise, if the function model is a mixture of both cyclic and acyclic processes, we need to convert the cyclic processes to the acyclic processes, and use the number of iterations to decide component weights and edge weights in the cyclic process. Therefore, if the stop condition is convergence, we have to assume the number of iterations based on prior experience. In this section, our discussion focuses on the function model that is a mixture of both cyclic and acyclic processes.

By using the hierarchical function model, we can simply cluster the components involved in a cyclic process into a module M. The number of iterations $Iter$, pre-defined or given based on prior knowledge, is assigned to this module. So the weight of the module is $w_c(M) \times Iter$. Since the communication within a module can be ignored, the number of iterations does not affect the edge weight of module M. The granularity of such model, however, may be too coarse and not appropriate for mapping. We therefore decompose the cyclic process module M in the hierarchical function model to small modules $[M_1, \cdots, M_{\|M\|}]$. The number of iterations is also related to these modules as well as the communications between them, in which the module weights are $[M_1, \cdots, M_{\|M\|}] \times Iter$, and the edge weights in the module M are

$$\begin{bmatrix} 0 & w_e(e_{12}) & \cdots & w_e(e_{1\|M\|}) \\ w_e(e_{21}) & 0 & \cdots & w_e(e_{2\|M\|}) \\ \vdots & \vdots & \ddots & \vdots \\ w_e(e_{\|M\|1}) & w_e(e_{\|M\|2}) & \cdots & 0 \end{bmatrix} \times Iter.$$

Let us use the pICA algorithm, which will be described in detail in Chapter 4, as an example to demonstrate the cyclic process decomposition procedure. In pICA, the sub-matrix estimation contains two cyclic processes: the one-unit process and the internal decorrelation process. The external decorrelation forms another cyclic process. The stop conditions of these cyclic processes are convergence. In Fig. 3.11, the middle level of the hierarchical function model shows the way we usually model the pICA algorithm.

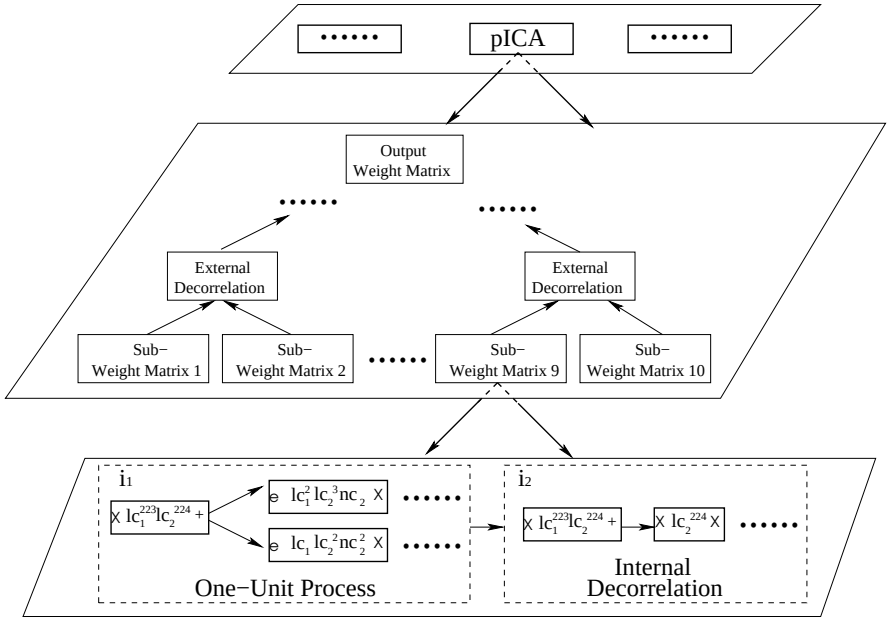


Figure 3.11: Decomposition of cyclic process.

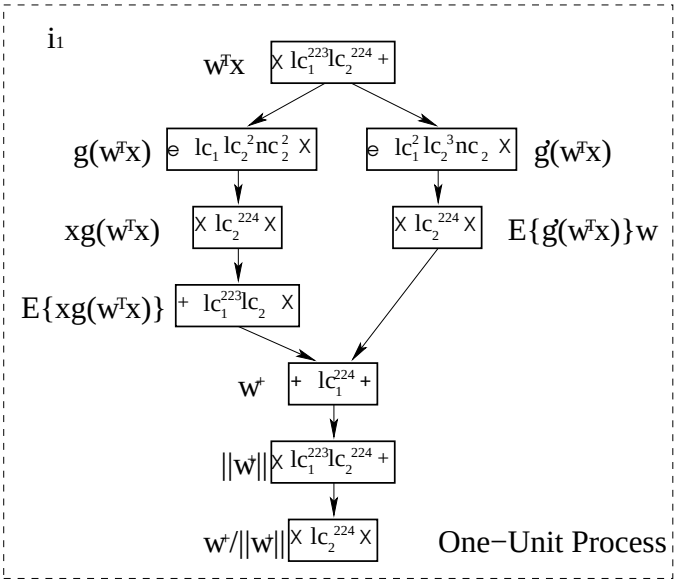
The sub-matrix estimation and the external decorrelation processes are respectively grouped into large modules. In the cyclic process modeling, as shown at the bottom level in Fig. 3.11, the cyclic processes are decomposed into small modules. Figure 3.12 demonstrates the detailed modeling of the one-unit and the internal decorrelation processes. In each of these two cyclic processes, we use small modules that consist of components to model the detail operations. Since the stop conditions of these two cyclic processes are convergence, the numbers of iterations i_1 and i_2 are given based on prior experience. In the internal decorrelation process, j_{21} represents the number of the decorrelated weight vectors. It varies depending on the layer of decorrelation process.

Such decomposition provides function model with finer granularity to the mapping procedure. We will show the effectiveness of the cyclic process modeling through experiments in Chapter 6.

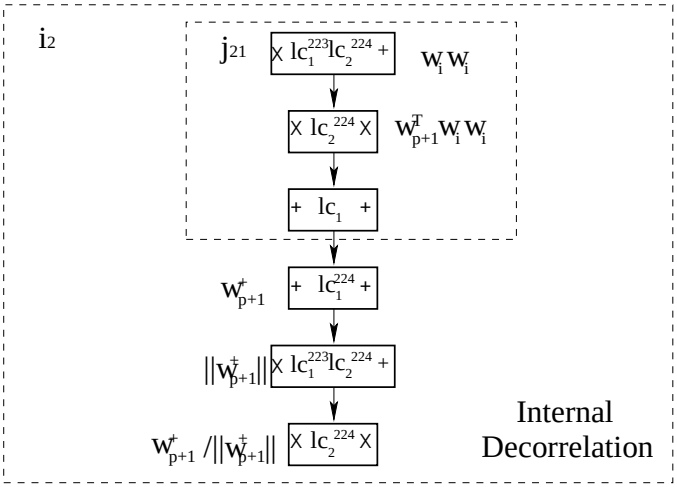
3.2 Data Dependency Analysis

Most image processing applications experience not only complicated algorithms but also large data sets. We have discussed the function modeling for algorithms in the previous section. In this section, we analyze the data dependency so as to efficiently distribute data to the participating resources in pipelined and parallel processing.

During the algorithm implementation procedure, the data dependency refers to a situation where the output of a process is based upon the results of processes that previously completed, or being conducted but not yet completed. According to the structure of various algorithms, we divide data dependency into three categories, i.e., independency, uniform dependency, and regional dependency.



(a) One-Unit.



(b) Internal decorrelation.

Figure 3.12: Cyclic process modeling.

3.2.1 Definitions

In order to analyze the data dependency existing in the input image, we need to give some definitions to represent the relationship between pixels. Let us first consider a single-band or gray-scale 2-D image, i.e., a pixel is associated with only one value, and then extend the analysis to 3-D images.

Suppose an input image I has $m \times n$ pixels and $I = (i(1,1), \cdots, i(x,y), \cdots, i(m,n))$, where (x,y) indicates the pixel coordinate. A function $f[.]$ takes an input pixel $i(x,y)$ from image I and produces an output $o(x,y)$. For the input image I, we define the dependency edge as follows.

Definition 11. *A dependency edge b is a uni-directional connection from $i(x_p, y_p)$ to $i(x_q, y_q)$, denoting that the output $o(x_p, y_p)$ depends on the value of both $i(x_p, y_p)$ and $i(x_q, y_q)$, i.e., $o(x_p, y_p) = f[i(x_p, y_p), i(x_q, y_q)]$.*

The dependency model of I is defined as $D = (I, B)$ that has $\|B\|$ dependency edges and $B = (b_1, b_2, \cdots, b_{\|B\|})$.

Proposition 1. *Given an $m \times n$ input image, the lower bound of the number of dependency edges is 0, and the upper bound of the number of dependency edges is $\|I\| \times (\|I\| - 1)$, where $\|I\| = m \times n$ indicating the number of pixels in the input image. Therefore $0 \leq \|B\| \leq \|I\| \times (\|I\| - 1)$.*

Proof: If every output depends only on the current input pixel, then no dependency edges exists and the number of dependency edges is 0.

If every output depends on all pixels of the input image, then the number of dependency edges for one output is $\|I\| - 1$, and totally $\|I\| \times (\|I\| - 1)$ edges for the entire image I. $\square$

3.2.2 Independency

The data independency is defined as follows.

Definition 12. *An input image is independent if and only if every pixel output depends only on the current input pixel but no others. That is, for each pixel output*

$$o(x, y) = f[i(x, y)], i \in I \tag{3.6}$$

In the area of image processing, algorithms that are performed on independent input images are also referred to as point-based image processing. A couple of commonly used such algorithms include contrast stretching, bit-plane slicing, log transformation, power-law transformation, etc. [69]. These algorithms are normally conducted in the processing stream(s) without the requirement of any extra storage space.

The data independency gives high efficiency to data distribution in parallel and pipelined computing diagrams. No coordination between pixels is required. The input image is partitioned and distributed only according to the computing capabilities of individual resources.

3.2.3 Uniform Dependency

Compared to the data independency, the data dependency needs information of both the current pixel and other pixels in the input image. The data dependency requires complicated coordination strategies in data distribution. Most likely it also needs extra storage space and frequent I/O operations.

Depending on the features of individual functions, some data dependencies may be *uniform* to every output and spread through the entire image. This kind of dependency is called the *uniform dependency*. Some dependencies may only exist regionally in the input image. This kind of dependency is called the *regional dependency*. This section focuses on the discussion on the uniform dependency and the regional dependency will be described in the next section.

The input images for many neighborhood-based image processing algorithms contain the uniform dependency. Examples include the spatial smoothing and sharpening filters, Gaussian and low-pass filters, etc. We take the general 3×3 spatial filter as an example to analyze the

uniform dependency. The 3×3 filter can be expressed as

$$o(x,y) = i(x-1,y-1) \times h(1,1) + i(x-1,y) \times h(1,2) + i(x-1,y+1) \times$$
$$h(1,3) + i(x,y-1) \times h(2,1) + i(x,y) \times h(2,2) + i(x,y+1) \times h(2,3) +$$
$$i(x+1,y-1) \times h(3,1) + i(x+1,y) \times h(3,2) + i(x+1,y+1) \times h(3,3) \quad (3.7)$$

where h is the 3×3 filter, as shown in Fig. 3.13.

Since the dependency exists for all outputs, the 3×3 filter window will go over the entire input image. By superimposing the filter on the image as demonstrated in Fig. 3.14, we observe that the window can move to 8 different directions among which movements along the horizontal and the vertical directions introduce three new input pixels while movements along the diagonal directions introduce five new input pixels.

We thus assign weights to the movement directions, as shown in Fig. 3.15, which represents the amount of new input pixels required by each movement. The weight of the dependency

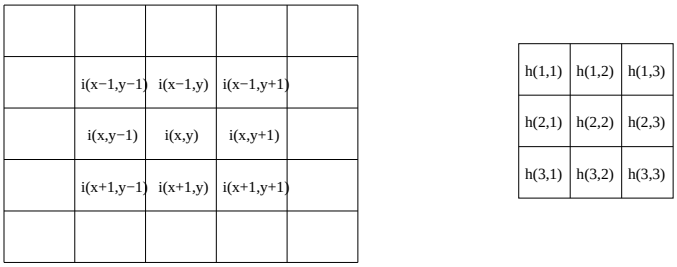


Figure 3.13: 3×3 spatial filter.

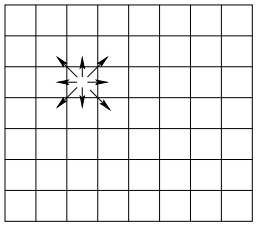


Figure 3.14: Uniform dependency.

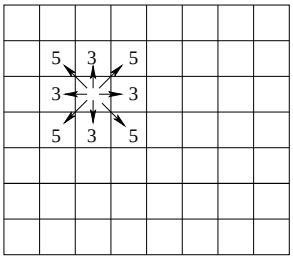


Figure 3.15: Uniform dependency with weight.

edge as can then be defined as follows.

Definition 13. *The weight $w_D(b)$ of a dependency edge b is the amount of new input pixels required if the window moves along the direction of the dependency edge.*

The data partitioning on image with uniform dependency is conducted according to both the computing capabilities of individual resources and the communication capabilities between each other. The objective of partitioning is to minimize the communications between data subsets.

3.2.4 Regional Dependency

For many image processing algorithms such as the edge detection filters, it is sometimes unnecessary to calculate all outputs in the same way, even though the uniform dependency exists in the input image. In other words, we can isolate some independent pixels in the input image by evaluating pixels before forwarding them to specific functions. Taking the 3×3 edge detection filter as an example, if the gray level of the new input pixel is similar to its neighbors, a predefined value can be directly set to the output without conducting the 3×3 filter calculation. Although we add comparator(s) as pre-processing blocks to such design, the comparator takes much less resource and leads to faster processing compared to multipliers and adders.

In regions with dependencies as shown in Fig. 3.16, the weights are assigned to individual directions. In each of these regions, we group all of the involved pixels into one block, which

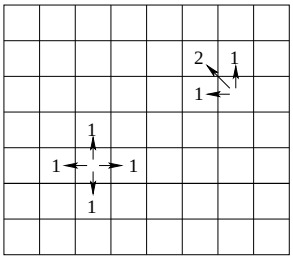


Figure 3.16: Regional dependency with weight.

is treated as a black box by other pixels. As demonstrated in Fig. 3.17, the entire image is then modeled as multiple blocks with different granularities according to the regional dependencies. In this model we observe that both single pixels and blocks are independent to each other. The data partitioning is then conducted as we have discussed in Sec. 3.2.2, that is, according only to the computing capabilities of individual resources.

Next, we analyze more complicated regional dependency. Suppose different data dependencies exist in one region as shown in Fig. 3.18. The calculation of the output at the central pixel depends on four of its neighbors along the vertical and the horizontal directions, while the calculations of other pixels in this region need only two neighbors.

In order to reduce communications, we give higher grouping priority to pixels with more dependencies. So we first group the central pixel and its four neighbors because its calculation requires more communications.

At this point, the 5-pixel block is a black box to its surrounding pixels. The data dependency at this level is uniform. Secondly, the region is grouped into a bigger 15-pixel block as shown in Fig. 3.19. At this point, the 5-pixel block is a black box to its surrounding pixels. The data dependency at this level is uniform. Finally, we obtain the model of entire image containing the 15-pixel block as unit. So the pixel clustering in the hierarchical model is a bottom-up procedure. The relationship between pixels and blocks at each level are either independency or uniform dependency.

On the contrary, the data partitioning is executed following a top-down procedure. At the

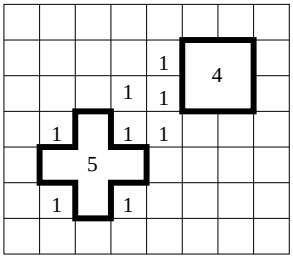


Figure 3.17: Distributing pixels with regional dependency. The number on each data indicates the assigned resource.

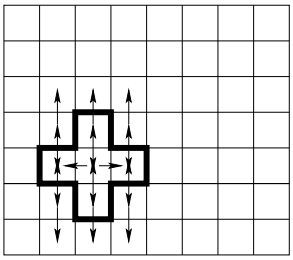


Figure 3.18: Different dependencies in the same region.

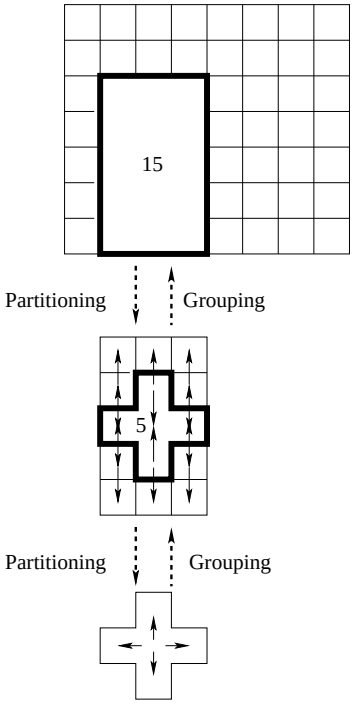


Figure 3.19: Hierarchical pixel clustering and partitioning.

top level, pixels and blocks are initially partitioned according to the computing capabilities of individual resources. If a block is very large and affects the load balance significantly, then we move to the lower level and further partition this block. The hierarchical partitioning procedure may continue until the lowest level.

3.2.5 Dependency in 3-D Images

In Secs. 3.2.2 to 3.2.4, we have analyzed the data dependency on 2-D images. In this section, we extend our discussion to 3-D images, including, for example, 2-D image sequences and multispectral images.

Suppose I is a 3-D image where (x, y, z) is the pixel coordinate. For example, in 2-D image sequences and multispectral images, z represents the temporal and spectral axis respectively, as demonstrated in Fig. 3.20.

Combining the previous discussion of the data dependency on 2-D images, we identify three types of dependencies for 3-D images.

Case 1. No dependency exists between different image frames along the z axis.

As shown in Fig. 3.21, if the image frames along the z axis are independent to each other, then the 3-D image can be treated as multiple 2-D images. Then the modeling and partitioning procedure would be exactly the same as the one we have presented in Secs. 3.2.2 to 3.2.4.

Case 2. Uniform dependencies exist along the z axis, or both along the z axis and within each individual image frame.

In many applications, dependency analysis is focused on the z axis. If the uniform dependency exists only along the z axis as demonstrated in Fig. 3.22(a), then the basic partitioning unit is one column along the z axis. We can follow the same rules described in Sec. 3.2.3 to distribute pixels according to both the computing capabilities of individual resources and the communication capabilities between each other.

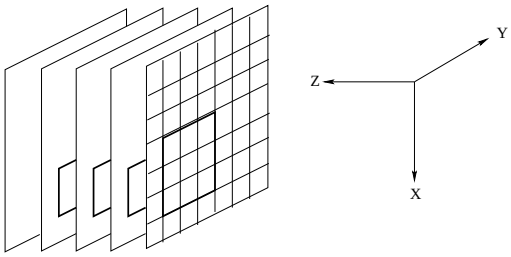


Figure 3.20: Data dependency on image sequence or multispectral image.

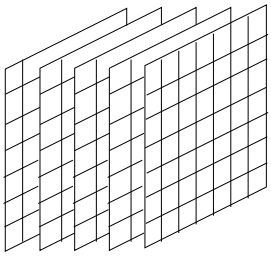
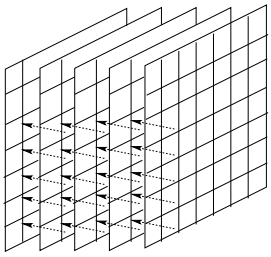
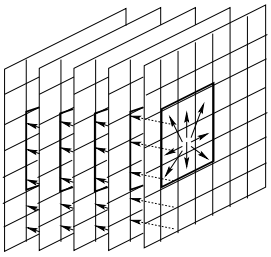


Figure 3.21: Independency along the z axis.



(a) No dependency on image frames.



(b) Both along the z axis and on the same image frame.

Figure 3.22: Uniform dependencies exist along the z axis.

If the uniform dependencies exist along all three axes as shown in Fig. 3.22(b), the processing window we described in Sec. 3.2.3 therefore moves in the 3-D image. Correspondingly, the weight along each possible movement direction is determined by the number of new input pixels required by each movement. According to the computing and the communication capabilities of resources, the data partitioning is conducted to minimize the communications between subsets.

Case 3. Regional dependencies exist along the z axis, or both along the z axis and within each individual image frame.

The most complicated case is that regional dependencies exist along the z axis, or both along the z axis and within the same image, as shown in Fig. 3.23. By following the principles of the hierarchical data clustering and partitioning described in Sec. 3.2.4, we expand this method from 2-D images to 3-D images. After grouping the involved pixels according to the regional dependencies, several 3-D blocks are formed in the input image. During the grouping procedure, the relationship between pixels and blocks at the same level are either independent or uniformly dependent. The partitioning is again conducted in a top-down procedure according to the computing and communication capabilities of resources.

3.2.6 Data Distribution Schemes

According to the data dependencies on 2-D and 3-D images, we have proposed the correspond-

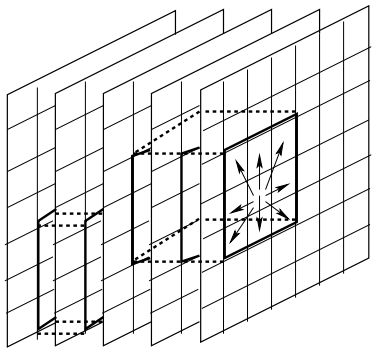


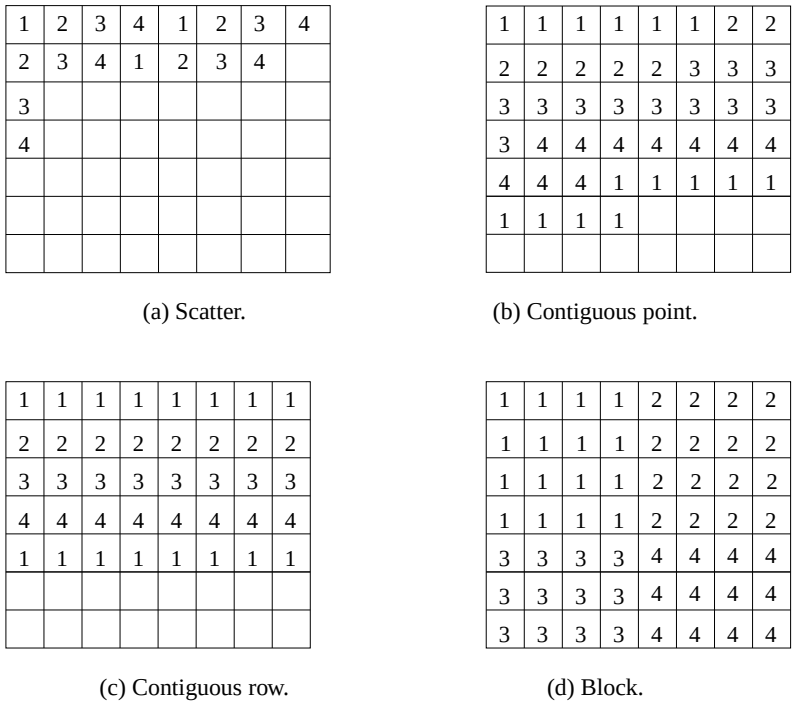
Figure 3.23: Regional dependencies exist both along z axis and on the same image.

ing partitioning strategies. It is safe to assume that the entire input image is homogeneous as the blocks formed after data dependency analysis is either independency or uniform dependency.

Then we apply four commonly used distribution schemes, i.e., (a) scatter, (b) contiguous point, (c) contiguous row, and (d) block, as shown in Fig. 3.24, where a unit is one input data (i.e., one pixel) or one data block.

The scatter scheme distributes pixels along two directions at the same time. This scheme alternately sends pixels to individual computing resources so as to balance the processing without considering the communication between resources. Hence, this scheme is suitable for independent images where no communication is required during the calculation of different outputs.

The contiguous point scheme distributes pixels along one direction until a pre-defined number or the resource constraint is reached. We can use this scheme to implement images with



(a) Scatter.

(b) Contiguous point.

(c) Contiguous row.

(d) Block.

Figure 3.24: Data distribution schemes.

independencies or uniform dependencies on heterogeneous systems where computing capabilities of participating computing resources may vary significantly.

The contiguous row scheme takes one row as the basic unit in data distribution. This scheme is appropriate to images with uniform dependencies. The difference between the contiguous point and the contiguous row schemes lies in that the contiguous row scheme does not interrupt the dependency along the row direction, thereby reducing the communication time caused by the dependency along the row direction. But the contiguous point scheme is more practical since it fully utilizes the computing capabilities of individual resources when the communication is not significant. In general, the contiguous row scheme can be treated as a special case of the contiguous point scheme.

Compared to the other three schemes, the block scheme is more flexible and fits images with independency, uniform dependency, and especially regional dependency. It divides the entire image into several subsets and distributes all pixels in each subset to the most appropriate resource. The advantage of this scheme lies in that the data distribution may completely rely on the dependency in the input image, thereby reducing the communication time. But the balance between the computing and the communication is desired at the same time.

Obviously, these four schemes are the basic cases in data distribution. They can be easily extended and combined to make the most appropriate distribution scheme for images with specific dependencies.

The distribution schemes in 3-D images are similar. For examples, the scatter scheme can be extended from two to three directions; the contiguous row scheme may be conducted along either the x, y, or z axis.

As we have observed, communication is a very important issue in both data partitioning and distribution schemes. In data dependency analysis we can only decide the amount of data to be transfered. The modeling and analysis of the communication will be discussed in next section.

3.3 Resource Modeling

In Secs. 3.1 and 3.2, we have modeled the functions in image processing algorithms using the operation-level model, and analyzed three categories of data dependencies. The mapping of both the function and the data set concerns the problem of

1. how to optimally assign operations and data partitions to individual resources according to the computing capabilities of the corresponding resources;

2. how to efficiently transfer data between operations and data partitions according to the communication capabilities between resources; and

3. how to balance the computing time and the communication time therefore obtaining the minimum overall processing time.

All of these concerns are related to the computing resources that eventually conduct the processing. In this section we model the computing resource with considerations of both the computing and the communication capabilities.

Generally speaking, the computing resources to implement the parallel and pipelined processes are either multi-processor systems, hardware/software co-processing systems, or VLSI systems as we have reviewed in Chapter 2. Depending on the features of individual computing resources, they can also be divided into homogeneous or heterogeneous systems.

In the computing resource modeling, the heterogeneous system is a more challenging instance. The homogeneous system can be treated as a special case of the heterogeneous system.

Sensors in VSNs form heterogeneous computing environments due to the variance of distances between sensors and the remaining power on individual sensors. Since computing and communication resources are very limited in VSNs, we need to conduct resource-oriented mapping so as to achieve on-site processing with less resource consumptions. In order to conduct resource-oriented mapping algorithms, we respectively assign weights $w_r(r_i)$ and $w_t(T_{jk})$ to

the computing and the communication resources. The weight is measured by certain performance parameters that may vary for different implementation systems. For example, the most concerned performance on multi-processor systems is normally the processing speed of each computing resource that is determined by the speed of the processor and the size of the memory. The VLSI systems concern either the speed (1/delay), the utilization area, or the power consumption, depending on specific applications.

Suppose there are p computing resources, the weight vector can then be expressed as $[w_r(r_1), \cdots, w_r(r_i), \cdots, w_r(r_p)]$. The communication resource may be bi-directional, uni-directional, or simply does not exist, depending on the available connection between two computing resources. If no physical connection exists between resource r_j and r_k, then $w_t(T_{jk}) = \infty$. In addition, $w_t(T_{jk})$ does not necessarily equal to $w_t(T_{kj})$. All communication resources are expressed in a $p \times p$ weight matrix

$$\begin{bmatrix} \infty & w_t(T_{12}) & \cdots & w_t(T_{1(p-1)}) & w_t(T_{1p}) \\ w_t(T_{21}) & \infty & \cdots & w_t(T_{2(p-1)}) & w_t(T_{2p}) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_t(T_{(p-1)1}) & w_t(T_{(p-1)2}) & \cdots & \infty & w_t(T_{(p-1)p}) \\ w_t(T_{p1}) & w_t(T_{p2}) & \cdots & w_t(T_{p(p-1)}) & \infty \end{bmatrix}$$

The diagonal elements of the matrix is assigned to ∞, which means a resource cannot communicate with itself or the operation is invalid.

Let us take the multi-processor system as an example. Since the communication is commonly evaluated by the transfer delay between resources in multi-processor systems, the weight of communication resource is decided by the network delay, the overhead, and the synchronization time. In this case, $w_t(T_{jk})$ is calculated by

$$w_t(T_{jk}) = 2 \times o + t_d \times n_d \times s_d + l \tag{3.8}$$

where o is the message preparation latency, t_d is the transmission latency (s/Byte) on connection and $t_d = \frac{1}{\textit{transfer speed}}$, n_d is the number of data items, s_d is the size of each data item to be transmitted, and l is the synchronization time.

3.4 Mapping Procedure

After modeling image processing algorithms in the multi-weighted operation-level function model and analyzing the input image according to dependencies, we conduct partitioning on functions and image so as to efficiently map them to specific computing resources. In this section, we focus on the function mapping and describe the mapping procedure in detail. We first formulate the objective functions of the mapping problem, then study the resource-oriented function mapping in heterogeneous computing environments.

3.4.1 Objective Functions

The objective of partitioning varies from one application to another, depending on specific application requirements and resource constraints. The objective function of partitioning in this work only concerns one objective at a time. For example, the partitioning objective of some power-aware sensor networks is to balance the load on participating resources so as to prolong the lifetime of the entire network, and the objective of some real-time processing embedded systems is to minimize the overall processing time. Commonly used objectives include minimizing (1) the load variance on individual partitions, (2) the communication between partitions, and (3) the overall processing cost, etc. In the load balance objective, since we have precisely modeled image processing algorithms at the operation level, we simply ignore the slack time, which is the amount of time that a sub-process can be delayed without delaying the entire process, in the mapping process.

In heterogeneous environments, the variances of both computing and communication resources are taken into account, and the partitioning performance is evaluated in either the pro-

cessing time or the power consumption. Let us take the processing time as an example and follow the same definitions given in Sec. 2.4.1. The component weight $w_c(c_i)$ therefore denotes the delay of component c_i, and the edge weight $w_e(e_{ij})$ still denotes the amount of data to be transfered on edge e_{ij}. The objective of load balance is to find a partitioning $\hat{P^k}$ such that the variance of computing times on individual resources is minimized. Given k partitions, the objective function is written as

$$\arg\min_{P^k}\{\sum_{i=1}^{k}[w_c(V_i) \times w_r(r_i) - E_r(V)]^2\} \tag{3.9}$$

where

$$E_r(V) = \frac{\sum_{j=1}^{k} w_c(V_j) \times w_r(r_j)}{k} \tag{3.10}$$

and $w_r(r_i)$ represents the processing speed of the resource r_i. The objective of minimizing communication is to find a partitioning $\hat{P^k}$ such that the overall communication time on k resources is minimized. The objective function is written as

$$\arg\min_{P^k}\{\sum_{i=1}^{k}\sum_{j=1,j\neq i}^{k} w_e(P_{ij}) \times w_t(T_{ij})\} \tag{3.11}$$

where $w_t(T_{jk})$ is the transmission speed of the connection T_{ij}. The objective of minimizing overall processing cost is to find a partitioning $\hat{P^k}$ such that the overall processing time, including the computing and the communication time, is minimized. The objective function is written as

$$\arg\min_{P^k}\{t_{comp} + t_{comm}\} \tag{3.12}$$

where

$$t_{comp} = \sum_{i=1}^{k} w_c(V_i) \times w_r(r_i) \tag{3.13}$$

and

$$t_{comm} = \sum_{i=1}^{k} \sum_{j=1, j \neq i}^{k} w_e(P_{ij}) \times w_t(T_{ij}) \tag{3.14}$$

If the objective is to minimize the power consumption, the formulations of the objective functions follow similar equations.

3.4.2 Resource-oriented Mapping in Heterogeneous Environments

In heterogeneous environment, computing capabilities on individual resources and communication capabilities between resources may vary significantly. So mapping algorithms for heterogeneous environments should take the variances of computing or communication capabilities into consideration in order to efficiently allocate components to resources. The objective functions, as shown in Eq. 3.9 to 3.12, also incorporate features of resources.

In order to satisfy the needs for different applications, we propose (1) the load attraction mapping algorithm that minimizes load variance between computing resources, and (2) the communication attraction mapping algorithm that minimizes the overall communication cost. Both of them can be combined with the local refinement to minimize the overall processing cost. The cost can be measured by processing time or power consumption, depending on specific applications. The objective function only incorporates one cost measurement at a time.

Load Attraction Mapping Algorithm

The idea of the load attraction mapping algorithm is to first allocate computationally expensive (large weight) components to resources with more computing capability, i.e., lower computing cost. Then the smaller and simpler tasks are allocated to resources for the purpose of load balancing.

Given p computing resources $(r_1, \cdots, r_p)$ and n components $(c_1, \cdots, c_n)$. Let the cost be measured by the processing time. The $1 \times p$ vector $\mathbf{W_r} = [w_r(r_1), \cdots, w_r(r_p)]$ represents the processing speed of individual resources. The $1 \times n$ vector $\mathbf{W_c} = [w_c(c_1), \cdots, w_c(c_n)]$

represents the weight of each component. The elements in the component weight vector, $\mathbf{W_c}$, is first sorted in the descending order to form a sorted vector $\mathbf{W'_c}$ according to the weight, where $w_{c_i} \geq w_{c_j}$ if $i < j$. The assignment process starts from the component with the largest weight to that with the smallest weight. For the i^{th} component c_i in $\mathbf{W'_c}$, we search for the target resource $\hat{r}$ such that

$$\arg\min_{\hat{r},j=1,\cdots,p}\{w_r(r_j) \times [w_c(V_j) + w_c(c_i)]\} \tag{3.15}$$

and the load weight $w_c(V_j)$ on the resource r_j is

$$w_c(V_j) = \sum_{c \in V_j} w_c(c) \tag{3.16}$$

Then component c_i is assigned to the target resource $\hat{r}$. The assignment process repeats until all components are assigned to resources. Obviously, the components with larger weights are preferably assigned to the resources with smaller delay during the assignment procedure, and components with smaller weights are used to balance the loads on individual resources. The objective function of the load attraction mapping algorithm has been expressed in Eq. 3.9.

The detailed algorithm is summarized in Algorithm 2.

The computational complexity of the load attraction algorithm is $O(\max\{np, nlogn\})$, where $O(nlogn)$ is the complexity of the sorting process and $O(np)$ is the complexity of the 2-level for loop.

Communication Attraction Mapping Algorithm

Similar to the load attraction algorithm, the key idea of the communication attraction algorithm is to allocate high-volume data communications to connections with smaller costs, i.e., higher transmission speed or lower power consumptions. The objective is to minimize the overall communication cost that is represented by the cut weight, as shown in Eq 3.11.

Input: computing resource weight vector $\mathbf{W_r}$, component weight vector $\mathbf{W_c}$

Output: assignment vector $\mathbf{G}$, load variance

initialization;

sort elements in $\mathbf{W_c}$ in the descending order and obtain $\mathbf{W'_c}$;

for *all components in* $\mathbf{W'_c}$ **do**

  for *all resources in* $\mathbf{W_r}$ **do**

    calculate the temporary load weight $w'_c(V_j) = \sum_{c \in V_j} w_c(c) + w_c(c_i)$;

  end

  search for the target resource $\hat{r}$ such that $\arg\min_{\hat{r}, j=1,\cdots,p}\{w_r(r_j) \times w'_c(V_j)\}$;

  assign current component c_i to the target resource $\hat{r}$;

  update g_i (the assignment of c_i) in $\mathbf{G}$;

  update the load weight $w_c(\hat{V})$ ($\hat{V}$ is the load on $\hat{r}$) ;

end

calculate load variance $\sum_{i=1}^{k}[w_c(V_i) \times w_r(r_i) - \frac{\sum_{j=1}^{k} w_c(V_j) \times w_r(r_j)}{k}]^2$;

output $\mathbf{G}$ and load variance;

Algorithm 2: The load attraction mapping algorithm.

Let the cost be measured by the communication. Given p computing resources $(r_1, \cdots, r_p)$ and n components $(c_1, \cdots, c_n)$. The $p \times p$ communication resource weight matrix $\mathbf{W_t} = \{w_t(T_{ij})\}$ is composed of the transmission speed between pairs of resources, where $w_t(T_{ii}) = \infty$ and $1 \leq i, j \leq p$. The $n \times n$ weight matrix $\mathbf{W_e} = \{w_e(e_{ab})\}$ represents the edge weights between components, where $w_e(e_{aa}) = 0$ and $1 \leq a, b \leq n$. Similarly, the assignment process starts from edges with larger weights. We first sort edges in the descending order to form vector $\mathbf{W'_e}$ according to the weight. For the edge e_{ab}, we search for the target connection $\hat{T}$ such that

$$\arg\min_{\hat{T}, i, j=1, \cdots, p} \{w_t(T_{ij}) \times [w_e(P_{ij}) + w_e(e_{ab})]\} \tag{3.17}$$

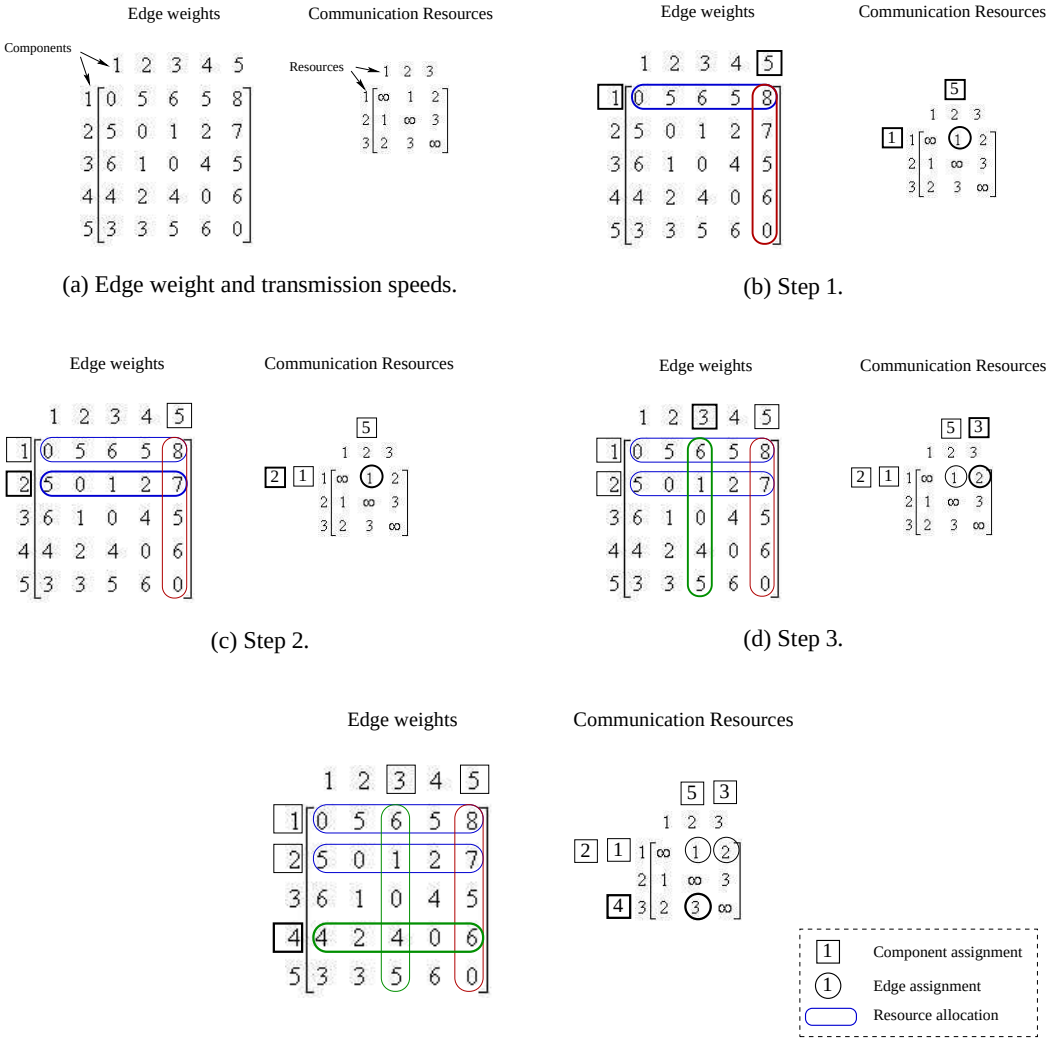
and the overall communication or cut weight $w_e(P_{ij})$ on connection T_{ij} is

$$w_e(P_{ij}) = \sum_{e \in P_{ij}} w_e(e) \tag{3.18}$$

So the edge e_{ab} is assigned to the target connection $\hat{T}$. In other words, the components c_a and c_b are respectively assigned to the starting resource $\hat{r}_s$ and the ending resource $\hat{r}_t$ of the target connection. The edge assignment process iterates until all components are assigned to resources. The cut weight in the communication attraction algorithm is calculated by

$$cut \;\; weight = \sum_{i=1}^{p} \sum_{j=1, j \neq i}^{p} w_e(P_{ij}) \times w_t(T_{ij}) \tag{3.19}$$

Let us use an example to present the communication attraction algorithm in detail. As shown in Fig. 3.25, the resource consists of 3 nodes and the function model includes 5 components. We start the assignment procedure from the largest edge weight $w_e(e_{15})$. In the communication weight matrix, T_{12} and T_{21} have the minimum weight, so the communication costs $w_t(T_{12}) \times w_e(e_{15})$ and $w_t(T_{21}) \times w_e(e_{15})$ are the minima. We randomly select T_{12} and assign components c_1 and c_5 to resources r_1 and r_2, respectively. The resource allocations at step



(a) Edge weight and transmission speeds.

(b) Step 1.

(c) Step 2.

(d) Step 3.

(e) Step 4.

Figure 3.25: Communication attraction mapping algorithm.

1 are highlighted on row 1 and column 5 of the edge weight matrix, representing the components and their edges are locked. To keep the figure less confusing, we only mark half of the symmetric resource allocation on the edge weight matrix. At step 2, $w_e(e_{25}) = 7$ is selected. Because component c_5 has been assigned to resource r_2, we can only select connections from T_{12} and T_{32}. Obviously, the overall cost $(w_t(T_{12}) \times [w_e(P_{12}) + w_e(e_{25})] = 15)$ is less than $(w_t(T_{32}) \times w_e(e_{25}) = 21)$. So edge e_{25} is assigned to connection T_{12}, and component c_2 is assigned to resource r_1. Following the same rule, edge e_{13} is assigned to connection T_{13} at step 3, and component c_3 is assigned to resource r_3. At step 4, since c_1 and c_3 have been locked, we ignore $w_e(e_{31})$ and move to $w_e(e_{45})$. Edge e_{45} is then assigned to T_{32}, and component c_4 is assigned to resource r_3. Up to this point, all components have been assigned to resources, so the process is terminated. For a function with n components, the assignment process can be finished in $n - 1$ steps. The computational complexity of the communication attraction mapping algorithm is thus $O(\max\{n^2p^2, n^2log(n^2)\})$, where $O(n^2log(n^2))$ is the complexity of the sorting process and $O(n^2p^2)$ is the complexity of the 2-level for loop.

The communication attraction mapping algorithm is summarized in Algorithm 3.

Local Refinement

During the mapping procedure, the load attraction algorithm concerns about the component weights and the computing capabilities on different resources, and the communication attraction algorithm takes the edge weights and the communication capabilities between resources into consideration.

In some applications, however, the overall processing cost is the primary concern. So we need combine the proposed algorithms with the local refinement in order to minimize the overall processing cost.

Input: communication resource weight matrix $\mathbf{W_t}$, edge weight matrix $\mathbf{W_e}$

Output: assignment vector $\mathbf{G}$, cut weight

initialization;

sort edges$\neq 0$ in $\mathbf{W_e}$ in the descending order and obtain $\mathbf{W'_e}$;

for *all edges in* $\mathbf{W'_e}$ **do**

  if *there is an unlocked component* **then**

    if *at least one component on current edge is unlocked* **then**

      for *all connections eligible for current edge* **do**

        calculate the temporary communication weight

        $w'_e(P_{ij}) = \sum_{e \in P_{ij}} w_e(e) + w_e(e_{ab})$;

      end

      search for the target connection $\hat{T}$ such that

      $\arg\min_{\hat{T},i,j=1,\cdots,p}\{w_t(T_{ij}) \times w'_e(P_{ij})\}$;

      assign current edge to the target connection $\hat{T}$;

      update the communication $w_e(\hat{P})$;

      assign components c_a and c_b to the starting resource $\hat{r}_s$ and the ending

      resource $\hat{r}_t$, respectively ;

      update assignment vector $\mathbf{G}$;

      lock components c_a and c_b ;

    end

  end

end

calculate $cut\ \ weight = \{\sum_{i=1}^{p} \sum_{j=1,j\neq i}^{p} w_e(P_{ij}) \times w_t(T_{ij})$;

output $\mathbf{G}$ and cut weight;

Algorithm 3: The communication attraction mapping algorithm.

Suppose the cost is measured by the processing time t. Given the assignment vector $\mathbf{G}$ for n components on p resources, t were expressed in Eq. 3.12 to 3.14. We conduct the K-L local refinement to minimize t. For two components assigned to two different resources, if the overall processing time decreases after we switch these two components, then we update the assignment vector $\mathbf{G}$ based on this switch. The detailed process is described in Algorithm 4.

The computational complexity of the K-L local refinement is $O(np^2)$, where $O(n)$ is the complexity of the loop and $O(p^2)$ is the complexity of the overall processing time calculation process.

The proposed load attraction, communication attraction algorithms, and their local refinement conduct the resource-based function mapping from different concerns, each of which has specific advantages for various applications. If (1) the balance between the computing time/power consumption on individual resources is more concerned, and (2) the variance of the computing capabilities of participating resources or the weights of components is more significant, then the load attraction mapping algorithm performs better than the communication attraction algorithm and other mapping approaches. If (1) the communication time/power consumption is more important, and (2) the variance of the transmission speed of connections or the weights of edges is more significant, then the communication attraction mapping algorithm is the better choice. If we want to minimize the overall processing time/power consumption, both algorithms with local refinements give similar mapping results, which are better than those generated by other mapping approaches. The selection of mapping algorithm depends on the specific requirements of different applications.

Note that the critical feature of these two resource-oriented mapping algorithms is to map sorted components and edges to limited resources according to resource capabilities. In this work, we directly use the computing and the transmission delays in the load and the communication attraction algorithms, respectively. We can also use other performance parameters or combinations according to specific requirements in different applications.

Input: $\mathbf{W_r}$, $\mathbf{W_c}$, $\mathbf{W_t}$, $\mathbf{W_e}$, assignment vector $\mathbf{G}$

Output: new assignment vector $\mathbf{G'}$, overall processing time t

initialization;

calculate the old overall processing time t_{all};

for *all components in* $\mathbf{G}$ **do**

  if *components* c_i *and* c_j *belong to different resources* r_a *and* r_b **then**

    calculate the overall processing time t'_{all} for $\{V_a + c_j - c_i, V_b + c_i - c_j\}$;

    if $t'_{all} < t_{all}$ **then**

      switch assignments of c_i and c_j;

      update t_{all} with t'_{all};

    end

  end

end

output $\mathbf{G'}$ and t_{all};

Algorithm 4: The K-L local refinement for the load and the communication attraction mapping algorithms.

3.5 Summary

In this chapter we have presented a new hierarchical operation-level function model, data dependency analysis, a computing resource model, and two function mapping algorithms. In the function model, a component clustering algorithm and the cyclic process modeling were proposed as well. These two approaches intend to provide a function model in appropriate granularity to be used in the mapping procedure. For function mapping, we proposed the load attraction and the communication attraction mapping algorithms that emphasize on minimizing the load variance and the communication, respectively on heterogeneous resources. Both algorithms may include the local refinement in order to minimize the overall processing time or power consumption. The effectiveness and performance comparisons of the proposed algorithms will be shown in Chapter 6. In the next chapter, we will apply some approaches developed in this chapter to various image processing IP designs on the virtual microsensor platform for visual sensor network applications.

Chapter 4

Virtual Micro-Sensor Platform

By using the modeling and partitioning methods presented in Chapter 3, we can improve the computation efficiency of various image processing algorithms with pipelined and parallel structures, and implement them directly on microsensor platforms. Some existing microsensors have been reviewed in Chapter 1. All of these microsensor platforms have significantly boosted the development of sensor networks, and inspired people to think what kind of applications can be deployed in such distributed processing environments. However, none of these microsensors can meet certain performance requirement without sacrificing others. For example, Mote [46] is ideal in size, but lack onboard processing capabilities. Sensoria sGate [42] is very powerful in both sensing and processing, however, its size and power consumption still need improvement.

In this chapter, we propose a virtual microsensor platform that provides a design environment with an image processing IP library. The IP library includes various pre-qualified image processing IPs in pipelined and parallel structures. With these developed IPs, user can select and quickly integrate the necessary IPs with predefined target design technologies, thereby achieving the fast application-specific microsensor design. In this chapter, we also present the implementation procedure from the virtual microsensor platform to the prototyping FPGAs and

the final SoC.

4.1 Virtual Microsensor Platform Structure

At the first glance, we expect microsensors to be as versatile as possible, and desire to integrate more functions and devices on one platform. However, the extra features added in might not be used for certain applications and the redundant functions and devices might slow down the overall system performance. Since microsensor is driven towards a high density, low power consumption, and high speed processing device, it is not meant to support dynamic reconfiguration. The conflict between the performance-constrained microsensor design and the desire of reconfiguration or reuse therefore pushes us to consider potential solution at the upper abstraction level, where the virtual microsensor platform is one choice.

Since VSNs are application-oriented, the microsensor design should be compact enough to just satisfy certain application's needs; and yet on the other hand, we would like the microsensor design to be flexible enough to be able to be deployed to different applications. In order to solve this conflict, we develop the virtual microsensor platform that provides various functions after synthesis but before the implementations on physical platforms. Therefore, the virtual microsensor platform can be versatile enough to incorporate any function, and the physical microsensor developed based on the virtual platform only integrates necessary functions according to specific applications. As the foundation of the virtual microsensor platform, the design reuse paradigm bridges the gap between design productivity and manufacturing capacity, leading to shorter time to market, higher performance/price ratio, and higher quality. Based on these advantages, the virtual microsensor platform supports easy plug and play of IP blocks in a seamless way to users.

According to the general architecture of microsensors, the virtual microsensor platform is composed of three sections: the sensing section, the digital computing section, and the communication section, as demonstrated in Fig. 4.1. Every section is expandable and contains various

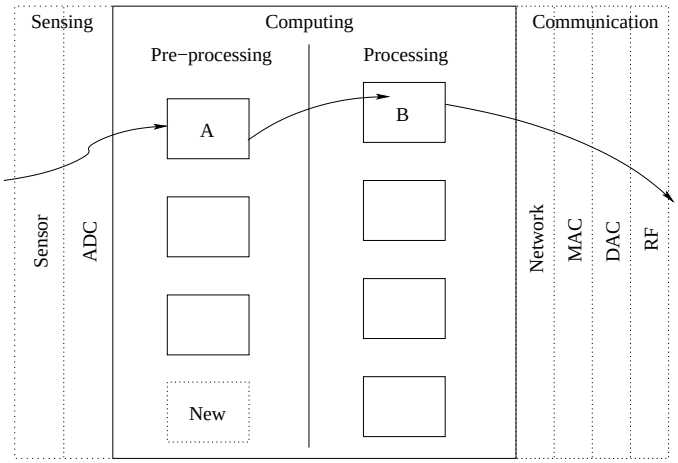


Figure 4.1: Virtual microsensor platform structure.

IPs. All IPs are synthesized and optimized using CAD tools, and ready to compose the entire microsensor design.

The sensing section handles variety of sensing modalities and the analog-digital conversion (ADC) circuit, which provides raw data of the monitored environment to the digital computing section. Since the virtual microsensor platform is developed specifically for VSNs, the obtained data set is supposed to be images. For other applications, we can easily expand the sensing section for 1-D data such as acoustic, seismic, and magnetic signals.

The digital computing section that mainly consists of the image processing IP library provides multiple processing IP blocks. From previous experience, the IP design of most applications is subject to a lot of constraints from application requirements and architecture limitations. For such application-oriented platforms, a promising approach is to drive the IP design at the algorithm level associated with the system integration constraints, that is, using the operation-level function model to partition processes of algorithms. At the same time, both temporal and functional application constraints and system I/O are also taken into account. Therefore, IP blocks in the IP library are evaluated in terms of extensive performance parameters including power consumption, execution delay/frequency, and utilization area. Users are able to select

appropriate IP blocks, A and B, for example, for specific applications regarding application constraints such as accuracy requirement, processing speed, and expense. In addition, since the image processing IP library is an expandable set, users may design, develop, and add new IP blocks to improve the virtual microsensor platform.

The communication section includes a communication IP library, the digital-analog conversion (DAC) circuit, and the RF communication circuits. The communication IP library contains various network protocol IPs of different layers. The data output from the network protocol IPs are sent to DAC, and then to the RF communication circuits that are responsible for sending and receiving signals.

The advantages of virtual microsensor platform are summarized as:

- Enabling the integration and simulation of new functional IP blocks in the existing environment.
- Allowing these IP blocks to be tested and evaluated on different VLSI.
- Admitting design reuse at any level of abstraction.

At any time a design obtained from the virtual microsensor platform can be easily tested on FPGAs for prototyping, and then converted to custom ASICs or potential SoCs.

In the sensing and the communication sections of the virtual microsensor platform architecture, analog and digital circuits coexist on a common substrate with the actual sensing and communication platform. It brings many challenges to the analog-digital mixed signal design and MEMS of SoC and System-on-Packing (SoP) integrations, which is out of the scope of this dissertation study. In the following discussions, we concentrate on the development of the image processing IP library that is the essential part of the digital processing section.

4.2 Image Processing IP Library

The virtual microsensor platform mostly relies on the image processing IP library that models commonly used image processing IPs. Obviously, it is impossible to prepare all kinds of design properties with possible functional and structural combinations in advance. The virtual microsensor platform has to be limited to a range of applications that share common features and requirements, such that the design cycle, quality and cost are predictable and comparable.

In this work, we target at a high-speed, low-energy image processing IP library, and develop IPs for the point-based processing, the neighborhood-based processing, and the image-based processing. In the area of image processing, algorithms that are performed on independent input images, i.e., every pixel output depends only on the current input pixel but no others, are referred to as the point-based image processing. In the neighborhood-based processing, every pixel output depends on both the current input pixel and its neighboring pixels. Similarly, every pixel output in the image-based processing depends on the entire input image. Although there are many algorithms within each category, the design flow is the same. Therefore, we only choose one or two examples in each category to develop image processing IPs, including the contrast stretching and the polynomial approximation-based geometric correction as examples of the point-based processing, the 3×3 filter as an example of the neighborhood-based processing, and the parallel ICA (pICA) algorithm as the image-based processing. In the next two sections, we will describe the development of these IPs in detail.

4.3 Algorithm Design and Improvement using Pipelined and Parallel Computing

During the algorithm implementation on VLSI, it is necessary to first design and improve the algorithm using pipelined and parallel structures. In the following discussion, we take one example to show how to use either pipelined or parallel structures to improve each category of

image processing algorithms we discussed in the previous section.

4.3.1 Contrast Stretching

Let us first take the contrast stretching algorithm [69] as an example of the point-based processing. Contrast stretching is a very popular image enhancement technique that increases the dynamic range of gray levels in the image being processed. It is especially effective to enhance the low-contrast images resulting from poor illumination or lack of dynamic range in the imaging sensor. The algorithm can be expressed in the following equation.

$$o(x,y) = f[i(x,y)] = m \times i(x,y) + b \tag{4.1}$$

where $i(x,y)$ and $o(x,y)$ are the gray level of the input and the output pixels, respectively; b and m denote the intercept and the slope of the transformation model, respectively.

In order to decide a pair of m and b, we draw the gray-level histogram that is a function showing the number of pixels at each gray level, and observe the intensity range in which most pixels fall. Two control points, $[i_1(x,y), o_1(x,y)]$ and $[i_2(x,y), o_2(x,y)]$ are generated, where $i_1(x,y)$ and $i_2(x,y)$ represent the observed intensity range, $o_1(x,y)$ and $o_2(x,y)$ represent the target range. By solving Eq. 4.1 with $[i_1(x,y), o_1(x,y)]$ and $[i_2(x,y), o_2(x,y)]$, m and b can be determined. If we want to enhance a specific range, i.e., apply the contrast stretching only to a subset of gray scale, a threshold is used to restrict pixels in the given range.

Contrast stretching has been an effective algorithm for various applications such as color restoration, where contrast stretching can be applied respectively to the red, green, and blue channels. During the processing, the gray-level histogram is used as the analysis tool for individual channels. It is a common method to find threshold and reveal the intensity distribution.

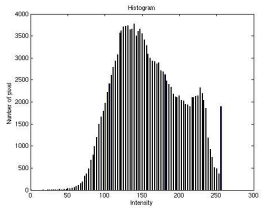
In order to demonstrate the effect of contrast stretching, we take a real image, shown in Fig. 4.2, as an example and conduct color restoration. Figure 4.3 respectively shows the original image at the red, the green, and the blue channel and the corresponding histograms.

Figure 4.2: The original image.

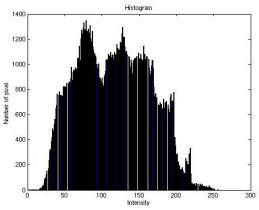
(a) Red channel.

(b) Green channel.

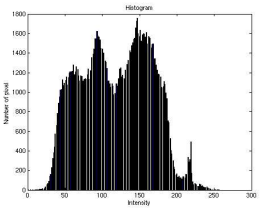
(c) Blue channel.



(d) Red channel histogram.



(e) Green channel histogram.



(f) Blue channel histogram.

Figure 4.3: Original image at red, green, blue channels and the corresponding histograms.

After analyzing the histograms of the three color channels, we select two control points for each channel, which are listed in Table 4.1. Then we respectively calculate the parameters m and b for the three channels, and apply contrast stretching in Eq. 4.1. The resulting image at the three channels and the corresponding histograms are shown in Fig. 4.4, respectively. Figure 4.5 shows the recovered image after color restoration using contrast stretching.

For contrast stretching, we evaluate three implementation architectures. According to the calculation of contrast stretching shown in Eq. 4.1, we can simply connect one multiplier to one adder and form a sequential processing as shown in Fig. 4.6, which we refer to as the traditional design. In the execution phase, the traditional design processes one input pixel in two clock cycles.

Although the traditional design is straightforward, it does not take full use of the available computing resources. We can stack several traditional designs and generate a parallel processing scheme. The number of the stacked traditional designs depends on the capability of resources. Let us set it to four as an example and demonstrate the 4-pixel parallel design in Fig. 4.7. Compared to the traditional design, the 4-pixel parallel design increases the processing speed by four times, i.e., four input pixels in two clock cycles. On the other hand, it also increases the utilization area by four times.

The traditional design and the 4-pixel parallel design are represented in the component clustering model as we proposed in the previous chapter. The communications in these two designs are minimized. However, the target implementation resource is VLSI where the communication may be ignorable. So we prefer the finer granularity at the operation level for the load balance purpose and decompose the contrast stretching processing into two individual operations, i.e., the multiplication and the addition. Figure 4.8 shows the 2-stage pipeline & 4-pixel parallel design. The change from the sequential processing to the pipelined processing increases the overall processing speed since different operations now execute temporally in parallel. It can also possibly increase the utilization area as well.

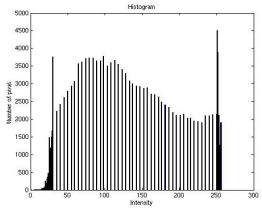
Table 4.1: Control point selection for the example image.

Channel	Control point 1	Control point 2
Red	(120, 30)	(225, 250)
Green	(70, 20)	(200, 225)
Blue	(70, 30)	(220, 225)

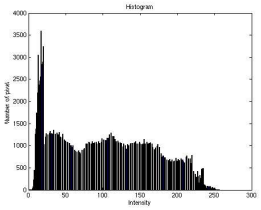
(a) Red channel.

(b) Green channel.

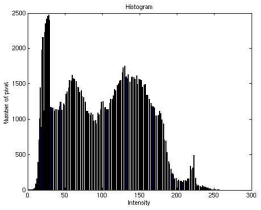
(c) Blue channel.



(d) Red channel histogram.



(e) Green channel histogram.



(f) Blue channel histogram.

Figure 4.4: Result image at red, green, blue channels and the corresponding histograms.

Figure 4.5: The result image of contrast stretching.

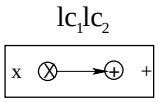


Figure 4.6: Traditional design.

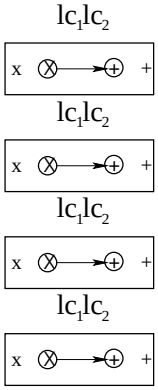


Figure 4.7: The 4-pixel parallel design.

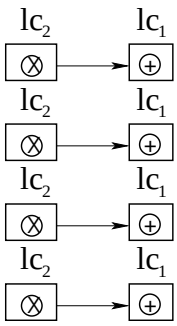


Figure 4.8: The 2-stage pipeline & 4-pixel parallel.

4.3.2 Polynomial Approximation-based Geometric Correction

Next, we take the geometric correction algorithm [116], which is more complex than the contrast stretching algorithm, as another example of the point-based processing. Most imaging systems display geometric distortions caused by various factors like motion of the imaging system, perspective projection of lenses, etc. According to the platform stability, we classify imaging systems into the *stable* system and the *unstable* system [57]. In the stable system, the characteristics of distortion remain the same, therefore the same correction model can be applied to all images taken from the same system. In unstable systems, parameters such as velocity, altitude, and orientation are constantly changing, resulting in a constant change of the characteristics of the distortion, thus render the dynamic derivation of the correction models necessary. This process is very time-consuming and affects the real-time performance required by certain applications such as automatic target detection on the spot.

Geometric correction is to restore the image from geometric distortions, which appear as bending of straight lines and especially at the corners of the image. The geometric correction removes such distortions by modifying the spatial relationships between pixels in the distorted image.

Typical solutions of geometric correction include *polynomial approximation* and *Thin-Plate Spline interpolation* [69]. In geometric correction, a transformation function is generally de-

signed to map the control points from a known pattern to their measured positions. As demonstrated in Fig. 4.9 [116], approximation methods generate transformations that map all of the control points close to their correspondence, such that the summation of displacements achieves a global minimum; whereas interpolation methods produce transformations where all the control points can be mapped to their correspondence exactly.

Suppose (u_i, v_i) is one control point from the corrected image, (x_i, y_i) is the corresponding point in the distorted image, P_x and P_y are the transformation functions for the $x-$ and $y-$ coordinates respectively, both of which are n^{th} degree polynomials. Then the transformation functions are expressed as

$$\hat{x} = P_x(u, v) = \sum_{i=0}^{n-1} \sum_{r+s=i} a_{irs} u^r v^s \tag{4.2}$$

$$\hat{y} = P_y(u, v) = \sum_{i=0}^{n-1} \sum_{r+s=i} b_{irs} u^r v^s \tag{4.3}$$

where $\hat{x}$ and $\hat{y}$ denote the approximated coordinates, a_{irs} and b_{irs} are the coefficients. The transformation functions should map (u, v)'s to (x, y)'s as closely as possible, in other words, the mean square error (MSE) defined below is minimized.

$$\epsilon = min_{a,b} \sum_{i=0}^{m-1} [(x_i - \hat{x_i})^2 + (y_i - \hat{y_i})^2] \tag{4.4}$$

where m is the number of control points.

Several solutions speed up the geometric correction process by partitioning the calculation into the software implementation and the hardware implementation. In 2002, Melis *et al.* [100] developed an FPGA-based reconfigurable computing platform, referred to as the SONIC, to implement the geometric correction process. The proposed platform consists of Xilinx Virtex XCV 1000E as processing core and synchronous SRAM as memory, where the interface to host system is a 64-bit PCI bus running at 64 MHz. In this implementation, the matrix inverse

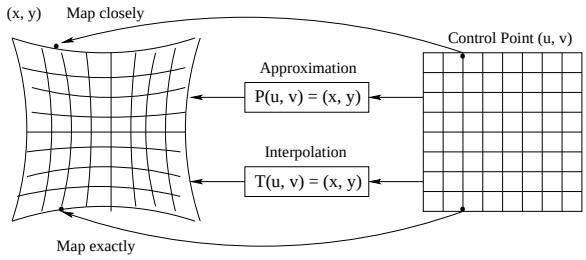


Figure 4.9: The transformation system between the distorted image and the corrected image.

operation is performed on the host computer using software tools. Most of the time taken by the estimation is in transferring the image data from the UltraSONIC PIPE to the host computer over the PCI bus. In the same year, Eadie *et al.* [60] also investigated the use of FPGAs to accelerate the execution of geometric correction using the third-order polynomial approximation. They used the Celoxica RC1000 development platform that includes a 2M gate Xilinx Virtex E FPGA, 8 MB SRAM, and PCI interface. They also left the matrix inverse calculation to the host computer.

In this work, we design a dynamic geometric correction IP based on the polynomial approximation and we use the 3^{rd} degree polynomial to model the transformations. Based on Eq. 4.2 and Eq. 4.3, we have

$$\begin{aligned}\hat{x} = a_{000} + a_{110}u + a_{101}v + a_{220}u^2 + a_{211}uv \\ +a_{202}v^2 + a_{330}u^3 + a_{321}u^2v + a_{312}uv^2 + a_{303}v^3\end{aligned} \tag{4.5}$$

and

$$\begin{aligned}\hat{y} = b_{000} + b_{110}u + b_{101}v + b_{220}u^2 + b_{211}uv \\ +b_{202}v^2 + b_{330}u^3 + b_{321}u^2v + b_{312}uv^2 + b_{303}v^3\end{aligned} \tag{4.6}$$

We then reformulate the transformations in matrix form. Let

$$\mathbf{A} = [a_{000}, a_{110}, a_{101}, \cdots, a_{303}]^T \tag{4.7}$$

$$\mathbf{B} = [b_{000}, b_{110}, b_{101}, \cdots, b_{303}]^T \tag{4.8}$$

$$\mathbf{X} = [x_0, \cdots, x_{m-1}]^T, \mathbf{Y} = [y_0, \cdots, y_{m-1}]^T \tag{4.9}$$

$$\hat{\mathbf{X}} = [\hat{x_0}, \cdots, \hat{x_{m-1}}]^T, \hat{\mathbf{Y}} = [\hat{y_0}, \cdots, \hat{y_{m-1}}]^T \tag{4.10}$$

$$\mathbf{W} = \begin{bmatrix} 1 & u_0 & \cdots & u_0 v_0^2 & v_0^3 \\ 1 & u_1 & \cdots & u_1 v_1^2 & v_1^3 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & u_{m-1} & \cdots & u_{m-1} v_{m-1}^2 & v_{m-1}^3 \end{bmatrix}$$

So we obtain

$$\hat{\mathbf{X}} = \mathbf{WA}, \hat{\mathbf{Y}} = \mathbf{WB} \tag{4.11}$$

and the objective function

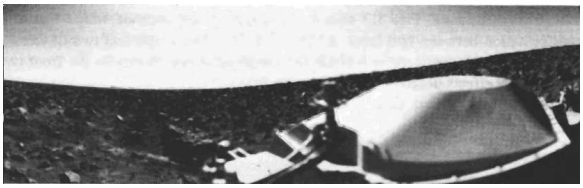
$$\epsilon = min_{\mathbf{A},\mathbf{B}}[(\mathbf{X} - \mathbf{WA})^T(\mathbf{X} - \mathbf{WA}) + (\mathbf{Y} - \mathbf{WB})^T(\mathbf{Y} - \mathbf{WB})] \tag{4.12}$$

Hence, the coefficient matrices are

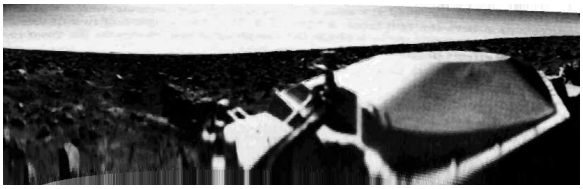
$$\mathbf{A} = \mathbf{W}^{-1}\mathbf{X}, \mathbf{B} = \mathbf{W}^{-1}\mathbf{Y} \tag{4.13}$$

Since $\mathbf{W}$ is not necessarily a square matrix depending on the number of control points, the pseudo-inverse is instead used, i.e., $\mathbf{W}^{-1} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T$.

Figure 4.10 shows an application example of the polynomial approximation-based geomet-



(a) The distorted image [9].



(b) The corrected image.

Figure 4.10: Example of the polynomial approximation-based geometric correction.

ric correction. In the distorted image [9], the exploration rover was landed on the surface of the moon, and the horizon is not straight but degraded. So we select the control points along the horizon and estimate the coefficient matrices. After applying the geometric correction, the restored or corrected image show less degradation.

According to the polynomial approximation described above, we design a pipelined processing structure for the dynamic geometric correction and illustrate it in Fig. 4.11. At the first step, the control points are saved in the internal RAM, and used to generate $\mathbf{W}$ that is expressed in Eq. 4.11. In order to obtain the pseudo-inverse ($\mathbf{W}^{-1} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T$), we conduct a series of matrix calculations, including matrix multiplication and matrix inverse. During these processes, all intermediate results required by further processing are saved in the internal RAM. After obtaining the pseudo-inverse $\mathbf{W}^{-1}$, the matrices $\mathbf{X}$ and $\mathbf{Y}$ are input to calculate $\mathbf{A}$ and $\mathbf{B}$ according to Eq. 4.13. Finally, we input $\mathbf{U}$ and $\mathbf{V}$ and obtain the output $\hat{\mathbf{X}}$ and $\hat{\mathbf{Y}}$. At this step, the intermediate results previously saved in RAM1, RAM2, RAM3, and RAM4 are not useful any longer. We can reuse these RAMs, therefore reducing the overall utilization area.

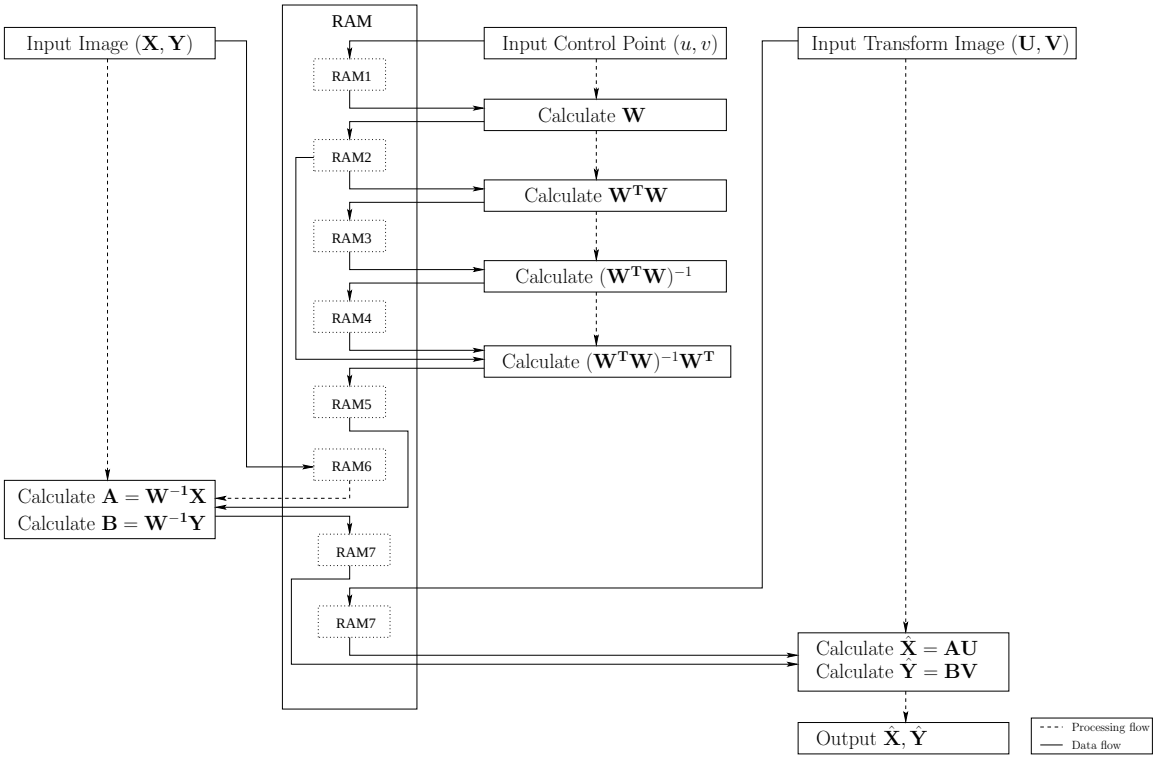


Figure 4.11: The pipelined design of polynomial approximation-based geometric correction.

From the structure shown in Fig. 4.11, we identify three function blocks including the formation of matrix $\mathbf{W}$, matrix multiplication, and matrix inverse. We first design the block that generates the initial matrix $\mathbf{W}$. In order to reduce the utilization area, we use a pipelined processing instead of the parallel processing structure in this block. For each control point (u, v), it takes 10 clock cycles to generate one row in matrix $\mathbf{W}$. A state machine is used to control the processing flow. We also include the delay verifications after reading in and before writing out data. The purpose is to make sure the I/O ports are not delayed by other blocks or interconnections. If the polynomial degree is greater than three, a rounder is suggested to be included.

Secondly, we design the matrix multiplication block. Technically speaking, the design of the matrix multiplication can be similar to that of the 3×3 filter. However, the size of the matrix in the dynamic geometric correction is not a constant and is larger than 3×3, which makes the parallel structure inappropriate for the matrix multiplication. So we design a pipelined structure as shown in Fig. 4.12. The manipulation block reads in two matrices row by row (or column by column) from RAMs, and sends the corresponding elements to the multiplication operation. The processing flow in the multiplication block is demonstrated in Fig. 4.13. The sign verification is first executed to decide the sign of the result. The state machine then controls the accumulation, the saving of intermediate results, and the output of the final result.

The third and the most complex function block is the matrix inverse. As the structure shown in Fig. 4.14, we use the LU factorization [52] to avoid large dimensional matrix operations that require many clock cycles and result in low processing speed. In the LU factorization, the

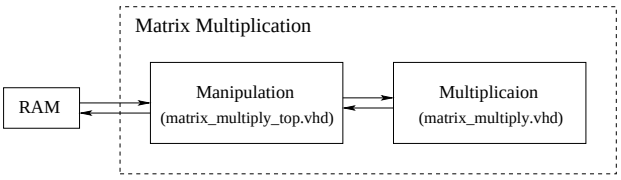


Figure 4.12: Structure of matrix multiplication.

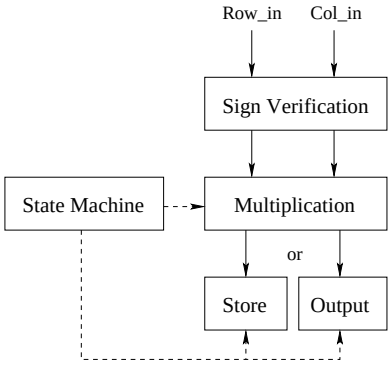


Figure 4.13: Processing flow of matrix multiplication.

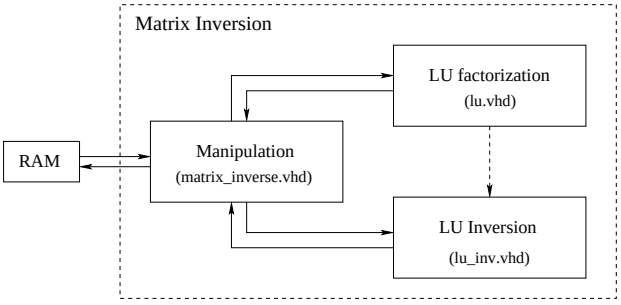


Figure 4.14: Structure of matrix inverse.

input matrix $\mathbf{A}$ is decomposed into

$$\mathbf{A} = \mathbf{LU} \tag{4.14}$$

where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix. Hence, the inverse matrix is

$$\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1} \tag{4.15}$$

where the element is expressed by

$$\mathbf{A}_{ij} = \sum_{k=1}^{min(i,j)} \mathbf{L}_{ik}\mathbf{U}_{kj} \tag{4.16}$$

The elements in matrix $\mathbf{L}$ and $\mathbf{U}$ are defined as

$$\mathbf{L}_{ii} = 1 \tag{4.17}$$

$$\mathbf{L}_{ij} = \mathbf{U}_{jj}^{-1}(\mathbf{A}_{ij} - \sum_{k=1}^{j-1} \mathbf{L}_{ik}\mathbf{U}_{kj}), j \leq i-1 \tag{4.18}$$

$$\mathbf{U}_{ij} = \mathbf{A}_{ij} - \sum_{k=1}^{j-1} \mathbf{L}_{ik}\mathbf{U}_{kj}, j \leq i \tag{4.19}$$

The processing flow of the LU factorization is demonstrated in Fig. 4.15. Since both the upper triangular elements of $\mathbf{L}$ and the lower triangular elements of $\mathbf{U}$ are 0's, and the diagonal elements of $\mathbf{L}$ are 1's, we store $\mathbf{L}$ and $\mathbf{U}$ in one matrix so as to save the RAM space. In this function block, we use several long vectors to express matrix because most synthesis tools do not support matrix directly. In addition, we decompose the time-consuming processing into several small parts to decrease the interval of clock cycle and avoid delay.

In the next step, we calculate the inverse of LU matrix. Each element of the upper triangular matrix is computed by

$$a'_{ij} = -a_{jj}^{-1} \sum_{k=i}^{j-1} a'_{ik}a_{kj} \tag{4.20}$$

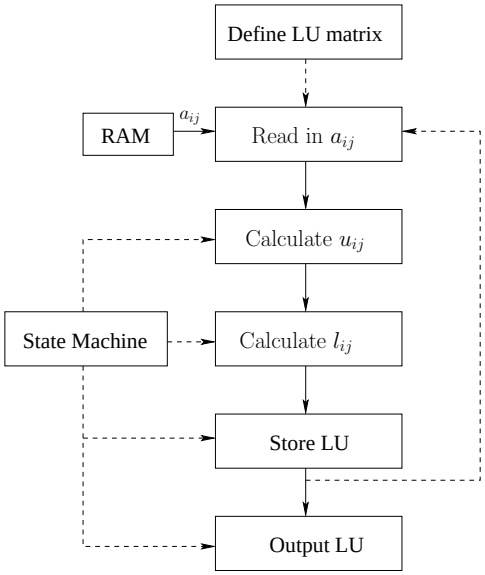


Figure 4.15: Processing flow of LU factorization.

where a_{ij} is the element in the original LU matrix, and a'_{ij} is the element in the inverse matrix. Each element of the lower triangular matrix is calculated by

$$a'_{ij} = -a_{ii}^{-1} \sum_{k=1}^{i-1} a_{ik} a'_{kj} \tag{4.21}$$

where $a_{ii} = 1$ in $\mathbf{L}$.

The processing flow of the LU inverse is demonstrated in Fig. 4.16. In this block, the dependency of elements decides the input sequence. A rounder is also included to avoid overflow.

As the overall design structure of the polynomial approximation illustrated in Fig. 4.17, all the function blocks are coordinated by a top level block, which also serves as an interface between the internal processing and the external I/O. The original image and the parameters that specify the bitwidth and the matrix dimensions are input to the top level block, which then forwards these data to internal RAMs and function blocks, respectively. The processing time flow is controlled by a state machine. The clock controller is used to control clock pulse of

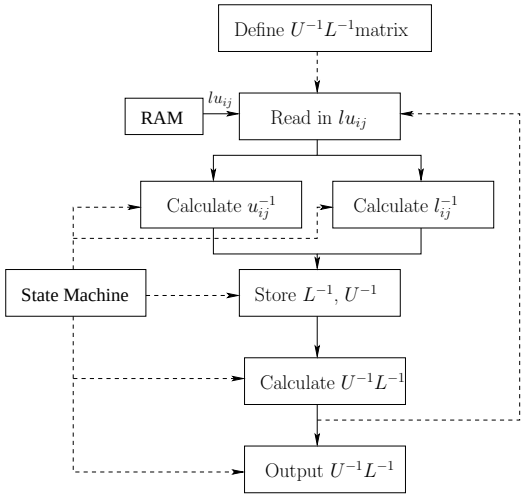


Figure 4.16: Processing flow of LU inverse.

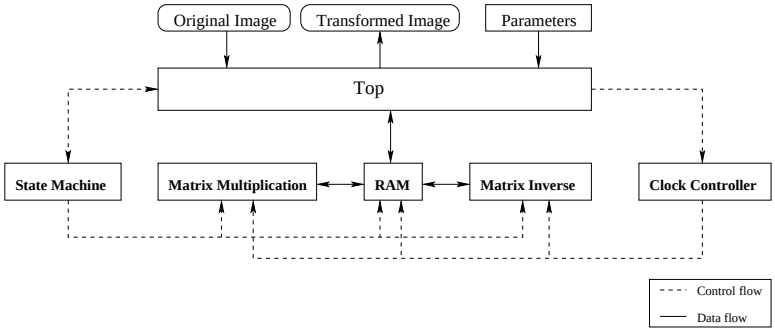


Figure 4.17: Structure of the polynomial approximation.

individual function blocks. If there is a need for a specific function, the clock controller wakes up the corresponding block and forwards the system clock to it; otherwise, all function blocks remain at the sleep state in order to save power. When all processes are finished, the top level obtains the final transformed image from RAM and output it.

4.3.3 3×3 Filters

In the image processing IP library, we develop designs for the 3×3 spatial filter [69] as an example of the neighborhood-based processing. The 3×3 mask-based filter is a general format in spatial domain for many neighborhood-based image processing algorithms. Examples include smoothing filters (or lowpass spatial filters) such as the Gaussian lowpass filter, sharping filters (or highpass spatial filters) such as the Laplacian highpass filter. Highpass filters like the Prewitt filter and the Sobel filter are popularly used in edge detection operations.

Although the spatial filters may be defined by a 3×3 matrix or other size matrices, the 3×3 filter is more commonly used. The general format of a 3×3 filter is $h = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$. Then the calculation of the output pixel $o(x, y)$ is expressed as:

$$\begin{aligned} o(x,y) = i(x-1,y-1) \times h_{11} + i(x-1,y) \times h_{12} + i(x-1,y+1) \times \\ h_{13} + i(x,y-1) \times h_{21} + i(x,y) \times h_{22} + i(x,y+1) \times h_{23} + \\ i(x+1,y-1) \times h_{31} + i(x+1,y) \times h_{32} + i(x+1,y+1) \times h_{33} \end{aligned} \tag{4.22}$$

where $i(x, y)$ is the input pixel.

Some filters are assigned a coefficient ($Koef$) [22] that serves as a normalization coefficient to keep the gray levels of pixels from going beyond the initial range after the transformation. Some filters are also assigned the $Bias$ that increases the gray levels of pixels so that dark pixels

appear brighter. The following equation expresses the complete transformation.

$$o(x,y) = \frac{o(x,y)}{Koef} + Bias \tag{4.23}$$

The example filters defined in 3×3 format are listed in Table 4.2, where Z is the maximum intensity level of the image. Figure 4.18 shows some examples of the 3×3 filters listed in Table 4.2.

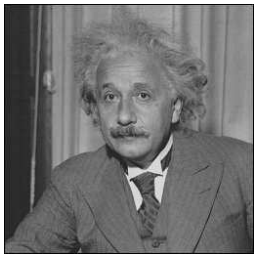
According to the features of the 3×3 filters we develop the traditional design, the parallel & pipelined design, and the parallel & pipelined design with partitioning. The traditional design is shown in Fig. 4.19. The nine multiplication operations between the input pixels and the corresponding component in the 3×3 filter are conducted in parallel. The results of multiplications are sent to the addition operation that includes eight adders in sequence.

The parallel & pipelined design, as shown in Fig. 4.20, decomposes the sequential addition into eight individual addition operations, and forms the parallel & pipelined structure. From the horizontal view, this design is in a binary tree structure where individual branches executes in parallel. From the vertical view, this design is in a pipelined structure where each level in the parallel structure is one stage in the pipelined structure. The first stage consists of nine multiplication operations in parallel. The second and the third stages respectively include four and two addition operations in parallel. Each of the fourth and the fifth stages contains one addition, and the fifth stage outputs the final result. Since the rightmost multiplier is not involved in any computing from the second to the fourth stages, a 3-stage buffer is added between the multiplier and the adder at the fifth stage for synchronization purpose.

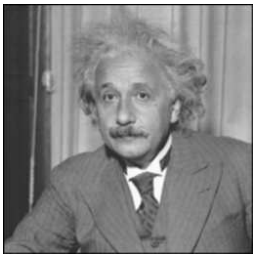
The parallel & pipelined design is further improved by the partitioning on the parallel structure. The function model we use for partitioning is improved by the component clustering algorithm described in Sec. 3.1.2, as shown in Fig. 3.10. We first conduct the function partitioning. Since the design will be synthesized on FPGA for prototyping, we assume the computing resource is homogeneous and randomly set the number of partitions to 5, i.e., $k = 5$. After

Table 4.2: 3×3 Filters.

Filter	Kernel	Koef & Bias
Laplacian lowpass filter	1 1 1 1 1 1 1 1 1	$Koef = 9$
Gaussian lowpass filter	1 2 1 2 4 2 1 2 1	$Koef = 16$
Laplacian highpass filter	0 1 0 1 -4 1 0 1 0	
Prewitt filter (horizontal)	-1 -1 -1 0 0 0 1 1 1	
Prewitt filter (vertical)	-1 0 1 -1 0 1 -1 0 1	
Sobel filter (horizontal)	-1 -2 -1 0 0 0 1 2 1	
Sobel filter (vertical)	-1 0 1 -2 0 2 -1 0 1	
Emboss filter	-1 0 0 0 0 0 0 0 1	$Bias = \frac{Z}{2}$
Enhanced focus	-1 0 -1 0 7 0 -1 0 -1	$Koef = 3$
Blur light	1 2 1 2 2 2 1 2 1	$Koef = 14$

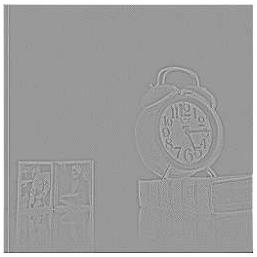


(a) Original image.



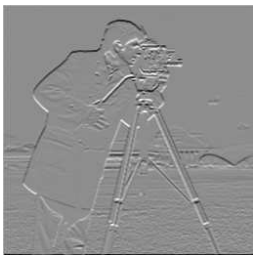
(b) Result of Gaussian lowpass filter.

(c) Original image.



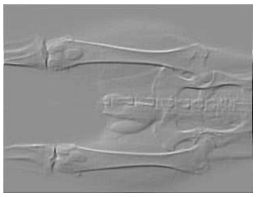
(d) Result of Laplacian highpass filter.

(e) Original image.



(f) Result of Sobel (horizontal) filter.

(g) Original image.



(h) Result of emboss filter.

Figure 4.18: Effects of applying 3×3 filters.

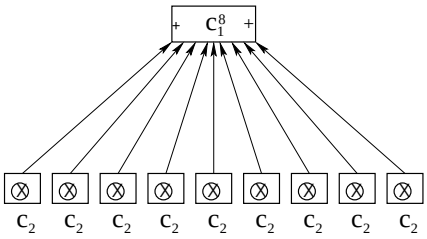


Figure 4.19: Traditional design.

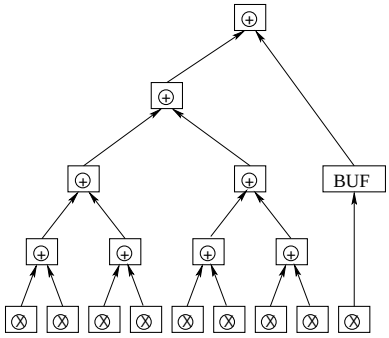


Figure 4.20: The parallel & pipelined design.

applying the scattered algorithm to the function model, we visualize the partitioning result in Fig. 4.21, where r_i and T_{ij} represent a participating computing and a communication resource, respectively. $w(P_{ij})$ denotes the cut weight between partitions V_i and V_j. The partition V_1 contains two multipliers and one adder, which sends one data item to the partition V_2 in every clock cycle, i.e., $w(P_{12}) = 1$. The partition V_2 contains two multipliers and two adders, and sends one data item to the partition V_3 in every clock cycle, i.e., $w(P_{23}) = 1$. Similarly, the partition V_4 and V_5 respectively includes two multipliers and two adders, two multipliers and one adder, and $w(P_{34}) = w(P_{45}) = 1$. The partition V_5 includes one multiplier, two adders, and one buffer.

After the function partitioning, we conduct the data distribution as we analyzed in Chapter 3. Since the multipliers that require the input data are distributed to five partitions, we send pixels of every 3×3 subset from the input image to five partitions. As shown in Fig. 4.21, the data set is partitioned along the horizontal and vertical directions where the dependency edges

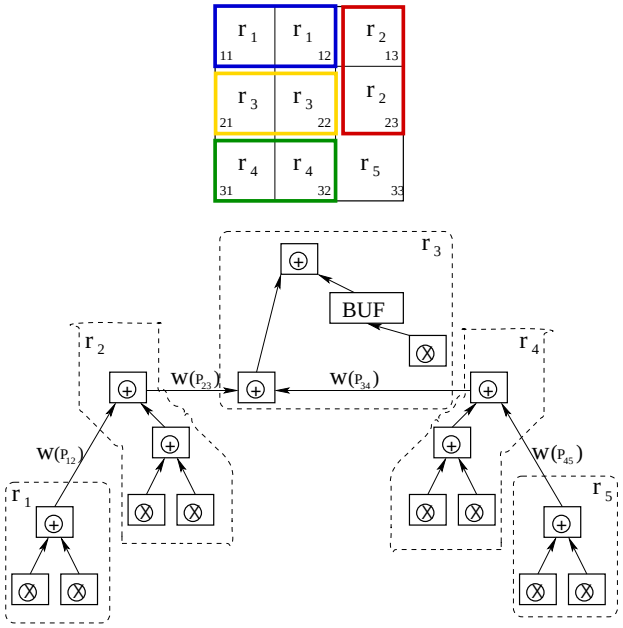


Figure 4.21: The parallel & pipelined design with partitioning.

between data have the minimum weights, and the corresponding partition is marked on each data block.

4.3.4 Parallel Independent Component Analysis

During the development of the image processing IP library, we take the pICA algorithm as an example of the image-based processing. Independent component analysis (ICA) is a method that searches for a linear or nonlinear non-orthogonal coordinate system in any multivariate data, in which the directions of the axes are determined by both the second and higher order statistics of the original data [41, 94]. By performing transformations through this system, the statistically independent source signals are extracted from the original data. As an unsupervised separation technique, ICA plays an important role in a variety of signal and image processing applications such as blind source separation (BSS) [75, 17, 74, 7], face recognition [15], hyperspectral image (HSI) analysis [55], and so forth. Although powerful, many ICA algorithms involve complex

computations and normally experience slow convergence rate in real-time image processing applications. In order to speed up the ICA process, various approaches have been pursued to speed up ICA algorithms. These approaches can be classified into two categories: hardware implementation and algorithm improvement.

Many solutions have been presented on hardware implementations that use either analog CMOS and analog-digital mixed signal VLSIs, digital ASICs, or FPGAs with millions of transistors.

Designs based on analog or analog-digital mixed technologies utilize the silicon in the most efficient manner. For example, analog CMOS chips have been designed to implement a simple ICA-based blind separation of mixed speech signals [40] and an infomax theory-based ICA algorithm [38]. Celik *et al.* [33] used a mixed-signal adaptive parallel VLSI architecture to implement the Herault-Jutten (H-J) ICA algorithm. The coefficients in the unmixing matrix were stored in digital cells of the architecture, which was fabricated on a 3mm $\times$ 3mm chip using a 0.5μm CMOS technology. But the 3$\times$3 chip could only unmix three independent components. The neuromorphic auto-adaptive systems project conducted at Johns Hopkins University [32] used the ICA VLSI processor as a front-end of the system integration. The processor separates the mixed analog acoustic inputs and feeds the digital output to Xilinx FPGA for classification purpose.

ASICs possess many advantages such as high circuit density and efficiency, low power consumption, and short design period. In 2003, the Computational NeuroSystems Laboratory of Korea Advanced Institute of Science and Technology [92] designed an ASIC chip for the InfoMax-based ICA algorithms and used it as a front-end to control noise in speech recognition.

FPGAs are the best selections for fast design implementations and allow end users to modify and configure their designs for multiple times. In 2001, Lim *et al.* [97] respectively implemented two small 7-neuron independent component neural network (ICNN) prototypes on Xilinx Virtex XCV 812E which contains 0.25 million logic gates. The prototypes are based

on mutual information maximization and output divergence minimization. Nordin *et al.* [105] proposed a pipelined ICA architecture for potential FPGA implementation. Since each block in the 4-stage pipelined FPGA array did not have data dependency with others, all blocks could be implemented and executed in parallel. In 2002, Satter and Charayaphan [123] implemented an ICA-based BSS algorithm on Xilinx Virtex E which contains 0.6 million logic gates. Due to the capacity limit, the maximum iteration number was pre-limited to 50 and the buffer size to 2,500 samples. Wei and Charoensak [143] implemented a non-iterative algebra ICA algorithm [140] that requires neither iteration nor assumption on Xilinx Virtex E in order to speed up the motion detection in image sequences. Although the design only used 90,200 of the 600,000 logic gates, the system could support the unmixing of only two independent components.

Approaches in algorithm improvement category can be further divided into two classes: including nonlinear learning parameters in the ICA estimation process, or combining the ICA process with other pre-processing methods. In many cases, using the nonlinear or adaptive learning parameter is computationally efficient in the improvement of the convergence speed. In 2000, for example, Matsuyama *et al.* [102] presented an α-ICA algorithm that used the α-logarithm to speedup the convergence of ICA process. In 2003, Lou and Zhang [98] introduced the so-called fuzzy inference system (FIS) to the ICA-based neural network. In parallel to adding nonlinear learning parameters, combining other pre-processing methods to the ICA process provides another option for convergence acceleration. The Spectral Screening ICA (SSICA) [119] and the Single-Input Multiple-Output based ICA (SIMO-ICA) [122] are good candidates in this branch.

However, none of the previously described approaches takes advantage of the SIMD parallelism. In this work, we seek a data parallel solution in SIMD and present a parallel ICA (pICA) algorithm based on the FastICA approach [75], which is one of the most efficient and practical ICA algorithms developed so far. FastICA involves two sequential processes, the one unit (weight vector) estimation and the decorrelation among weight vectors. In pICA, the process

of weight matrix estimation is divided into sub-processes that can be conducted on multiple computing resources in parallel. The decorrelation process consists of both internal decorrelation and external decorrelation, which respectively performs decorrelation of weight vectors generated within the same computing resource and between different computing resources. In order to measure the performance of the proposed pICA algorithm, a performance prediction model is set up based on the LogP model. In the rest of this section, we will briefly introduce the ICA and the FastICA algorithm, present the structure and the derivation of the parallel ICA algorithm, describe the performance prediction model for pICA, and develop the IP design for the pICA.

Independent Component Analysis

In many applications, the observed signals can be modeled as the linear/nonlinear mixtures of the source signals. For example, in the cocktail party problem, the acoustic signals captured from any microphone (observed signal) is a mixture of individual speakers (source signal) speaking at the same time. Assume the source signals are statistically independent and no more than one signal is Gaussian distributed, that is, no source signals gives information on one another, ICA can then be used to find an optimal transformation that unmixes the observed signals to source signals with minimal second and higher order statistical dependences between each other. Let $\mathbf{x}_1, \cdots, \mathbf{x}_n$ be the n observed signals, and $\mathbf{s}_1, \cdots, \mathbf{s}_m$ be the m source signals, the ICA unmixing model defined as $\mathbf{s} = \mathbf{W}\mathbf{x}$ unmixes the observed signal $\mathbf{x}$ by an $m \times n$ unmixing matrix or weight matrix $\mathbf{W}$ to the source signal $\mathbf{s}$.

In ICA, since the source signals $\mathbf{s}_j$, $j = 1, \cdots, m$ are desired to contain the least Gaussian components, the measure of nongaussianity is therefore the key to estimating the weight matrix and correspondingly the independent components. Hence, the definition and estimation of an objective function which measures the nongaussianity of independent components is necessary for the identifiability of the model. Since Gaussian variable has the largest entropy among all

random variables of equal variance [44], an approximation of negentropy (Eq. 4.24) is usually given [75] as the objective function.

$$J(\mathbf{Y}) \approx \{E[G(\mathbf{Y})] - E[G(\mathbf{Y}_{gauss})]\}^2 \tag{4.24}$$

where $\mathbf{Y}$ is a random variable, $G(\mathbf{Y})$ is a non-quadratic function, and $G(\mathbf{Y}_{gauss})$ is the entropy of a Gaussian random variable with the same covariance matrix as $\mathbf{Y}$. The choice of $G(u) = \frac{1}{a}\log\cosh(au)$, where $1 \leq a \leq 2$ and often $a = 1$, has been proved [75] useful.

To find $\mathbf{W}$ that maximizes the objective function of Eq. 4.24, Hyvärinen [75] developed the FastICA algorithm that involves the processes of *one unit estimation* and *decorrelation*. In this algorithm, $\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^T & \cdots & \mathbf{w}_i^T & \cdots & \mathbf{w}_m^T \end{bmatrix}^T$, where $\mathbf{w}_i$ is an $n \times 1$ weight vector. The one unit process estimates the weight vectors $\mathbf{w}_i$ with Eqs. 4.25 and 4.26,

$$\mathbf{w}_i^+ = E\{\mathbf{x}g(\mathbf{w}_i^T\mathbf{x})\} - E\{g'(\mathbf{w}_i^T\mathbf{x})\}\mathbf{w}_i \tag{4.25}$$

$$\mathbf{w}_i = \mathbf{w}_i^+ / ||\mathbf{w}_i^+|| \tag{4.26}$$

where g denotes the derivative of the non-quadratic function G, and $g(u) = \tanh(au)$, g' denotes the derivative of g, and $g'(u) = sech^2(u)$.

The purpose of the decorrelation process is to keep different weight vectors from converging to the same maximum. For example, the $(p+1)^{th}$ weight vector is decorrelated from the preceding p weight vectors by Eqs. 4.27 and 4.28.

$$\mathbf{w}_{p+1}^+ = \mathbf{w}_{p+1} - \sum_{i=1}^{p,p\leq m-1} \mathbf{w}_{p+1}^T\mathbf{w}_i\mathbf{w}_i \tag{4.27}$$

$$\mathbf{w}_{p+1} = \mathbf{w}_{p+1}^+ / ||\mathbf{w}_{p+1}^+|| \tag{4.28}$$

The Parallel ICA Algorithm

Due to the computation complexity and slow convergence rate, the implementations of many ICA algorithms, even FastICA, is very time-consuming. In order to speed up the FastICA execution, we seek a data parallel solution in SIMD parallelism and propose a pICA algorithm based on FastICA.

In SIMD, same processes are executed simultaneously on multiple data items instead of being synchronized at individual operation level. The principle of pICA is to conduct the weight matrix estimation process using several parallelly performed sub-processes by equally dividing the weight matrix $\mathbf{W}$ into k sub-matrices, $\mathbf{W} = \begin{bmatrix} \mathbf{W}_1 & \cdots & \mathbf{W}_z & \cdots & \mathbf{W}_k \end{bmatrix}^T$, where $\mathbf{W}_z = \begin{bmatrix} \mathbf{w}_{z,1}^T & \cdots & \mathbf{w}_{z,i}^T & \cdots & \mathbf{w}_{z,m_z}^T \end{bmatrix}^T$, m_z is the number of weight vectors in $\mathbf{W}_z$, and the total number of weight vectors $m = m_1 + \cdots + m_z + \cdots + m_k$. The division can also depend on the computing speed of a certain computing resource. Each sub-process estimates a sub-matrix $\mathbf{W}_z$, $z = 1, \cdots, k$ by a one-unit process and an *internal decorrelation*. The internal decorrelation decorrelates weight vectors within the same sub-matrix and is conducted by Eqs. 4.29 and 4.30.

$$\mathbf{w}_{z(p+1)}^+ = \mathbf{w}_{z(p+1)} - \sum_{j=1}^{p,p\leq m_z-1} \mathbf{w}_{z(p+1)}^T \mathbf{w}_{zj} \mathbf{w}_{zj} \tag{4.29}$$

$$\mathbf{w}_{z(p+1)} = \mathbf{w}_{z(p+1)}^+ / ||\mathbf{w}_{z(p+1)}^+|| \tag{4.30}$$

where $\mathbf{w}_{z(p+1)}$ denotes the $(p+1)^{th}$ weight vector in the z^{th} sub-matrix.

Although the internal decorrelation process keeps different weight vectors within the same sub-matrix from converging to the same maximum, two weight vectors generated from different sub-matrices could still correlate with each other. Therefore, an *external decorrelation* is needed to decorrelate weight vectors generated from different sub-processes. We develop the following theorem for performing external decorrelation such that the parallel processing of sub-matrices,

or the integration of internal and external decorrelations, does not affect the accuracy of ICA.

Theorem 1. *Assume $\mathbf{w}_{z(q+1)}$ is a weight vector of the sub-matrix $\mathbf{W}_z$ and $\mathbf{w}_j$ is a weight vector from another sub-matrix, then the external decorrelation is conducted following Eqs. 4.31 and 4.32.*

$$\mathbf{w}^+_{z(q+1)} = \mathbf{w}_{z(q+1)} - \sum_{j=1}^{q,q\leq(m-m_z-1)} \mathbf{w}^T_{z(q+1)}\mathbf{w}_j\mathbf{w}_j \tag{4.31}$$

$$\mathbf{w}_{z(q+1)} = \mathbf{w}^+_{z(q+1)}/||\mathbf{w}^+_{z(q+1)}|| \tag{4.32}$$

Proof: Let us first consider the two sub-matrix problem. Assume the weight matrix $\mathbf{W}$ is divided into two sub-matrices, $\mathbf{W}_z$ and $\mathbf{W}_j$ of dimension $m_z \times n$ and $m_j \times n$ respectively, $m = m_z + m_j$. Without loss of generality, we assume $\mathbf{W}_j$ is prior to $\mathbf{W}_z$. Then we decorrelate the $(p+1)^{th}$ weight vector $\mathbf{w}_{z(p+1)}$ in $\mathbf{W}_z$ with weight vectors from $\mathbf{W}_j$ using Eq. 4.33.

$$\mathbf{w}^+_{z(p+1)} = \mathbf{w}_{z(p+1)} - \sum_{v=1}^{m_j} \mathbf{w}^T_{z(p+1)}\mathbf{w}_{jv}\mathbf{w}_{jv} \tag{4.33}$$

where $\mathbf{w}_{jv}$ denotes the v^{th} weight vector in $\mathbf{W}_j$, and $v = 1, \cdots, m_j$.

The weight vector $\mathbf{w}_{z(p+1)}$ also needs to go through the internal decorrelation process, defined as Eq. 4.29,

$$\mathbf{w}^+_{z(p+1)} = \mathbf{w}_{z(p+1)} - \sum_{v=1}^{p,p\leq m_z-1} \mathbf{w}^T_{z(p+1)}\mathbf{w}_{zv}\mathbf{w}_{zv} \tag{4.34}$$

where $p = 1, \cdots, m_z - 1$.

After both the external decorrelation (Eq. 4.33) and the internal decorrelation (Eq. 4.34), we get

$$\mathbf{w}^+_{z(p+1)} = \mathbf{w}_{z(p+1)} - \sum_{v=1}^{p,p\leq m_z-1} \mathbf{w}^T_{z(p+1)}\mathbf{w}_{zv}\mathbf{w}_{zv} - \sum_{v=1}^{m_j} \mathbf{w}^T_{z(p+1)}\mathbf{w}_{jv}\mathbf{w}_{jv} \tag{4.35}$$

where the second component comes from the internal decorrelation and the third component

from the external decorrelation. That is,

$$\mathbf{w}^{+}_{z(p+1)} = \mathbf{w}_{z(p+1)} - \sum_{v=1}^{p+m_j, p \leq m_z - 1} \mathbf{w}^{T}_{z(p+1)} \mathbf{w}_v \mathbf{w}_v \tag{4.36}$$

where the p weight vectors are from the same sub-matrix $\mathbf{W}_z$, and the m_j weight vectors come from the other sub-matrix $\mathbf{W}_j$.

Comparing Eq. 4.36 with Eq. 4.27, we draw the conclusion that using the external and internal decorrelation, each weight vector in the two sub-matrices can be decorrelated as if it is computed within one matrix.

For decorrelation of more than two sub-matrices, the proof is similar. □

The structure of pICA is illustrated in Fig. 4.22, where the process of pICA is organized as a binary tree. All the internal decorrelation of sub-matrices are performed on separate computing resources in parallel. This layer serves as the "leave" of the binary tree. Then the results generated from a pair of computing resources will go through the external decorrelation. This process propagates until the final two results are decorrelated at the "root" of the binary tree. If there are k computing resources, that is, k sub-matrices, then the number of layers of the binary tree is $c = 1 + \lceil log_2 k \rceil$.

We observe that in the above binary tree construction, both the internal and the external decorrelations can be run in parallel mode, therefore, the computation burden is distributed from single process to multiple sub-processes in parallel.

Performance Modeling and Analysis

Next, we present a pICA performance prediction model with consideration of the parallel computing environment [56]. Since the proposed pICA algorithm utilizes SIMD structure for the purpose of processing high volume data sets, the LogP model [47] is a suitable prototype parallel computing model that takes into account both the computer architecture and the application

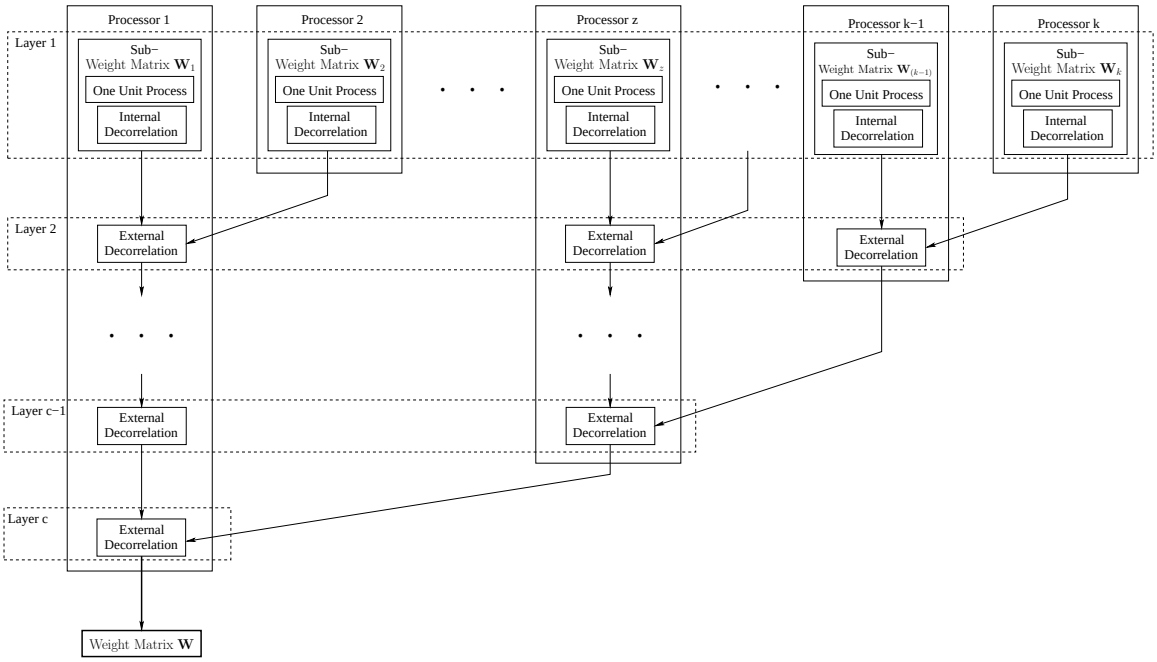


Figure 4.22: Structure of the pICA algorithm.

architecture. The computational resource is mostly affected by communication network and computing node. The application architecture is generally described by $T_{app} = T_{comp} + T_{comm}$.

During the performance modeling and analysis procedure, we use the multi-processor environment as the testbed. Given a pICA process running on p processors, the number of layers is $c = 1 + \lceil log_2 p \rceil$. The overall execution time $T_{overall} = \sum_{z=1}^{c} T_z$, where at each layer z, the execution time is determined by the slowest process. At the first layer, we transfer the observed data to each processor, then perform the one unit (weight vector) estimation and the internal decorrelation on individual sub-matrices. The "superstep" that includes computation, communication, and synchronization at this layer is written as

$$T_1^{Superstep} = (t_{oneunit}^{Superstep} + t_{id}^{Superstep}) + (L + 2 \times o^{Superstep}) \times (h_{data}^{Superstep} + h_{wv}^{Superstep}) + l_1^{Superstep} \tag{4.37}$$

where $t_{oneunit}^{Superstep}$ and $t_{id}^{Superstep}$ denote the processing time for the one-unit weight vector esti-

mations and internal decorrelations respectively, $h_{data}^{Superstep}$ and $h_{wv}^{Superstep}$ are the observation data and weight vectors being transfered, $l_1^{Superstep}$ is the synchronization time on individual processors at this layer. Since the execution time is determined by the slowest processor, the synchronization time is zero. The overall execution time at layer 1, T_1, is then defined as

$$T_1 = (t_{oneunit} + t_{id}) + (L + 2 \times o) \times (h_{data} + h_{wv}) \tag{4.38}$$

where $t_{oneunit}$ and t_{id} are the slowest processing time among all processors for the one-unit weight vector estimations and internal decorrelations.

At other layers, we only conduct external decorrelations. So the superstep and the execution time of these layers are written as

$$T_\alpha^{Superstep} = t_{ed}^{Superstep} + (L + 2 \times o^{Superstep}) \times h_{wv}^{Superstep} + l_\alpha^{Superstep} \tag{4.39}$$

$$T_\alpha = t_{ed} + (L + 2 \times o) \times h_{wv} \tag{4.40}$$

where $\alpha = 2, \cdots, c$, and $\mathbf{t}_{ed}$ denotes the slowest processing time for external decorrelations.

Figure 4.23 illustrates the computing and communication flow of the pICA process on ten processors. Since it is impossible to reduce the amount of observation data h_{data} in the communication, we then use the pipelined structure to speed up data transfer and balance the network traffic. The master processor first sends a portion of the observation data to one slave processor, which in turn forwards the data it receives to the next slave processor. When conducting the external decorrelation at upper layers, we use a parallel external decorrelation that hierarchically decorrelates weight vectors from sub-matrices and sends the result back to the master processor. This performance prediction model for the pICA algorithm will be validated in the next chapter by experimental study.

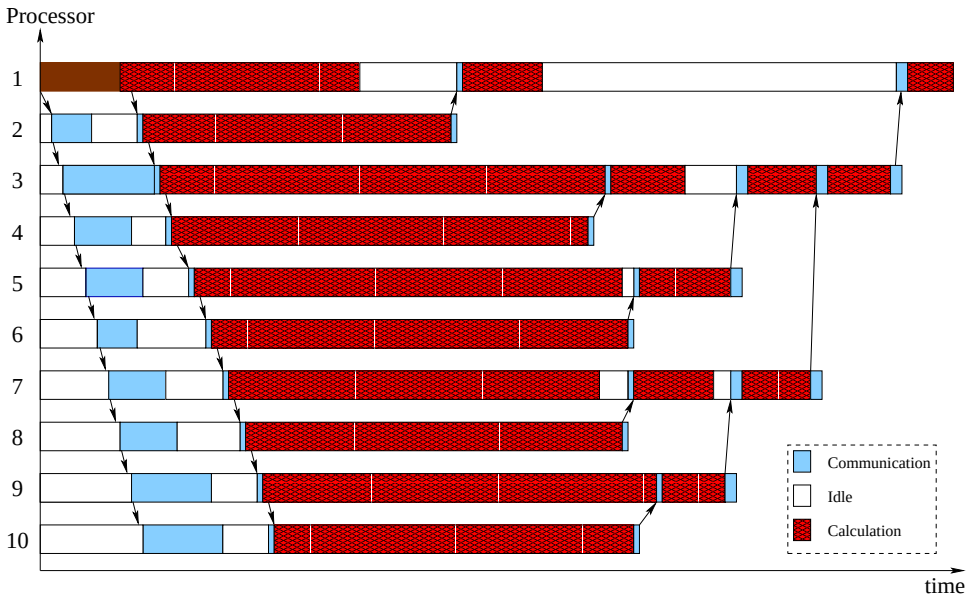


Figure 4.23: Processing diagram of pICA on ten processors.

IP Design

According to the structure of the pICA algorithm shown in Fig. 4.22, we design an IP for the pICA algorithm. Since our target prototyping FPGA, Xilinx VIRTEX 1000E, can only accommodate a pICA process with, at most, four weight vector estimations as we analyzed in [54], we estimate four independent components, i.e., $m = 4$, in this IP design. The structure of the IP design is demonstrated in Fig. 4.24. At the first step, two sub-matrices, each of which contains two one-unit estimation, totally generate four weight vectors by using the input observed signal $\mathbf{x}$. Secondly, every two weight vectors in the same sub-matrix execute the internal decorrelation. The four weight vectors then respectively conduct the external decorrelation with weight vectors from the other sub-matrix. So the decorrelated weight vectors generate the weight matrix $\mathbf{W}$. The observed signal $\mathbf{x}$ is then input again to execute the transformation $\mathbf{s} = \mathbf{W}\mathbf{x}$. So the observed signal $\mathbf{x}$ is unmixed to the source signal $\mathbf{s}$, which is finally output from the pICA IP.

From the functionality point of view, the structure of the pICA algorithm consists of two

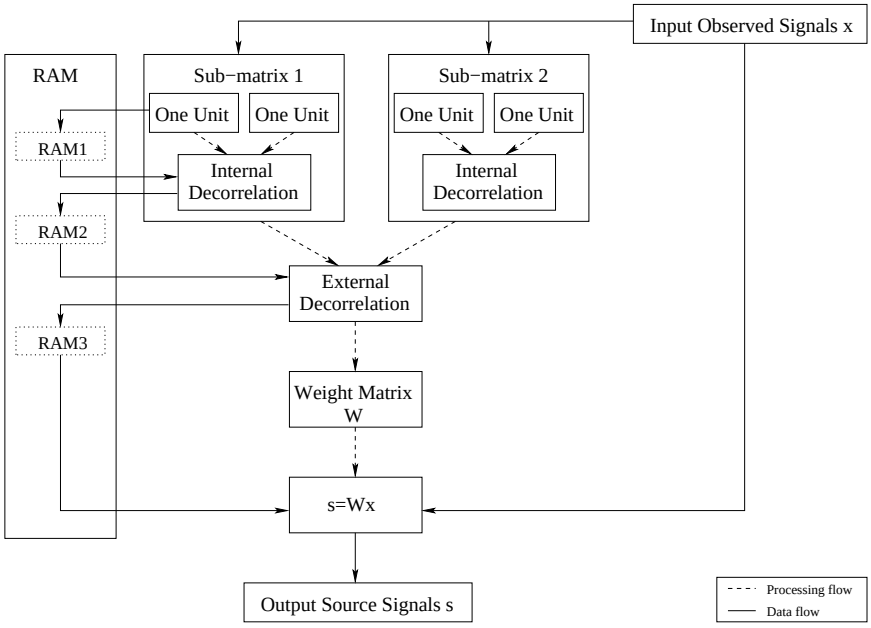
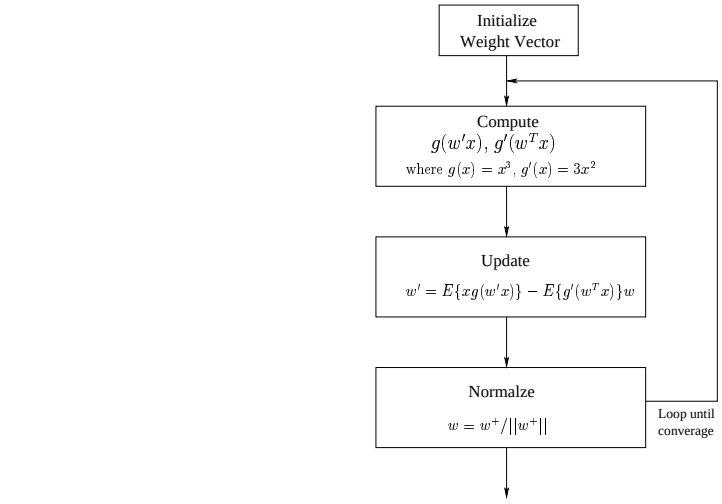


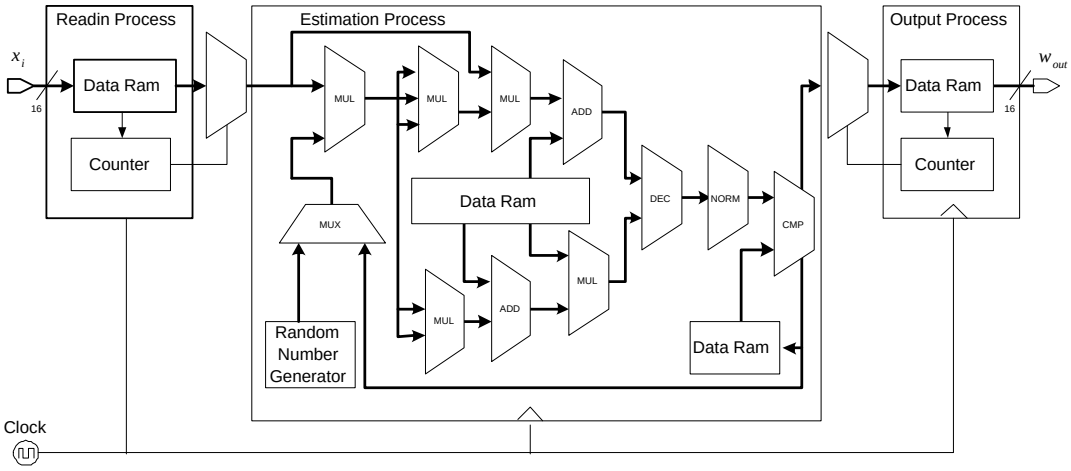
Figure 4.24: The IP design of the pICA algorithm.

main processes: the one unit weight vector estimation, the internal and external decorrelations. We therefore develop two function blocks for ICA-related applications. These blocks are parameterized by generics so as to make them highly flexible for future instances. In VHDL, the use of generics is a mechanism for passing information into a function block, similar to what Verilog provides in the form of PARAMETERS. Without loss of generality, we assume the number of the observed signals $n = 4$ and set the bitwidth of the input pixel to 16 in our designs. Both of them are adjustable for different applications by customizing the reconfigurable generics.

The process and the structure of the one unit estimation block are shown in Fig. 4.25. This function block consists of three processes, including readin, estimation, and output. The input ports of the one unit block consist of a 16-bit observed signal input and a 1-bit clock input that synchronizes the interconnected blocks. The output is the estimated weight vector $\mathbf{w}_i$ that needs to be decorrelated with others in the decorrelation process. Inside the one unit block, the 16-bit observed signal is fed in to estimate one weight vector. The weight vector is then iteratively



(a) Process.



(b) Function block.

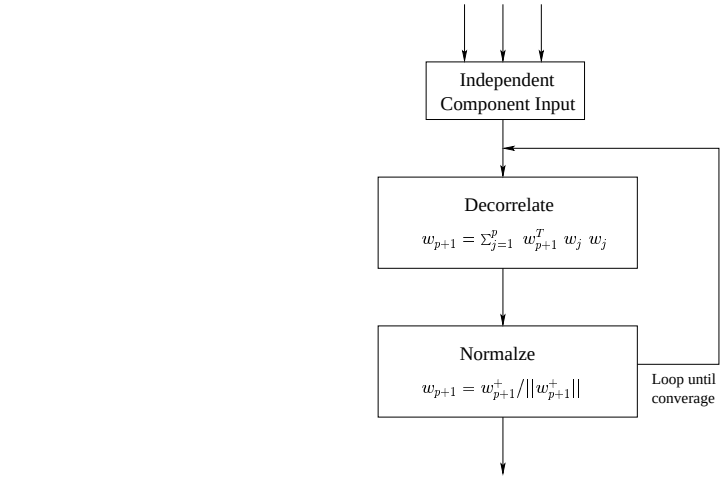
Figure 4.25: Design of the one unit estimation process.

updated until convergence. And the final estimated weight vector is then sent to the output port. By keeping the observation data and previously estimated weight vectors in the data RAM, Fig. 4.25(b) demonstrates how the input process, the estimation process, and the output process in the one unit block be assembled in a pipelined processing structure.

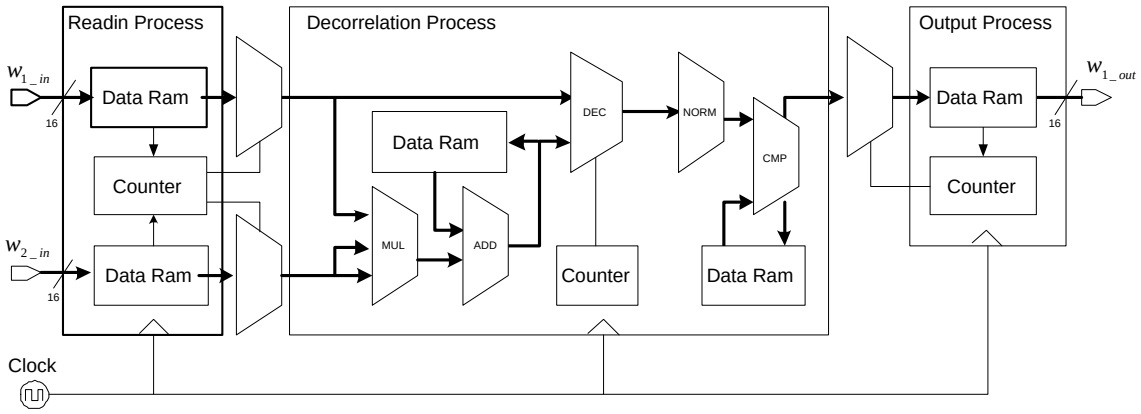
Figure 4.26 demonstrates the process and the structure of the decorrelation block that is designed for both the internal and the external decorrelations. The function block of decorrelation also contains three processes: readin, decorrelation, and output. The input ports include a 1-bit clock pulse (*clock*) and two 16-bit weight vector inputs ($\mathbf{w}_{1_in}$, $\mathbf{w}_{2_in}$), with $\mathbf{w}_{1_in}$ being the weight vector to be decorrelated, and $\mathbf{w}_{2_in}$ the sequence of previously decorrelated weight vectors. The generics parameterize the amount and dimension of the decorrelated weight vectors. The output of the block is a 16-bit decorrelated vector $\mathbf{w}_{1_out}$). The decorrelation block also sets up a pipelined processing flow that includes the input process, the decorrelation process, and the output process.

Since the decorrelation block involves both the internal and the external decorrelation that are interconnected to each other, it is necessary to respectively analyze the processing of these two cases. In the internal decorrelation, one initial weight vector is fed to the first 16-bit data port, while the weight vector that does not need to be decorrelated or the previously already decorrelated weight vector sequence is input to the other 16-bit data port. The weight vectors within one sub-matrix are then iteratively decorrelated. As shown in Fig. 4.27, the output decorrelated weight vector is then combined with the previously decorrelated weight vector sequence using multiplexer to serve the consequent round as a new decorrelated weight vector sequence.

In the external decorrelation process, if we only use one decorrelation block, the process works in virtually the same way as the internal decorrelation. The only difference is that the input decorrelated weight vector sequence is from another weight sub-matrix without multiplexing the output decorrelated weight vector. In order to speed up the external decorrelation process, we can set up parallel processing using multiple decorrelation blocks, as demonstrated



(a) Process.



(b) Function block.

Figure 4.26: Design of the decorrelation process.

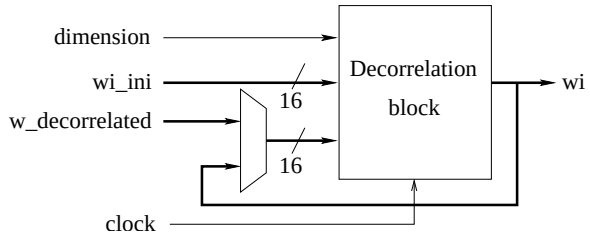


Figure 4.27: Internal decorrelation.

in Fig. 4.28. The initial weight vectors from current weight sub-matrix are respectively input to individual decorrelation blocks, while the decorrelated weight vector sequence from another weight sub-matrix are concurrently input to all blocks. The clock inputs are uniformly configured by external input for synchronization purpose.

Take a pICA process containing the estimation of four weight vectors as an example, the structure is shown in Fig. 4.29. The one unit module of this design consists of four one unit blocks in parallel, the decorrelation module includes three decorrelation blocks, two for the internal decorrelation in parallel and one for the external decorrelation. A top level block is designed to configure individual function blocks and interconnect collaborative blocks. In addition, the top level block serves as the input/output interface that distributes the input data, synchronizes the clock and sends out the final results. When the observed signal $\mathbf{x}$ is input to the pICA process, the top level block distributes them to the four one unit blocks. The weight vectors are then estimated in parallel and fed to the top level which in turn forwarded to the decorrelation process. Finally, the transformation process receives the weight matrix $\mathbf{W}$ and the observed signal $\mathbf{x}$ through the top level block, generates the source signal $\mathbf{s}$ that is output from the top level.

4.4 Implementation Challenges

After improving various image processing algorithms, we implement designs at the RTL level using VHDL. Then the designs are synthesized with target technologies, and finally carried out

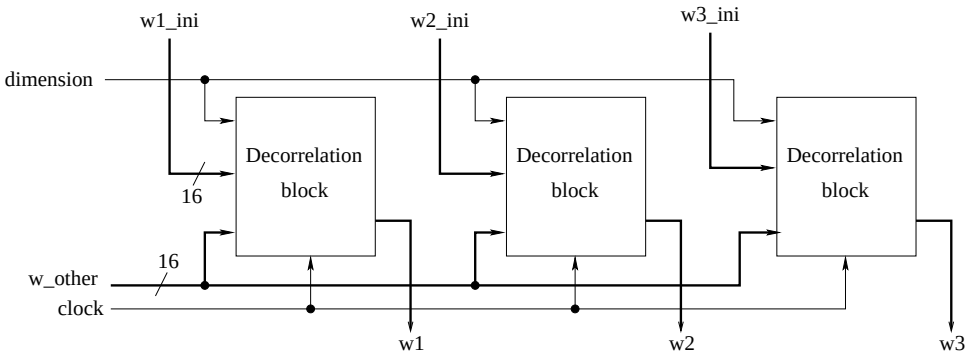


Figure 4.28: External decorrelation with multiple RCs in parallel.

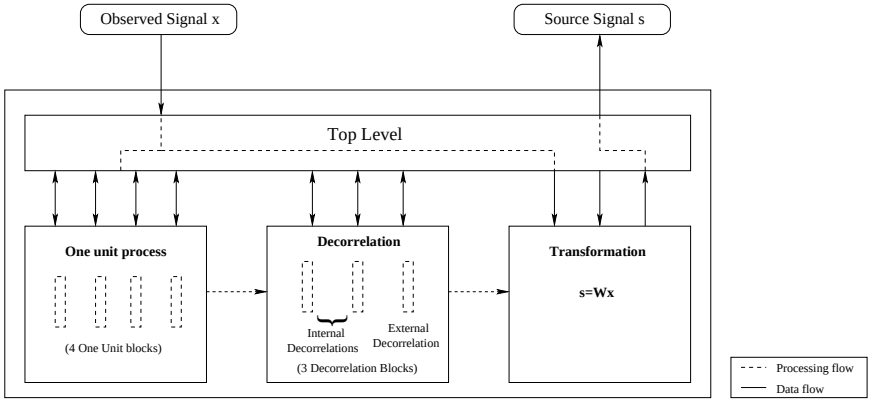


Figure 4.29: Structure of the pICA. (Solid lines denote data exchange and configuration. Dotted lines indicate the virtual processing flow.)

on prototyping FPGA or potential SoC.

During the implementation procedure, several design challenges should be highlighted for these image processing algorithms, especially the complex ones like the polynomial approximation-based geometric correction and the pICA algorithm. These challenges include the bitwidth, I/O bandwidth, internal RAM, and relationship between the processing frequency and the algorithm computation complexity.

The bitwidth is a big concern in hardware implementations for image processing applications that involve the handling of large volume of data. The short bitwidth easily leads to the overflow problem, while the long bitwidth dramatically increases the delay and the utilization area, thereby wasting resource. The bitwidth highly depends on specific applications. In addition, the rounder can be used to avoid overflow.

The I/O bandwidth is a common challenge in image processing implementations on VLSIs. For some algorithms, images need to be input and processed frame by frame, while the digital circuit only reads in data bit by bit from multiple I/O ports whose amount is far less than the size of most images. Therefore, if the size of the input image is decreased, the size of the internal RAM will correspondingly decrease, which results in the reduction in delay and the increase in the image input frequency.

Capacity concerns like the RAM size are also general for image processing applications. It reflects a balance among several other challenges such as bitwidth, I/O bandwidth, and algorithm complexity. If more functions are to be implemented, the RAM size has to be reduced in order to match resource limitations.

Another challenge is how to balance the processing frequency and the algorithm computation complexity. If a computation needs a long clock cycle to finish the process, it decreases the processing frequency and prolongs the overall processing time. To solve this problem, the pipelined structure that we previously discussed is broadly used in fast circuit designs. Some efficient approximations or simpler substitutes can also be used. The purposes of both meth-

ods are to reduce the delay within processes and increase the overall processing frequency. In addition, the algorithm computation complexity should be restricted according to the need of specific applications. For example, the degree of the polynomial is a very important issue in the implementation of the polynomial approximation-based geometric correction. Increasing the degree by one will lead to significant increase in bitwidth and the RAM size, resulting in longer delay and more power consumption. If the design could be simplified at the early algorithm improvement stage without sacrificing many quality requirements, then the performance of the implementation will be dramatically improved.

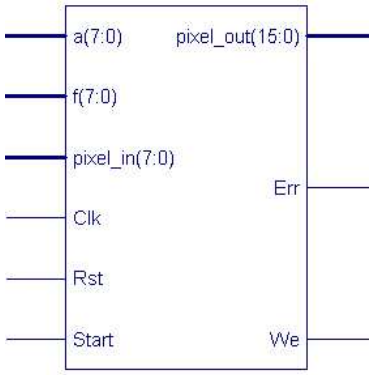
4.5 Algorithm Implementation on Virtual Microsensor Platform

After the algorithm improvement that incorporates partitioning and clustering, we describe the detailed implementations of the contrast stretching, 3×3 filter, polynomial approximation-based geometric correction, and pICA algorithm on the virtual microsensor platform.

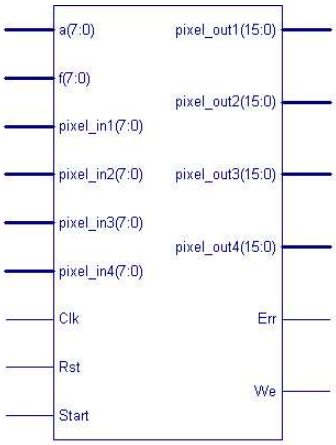
4.5.1 Contrast Stretching

For the contrast stretching, we respectively implement the three designs including the traditional, the 4-pixel parallel, and the 2-stage pipeline & 4-pixel parallel designs according to Figs. 4.6 to 4.8. Without loss of generality, we set the bitwidth to 8 in these implementations. Figure 4.30 illustrates these designs. The traditional design has only one pixel input and one pixel output. The 4-pixel parallel design is the stack of four traditional designs, therefore having four pixel inputs and four pixel outputs. The 2-stage pipeline & 4-pixel parallel design has the same symbol as that of the 4-pixel parallel design.

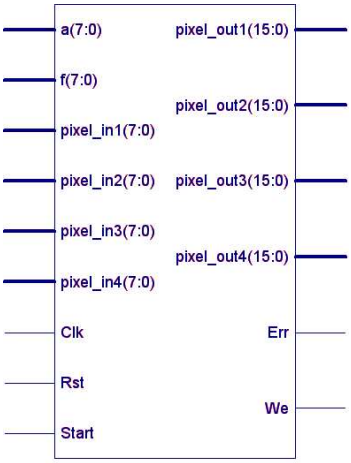
Figure 4.31 elaborates the RTL schematics of the traditional and the 4-pixel parallel designs. From the RTL schematic of the traditional design, we can clearly observe the sequential processing of the multiplier and the adder. Each block in the RTL schematic of the 4-pixel parallel design has exactly the same schematic of the traditional design.



(a) Traditional design.

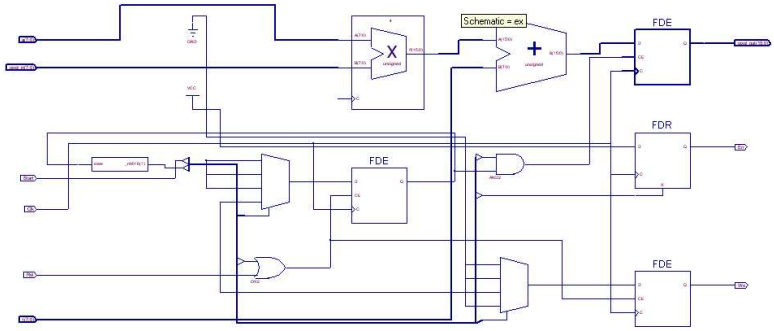


(b) 4-pixel parallel design.

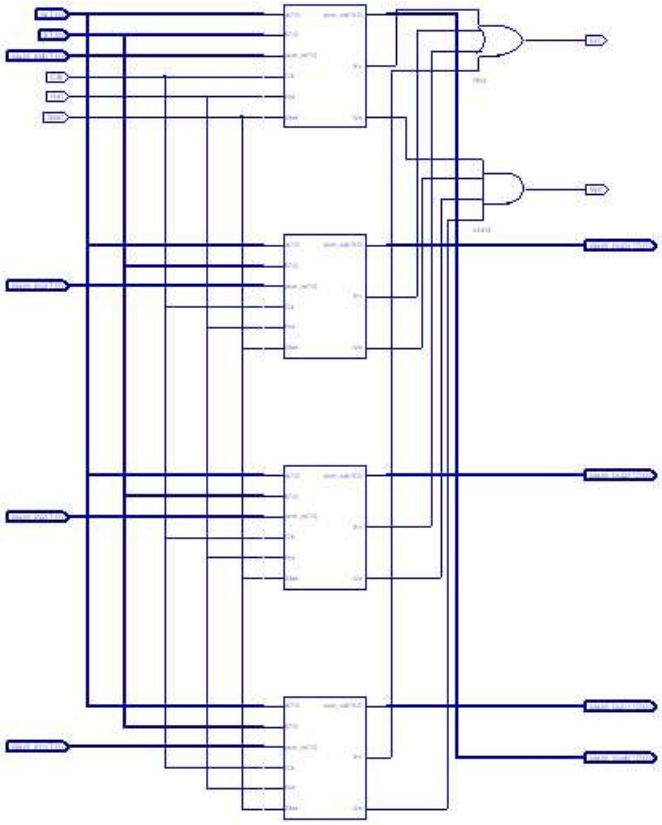


(c) 2-stage pipeline & 4-pixel parallel design.

Figure 4.30: Contrast stretching designs.



(a) Traditional design.



(b) The 4-pixel parallel design.

Figure 4.31: RTL schematics of contrast stretching (1).

The RTL schematic of the 2-stage pipeline & 4-pixel parallel design is shown in Fig. 4.32. On each block, we can see the pipelined structure, where the output of the first stage connects to the input of the second stage. Figure 4.33 shows the internal schematic of each block, where the multiplier executes at the first stage, and the adder executes in sequence at the second stage.

4.5.2 Polynomial Approximation-based Geometric Correction

The design of the polynomial approximation-based geometric correction is more complex than the contrast stretching. It includes three main function blocks, including matrix **W** formation, matrix multiplication, and matrix inverse.

The top view schematic of the **W** formation is shown in Fig. 4.34, which contains the **W** formation block, the state machine used to control the procedure, and the RAM used to save the temporary results. The detailed schematic of the **W** formation block is shown in Fig. 4.35. This schematic includes several pipelined structures of multipliers whose purposes are to save resources during the matrix element calculation procedure. When the control points are input to the **W** formation block, the corresponding elements in **W** are calculated and saved in the internal RAM for future processing.

As the schematic shown in Fig. 4.36, the matrix multiplication block reads in the elements of row and column simultaneously, accumulates the intermediate result internally, until finally outputs the element of the result matrix. This sequential processing is appropriate to the multiplication of large matrices. For the multiplication of smaller matrices, we can use parallel & pipelined structures to speed up the process.

In the matrix inverse operation, as shown in Fig. 4.37, we first sequentially input the elements of the matrix **A** to the LU factorization block, and decompose the matrix into **L** (the lower triangular matrix) and **U** (the upper triangular matrix). Then the elements of the LU matrix is sent to the LU inverse block to calculate the elements of the inverse matrix that are sequentially

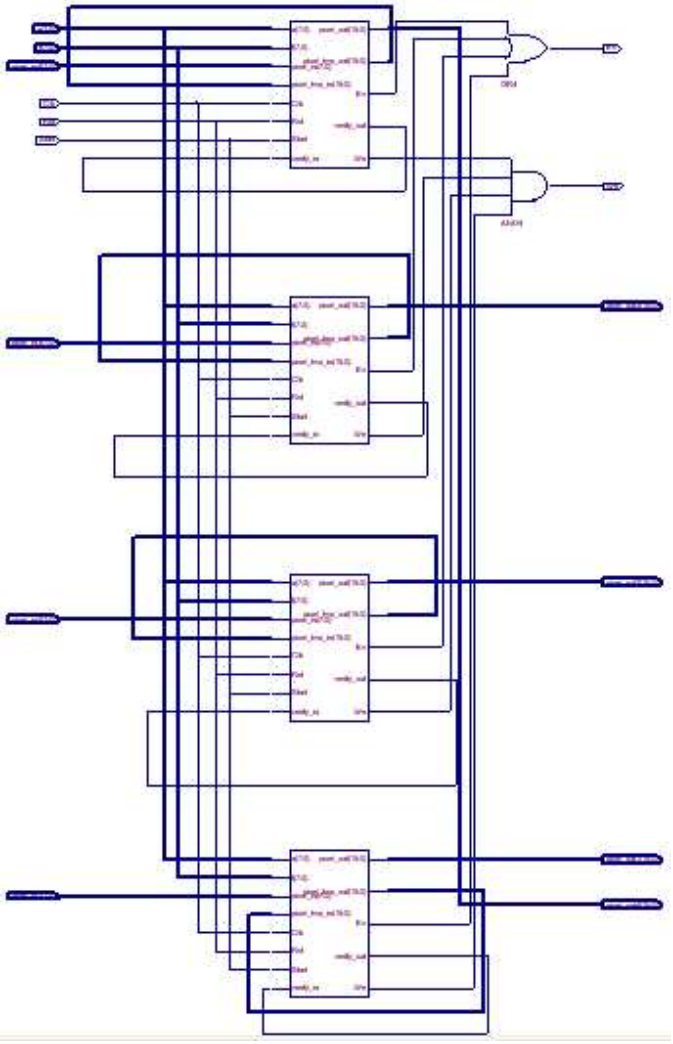


Figure 4.32: RTL schematic of contrast stretching (2). (The 2-stage pipeline & 4-pixel parallel design)

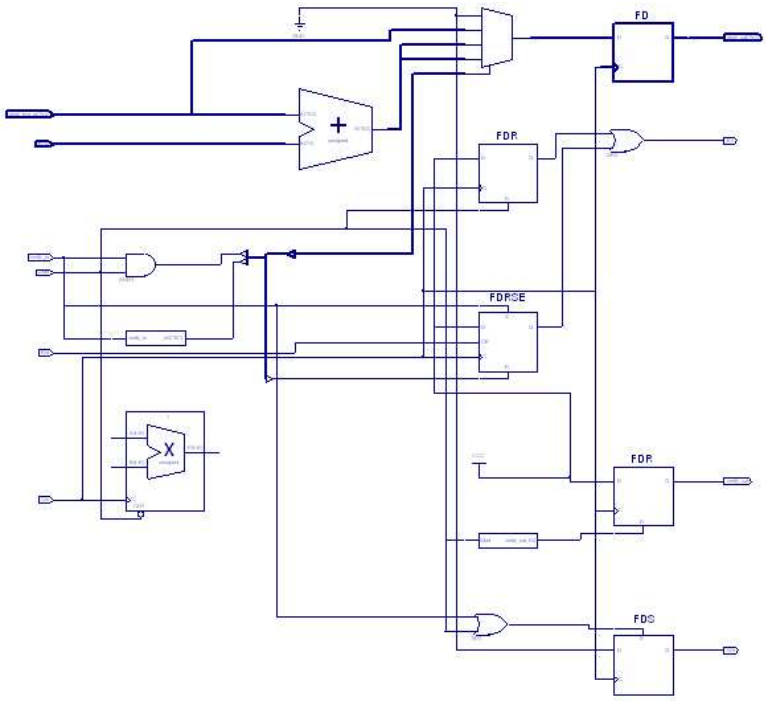
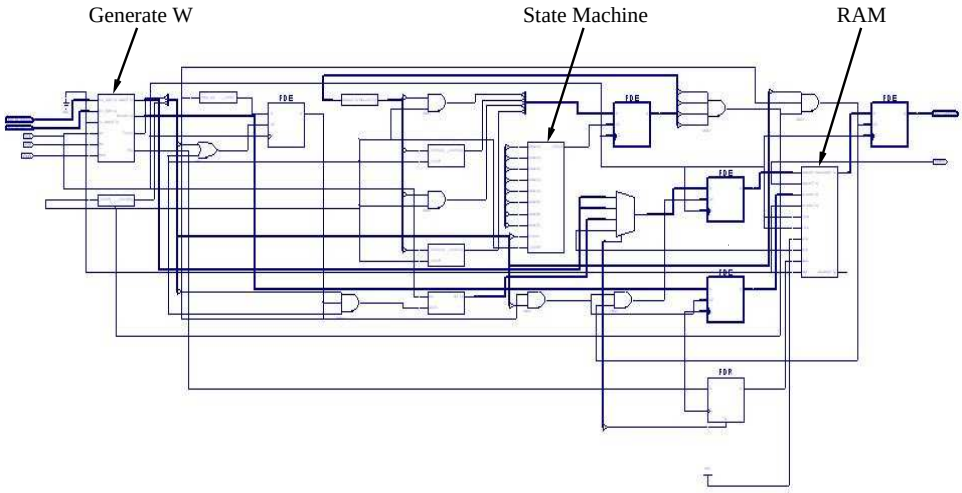
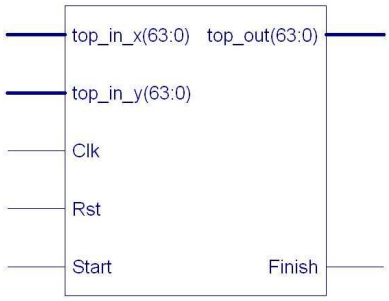


Figure 4.33: Block of the 2-stage pipeline.



(a) Schematic.



(b) Symbol.

Figure 4.34: RTL schematic of the **W** formation block at top view.

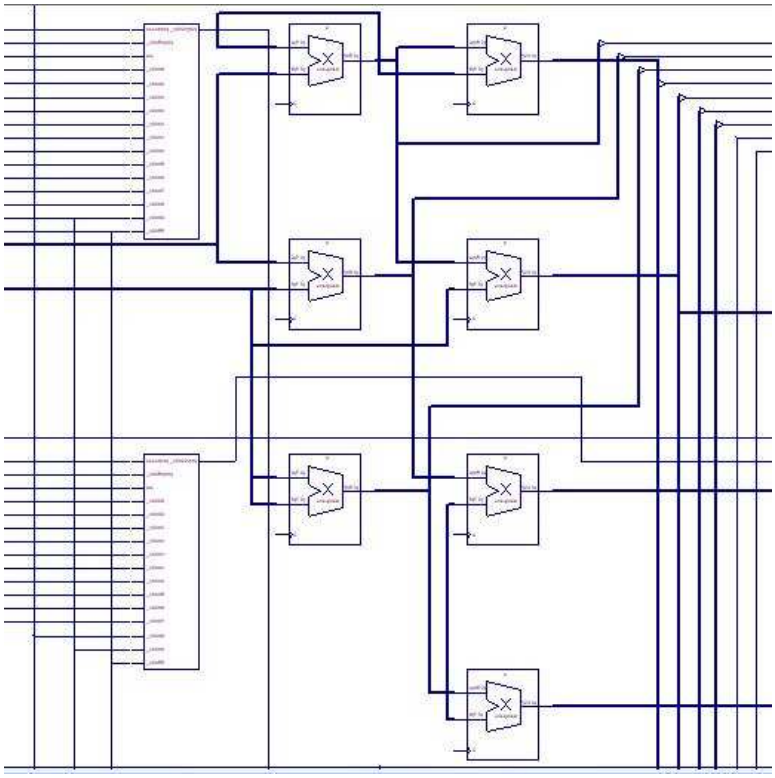
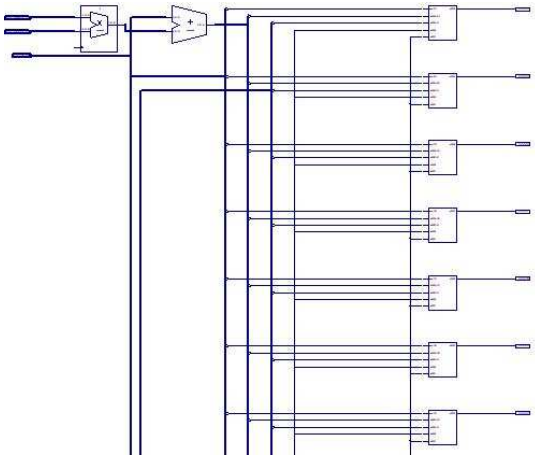
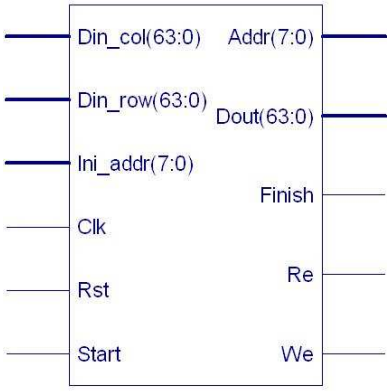


Figure 4.35: RTL schematic of the **W** formation block.

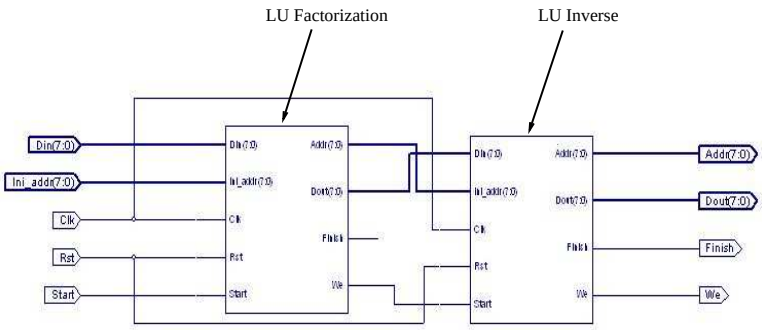


(a) Schematic.



(b) Symbol.

Figure 4.36: RTL schematic of the matrix multiplication block.



(a) Schematic.

(b) Symbol.

Figure 4.37: RTL schematic of the matrix inverse block.

output to RAM. Both of the LU factorization and the LU inverse blocks contain internal RAMs to save the temporary results. We skip the detailed schematics of these two blocks since they are too large to effectively demonstrate the design structures.

By including the $\mathbf{W}$ formation, the matrix multiplication, and the matrix inverse blocks, Figs. 4.38, 4.39, and 4.40 show the overall RTL schematic of the polynomial approximation-based geometric correction design. From the schematic in part 1, we can see the pipelined processing of the pseudo-inverse $\mathbf{W}^{-1} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T$. Then the calculations of $\mathbf{A} = \mathbf{W}^{-1}\mathbf{X}$ and $\mathbf{B} = \mathbf{W}^{-1}\mathbf{Y}$ are conducted in parallel. Finally, $\hat{\mathbf{X}} = \mathbf{WA}$ and $\hat{\mathbf{Y}} = \mathbf{WB}$ are also calculated simultaneously. Figure 4.41 shows the symbol of the overall design. The input ports include the control points (u, v), the corresponding points (x, y) in the distorted image, and the coordinates in the transformation image. The output ports are the coordinates in the distorted image.

4.5.3 3×3 Filters

Designs of the 3×3 filter include the traditional design, the parallel & pipelined design, and the parallel & pipelined design with partitioning.

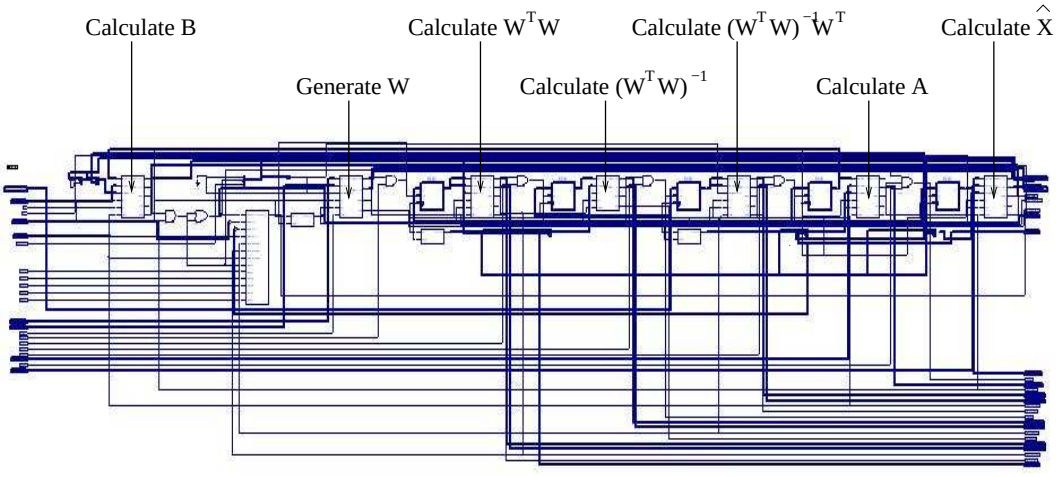


Figure 4.38: RTL schematic of the geometric correction design (Part 1).

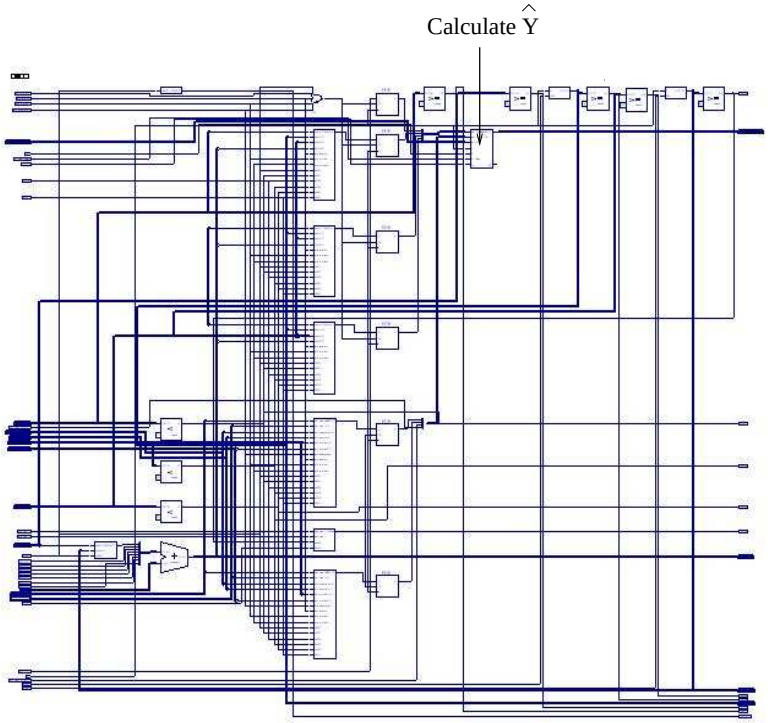


Figure 4.39: RTL schematic of the geometric correction design (Part 2).

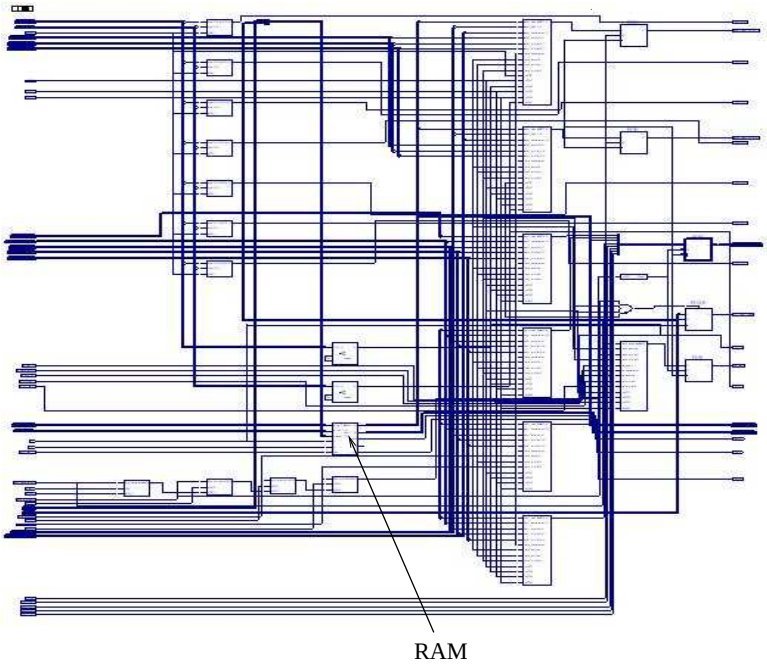


Figure 4.40: RTL schematic of the geometric correction design (Part 3).

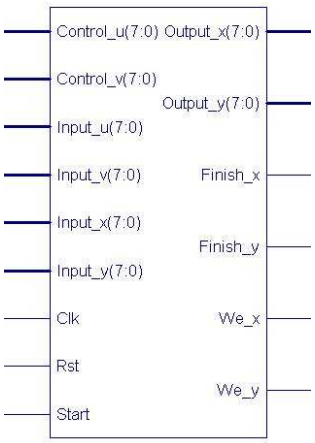


Figure 4.41: Symbol of the geometric correction design.

These designs have the same symbol, as shown in Fig. 4.42. Each design has eight pixel inputs and one pixel output.

From the RTL schematic of the traditional design shown in Fig. 4.43, we can observe the pipelined structure of the multiplication and the addition operations.

Figures 4.44 and 4.45 respectively demonstrate the internal structures of the nine multiplication operations in parallel and the eight sequential addition operations.

Figure 4.46 and 4.47 show the RTL schematic of the parallel & pipelined design. When we examine the schematic from the top-down view, we can see the parallel structures of the nine multipliers at the first level, the four adders at the second level, and the two adders at the third level. If we examine the schematic from left to right, we can recognize the 5-stage pipeline structure, where the stages one, two, and three are located in the Fig. 4.46, and the stages four and five in Fig. 4.47.

Obviously, the multiplier, adder, and buffer are basic blocks in the parallel & pipelined design. We therefore demonstrate the schematics of these blocks in detail in Figs. 4.48, 4.49, and 4.50. The multiplier has two inputs, one for the element from the kernel of 3×3 filter, one for the input pixel. The output of the multiplier is sent to the adder that have two inputs connected to two multipliers. The buffer contains a stack that can store 3 intermediate results simultaneously.

Figure 4.51 demonstrates the schematic of the parallel & pipelined design with partitioning. We find that the partitions V_1 and V_5 are located at the left side, the partitions V_2 and V_4 in the middle, and the partitions V_3 at the right side. Since the multiplication and addition operations are respectively grouped in individual partitions, we only see the symbols of partitions but not the operations.

According to the design partitioning shown in Fig. 4.21, we identify three types of partitions, i.e., the type of the partitions V_1 and V_5, that of the partitions V_2 and V_4, and that of the partition V_3.

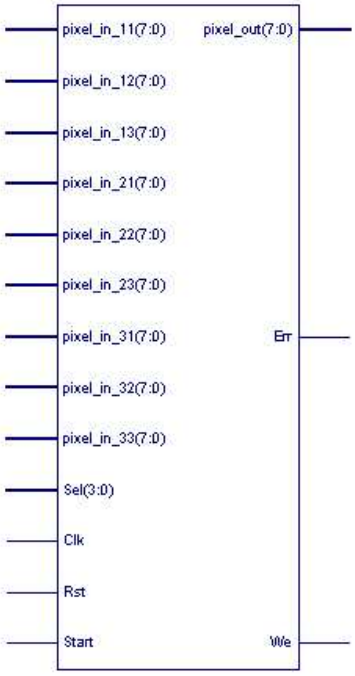


Figure 4.42: Symbol of 3×3 filter designs.

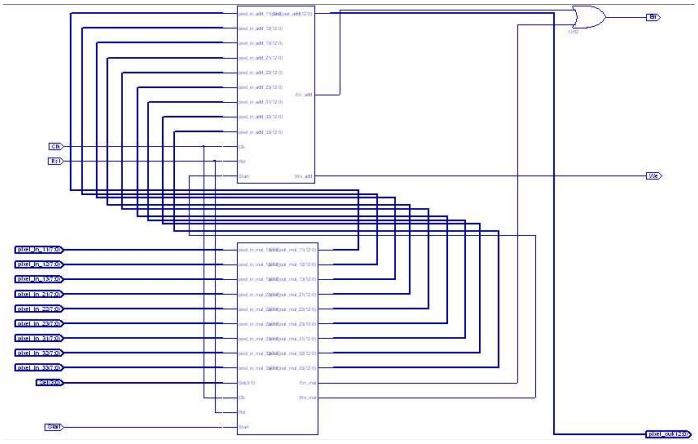


Figure 4.43: RTL schematic of the traditional 3×3 filter design.

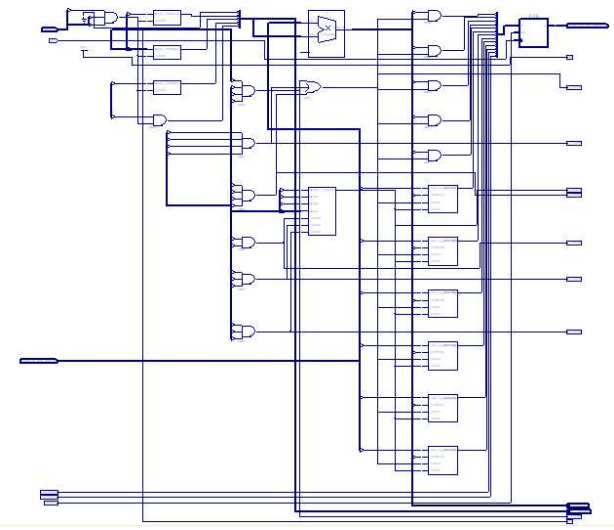


Figure 4.44: RTL schematic of the multiplication.

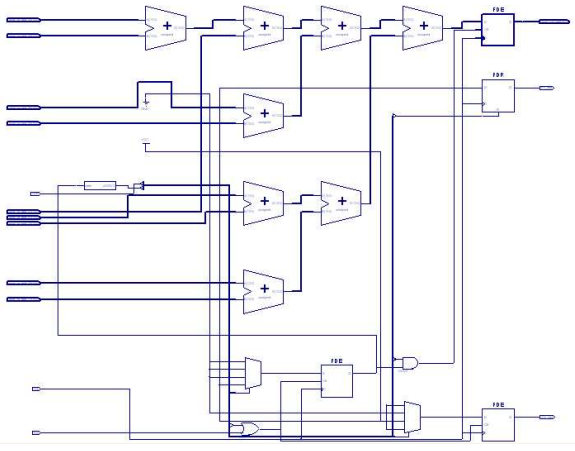


Figure 4.45: RTL schematic of the addition.

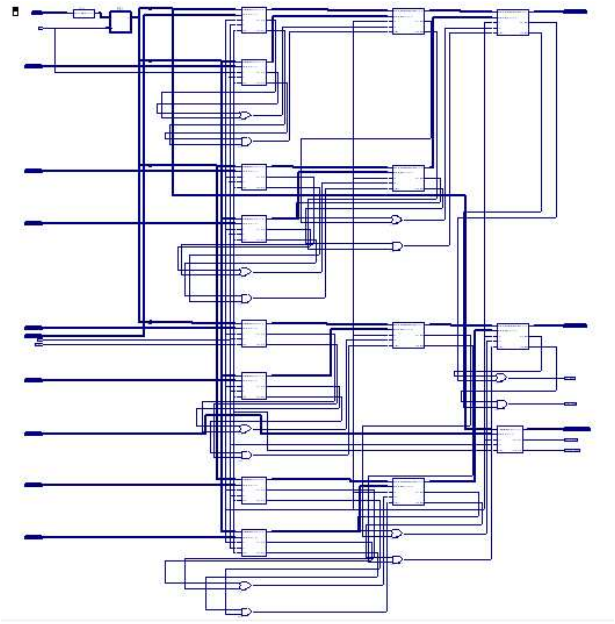


Figure 4.46: RTL schematic of the parallel & pipelined design (Part 1).

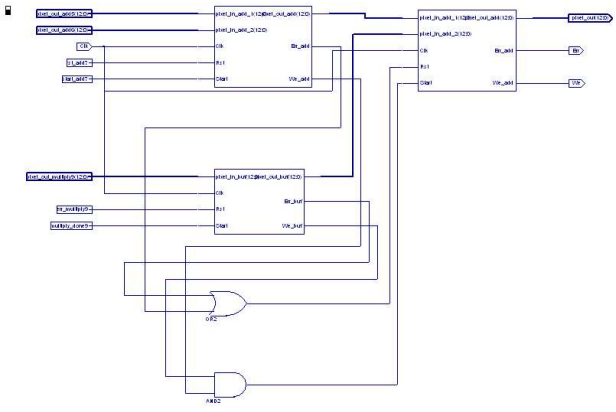
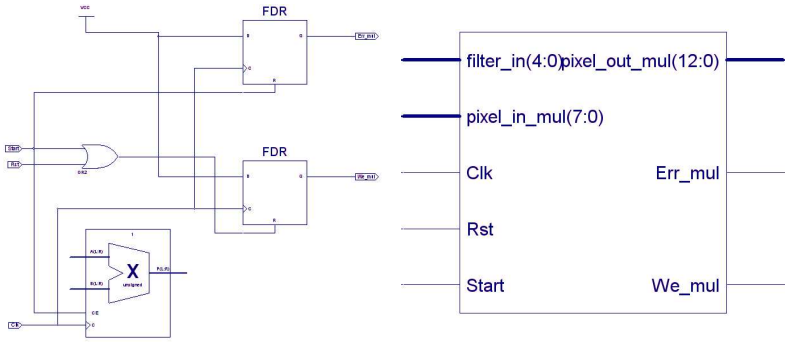


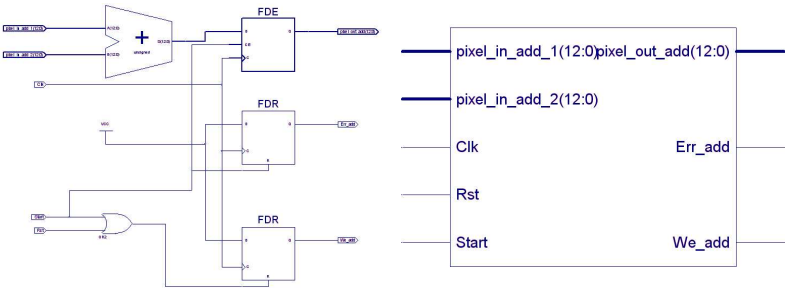
Figure 4.47: RTL schematic of the parallel & pipelined design (Part 2).



(a) Schematic.

(b) Symbol.

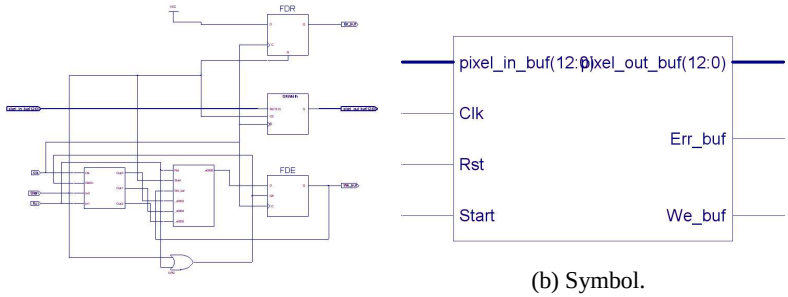
Figure 4.48: RTL schematic of the multiplier in parallel & pipelined design.



(a) Schematic.

(b) Symbol.

Figure 4.49: RTL schematic of the adder in parallel & pipelined design.



(a) Schematic.

(b) Symbol.

Figure 4.50: RTL schematic of the buffer in parallel & pipelined design.

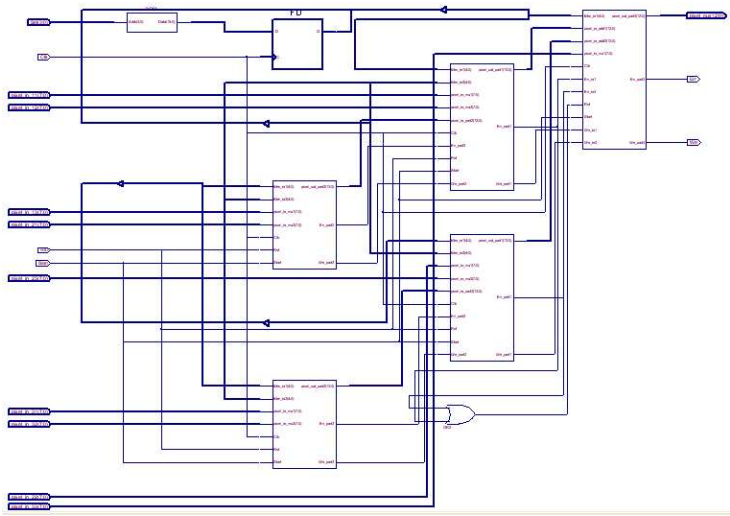


Figure 4.51: RTL schematic of the parallel & pipelined design with partitioning.

The detailed schematic of these partitions are shown in Figs. 4.52, 4.53, and 4.54. Each partition contains multipliers, adders, or buffer that are connected to each other as described in Sec 4.3.3. In the designs for the 3×3 filter, some kernels such as those listed in Table 4.2 have been stored at the top level of the design. Users can also specify their own kernels and eliminate the pre-defined ones, thereby further reducing the utilization area.

4.5.4 Parallel Independent Component Analysis

According to the design presented in Sec. 4.3.4, we implement the pICA process containing the estimation of four weight vectors. The pICA design mainly contains two kinds of function blocks: the one unit weight vector estimation and the decorrelation. Figure 4.55 shows the symbols of these two blocks. The one unit block sequentially takes the observed signals $\mathbf{x}$ and outputs one estimated weight vector. The decorrelation block has inputs for two weight vectors, one for the estimated weight vector from the one unit block or the uncorrelated weight from other decorrelation block, the other for the decorrelated weight vector from its own output or the RAM. The output is the decorrelated weight vector. The detailed structures of these two

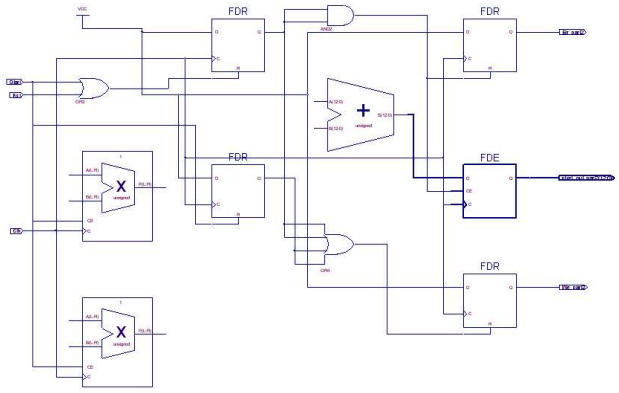


Figure 4.52: RTL schematic of partition type 1 (V_1, V_5).

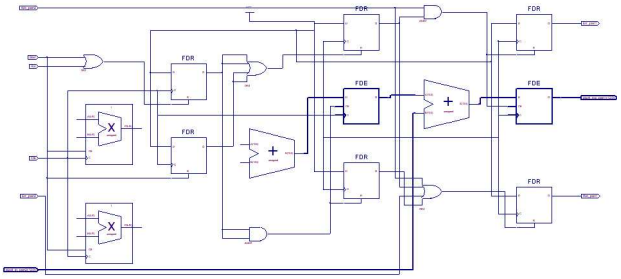


Figure 4.53: RTL schematic of partition type 2 (V_2, V_4).

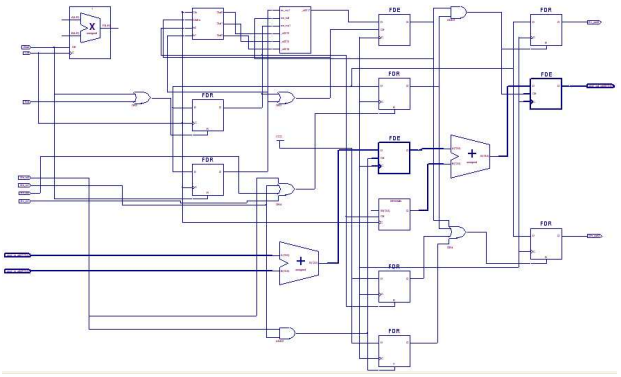
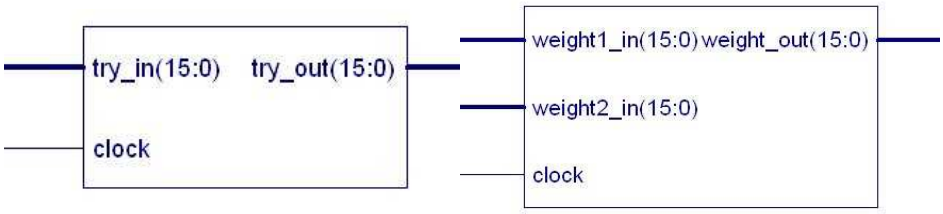


Figure 4.54: RTL schematic of partition type 3 (V_3).



(a) One unit estimation.

(b) Decorrelation.

Figure 4.55: Symbols of function blocks in the pICA design.

blocks can be refereed to Fig. 4.25 and 4.26.

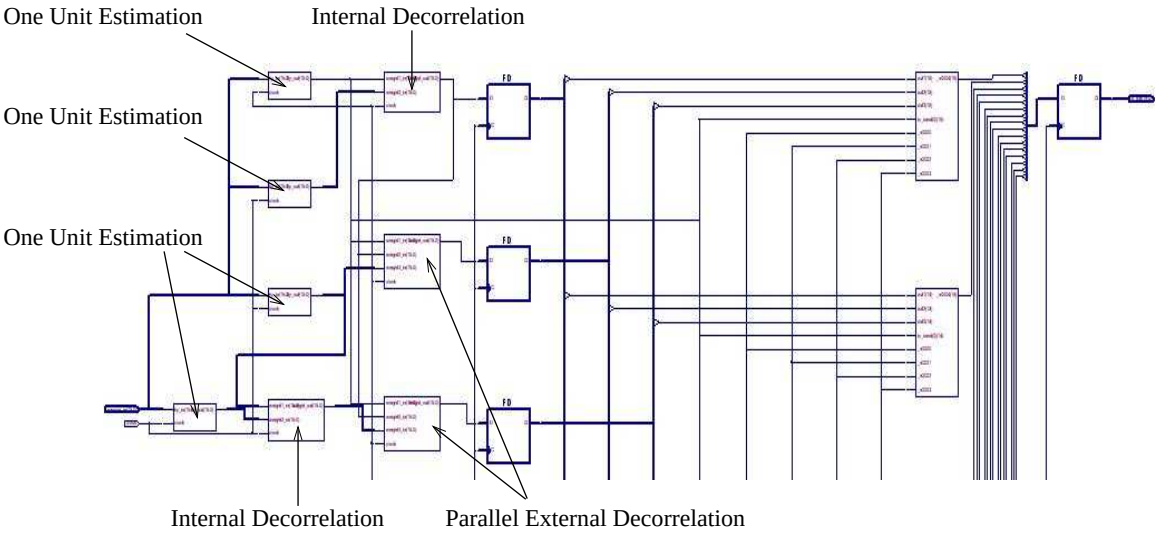
The overall schematic view of the pICA design is shown in Fig. 4.56. We can see that the four one unit blocks conduct the weight vector estimations in parallel. Then every two weight vectors respectively conduct the internal decorrelation. Finally, a parallel structure consisting of two decorrelation blocks conduct the external decorrelations for all weight vectors.

4.6 Microsensor Integration

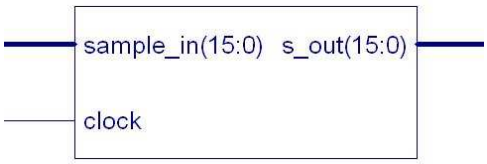
By integrating the pre-qualified image processing IPs that are developed from the improved algorithms, users can quickly obtain the microsensor designs according to the requirements of specific applications. In this section, we use an example that consists of a series of image processing IPs to demonstrate the integration and implementation procedure.

As the structure of the example integration design shown in Fig. 4.57, we first apply the contrast stretching to the input image. The intermediate results are saved in RAMs. The image is then sent to the Sobel edge detector (horizontal), whose kernel is listed in Table 4.2. Finally, the result image is output from the design.

Figure 4.58 demonstrates the symbol of the integration design. The input ports include three pixel inputs in parallel, since the Sobel edge detector that is a 3×3 filter requires three new pixel inputs for every movement of the processing window. The output port generates pixels in



(a) RTL schematic.



(b) Symbol.

Figure 4.56: RTL schematic of the pICA design.

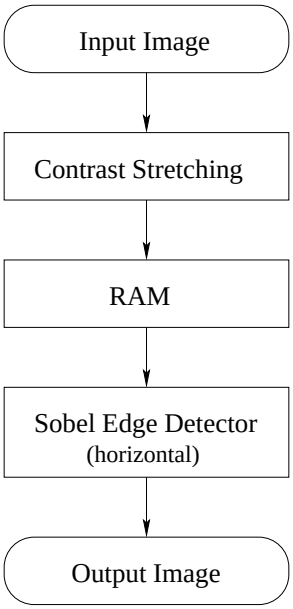


Figure 4.57: Structure of the integration design.

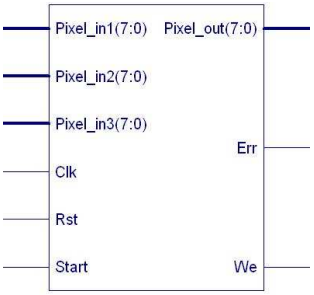


Figure 4.58: Symbol of the integration design.

the result image, one pixel per clock cycle.

We find that this design requires the integration of one contrast stretching IP, one 3×3 filter IP, and several RAMs. Since the required IPs are available in the image processing IP library, we directly use them after configuring parameters, where the bitwidth is set to 8. The RTL schematic is shown in Fig. 4.59. We choose the 2-stage pipeline & 4-pixel parallel design as the contrast stretching IP. Since the Sobel filter requires three new inputs in every one clock cycle, only three of the four parallel components are used. In order to save the results of the contrast stretching, we design nine RAM blocks and form a 3-stage pipeline structure. Each stage contains three RAM blocks. RAMs at the first stage receive results from the contrast stretching IP in current clock cycle and pass them respectively to the three RAMs at the second stage in the next clock cycle. The RAMs at the second stage perform the same way and forward results to RAMs at the third stage. The update of the pipelined RAMs therefore satisfies the input requirement of the Sobel filter for each movement. We select the parallel & pipeline design with partitioning as the 3×3 filter IP for the Sobel filter, which outputs one result pixel in every clock cycle.

4.7 VLSI Implementation

The microsensor developed from the virtual microsensor platform usually consists of several pre-qualified image processing IPs. After integrating the desired IPs and synthesizing the whole design with target technologies, we carry the application-specific microsensor design to the physical implementation level. Like most existing VLSI implementation works, this implementation procedure includes two steps: the implementation on the prototyping FPGA platform, and the re-targeting at potential SoC. The implementation on the prototyping FPGAs is for testing purposes. We introduce two FPGA test boards: the Pilchard [96] and the Amirix testing platforms [8]. The final SoC is either a digital ASIC embedded with soft CPU cores or an FPGA containing hardware CPU cores. In this work, we use the XUP (Xilinx University Pro-

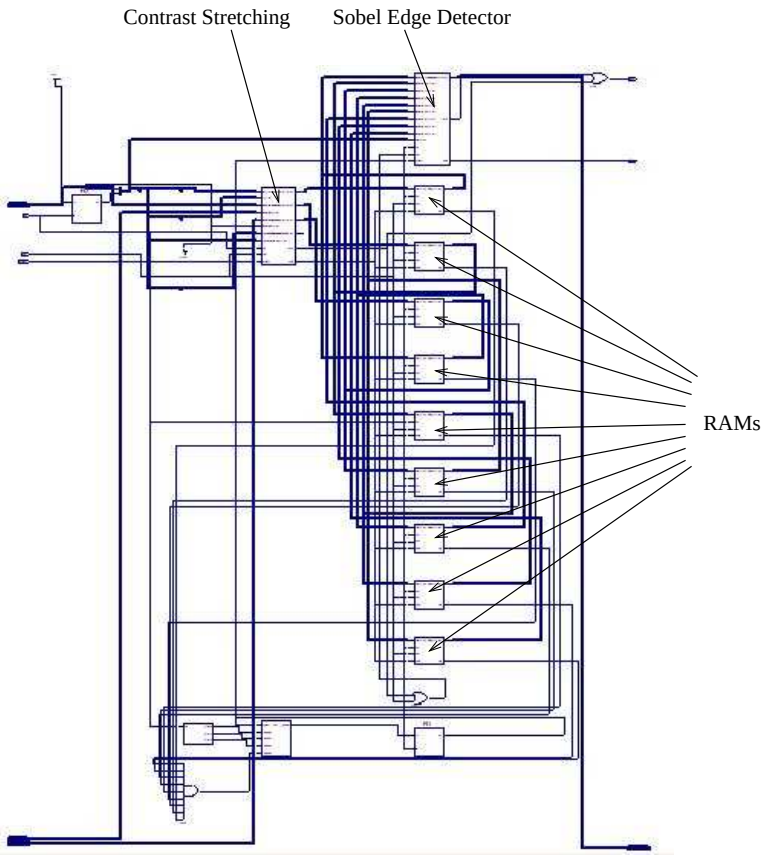


Figure 4.59: RTL schematic of the integration design.

gram) board [138] developed by the Digilent company as the target platform. The XUP board is embedded with a Xilinx Virtex II Pro FPGA that includes two PowerPC CPU cores.

4.7.1 Implementation Procedure

One important advantage of the virtual microsensor platform is the isolation of the functional IP design that depends on specific applications and the physical implementation on hardware that eventually executes the functions. The IP designs and the related image processing library come in the form of hardware description languages. And the physical implementation comes in the form of integrated circuits on FPGAs or ASICs. The hardware resource is considered in the virtual microsensor platform all through the development procedure, i.e., by resource modeling at the algorithm improvement stage, by CAD tools at the synthesis stage, and by performance evaluation at the integration stage. On the contrary, the hardware resource is not a big concern to users who only need to select the necessary IPs for the target platforms. Figure 4.60 illustrates the implementation procedure for a microsensor design from the virtual platform to the final SoC.

Given a specific application, users first decide the target platform and select the appropriate IPs, for example, IPs A and B, from the image processing library in the virtual microsensor platform. Since the virtual platform is designed to interpret specific applications at a higher implementation level, it is necessary to specify IPs restrictively according to the need of the applications. The configuration for applications before synthesis instead of on hardware platform is another advantage of the virtual platform design, whose purpose is to reduce hardware cost and reserve energy. After the IP selection and configuration, we integrate the IPs into one microsensor design and evaluate the overall performance by synthesizing with the target technology. Then the design can be implemented on the prototyping FPGAs for validation. In order to achieve fast validation and test on hardware at low cost, the FPGA prototyping is a necessary step in the physical implementation procedure. Although some applications may

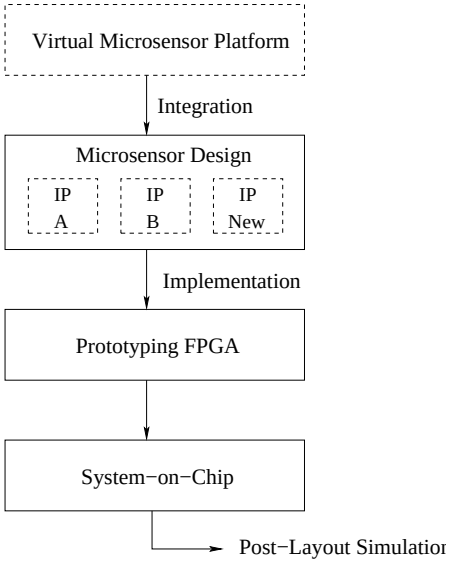


Figure 4.60: Implementation procedure.

desire platforms with reconfigurable capability that direct to the use of FPGAs, FPGAs are energy consuming devices and not suitable to be backbones of the battery supplied microsensors. Therefore, it is better to leave the configuration work at the early stage in the procedure and use FPGA mainly as prototyping platform before SoC implementation. After the prototyping on FPGA, the microsensor design is ready to be implemented on the SoC platform with post-layout simulation.

4.7.2 Prototyping FPGA Platforms

For the physical implementation of the microsensor design, the prototyping FPGA plays an important role in validation and test. With the fast development of physical, material, and electronics technologies, current FPGAs have provided users sufficient capacity and powerful computing capability. The fast development of FPGA technologies is mainly driven by the more complicated applications that desire direct implementations on hardware. Current phenomenal growth of FPGA technologies have been beyond the Moore's law prediction of doubling in the number of transistors per integrated circuit every 18 months. A comparison between the FPGA

capacity tendency and the Moore's Law is demonstrated in Fig. 4.61.

In order to verify and prototype the microsensor design, we introduce two FPGA prototyping platforms: the Pilchard and the Amirix platforms. The Pilchard reconfigurable computing platform [96] is plugged in memory slot on a SUN workstation, as shown in Fig. 4.62. The Pilchard platform is embedded with a Xilinx VIRTEX V1000E FPGA that contains 1M equivalent gates for programming. In general, FPGA platforms use PCI or PCMCIA slots to exchange data with memory and communicate with CPU. However, the data transfer speed can be extremely slow for applications with large data sets. The Pilchard platform uses the DIMM RAM slot as an interface and is compatible with the 168 pin, 3.3V, 133MHz, 72-bit, registered synchronous DRAM in-line memory modules (SDRAM DIMMs) PC133 standard [96], thereby achieving very high data transfer rate.

Compared to the Pilchard platform, the Amirix board is embedded with a more powerful Xilinx Virtex II Pro FPGA [8]. This FPGA contains two PowerPC cores, and is manufactured based on the technology of 0.15 μm 8-Layer Metal process with 0.12 μm high speed transistors. It allows users to implement designs on 8M logic gates with 420 MHz internal clock speed and 840Mb/s I/O [139]. Figure 4.63 shows the Amirix board and its embedded Xilinx Virtex II Pro FPGA. The Amirix board is in compliance with the PCI-X 133MHz, PCI 66MHz and 33 MHz standards.

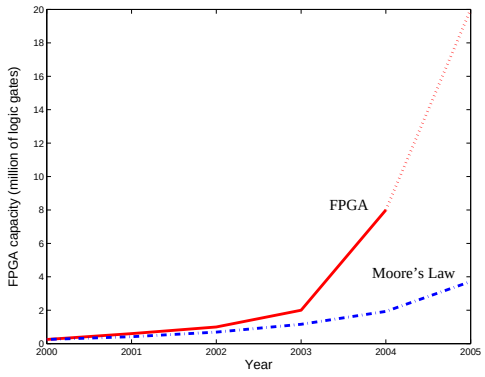


Figure 4.61: FPGA developments vs. Moore's Law.

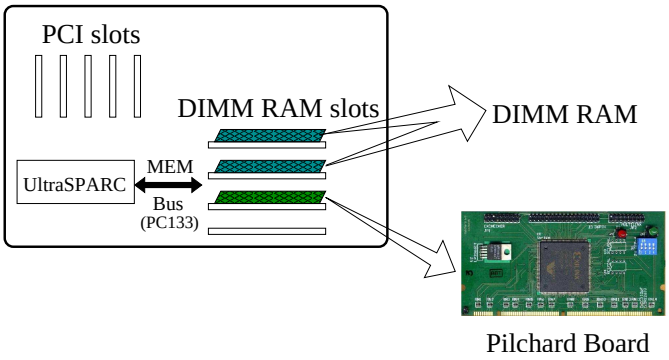


Figure 4.62: The Pilchard platform embedded with Xilinx Virtex V1000E FPGA.

Figure 4.63: The Amirix platform embedded with Xilinx Virtex II Pro FPGA.

4.7.3 SoC Platform

After the FPGA prototyping validation, we can finalize the design with the SoC implementation for the physical microsensor node. The development of SoC benefits from the advances in MEMS technologies such as high density integration. In general, SoC refers to the integration of all the necessary electronic circuits of diverse functions onto a single chip, to come up with a complete electronic system that performs the more complex but more efficient final function [124]. SoC technology allows various chips and components to be fabricated together on a single chip, instead of assembling these parts on a circuit board. Due to the circuit integration on a single chip, the advantages of SoC includes higher performance, smaller space requirements, higher system reliability, and lower power consumption. Table 4.3 summarizes and predicts the trend of change in SoC design productivity. It shows that the change of manpower on SoC design depends on the improvement of reuse overhead and the efficiency of design methodology. The first two items are the general technology parameters in VLSI fabrication process. The smaller number in technology leads to the higher circuit density and lower power consumption. The larger wafer size results in high quality and lower cost of unit design. We find that the area ratio of logic gates dramatically decreases and the gate count increases in SoC development. Under the assumptions of the reuse rate of circuits, the design manpower ratio, and the improvement of reuse overhead, the manpower involved in new designs steadily decreases with the size of the new designs increases. This advantage of SoC benefits from the design reuse methodology, which is also the prime advantage of the application-driven virtual microsensor platform.

The design reuse generally includes two levels of reuse, called the *source reuse* and the *integration-driven reuse* [118]. The source reuse is a high level design reuse among the design groups which directly reuses designs created elsewhere to achieve high productivity rates in new SoC designs. However, the source reuse is not very efficient in most designs, since users still need to understand and re-design the IP blocks according to the requirements of their

Table 4.3: Change of SoC design productivity. [108, 130]

Year	1999	2001-2002	2005-2006	2011-2012
Technology (um)	0.18	0.13	0.065	0.032
Wafer Size	/	300mm (12in)	/	450mm (18in)
(a) Area ratio of logic gates to SoC (%)	80	50	35	15
(b) Gate count (Mgates)	4	6.75	11.64	30.41
(c) Reuse rate of circuits (%)(Assumption)	20	50	70	90
(d) Design manpower ratio (Assumption: 30%/3 years)	1	0.7	0.49	0.24
(e) Improvement of reuse overhead (%) (Assumption: 30%/3 years)	50	35	24.5	12.01
Manpower				
New design parts of SoC = (a)×(b) (Mgates)	3.2	3.38	3.49	3.04
Substantial new design = (a)×(b)×(d) (Mgates)	3.2	2.37	1.71	0.73
Reuse resource= (b)×(c)×(e) (Mgates)	0.4	1.18	2	3.29
New design + reuse resource= (a)×(b)×(d) + (b)×(c)×(e) (Mgates)	3.6	3.55	3.71	4.02

own applications thereby making them usable in the new designs. The integration-driven reuse is a more effective design reuse which allows users to reuse the IP blocks without having to make changes on them. The integration-driven reuse is commonly based on an integration platform, which provides a SoC design environment that includes architectural specifications and pre-qualified IP blocks designed to work together on that platform. For example, Philips Semiconductors [101] employs a SoC integration platform, called *Nx-Builder* to reuse the fixed and the reconfigurable IPs. The virtual microsensor platform is also an example of the integration-driven reuse.

A typical SoC consists of [135]: (1) one or more micro-controller, microprocessor or DSP core(s); (2) memory blocks such as ROM, RAM, EEPROM and Flash; (3) timing sources including oscillators and phase-locked loops; (4) peripherals including counter-timers, real-time timers and power-on reset generators; (5) external interfaces with industry standards such as USB, FireWire, Ethernet, USART, SPI; (6) analog interfaces including ADCs and DACs; and (7) voltage regulators and power management circuits.

The final SoC can be developed from two approaches. One is to integrate the soft CPU cores such as Leon [64] that is an implementation of the full SPARC V8 standard core and ARM [10] that is a 32-bit embedded RISC microprocessor, the required components listed above, and all necessary IP blocks together in a single design before synthesis with target technologies. The SoC design is then synthesized on full-custom ASICs with the analog-digital mixed signal technology. Another approach is to directly use FPGAs that contain hardware CPU core(s) and pre-defined architectures. The necessary IP blocks are then included in the architectures, and the overall designs are synthesized and implemented on those FPGAs.

Due to the complexities and high development costs of the first approach, even large semiconductor companies have cooperated to co-develop SoC-based products. For instance, IBM, Toshiba and Sony have started working together to develop a new SoC processor architecture called *Cell processor* [113], which is expected to be used in Sony's Play Station 3 game con-

sole. For the easy development purpose, we focus on the second approach and introduce the XUP board developed by the Digilent company [50] as the target platform. The XUP board, as shown in Fig. 4.64, is embedded with a Xilinx Virtex II Pro FPGA that includes two PowerPC CPU cores.

The pre-defined architecture on the Virtex II pro FPGA for SoC design is demonstrated in Fig. 4.65. The symbol and the detailed internal structure of the PowerPC 405 CPU core that is embedded in the Virtex II Pro are shown in Figs. 4.66 and 4.67.

4.8 Summary

In this chapter, we presented the structure of the virtual microsensor platform, and focused on the digital computing section that includes the algorithm improvement using pipelined and parallel structures and the development of the image processing IP library. We developed the parallel ICA (pICA) algorithm, which distributes complex computations from one resource to multiple resources, as an algorithm improvement example to show how to provide the parallel version of complex image processing algorithms. In the image processing IP library, we developed IPs for the contrast stretching and the geometric correction as examples of the point-based processing, the 3×3 filter as an example of the neighborhood-based processing, and the parallel ICA (pICA) algorithm as the image-based processing. Through an example of microsensor integration, we demonstrated the fast design of application-specific microsensor. After integrating and synthesizing the selected image processing IPs, the microsensor design is eventually implemented on the prototyping FPGA or SoC. We presented the implementation procedure from the virtual microsensor platform to the final SoC, and introduced the Pilchard and the Amirix boards as two FPGA prototyping platforms, the XUP board that is embedded with a Xilinx Virtex II Pro FPGA with two PowerPC CPU cores as the target SoC platform. Before testing our designs on FPGAs and finalizing on SoC for physical microsensor implementation, we need to first evaluate the synthesis performance of our designs, which will be shown in the next chapter.

Figure 4.64: XUP development board.

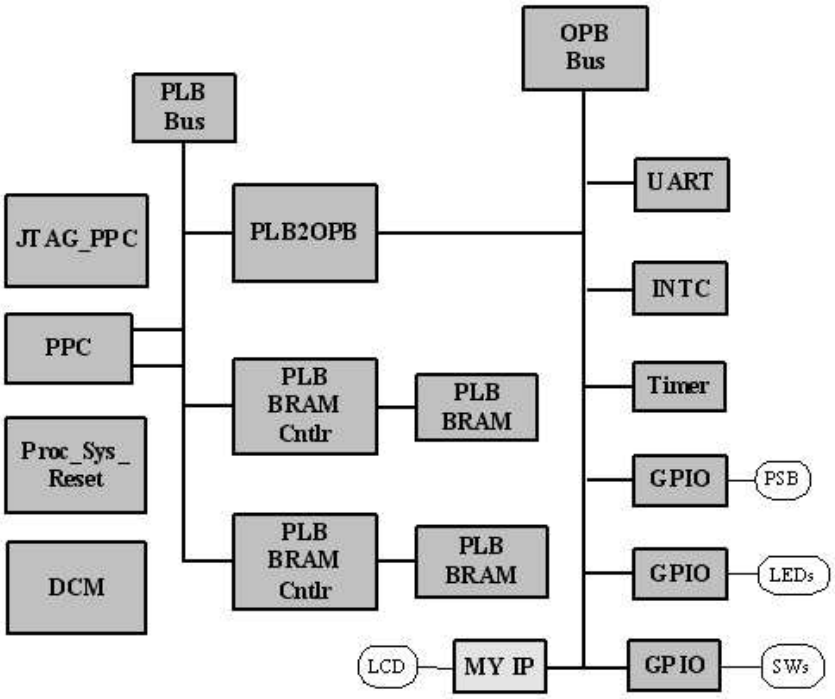


Figure 4.65: Architecture of SoC design.

Figure 4.66: Symbol of PowerPC 405.

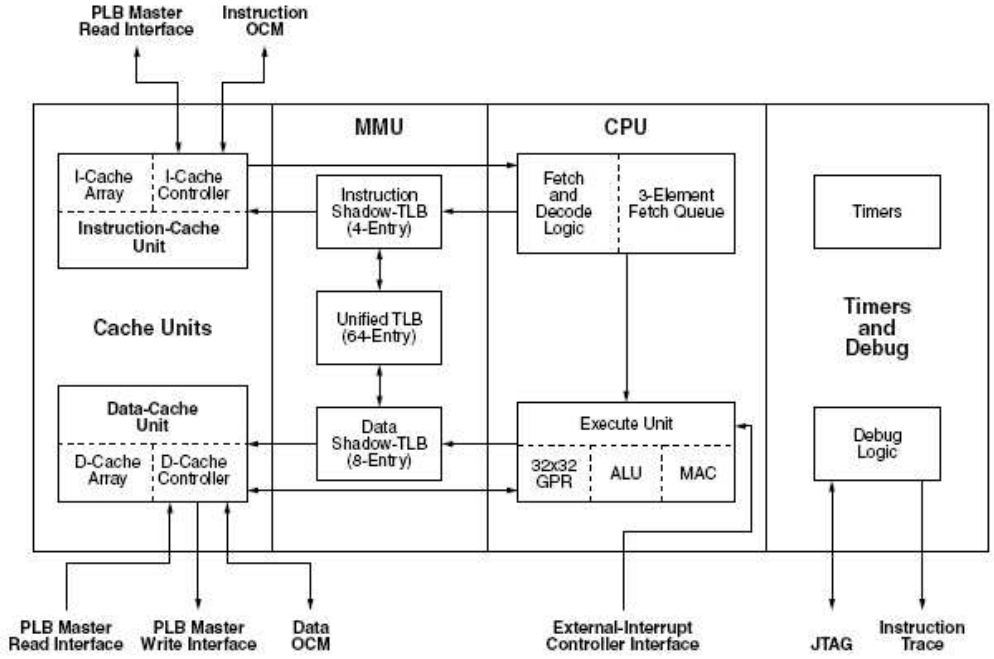


Figure 4.67: Internal structure of PowerPC 405.

Chapter 5

Experimental Results and Comparisons

According to discussions in Chapters 3 and 4, the image processing algorithms can be improved by pipelined and parallel processing, partitioned, clustered, and mapped for better load balancing and less communication, and finally implemented in the IP library for use by the virtual platform, then targeted at different prototyping FPGAs with specific technologies. In this chapter, four sets of experiments will be designed and developed to demonstrate the effectiveness of the algorithm improvement, function partitioning, clustering, and mapping, IP implementation, and microsensor integration.

First of all, we would like to show the advantages of algorithm improvement through pipelined and parallel computing. In Section 5.1, the proposed pICA algorithm is compared to the FastICA algorithm when applied to dimensionality reduction in hyperspectral image (HSI) analysis. The performance is evaluated from three perspectives, including the performance gain of parallelism, scalability, and effect of data size during communication.

Secondly, in Section 5.2, we evaluate the performance of the proposed component clustering algorithm and the cyclic process modeling discussed in Chapter 3. We also compare the load

attraction and the communication attraction algorithm, and their local refinement.

Thirdly, Section 5.3 demonstrates IP synthesis and implementation results on prototyping FPGAs. We implement four image processing algorithms, contrast stretching, polynomial approximation-based geometric correction, 3×3 filter, and pICA algorithm, on Xilinx Virtex II Pro FPGA to show the effectiveness of our IP designs. The geometric correction and the pICA designs are also implemented on the Xilinx Virtex 1000E FPGA in order to compare the performance to other existing approaches.

Finally, Section 5.4 shows the implementation results of one integration example described in Chapter 4 to validate the usage of the virtual microsensor platform in fast microsensor development.

5.1 Experiments for Algorithm Improvement

In order to evaluate performance and analyze impact factors, both the proposed pICA algorithm and the LogP based pICA performance prediction model are applied to the dimensionality reduction in HSI analysis. Unlike broadly used digital cameras, hyperspectral sensor systems provide images with hundreds of contiguous spectral bands. The high volume of information results in excessive computation burden. Since most materials have specific characteristics only at certain bands, a lot of this information is redundant. This property of hyperspectral images has motivated many researchers to study various dimensionality reduction algorithms, and ICA is one of the most popular techniques [55], which minimizes the statistical dependence between spectral bands and selects the most effective bands, therefore eliminating superfluous bands but retaining practical information given only the observations of hyperspectral images. In this experiment, we evaluate the significance of individual spectral bands by comparing the average magnitude of weight coefficients $\bar{R_j} = \frac{1}{m}\sum_{i=1}^{m}|w_{ij}|$, where $j = 1 \cdots n$, n denotes the dimensionality of the observed signal, and m denotes the dimensionality of the source signal. After sorting $\bar{R_j}$ for all spectral bands, we select the bands with the highest $\bar{R_j}$'s. Then a subset of the

original hyperspectral image is generated according to these selected bands, thereby achieving the purpose of dimensionality reduction.

5.1.1 Experiment Setup

All experiments in this section are conducted in an MPI environment with 10 computers. MPI is a message-passing library standard which extends the message-passing model of [131]. It specifies both point to point communication in the forms of various sending/receiving calls, collective communication calls, and the ability to define complex data types and virtual topologies of communications. Our MPI environment uses MPICH developed by the Argonne National Laboratory and Mississippi State University [51]. It consists of 10 computers whose system configuration parameters are listed in Table 5.1. As described in Chapter 4, the weight matrix is evenly divided into 10 sub-matrices ($k = 10$), each of which is processed on an individual computer, as demonstrated in Fig. 5.1. After performing the sub-matrix estimations and the internal decorrelations, Slaves 1, 3, 5, 7, and 9 respectively send their sub-matrices to Slaves 2, 4, 6, 8, and the Master. These computers then execute the corresponding external decorrelation processes, and fulfill the hierarchical processing as shown in Fig. 5.1. Finally, the Master collects all decorrelated weight vectors, compares the average magnitude of weight coefficients,

Table 5.1: MPI environment specification.

Computer	CPU	Memory
Master	Pentium 4 2.4 GHz	1 GB
Slave 1	Pentium III 1 GHz	256 MB
Slave 2	Pentium 4 2.4 GHz	384 MB
Slave 3	Pentium III 1 GHz	256 MB
Slave 4	Pentium III 1 GHz	1 GB
Slave 5	Pentium III 1 GHz	256 MB
Slave 6	Pentium 4 1.7 GHz	256 MB
Slave 7	Pentium III 866 MHz	128 MB
Slave 8	Pentium III 1 GHz	512 MB
Slave 9	Pentium III 550 MHz	384 MB

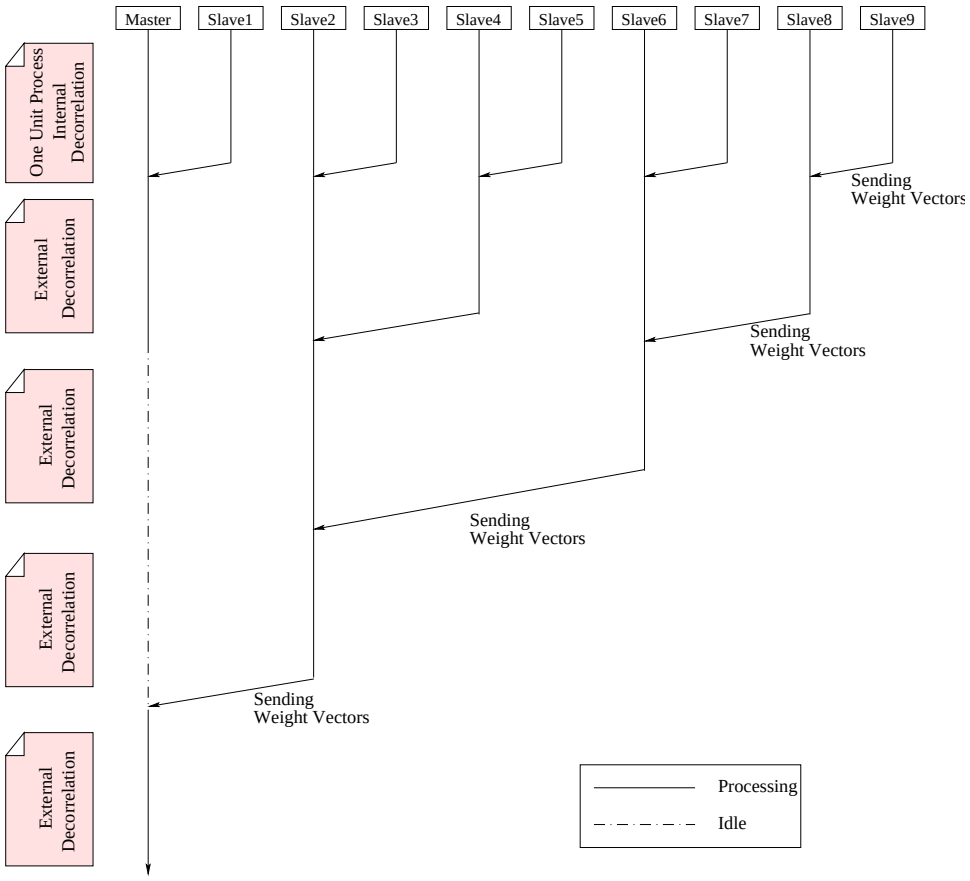
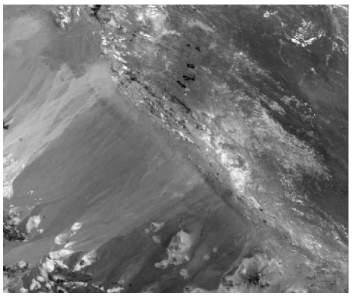


Figure 5.1: MPI diagram of pICA.

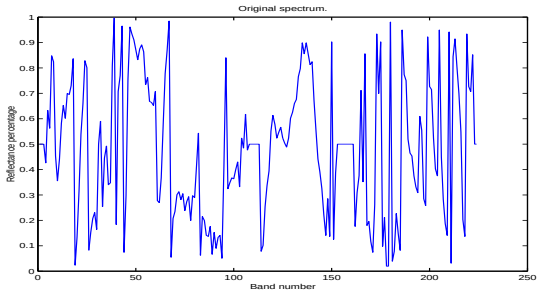
and outputs the most important spectral bands.

We take the NASA AVIRIS 224-band hyperspectral image (Fig. 5.2(a)) as our testing example [103], which was taken over the Lunar Crater Volcanic Field in Northern Nye County at Nevada. The size of this 614×512 hyperspectral image is 140.8Mb. Figure 5.2(b) shows the spectral profile of one pixel randomly selected from this image.

Three experiments are conducted to evaluate the performance of pICA from three perspectives, including the performance gain of parallelism, scalability, and the effect of data size during communication. In Experiment 1, we compare the performance of pICA and FastICA in weight matrix estimation with the number of weight vectors changing from 10 to 100. In this experiment, pICA is executed on the MPI framework with 10 computers, while FastICA is executed on a Pentium 4 2.4GHz computer with 1GB memory. In Experiment 2, we evaluate the scalability of the pICA algorithm by re-constructing the MPI environment with 2 to 10 computers respectively, and execute the estimation of weight matrices with 10, 20, 30, 40, and 50 weight vectors. In Experiment 3, we study the effect of the data size on communication in the pICA process by increasing the observation data from one-fifth (28MB) to the full-size hyperspectral image (140MB).



(a)

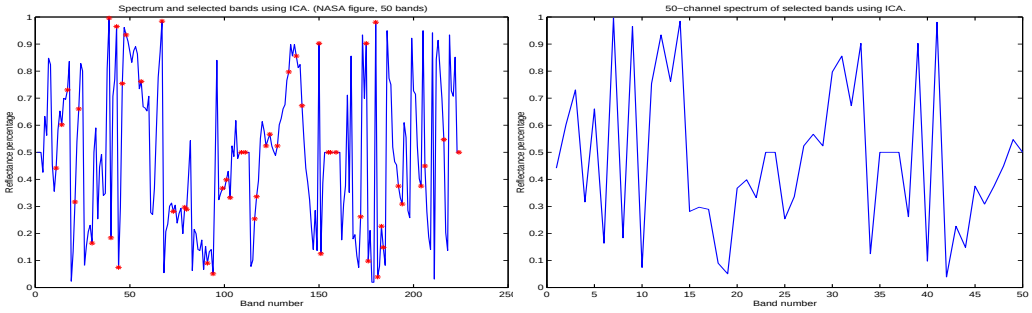


(b)

Figure 5.2: (a) The AVIRIS hyperspectral image scene [103]. (b) Original 224-band spectrum curve

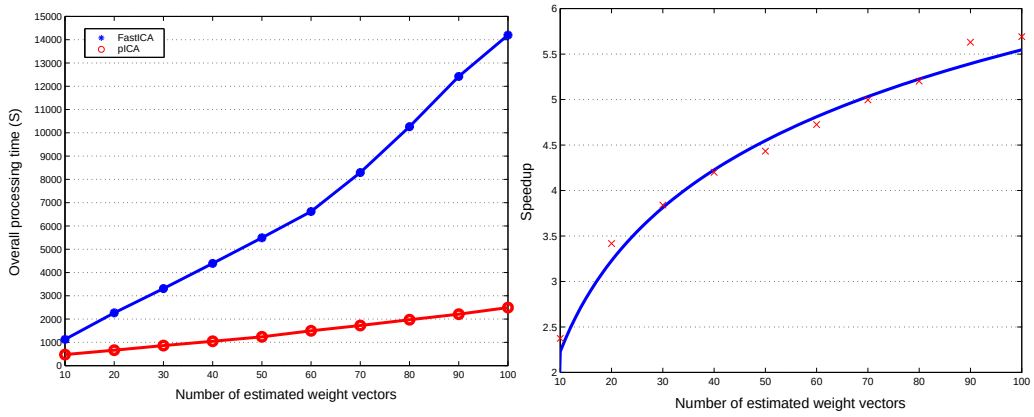
5.1.2 Effect of the Number of Estimated Weight Vectors

First we apply both the pICA and the FastICA algorithms to estimate the unmixing matrix **W** of 100 weight vectors, and then select 50 of the most important spectral bands for this image, thereby reducing the data set by 22.3%. As shown in Fig. 5.3(a), the selected 50 bands from the spectral profile contain the most important information that describes the original spectral curve, such as the maxima, the minima and the inflection points. We repeat these experiments by estimating the unmixing matrices in a range of 10 to 100 weight vectors. As the performance comparison in Fig. 5.4 shows, when the number of estimated weight vectors is linearly increased, the increase of the overall processing time for pICA is much slower than that for FastICA. The speedup of pICA, defined as the ratio between the processing time of FastICA and that of pICA, also steadily increases. The processing time of FastICA ranges from 1129.5 seconds for 10 weight vectors, to 14198.5 seconds for 100 weight vectors. Even though the pICA process running on 10 processors spends more time on communication, the overall processing time ranges from 475.5 seconds for 10 weight vectors to 2494.5 seconds for 100 weight vectors.



(a) (b)

Figure 5.3: (a) The selected 50 spectral bands. (b) Spectrum curve plotted by the selected 50 bands.



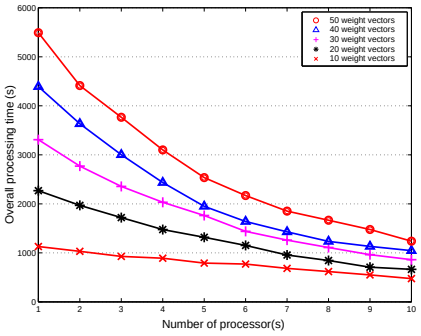
(a) Overall processing time.

(b) Speedup.

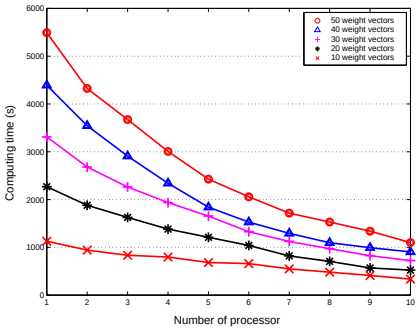
Figure 5.4: Performance comparison between FastICA and pICA (10 processors).

5.1.3 Effect of the Number of Processors

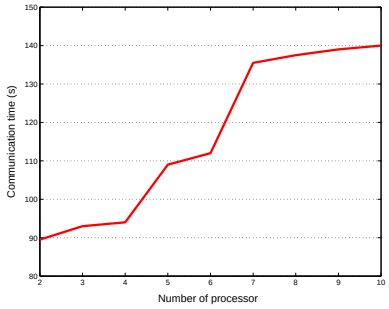
In the second experiment, we evaluate the scalability of the pICA algorithm by re-constructing the MPI environment with 2 to 10 computers respectively. Both the computing time and the communication time are separately recorded in each MPI environment. The computing time is defined as the sum of the slowest processing time among all processors for sub-matrix estimations on layer 1 and the slowest processing time for external decorrelations on other layers. The communication time is the sum of the slowest data transfer time between every two layers. Figure 5.5 illustrates the comparison in terms of the overall processing time, the computing time, and the communication time. For estimations of different numbers of weight vectors, the difference of the communication time is within 0.5 second which is very small. Therefore, we only plot one curve in Fig. 5.5(c) for communication. As can be observed, when we increase the number of processors, the computing time of weight matrix estimation decreases exponentially, while the communication time increases steadily. If the number of weight vectors in the weight matrix is reduced, the computing time correspondingly decreases, while the communication time changes very little.



(a) Overall.



(b) Computing.



(c) Communication.

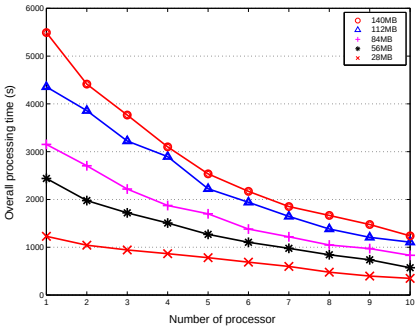
Figure 5.5: Scalability evaluation of pICA for different number of processors.

5.1.4 Effect of the Data Size

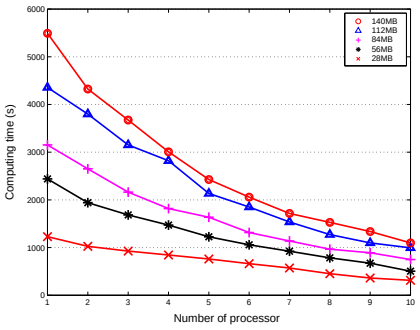
In the third experiment, we study the effect of data size on the communication during the pICA process. The input image size varies from 28MB to 140MB, with the number of processors changing from 1 to 10 in each case. The communication time, as well as the computing time on each data set, are measured and plotted in Fig. 5.6. It is easy to observe that when the size of the observation data set changes linearly, the communication time varies correspondingly. In other words, the communication time is mainly determined by the size of the hyperspectral image. In the meanwhile, we find that the communication time is far less than the computing time for any size of weight matrices. Hence, the parallel computing of the ICA algorithm is mainly a computation improvement problem, instead of a trade-off problem between computation on individual computer and communication between computers in the network.

5.1.5 Prediction Model Validation

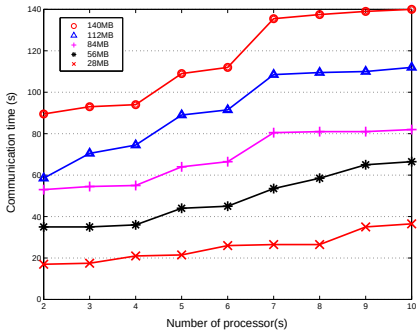
Finally, the effectiveness of the prediction model is examined by the comparison between the model prediction and real experiment on the estimation of 50 weight vectors. The detailed comparison is listed in Table 5.2. In the MPI environment, computers transfer data through the campus LAN connection with 2MB bandwidth, that is, the latency $L = 0.488ms$. The overhead o, the computing time that includes $t_{oneunit}$, t_{id}, and t_{id}, are respectively measured on the slowest computer at each layer. For example, in the 2-computer environment we first send the full hyperspectral image from the master to the slave at layer 1. The estimated communication time is 85.7 seconds. At layer 2, the slave sends the weight vectors back to the master, which takes 3.4 seconds. By adding t_{comp} and t_{comm} at all layers respectively, we obtain the model prediction. Figure 5.7 demonstrates the difference between the overall processing time predicted by the performance model and that from the experiments. In MPI environments with different numbers of processors, we observe that the model prediction is close to real experimental results. In other words, this model is effective to analyze the performance of pICA in



(a) Overall.



(b) Computing.



(c) Communication.

Figure 5.6: Performance evaluation of pICA for different size of observation data set.

Table 5.2: Comparison between the model prediction and experimental result.

Layer	Time (s)	Number of processors								
		2	3	4	5	6	7	8	9	10
1	$t_{oneunit}$ + t_{id}	3746	2563	2015	1494	1071	805	677	653	549
	t_{comm}	85.7	87.4	88.2	101.4	103.6	123.5	123.8	126.8	127.1
	o (ms)	0.058	0.062	0.065	0.112	0.12	0.191	0.193	0.203	0.204
	h_{data}	140	140	140	140	140	140	140	140	140
2	t_{ed}	557	481	348	291	225	182	135	87	83
	t_{comm}	3.4	2.2	1.7	1.6	1.3	1.2	1.2	1.0	1.0
	o (ms)	0.058	0.062	0.065	0.112	0.12	0.191	0.193	0.203	0.204
	h_{wv}	5.47	3.5	2.63	2.19	1.75	1.53	1.31	1.09	1.09
3	t_{ed}	/	481	697	582	450	364	270	174	165
	t_{comm}	/	2.2	3.3	3.2	2.6	2.4	2.3	2.0	2.0
	o (ms)	/	0.062	0.065	0.112	0.12	0.191	0.193	0.203	0.204
	h_{wv}	/	3.5	5.25	4.38	3.5	3.06	2.63	2.19	2.19
4	t_{ed}	/	/	/	291	450	546	540	348	330
	t_{comm}	/	/	/	1.6	2.6	4.6	4.6	4.0	4.0
	o (ms)	/	/	/	0.112	0.12	0.191	0.193	0.203	0.204
	h_{wv}	/	/	/	2.19	3.5	4.59	5.25	4.38	4.38
5	t_{ed}	/	/	/	/	/	/	/	87	165
	t_{comm}	/	/	/	/	/	/	/	1.0	2.0
	o (ms)	/	/	/	/	/	/	/	0.203	0.204
	h_{wv}	/	/	/	/	/	/	/	1.09	2.19
Model	t_{comp}	4303	3525	3060	2658	2196	1897	1622	1349	1292
	t_{comm}	89.1	91.8	93.2	107.8	110.1	131.7	131.9	134.8	136.1
	t_{all}	4392.1	3616.8	3153.2	2765.8	2306.1	2028.7	1753.9	1483.8	1428.1
Experiment	t_{comp}	4324.2	3673.8	3006.9	2426.6	2058.1	1716.1	1529.2	1337.5	1099.2
	t_{comm}	90	93.5	94.5	109.5	112.5	136	138	139.5	140.5
	t_{all}	4414.2	3767.3	3101.4	2536.1	2170.6	1852.1	1667.2	1477	1239.7

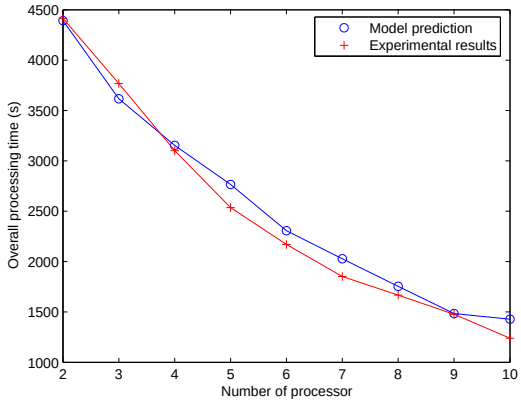


Figure 5.7: Overall processing time comparison between model prediction and experimental results.

different parallel computing environments.

5.2 Experiments for Function Clustering and Mapping

In this section, we first demonstrate the experiments conducted on the component clustering algorithm, followed by the experiments of function mapping in homogeneous and heterogeneous environments. For the heterogeneous environment, we compare the load attraction, the communication attraction mapping algorithms, and their local refinements. Finally, we evaluate the cyclic process modeling by comparing different mapping results on the pICA algorithm.

As the framework of task partitioning shown in Fig. 5.8, the performance of partitioning can be affected by two resources, namely, the task requirements, the resource provided, the clustering algorithm used, and the mapping algorithm used.

In the first set of experiments, the main goal is to evaluate different component clustering algorithms. Three tasks have been selected, including contrast stretching (an example of using parallel structure), 3×3 filtering (an example of using pipelined and parallel structure), and 7×7 filtering. Seven existing mapping schemes described in Chapter 3 have been developed and will be applied to map the operations to the resources, including linear algorithm, linear

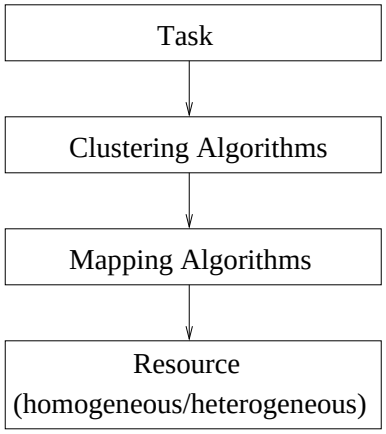


Figure 5.8: Framework of task partitioning.

algorithm with K-L refinement, scattered algorithm, scattered algorithm with K-L refinement, spectral algorithm, spectral algorithm with K-L refinement, and multi-level K-L algorithm. Two types of resource maps will be used, where the contrast stretching task uses 6 homogeneous processors and the filtering task uses 5 homogeneous processors. Since we will use 10 parallel processes in the contrast stretching, 6 processors, instead of 5 processors, are used in order to avoid even mapping. With all these different setups, we will have a thorough investigation on the performance of the proposed component clustering algorithm and existing ones. The following list itemizes the different experiments carried out in the first group, which will be detailed in Sec. 5.2.1:

- Experiment 1.1: Implement the contrast stretching on 6 homogeneous resources using the 7 existing mapping schemes with and without the proposed component clustering
- Experiment 1.2: Implement the 3×3 filtering on 5 homogeneous resources using the 7 existing mapping schemes with and without the proposed component clustering
- Experiment 1.3: Implement the 7×7 filtering on 5 homogeneous resources using the 7 existing mapping schemes with and without the proposed component clustering and two

other coarser clustering models

The performance will be measured using two metrics in this set of experiments, including the load weight variance and the cut weight. The load weight variance measures the difference of load weight between computing resources. The cut weight measures the overall communications on all communication resources.

In the second set of experiments, the main goal is to evaluate the performance of the proposed mapping schemes, including the load attraction algorithm, the communication attraction algorithm, and the corresponding local refinement, on *heterogeneous* resources. Since from the first set of experiments, we have already concluded that the proposed component clustering algorithm performs better than existing ones in terms of load variance and cut weight, in this set of experiments, we will fix the clustering algorithm to the proposed one. In addition, we will only implement the 7×7 filter in the first seven experiments, and implement the pICA task in the last experiment. The proposed load attraction algorithm, the communication attraction algorithm, and the corresponding local refinement will be compared with five existing mapping schemes [23], including opportunistic load balancing (OLB), minimum execution time (MET), Min-Min, Max-Min, and Duplex that will be described in Sec. 5.2.2. Different types of resource maps will be randomly generated. In order to provide reference point for comparison, we use the brute-force technique to manually find out the optimal mapping. The following list itemizes the different experiments carried out in the second group, which will be detailed in Sec. 5.2.2:

- Experiment 2.1 (Effect of the proposed mapping): Implement the 7×7 filtering on 5 heterogeneous resources using the load attraction algorithm, the communication attraction algorithm, and the 5 existing mapping schemes mentioned above. The results will be compared to the optimal mapping

- Experiment 2.2 (Effect of different numbers of resources): Implement the 7×7 filtering on 3 to 8 heterogeneous resources using the load attraction algorithm, the communication attraction algorithm, and the 5 existing mapping schemes

- Experiment 2.3 (Scalability of the processing speed): Implement the 7×7 filtering on 5 heterogeneous resources with different processing speeds using the load attraction algorithm and the communication attraction algorithm
- Experiment 2.4 (Scalability of the transmission speed): Implement the 7×7 filtering on 5 heterogeneous resources with different transmission speeds using the load attraction algorithm and the communication attraction algorithm
- Experiment 2.5 (Scalability of the component weight): Implement the 7×7 filtering with different component weights on 5 heterogeneous resources using the load attraction algorithm and the communication attraction algorithm
- Experiment 2.6 (Scalability of the edge weight): Implement the 7×7 filtering with different edge weights on 5 heterogeneous resources using the load attraction algorithm and the communication attraction algorithm
- Experiment 2.7 (Effect of the local refinement): Implement the 7×7 filtering on 5 heterogeneous resources using the load attraction algorithm and the communication attraction algorithm with or without the local refinement
- Experiment 2.8 (Mapping of complex task): Implement the pICA algorithm on 10 heterogeneous resources using the load attraction algorithm with and without the local refinement

Since the communication time in the pICA algorithm is only a very small portion in the overall processing time as we analyzed in Chapter 4, we only compare the load attraction algorithm with and without the local refinement in Experiment 2.8 and the next set of experiments. The performance will be measured using four metrics in this set of experiments, including the load weight variance, the cut weight, the overall processing cost in time, and the processing time. The overall processing cost in time is the sum of the processing time on all computing resources. The processing time measures the time consumption to complete a task.

In the third set of experiments, the main goal is to evaluate the performance of the proposed cyclic process modeling. We fix on one task, the pICA algorithm, in this set of experiments. Different types of resource maps will be generated from Table 5.1. The following list itemizes the different experiments carried out in the third group, which will be detailed in Sec. 5.2.3:

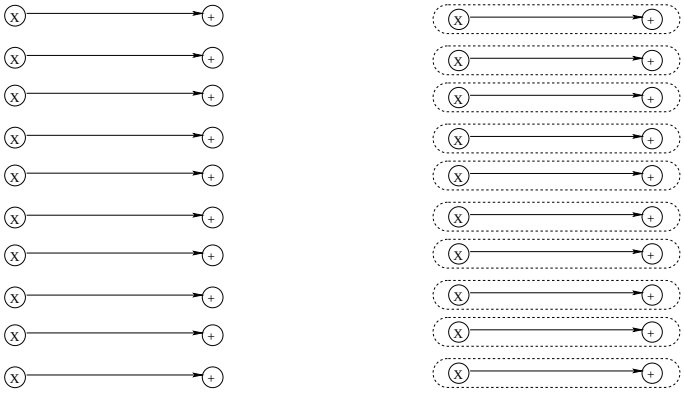
- Experiment 3.1 (Effect of the cyclic process modeling): Implement pICA with and without the cyclic process modeling on 10 heterogeneous resources using the load attraction algorithm with and without the local refinement
- Experiment 3.2 (Scalability of the computing resource): Implement pICA with and without the cyclic process modeling on 1 to 10 heterogeneous resources using the load attraction algorithm with and without the local refinement
- Experiment 3.3 (Effect of the pre-defined number of iterations): Implement pICA with different pre-defined number of iterations on 10 heterogeneous resources using the load attraction algorithm with and without the local refinement

The performance will be measured using five metrics in this set of experiments, including the load weight variance, the cut weight, the overall processing cost in time, the processing time, and the mapping time. The mapping time is the time cost of the mapping process. Since the mapping time is ignorable for simple tasks but significant for complex tasks, we only compare the mapping time in this set of experiments.

5.2.1 Component Clustering Algorithm Evaluation

We conduct three experiments to show the performance of the proposed component clustering algorithm.

In the first experiment (Experiment 1.1), contrast stretching is implemented on 6 homogeneous resources using the 7 existing mapping schemes with and without the proposed component clustering. Figure 5.9 shows the original model (without clustering) and the component



(a) Original model.

(b) Component clustering result.

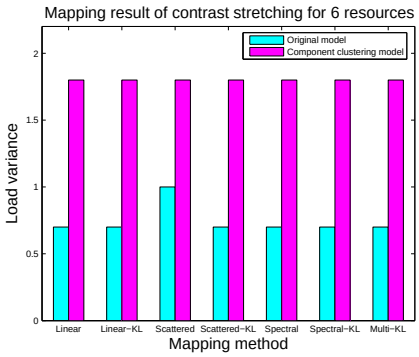
Figure 5.9: Function models to implement contrast stretching.

clustering result of contrast stretching with 10 processes running in parallel.

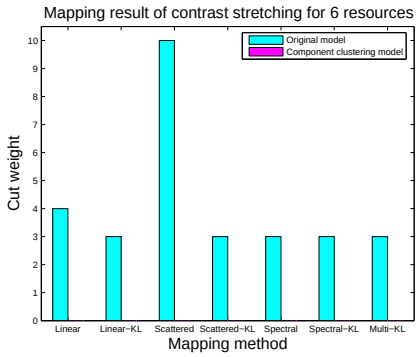
Figure 5.10 shows the mapping result comparisons on these two models. Although the load variances on the component clustering model are not improved, the cut weights are decreased to zero.

In the second experiment (Experiment 1.2), the 3×3 spatial filter is implemented on 5 homogeneous resources using the 7 existing mapping schemes with and without the proposed component clustering. Figure 3.10 in Chapter 3 has shown the original model (without component clustering) and the component clustering result of the 3×3 filter. Figure 5.11 shows the mapping result comparisons on these two models. As we observe, the component clustering model in most time improves the performance on either load variance or cut weight, or both.

In order to extensively evaluate the component clustering algorithm, in the third experiment (Experiment 1.3), we evaluate the 7×7 spatial filter implemented on, again, 5 homogeneous resources using all the 7 existing mapping schemes. In this experiment, we evaluate four granularities, the original model which does not perform any component clustering, the model generated using the proposed component clustering, and two other clustering models with coarser granu-

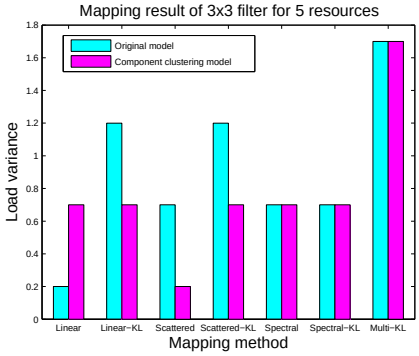


(a) Load variance.

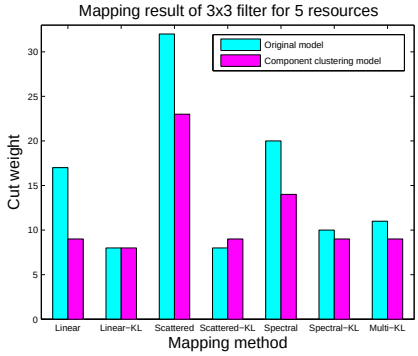


(b) Cut weight.

Figure 5.10: Mapping result comparisons of two models for contrast stretching.



(a) Load variance.



(b) Cut weight.

Figure 5.11: Mapping result comparison of two models for 3×3 filter.

larities, as shown in Figs. 5.12 and 5.13. Figure 5.14 shows the mapping result comparisons on these four models. Obviously, the component clustering model performs the best compared to other models measured by both the load variance and the cut weight. In other words, if the granularity of a model is too small or too large, the mapping result will be definitely affected. This comparison also shows that the component clustering algorithm provides function models with appropriate granularity to the mapping process.

5.2.2 Mapping in Heterogeneous Environment

In this set of experiments, we evaluate the proposed load attraction algorithm, the communication attraction algorithm, and the corresponding K-L local refinement for function mapping in heterogeneous environment.

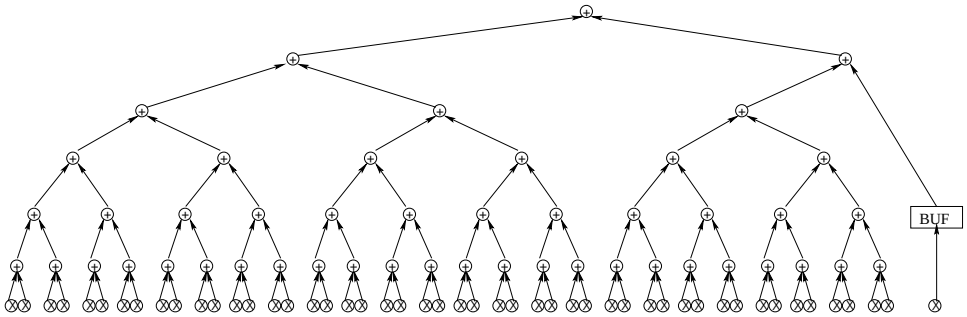
Experimental Environment Setup

In this set of experiments, we use the 7×7 filter as an image processing application, and compare mapping results of different algorithms on it. As we previously discussed, the component clustering algorithm provides function model with appropriate granularity. Therefore, we directly use the previous result shown in Fig. 5.12.

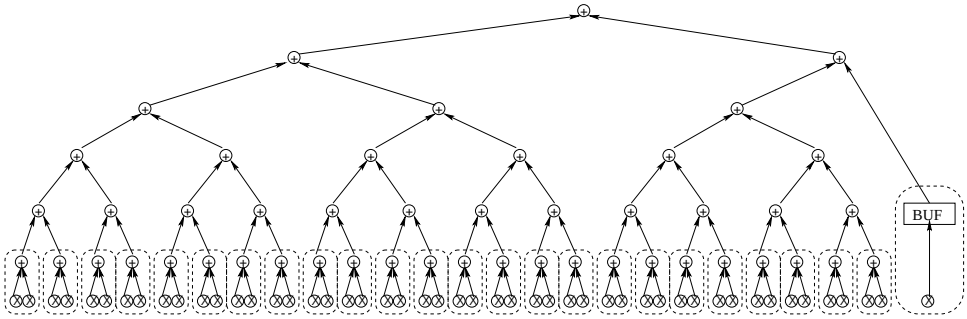
The heterogeneous computing environment we initially set up includes 5 computing resources and bi-directional transmissions between each pair of resources. The computing and communication capabilities are randomly set to resources and connections, as shown in Fig. 5.15.

Effect of the Proposed Mapping Algorithms

In the first experiment (Experiment 2.1), the 7×7 filtering is implemented on 5 heterogeneous resources using the load attraction algorithm, the communication attraction algorithm, and the 5 existing mapping schemes, and compare their performance with the optimal mapping.

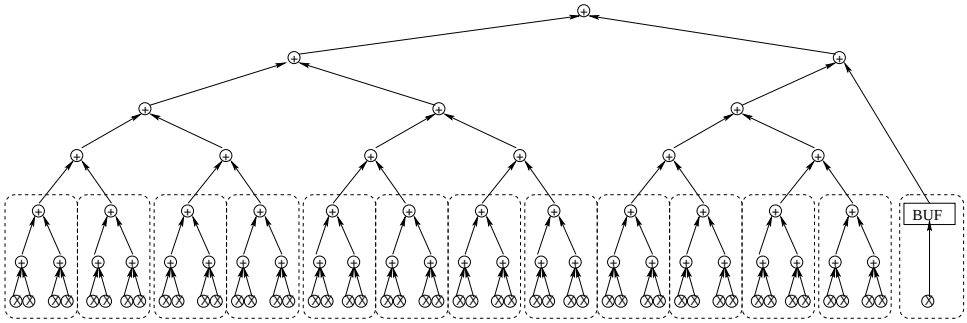


(a) Original model.

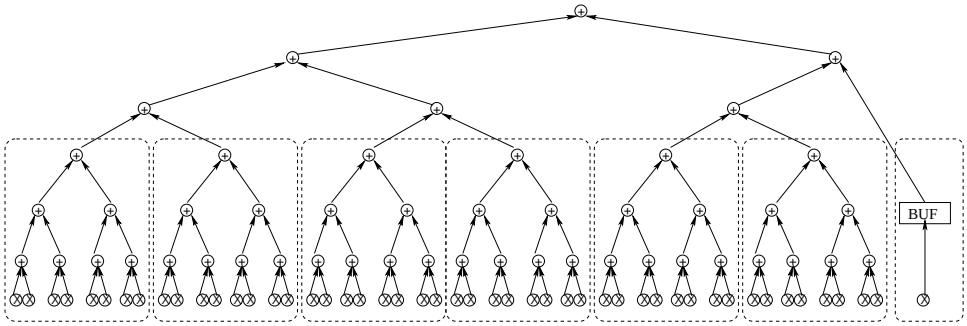


(b) Component clustering result.

Figure 5.12: Function models of the 7×7 filter.

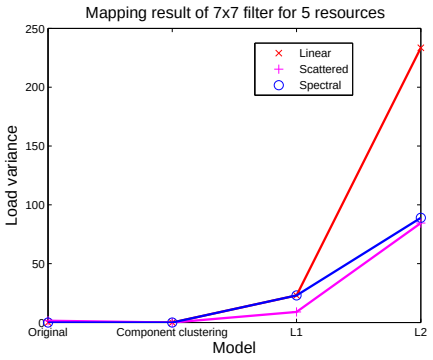


(a) Model with coarse granularity (L1).

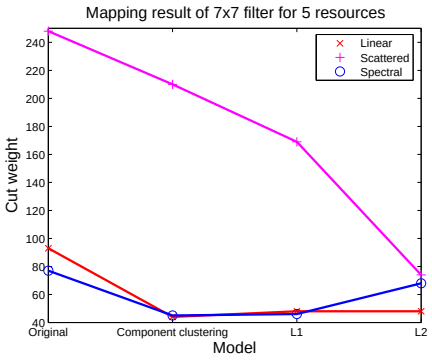


(b) Model with coarser granularity (L2).

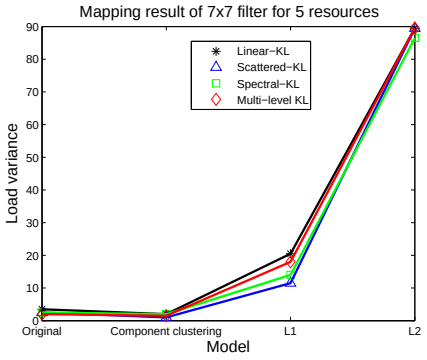
Figure 5.13: Function models of the 7×7 filter with coarser granularities.



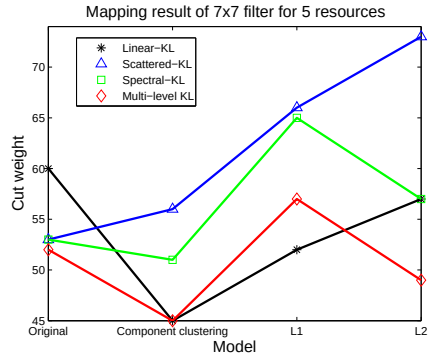
(a) Load variance.



(b) Cut weight.



(c) Load variance.



(d) Cut weight.

Figure 5.14: Mapping result comparison for the 7×7 filter models with different granularities.

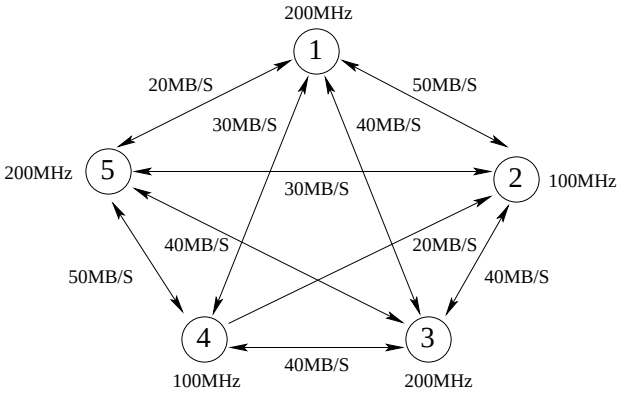


Figure 5.15: Computing resources in heterogeneous environment.

First of all, we use the brute-force technique to manually find out the optimal mapping to provide reference point for comparison.

Secondly, we extensively compare the load attraction and the communication attraction algorithms with some existing mapping approaches for heterogeneous environments [23], including opportunistic load balancing (OLB), minimum execution time (MET), Min-Min, Max-Min, and Duplex.

In OLB, components or edges are assigned to resources in an arbitrary order, regardless of the weights of components or edges [11, 62]. One advantage of OLB is its simplicity. But OLB may result in very poor load balance or a lot of communications depending on the granularity of the function model and the order of the components or the edges.

Compared to OLB, MET arbitrarily assigns components or edges to resources with the shortest computing or communication time for the assigned components or edges [11, 62]. Although MET is to assign each component or edge to the most suitable resource, it causes a severe load imbalance or large communication across resources.

The Min-Min algorithm first sorts components or edges in ascending order [11, 77]. Then a component or edge is assigned to the resource with the minimum computing or communication time, hence the name Min-Min. The assignment process repeats until all components are

mapped. Min-Min maps components or edges in the order that changes resource status by the least amount.

The Max-Min algorithm is similar to Min-Min. Max-Min also sorts components or edges in the ascending order, but assigns a component or edge to the resource with the maximum computing or communication time [11, 77]. The idea of Max-Min is to minimize the penalties caused by mapping components or edges with larger weights.

The Duplex algorithm is a combination of Min-Min and Max-Min [11, 62]. Duplex performs both Min-Min and Max-Min mappings and then selects the better solution.

First, we apply these mapping algorithms and the load attraction mapping to the function model of 7×7 filter. Table 5.3 shows the comparison of load variances between the optimal mapping, existing approaches, and the proposed load attraction mapping algorithm. We find that the load variance of the load attraction mapping algorithm is very close to that of the optimal mapping, and significantly smaller compared to other existing mapping algorithms.

Secondly, we compare the cut weights of the optimal mapping, existing mapping approaches, and the proposed communication attraction mapping algorithm, as shown in Table 5.4. Compared to other existing mapping approaches, the communication attraction algorithm has the minimum cut weight among existing algorithms and is the closest to that of the optimal mapping.

Table 5.3: Comparison of load variances of different mapping algorithms on the 7×7 filter.

Algorithm	Load variance
Optimal	1.75e-5
OLB	92e-5
MET	20e-5
Min-Min	18e-5
Max-Min	45000e-5
Duplex	18e-5
Load attraction	3.0e-5

Table 5.4: Comparison of cut weights of different mapping algorithms on the 7×7 filter.

Algorithm	Cut weight (us)
Optimal	5.15
OLB	7.4933
MET	7.72
Min-Min	6.9333
Max-Min	9.8
Duplex	6.9333
Communication attraction	6.6567

The above comparisons show that the load attraction and the communication attraction algorithms are very effective in function mapping for heterogeneous environment, and perform better than most existing mapping heuristics.

Effect of Different Numbers of Resources

In order to extensively evaluate the performances of the load attraction and the communication attraction mapping algorithms, we change the topology of the resource graph in this experiment (Experiment 2.2) by increasing the number of resources from 3 to 8, and implement the 7×7 filtering using the load attraction algorithm, the communication attraction algorithm, and the 5 existing mapping schemes. For each case, we conduct 50 experiments and calculate the mean and the standard deviation (STD) of load variances and cut weights. For each experiment, we randomly select the corresponding number of resources and randomly set the connections to be on or off.

Table 5.5 compares the mean of load variances of different mapping algorithms on different topologies of the resource graph. Table 5.6 lists the STD of load variances of these mapping algorithms on different topologies. Obviously, the proposed load attraction mapping algorithm has not only the minimum mean but also the minimum STD of load variances. In other words, the load attraction algorithm performs the best on function mapping for different topologies

Table 5.5: Mean of load variances on different topologies of the resource graph.

Algorithm	3	4	5	6	7	8
OLB	8e-4	8e-4	8e-4	9e-4	1e-3	1.2e-3
MET	8.404e-5	7.857e-5	6.13e-5	5.3e-5	5.393e-5	3.806e-5
Min-Min	3.807e-4	2.39e-4	3.075e-4	2.856e-4	3.201e-4	1.346e-4
Max-Min	0.7539	0.6252	0.5434	0.5058	0.4588	0.4395
Duplex	3.807e-4	2.39e-4	3.075e-4	2.856e-4	3.201e-4	1.346e-4
Load attraction	1.408e-5	6.58e-6	1.261e-5	1.059e-5	1.315e-5	6.7e-7

Table 5.6: STD of load variances on different topologies of the resource graph.

Algorithm	3	4	5	6	7	8
OLB	3.633e-4	2.923e-4	3.574e-4	2.464e-4	3.927e-4	4.017e-4
MET	5.688e-5	5.773e-5	4.383e-5	4.831e-5	3.987e-5	2.904e-5
Min-Min	3.694e-4	2.188e-4	1.74e-4	9.1e-5	6.6e-5	3.1e-6
Max-Min	0.2444	0.1402	0.1278	0.1034	0.078	0.052
Duplex	3.694e-4	2.188e-4	1.74e-4	9.1e-5	6.6e-5	3.1e-6
Load attraction	1.291e-5	5.99e-6	7.21e-6	6.28e-6	1.104e-5	2.3e-7

in heterogeneous environments. Meanwhile, it is more stable than other existing mapping approaches.

Table 5.7 compares the mean of cut weights of different mapping algorithms on different topologies of the resource graph, and Table 5.8 lists the STD of cut weights of these mapping algorithms on different topologies. We find that the proposed communication attraction mapping algorithm has both the minimum mean and the minimum STD of load variances in 50 experiments for different topologies. That means the communication attraction algorithm is very effective in minimizing communication for different topologies in heterogeneous environments. In addition, it is more stable than other existing mapping approaches.

Scalability of the Processing Speed

In the third experiment (Experiment 2.3), the 7×7 filtering is implemented on 5 heterogeneous resources with different processing speeds using the load attraction algorithm and the communication attraction algorithm. The purpose is to show the mapping performance of the load attraction and the communication attraction mapping algorithms in heterogeneous environments with different computing resources.

We keep the original setting of transmission speeds shown in Fig. 5.15 and randomly change the processing speeds of individual resources. We limit the range of the uniformly distributed random function into $[1, 10]$, and conduct the experiment for 100 times with 10MHz increment. The mean and STD of computing resources in each case, and the performance comparisons between the load and the communication attraction algorithms are shown in Fig. 5.16. The load variance of the load attraction algorithm is better than that of the communication attraction algorithm. But the difference on the overall processing cost in time is not obvious.

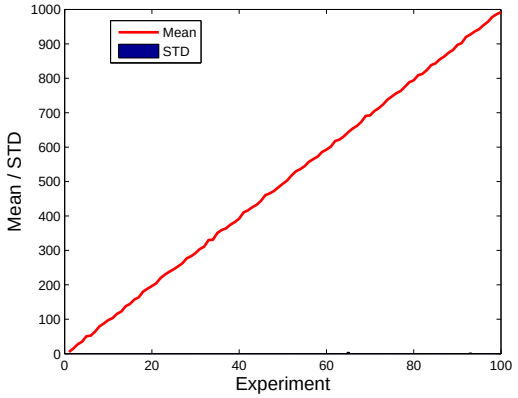
In order to further compare and analyze the performance of the load and the communication attraction algorithms, we expand the range of the uniformly distributed random function into $[1, 1000]$MHz. Then we repeat the experiment 100 times, and plot mean and STD of computing

Table 5.7: Mean of cut weights on different topologies of the resource graph.

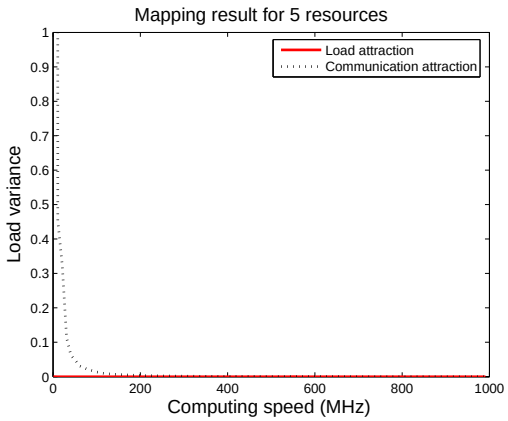
Algorithm	3	4	5	6	7	8
OLB	9.346	8.6295	8.225	7.8067	7.1461	6.7764
MET	9.5127	8.7107	7.8478	8.2067	7.2823	6.3257
Min-Min	10.4898	8.4637	8.3562	8.19	7.7354	7.3523
Max-Min	9.658	10.271	10.3092	10.054	8.4253	8.0039
Duplex	8.6984	8.3877	8.3342	8.19	7.6033	7.2795
Communication attraction	7.0957	5.841	7.242	7.3557	7.0622	6.0945

Table 5.8: STD of cut weights on different topologies of the resource graph.

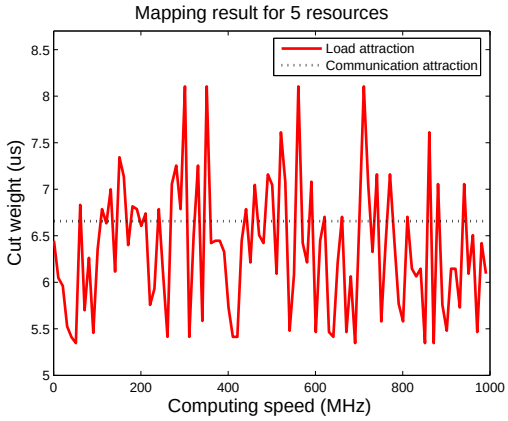
Algorithm	3	4	5	6	7	8
OLB	2.5317	1.4333	0.7016	0.4104	0.3974	0.3205
MET	3.9489	1.3818	1.0946	0.5925	0.9069	0.3548
Min-Min	3.1067	1.2087	0.8668	0.4308	0.4629	0.3399
Max-Min	5.0539	4.3175	1.9959	0.5046	1.1353	0.5161
Duplex	4.3917	1.2036	0.8983	0.4308	0.6611	0.2651
Communication attraction	1.9307	0.869	0.5183	0.405	0.3242	0.0879



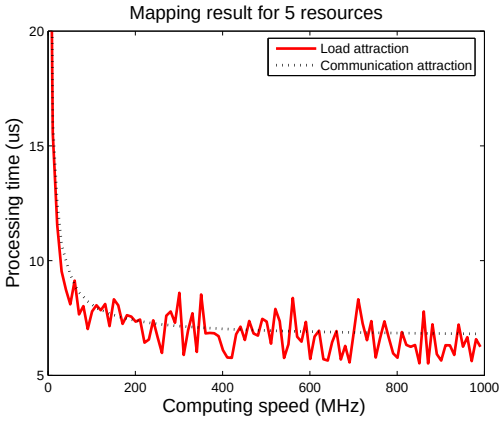
(a) Mean and STD of computing resources.



(b) Load variance.



(c) Cut weight.



(d) Overall processing cost in time.

Figure 5.16: Performances on random computing resources (10MHz increment).

resources in each case in Fig. 5.17(a). The performance comparisons between the two mapping algorithms are shown in Fig. 5.17. The mean and STD of load variance, cut weight, and processing time of these experiments are listed in Table 5.9. We find that the load attraction algorithm performs much better than the communication attraction algorithm on load variance, because the former takes the difference of computing capabilities into consideration and balances the computing time on individual resources. Since the difference between computing resources is more significant than that in the previous experiment, the overall processing cost of the load attraction is lower than that of the communication attraction.

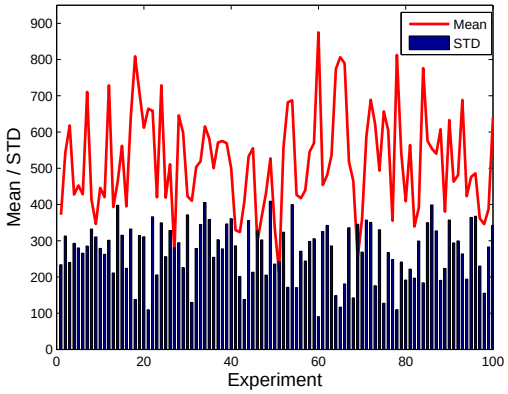
From these two experiments we find that the load attraction algorithm gives better mapping results if the variance between computing resources is larger. On the other hand, these two experiments also show that if the computing resources and their topology approach to homogeneous, we may ignore the considerations of resource and directly map functions given only the number of resources.

Scalability of the Transmission Speed

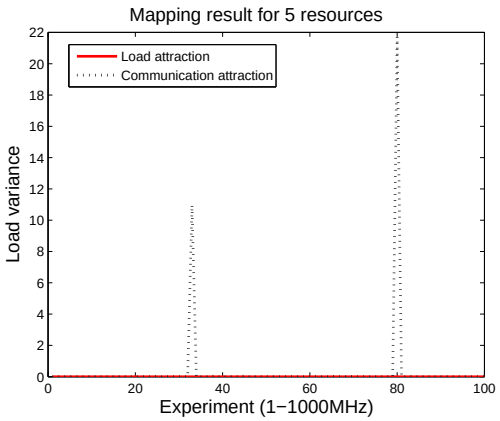
In the fourth experiment (Experiment 2.4), the 7×7 filtering is implemented on 5 heterogeneous resources with different transmission speeds using the load attraction algorithm and the communication attraction algorithm. The purpose is to show the mapping performance of the load

Table 5.9: Mean and STD of experiments on random computing resources.

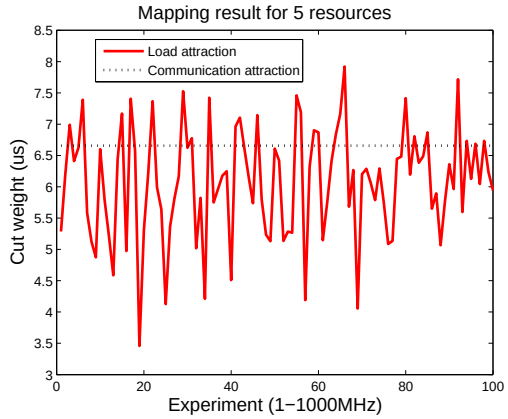
Algorithm	Mean (us)	STD
Load attraction		
Load variance	0.102835	0.171450
Cut weight	6.077267	0.884060
Overall processing cost in time	6.385405	0.859387
Communication attraction		
Load variance	7.799774	21.187610
Cut weight	6.656667	0.000000
Overall processing cost in time	7.902424	2.674793



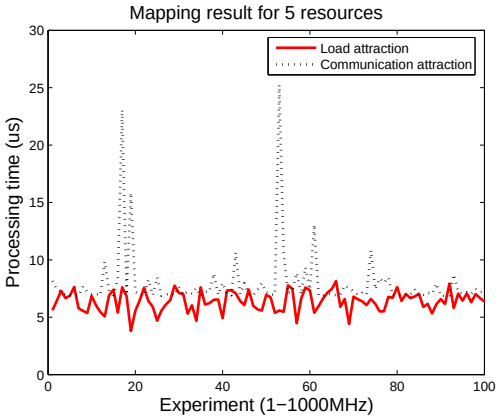
(a) Mean and STD of computing resources.



(b) Load variance.



(c) Cut weight.



(d) Overall processing cost in time.

Figure 5.17: Performances on random computing resources (1-1000MHz).

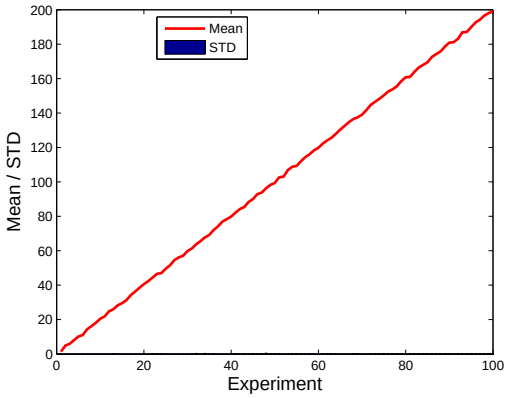
attraction and the communication attraction mapping algorithms in heterogeneous environments with different communication resources.

We keep the original setting of computing resources and randomly change the transmission speed between a pair of nodes. We first limit the range of the random number to $[1, 2]$, and conduct the experiment for 100 times with 2MB/s increment. Figure 5.18 shows the mean and STD of communication resources in each case and the performance comparisons between the two algorithms. We can see that the cut weight and the overall processing cost in time of the communication attraction algorithm are a little lower than those of the load attraction algorithm.

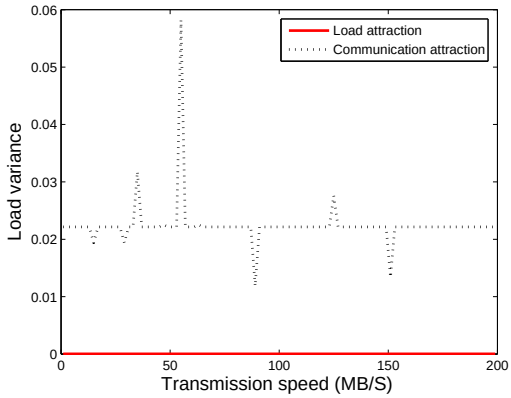
In the next experiment, the transmission speed falls in the range of $[1, 200]$MB/s with uniform distribution. Similar to the experiments on computing resource, we repeat the experiment 100 times. The mean and STD of communication resources in each case and the performance comparisons between the two mapping algorithms are shown in Fig. 5.19. The mean and STD of load variance, cut weight, and overall processing cost of these experiments are listed in Table 5.10. Since the communication attraction algorithm gives consideration to the difference on transmission speeds, its cut weight is much lower than that of the load attraction algorithm in most cases. Because the difference between communication resources are significant, the communication attraction algorithm always has lower overall processing cost than the load attraction algorithm. Although the load variance of the load attraction is better than that of the

Table 5.10: Mean and STD of experiments on random communication resources.

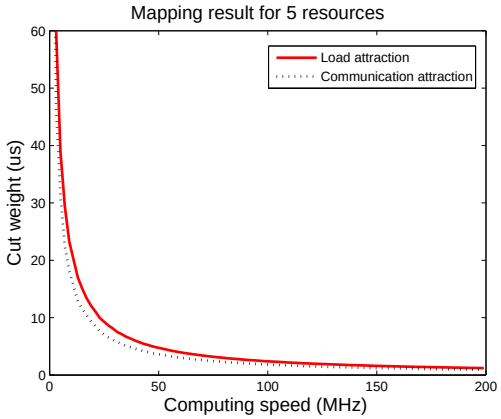
Algorithm	Mean	STD
Load attraction		
Load variance	0.120000	0.000000
Cut weight	6.359057	6.770439
Overall processing cost in time	7.289057	6.770439
Communication attraction		
Load variance	2.734600	0.900935
Cut weight	2.411844	0.415354
Overall processing cost in time	3.466744	0.430632



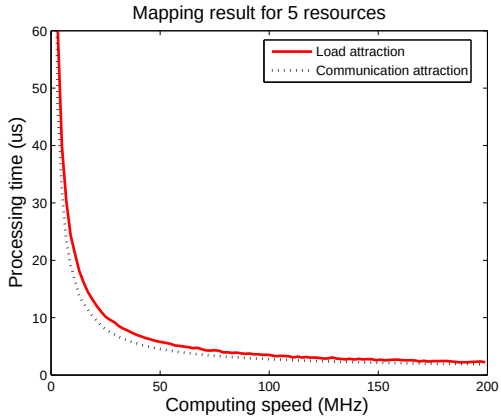
(a) Mean and STD of communication resources.



(b) Load variance.

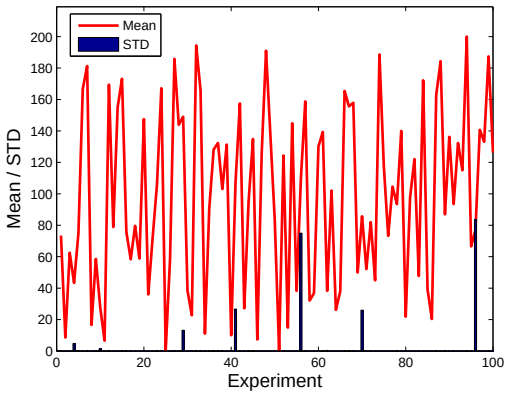


(c) Cut weight.

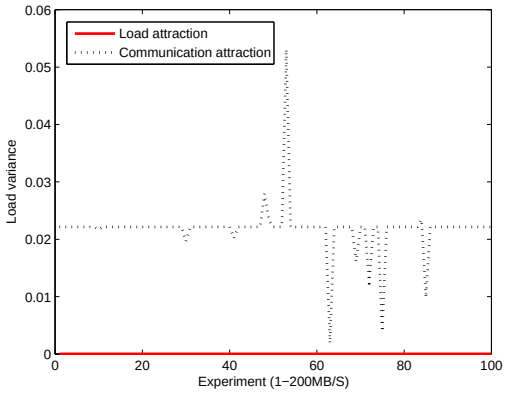


(d) Overall processing cost in time.

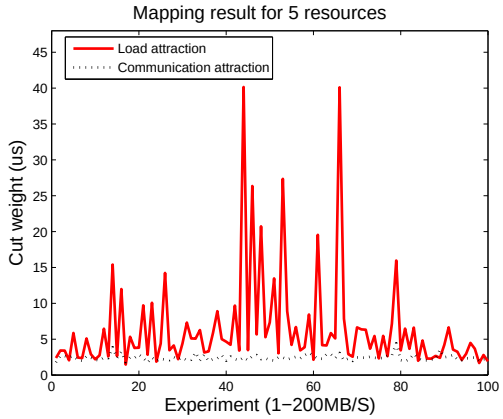
Figure 5.18: Performances on random communication resources (2MB/s increment).



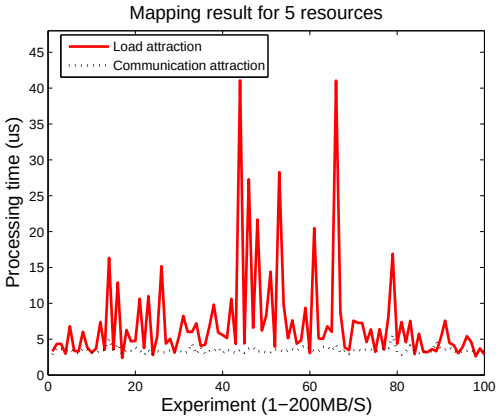
(a) Mean and STD of communication resources.



(b) Load variance.



(c) Cut weight.



(d) Overall processing cost in time.

Figure 5.19: Performances on random communication resources (1-200MB/s).

communication attraction, the computing time is only a small portion in the overall processing cost in time.

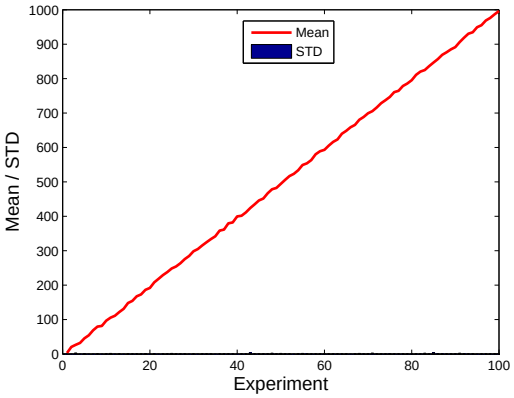
As we have observed from the scalability experiments of both processing and transmission speeds, the load attraction mapping algorithm is appropriate to heterogeneous computing resources with large variance, and the communication attraction mapping algorithm is suitable for heterogeneous communication resources with large variance. If the difference on computing or communication resources is insignificant, the mapping performance of these two algorithms are similar.

Scalability of the Component Weight

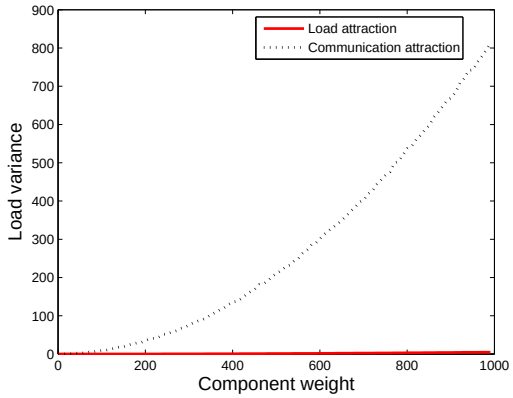
In the fifth experiment (Experiment 2.5), the 7×7 filtering with different component weights is implemented on 5 heterogeneous resources using the load attraction algorithm and the communication attraction algorithm. The purpose is to evaluate the feasibility of the proposed load and communication attraction algorithms for both simple and complex image processing algorithms. In this experiment, we keep the original setting of the resources as shown in Fig. 5.15, and randomly change the component weights.

The range of the uniformly distributed random weight is limited to $[1, 10]$. We conduct the experiment for 100 times with increment of 10. The mean and STD of component weights in each case and the performance comparisons between the two mapping algorithms are shown in Fig. 5.20. We find that the load attraction algorithm performs better on load variance and overall processing cost than the communication attraction algorithm, because the former uses the component weight that vary in this experiment as the criterion in the mapping process.

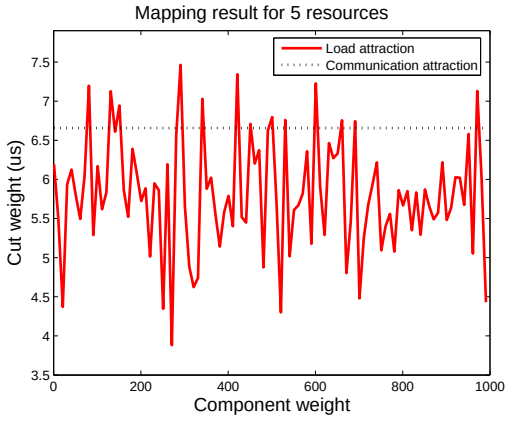
In the next experiment, the uniformly distributed random weight falls in the range of $[1, 1000]$. We also repeat the experiment for 100 times. Figure 5.21 shows the mean and STD of component weights in each case and the performance comparisons of load variance, cut weight, and overall processing cost in time between the two mapping algorithms. Table 5.11 lists the mean



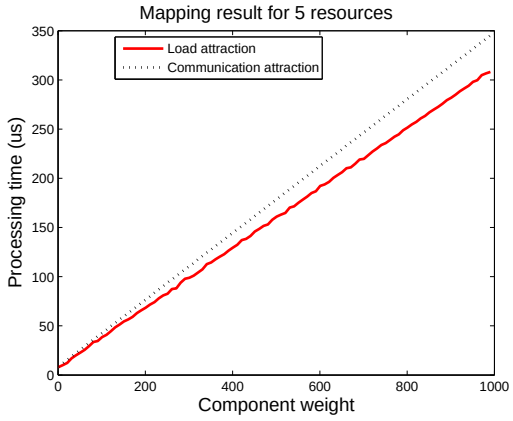
(a) Mean and STD of communication resources.



(b) Load variance.

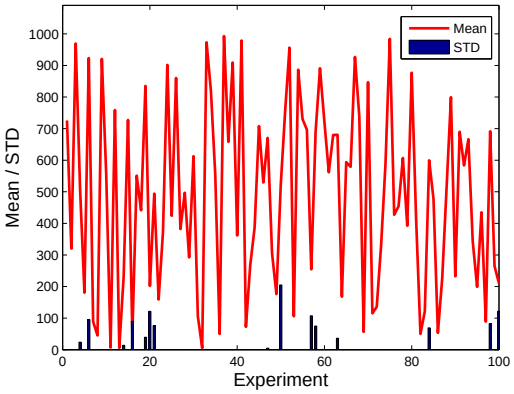


(c) Cut weight.

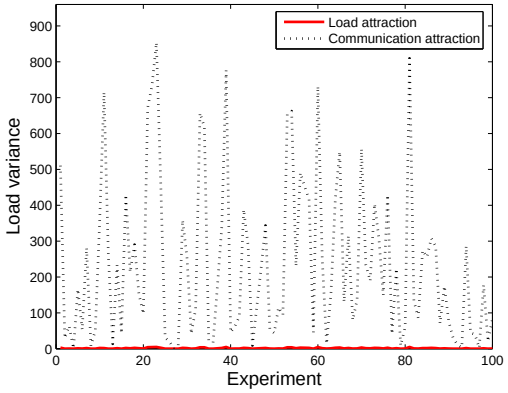


(d) Overall processing cost in time.

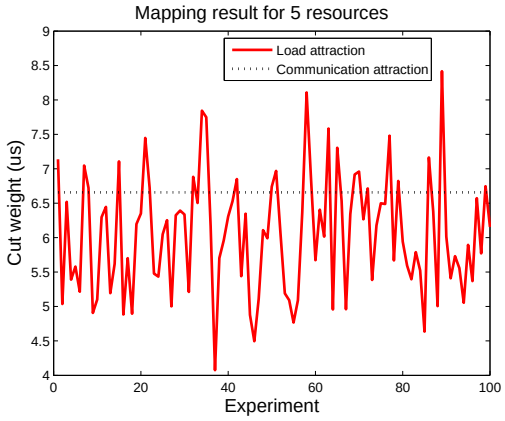
Figure 5.20: Performances on random component weight (increment of 10).



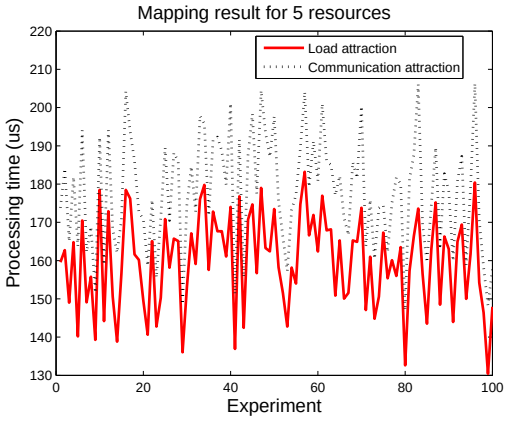
(a) Mean and STD of component weights.



(b) Load variance.



(c) Cut weight.



(d) Overall processing cost in time.

Figure 5.21: Performances on random component weights (1-1000).

Table 5.11: Mean and STD of experiments on random component weights.

Algorithm	Mean	STD
Load attraction		
Load variance	2.818570	1.440353
Cut weight	6.054567	0.854289
Overall processing cost in time	159.846645	12.210387
Communication attraction		
Load variance	384.499205	78.951638
Cut weight	6.656667	0.000000
Overall processing cost in time	177.829376	14.627896

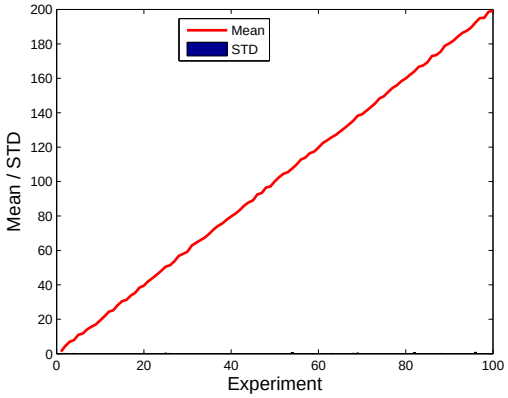
and STD of the performance parameters for these 100 cases. Obviously, the load attraction algorithm has better performance on load variance and overall processing cost than the communication attraction algorithm. Table 5.11 also shows that the load attraction algorithm always gives better mapping results when the difference of the component weight is considerable.

Scalability of the Edge Weight

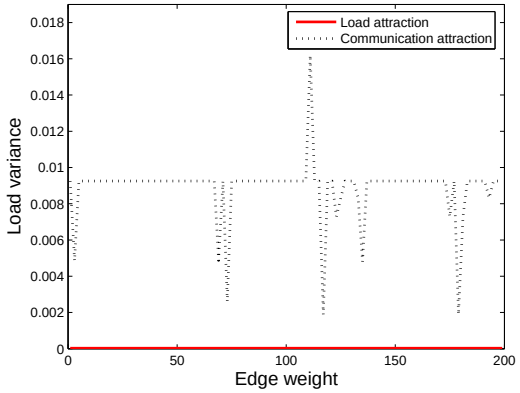
In the sixth experiment (Experiment 2.6), the 7×7 filtering with different edge weights is implemented on 5 heterogeneous resources using the load attraction algorithm and the communication attraction algorithm. The purpose is to evaluate the feasibility of the proposed load and communication attraction algorithms for tasks with different edge weights.

We keep the original component weights and randomly change the edge weights. We first set the random edge weights in the range of $[1, 2]$ and repeat 100 times with increment of 2. The mean and STD of edge weights in each case and the performance comparisons between the two mapping algorithms are shown in Fig. 5.22. Because the mapping criterion of the communication attraction algorithm is the edge weight that randomly changes in this experiment, the communication attraction algorithm has lower cut weight and overall processing cost than the load attraction algorithm.

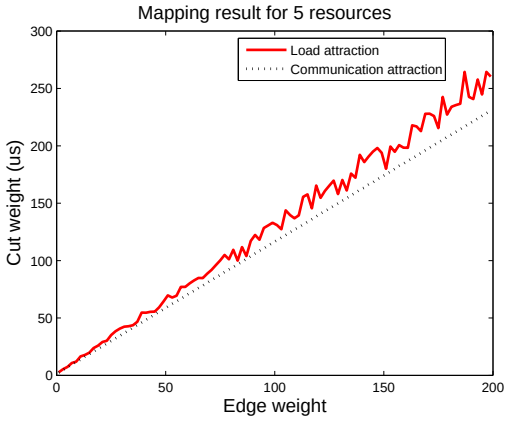
We then expand the range of the uniformly distributed random edge weight to $[1, 200]$, and



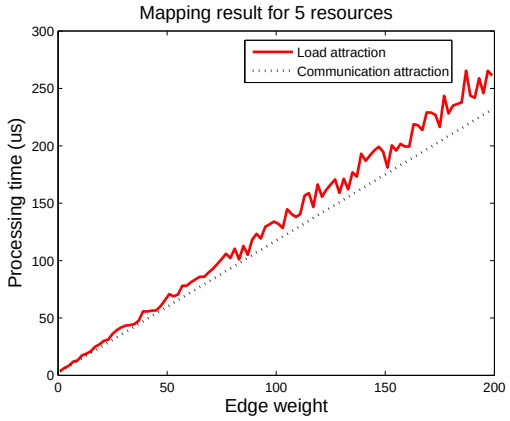
(a) Mean and STD of edge weights.



(b) Load variance.



(c) Cut weight.



(d) Overall processing cost in time.

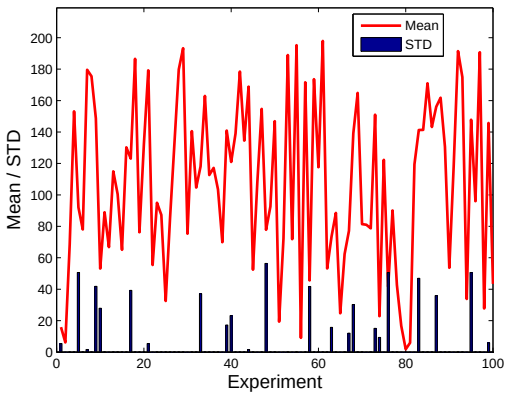
Figure 5.22: Performances on random edge weight (increment of 2).

repeat the experiment for 100 times. The mean and STD of edge weights in each time and the performance comparisons between the two mapping algorithms are shown in Fig. 5.23. The mean and STD of load variance, cut weight, and overall processing cost in time for these 100 cases are listed in Table 5.12. We can see that the load attraction algorithm performs better on load variance, where the communication attraction algorithm performs better on cut weight. Since the communication time is the major player in time consumption, the overall processing cost of the communication attraction algorithm is always lower than that of the load attraction algorithm.

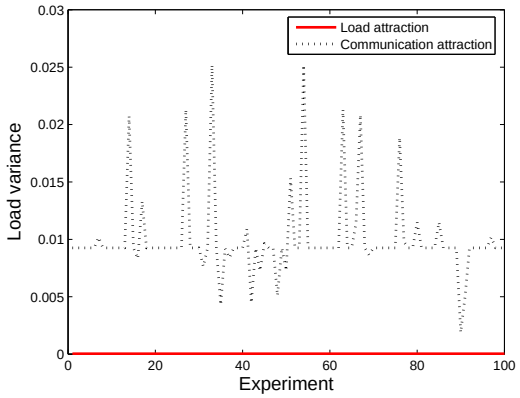
From the above experiments we find that the load attraction mapping algorithm is suitable for applications with large variance on component weights, while the communication attraction mapping algorithm is suitable for applications with large variance on edge weights. In other words, if an image processing algorithm experiences time consuming computation, such as pICA, the load attraction algorithm can give better mapping result. If an image processing algorithm has significant back-and-forth transmissions between processes, the communication attraction algorithm may be the better mapping scheme.

Table 5.12: Mean and STD of experiments on random edge weights.

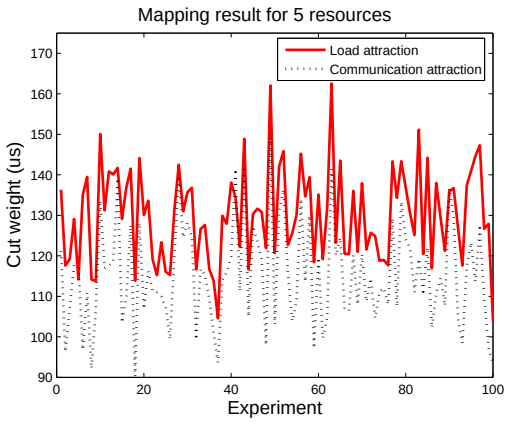
Algorithm	Mean	STD
Load attraction		
Load variance	0.120000	0.000000
Cut weight	130.030021	11.532345
Overall processing cost in time	131.073921	11.527426
Communication attraction		
Load variance	2.171600	0.742043
Cut weight	115.223121	12.591191
Overall processing cost in time	116.153121	12.591191



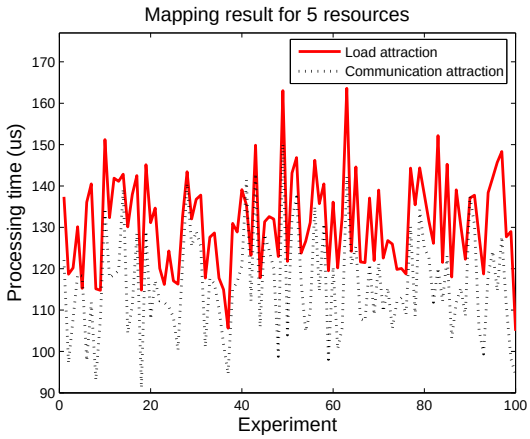
(a) Mean and STD of edge weights.



(b) Load variance.



(c) Cut weight.



(d) Overall processing cost in time.

Figure 5.23: Performances on random edge weights (1-200).

Effect of the Local Refinement

In the previous two experiments, we have demonstrated that the load attraction algorithm emphasizes the computing resources and the components during the mapping procedure, while the communication attraction algorithm focuses on the communication resources and the edges. In this experiment (Experiment 2.7), the 7×7 filtering is implemented on 5 heterogeneous resources using the load attraction algorithm and the communication attraction algorithm with and without the local refinement. The purpose is to show how the K-L local refinement further improves both algorithms and decreases the overall processing cost.

We extensively compare the mapping performance between the two proposed algorithms with and without the local refinement on contrast stretching, 3×3 filter, and 7×7 filter. Each image processing application has been pre-processed with the component clustering algorithm as we previously described. The resource we use is in the original structure and setting as shown in Fig. 5.15. Table 5.13 compares the overall processing time of the mapping algorithms with and without local refinements, based on the load attraction and the communication attraction algorithms, respectively. In contrast stretching, the local refinement does not decrease the overall processing cost because no communication exists between components. For the 3×3 and 7×7 filters, the local refinement sacrifices load balance to decreases communication, so the

Table 5.13: Overall processing time of mapping algorithms with and without the local refinements (us).

Algorithm	without local refinement	with local refinement
Based on load attraction		
Contrast stretching	0.18	0.18
3×3 filter	0.9867	0.4883
7×7 filter	7.13	3.3917
Based on communication attraction		
Contrast stretching	0.195	0.195
3×3 filter	0.8117	0.4483
7×7 filter	7.6867	3.37

overall processing cost of both applications is significantly decreased. Both the load and the communication attraction algorithms with the K-L local refinement give similar results.

From this set of experiments we find that the local refinement compensates for some drawbacks of the load and the communication attraction mapping algorithms, and decreases the overall processing cost. However, the local refinement itself has some drawbacks in mapping procedure. First, the local refinement consumes much more mapping time, since it evaluates every pair of partitions to improve the mapping result. Secondly, the objective of the local refinement is to decrease the overall processing cost but not emphasize either load balance or communication, which may not be appropriate to the power-aware resources such as sensor networks.

Mapping of Complex Task

In the previous experiments, we have evaluated the proposed load and communication attraction mapping algorithms with and without the local refinement for heterogeneous environments. In this experiment (Experiment 2.8), a complex task, the pICA algorithm, is implemented on 10 heterogeneous resources using the load attraction algorithm with and without the local refinement. As we analyzed in Chapter 4, the parallelism of ICA is a computation improvement problem rather than a trade-off between computing and communication. The communication time is only a very small portion in the overall processing time. So we use only the load attraction algorithm and its K-L refinement to compare the mapping results.

In the following experiments, we simulate the mapping on 10 computers whose configurations are listed in Table 5.1. The pICA process estimates 100 weight vectors based on the NASA AVIRIS 224-band hyperspectral image. The original mapping of pICA that is used to compare performance between pICA and FastICA is shown in Fig. 5.24, where estimations of sub-matrices are evenly distributed to the 10 computers. The mapping results of the load attraction algorithm with and without the K-L local refinement are demonstrated in Figs. 5.25

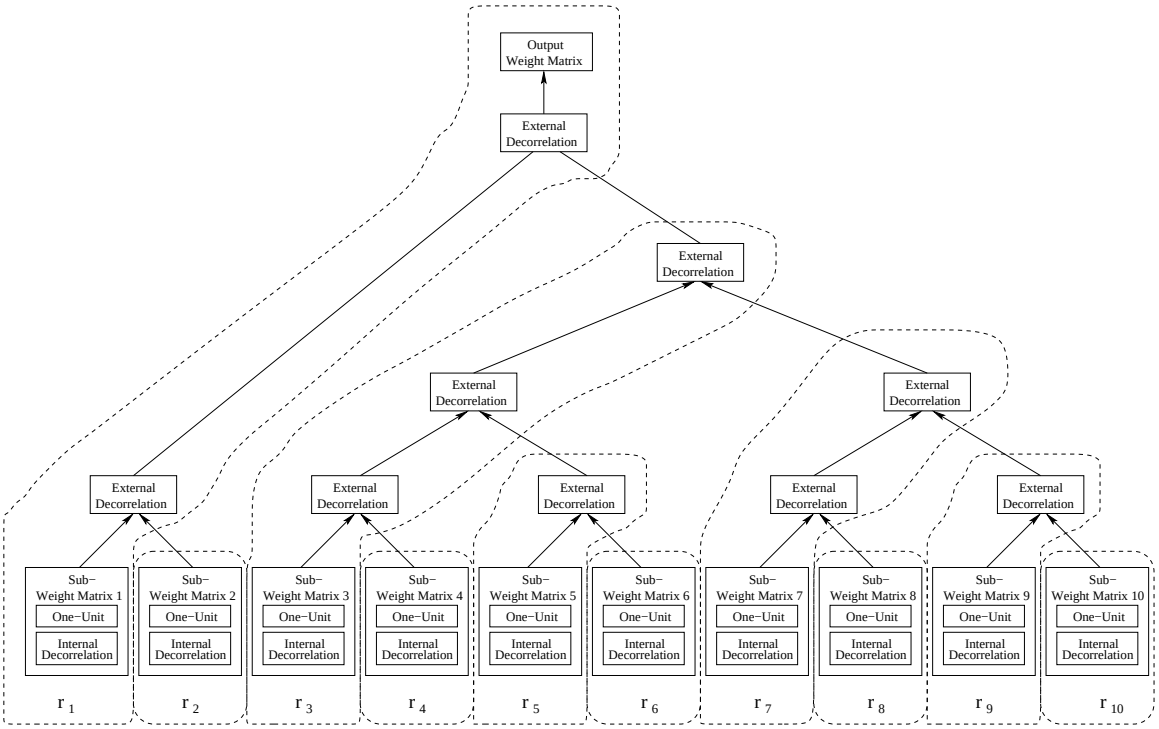


Figure 5.24: Original mapping of pICA.

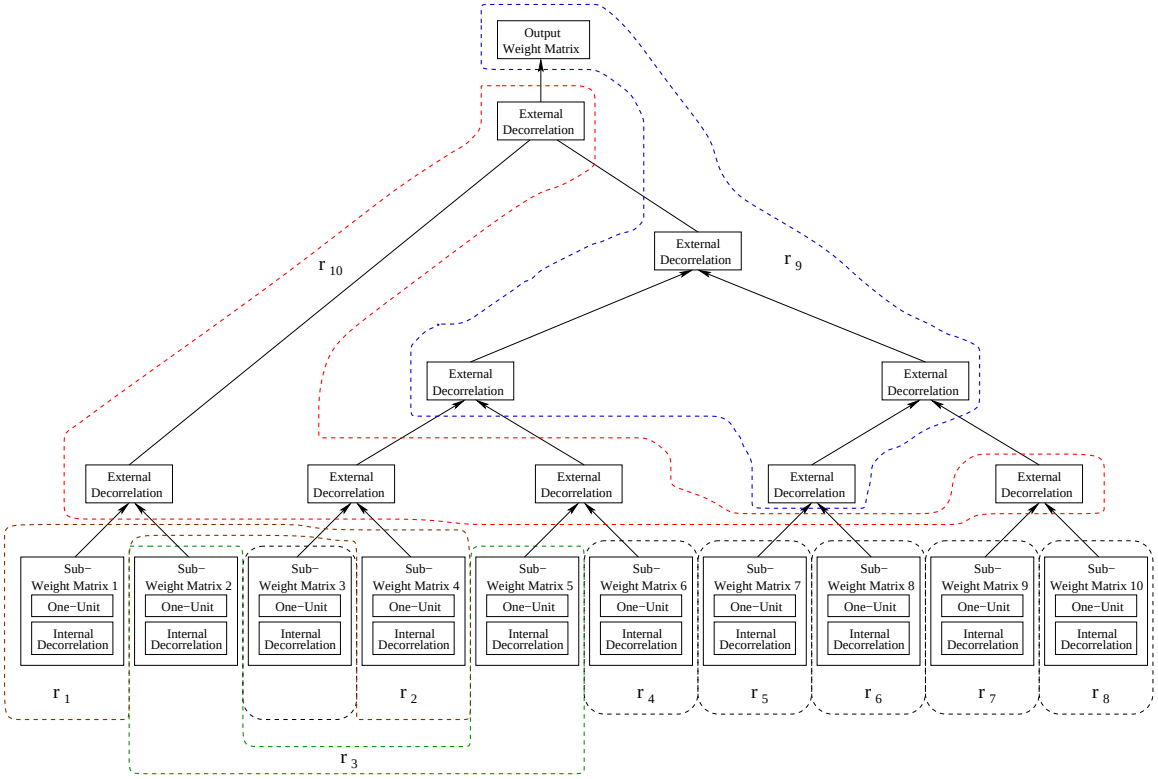


Figure 5.25: Mapping result of the load attraction algorithm without the local refinement on pICA.

and 5.26, respectively. We find that both algorithms allocate more sub-matrix estimations to faster computers such as r_1 and r_3. The detailed mapping performance comparison is shown in Fig. 5.27. We measure load variance, cut weight, processing time, and overall processing cost in time for each mapping result. The result obtained from the load attraction algorithm has the best load variance and processing time. The cut weight is tiny since the communication time is ignorable. In terms of the overall processing cost, the load attraction algorithm with the K-L refinement performs better than the other two.

In order to evaluate the scalability of our mapping algorithms, we re-construct the environment with 1 to 10 computers respectively, and compare the mapping results in Fig. 5.28. As we observe, the load attraction algorithm has lower load variance and processing time most of the time, and its K-L refinement always has the lowest overall processing cost in all the three mapping schemes.

From these experiments we find that the load attraction algorithm and its K-L refinement improve the original mapping and more efficiently allocate functions to resources. They can be applied to different applications regarding to specific needs.

5.2.3 Cyclic Process Modeling

In this experiment, we evaluate the cyclic process modeling by using pICA as an example.

In the first experiment (Experiment 3.1), pICA with and without the cyclic process modeling are implemented on 10 heterogeneous resources, as shown in Table 5.1, using the load attraction algorithm with and without the local refinement. We assume the numbers of iterations $i_1 = i_2 = 20$. In order to observe the time consumption of the mapping process for models with different granularities, we also measure the mapping time for comparison purposes. Figure 5.29 compares the mapping performances between the original one that we used in the algorithm improvement, the load attraction algorithm with its K-L refinement on the acyclic model, and that on the cyclic model.

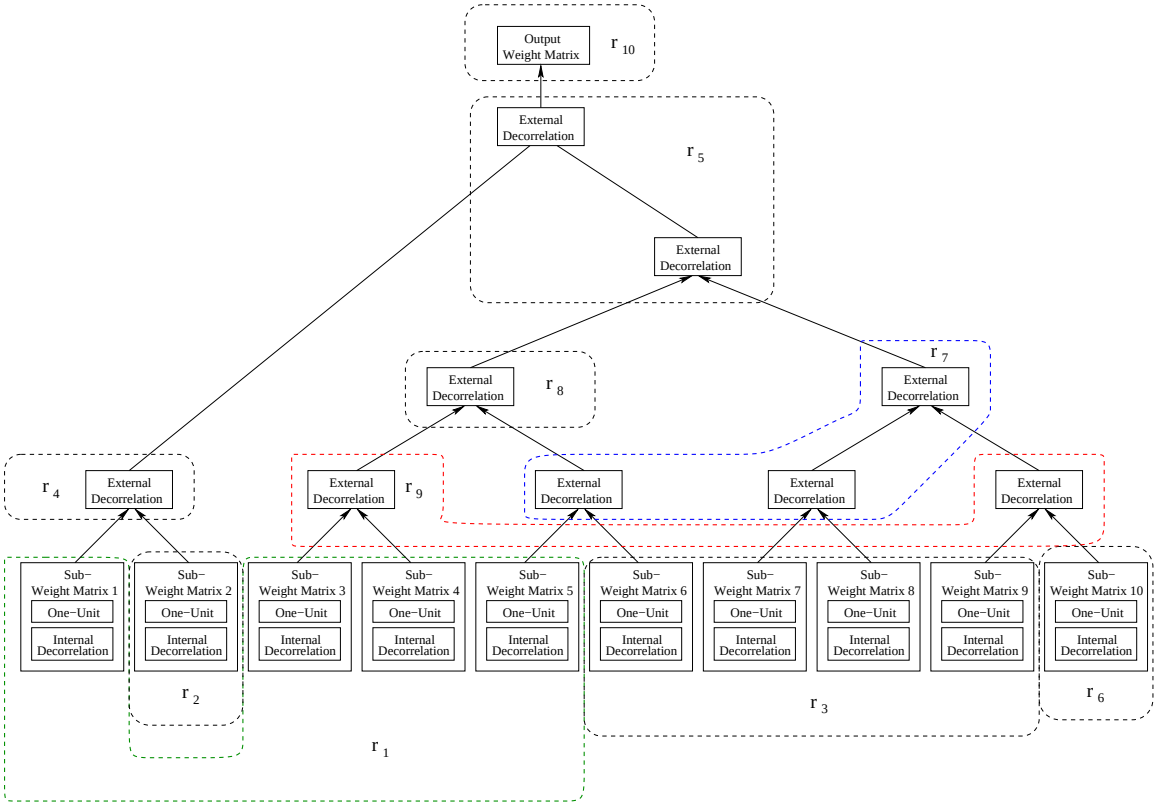


Figure 5.26: Mapping result of the load attraction with the K-L local refinement on pICA.

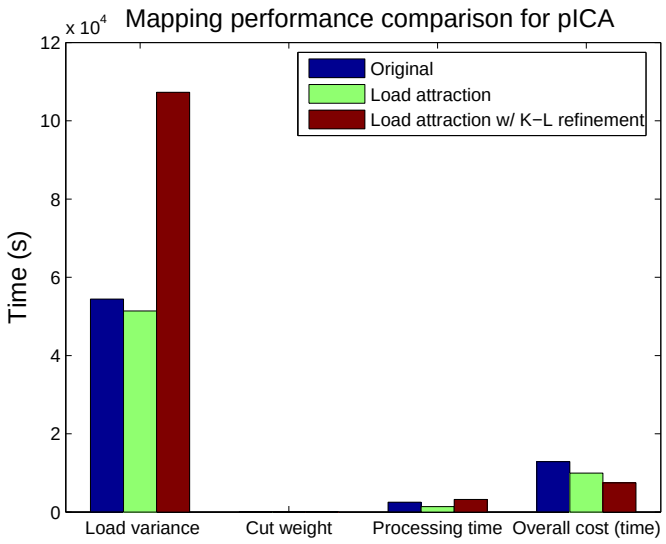
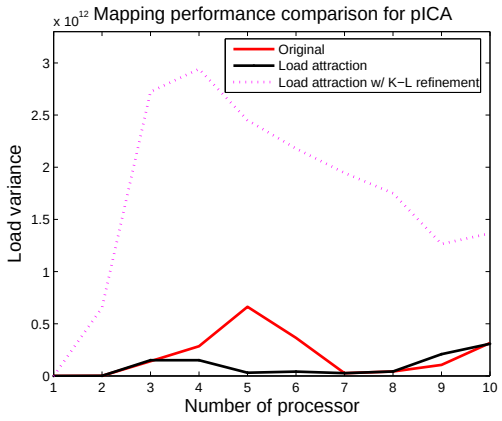
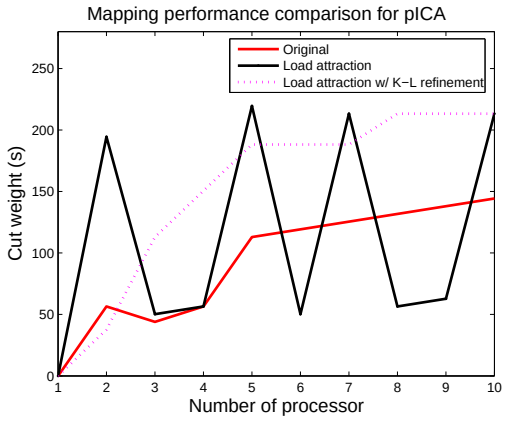


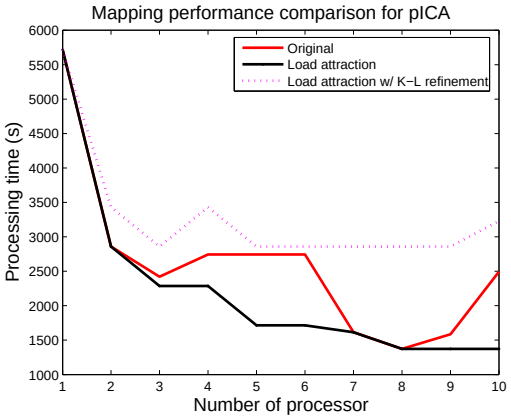
Figure 5.27: Performance comparison of different mapping algorithms.



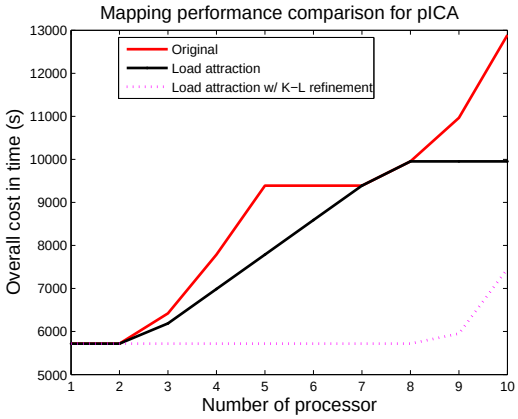
(a) Load variance.



(b) Cut weight.

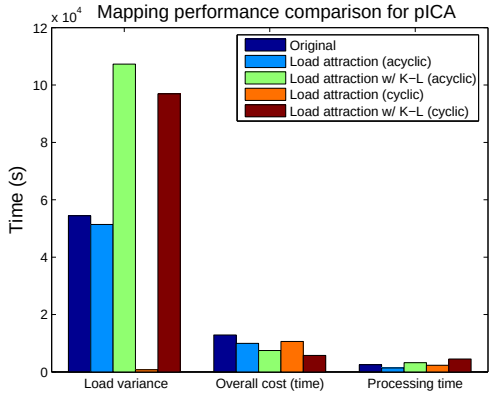


(c) Processing time.

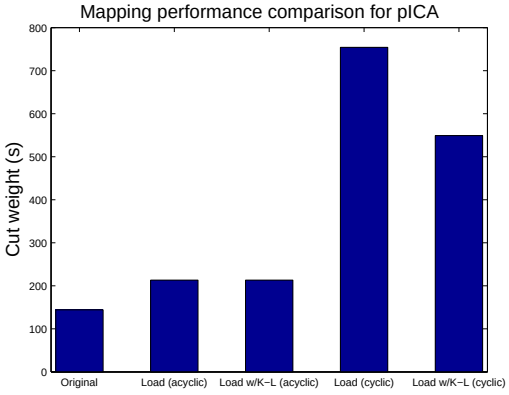


(d) Overall processing cost in time.

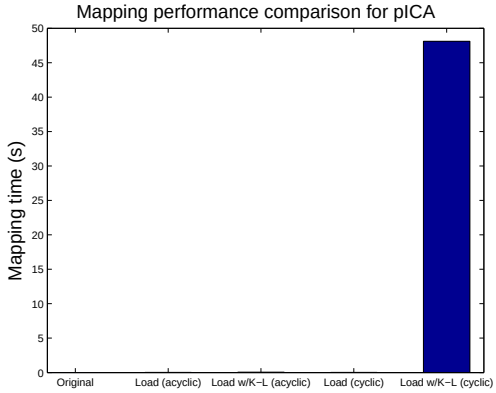
Figure 5.28: Scalability of mapping algorithms on pICA.



(a) Load variance, overall cost, processing time.



(b) Cut weight.



(c) Mapping time.

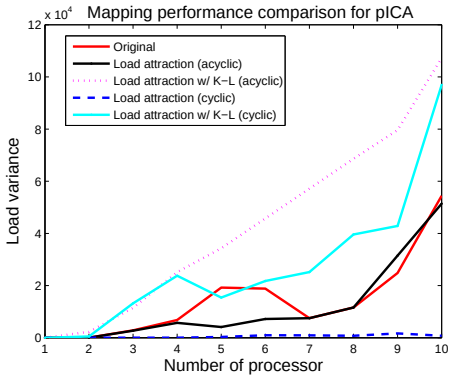
Figure 5.29: Performance comparison for cyclic process modeling in pICA.

Obviously, the load attraction on the cyclic model has much better load variance than other four results. In other words, the cyclic process modeling is very effective in the load attraction mapping algorithm. Meanwhile, the mapping process on the cyclic model consumes as little time as those on the acyclic model. Although the load attraction with the K-L refinement on the cyclic model has better overall cost, its mapping process is too slow comparing to others.

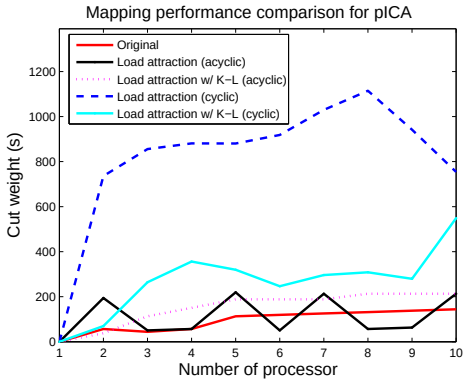
In the second experiment (Experiment 3.2), we evaluate the scalability of these mapping schemes by implementing pICA with and without the cyclic process modeling on 1 to 10 heterogeneous resources. The comparison is show in Fig. 5.30. The load attraction algorithm on the cyclic model always gives the best load variance in all schemes. We find that the K-L refinement on the cyclic model does not always perform better in overall processing cost than that on the acyclic model. But its processing time and the mapping time are higher than others.

In the third experiment (Experiment 3.3), pICA with different pre-defined number of iterations is implemented on 10 heterogeneous resources using the load attraction algorithm with and without the local refinement. In the mapping process on the cyclic model, we need to assume the number of iterations based on prior experience. Hence, we change the number of iterations from 1 to 200, and observe the mapping performance. As shown in Fig. 5.31, the mapping performances (load variance, overall cost, and processing time) of the load attraction algorithm on the cyclic model linearly increases when the number of iterations increases. The mapping time of the load attraction algorithm on the cyclic model is always low. In other words, the load attraction algorithm is very stable on the cyclic model for different pre-assumed number of iterations.

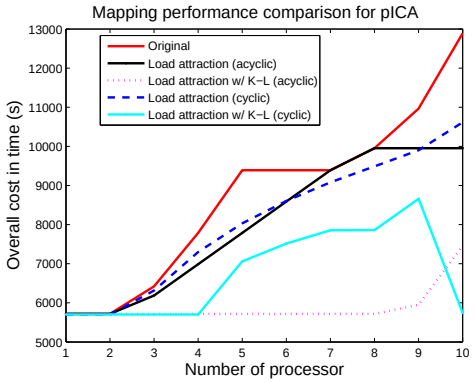
From these experiments we find that the cyclic process modeling is very effective in the load attraction mapping algorithm, but not suitable to the K-L local refinement.



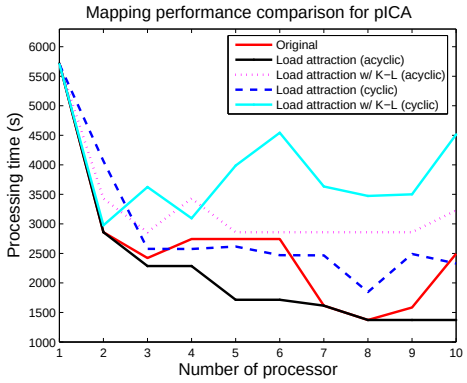
(a) Load variance.



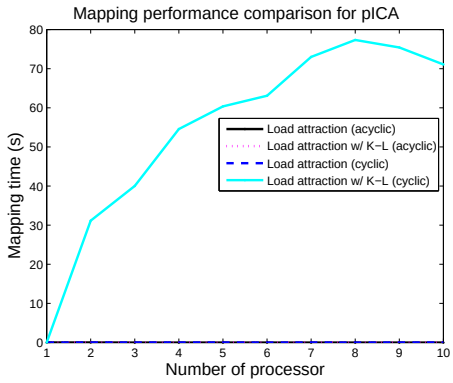
(b) Cut weight.



(c) Overall cost in time.

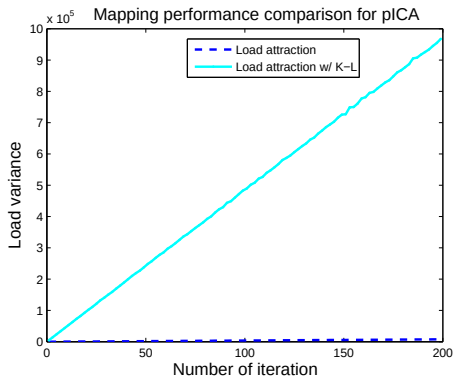


(d) Processing time.

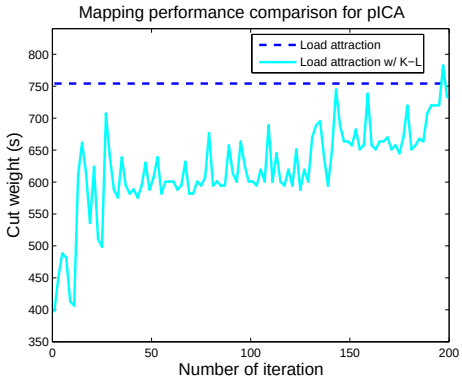


(e) Mapping time.

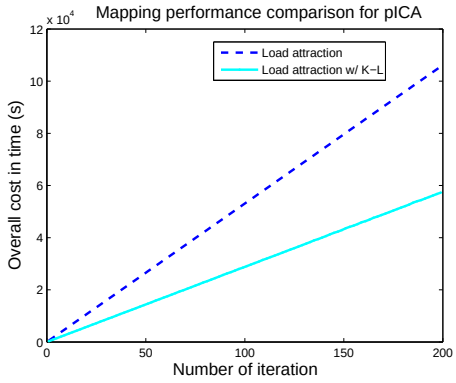
Figure 5.30: Scalability of number of processors.



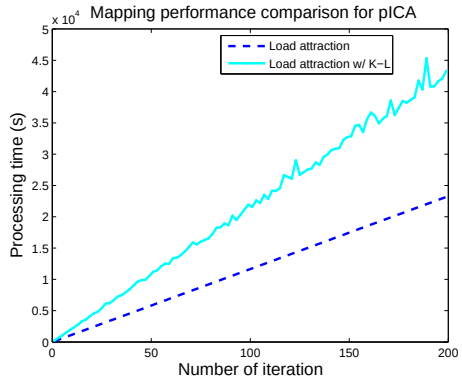
(a) Load variance.



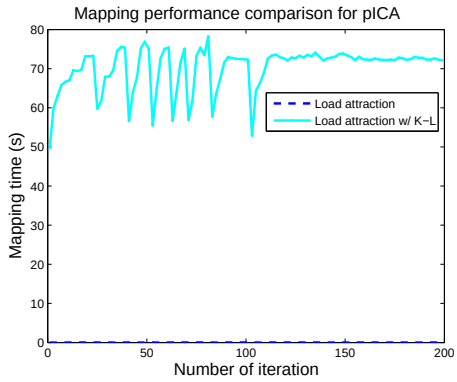
(b) Cut weight.



(c) Overall cost in time.



(d) Processing time.



(e) Mapping time.

Figure 5.31: Scalability of number of iterations.

5.3 IP Synthesis for Image Processing Library

In this section, we demonstrate the synthesis results of image processing IPs that we developed in Chapter 4. Synthesis focuses on high level design and relies on CAD software with target technologies to produce prototypes in short time. The prototypes are then downloaded to prototyping FPGAs where the programmable logic is specified, or directed to non-programmable application-specific ASICs. Since the synthesis procedure highly depends on the CAD tools in which users do not need to consider details, the prototypes can be easily modified if the application requirement changes. The synthesis procedure mainly consists of:

1. Translating VHDL into Boolean mathematical representations.
2. Optimizing the representations based on criteria such as size, delay and testability.
3. Mapping the optimized mathematical representations to a technology-specific library of components.

For each design of contrast stretching, 3×3 filter, polynomial approximation-based geometric correction, and pICA, we synthesize it on prototyping FPGAs described in Chapter 4. The detailed synthesis results are shown below.

5.3.1 Contrast Stretching

For contrast stretching, we have three IP designs, including the traditional design, the 4-pixel parallel design, and the 2-stage pipeline & 4-pixel parallel design. All of these designs are targeted at the Xilinx Virtex II Pro FPGA for prototyping. Figures 5.32, 5.33, and 5.34 respectively illustrate the schematic views on the Virtex II Pro for these designs.

Before downloading these designs to the prototyping FPGA on the test board, we can examine the layouts that are automatically generated by CAD tools, as shown in Figs. 5.35, 5.36, and 5.37. Obviously, the traditional design takes very little area that denotes the utilization

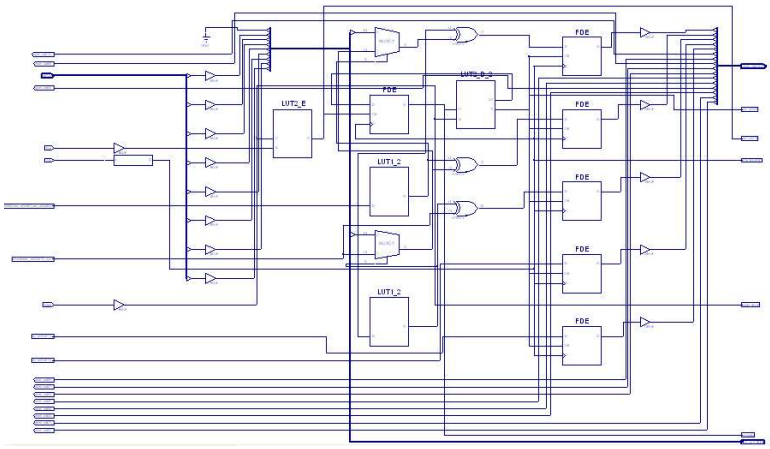


Figure 5.32: Schematic of Virtex II Pro for contrast stretching (the traditional design).

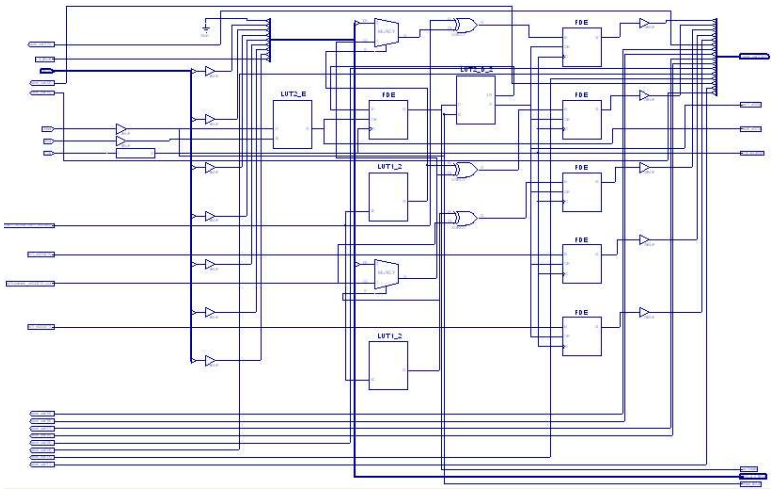


Figure 5.33: Schematic of Virtex II Pro for contrast stretching (the 4-pixel parallel design).

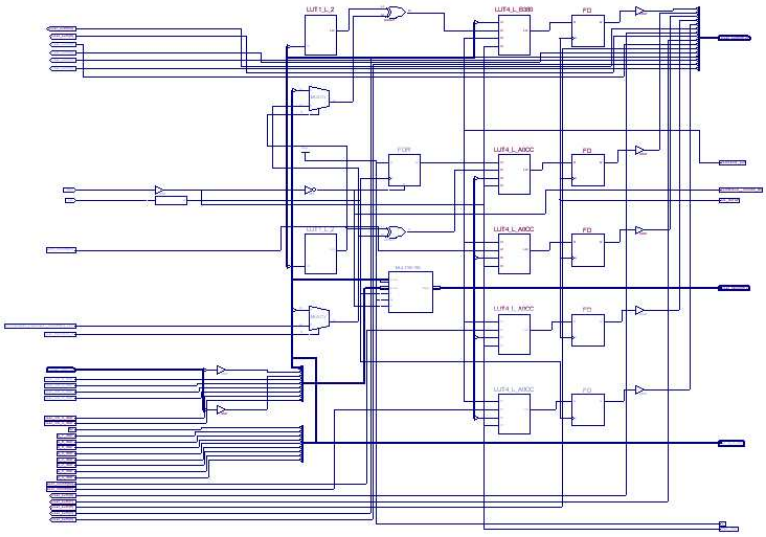


Figure 5.34: Schematic of Virtex II Pro for contrast stretching (the 2-stage pipeline & 4-pixel parallel design).

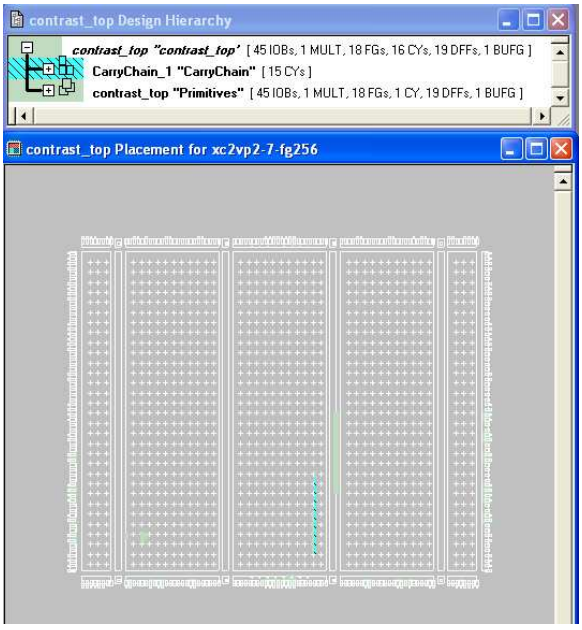


Figure 5.35: Layout on Virtex II Pro for contrast stretching (the traditional design).

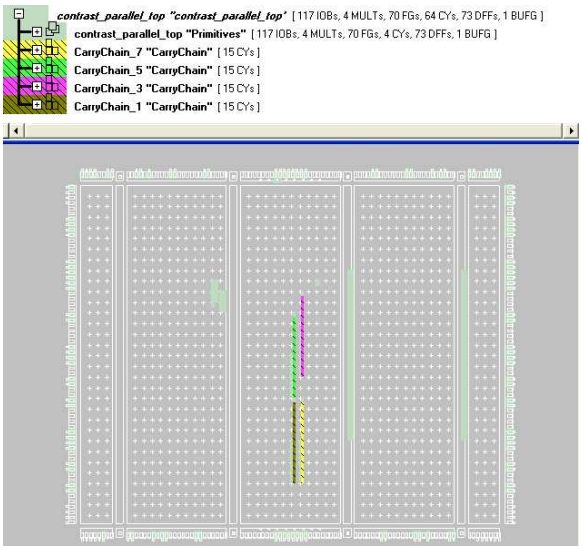


Figure 5.36: Layout on Virtex II Pro for contrast stretching (the 4-pixel parallel design).

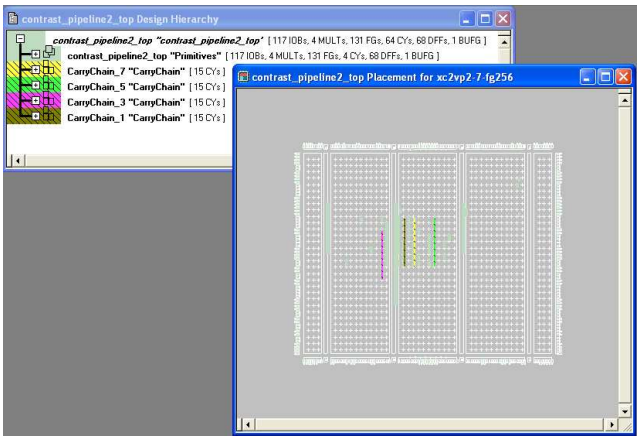


Figure 5.37: Layout on Virtex II Pro for contrast stretching (the 2-stage pipeline & 4-pixel parallel design).

area on FPGA, the parallel design takes almost four times the area of the traditional one, and the 2-stage pipeline & 4-pixel parallel design also takes more area than the traditional design.

The detailed performance comparison of the three designs are listed in Table 5.14 according to parameters including area, clock cycle, the maximum delay, the minimum period / maximum frequency, and power consumption estimation. It is evaluated by

1. BUFGMUXs: portal to the clock net that is used to clock Flip Flops (FFs);
2. IOBs: the Input/Output Blocks;
3. MULT18X18s: multipliers; and
4. SLICEs: combination of LookUp Tables (LUTs) and FFs.

The clock cycle is the number of the consumed clock cycles to generate one output or multiple outputs in designs with or without parallel processing. The maximum delay is caused by the interconnections between LUTs and FFs. If the configured programmable logics are close to each other, the length of interconnections is comparatively short and the maximum delay decreases; otherwise, the maximum delay increases. In high-speed circuits, the maximum delay is an important concern, since it could lead to functional fault if it is close to the minimum period. The minimum period denotes the minimum time interval required by one clock cycle to complete the process. If the function to be completed within one clock cycle is complicated, the period will be increased. Therefore, we need to partition the complicated function into smaller ones to decrease the minimum period. The minimum period and the clock cycles together exhibit the processing speed of designs, where we know how long the system generates one or multiple outputs. The maximum frequency corresponds to the minimum period, where $Max\ Frequency = \frac{1}{Min\ Period}$. And the power issue includes the working temperature and consumption estimation.

From Table 5.14 we find that the traditional design uses less IOBs than others, since it processes only one input data each time. Both the parallel design and the 2-stage pipeline & 4-

Table 5.14: Performance comparison of FPGA implementations for contrast stretching.

Design	Traditional design	4-pixel parallel design	2-stage pipeline & 4-pixel parallel
Area			
BUFGMUXs:	1/16 (6%)	1/16 (6%)	1/16 (6%)
External IOBs:	45/140 (32%)	117/140 (83%)	117/140 (83%)
MULT18X18s:	1/12 (8%)	4/12 (33%)	4/12 (33%)
SLICEs:	10/1408 (1%)	38/1408 (2%)	69/1408 (4%)
Clock Cycles	2 for 1 output	2 for 4 output	1 for 4 output
Max Delay	0.714 ns	0.718 ns	0.718 ns
Min Period / Max Frequency	2.014 ns / 496.5 MHz	2.014 ns / 496.5 MHz	4.114 ns / 263.11 MHz
Power			
Temperature	36 $^{\circ}C$	36 $^{\circ}C$	36 $^{\circ}C$
Estimation	420 mW	420 mW	420 mW

pixel parallel design process four input data. The traditional and the parallel designs respectively generate one and four results in every two clock cycles, while the pipelined design generates four results in every one clock cycle. The maximum delay of the traditional design is a little less than the other two designs since the interconnection is comparatively shorter. We can also observe this difference from the layouts of the four designs shown in Figs. 5.35, 5.36, and 5.37. The min period / max frequency of the traditional and the parallel designs are the same because the parallel design has the same process as the traditional design. As the result of the connection between two stages, the 2-stage pipeline & 4-pixel parallel design has higher min period and lower max frequency. The power consumptions of these four designs are the same. From the processing speed point of view, the 2-stage pipeline & 4-pixel parallel design is the fastest one due to the effect of pipelining on the multiplier and the adder. The traditional design uses the least utilization area but consumes the longest processing time. Each of the three designs has pros and cons in performance. So the selection of the appropriate design still depends on the specific applications and conditions.

5.3.2 Polynomial Approximation-based Geometric Correction

The IP design of the polynomial approximation-based geometric correction is more complicated than the contrast stretching and the 3×3 filter. As the layout on the Xilinx Virtex II Pro shown in Fig. 5.38, this IP design uses most of the resources of the prototyping FPGA.

Since the polynomial approximation includes three main function blocks including the formation of matrix $\mathbf{W}$, the matrix multiplication, and the matrix inverse, we respectively implement them and demonstrate the performance in Table 5.15. The matrix formation and the matrix multiplication blocks use less utilization areas and have smaller maximum delays than the matrix inverse block. The minimum delay of the matrix inverse block is the largest in the three blocks. In other words, the maximum frequency of the overall polynomial approximation design will not exceed that of the matrix inverse block.

Table 5.16 shows the performance of the polynomial approximation design. This design uses 63 IOBs: 8 IOBs for each of the control points (u, v), the corresponding points (x, y), the coordinates in the transformation image, and the output coordinates $(\hat{x}, \hat{y})$; 7 IOBs for the control signals. The minimum period of the polynomial approximation is 8.468 ns and the maximum frequency is 118.088 MHz, which is less than that of the individual function blocks.

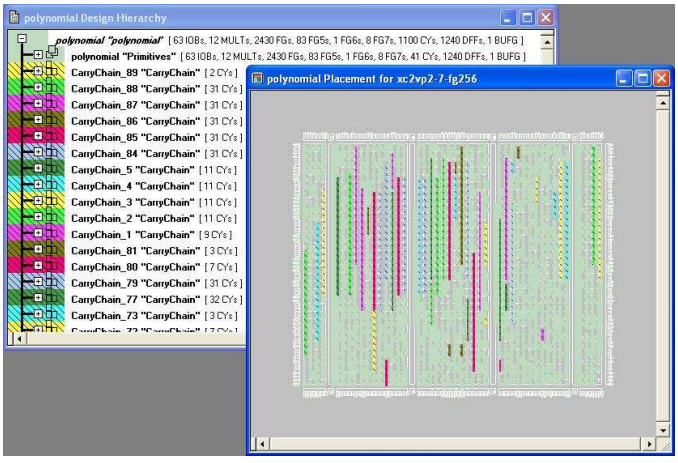


Figure 5.38: Layout on Virtex II Pro for the polynomial approximation-based geometric correction.

Table 5.15: Performance of Virtex II Pro implementations for blocks in polynomial approximation.

Design	**W** formation	Matrix multiplication	Matrix inverse
Area			
Flip Flops:	189/2816 (6%)	124/2816 (4%)	310/2816 (11%)
MULT18X18s:	7/12 (58%)	1/12 (8%)	2/12 (17%)
SLICEs:	199/1408 (14%)	109/1408 (7%)	755/1408 (53%)
Max Delay	0.689 ns	0.689 ns	0.716 ns
Min period / Max Frequency	4.325 ns / 231.192 MHz	5.560 ns / 179.851 MHz	7.232 ns / 138.265 MHz
Power			
Temperature	37 $^{\circ}C$	37 $^{\circ}C$	37 $^{\circ}C$
Estimation	450 mW	450 mW	450 mW

Table 5.16: Performance of Virtex II Pro implementations for polynomial approximation.

Area	
BUFGMUXs:	1/16 (6%)
External IOBs:	71/140 (51%)
MULT18X18s:	12/12 (100%)
SLICEs:	1206/1408 (86%)
Max Delay	0.719 ns
Min period / Max Frequency	8.468 ns / 118.088 MHz
Power	
Temperature	37 $^{\circ}C$
Estimation	450 mW

In order to compare the performance of our design to another existing design that uses Xilinx Virtex E FPGA, we also implement the geometric correction IP on Xilinx V1000E. After the synthesis, we achieve the minimum period of 54.726 ns (maximum frequency of 18.273 MHz) and the maximum delay of 19.690 ns. The geometric correction design uses 87% slices of the Xilinx V1000E. The performance is reported in Table 5.17. The comparison with the real-time FPGA system implementation in [60] is shown in Table 5.18. From the performance comparison, we find that our polynomial approximation-based geometric correction IP is efficient in providing higher processing speed and circuit density.

5.3.3 3×3 Filter

For the 3×3 filter, we have three IP designs, including the traditional design, the parallel & pipelined design, and the parallel & pipelined design with partitioning. For these three designs, we also use the Xilinx Virtex II Pro as the prototyping FPGA. The schematic view of the traditional design on the Virtex II Pro is shown in Fig 5.39.

The layout of the traditional design, the parallel & pipelined design, and the parallel & pipelined design with partitioning are shown in Figs. 5.40, 5.41, and 5.42. We find that the layout of the traditional design is very loose, while that of the other two designs are

Table 5.17: Design and device utilization.

Item	Number	Percentage
Slices:	10,703	87%
Flip Flops	5,556	22%
Input LUTs	12,660	51%
IOBs	24	15%
Equivalent gate count	217,027	
After Placing and Routing		
Paths	126,932,424,793	
Nets	25,323	
Connections	68,291	

Table 5.18: Performance comparison of FPGA implementations.

Design	Real time FPGA calculation	IP in virtual microsensor platform
FPGA	Xilinx Virtex E	Xilinx Virtex 1000E
Area (%)	63	87
SRAM	5 MB	84.375 KB
Maximum frequency	10 MHz	18.273 MHz

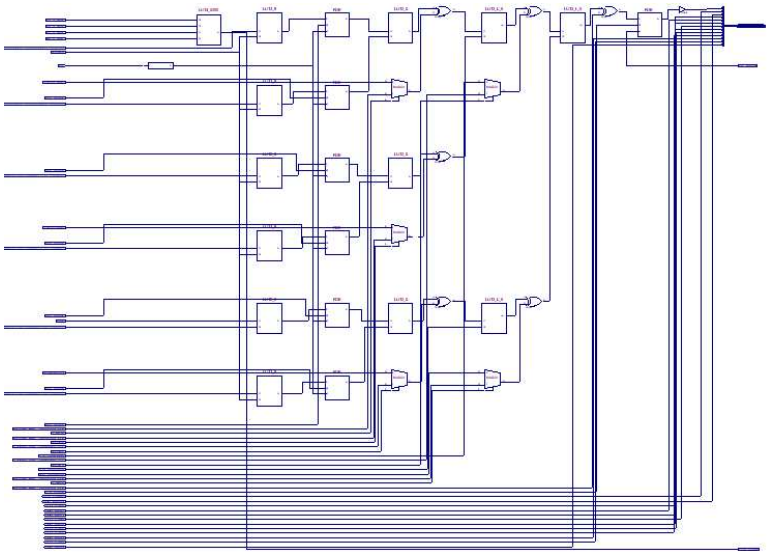


Figure 5.39: Schematic of Virtex II Pro for traditional 3×3 filter.

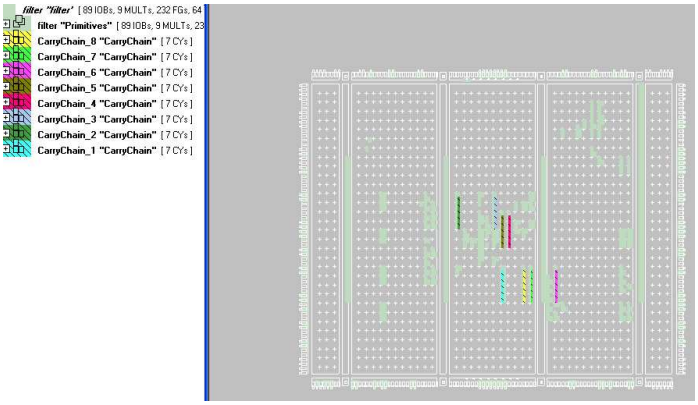


Figure 5.40: Layout on Virtex II Pro for 3×3 filter. (the traditional design).

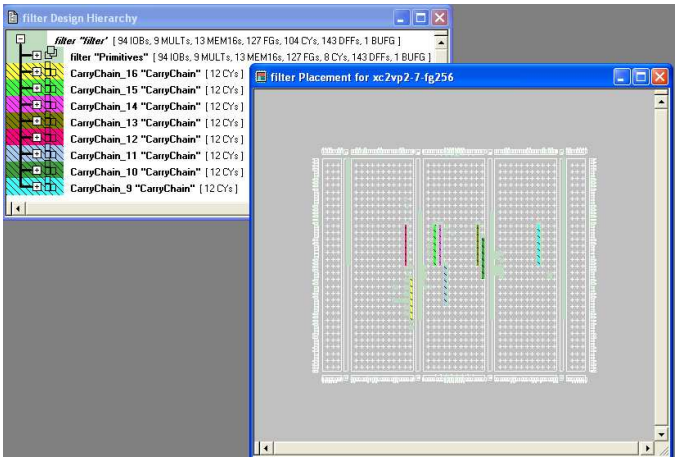


Figure 5.41: Layout on Virtex II Pro for 3×3 filter. (the parallel & pipeline design).

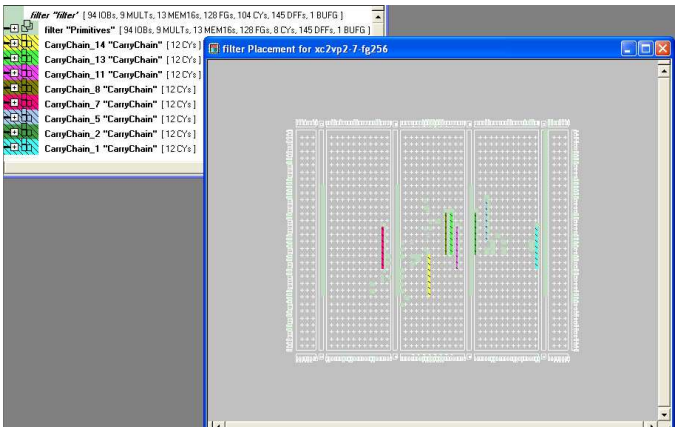


Figure 5.42: Layout on Virtex II Pro for 3×3 filter. (the Parallel & pipeline design with partitioning).

comparatively compact. In the parallel & pipelined design, the multiplication and addition operations are decomposed into individual processes. In the partitioning design, the entire set of operations are divided into several partitions. Therefore, these two designs permit the CAD tools to more efficiently place and route processes. As we previously analyzed, the loose layout may lead to higher maximum delay due to the length of interconnections.

Table 5.19 compares the detailed performance of these three designs for the 3×3 filter. The parameters used in this table are the same as those in Table 5.14. From the functionality point of view, all designs perform the same operations but in different structures. Therefore, they have the same numbers of BUFGMUXs, IOBs, and multipliers. As we compared in the layouts of these designs, the traditional design has very loose layout therefore using more SLICEs than the other two. The parallel & pipelined design uses a little more SLICEs than the parallel & pipelined design with partitioning, which is caused by the effectiveness of the partitioning on the parallel and pipelined structures. Compared to the traditional design that generates one output in five clock cycles, the parallel & pipeline designs dramatically increase the processing

Table 5.19: Performance comparison of FPGA implementations for 3×3 filter.

Design	Traditional	Parallel & pipeline	Parallel & pipeline with partitioning
Area			
BUFGMUXs:	1/16 (6%)	1/16 (6%)	1/16 (6%)
External IOBs:	94/140 (67%)	94/140 (67%)	94/140 (67%)
MULT18X18s:	9/12 (75%)	9/12 (75%)	9/12 (75%)
SLICEs:	175/1408 (12%)	92/1408 (6%)	91/1408 (6%)
Clock Cycles	5 for 1 output	1 for 1 output	1 for 1 output
Max Delay	0.718 ns	0.717 ns	0.717 ns
Min period / Max Frequency	8.008 ns / 124.875 MHz	3.260 ns / 306.796 MHz	3.260 ns / 306.796 MHz
Power			
Temperature	36 $^{\circ}C$	37 $^{\circ}C$	37 $^{\circ}C$
Estimation	420 mW	450 mW	450 mW

speed and generate one output in one clock cycle. In addition, the minimum delays of the parallel & parallel designs are smaller than that of the traditional design, since they have more compact layout and optimized processing structures. Therefore, the overall processing speed of the parallel & parallel designs is much faster than the traditional design.

5.3.4 pICA

The pICA IP is also a very complex design. Figure 5.43 shows the layout on the Xilinx Virtex II Pro FPGA. We observe that the pICA design uses most of the resources on this FPGA.

The pICA design contains two main function blocks, including the one unit estimation and the decorrelation. The performance of these two blocks are listed in Table 5.20. The one unit block involves more complex operations than the decorrelation block, therefore using more resources of flip flops, multipliers, and slices. In addition, the one unit estimation uses longer minimum period and smaller maximum frequency than the decorrelation process. Both processes have the same maximum delay and power consumption.

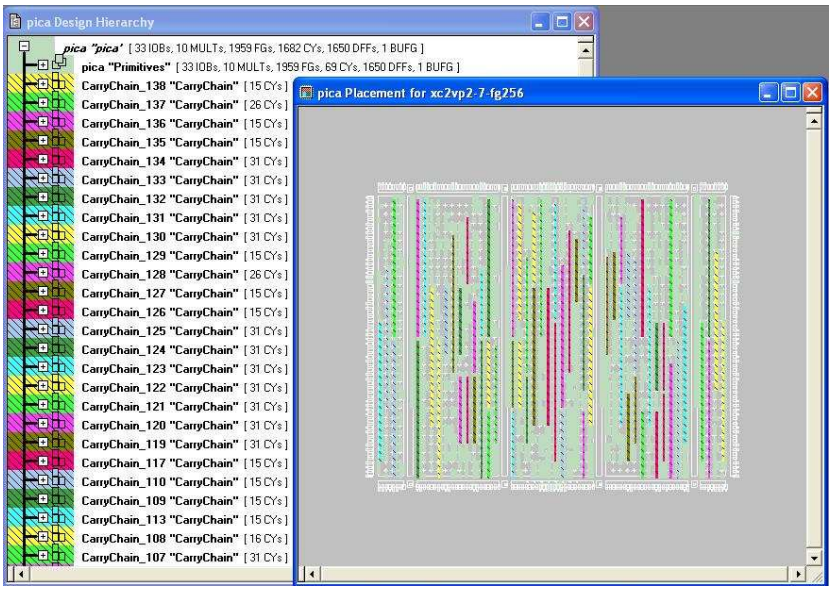


Figure 5.43: Layout on Virtex II Pro for the pICA algorithm.

Table 5.20: Performance of Virtex II Pro implementations for blocks in the pICA algorithm.

Design	One unit block	Decorrelation block
Area		
Flip Flops:	768/2816 (54%)	450/2816 (15%)
MULT18X18s:	10/12 (83%)	3/12 (25%)
SLICEs:	719/1408 (25%)	403/1408 (28%)
Max Delay	0.718 ns	0.718 ns
Min period / Max Frequency	11.965 ns / 83.578 MHz	10.364 ns / 96.491 MHz
Power		
Temperature	37 $^{\circ}C$	37 $^{\circ}C$
Estimation	450 mW	450 mW

Table 5.21 lists the performance parameters of the entire design of pICA. This IP design uses 33 IOBs: 16 for the observed signal x, 16 for the source signal s, and 1 for the clock input. It also uses 10 multipliers and covers 1318 slices. The minimum period of the pICA design is 12.159 ns and the maximum frequency is 82.244 MHz.

For the purpose of comparison between our design and existing works, we also implement the pICA design on the Xilinx Virtex E FPGA that is embedded on the Pilchard re-configurable test board [96]. As the implementation procedure shown in Fig. 5.44, the pICA algorithm is first simulated by ModelSim from Mentor Graphics, then synthesized by Synopsys FPGA Compiler2, finally placed and routed by Xilinx XVmake. After implementing the pICA on the Xilinx V1000E embedded on the Pilchard board, we achieve the minimum period of 49.6 ns (maximum frequency of 20.161 MHz) and the maximum delay of 13.119 ns. The pICA uses 92% slices of the V1000E FPGA. The detailed design and device utilization are listed in Table 5.22. The comparison is conducted between the pICA IP and another two ICA-related FPGA implementations, including the 7-neuron ICNN and the ICA-based BSS algorithm. Table 5.23 lists the FPGAs used in this comparison. The detailed comparisons are performed from aspects of target FPGA capacity, frequencies of synthesized designs, and size of data sets. As

Table 5.21: Performance of Virtex II Pro implementations for the pICA algorithm.

Area	
BUFGMUXs:	1/16 (6%)
External IOBs:	33/140 (23%)
MULT18X18s:	10/12 (83%)
SLICEs:	1318/1408 (93%)
Max Delay	0.717 ns
Min period /	12.159 ns /
Max Frequency	82.244 MHz
Power	
Temperature	37 $°C$
Estimation	450 mW

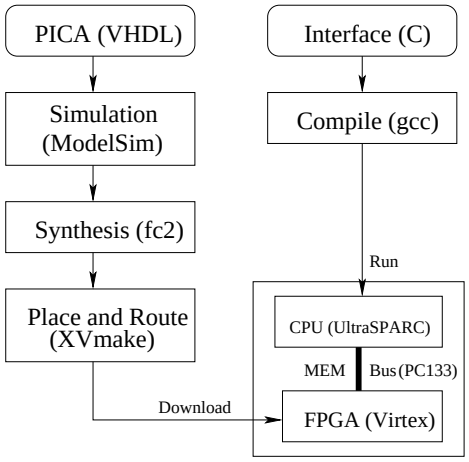


Figure 5.44: Implementation procedure of the pICA algorithm on Pilchard board.

Table 5.22: Design and device utilization.

Item	Amount	Percentage
Slices:	11,318	92%
Flip Flops	6,061	24%
Input LUTs	19,114	77%
IOBs	32	20%
Equivalent gate count	229,500	
After Placing and Routing		
Paths	129,753,145,344	
Nets	26,884	
Connections	73,169	

Table 5.23: FPGAs used in comparison.

Implementation	FPGA	Capacity (million gates)
7-neuron ICNN	Xilinx Virtex XCV 812E	0.25
BSS algorithm	Xilinx Virtex 600E	0.6
pICA algorithm	Xilinx Virtex 1000E	1.0

shown in Fig. 5.45, the target FPGA capacity of the pICA implementation is respectively 1.68 and 4 times larger than the previous ICNN and BSS implementations. Due to the complexity of the pICA algorithm, the frequency of the developed FPGA is slower than ICNN prototypes, but the same as the BSS algorithm. In the application of dimensionality reduction in HSI analysis, the synthesized pICA algorithm processes many more observation signals than the other two applications. Overall, this FPGA implementation for the pICA algorithm is a successful adventure for complicated algorithms and large volume data sets.

5.4 Implementation of Microsensor Integration

In the microsensor integration design, the input image is processed through two popular image processing algorithms for enhancement and edge detection purposes. These two algorithms are the contrast stretching for image enhancement and the Sobel edge detector (horizontal) for edge identification. We use a real image to demonstrate the effect of this design. As shown in Fig. 5.46, the original image is a little dark to effectively display the details of the mountain, while the result of the contrast stretching shows more details with better contrast levels. The result of the Sobel filter clearly displays the edges in this image.

After implementing the overall design on the Virtex II Pro, we plot the schematic at technology view in Fig. 5.47. Since the whole schematic is very large, we only show partial schematic of the integration design.

Figure 5.48 shows the layout of the integration design on the Xilinx Virtex II Pro FPGA.

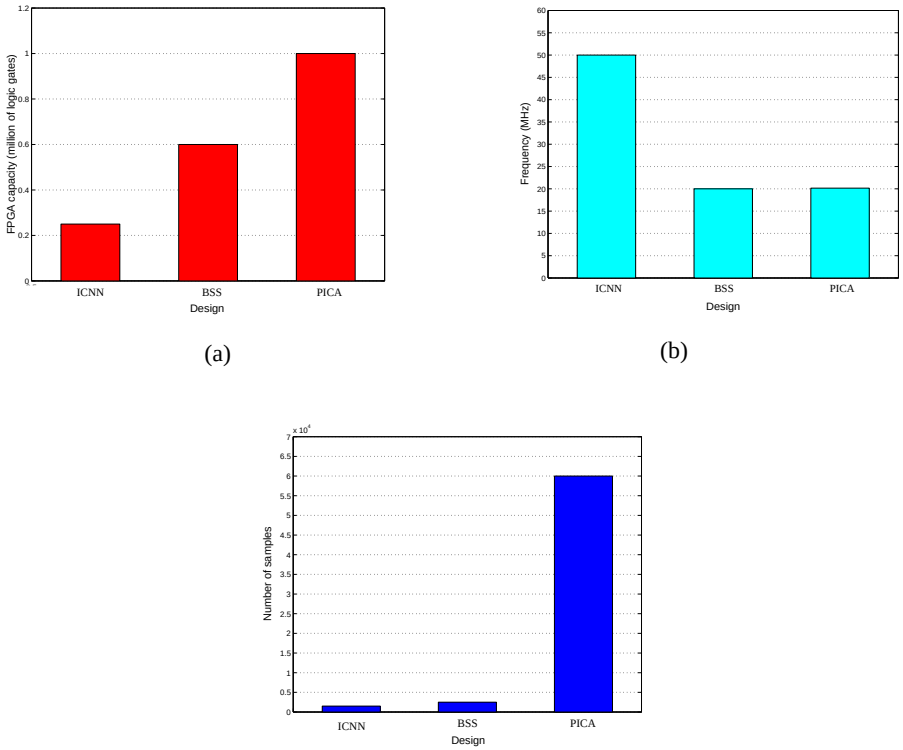
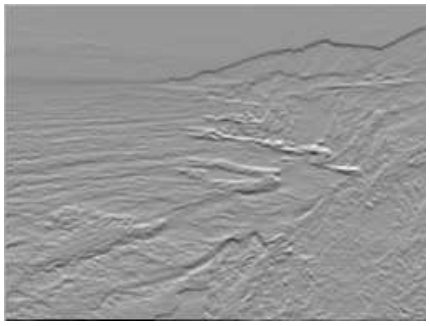


Figure 5.45: Performance comparisons [97, 123]. (a) Target FPGA capacity. (b) Frequencies of synthesized designs. (c) Size of observation data sets.

(a) Original input image.

(b) Result of contrast stretching.



(c) Result of Sobel edge detector (horizontal).

Figure 5.46: Original and result images of the integration design.

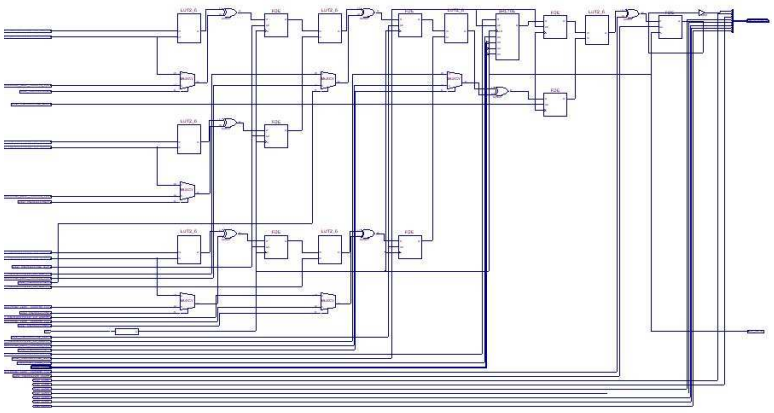


Figure 5.47: Schematic of Virtex II Pro for the integration design.

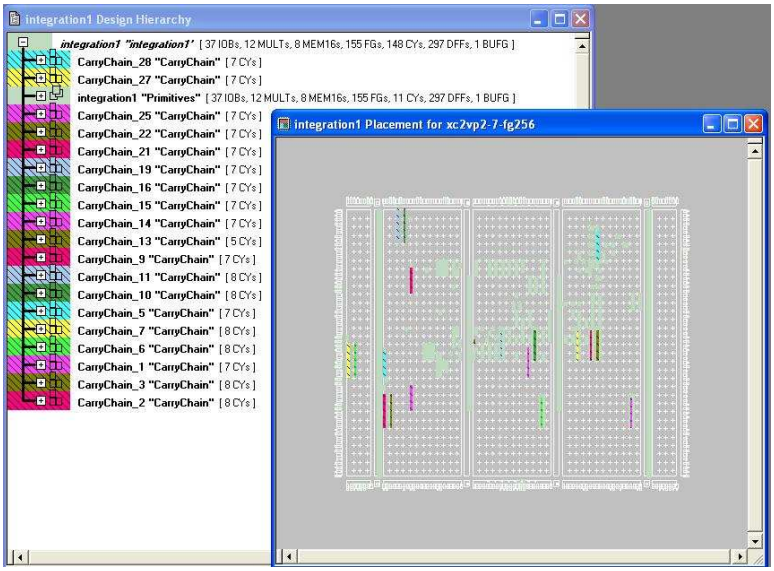


Figure 5.48: Layout on Virtex II Pro for the integration design.

We observe that this design does not use many resources of the prototyping FPGA. The loose circuit distribution is mainly caused by the use of many RAM blocks.

Table 5.24 lists the performance of the integration design. This design totally uses 37 IOBs: 24 for the three input pixels in parallel, 8 for the output pixels, and 5 for clock input and control signals. It uses all the 12 multipliers and covers 211 slices. This design generates one output pixel in one clock cycle. The minimum period of 12.159 ns and the maximum frequency of 82.244 MHz demonstrate the fast processing speed on the input image and the effectiveness of this design.

Through the above example we find that the virtual microsensor platform provides a flexible and reliable integration environment for fast microsensor design and development.

5.5 Summary

In this chapter, we have shown experimental results and comparisons for the algorithm improvement, function partitioning, clustering, and mapping, and IP implementation. In our case study,

Table 5.24: Performance of Virtex II Pro implementations for the integration design.

Area	
BUFGMUXs:	1/16 (6%)
External IOBs:	37/140 (26%)
MULT18X18s:	12/12 (100%)
SLICEs:	211/1408 (14%)
Clock Cycles	1 for 1 output
Max Delay	0.718 ns
Min period /	4.966 ns /
Max Frequency	201.371 MHz
Power	
Temperature	37 $^{\circ}C$
Estimation	450 mW

the proposed pICA algorithm speeds up the FastICA algorithm by a factor ranging from 2.4 to 5.7. The function model derived from the proposed component clustering algorithm has the best mapping performance compared to other function models. The cyclic process modeling is very effective for complex image processing algorithms such as pICA. The proposed load attraction mapping algorithm improves load variances of by 5 to 15 times compared to other existing mapping algorithms, and is close to the optimal mapping result. The proposed communication attraction mapping algorithm improves cut weights of existing mapping approaches by 4.15% to 47.22%, and is also close to the optimal mapping. In addition, the image processing IP designs perform better than existing works with similar setups, and provide reliable integration components for specific VSN applications.

Chapter 6

Conclusions and Future Work

In this dissertation, we have proposed several approaches to fast image processing in resource-constrained visual sensor networks (VSNs). Since VSNs employ content-rich 2-D images or image sequences as basic media, challenges such as high volume data transmission and high speed image processing have been brought to various research communities, and the design of microsensors with on-board image processing capability remains one of the most challenging problems because of limitations on energy consumption and communication bandwidth.

6.1 Summary of Contributions

Three contributions are made in this dissertation, including algorithm improvement using parallel computing, function and data modeling, clustering, and mapping in heterogeneous environment, and development of the application-driven virtual microsensor platform.

First of all, many image processing related applications, such as independent component analysis (ICA), involve high volume of data and complex algorithms. Parallel and pipelined computing are common solutions to speed up the processing. Therefore, we utilize SIMD parallelism and present a parallel ICA (pICA) algorithm that distributes the computation burden from a single process to multiple sub-processes in parallel without the loss of quality. Mean-

while, a pICA performance prediction model is developed based on the LogP model. The performance comparison experiments conducted in the MPI environment with 10 computers showed that pICA accelerated the overall processing time by 2.4 to 5.7 times compared to the FastICA algorithm. The scalability experiment conducted on 2 to 10 processors demonstrated that the computing time steadily decreased when we increased the number of processors. The scalability experiment conducted using different data sizes showed that the transmission time of the observation data set was only a small portion of the overall processing time. Therefore, the parallel computing of ICA is mainly a computation improvement problem, instead of a trade-off problem between computations on individual computers and communications between computers on the network. Finally, the validation of the performance prediction model showed its effectiveness in performance analysis of pICA for different computing environments.

The second contribution we made is to conduct modeling, partitioning, clustering, and mapping for function and data to more efficiently implement image processing algorithms. Since a microsensor is an autonomous entity supplied by a battery with limited power, all aspects of computing, communication, resource, and interactions between each other should be comprehensively considered. In order to partition image processing algorithms in finer granularity, we proposed a multi-weight operation level function model, which decomposes an image processing algorithm into basic arithmetic operations. These operations, represented as components, are then connected by edges according to the processing flow. Multiple weights are assigned to components and edges in respect to performance parameters such as processing speed, utilization area, and data transmission volume. A component grouping algorithm and cyclic process modeling were also proposed in order to provide function models with appropriate granularity to the mapping process. In parallel to function partitioning and clustering, we analyzed data dependency respectively from the independency, uniform, and regional dependencies for 2-D and 3-D images. By isolating the regional dependency but retaining independency and uniform dependency, a data set is effectively partitioned and distributed to multiple resources for parallel

processing. Based on the operation level function model, we propose the load attraction and the communication attraction mapping algorithms, and the K-L algorithm-based local refinement for function mapping in a heterogeneous environment. The proposed algorithms allow users to efficiently map functions to limited computing resources for different objectives, including balancing load, minimizing communication or overall processing cost. Load attraction is an appropriate mapping algorithm for computing resources with large variance and applications with large variance on component weights. Communication attraction performs better mapping for communication resources with large variance and applications with large variance on edge weights. In our case study, the function model processed by the component clustering algorithm had the best mapping performance compared to other function models. The cyclic process modeling was very effective for complex image processing algorithms. The proposed load attraction mapping algorithm improved load variances over other existing mapping algorithms by 5 to 15 times, and was close to the optimal mapping. The proposed communication attraction mapping algorithm improved the cut weights over other existing mapping approaches by 4.15% to 47.22%, and was also close to the optimal mapping.

With the improved algorithms and efficient partitioning and mapping methods, the third contribution of this dissertation is the implementation of the microsensor design for various image processing applications. In general, microsensor design is driven by specific applications. In order to satisfy the resource constraints and prolong lifetime in the real environments, microsensors should include only necessary functions. We therefore transfer the reuse and reconfiguration features in hardware implementation and integration procedure to a higher design level, and present the virtual microsensor platform that consists of the sensing, digital processing, and communication sections. Each section is expandable and contains various IP blocks. By using the virtual microsensor platform, users can quickly design and implement microsensors according to the specific requirements of different applications. During the development of virtual microsensor platform, we focus on the digital processing section that mainly contains

a real-time image processing IP library. We design four image processing IPs, including the contrast stretching and the polynomial approximation-based geometric correction as examples of point-based processing, the 3×3 filter as an example of neighborhood-based processing, and the pICA algorithm as an example of image-based processing. All of the four image processing algorithms use either pipelined, parallel, or both computing structures. For the contrast stretching and the 3×3 filter, multiple designs with different performance features were proposed and compared. Experimental results of IP implementations on the prototyping FPGAs showed that the proposed image processing IP designs performed better than existing works, and provided reliable integration components with different performances. Through the experiment of microsensor integration, we also found that the proposed virtual platform-based microsensor design methodology provided a fast and reliable microsensor design environment for specific VSN applications.

6.2 Directions of Future Work

The ideas and concepts in this dissertation offer promising solutions to the problem of fast image processing in VSNs. They also suggest several interesting avenues for future research.

- **IP library expansion.** In this dissertation, the structure of the virtual microsensor platform and the initial image processing IP library have been developed. It is attractive to incorporate more DSP blocks in the sensing section, more image processing algorithms in the digital processing section, and more network protocols in the communication section for various sensor network applications.

- **Integration framework development.** In order to efficiently process images and image sequences in the proposed virtual microsensor platform, it is necessary to develop one or more integration frameworks that include data distribution and compression. These frameworks also integrate image processing IPs at different processing stages. The de-

velopment of the integration framework is an expansion and improvement of the proposed virtual microsensor platform.

- **Physical implementation.** In the implementation of virtual microsensor platform, various image processing IPs have been synthesized for target prototyping FPGAs. The microsensor design desires implementation on SoCs, which will take a long fabrication period.

Bibliography

Bibliography

[1] M. Afridi, A. Hefner, D. Berning, C. Ellenwood, A. Varma, B. Jacob, and S. Semancik. Mems-based embedded sensor virtual components for system-on-a-chip (soc). *Solid-State Electronics*, 48(10/11):1777–1781, Oct 2004.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data*, Seattle, Washington, June 1998.

[3] J. Agre and L. Clare. An integrated architecture for cooperative sensing networks. *Computer*, 33(5):106–108, May 2000.

[4] T. Akiyama, H. Yamashita, T. Hara, S. Kato, S. Shimojo, and S. Nishio. A proposal of pipelined image processing in a grid environment. In *Proc. International Symposium on Applications and the Internet Workshops (SAINT)*, pages 596–601, 26-30 Jan. 2004.

[5] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, August 2002.

[6] C.J. Alpert, Jen-Hsin Huang, and A.B. Kahng. Multilevel circuit partitioning. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, Aug. 1998.

[7] H. Ameen, M.and Szu. Early vision image analyses using ica in unsupervised learnin g ann. In *IJCNN '99. International Joint Conference on Neural Networks*, volume 2, pages 1022–1027, 10-16 July 1999.

[8] Amirix. Ap100 and ap1000 platform fpga development boards and reference designs. Technical report, Amirix Systems Inc., `http://www.amirix.com/products/`, 2004.

[9] H.C. Andrews and B.R. Hunt. *Digital Image Restoration*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[10] ARM. Processor core overview. Technical report, ARM company, `http://www.arm.com/products/CPUs/`, 2002.

[11] R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Proc. 7th IEEE Heterogeneous Computing Workshop*, pages 79–87, 1998.

[12] S. Bakshi and D.D. Gajski. A scheduling and pipelining algorithm for hardware/software systems. In *Proc. Tenth International Symposium on System Synthesis*, pages 113–118, 17-19 Sept. 1997.

[13] S. Bakshi and D.D. Gajski. Partitioning and pipelining for performance-constrained hardware/software systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 7(4):419–432, Dec. 1999.

[14] S. Banerjee, T. Hamada, P.M. Chau, and R.D. Fellman. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE Transactions on Signal Processing*, 43(6):1468–1484, Jun. 1995.

[15] M. Bartlett and T. Sejnowski. *Viewpoint invariant face recognition using independent component analysis and attractor networks*, chapter Neural Information Processing Systems-Natural and Synthetic 9, pages 817–823. MIT Press, 1997.

[16] BBNT. SensIT august '00 experiment plan. Technical report, DARPA SensIT Program, 2000.

[17] A. J. Bell and T. J. Sejnowski. An information-maximisation approach to blind separation and blind deconvolution. *Neural Computation*, 7(6):1129–1159, 1995.

[18] L. Benini, D. Bruni, N. Drago, F. Fummi, and M. Poncino. Virtual in-circuit emulation for timing accurate system prototyping. In *15th Annual IEEE Int. ASIC/SOC Conf.*, pages 49–53, Sep 2002.

[19] J. Bieger, S. Huss, M. Jung, S. Klaus, and T. Steininger. Rapid prototyping for configurable system-on-a-chip platforms: a simulation based approach. In *Proc. 17th Int. Conf. on VLSI Design*, pages 577–582, 2004.

[20] Mark L. Bilderback and Prashant Soni. Integrating efficient partitioning techniques for graph oriented applications.

[21] S.H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, Jan. 1988.

[22] J. Boros. Image processing for fine arts. In *Web Proceedings of Central European Seminar on Computer Graphics (CESCG)*, page `http://www.cescg.org/CESCG97/boros/`, 1997.

[23] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping

a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[24] S. Brini, D. Benjelloun, and F. Castanier. A flexible virtual platform for computational and communication architecture exploration of dmt vdsl modems. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 164–169, 2003.

[25] R. R. Brooks and S. S. Iyengar. Robust distributed computing and sensing algorithm. *Computer*, 29(6):53–60, June 1996.

[26] R. R. Brooks and S. S. Iyengar. *Multi-sensor fusion: fundamentals and applications with software*. Prentice Hall, Inc, New Jersey, 1997.

[27] M. P. S. Brown, W. N. Grundy, D. Lin, N. Cristianini, C. W. Sugnet, T. S. Furey, M. Ares Jr., and D. Haussler. Knowledge-based analysis of microarray gene expression data by using suport vector machines. *Proc. Natl. Acad. Sci*, 97:262–267, 2000.

[28] T. Bultan and C. Aykanat. Circuit partitioning using parallel mean field annealing algorithms. In *Proc. of the Third IEEE Symp. on Parallel and Distributed Processing*, pages 534–541, Dec 2-5 1991.

[29] A.E. Caldwell, A.B. Kahang, A.A. Kennings, and I.L. Markov. Hypergraph partitioning for vlsi cad: methodology for heuristic development, experimentation and reporting. In *Proc. of the 36th Design Automation Conference*, pages 349–354, Jun 21-25 1999.

[30] Z. Cao, Z. Ji, and M. Hu. An image sensor node for wireless sensor networks. In *Proc. Int. Conf. on Information Technology: Coding and Computing*, volume 2, pages 740–745, Apr 2005.

[31] J.P. Castellano, D. Sanchez, O. Cazorla, and A. Suarez. Pipelining-based tradeoffs for hardware/software codesign of multimedia systems. In *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*, pages 383–390, 19-21 Jan. 2000.

[32] G. Cauwenberghs. Neuromorphic autoadaptive systems and independent component analysis. Technical report, Johns Hopkins University, `http://bach.ece.jhu.edu/gert/yip/`, 2003.

[33] A. Celik, M. Stanacevic, and G. Cauwenberghs. Mixed-signal real-time adaptive blind source separation. In *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS'2004)*, Canada, May 2004.

[34] A. Cerpa, J. Elson, M. Hamilton, and J. Zhao. Habitat monitoring: application driver for wireless communications technology. In *2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, April 2001.

[35] K.S. Chatha and R. Vemuri. A tool for partitioning and pipelined scheduling of hardware-software systems. In *Proc. 11th Int. Symp. on System Synthesis*, pages 145–151, Dec. 1998.

[36] K.S. Chatha and R. Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 10(3):June, 193-208 2002.

[37] H.-D. Cheng, H.-S. Don, and L.T. Kou. Vlsi architecture for digital picture comparison. *IEEE Trans. on Circuits and Systems*, 36(10):1326–1335, Oct. 1989.

[38] K.S. Cho and S.Y. Lee. Implementation of infomax ica algorithm with analog cmos circuits. In *Proc. Int. Workshop on Independent Component Analysis and Blind Signal Separation*, USA, Dec 2001.

[39] Chee-Yee Chong and S.P. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247 – 1256, Aug 2003.

[40] M.H. Cohen and A.G. Andreou. Analog CMOS integration and experimentation with an autoadaptive independent component analyzer. *IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing*, 42(2):65–77, Feb 1995.

[41] P. Comon. Independent component analysis, a new concept. *Signal Processing*, 36(3):287–314, April 1994.

[42] Sensoria Corporation. sGate developer's platform. `http://www.sensoria.com/sgate.html`.

[43] P. Coussy, A. Baganne, and E. Martin. Virtual component ip re-use in telecommunication systems design: a case study of mpeg-2/jpeg2000 encoder. In *9th Int. Conf. on Electronics, Circuits and Systems*, volume 2, pages 733–736, Sep 2002.

[44] T.M. Cover and J.A. Thomas. *Element of Information Theory*. John Wiley & Sons, 1991.

[45] P.E. Crandall and M.J. Quinn. Data partitioning for networked parallel processing. In *Proc. of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 376–379, 1-4 Dec. 1993.

[46] Crossbow: smarter sensors in silicon. `http://www.xbow.com/Products/Wireless\_Sensor\_Networks.htm`.

[47] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. Eicken. LogP: Towards a realistic model of parallel computation. *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[48] A. Dandalis and V.K Prasanna. Mapping homogeneous computations onto dynamically configurable coarse-grained architectures. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 314 – 315, April 1998.

[49] K. A. Delin and S. P. Jackson. Sensor web for in situ exploration of gaseous biosignatures. In *Proceedings of 2000 IEEE Aerospace Conference*, Big Sky, MT, March 2000.

[50] Digilent Inc., `http://www.digilentinc.com/Products/Detail.cfm?Prod=XUPV2P&Nav1=Product%s&Nav2=Programmable`. *Virtex-II Pro Development System, Curriculum on a Chip*, 2005.

[51] Web documents. Mpich - a portable implementation of mpi. Technical report, Argonne National Laboratory, `http://www-unix.mcs.anl.gov/mpi/mpich/`, 2004.

[52] Jack Dongarra, Jerzy Wasniewski, and Kaj Madsen. *Applied Parallel Computing*. Springer, 2006.

[53] T.H. Drayer, W.E.IV King, J.G. Tront, R.W. Conners, and P.A. Araman. Using multiple fpga architectures for real-time processing of low-level machine vision functions. In *Proc. of the 1995 IEEE IECON 21st International Conference on Industrial Electronics, Control, and Instrumentation*, volume 2, pages 1284–1289, 6-10 Nov. 1995.

[54] H. Du and H. Qi. A reconfigurable fpga system for parallel independent component analysis. *EURASIP Journal on Embedded Systems*, Under Publication.

[55] H. Du, H. Qi, and G. D. Peterson. Modeling mobile-agent-based collaborative processing in sensor networks using generalized stochastic petri nets. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 1, pages 563–568, Washington, D.C., Oct 5-8 2003.

[56] H. Du, H. Qi, and X. Wang. A parallel independent component analysis. In *Proc. IEEE International Conference on Parallel and Distributed Systems*, volume 1, page 151160, 2006.

[57] Hongtao Du and Hairong Qi. An fpga implementation of parallel ica for dimensionality reduction in hyperspectral images. In *Proc. IEEE International Geoscience and Remote Sensing Symposium*, volume 5, pages 3257 – 3260, 2004.

[58] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. John Wiley & Sons, 2nd edition, 2001.

[59] Shantanu Dutt and Wenyong Deng. Vlsi circuit partitioning by cluster-removal using iterative improvement techniques. In *Proc. of IEEE/ACM Int. Conf. on Computer-Aided Design*, pages 194–200, Nov 10-14 1996.

[60] D. Eadie, F. Shevlin, and A.P. Nisbet. Geometric correction of image distortion using fpgas. In *Proc. of SPIE Conference on Optical Metrology, Imaging and Machine Vision*, volume 4877, pages 28–37, Galway, Ireland, Sep. 2002.

[61] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. of the 19th Design automation Conf.*, pages 175–181, 1982.

[62] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *Proc. 7th IEEE Heterogeneous Computing Workshop*, pages 184–199, 1998.

[63] N. Frohlich, B.M. Riess, U.A. Wever, and Q. Zheng. A new approach for parallel simulation of vlsi circuits on a transistor level. *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, 45(6):601–613, Jun. 1998.

[64] J. Gaisler. Hardware ip: Leon core. Technical report, LEOX.org, `http://www.leox.org/resources/hw.html`, 2001. Free Hardware and Software resources for System on Chip.

[65] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, `http://www.netlib.org/pvm3/book/pvm-book.html`, 1994.

[66] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli. Automotive virtual integration platforms: why's, what's, and how's. In *Proc. Int. Conf. on Computer Design: VLSI in Computers and Processors*, pages 370–378, San Jose, USA, Sep 2002.

[67] T. Givargis and F. Vahid. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1317–1327, 2002.

[68] M. Gokhale, J. Frigo, K. McCabe, J. Theiler, and D. Lavenier. Early experience with a hybrid processor: k-means clustering. In *Proc. First International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2000.

[69] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, third edition, 2002.

[70] ARC Group. Wireless ITS - intelligent transportation systems signaling, traffic control and information systems. Technical report, ARC, 2001.

[71] P. Hansen and K.-W. Lih. Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 41(6):769–771, Jun. 1992.

[72] J. Henkel and R. Ernst. A hardware/software partitioner using a dynamically determined granularity. In *Proc. of the 34th ACM IEEE annual conference on Design automation*, pages 691–696, Anaheim, CA, 1997.

[73] S. Hwang. Control of parallel task granularity by throttling decomposition. In *Proc. of Intl. Conf. on Parallel and Distributed Systems*, pages 802–807, 10-13 Dec. 1997.

[74] A. Hyvarinen. Survey on independent component analysis. *Neural Computing Surveys*, pages 94–128, 1999.

[75] A. Hyvärinen and E. Oja. A fast fixed-point algorithm for independent component analysis. *Neural Computation*, pages 1483–1492, 1997.

[76] Inter-Networking Research Group (i NRG). Visual sensor networks. Technical report, University of California Santa Cruz, `http://inrg.cse.ucsc.edu/vsn.html`, 2006.

[77] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. Assoc. Comput. Mach.*, 24(2):280–289, April 1977.

[78] M.A. Iqbal, S. Iqbal, and M.E. Shaaban. Partitioning of image processing tasks on heterogeneous computer systems. In *Proc. Heterogeneous Computing Workshop*, pages 43–50, 26 April 1994.

[79] A.K. Jain, A. Topchy, M.H.C. Law, and J.M.; Buhmann. Landscape of clustering algorithms. In *Proc. the 17th Int. Conf. on Pattern Recognition*, volume 1, pages 260 – 263, 23-26 Aug. 2004.

[80] S. Jung and S. Lee. A 4-way pipelined processing architecture for three-step search block-matching motion estimation. *IEEE Transactions on Consumer Electronics*, 50(2):674–681, May 2004.

[81] A. B. Kahng. Futures for partitioning in physical design. In *Proc. of Int. Symp. on Physical Design*, pages 190–193, Monterey, CA, USA, 1998.

[82] A. Kaplan, M. Sarrafzadeh, and R. Kastner. A survey of hardware/software system partitioning. Technical report, University of California at Santa Barbara, `http://www.ece.ucsb.edu/~kastner/ece253/reader/kaplan03.pdf`, 2003.

[83] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, Mar. 1999.

[84] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proc. of the 36th Design Automation Conference*, pages 343–348, Jun 21-25 1999.

[85] W. Ke and K. Truong. Design with testability for a platform-based soc design methodology. In *Proc. The First IEEE Asia Pacific Conf. on ASICs*, pages 307–310, Aug 1999.

[86] D. Kearney, G. Veldman, and D. Warren. Abstractions and primitives enabling runtime resource allocation for dynamic ip cores using virtual platform fpgas. In *Proc. IEEE Int. Conf. on Field-Programmable Technology (FPT)*, pages 403–406, Dec. 2003.

[87] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–307, 1970.

[88] A. Khokhar, V.K. Prasanna, M. Shaaban, and C. Wang. Heterogeneous supercomputing: Problems and issues. In *Proc. Workshop on Heterogeneous Processing*, Mar 1992.

[89] C.-T. King, W.-H. Chou, and L.M. Ni. Pipelined data parallel algorithms-i: concept and modeling. *IEEE Trans. on Parallel and Distributed Systems*, 1(4):470–485, Oct. 1990.

[90] C.-T. King, W.-H. Chou, and L.M. Ni. Pipelined data parallel algorithms-ii: design. *IEEE Trans. on Parallel and Distributed Systems*, 1(4):486–499, Oct. 1990.

[91] A. N. Knaian. A wireless sensor network for smart roadbeds and intelligent transportation systems. Master's thesis, Massachusetts Institute of Technology, June 2000.

[92] Computational NeuroSystems Laboratory. Digital implementation of independent component analysis algorithm. Technical report, Dept. of Biosystems, Korea Advanced Institute of Science and Technology, `http://cnsl.kaist.ac.kr/Research/kscho/icachip.htm`, 2003.

[93] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart, and Winston, 1976.

[94] T.W. Lee, M.S. Lewicki, and T.J Sejnowski. Ica mixture models for unsupervised classification of non-gaussian classes and automatic context switching in blind signal separation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 22(10):1078–1089, 2000.

[95] T. Leighton, F. Makedon, and S.G. Tragoudas. Approximation algorithms for vlsi partition problems. In *IEEE Int. Symp. on Circuits and Systems*, volume 4, pages 2865–2868, May 1-3 1990.

[96] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee. Pilchard - a reconfigurable compouting platform with memory slot interface. In *Proc. The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 01)*, page 170179, Rohnert Park, Calif, USA, April-May 2001.

[97] A.B. Lim, J.C. Rajapakse, and A.R. Omondi. Comparative study of implementing icnns on fpgas. In *Proc. of Int. Joint Conf. on Neural Networks*, volume 1, pages 177–182, Jul 2001.

[98] Shun-Tian Lou and Xian-Da Zhang. Fuzzy-based learning rate determination for blind source separation. *IEEE Trans. on Fuzzy Systems*, 11(3):375–383, June 2003.

[99] R. C. Luo and M. G. Kay. Multisensor integration and fusion in intelligent systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(5):901–931, September/October 1989.

[100] W.J.C. Melis, P.Y.K. Cheung, and W. Luk. Image registration of real-time video data using the sonic reconfigurable computer platform. In *Proc. on the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 22-24 Apr. 2002.

[101] G. Mole, M. Strik, and M. Muschol. Philips semiconductors next generation architectural ip reuse developments for soc integration. Technical report, Philips Semiconductors, `http://www.us.design-reuse.com/articles/article9769.html`, 2005.

[102] Y. Mtsuyama, T. Nimoto, N. Katsumata, Y. Suzuki, and S. Furukawa. α-EM algorithm and α-ICA learning based upon extended logarithmic information measures. In *Proc. of the IEEE-INNS-ENNS Int. Joint Conf. on Neural Networks*, volume 3, pages 351–356, 24-27 July 2000.

[103] NASA, Jet Propulsion Laboratory, California Institute of Technology, `http://popo.jpl.nasa.gov/html/aviris.concept.html`. *AVIRIS concept*, 2001.

[104] B. Nickerson and R. Lally. Development of a smart wireless networkable sensor for aircraft engine health management. In *Proc. of 2001 Aerospace Conf.*, volume 7, pages 3255–3262, 2001.

[105] A. Nordin, C. Hsu, and H. Szu. Design of FPGA ICA for hyperspectral imaging processing. In *Proc. SPIE, Wavelet Applications VIII*, volume 4391, pages 444–454, 2001.

[106] K. Obraczka, R. Manduchi, and J. Garcia-Luna-Aveces. Managing the information flow in visual sensor networks. In *Proc. The 5th Int. Symp. on Wireless Personal Multimedia Communications*, volume 3, pages 1177–1181, Oct 2002.

[107] Y. Onishi, M. Muraoka, M. Utsuki, and N.; Tsubaki. Vcore-based platform for soc design. In *Proc. Asia and South Pacific Design Automation Conf., ASP-DAC*, pages 453–458, Jan 2003.

[108] ORTC. *International Technology Roadmap for Semiconductors*, 2001.

[109] C. Ou and S. Ranka. Parallel incremental graph partitioning. *IEEE Trans. on Parallel and Distributed Systems*, 8(8):884–896, Aug. 1997.

[110] Neungsoo Park, Jongwsoo Bae, and V.K. Prasanna. Synthesis of vlsi architectures for tree-structured image coding. In *Proc. of Int. Conf. on Image Processing*, volume 2, pages 999–1002, Sept.16-19 1996.

[111] D.A. Patterson and J. Hennessey. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Pub, 3 edition, June 2002.

[112] P. Paulin, C. Pilkington, and E. Bensoudane. Stepnp a system-level exploration platform for network processors. *IEEE Design & Test of Computers*, 19(6):17–26, Nov.-Dec. 2002.

[113] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, Jan. 2006.

[114] K. Pister. Smart Dust: autonomous sensing and communication in a cubic millimeter. http://robotics.eecs.berkeley.edu/~pister/SmartDust/, 2001.

[115] J. Plosila, P. Liljeberg, and J. Isoaho. Pipelined on-chip bus architecture with distributed self-timed control. In *Proc. Int. Symp. on Signals, Circuits and Systems*, volume 1, pages 257–260, Jul. 2003.

[116] H. Qi. *A High-Resolution, Large-Area, Digital Imaging System*. PhD thesis, North Carolina State University (NCSU), Raleigh, NC, USA, 1999.

[117] M. Rabinovich, K. Robarge, J. Szmyd, , and J. Carletta. An experimental evaluation of algorithms for vlsi partitioning. Technical report, Case Western Reserve University, 1998.

[118] R. Reis and J.A.G. Jess, editors. *Design of System on a Chip*. Kluwer Academic Publishers, 2004.

[119] S.A. Robila and P.K. Varshney. A fast source separation algorithm for hyperspectral image processing. In *Proc. of IEEE Int. Geoscience and Remote Sensing Symp.*, volume 6, pages 3516–3518, 24-28 June 2002.

[120] Rockwell. Wireless sensing network (WSN). http://wins.rsc.rockwell.com/.

[121] SensorView. http://www.sensorview.com/.

[122] H. Saruwatari, H. Yamajo, T. Takatani, T. Nishikawa, and K. Shikano. Parallel structured independent component analysis for SIMO-model-based blind separation and deconvolution of convolutive speech mixture. In *Proc. of the Int. Joint Conf. on Neural Networks*, volume 1, pages 714–719, July 20-24 2003.

[123] F. Sattar and C. Charayaphan. Low-cost design and implementation of an ica-based blind source separation algorithm. In *Proc. on the 15th Annual IEEE Int. ASIC/SOC Conf.*, pages 15–19, 2002.

[124] SemiconFarEast. System on a chip (soc). Technical report, SemiconFarEast.com, `http://www.semiconfareast.com/soc.htm`, 2005.

[125] F.Y. Shih, T. K. Chung, and C.C. Pu. Pipeline architectures for recursive morphological operations. *IEEE Trans. on Image Processing*, 4(1):11–18, Jan. 1995.

[126] B. Shim and J.C. Suh. Pipelined vlsi architecture of the viterbi decoder for imt-2000. In *Proc. Global Telecommunications Conference (GLOBECOM)*, volume 1a, pages 158–162, 1999.

[127] M. Southworth and D. Godso. Chem-bio sensor platform leverages pc/104. *COTS Journal*, pages 74–76, March 2003.

[128] M. Srivastava, R. Muntz, and M. Potkonjak. Smart kindergarten: sensor-based wireless networks for smart developmental problem-solving environment. In *Proceedings on 7th International Conference on Mobile Computing and Networking (MobiCom 2001)*, pages 132–138, New York, 2001.

[129] G. Stitt and F. Vahid. A decompilation approach to partitioning software for microprocessor/fpga platforms. In *Proc. Design, Automation and Test in Europe*, volume 1, pages 396–397, 2005.

[130] STRJ. *JEITA STRJ Report*, 1999.

[131] University of Tennessee, Knoxville, Tennessee., `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`. *MPI: A Message-Passing Interface Standard*, version 1.1 edition, October 1998.

[132] S. Wakabayashi, K. Isomoto, T. Koide, and N. Yoshida. A systolic graph partitioning algorithm for vlsi design. In *IEEE Int. Symp. on Circuits and Systems*, volume 1, pages 225–228, May 30 - Jun 2 1994.

[133] B. W. West, P. G. Flikkema, T. Sisk, and G. W. Koch. Wireless sensor networks for dense spatio-temporal monitoring of the environment: a case for integrated circuit, system, and network design. In *Proceedings of 2001 IEEE CAS Workshop on Wireless Communications and Networking*, August 2001.

[134] Wikimedia Foundation, Inc., `http://en.wikipedia.org/wiki/Real-time`. *Real-time computing*.

[135] wikipedia.org. System-on-a-chip structure. `http://en.wikipedia.org/wiki/SoC#SOC_structure`.

[136] P. Willet. Recent trends in hierarchical document clustering: a critical review. *Information Processing and Management*, 24(5):577– 597, 1988.

[137] Y. Wu, C Cheung, D.I. Cheng, and H. Fan. Further improve circuit partitioning using gbaw logic perturbation techniques. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 11(3):451–460, Jun. 2003.

[138] Xilinx. Xilinx xup virtex ii pro development system. Technical report, Xilinx Inc., `http://www.xilinx.com/univ/xupv2p.html`, 2006.

[139] Xilinx Company, `http://direct.xilinx.com/bvdocs/publications/ds031.pdf`. *Virtex-II Platform FPGAs: Complete Data Sheet*, March 2004.

[140] T. Yamaguchi and K. Itoh. An algebra solution to independent component analysis. *Optics Communications*, 178:59–64, 2000.

[141] X. Yang, K. G. Ong, W. R. Dreschel, K. Zeng, C. S. Mungle, and C. A. Grimes. Design of a wireless sensor network for long-term, in-situ monitoring of an aqueous environment. *Sensors*, 2(7):455–472, 2002.

[142] X. Yang, M. Zhu, H. Xue, J. Bian, and X Hong. A platform for system-on-a-chip design prototyping. In *Proc. 4th Int. Conf. on ASIC*, pages 781–784, Oct 2001.

[143] W. Yu and C. Charoensak. Fpga implementation of non-iterative ica for detecting motion in image sequences. In *7th Int. Conf. on Control, Automation, Robotics and Vision (ICARCV)*, volume 3, pages 1332–1336, Dec 2002.

[144] Y. Zhang, J. Xiao, and A.J. Roberts. Parallel algorithms for spatial data partition and join processing. In *Proc. 3rd International Conference on Algorithms and Architectures for Parallel Processing (ICAPP)*, pages 703–716, 10-12 Dec. 1997.

[145] Y. Zhao and G. Taubin. Real-time median filtering for embedded smart cameras. In *Proc. on IEEE Int. Conf. on Computer Vision Systems (ICVS 06)*, page 55, 04-07 Jan. 2006.

[146] Q. Zhuge, E.H.-M. Sha, X. Bin, and C. Chantrapornchai. Efficient variable partitioning and scheduling for dsp processors with multiple memory modules. *IEEE Trans. on Signal Processing*, 52(4):1090–1099, April 2004.

[147] J.Y. Zien, M.D.F. Schlag, and P.K. Chan. Multilevel spectral hypergraph partitioning with arbitrary vertex sizes. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(9):1389 – 1399, Sept 1999.

Vita

Hongtao Du was born in Beijing, P. R. China. After graduating in 1993 from Beijing 171 High School, he attended Northeastern University in Shenyang, where he received both a Bachelor of Science degree in 1997 and a Master of Science degree in 2000 from the College of Information Science and Engineering. His research area during this period was intelligent control. In the summer of 2001, he enrolled into the master program at the University of Tennessee in Electrical and Computer Engineering. In the summer of 2002, he joined the Advanced Imaging and Collaborative Information Processing group as a graduate research assistant where he completed his Master degree in 2003. In the fall of 2003, he continued on the doctoral program and completed his Doctor of Philosophy degree in 2006. He had been working on a 3-year project sponsored by the Office of Naval Research, titled “Firmware Approaches to Smart Algorithms”, and an 1-year project sponsored by the US Army Space and Missile Defense Command, titled “Smart Automated Target Recognition using Weighted Spectral and Geometric Information”. His major research areas are parallel/distributed image and signal processing, task and data partitioning, re-configurable and virtual platform, and high performance computing. He is listed on the Who’s Who in America, and is the recipient of the Extraordinary Professional Promise Award.