S. Arash Ostadzadeh

Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures

Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.Ch.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 18 december 2012 om 12:30 uur

door

Sayyed Arash OSTADZADEH

Master of Science in Computer Engineering
Ferdowsi University of Mashhad
geboren te Mashhad, Iran

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. K.L.M. Bertels	Technische Universiteit Delft, promotor
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft
Prof. Dr.-Ing. Michael Hübner	Ruhr-Universität Bochum, Duitsland
Prof. Dr.-Ing. Mladen Berekovic	Technische Universität Braunschweig, Duitsland
Prof. dr. Henk Corporaal	Technische Universiteit Eindhoven
Prof. dr. ir. Dirk Stroobandt	Universiteit Gent, België
Dr. G.K. Kuzmanov	Technische Universiteit Delft
Prof. dr. ir. F.W. Jansen	Technische Universiteit Delft, reservelid

S. Arash Ostadzadeh
Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures

Met samenvatting in het Nederlands.

Subject headings: Dynamic Binary Instrumentation, Application Partitioning, Hardware/Software Co-design.

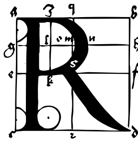
The cover images are abstract artworks created by the *Agony* drawing program developed by Kelvin (http://www.kelbv.com/agony.php).

Copyright © 2012 S. Arash Ostadzadeh
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the permission of the author.

Printed in The Netherlands

Dedicated to my dear parents

Abstract



ECENT trends show a steady increase in the utilization of heterogeneous multicore architectures in order to address the ever-growing need for computing performance. These emerging architectures pose specific challenges with regard to their programmability. In addition, they require efficient application mapping schemes to fully harness their processing power and avoid bottlenecks. In this respect, it is of critical importance to analyse application behaviour, and the data communication between tasks, in particular.

In this dissertation, we present a profiling framework that helps developers to gain an insight into the behaviour of an application. The presented profiling framework is generic and not restricted to a particular platform, application, or purpose. We utilize this framework with the primary goal of mapping applications onto a heterogeneous multicore architecture. The framework includes a memory access profiling toolset, called *Quad*, that provides quantitative information regarding the memory accesses in an application. *Quad* utilizes Dynamic Binary Instrumentation (DBI) to detect the actual data dependencies that occur between the tasks of an application at runtime. Additionally, it also provides accurate memory access measurements, such as the amount of data transferred between tasks and the memory size required for their communication. Such information can be utilized to identify critical parts of an application, to highlight coarse-grained parallelism opportunities, and to guide code optimizations.

As a proof of concept to substantiate the usefulness of the extracted profiling information, we utilize the main output of *Quad*, the Quantitative Data Usage (QDU) graph, as the input model to formulate a general application partitioning problem. The formulation of this intractable problem is flexible and accommodates different design objectives and constraints. Subsequently, we propose a heuristic algorithm to find high quality partitions of an application in a reasonable amount of time. In addition to the complexity analysis of the proposed algorithm, we present a thorough theoretical analysis of the application partitioning problem. In order to evaluate the quality of the solutions, we developed a test bench for generating synthetic QDU graphs and compared the results against the optimal partitions obtained using an exhaustive search. The comparison results show that the proposed heuristic algorithm is able to provide optimal or near-optimal solutions.

To further prove the applicability of the profiling framework, we investigate in detail the utilization of the framework in practice, by mapping two real applications onto

a heterogeneous reconfigurable architecture. To achieve this goal, we propose a hardware/software partitioning methodology that introduces the concept of merging tightly-coupled tasks based on the data communication analysis. Moreover, the profiling information is utilized to fine-tune the applications and optimize their data flow. The obtained results show a performance increase of 192% and 30%.

Acknowledgements

My interest in computers dates back to 1987, when I managed to get my hands on a Commodore 64. I can still vividly recall the day my brother came up with a magic box in his hands. All it needed was just a "poke" to make my already hypnotized eyes poke out! and yes, I do remember the magical number after all these 25 years! POKE 53280,<color code: 0-15> and bingo... you have the desired border color! Simple, but it was more than enough to cast a spell on me. If I am where I am standing today, it is because of you, Shervin. I decided to study computer science because I was enchanted by your programming skills and enthusiasm. I will never forget all those good times when I used to sit beside you, trying to learn something new about computers. You are not only a dear friend and a true brother to me, but also a perennial source of inspiration and fortitude. Thanks for your selfless support and encouragement through all these years.

This thesis is not only the outcome of my endeavor over the last years, but also the kind guidance, assistance and support of several individuals, to all of whom I am deeply grateful. Words fail to stand for the deep gratitude that I wish to express to all of you. I would like to stress the fact that the order in which I acknowledge the names is not representative of the value that I place on their roles in this respect.

First of all, I would like to sincerely thank my advisor and promotor, Prof. Koen Bertels, who gave me the chance to step into the PhD journey. Koen, I kindly value your continuous support, commitment, and patience, which immensely influenced my research view. You gave me the opportunity to develop myself in different aspects and to have a vision for future research. I am grateful for your dedication to guide me along the entire journey, our fruitful discussions, and the freedom that you granted me to pursue the research work. I am also thankful for your invaluable comments on my thesis. I would like to extend my gratitude to my defense committee for the time that they invested in reading the thesis manuscript. I appreciate their insightful discussions and suggestions to improve the quality of this thesis.

I am indebted to my dear friend, Faisal, for all the proofreading of my thesis. Faisal, I value the time we spent for the research collaboration; but above that, I highly appreciate your genial friendship. Thanks for the helping hand whenever I needed you. My appreciation also goes to Roel for the comments and discussions on our collaborative research work. Roel, thanks for helping me during the recent years in Holland. I would like to express my gratitude to Imran for his valuable contributions to the extension of this research work. Imran, you are a smart, dedicated, hardworking researcher that anyone would cherish working with. At the same time, you are a modest and trustworthy

friend. I would also like to thank Kamana for her friendship and support. I am grateful to Carlo for all the proofreading of my manuscripts and his comments over the last few years. I would also like to acknowledge Valery and Marco for their efforts to improve the *Quad* toolset, Andrew for kindly proofreading the abstract and propositions of the thesis, Roel and Motta for their translation into Dutch.

My appreciation goes to Iranian friends in Holland who helped me settle down here and made me feel at home: Mahmood, Mojtaba, Alireza, Javad, Mehdi, Behnaz, Mahyar, Rahim, Hamed, Asad, Roya, Mehdi, Gholam Reza, Vahid, Azadeh, Ashkan, Ghazaleh, Sepideh, Mohammad Reza, Reza, Masoud, Mohammad, Amin, Behzad, Hossein, Hadi, Arash, Mohamad Reza, Ali, Hossein, and other friends that I have failed to name here. A special thanks goes to Alireza and Javad for tolerating me when I was falling asleep where I was not supposed to! ... thanks for being supportive through all these years. Mahyar, I appreciate all your invaluable support and kindness.

I would also like to acknowledge my present and former colleagues in the Computer Engineering research group at TU Delft: Zubair, Luyi, Thomas, Dimitrios, Sebastian, Jae, Tariq, Fakhar, Seyab, Aqeel, Mafalda, Innocent, Laiq, Bogdan, Omar, Hamid, Roel, Saleh, Elena, Cuong, Vlad, Razvan, Muhammad, Chunyang, and Demid. I am grateful to Lidwina for taking care of all the administrative work during these years. I also wish to thank Bert, Erik and Eef for their technical support.

I would like to take this opportunity to express my sincere appreciation to all my teachers who have taught me since I went to school, and to all my wonderful friends in Iran for their prayers, kind words, and moral support.

Finally, I wish to express my deepest gratitude to my dear parents for their endless love, support, and commitment throughout my life. Mom, Dad, your incomparable love gave me the strength to overcome all the troubles that I faced in my life. You were my one and only motivation to stay and complete this journey. Mom, Dad, I endured just to see the smile on your face, which means more than the world to me. You will always be the dearest to my heart. And I am forever thankful to God because of you and all the blessings that He has given me. He has always been there for me during the saddest and happiest times of my life. Though I have been into problems, and sometimes into pains, I have always felt blessed because of believing in God. I am eternally thankful to God for all the support and guidance, for giving me the courage to face problems, for giving me the strength to fight the difficulties in life, for allowing me to learn from my mistakes, and for hearing my prayers and granting me what I wished for.

Arash Ostadzadeh — Delft, The Netherlands, August 2012

Table of contents

Abstract iii

Acknowledgements v

Table of contents vii

List of figures xi

List of tables xiii

List of listings xv

List of Acronyms xvii

Terminology xxi

1 Introduction 1
- 1.1 Problem Overview 3
- 1.2 Research Challenges 4
- 1.3 Dissertation Contributions 6
- 1.4 Dissertation Organization 7

2 Profiling 11
- 2.1 Program Profiling 13
- 2.2 Program Tracing 14
- 2.3 Profiling Usage 14
- 2.4 Profiling Classification 15
  - 2.4.1 Static Analysis 15

2.4.2 Dynamic Analysis 16
2.4.3 Static vs. Dynamic Analysis 16
2.5 Software Profiling 17
2.5.1 Instrumentation Based Profiling 18
2.5.2 Sampling Based Profiling 19
2.5.3 Simulation Based Profiling 19
2.5.4 Instrumentation vs. Sampling Based Profiling 20
2.6 Hardware Profiling 21
2.7 Data Structures for Profiling 21
2.8 Profiling Approaches 24
2.8.1 Basic Block Profiling 24
2.8.2 Control Flow Profiling 25
2.8.3 Value Profiling 26
2.8.4 Variational Path Profiling 26
2.9 Summary 27

3 Dynamic Profiling Framework **29**
3.1 Project Context 30
3.1.1 Molen Abstraction Layer 30
3.1.2 Delft Workbench 32
3.2 Q^2 Profiling Framework 34
3.2.1 *Quipu* Modeling Approach 35
3.2.2 *Quad* Memory Access Profiling Toolset 37
3.3 Runtime Memory Access Profiling 40
3.3.1 Pin Dynamic Binary Instrumentation 42
3.3.2 *Quad*-core Development 44
3.3.3 Memory Access Tracing 46
3.3.4 Identifying Memory-Intensive Kernels 49
3.3.5 Bulk Data Flow Detection 52
3.4 Memory Access Intensity Profiler (*Maip*) 56
3.4.1 *Maip* Implementation 58
3.4.2 Computation Time vs. Communication Time 58
3.5 Runtime Extraction of Source-Level Data 60
3.5.1 DWARF Debugging Information 61
3.5.2 *xQuad* Implementation 62

3.6 Kernel Ranking Based on Memory Access Intensity (MAI) 63
3.7 Summary 67

4 Temporal Memory Bandwidth Analysis **69**
4.1 Background 71
4.2 Temporal Data Extraction 72
4.3 *tQUAD* Implementation 74
4.4 Case Study: Wave Field Synthesis 77
4.4.1 Experimental Setup 79
4.4.2 Kernels Overview 80
4.4.3 Quantification of Data Communication 80
4.4.4 Temporal Information Extraction 84
4.4.5 Phase Detection 85
4.5 Summary 89

5 Task Clustering: A Greedy Approach **91**
5.1 Application Partitioning 92
5.2 Partitioning Methods 96
5.3 Problem Formulation 99
5.4 Multi-Objective Task Clustering 105
5.4.1 Input Data Model 106
5.4.2 Greedy Algorithm 107
5.4.3 Application Partitioning Algorithm 109
5.5 Complexity Analysis 114
5.5.1 Time Complexity 114
5.5.2 Space Complexity 118
5.6 Synthetic Analysis 119
5.6.1 Exhaustive Application Partitioning 120
5.6.2 Experimental Results 124
5.7 MJPEG Case Study 125
5.8 Summary 127

6 Utilizing Q^2 in HW/SW Partitioning: Case Studies **131**
6.1 HW/SW Partitioning 132
6.1.1 HW/SW Co-design: Research Directions and Challenges 134
6.1.2 HW/SW Co-design Objectives 136

6.1.3 Profile-guided HW/SW Partitioning 139
6.2 The Q^2 Partitioning Methodology 141
6.3 Canny Edge Detection 145
6.3.1 Edge Detection Overview 145
6.3.2 Experimental Setup 147
6.3.3 Experimental Analysis 148
6.3.4 Observations and Results 158
6.4 Mixed Excitation Linear Prediction 159
6.4.1 MELP Overview 159
6.4.2 Experimental Setup 161
6.4.3 Experimental Analysis 162
6.4.4 Observations and Results 168
6.5 Summary 168

7 Conclusions **171**
7.1 Summary 172
7.2 Main Contributions 173
7.3 Research Opportunities 174

Bibliography **179**

List of Publications **199**

Samenvatting **205**

Curriculum Vitae **209**

List of figures

Chapter 1

1.1 An outline of the different chapters, challenges, and contributions in this dissertation. 8

Chapter 2

2.1 An outline of the different classes of application profiling. 22

Chapter 3

3.1 An overview of the Molen Machine Organization. 32
3.2 An overview of the Delft Workbench toolchain. 33
3.3 An overview of the Q^2 profiling framework in the *Delft Workbench*. . . . 35
3.4 An overview of the *Quipu* modeling approach. 36
3.5 An architectural overview of the dynamic part of the Q^2 profiling framework. 39
3.6 Implementation overview of the *QUAD*-core tool. 45
3.7 An outline of the dynamic trie data structure of base 16. 47
3.8 Partial Quantitative Data Usage (QDU) graph of a sample application using the **libdwt** library 55
3.9 A sample Debugging Information Entry (DIE). 61

Chapter 4

4.1 An overview of the *tQUAD* implementation. 75

4.2 Memory bandwidth usage of the kernels in the *hArtes wfs*, considering only the read accesses including the stack area. 85
4.3 Memory bandwidth usage of kernels in *hArtes wfs*, considering only the write accesses excluding the stack area. 86

Chapter 5

5.1 Domain vs. functional decomposition. 93
5.2 Different application partitioning factors. 96
5.3 A typical example of the data dependency among functions in an application. 107
5.4 The outline of the task clustering algorithm. 112
5.5 The generalized harmonic number of order k of 1 ($\mathscr{H}_{k,1}$). 116
5.6 An example of partitioning a set of five elements. 122
5.7 Summary of the experimental results for synthetic data compared with the optimal partitions. 126
5.8 A partitioned QDU graph of the Motion Joint Photographic Experts Group (MJPEG) application. 128

Chapter 6

6.1 HW/SW co-design objectives. 137
6.2 The Q^2 partitioning approach. 143
6.3 The steps of the Canny Edge Detection (CED) implementation. 148
6.4 QDU graph for the hardware version of the CED application 155
6.5 Partial QDU graph of the CED application after merging. 158
6.6 Overview of the live ranges of memory blocks in the CED application. . 159
6.7 The Mixed Excitation Linear Prediction (MELP) vocoder block diagram. . 162
6.8 Partial QDU graph of the MELP application before merging 166
6.9 Partial QDU graph of the MELP application after the first merging step. . 169
6.10 Partial QDU graph of the MELP application after the second merging step. 169

List of tables

Chapter 2

2.1 Static Code Analysis vs. Dynamic Code Analysis. 17

2.2 Instrumentation vs. sampling based profiling. 20

Chapter 3

3.1 *gprof* flat profile of the *x264* application on the *Intel x86* architecture. . . 50

3.2 Summary of the data production/consumption of the **satd**- and **sad**-related kernels in the *x264* application. 51

3.3 *gprof* flat profile of the revised *x264* application, both for un-instrumented and *Quad*-instrumented binaries. 53

3.4 Summary of the data production/consumption of **pixel_satd_wxh** and the **sad**-related functions in the revised version of the *x264* application. . 54

3.5 Memory access statistics for the *hArtes wfs* application, divided in stack, heap, and data sections. 64

3.6 The *gprof* profiling data for the *hArtes wfs* application on the *Intel x86* architecture. 65

3.7 Communication vs. computation profiling data of the *hArtes wfs* application on the *Intel x86* architecture. 66

3.8 A Ranking based on the MAI of the kernels in the *hArtes wfs* application. 67

Chapter 4

4.1 *gprof* flat profile for the *hArtes wfs* application. 81

4.2 Summary of the data produced/consumed by the kernels in the *hArtes wfs* application. 82

4.3 *gprof* flat profile for QUAD-instrumented version of the *hArtes wfs* application. 83
4.4 Identified phases in the execution path of the *hArtes wfs* application. . . 87

Chapter 5

5.1 An overview of various application partitioning approaches previously appeared in literature. 100
5.2 Total number of possible partitions in an exhaustive search of the solution space regarding different problem sizes. 125
5.3 Clusters in the MJPEG application. 127

Chapter 6

6.1 *gprof* flat profile for the CED application on the *Intel x86* architecture. . 149
6.2 *gprof* flat profile for the CED application on the embedded PowerPC (PPC). 150
6.3 *MAIP* flat profile for the CED application. 151
6.4 Area predictions and theoretical speedups for the kernels in the CED application. 153
6.5 Area predictions and theoretical speedups for the merged and optimized versions of the CED application. 157
6.6 *MAIP* flat profile for the MELP application. 163
6.7 Area predictions and theoretical speedups for the kernels in the MELP application. 164
6.8 Results of the analysis of the merging options, final merged kernels, and the actual synthesis results for the MELP application. 167

List of listings

Chapter 3

3.1 Memory access tracing implementation in *Quad*-core. 48

Chapter 4

4.1 *tQuad* main interface. 76
4.2 *tQuad* instruction instrumentation. 78
4.3 *tQuad* routine instrumentation. 79

List of Acronyms

ACO	Ant Colony Optimization 140
ANSI-C	American National Standards Institute standard for the C programming language
API	Application Programming Interface 39
ASCII	American Standard Code for Information Interchange, a character-encoding scheme.
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor 134
AST	Abstract Syntax Tree 16
BB	Branch and Bound 98
BCS	Binary-Constraint Search
bpp	bits per pixel
bps	bits per second, also written as bit/s or b/s
BRAM	Block RAM, a local block of RAM on a Virtex FPGA 147
CCU	Custom Computing Unit 31
CDFG	Control and Data Flow Graph 23
CED	Canny Edge Detection 173
CFG	Control Flow Graph 15
CISC	Complex Instruction Set Computer 59
CMP	Chip Multi-Processor 2
CPI	Cycles Per Instruction 74
CPU	Central Processing Unit 12
CU	Compilation Unit 61
DAG	Directed Acyclic Graph 22
DBA	Dynamic Binary Analysis 175
DBI	Dynamic Binary Instrumentation 172
DCA	Dynamic Code Analysis 15

DCCPD Data Communication Channel Pattern Detection . . . 38

DCT Discrete Cosine Transform . . . 126

DES Data Encryption Standard

DFG Data Flow Graph . . . 23

DFL Dataflow Language, a graphical workflow language for dataflows

DFT Discrete Fourier Transform . . . 80

DIE Debugging Information Entry . . . 61

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

DSE Design Space Exploration . . . 35

DSP Digital Signal Processor or Digital Signal Processing . . . 14

DTMF Dual Tone Multi Frequency . . . 160

DWARF Debugging With Attributed Record Formats . . . 60

DWARV Delft Workbench Automated Reconfigurable VHDL generator . . . 49

DWB Delft Workbench

DWT Discrete Wavelet Transform . . . 52

EA Effective Address . . . 45

EFG Execution Flow Graph

ELF Executable and Linkable Format, formerly known as Extensible Linking Format . . . 61

ESG Extended Syntax Graph

FFT Fast Fourier Transform . . . 161

FPGA Field Programmable Gate Array . . . 23

FSM Finite State Machine . . . 24

GA Genetic Algorithm . . . 98

GCLP Global Criticality/Local Phase . . . 98

GPP General-Purpose Processor . . . 32

GPU Graphical Processing Unit . . . 133

HCDFG Hierarchical Control- and Data-Flow Graph . . . 23

HDL Hardware Description Language . . . 34

HDS Hardware Debug System . . . 21

HDVL Hardware Description and Verification Language . . . 136

HLL High-Level (Programming) Language . . . 11

HLS High-Level Synthesis . . . 133

HPC High Performance Computing . . . 1

HTG Hierarchical Task Graph . . . 24

IBF	Interleaving Balance Factor	176
IC	Integrated Circuit	132
ILP	Instruction-Level Parallelism	
ILP	Integer Linear Programming	98
IP	Intellectual Property	34
IP	Instruction Pointer, also called program counter (PC) or Instruction Address Register (IAR)	19
IPC	Instructions Per Cycle	74
IR	Intermediate Representation	21
ISE	Integrated Software Environment	162
JIT	Just-In-Time	39
KLFM	Kernighan-Lin/Fiduccia-Matheyes	98
KL	Kernighan-Lin	98
LPC	Linear Predictive Coding, a powerful speech analysis technique.	160
LUT	Look-Up Table	
MAI	Memory Access Intensity	60
MAIP	Memory Access Intensity Profiler	172
MAL	Molen Abstraction Layer	172
MAR	Memory Access Ratio	57
MAT	Memory Access Tracing	172
MELP	Mixed Excitation Linear Prediction	173
MILP	Mixed Integer Linear Programming	
MIMO	Multiple-Input and Multiple-Output	88
MJPEG	Motion Joint Photographic Experts Group	92
MOR	Memory Operand Ratio	57
MPSoC	Multi-Processor System on Chip	140
NLOC-MAR	Non-Local Memory Access Ratio	57
NLOC-MOR	Non-Local Memory Operand Ratio	57
NPP	Noise Pre-Processor	160
OSCI	Open SystemC Initiative	135
OS	Operating System	12
PCM	Pulse-Code Modulation, a method to encode digitally sampled analog signals.	162
PE	Processing Element	173
PGM	Portable GrayMap, an image file format defined by the Netpbm project.	147
PLD	Programmable Logic Device	133
PPC	PowerPC	147

PSO	Particle Swarm Optimization	98
QDU	Quantitative Data Usage	171
RP	Reconfigurable Processor	32
SA	Simulated Annealing	98
SCA	Static Code Analysis	15
SCM	Software Complexity Metric	35
SDRAM	Synchronous Dynamic Random Access Memory	
SIMD	Single Instruction Multiple Data	49
SISAL	Streams and Iteration in a Single Assignment Language	
SLDL	System-Level Design Language	136
SLIF	System-Level Intermediate Format	
SoC	System on Chip	19
SP	Stack Pointer	45
SSA	Static Single Assignment	
SSE	Streaming SIMD Extensions	49
STG	State Transition Graph	24
TB	Time Base, a counting register to keep track of system time.	161
TDFG	Task Data Flow Graph	
TLM	Transaction-Level Modeling	136
UnDV	Unique Data Value	175
UnMA	Unique Memory Address	38
USDoD	United States Department of Defence	159
UVM	Universal Verification Methodology	136
VHDL	VHSIC Hardware Description Language (VHSIC stands for Very-High-Speed Integrated Circuit)	23
VLIW	Very Long Instruction Word	
VLSI	Very-Large-Scale Integration	
VM	Virtual Machine	39
VQ	Vector Quantization	161
WCET	Worst Case Execution Time	72
WFS	Wave Field Synthesis	63
XDL	Xilinx Design Language	37
XML	eXtensible Markup Language	37
πISA	Polymorphic Instruction Set Architecture	31
ρμ-code	configuration microcode, or Reconfigurable Micro-code, used in the Molen Machine Organization.	31

Terminology

In this dissertation, we refer to several terms that are ambiguous and may specifically cause confusion when used in the context of Computer Science. In the following, we clarify the most important and frequently used terms.

Quad It stands for Quantitative Usage Analysis of Data. By data, we mean data that is communicated (produced, stored, retrieved, and consumed) via main memory in a computing system between a pair of functions. Apart from the main *Quad* tool, called *Quad*-core, there are three dependent tools that are called: *tQuad*, *cQ uad*, and *xQuad*. Each of these tools focus on a particular aspect of the analysis. ***t*** denotes *Temporal*, ***c*** marks *Communication*, and ***x*** is for *eXactitude*. The term *Quad* is generally used to refer to the whole *Quad* toolset. For more details, see Chapter 3 and Chapter 4.

Application A computer software or simply a program that is developed to perform a specific task.

Source Code Any collection of computer instructions (possibly with comments) written using some human-readable computer language, usually as text. The term 'code' may also be used as the short form of 'source code', but usually used where the nature of the code is not relevant, thus 'code' can be in any format.

Function Part of a source code that is an independent unit in relation to the rest of the source code, with clearly defined inputs (formal parameters) and outputs (results). A function can be executed as a whole by calling the function with a set of parameters. Other terms that may be used, in the general sense, for *function* include: (sub)routine, procedure, or method. Caution should be taken as these terms can have their own specific meanings in different contexts, which distinguishes them from the term *function*.

Kernel A code segment in the context of a larger application which performs a set of operations. It contributes to a relatively independent task in the context of the application algorithm. A kernel can be a function or a loop nest. Usually, a kernel consists of *consecutive* instructions of a program, however, this should not be regarded as a restriction. Since we introduce the idea of merging code segments in this thesis, in the general sense, a kernel may refer to a collection of inconsecutive code segments. Furthermore, we mostly use 'kernel' to refer to a candidate for

hardware acceleration in reconfigurable systems. In this dissertation, the terms 'function', 'kernel', and 'code segment' are used interchangeably when no confusion arises.

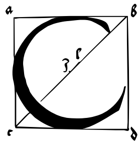
Self-contribution The self-contribution of a function refers to the execution time required by the function alone, without considering the time spent in its descendants (the functions invoked by that function). We use the expression 'entire contribution' to denote the whole execution time needed for the function including its descendants. In the case of direct recursion, the execution time of a recursive function denotes the entire contribution, and self-contribution makes no sense in this respect.

CHAPTER 1

Introduction

"If something is to be done, I have a feeling that I should start doing it." †

— Ehsan Yarshater



OMPUTING SYSTEMS today face some big challenges, at the same time, they provide exciting opportunities, due to the end of single-processor performance scaling, new demands imposed by High Performance Computing (HPC), embedded computing, and mobile computing. Furthermore, there is an ever increasing need for energy efficiency across the computing spectrum in general. In this respect, multidisciplinary research is becoming increasingly important, as the boundaries between hardware/software and general-/special-purpose processing blur in today's heterogeneous systems, as the architectures and capabilities of computing systems are becoming ever more varied, and most importantly, as applications continue to expand, both in terms of requirements and sophistication.

The growing disparity of the speed between processor(s) and the memory residing outside the chip(s), referred to as the *Memory Wall*[1], has created a severe obstacle in the performance gain of computing systems. An important reason for this disparity is the limited communication bandwidth across chip boundaries. From the mid 80s to the beginning of the 21th century, the speed of the Central Processing Unit (CPU) improved at a rate of approximately 55% each year, while the rate of improvement for memory speed was only 10%. Given these trends, it was apparent that the memory latency would (potentially) become an overwhelming bottleneck in the performance of the computing systems. Today, improvements in the CPU speed have significantly decelerated, partly due to major physical obstacles, and partly due to the fact that contemporary CPU designs have already hit the memory wall to some extent. Intel, the world's prominent

† Quoted from "***A Lifetime Quest to Finish a Monumental Encyclopedia of Iran***", An article by Patricia Cohen, The New York Times, published on 12th August 2011.

[1] The term was initially coined by Wulf and McKee in 1994 (*Hitting the Memory Wall: Implications of the Obvious*) [228]

chip manufacturer, has highlighted this issue in its *Platform 2015 White Paper*[2], which describes the evolution of the Intel's microprocessor architecture over the decade from 2005 to 2015:

> *In the past, performance scaling in conventional single-core processors has been accomplished largely through increases in clock frequency (accounting for roughly 80 percent of the performance gains). But frequency scaling is running into some fundamental physical barriers. First of all, as chip geometries shrink and clock frequencies rise, the transistor leakage current increases, leading to excess power consumption and heat. Secondly, the advantages of higher clock speeds are in part negated by memory latency, since memory access times have not been able to keep pace with increasing clock frequencies. Third, for certain applications, traditional serial architectures are becoming less efficient as processors get faster (due to the so-called Von Neumann bottleneck), further undercutting any gains that frequency increases might otherwise buy. In addition, resistance capacitance delays in signal transmission are growing as feature sizes shrink, imposing an additional bottleneck that frequency increases do not address.*

Sequential computing has dominated the computer architecture landscape for about five decades. Designers were able to design and build faster and faster computers by relying on improvements of fabrication technologies and architectural/organization optimizations. However, due to the aforementioned critical limitations, computing systems now need to achieve performance gains by other means than increasing the clock speed of Processing Elements (PEs). The main idea is that rather than performing operations in a sequence at an extremely high clock frequency, multiple PEs execute large quantities of operations in parallel at moderate clock rates to achieve increased performance. This implies that the running application should somehow be divided into (many) concurrent operational blocks and distributed among the PEs. This radical shift in application development and execution has already forced the industry to move into and promote the *concurrency era*.

The switch to multiprocessor systems has elevated concurrency as a major issue in utilizing the ever increasing number of PEs in computing systems. As a result, the most important direction in microprocessor architecture pertains to *increasing parallelism for increased performance*. The progress initially started with superscaler architectures, then came the multiprocessing functionality, and it continued with some influential capabilities such as out-of-order execution and hyper-threading. These features all laid the tiles for a major milestone in microprocessor architecture, *the movement away from a monolithic processing core to multiple cores on a single chip*. Chip manufactures are making *multicore* processor-based platforms mainstream. These platforms started with two cores and are now evolving to many more. Processors containing dozens and even hundreds of cores are already envisioned in the near future. There is no doubt that Chip Multi-Processors (CMPs) will gain control over the future microprocessor architectures, delivering excellent performance scaling, while, at the same time, solving the power consumption problem.

[2] Platform 2015: Intel® Processor and Platform Evolution for the Next Decade [41]

Recently, there has been a substantial growth in applications that require special types of processing in addition to conventional general-purpose and/or high-performance processing. This requirement, in turn, created a trend for the fabrication of chips with specialized functions, such as signal processing, media processing, and network processing. Furthermore, it even brought about versatility and adaptability as major factors in General-Purpose Processors (GPPs) to enable them to dynamically match their capabilities to a diverse range of applications. *Hardware acceleration* is an umbrella term that encompasses the idea of this special form of processing. Generally, hardware acceleration refers to the exploitation of specialized hardware to perform a function more efficiently than is possible in software running on a GPP. Examples of hardware acceleration include motion compensation in Graphical Processing Units (GPUs), matrix operations in Digital Signal Processors (DSPs), and instructions for dealing with complex numbers in conventional CPUs.

In the meantime, *reconfigurable architectures* have also attracted considerable attention due to the fact that they are identified as powerful alternatives for creating highly efficient computing systems. Reconfigurable architectures offer substantial performance improvements via custom design and reconfiguration capabilities, compared against traditional processing architectures. Reconfiguration is characterized by the ability of hardware devices to rapidly alter the functionalities of their components and the interconnection between them as needed. The primary advantage of these emerging architectures is the ability to increase performance with accelerated hardware implementation, while maintaining the flexibility of a software solution. This is generally accomplished by mapping computationally intensive parts of an application onto reconfigurable hardware. The most widely-used example of reconfigurable devices are Field Programmable Gate Arrays (FPGAs) [173]. FPGA devices are commonly perceived as co-processing units coupled with GPPs to provide hardware acceleration functionalities. In fact, a considerable share of hardware accelerators are built on top of FPGA devices. For example, they are used in automotive navigation systems and rear-seat displays, ultrasound imaging systems, robotically-assisted surgical systems, 3-D televisions, and sophisticated mobile communication systems. Xilinx [230] and Altera [6] are the world's leading providers of FPGA fabrics, which have control over 80% of the whole market[3].

1.1 Problem Overview

Heterogeneous multicore systems have gained increasing attention over the last couple of years, because the end of era for single-processor computing systems is imminent. In this respect, multiprocessor systems utilizing reconfigurable fabrics are in the focus of attention because they constitute a very interesting coupling between the performance of hardware and the flexibility of software. Reconfigurable fabrics such as FPGAs can be used as stand-alone processors or in combination with GPPs. The functions executed on the reconfigurable fabric can be changed (at runtime or at compile time) with respect to the target application. However, for this technology to really be adopted on a large scale, a number of important gaps have to be bridged, of which some are considered to be difficult. One of these challenges is the need for a machine organization that provides

[3] Xilinx, by itself, had nearly 50% of the market share in 2011.

a generic way in which different components such as a GPP and various reconfigurable devices can be combined in a transparent way. Another challenge is that we need the necessary tools to transform (existing or new) applications in such a way that we can ultimately unleash the performance of these systems to the full extent.

We need such tools because application development in this context no longer lends to the conventional sequential model. Furthermore, there is a wide range of legacy applications that need to be mapped onto these emerging architectures. Thus, in the first place, there is a critical need to thoroughly understand and analyse what is happening inside the application. In this respect, the memory access behaviour of the application is of critical importance, as it turns out that data communication is the primary obstacle in achieving the anticipated speedups in parallel systems. Moreover, application developers require detailed information about the memory accesses in applications to fine-tune and customize them for maximum performance on any given architecture.

The complexity of non-trivial applications makes it difficult to manually find the required information, hence there is an obvious need for tools to help developers in pinpointing performance bottlenecks. Extracting the potential coarse-grained parallelism to efficiently map an application onto these systems is only possible if quantitative information about the inter-task data dependencies are available. This, in turn, necessitates the development and usage of tools than can provide these information. Furthermore, an appropriate partitioning approach is needed in order to use these information for mapping the application in such a way that design objectives are met.

1.2 Research Challenges

The problem of mapping an application onto a heterogeneous reconfigurable architecture has various aspects. This includes parallelism detection, application partitioning, Design Space Exploration (DSE), among others. In this work, we do not address all the aspects of this research domain. For that reason, we only focus on some specific challenges, which were briefly mentioned earlier in this chapter and are highlighted in the following.

Challenge 1 — ***How to formulate the partitioning of an application to allow dealing with different design objectives, requirements, and purposes?***

While heterogeneous multicore architectures are excellent candidates for parallel processing, there is an important problem not solved yet. It is not easy nor straightforward to map an existing sequential application onto these systems while fully utilizing their processing potential. As these systems have multiple PEs and diverse shared resources, the problem of balancing the workload among multiple cores becomes critical. The performance of the system is not only determined by the workload of the application tasks, but also by the way these tasks communicate and share the available resources. It is not obvious how to reasonably assign tasks to each PE so that there will be no bottleneck which compromises the performance of the system. Furthermore, not all the mapping objectives are within the same direction, which makes the application partitioning, and subsequent mapping, even more complex. As an example, optimizing energy consumption in

these systems may prefer the idleness of some PEs on the availability of extra resources for certain application. This is incongruous with the workload balancing strategy among all available PEs. In this thesis, we formulate a general application partitioning problem, where various objectives are taken into account to find an appropriate solution for the partitioning problem. This will let the application developer efficiently program a heterogeneous multicore system to unleash its potential.

The problem of general application partitioning is formulated in Chapter 5 and a practical methodology to address HW/SW partitioning for a heterogeneous reconfigurable system is discussed in Chapter 6.

Challenge 2 — ***How can we accurately measure the amount of data that is transferred between different parts of an application?***

The availability of parallel processing can potentially offer extra processing power compared to sequential processing. However, while an application conventionally exploits the full capacity of a single processor, it is hard to decompose and distribute the application in a way that it actually runs faster on some parallel system. The major problem inherent in the application is data dependencies between tasks or communication costs, which limit the achievable speedup. A critical research problem for heterogeneous multicore systems is how to measure the amount of data that is transferred between a pair of tasks. This is not easy since the exact amount, in most cases, can only be determined dynamically during the execution of the application. What makes the problem complicated is the fact that we have to keep track of all the memory accesses occurring during the execution of an application in order to have an accurate estimation of inter-task data communication. The dynamic profiling framework presented in this thesis addresses this problem by identifying the actual data dependencies arising during the execution of an application.

The dynamic memory access profiling toolset is presented in Chapter 3. An extension of the profiling toolset to extract temporal information is described in Chapter 4.

Challenge 3 — ***How can different partitioning algorithms be evaluated in terms of the quality of solutions and the execution time?***

One major problem regarding different application partitioning algorithms is the lack of a robust and fair basis of comparison. This research has not been appropriately addressed since it is difficult to compare the results of different partitioning strategies. This is due to the different input models, objective functions, assumptions, test cases, and target architectures that are used in each research work. The diversity of critical factors in these works is such that it is nearly impossible or very difficult and unclear to prefer one over another. Even worse, there is no standard metric to assess the quality of the results. Apart from limited research work that propose deterministic methods to find the optimal solution, for heuristic methods, no solid proof is given to validate the quality of the found solutions. In this thesis, we present a synthetic test bench that can be used as a starting point

to allow the comparison of various partitioning algorithms.

The synthetic test bench for the comparison of partitioning algorithms and a strategy to estimate the quality of the found solutions is discussed in Chapter 5.

1.3 Dissertation Contributions

The focus of this dissertation is on memory access profiling and application partitioning. In these areas of research, we have made the following contributions.

Contribution 1 — ***An efficient memory access profiling framework that enables the extraction of detailed quantitative information from applications.*** We develop a set of memory access profiling tools, which are based on the Dynamic Binary Instrumentation (DBI) mechanism to inspect the behaviour of an application. We introduce the Quantitative Data Usage (QDU) graph as the primary output of the toolset. It not only reveals the actual data dependencies between the functions of the application, but also provides profiling data which quantifies the data communication between those functions. The information measured by this toolset is quite accurate and verified in several cases where real applications are used. Moreover, the framework is designed in a structured way to make further improvements simple and straightforward. We show how the extracted profiling information can be used in different aspects, including source code optimizations and function merging, among others.

Contribution 2 — ***The runtime extraction of the relative timing information enables the identification of different execution phases in an application.*** Although not in the primary focus of the profiling framework, the developed toolset allows the extraction of the relative temporal information during the execution of an application. This information improves the perception of the user in the sense that he can inspect at what time a particular memory access behaviour occurs in the application, or how the data communication between functions proceeds over time. Furthermore, the extracted temporal information can be utilized to give an account of the memory bandwidth requirements of the application during its execution. This is of particular importance in exploring task mapping and scheduling opportunities in multicore systems. However, in this work, we do not investigate these issues, instead we only use the temporal information to identify different phases in the application.

Contribution 3 — ***A heuristic solution for the general application partitioning problem with a customizable objective function.*** The extracted information regarding the actual data dependencies in an application along with additional quantitative profiles can be beneficial for a number of purposes. Data dependency detection triggers parallelism exploitation, which, in turn, initiates the concept of application partitioning. In our work, we first formulate the application partitioning problem in a general and flexible way, and then propose a heuristic approach to solve the problem. Although the proposed approach may not eventually converge to the optimal solution — as non-heuristic

approaches would inherently do — the intractability nature of the problem puts our approach on the plus side for large problem sizes.

Contribution 4 — ***An elaborate validation of the quality of the solutions provided by the partitioning algorithm through an exhaustive search of the solution space.***
In case a heuristic algorithm is proposed to address an optimization problem, it would be assumed a severe flaw if we cannot assess the quality of the found solutions. This is because there is no guarantee that the optimal solution is found. Thus, one should not only care about finding a solution which meets the defined constraints, but also should set the criteria to estimate the value of the found solution among all possible solutions. As such, the first step involves defining the metric against which the quality of the found solution is examined. For this purpose, we present a complete theoretical analysis of an exhaustive solution search for the proposed partitioning model. Subsequently, we perform extensive simulations — using synthetically generated input data — to investigate the actual standing position of the heuristically-found solution against all the possible solutions. This, undoubtedly, represents the best quality assessment for such a partitioning algorithm. The simulation results show that, for the majority of cases, the found solutions stand in acceptable positions in the solution space to be considered as near-optimal ones.

Contribution 5 — ***Two case studies, where the memory access profiling toolset is utilized to analyze and partition an application.***
In order to evaluate the practical usage of the developed profiling framework, we present two case studies regarding application analysis and partitioning. The first application is a well-known edge detection algorithm from the domain of image processing. The second application is an advanced voice codec featuring good voice quality even at extremely low bit rates. Based on the extracted profiling information, we proposed a hardware/software partitioning methodology to formulate the application mapping procedure. This information is used to guide merging relevant tasks of each application together, while ensuring the feasibility of porting the application to the target platform at hand.

1.4 Dissertation Organization

The remainder of this dissertation is organized in several chapters. First, we present an overview of application profiling in Chapter 2. Then, in Chapter 3, we describe the developed profiling framework and further detail the dynamic memory access profiling toolset. After that, we describe the extraction of timing information during the runtime of an application in Chapter 4. Subsequently, we discuss in detail the problem of application partitioning in Chapter 5. Chapter 6 presents two detailed case studies where the profiling framework is validated in practice. Finally, we conclude this dissertation in Chapter 7.

A visual outline of this dissertation is depicted in Figure 1.1. This figure concisely presents the relation between different chapters, the research challenges, and the con-

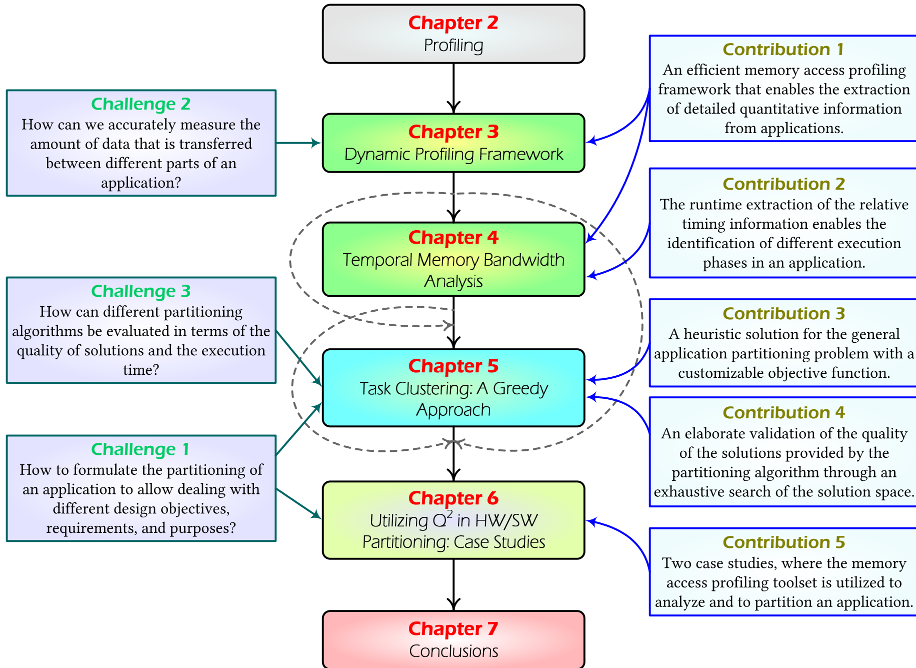


Figure 1.1: *An outline of the different chapters, challenges, and contributions in this dissertation. Chapter 2 presents only some background materials. Chapter 3 and Chapter 4 are both related to the development of the dynamic memory access profiling framework. While Chapter 5 deals with theoretical analysis of the application partitioning problem, Chapter 6 focuses only on the practical aspect of partitioning in reconfigurable systems. The dashed lines indicate the possibility of skipping intermediate chapter(s), as it will not interfere with understanding the contents.*

tributions of this work. In the following, we present a brief summary of each chapter.

Chapter 2 — *Profiling*

In Chapter 2, we start by describing the concept of profiling an application and discuss how profiling helps to analyse the behaviour of the application. We set out stressing the importance of such analysis in understanding the behaviour of applications, which, in turn, is of great value to application developers and computer architects. Profiling tools are a necessity to evaluate how well applications perform on different platforms as well as to identify the critical parts which pose potential bottlenecks for the performance of a system. Additionally, we explain the idea of application tracing in contrast to profiling and how their objectives differ in the context of application analysis. Furthermore, we describe the different aspects in which profiling can be useful. Subsequently, we list different types of data structures that are used in various profiling and tracing techniques. The choice of proper data structures has crucial effect on the performance of profilers as well as on the execution time of the profiled application itself. We describe the two main categories of profiling, namely the static analysis and the dynamic analysis. From a different perspective, we subsequently discuss the differences

between software and hardware profiling. In particular, we concisely explore different instrumentation techniques. The chapter also provides a brief account of several existing profiling tools used in analysing applications.

Chapter 3 — *Dynamic Profiling Framework*
The *QUAD* memory access profiling toolset is introduced in Chapter 3. In this chapter, we first present the project context of our work, focusing on the Molen Abstraction Layer (MAL), the *Delft Workbench* tool platform, and the Q^2 profiling framework. The chapter continues with a detailed description of the development issues in the dynamic part of the profiling framework. In particular, we elaborate on the description of the *Pin* DBI framework, and the implementation of the *QUAD*-core tool, the Memory Access Tracing (MAT) module, Memory Access Intensity Profiler (*MAIP*), and the *xQUAD* tool. Furthermore, using the profiling information extracted by *MAIP*, we set out to estimate the time spent on memory operations in distinction of the time spent on computations. Based on this estimation, we propose a ranking strategy that provides a preliminary assessment of the criticality of a function regarding its memory access intensity. In order to demonstrate how the profiling information can be interpreted and used in different aspects, we investigate three real-world applications as case studies. For each application, we highlight some major observations followed by detailed comments.

Chapter 4 — *Temporal Memory Bandwidth Analysis*
Chapter 4 presents the *tQUAD* tool as an extension to the *QUAD* toolset. It enables *QUAD* to extract relative timing information from an application during its execution. This is of critical significance, particularly with respect to task scheduling and mapping in heterogeneous multicore systems. The original *QUAD*-core tool provides no track of temporal information, mainly because of the high instrumentation overhead. The *tQUAD* tool collects the relative timing profiles as an indication of the progress of the application. We present a concise overview of how this functionality is implemented in *tQUAD*. In the presence of the memory access data, the extracted temporal profiles by *tQUAD* give an account of the memory bandwidth usage of the functions in an application over time. Additionally, we utilize the extracted temporal information to discover the different phases of an application. The chapter ends with a detailed case study of a real application to demonstrate the potential and the applicability of *tQUAD* in practice. It should be stressed that the extracted timing profiles primarily target temporal task partitioning, in contrast to the spatial task partitioning, which does not fit within the scope of this thesis. Thus, the extracted timing information is not used in subsequent chapters. Nevertheless, one may opt to utilize this extra information as hints for identifying related functions in task clustering.

Chapter 5 — *Task Clustering: A Greedy Approach*
The focus of Chapter 5 is on the problem of the coarse-grained application partitioning in its general sense. We present a detailed investigation into the factors that characterize a partitioning scheme and the methods that are utilized to perform partitioning. In addition to a comprehensive formulation of the general application partitioning problem, we propose a heuristic approach to tackle this intractable problem with the aim of working out a near-optimal (or optimal) solution in a feasible amount of time. The proposed partitioning approach utilizes

a greedy strategy with the primary objective of minimizing (and maximizing) the inter-cluster (intra-cluster) data communication, and the uniformity of the processing workload. An application partitioning algorithm is susceptible to failure without a proper input model to fully capture the data transfers in the application. To address this critical issue, we utilize the QDU graph to drive the partitioning procedure. Furthermore, we provide a detailed complexity analysis of the proposed partitioning algorithm, both in terms of time and space. We also present a thorough analysis of the application partitioning problem from a combinatorial mathematics perspective. This is required to conduct an exhaustive search of the solution space in order to have a strictly accurate assessment of how close we can get to the optimal solution. The chapter concludes with experimental results for a real-world application as well as for synthetic data in comparison with the optimal solution.

Chapter 6 — *Utilizing Q^2 in HW/SW Partitioning: Case Studies*
We demonstrate how the dynamic profiling framework can be applied in real scenarios by investigating two realistic cases in Chapter 6. For this purpose, we propose the Q^2 partitioning methodology which divides an application into hardware and software parts. We evaluate the Canny Edge Detection (CED) application, a well-known edge detection algorithm, and the Mixed Excitation Linear Prediction (MELP) application, a high-grade voice coder targeting very low bit rates. Both applications are mapped onto the Molen heterogeneous platform. To this purpose, an elaborate analysis of each application is performed beforehand. During the analysis phase, memory access profiling information provided by *QUAD* is utilized for source code modifications and optimizations. We employ the QDU graph as the main reference to analyse the data transfers between functions, find memory bottlenecks and deficiencies, and spot opportunities to merge functions.

Chapter 7 — *Conclusions*
In Chapter 7, a summary of the work in this dissertation is presented. Several conclusions with respect to the contributions anticipated in the introduction are drawn. Subsequently, the chapter lists several open issues and opportunities for future research.

Although the authors have made an attempt to document in this dissertation the work that has been carried out in the context of this research, it simply cannot be fully representative of what has been done. It is our sincere hope that an enthusiastic reader refer to the accompanying source code, which is — with no doubt — an inseparable part of this work. As Jeff Atwood perfectly puts it in a post at *Coding Horror*: **no matter what the documentation says, the source code is the ultimate truth, the best and most definitive and up-to-date documentation you're likely to find!**[4]

[4] ***Learn to Read the Source, Luke***, Jeff Atwood, April 16, 2012.

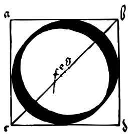
CHAPTER 2

Profiling

"Indeed, researchers love to find problems to work on." †

— Dennis M. Ritchie

In this chapter, we discuss program profiling as the primary technique to investigate the behaviour of an application in order to highlight performance issues. In addition to describing its usage, we present a classification of different profiling techniques. A particular attention is given to instrumentation as the main technique used in the development of our dynamic profiling framework.



PTIMIZING an application to execute as fast as possible on a given computing platform has never been a trivial task. Conventionally, programmers achieved such goal by carefully studying the system details, trying to find the proper combination of machine instructions that would result in the level of desirable performance. In the past, it was relatively easy for a programmer to decide on the types of code adjustments that would work best on a given architecture, partly because computing systems were functioning in a completely deterministic way. However, in the recent decade, the software development landscape has evolved dramatically, as the general public has embraced computing devices of all types and become increasingly reliant on them to accomplish everyday tasks. As the demand for more sophisticated applications increased, developers turned to use High-Level (Programming) Languages (HLLs) and frameworks in order to reduce development costs and remain competitive in the marketplace. Accordingly, applications have grown increasingly complex in terms of both code size and the interactions that occur inside them [49]. While this *layered* approach to development may save time and money in the short run, it complicates the task of determining whether an observed performance issue is internal to an application or caused by the framework that it is built upon.

† ***Reflections on Software Research***, 1983 Turing Award Lecture, Communications of the ACM, Vol. 27, No. 8, August 1984, pp. 758-760.

On top of that, computer hardware has changed drastically to keep up with the continuous demand for computing power. The simple single-issue processors of the past surrendered to super-scalar designs capable of executing multiple instructions in a single cycle, while simultaneously reordering operations to maximize the overall performance. Thus, the instructions fed into the processor have become merely a guideline for execution, as opposed to the written rules they were viewed as in the past. Since a developer now has no way to determine precisely how the processor will operate, the act of hand-tuning an application at the assembly code level is no longer a straightforward task [49].

For the last three decades, researchers have been aware of these trends in computing and have started developing tools to automate the task of application performance analysis. The initial tools, much like the computers of the time, were simple in nature and capable of gathering only basic performance statistics [89]. Furthermore, due to technical limitations, the early tools focused exclusively on quantifying application-level performance, and were unable to characterize the effects of items such as library code or the Operating System (OS) itself. As computers became more complex, however, advanced tools were developed to cut through the layers of abstraction caused by the use of advanced OSs and other development frameworks in order to gain meaningful performance statistics for the entire software system [49, 101, 134]. More recently, hardware designers have begun to embed counters inside the Central Processing Unit (CPU) that can record cache hit statistics and other meaningful information, thus allowing developers to obtain various performance profiles for their applications [10, 118, 155].

In the last couple of years, the emergence of multicore systems in general, and heterogenous reconfigurable systems in particular, is raising new requirements for their application development. These requirements appear in terms of performance, cost-efficient development, low power, functional flexibility and attainability. This increases the design complexity in terms of performance improvement, memory optimization, power optimization, etc. In order to obtain these goals, during the development process, it is necessary to identify what application or parts of an application can be implemented on different Processing Elements (PEs). In accordance with the Amdhal's law [8], in order to achieve performance improvement, it is important to identify the *critical parts* of the application to address potential bottlenecks. This indeed requires a comprehensive analysis of the application.

It is known that an application tends to spend most of its execution time in a small fraction of code, a feature known as the "80-20 rule", i.e. 80% of the execution time comes from 20% of the code [63, 190]. However, based on the application, it is generally difficult to identify at compile time where this small portion of the code lies. In order to identify the critical part(s) of the code, one needs to *profile* the application. Analysing an application at the high-level code is the first step towards any optimization. Information derived from the application analysis, such as the number of times a function has been invoked, exposes new optimization opportunities that are not visible in a traditional code optimization.

In the following, we present an overview of different application analysis techniques and their usage in today's computing systems. The remainder of this chapter is organized as follows. In Section 2.1, we present the concept of profiling as the main technique to measure the performance of an application. Section 2.2 discusses tracing an applica-

tion to record a sequence of events that occur during its execution. Profiling usage is discussed in Section 2.3. Section 2.4 presents a thorough classification of different profiling techniques and briefly describes each class. The two main categories of dynamic profiling, namely *software* and *hardware* profiling, are further detailed in Section 2.5 and Section 2.6, respectively. In Section 2.7, we list the common data structures that are used in application profiling. Section 2.8 is devoted to different approaches used to profile applications. Finally, Section 2.9 concludes this chapter.

2.1 Program Profiling

An application is the key contributor to the performance of any computing system in terms of CPU time, memory, power consumption, etc. Application analysis is extremely important for understanding such application behaviors. Computer architects need program analysis tools to evaluate how well programs perform on a given architecture. Similarly, software developers need these tools to analyze their programs and identify critical parts of the code, and compiler developers often use such tools to find out how well their techniques, such as instruction scheduling or branch prediction, are performing. Generally, the output of any program analysis tool is a statistical summary of the events observed, called a ***profile***, or is in the form of a sequence of recorded events, called a ***trace***.

Profiling is a commonly-used technique for evaluating the performance of an application. Code profilers maintain a statistical summary of performance metrics such as the execution time. These statistics are further analyzed to evaluate the performance of the program. This information is subsequently used to identify program *hotspots* in the code. The hotspots can then be analyzed further to pinpoint performance defects and/or bottlenecks.

A program hotspot is the point or segment in a program where there is a significant amount of activity. These points can be anywhere in the program, such as a memory address, an OS process, an OS thread, an executable file or module, a function, a line of code, or an instruction. The activities can vary from the time spent in processing to any kind of internal processor event. When it comes to profiling, the fact that what are these activities and exactly where they appear is the main concern. However, the fact that these activities occur infrequently or frequently is not that much of interest.

In its most common form, *program profiling* involves obtaining the execution time of a program, basically within the software code level (at the function level, the statement level, or the loop level) and analyzing it to get an idea of where the processor is spending most of its time. This will help to answer the questions like: which lines of the code are responsible for the bulk of the execution time? how many times a loop is executed? which approach is more efficient to code a block? and so on. From the implementation viewpoint, program profiling is the process of recording the summary information during program execution, e.g., execution time, number of function calls, hardware statistics in order to reflect the performance behavior of program entities such as functions, loops, basic blocks, user defined semantics, etc. In summary, program profiling reflects the performance behavior of program entities, such as functions, loops, and basic blocks. This mechanism is useful for *performance evaluation* and *code optimization*.

One of the traditional ways for code optimization is to reduce the execution time by removing the redundant code (e.g., dead code elimination, common sub expression elimination, and copy propagation). This kind of analysis can be done is various stages of the program: algorithm construction, program compilation or program execution. Another way of optimizing code is to move frequently executed program regions to infrequently executed program regions, such as loop induction variable elimination and loop invariant code removal.

2.2 Program Tracing

Unlike *program profiling*, which mainly concerns exposing the behavior of a program by monitoring its execution, *program tracing* focuses on recording a sequence of events occurring during the execution. In application analysis terminology, the *tracing problem* refers to recording enough information about the execution of a program in order to be able to reproduce the entire execution [22]. A straightforward way to solve this problem is to monitor each basic block so that whenever it executes, a unique token (also called a *witness*) is written to a trace file. In this case, to regenerate the execution, one only needs the trace file.

The difficulties in obtaining a complete program trace stem from the high cost of recording every instruction and/or data address during the execution of the program. In addition, program tracing usually results in large trace files. Straightforward, but inefficient, tracing systems examine every instruction as a program executes. The tracing overhead can be reduced by modifying either the computer hardware or the application software to record subsets of the data. Although these modifications can significantly improve tracing performance, the overhead still remains too high for many applications. Detailed program traces are used for various simulation purposes, e.g., in the design of processor instruction sets and memory systems, the study of storage reclamation and virtual memory page-replacement algorithms, and the analysis of input to parallelizing compilers. Some early approaches for capturing detailed program traces are discussed in [122].

The main problem with trace-driven simulations is that they are both time- and space-consuming, which makes them sometimes impractical. To address this problem, the execution of a program can be modeled by a statistical profile in order to generate a synthetic benchmark trace from it. This technique is used to accelerate the design process. Due to the statistical nature of this technique, performance characteristics quickly converge to a steady state solution during simulation, which makes it appropriate for fast Digital Signal Processors (DSPs) [71, 72].

2.3 Profiling Usage

Program analysis tools are extremely important for understanding the program behavior in general, while runtime program profilers are crucial for understanding the dynamic behavior of programs. In this respect, profiling can be used in various aspects:

- Profiling provides insight into a program's resource utilization and helps a programmer to identify performance bottlenecks in the program code.
- Profiling is used to guide code optimizations for compiler development. The feedback from the profile information is a useful mechanism for providing the compiler with information about how the code behaves at runtime. Having this information can lead to significant improvements in the performance of the application.
- Profiling helps to evaluate how well programs will perform on new or existing architectures from computer architecture point of view.
- Profiling is used to check the implementation of an algorithm. It helps to assess how the code carries out a given task and to check whether or not the algorithm behaves correctly. Furthermore, profiling may be used to suggest algorithm optimizations by avoiding costly implementation.
- Information obtained from profiling can also be used for program debugging and testing.

2.4 Profiling Classification

Basically, there are two different ways of analyzing programs: Static Code Analysis (SCA) and Dynamic Code Analysis (DCA). In the following, we discuss each type in more detail.

2.4.1 Static Analysis

In Static Code Analysis (SCA), the source code of a program is analysed without actually executing the program. In most cases, the static analysis is performed on some version of the source code, and in the other cases, it is performed on some form of the object code. The ability to predict, at compile time, the probability of a particular branch being taken provides valuable information for several optimizations. However, the sophistication of the static analysis performed varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis.

Several static profiling techniques are discussed in [85, 110, 160, 164, 168, 188, 216, 227], in which static estimates are derived by examining the structure of the programs, loop nests, conditional branch expressions, static call graphs, etc. In static profiling, a compiler usually applies several program-based heuristics to determine the most likely path of each branch. [21] presents a discussion of some of these heuristics. Using the heuristics in [21], [227] discusses several branch prediction techniques. For this purpose, first, several heuristics are combined to estimate the branch probabilities, and then these branch probabilities are propagated along each procedure's Control Flow Graph (CFG) to obtain local block and edge frequencies. Finally, these local estimates are used to compute function call and invocation frequencies. Furthermore, in [216], the authors present a series of static estimation techniques and measure their accuracy by comparing them to runtime program profiles. Several heuristics, similar to the ones defined in [21], are utilized

in a compiler to turn Abstract Syntax Tree (AST) structure and data flow information into branch predictions. The obtained results show that static estimates are competitive with those derived from runtime profiles. In [152], the authors present an extensive collection of SCA techniques for computing reliable information about the dynamic behavior of programs. It provides an overview of the different major approaches for SCA, such as data flow analysis, constraint-based analysis, and abstract interpretation.

Apart from predicting the dynamic behaviour of an application, SCA can also be used to produce estimates for different hardware resource utilization based on some heuristics. For example, analyzing the number of multiplications or additions can indicate the number of multipliers or adders that need to be instantiated in hardware. For a detailed discussion of using SCA to predict hardware resources refer to [142].

2.4.2 Dynamic Analysis

In Dynamic Code Analysis (DCA), a program is analysed while it is actually executing. This involves investigating the behavior of the program using the information gathered as the program runs. A usual goal of DCA is to determine which parts of a program can be optimized for speed and/or memory usage. Conventionally, in DCA, a profiler keeps track of the events that occur during the execution of the program. It involves the process of selecting a set of inputs for the program, executing the program with these inputs, and recording its runtime behavior. As an example, a compiler can use these profiles in subsequent recompilation of the program to determine the most frequent direction of each branch. In a very general sense, profiling information can be gathered in two ways: profilers that present counts of statements or routine invocations, and profilers that extract timing information about statements or routines. However, with regard to the methods for data collection, profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, OS hooks, and performance counters.

2.4.3 Static vs. Dynamic Analysis

Using static program analysis for optimization has several advantages over dynamic profiling. The most significant advantage is speed. With static analysis, there is no need to profile the program and then recompile it again. Additionally, static analysis is feasible in all environments, as it does not require runtime information. Similarly, static analysis is independent of the input data sets, allowing coverage of all branches. However, static profiles are less accurate, as they are based more on estimations. Nevertheless, runtime-independent information can best be gathered by static analysis, e.g., the number of multiplications, existence of floating points, recursions, etc. Furthermore, when hardware implementation of a program is the purpose of the profiling, hardware resource estimation does not necessarily require runtime information (see Section 3.2.1).

Dynamic analysis, on the other hand, has the benefit of accuracy in predicting branch directions as dynamic analysis is performed during the execution of the program. Furthermore, runtime information is essential when making predictions for speedup and latency of the hardware implementation of a program. Nevertheless, there are sever-

Table 2.1: *Static Code Analysis vs. Dynamic Code Analysis.*

Analysis Type	Advantage	Disadvantage
SCA	-**fast** (no program recompilation and execution) -**applicable** in all environments (no runtime information required) -**independent** of the input data (covers all execution paths)	-**inaccurate** (uses branch prediction heuristics; not fully precise) -**restricted** in scope (cannot be implemented for all purposes)
DCA	-**accurate** (runtime data collection ensures accuracy provided that appropriate input is used) -**unrestricted** in scope (can be implemented for any purpose)	-**slow** (may be slow due to the execution overhead) -**unapplicable** in some environments (not all environments support DCA) -**dependent** on the input data (the execution flow is dependent on the input data; profiles may not be completely representative of the program's behavior)

al disadvantages associated with dynamic profiling. First, the process of compilation, profiling, and recompilation in dynamic analysis is extremely time-consuming. Second, it requires programmer intervention, i.e. a programmer must instrument the program with some measurement code. Third, dynamic analysis may not be applicable in all environments, specially in embedded system environments, where gathering the dynamic profiling data is usually infeasible. Finally, with dynamic analysis, the prediction of the program flow (such as branch prediction) is dependent on the input data. When the program input changes, the process has to be repeated again with a new set of input data in order to get precise information. This is because the profiling information obtained with the new input set can be (significantly) different from the initial one. As a result, when the input data set changes, the recompilation process has to be repeated over. Table 2.1 presents a summary of the comparison between SCA and DCA.

Based on the method used for data collection, profilers can be divided into two categories: *software* profilers and *hardware* profilers. Software profilers collect the profiling information in various ways, e.g., with instrumentation, sampling, or simulation. However, hardware profilers use special hardware units to collect profiling information. In the following, we elaborate on these two categories.

2.5 Software Profiling

In *software* profiling, a compiler adds statements to a program that take time measurements as the program is running. For this purpose, it adds statements to capture the current time at the beginning and the end of a function. The time difference between these two is measured to find the time spent in the function. At the end of the program, the percentage of the program time spent in the function is calculated by dividing its total time by the total execution time of the program. Software profiling can be somehow inaccurate, as it has profiling overhead that can change the behavior of the examined program. Various methods are used to extract profiling information from programs. In

the following, we briefly describe each method.

2.5.1 Instrumentation Based Profiling

Profiling based on instrumentation works by inserting (or injecting) special code packets — called instrumentation code — at points of interest, such as function entry/exit, basic blocks, or edges in the flow graph of the application to be profiled. When the injected code is executed, it generates events, such as method entry/exit or object allocation. This data, usually in the processed form (e.g., the total time spent in each method or the total number of allocated objects for each data type), is eventually presented to the user. The main advantage of the instrumentation based profiling over other known techniques is its flexibility. Virtually any kind of data, ranging from low-level events to high-level data, can be collected with the instrumentation approach. Instrumented *hooks* record the exact number of events, such as method invocations. Moreover, they are capable of measuring precise timings, depending on the type of the instrumentation performed.

One of the problems associated with the instrumentation based profiling is that it has usually high performance overhead. This overhead can be substantially reduced if only a small part of the target application — e.g., one that has previously been identified as a performance bottleneck — is instrumented, while the rest of the application runs at full speed. Such an approach may also solve scalability issues caused by a high volume of profiling information generated by the instrumented code [66]. In an instrumentation based technique, the instrumentation code introduced into a program can change its behavior, which, in turn, can lead to skewed information being collected, i.e. the profiling information does not represent the original uninstrumented program. This problem, however, can be mitigated to some extent by minimizing the amount of inserted code.

The advantage of instrumentation based profiling is that the instrumentation code is executed exactly before and after a profile region is entered and exited, respectively. This means that the differential profile metrics can be accurately attributed to the region enclosing the entry and exit instrumentation code. Instrumentation based profiling is more useful in the cases when it is required to profile a number of short-running and infrequently executed methods. In the case when profiling is required for the whole application, other techniques such as *sampling* — which generally has less overhead than instrumentation — might result in a more acceptable overhead.

In general, instrumentation can be performed in the following ways:

Manual Instrumentation - Manual instrumentation is done by the programmer, e.g., by adding user instructions to explicitly calculate runtime information.

Compiler Assisted Instrumentation - In compiler assisted instrumentation, compiler adds instrumentation code. User just has to run the compiler with special options provided for this purpose, e.g., running *gcc* with *-pg* option for instrumentation.

Binary Instrumentation In this approach, an instrumentation tool adds instrumentation code to a compiled binary. Binary instrumentation can be done statically or dynamically. Static binary instrumentation was pioneered by *ATOM* [186]. Static

approach is easy to use, however, with static instrumentation, there is a possibility to mix code and data in an executable, and these instrumentation tools may not have enough information to distinguish between the two. In order to avoid this problem, researchers suggest dynamic approach, in which the instrumentation is done dynamically, thus these tools can rely on the execution to discover all the code at runtime. There are two classes of dynamic binary instrumentation. These are described as follows.

Dynamic Runtime Instrumentation - This approach works by dynamically compiling the binary. It can insert the instrumentation code or calls anywhere in the binary. In this approach, the program code is instrumented directly before the execution. The program execution is fully supervised and controlled by the tool, e.g., *Pin* [134] and *Valgrind* [151].

Dynamic Runtime Injection - This kind of instrumentation works by dynamically replacing instructions in the original program with program hooks that branch to the instrumentation code. This is more light weight than runtime instrumentation. The program code is modified at runtime to have jumps to helper functions, e.g., *Dyninst* [45, 98].

2.5.2 Sampling Based Profiling

In the sampling based profiling, also known as *stochastic profiling*, the profiler probes the Instruction Pointer (IP) of the target program at regular time intervals, using OS interrupts. Sampling profiles are typically less accurate and specific to particular system architecture, but allow the target program to run at nearly full speed. Sampling based approaches avoid the high runtime overhead of instrumentation. In these approaches, a timer is set up to generate an interrupt at the required time interval. This time interval is associated with different program constructs, such as function calls or loops, depending on the location determined by the IP. Sampling based profiling is less accurate than instrumentation based profiling, since it approximates the association of the sampling interval to a program construct enclosing the IP value at the instant when the sample is taken.

The resulting data obtained with this kind of profilers is not exact, but a statistical approximation. The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error is the square root of n sampling periods. The most commonly used statistical profilers are *GNU gprof* [89] and *Intel VTune* [215].

2.5.3 Simulation Based Profiling

Simulation is another method for collecting profiling information about the behavior of a program. In simulation based profiling, an application is run on an instruction set simulator, such as the *SimpleScalar* simulator [46], with the simulator keeping track of detailed profiling information. Despite being accurate, simulation based profiling is extremely slow, especially when simulating a System on Chip (SoC), where simulating an application for several hours may cover only a few seconds of the real time, hence lim-

***Table 2.2:** Instrumentation vs. sampling based profiling.*

Profiling Type	Advantage	Disadvantage
Sampling Based Profiling	-It works with unmodified executable and requires no special hardware support. -As compared to instrumentation, sampling is easy to use, as it is an external tool and no user intervention is required for the program code. -It is not affected by threading.	-It needs enough samples to be accurate. -It is not useful for the applications that execute quickly (data sampling is difficult). -It requires an external tool to take samples.
Instrumentation Based Profiling	-It delivers highly precise and accurate results. -It does not need a separate sampler. -It does not require debugging information to interpret the data.	-With instrumentation, an application must be recompiled for profiling. -Results obtained may be skewed by the insertion of profiling hooks.

iting how much of an application's execution can be realistically profiled. Furthermore, setting up such simulations can be difficult, if not impossible for embedded systems, due to the complex external environments that may also need to be modeled. In summary, although simulation is flexible, it is very slow compared to other profiling techniques (instrumentation or sampling based). Moreover, it is complicated and not readily applicable in all environments.

2.5.4 Instrumentation vs. Sampling Based Profiling

The instrumentation and sampling based profiling approaches each have their own advantages and disadvantages. The biggest advantage of instrumentation based profiling is that it can provide highly precise timing data when the necessary hardware support is available. However, the major drawback is that the insertion of the profiling code may significantly alter the performance characteristics of the application, and the extracted profile data — while very precise — may not accurately reflect the true application performance. In other words, the application with the profiling hooks is not the same as the original application.

The biggest advantage of sampling based profiling is that it requires no modification of the application, only line number debugging information is enough for the profiler to correlate the recorded IP locations to the source code of the application. Another advantage of sampling is that it is system-wide, i.e. one can see the activity in the OS code, including drivers, and not just the user application. However, in order to provide an accurate picture of the application performance, it needs to have enough samples, thus it is unusable for applications that execute quickly. Table 2.2 summarizes the advantages and the disadvantages of instrumentation and sampling based profiling. Furthermore, the combination of these two approaches is also reported [37].

2.6 Hardware Profiling

In the hardware profiling, measurements are taken with hardware. Hardware performance counter based profilers, such as *Intel VTune*, are used to determine program regions that incur a large number of cache misses and branch mispredictions. Many modern processors include hardware counters for profiling. There are specific hardware components attached to the motherboard that take timing measurements without changing how the program executes. A software can then sample these counters to find information, such as delay in cycles or number of cache misses. The hardware counters are used to record counts and then generate an interrupt to the software. The software does the random sampling of the interrupts and records information for later use. Processors, such as Lucent 16K, ARM, Philips Trimedia, are provided with typical hardware support, called Hardware Debug System (HDS), which is used for program debugging. This special hardware is connected to the program address bus, which allows monitoring the activity of the processor. A profiler can use this hardware to collect a number of interesting program metrics, such as cycle counts, dynamic instruction counts, execution counts, jump and call counts, by placing the address of the data collection routine in the interrupt routine address register. Some profiling techniques that rely on hardware features of the processors are presented in [146, 147, 149, 166].

Figure 2.1 depicts the different classes of the application profiling. At the highest level, program analysis is either static (not via execution, usually based on the source code or some Intermediate Representation (IR)) or dynamic (requires to execute the program). Static analysis is performed to estimate parameters related to either software or hardware. On the other hand, profiling measurements collected in the dynamic analysis are either via hardware facilities integrated in computing systems or they are totally based on software. Software based profiling is further divided into three categories based on the approach utilized to inspect the execution of the program. It is either taking sample measurements in predefined moments in time — thus, not completely covering the execution of the program — or the measurements are taken while the application is executed in a simulator, or they are taken by inserting extra code into the application, called instrumentation. Instrumentation can be performed manual in the source code by the user, or automatic by the compiler, or on the binary code of the application. In static binary instrumentation, application has to be recompiled, however, in dynamic binary instrumentation, the extra code is inserted in the program during the execution. There are two approaches to achieve this goal: 1) the instructions are instrumented directly before the execution, 2) the instructions are replaced in the original program with hooks to jump into instrumentation routines.

2.7 Data Structures for Profiling

Different profiling and tracing techniques use different types of data structures for capturing system information. Several data structures might capture different aspects of a system and they might have different levels of efficiency. Selecting an appropriate data structure significantly affects the performance of the application as well as the application analyzer. In the following, we briefly describe the common data structures used in

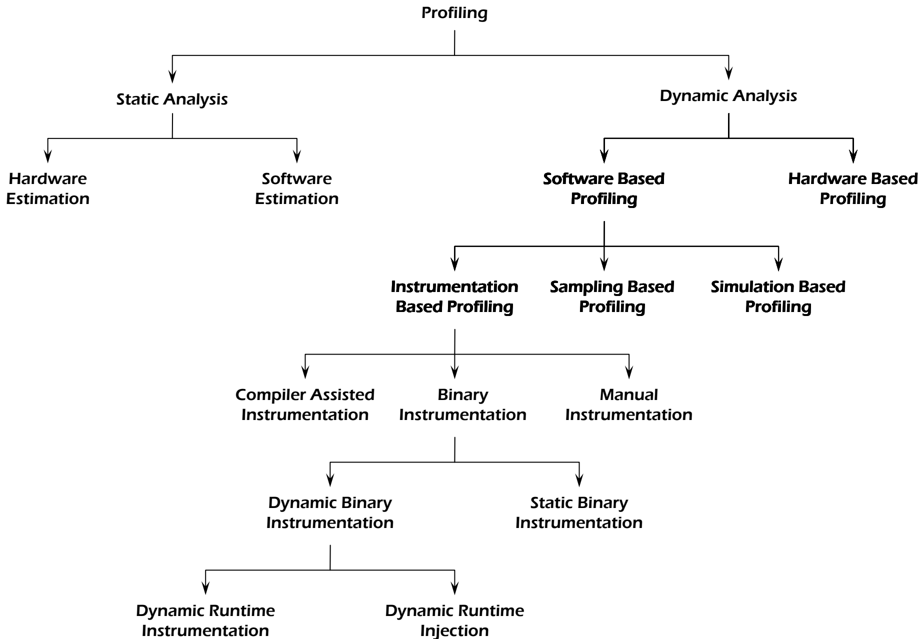


Figure 2.1: *An outline of the different classes of application profiling. Generally, profilers are either static (do not execute the program) or dynamic (based on the execution of the program). Dynamic program analysis is performed by utilizing hardware facilities or is completely software-based. On the basis of the used approach to take measurements, software based profiling is further classified into sampling, simulation, or instrumentation. Furthermore, instrumentation is either performed manual by the user, or automatic by the compiler, or by a dedicated instrumentation framework (toolkit). An instrumentation framework may statically instrument the binary code of the application before the actual execution or it may perform this task dynamically during the execution. In dynamic binary instrumentation, which is the most efficient one, the instructions are either instrumented right before the execution or they are replaced by other instructions to jump into instrumentation routines.*

the analysis of a program.

Directed Acyclic Graph (DAG)

A DAG is a directed graph with no directed cycles. That is, it is formed by a collection of vertices and directed edges, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again. DAGs have many important applications in the construction of parse trees for compilers and their optimizations. DAG may be considered as a compact and lossless representation of the dynamic control flow of a program.

Control Flow Graph (CFG)

A CFG is a graph representation of all the paths that might be traversed through a program during its execution. Each node in the graph represents a basic block, i.e. a straight line piece of code without *jumps* or *jump targets*. *Jump targets* start a block and *jumps* end a block. Directed edges are used to represent jumps in the control flow. In most representations, there are two specially designated blocks: the *entry* block and the *exit* block. The control enters into the flow graph through *entry* block and leaves through *exit* block. CFG is the most widely-used data structure for many compiler optimizations and analysis tools.

Data Flow Graph (DFG)

A DFG is a graph where nodes represent operations and edges represent data paths. It represents data dependencies between operations. DFGs are used as an intermediate representation between the algorithmic programming language and the circuit-level Field Programmable Gate Array (FPGA) configuration. DFG makes data dependencies explicit and it is a convenient representation for many compiler optimizations. DFGs are transferred to VHSIC Hardware Description Language (VHDL) circuits by mapping edges onto wires and nodes onto VHDL components.

Control and Data Flow Graph (CDFG)

A CDFG is a DAG in which a node can be either an operation node or a control node (representing a branch, a loop, etc.). The directed edges in a CDFG represent the transfer of values or control from one node to another. An edge can be conditional, representing a condition while implementing if/case statements or loop constructs. In general, the nodes in CDFG can be classified as one of the following types:

- *Operational* nodes are responsible for arithmetic, logic, or relational operations.
- *Call* nodes denote calls to subprogram modules.
- *Control* nodes are responsible for operations, such as conditionals and loop constructs.
- *Storage* nodes represent assignment operations associated with variables.

Most compilers have an IR that can easily be transformed into a CDFG. Data flow analysis techniques, such as reaching definitions, live variables, and constant propagation, can be applied directly to CDFGs.

Hierarchical Control- and Data-Flow Graph (HCDFG)

A HCDFG is an extension of the CDFG. This representation enables multi-level granularity specification and characterization. It captures control constructs as nodes in a DFG. A loop then becomes a node in a DFG, while loop body is represented as a DFG on a lower level.

Hierarchical Task Graph (HTG)

A HTG is a directed graph that consists of three types of nodes. *Single* nodes represent nodes that have no sub-nodes, which are used to encapsulate basic blocks. *Compound* nodes are hierarchical in nature, i.e. they contain other HTG nodes. They are used to represent structures, such as *if-then-else* blocks, *switch-case* blocks, or a series of HTGs. *Loop* nodes are used to represent various types of loops (*for*, *do-while*, *while-do*). A *loop* node consists of a loop head and a loop tail that are *single* nodes and a loop body that is a *compound* node. CFGs are mostly useful for traversing the design during scheduling, while HTGs are used for moving operations hierarchically across large pieces of code, without visiting each intermediate node.

State Transition Graph (STG)

A STG is a directed graph whose nodes represent the states and the edges represent the transitions between the states. One or more actions (outputs) may be associated with each transition. The diagram is mainly used to represent a Finite State Machine (FSM). As such, they require that the system be composed of a finite number of states; sometimes, this is indeed the case, while at other times, this is a reasonable abstraction.

Abstract Syntax Tree (AST)

An AST is a finite and labeled directed tree, where the internal nodes are labeled by operators and leaf nodes represent the operands of the operators. It is often used as an internal representation of a program in compilers or interpreters, while the program is being optimized. Furthermore, the program code is generated from the AST.

2.8 Profiling Approaches

In the simplest form, a profiler places a counter at every function, basic block, or each line of code. Every time a corresponding execution of the designated element takes place, the counter value is incremented. Not all the profiling approaches adopt only this simple technique to monitor the behaviour of an application, however, it reflects the basis of quite some techniques used in different profilers. In the following, we briefly describe some of the common approaches used for profiling and tracing.

2.8.1 Basic Block Profiling

Basic block profiling is a simple approach that places a counter at every basic block to count the number of times each basic block executes. Another variation of this approach is to measure the time taken to execute each basic block by placing time measurement counters at the start and the end of a basic block. However, there are two drawbacks of such approach: 1) too many counters are used, 2) the total number of increments during an execution is larger than necessary. In order to avoid these overheads, various algorithms have been proposed.

2.8.2 Control Flow Profiling

Most profile-guided optimizations depend on CFG profiles, which can be gathered via code instrumentation (see Section 2.5.1) or statistical sampling (see Section 2.5.2). There are three basic types of control flow profiling: edge profiling, vertex profiling, and path profiling.

Edge Profiling

In edge profiling, the profiler counts the number of executions of each edge in a CFG. Edge profiling determines the transition frequency between basic blocks by incrementing a counter each time a branch instruction is executed. It determines the most frequently executed paths through an application by following the highest frequency edges.

Vertex Profiling

Vertex profiling counts the number of executions of each vertex at the basic block level in a CFG. Vertex profiling counts how many times each basic block is executed during a run of the application. In edge profiling with edge counters, the profiler places a counter on the outgoing edges of each predicate vertex, while in the vertex profiling with vertex counter, the counter is placed on each predicate vertex. Edge profiling demonstrates a clear advantage over vertex profiling, as it provides more accurate information about the behavior of branches, which is extremely important for many optimizations [22].

An edge profile can determine a vertex profile, as a vertex counter can always be replaced by counters on all its outgoing edges to an equal cost (kirchoff's law), hence, there is no need to consider vertex counters. An optimal solution to edge profiling with edge count never has lower cost than an optimal solution to vertex profiling with edge count. The profile of a vertex is either the sum of the profile of all incoming or all outgoing nodes. If all unknown edges connected to a node lead either to or from the node, the profile can be computed by using the nodes in the opposite direction. Various algorithms to efficiently collect profiling information by using these concept are discussed in [22].

Path Profiling

In path profiling, the number of times each path is executed in a CFG is calculated. The performance of large complex systems can be improved by identifying heavily executed paths and *streamlining* them into fast paths. Basic block and edge profiles are inexpensive and widely available, but they do not always correctly predict frequencies of overlapping paths. Path profiling counts how many times each acyclic path in a routine executes. Knowing the edge frequency is not sufficient to find the efficient profile information in the cases where there are possibilities of two or more common paths. Several techniques to determine high frequency paths are discussed in [23]. Path profiling uses a spanning tree to determine a minimal, low-cost set of edges to instrument. Edge profiles are generally thought to be easier and cheaper to collect than path profiles. On the other hand, there are several evidences that suggest that path profiles are superior to

edge profiles in practice. In [24], the authors compare edge profiling and path profiling, and determine when edge profiling and path profiling are good predictors of hot paths and when they are poor predictors.

Several other variations of path profiling are available in literature, e.g., *targeted path profiling* [106] and *practical path profiling* [38]. Targeted path profiling collects profiling information by avoiding counting paths whose frequencies can be derived unambiguously from edge profiles, thus reducing the number of possible potential paths with the elimination of cold path counts. For this purpose, a criterion is defined for cold path that removes cold edges being a part of the potential path if the ratio of that edge's execution frequency to its source block is below some provided threshold. Practical path profiling is the continuation of the targeted path profiling, which counts the paths in a similar way, with more restrictions on cold path elimination. Along with the existing criterion for cold edges, an edge is considered to be cold and will be removed from the potential path if its frequency, as a percentage of the total program flow, falls below a threshold. There is another variation of path profiling that is called *whole program path* [124]. The approach records a complete control flow of a program. At first, it produces a trace of the acyclic paths executed by a program, and then transforms them to a compact form by finding their inherent regularity (i.e. repeated code) in order to identify heavily executed hot paths.

2.8.3 Value Profiling

If it is known that certain values occur very frequently at certain program points, it may be possible to take advantage of this information to improve the performance of the program. This kind of information is generally given by a value profiler, which is a (partial) probability distribution on the values taken on by a variable when control reaches the program point under consideration at runtime. The invariant behavior is identified by a profiler, which can then be used to automatically guide compiler optimizations and dynamic code generation. Value profiling concentrates on profiling at the instruction level, finding the invariance of the written register values for instructions. A value profiler collects two types of information: 1) the invariance of an instruction, 2) the top results for an instruction or a popular range of values. Value profiling is discussed in [47, 218].

2.8.4 Variational Path Profiling

Variational path profiling [165] finds the acyclic control flow paths that vary the most in execution. For this purpose, the profiler records the execution time that it takes to execute frequent acyclic control flow paths using hardware counters. The hardware cycle counter is recorded at the start of the path's execution and again at the end of its execution. Variational path profiling takes these timing measurements and finds a program path that can have significant variation in its execution time across different dynamic traversals in the same program run. This value represents the potential speedup one could achieve if these variations are optimized.

2.9 Summary

In this chapter, we described the concept of program profiling in order to inspect the behaviour of an application. It is mainly used to find the performance bottlenecks of the application and optimization opportunities in addition to ensuring the correct execution of the application. We presented a classification of different profiling techniques and described each technique briefly. Furthermore, we gave an overview of different data structures used in application profiling and tracing techniques. In the end, we described various approaches used in profiling an application.

CHAPTER 3

Dynamic Profiling Framework

"... I could have made money this way [joining the proprietary software world], and perhaps amused myself writing code. But I knew that at the end of my career, I would look back on years of building walls to divide people, and feel I had spent my life making the world a worse place." †

— Richard M. Stallman

This chapter presents an overview of the proposed profiling framework in the context of the Delft Workbench platform. As the main focus of this thesis, the dynamic part of the profiling framework, comprising the QUAD toolset[1]*, is detailed. In particular, we describe the design and implementation of the efficient Memory Access Tracing (MAT) module, which serves at the core of our dynamic memory profiling toolset. The usability of QUAD is demonstrated through examples from real-world applications.*

MULTICORE architectures, especially when containing heterogeneous PEs, pose specific challenges regarding their programmability. Programming such platforms implies, among other issues, determining what parts of the application should be mapped on what PEs. Various criteria can drive this mapping, such as the nature of the computation or the number of cycles required by individual tasks. However, in multicore platforms, *data communication* is often the primary bottleneck in achieving the anticipated speedups. This is especially true for legacy applications, which have to be ported to such platforms. Furthermore, in the case of reconfigurable architectures, the application development process involves building and synthesizing hardware blocks, which is quite time-consuming. As a consequence, there is a need for fast and early predictions of the hardware costs of different parts of an application.

† Quoted from ***Open Sources : Voices from the Open Source Revolution***, 1st Ed., O'Reilly , Jan. 1999, also cited in ***Free Software, Free Society***, Selected Essays of Richard M. Stallman, 2nd Ed., Free Software Foundation, Inc, ch. 2, p. 9, 2010.

[1] QUAD is open source and available on *sourceforge* at: http://sourceforge.net/projects/quadtoolset/

In order to facilitate the complex task of mapping an application onto these systems, there is a clear need for tools that investigate the use of critical resources, such as memory and hardware area. For this purpose, we have developed the Q^2 *profiling framework*. It consists of two main parts: a) a runtime memory access profiling toolset, which provides detailed information on the data communication that occurs inside an application, and b) a statistical modeling part, which makes hardware area predictions early in the design phase based on software metrics. The ultimate goal of the Q^2 profiling framework is to efficiently partition an application into hardware and software parts (see Section 6.2). The extracted profiling information is utilized to guide developers to reduce the data communication between the hardware and the software components so as to maximize the potential speedup, while satisfying resource constraints.

The remainder of this chapter is organized as follows. In Section 3.1, we describe the context of our research work. This includes *Molen* as our heterogeneous reconfigurable architecture, and *Delft Workbench* which is a tool platform to develop applications for such an architecture. Section 3.2 talks about the profiling framework in the *Delft Workbench*, which is comprised of *static* and *dynamic* parts. Subsequently, we further elaborate on the dynamic part of the profiling framework, which is the main focus of this work. In particular, we detail the development of the *QUAD*-core tool in Section 3.3, the Memory Access Intensity Profiler (*MAIP*) in Section 3.4, and the *xQUAD* extension in Section 3.5. How to assess the criticality of candidate functions based on their memory access intensity, is discussed in Section 3.6 using a real case study. Finally, Section 3.7 summarizes this chapter.

3.1 Project Context

Prior to describing the developed profiling framework in more details, in this section, we briefly describe some essential background and the context of our framework. First, the Molen Abstraction Layer (MAL) is presented, which has an important role in practical utilization of the profiling framework (see Section 6.3 and Section 6.4). The abstraction is critical for programming the Molen heterogeneous platform. Subsequently, we discuss the *Delft Workbench*, which is a semi-automatic toolchain platform targeting heterogeneous computing systems containing reconfigurable components. The work in this thesis is carried out as an integral part of this toolchain platform.

3.1.1 Molen Abstraction Layer

The MAL is an abstraction for programming heterogeneous platforms. The MAL ensures sequential consistency, which means that the execution of the program will produce the same results as when the program would have been executed sequentially. In order to provide sequential consistency, the MAL adopts the shared memory paradigm. It provides a unified presentation of the memory, removing the need to manually write complex memory management code. Furthermore, the MAL takes care of the configuration, placement, and parallel execution of accelerated program parts. The result is that application developers can focus on mapping compute-intensive parts to different PEs using familiar program annotations and APIs, such as OpenMP [158] or OpenCL [157].

The MAL consists of a programming paradigm and a machine organization, which will be discussed in the following sections.

Molen Programming Paradigm

The Molen programming paradigm [211] was devised to address several problems in the domain of Reconfigurable Computing. This domain allows for many different accelerators to be incorporated in a computing system through reconfiguration, greatly increasing the need for new instruction opcodes. In addition, the flexibility of reconfigurable architectures allow for changing numbers of input and output parameters, putting additional pressure on the instruction encoding. The Molen programming paradigm addresses these problems by introducing the following one-time Polymorphic Instruction Set Architecture (πISA) extension that can support any number of processor extensions with variable numbers of arguments:

- *SET*. In order to configure the different coprocessors in the Molen Machine Organization, this instruction loads a configuration microcode (ρμ-code), which performs the actual configuration. In this way, different types of reconfigurable units can be configured without the need to change the Molen Machine Organization. This ρμ-code is loaded into the ρμ-unit and the reconfiguration is initiated;
- *EXECUTE*. After the reconfiguration is finished, the loaded configuration can be executed using this instruction. In order to support complex accelerators, this instruction loads execution microcode at a certain address and executes it on the specified Custom Computing Unit (CCU) or PE;
- *MOVTX / MOVFX*. Data can be exchanged among PEs through the main memory, which can be rather slow. Instead these instructions allow the exchange of data among PEs through the exchange registers;
- *BREAK*. As the MAL supports parallel execution of different CCUs, it is important to introduce synchronization points in order to ensure sequential consistency. This instruction denotes such a synchronization point.

Molen Machine Organization

The Molen Machine Organization [212], depicted in Figure 3.1 is an architectural template that implements the πISA proposed in the Molen Programming Paradigm. It follows the processor-coprocessor template, which means that the main control of an application is located in the *Core Processor*, while the other PEs operate as slaves controlled by the Core Processor. This organization can be imposed on any heterogeneous system in order to address the system using the MAL. The overall operation of the Molen Machine Organization is as follows:

1. *Arbiter*. First, instructions are fetched from the main memory and partially decoded by the *arbiter*. The arbiter then decides where to redirect each particular instruction. Regular instructions are redirected to the *Core Processor*. The *SET* and

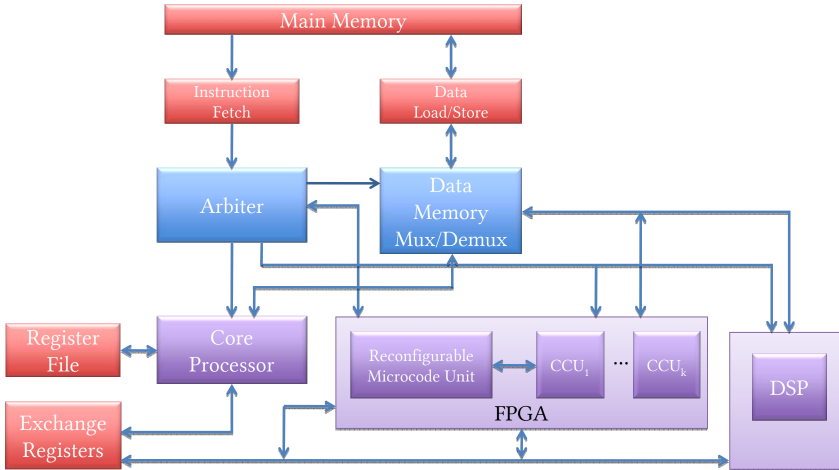


Figure 3.1: *An overview of the Molen Machine Organization.*

EXECUTE instructions are redirected to the target PEs or the ρμ-unit. The *MOVTX* and *MOVFX* instructions are either processed directly or translated to equivalent instructions in the *Core Processor*. Finally, the *BREAK* is processed by the *Arbiter* itself.

2. *Core Processor*. This master General-Purpose Processor (GPP) executes all the non-accelerated code and runs any available OS, interrupts, and management code. πISA instructions are translated to NOPs.

3. *Reconfigurable Processor (RP)*. The RP consists of several CCUs and a *ρμ-unit*. The CCUs consist of reconfigurable hardware and memory, and can contain configuration bit-streams for an accelerator. The *ρμ-unit* processes the ρμ-code and the execution microcode that configures and executes the hardware on the CCUs.

4. *Exchange Registers*. These registers are used for passing the function arguments and return values for the accelerated application code.

5. *DSP*. Apart from an RP, the Molen Machine Organization supports multiple other PE components. For example, one or more DSP processors could be included.

6. *Data Load/Store* and *Memory (DE)MUX*. The transfer of data to and from the main memory is handled by the load/store unit. The parallel access to the memory by all the PEs is handled by the memory multiplexer/demultiplexer.

3.1.2 Delft Workbench

The profiling framework presented in this thesis is within the context of the *Delft Workbench* (*DWB*), a semi-automatic tool platform for integrated hardware/software co-design targeting heterogeneous computing systems containing reconfigurable components [33].

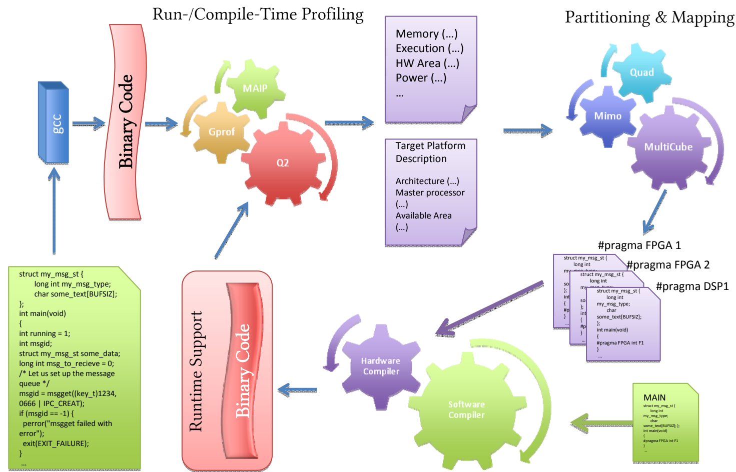


Figure 3.2: *An overview of the Delft Workbench toolchain.*

The project provides application developers with a comprehensive set of tools that support the design and implementation of such heterogeneous systems. The Delft Workbench has several research objectives from every level of the design process. Each objective targets a specific process in the design flow, as depicted in Figure 3.2. In the following, we describe these objectives within their context:

1. ***Profiling and Cost Modeling***
   In order to identify which parts of an application should be moved to hardware, important aspects of these parts need to be characterized, such as their performance, resource requirements, or power needs. This stage aims to prune the design space by identifying which parts of an application are of interest. Furthermore, the information gathered in this stage drives the subsequent partitioning, mapping, and optimization stages.

2. ***HW/SW Partitioning and Mapping***
   Given the heterogeneous nature of the target platform, it is important to map the parts of an application to the various PEs that are available. In case there is no specific architecture defined yet, HW/SW partitioning can be used to map application parts to the general domains of hardware and software. Later, in the design process, when an architecture has been defined, more specific application mapping can be performed.

3. ***Graph Transformations and Optimizations***
   In order to make code more suitable or efficient for hardware implementation, the *DWB* employs code transformations and optimizations, like loop transformations and partial evaluation. These transformations can be general or tailored for the hardware domain or specific PEs.

4. ***Retargetable Compilation***
   The *DWB* provides a retargetable compiler, making it possible to combine configuration bit-streams, GPP code, DSP code, and other binaries. Furthermore, the

compiler schedules the configuration and execution of the different components without compromising sequential consistency.

5. ***Hardware Generation***
   In order to alleviate the effort in translating C-code to Hardware Description Language (HDL), it is important to either automatically generate HDL or select appropriate Intellectual Property (IP)-cores from a library. The *DWB* Automated Reconfigurable VHDL generator (DWARV), a C-to-VHDL compiler, was developed to address this problem within the *DWB* context.

6. ***Runtime Support***
   Finally, the *DWB* aims to provide a runtime support system that enables OSs to support reconfiguration in a multi-tasking environment and applications to adapt to dynamic behavior and real-time changes.

3.2 Q² Profiling Framework

As briefly mentioned in the *Delft Workbench* overview (Section 3.1.2), ***profiling*** consists of characterizing an application with respect to criteria such as execution time, data communication, or resource consumption with the purpose to drive subsequent partitioning and mapping onto a heterogeneous platform. Based on this characterization, the potential candidates are identified for mapping, while the infeasible parts are ruled out. The goal is thus to end up with certain parts that will be accelerated as CCUs and the remaining parts of the application will be executed on a regular GPP. A *part* of an application can be a whole function but it can also be any group of instructions or code segments that are scattered throughout the application. The partitioning itself will be determined in view of predefined objectives, such as increased performance, reduced power consumption or smaller footprint.

Even though the primary objective dictates the ultimate decision criterion, more aspects can come into play as well. The code that will be accelerated must not only fit on the available area of the reconfigurable chip, the cycle time should not be affected too much. This cycle time can go down as the code segment executed on the reconfigurable device grows in size, resulting in an overall slower execution. So one of the trade offs the designer has to make is to choose between larger accelerated code segments and higher performance improvement but then also more area and longer cycle time. Furthermore, data bandwidth usage, which is mainly affected by the amount of data transfer between the hardware-software boundary, should adhere to the physical limitations of the memory devices, both for on- and off-chip memories. These different trade offs that have to be taken into account, combined with the large number of possible code segments, make the design space very large.

The primary concern of Q² is to provide essential profiling information in order to drive efficient application mapping onto heterogeneous reconfigurable systems. As shown in Figure 3.3, Q² has two main parts: *static* and *dynamic*[2]. The static part provides estimations of reconfigurable resource consumption for each part of the application, while

[2] Q² is coined to denote the union of the two acronyms, *QUAD+Quipu*, each referring to one of the parts of the profiling framework.

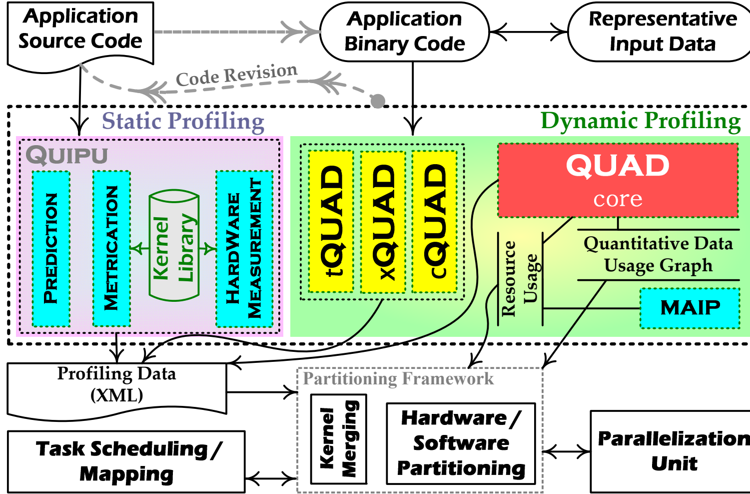


Figure 3.3: *An overview of the Q² profiling framework in the Delft Workbench. The static profiling part extracts code characteristics from the application source code. These characteristics are used by linear models to make fast and early predictions of reconfigurable implementation details. The dynamic profiling part extracts memory access related information as well as data communication between functions by examining the runtime behavior of the application, thus it is not as fast as the static part.*

the dynamic part is concerned with the data communication that occurs inside an application. The main goal is to partition the application into hardware and software, in order to maximize the potential speedup, while adhering to the resource constraints. For this purpose, we characterize an application by determining the computational hotspots, the communication bottlenecks, and the resource-intensive parts within an application (refer to Section 6.2). In the following sections, we describe the static and dynamic parts of the profiling framework in more details.

3.2.1 *Quipu* Modeling Approach

In order to provide resource estimates for various toolchains and platforms, the static profiling part of Q² contains a generic approach for building prediction models, called *Quipu* [142]. *Quipu* is a high-level quantitative prediction modeling approach for early Design Space Exploration (DSE). *Quipu* models are able to predict the important hardware aspects of kernels to be mapped to reconfigurable components. They take a HLL description (C code) as input and estimate area, interconnect, static power, clock period, and other FPGA-related measures for a particular combination of a platform and a toolchain. *Quipu* generates models that accurately capture the relation between hardware and software metrics, based on linear regression, neural networks, and other statistical techniques. The *Quipu* modeling approach is not restricted to any platform or toolchain and appropriate *Quipu* models can be generated in different contexts. By using linear prediction models based on Software Complexity Metrics (SCMs), the time required by *Quipu* prediction models to determine estimates becomes several orders of magnitude

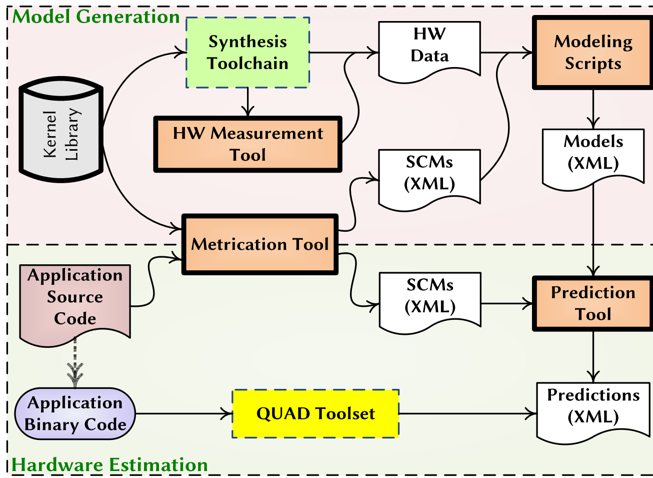


Figure 3.4: *An overview of the Quipu modeling approach. The tools that are signified with thick borders are part of Quipu, and the boxes with dashed borders indicate the accompanying tools.*

smaller than the time-consuming process of hardware synthesis required to obtain the final results. Consequently, developers can quickly evaluate the costs of different parts of their applications in the final design, or select the right PEs for these parts. Furthermore, resource estimates have a crucial role in optimizations such as loop-unrolling, code parallelization, or recursive variable expansion, due to the restrictions on resource availability.

Quipu modeling approach is based on statistical analysis, thus, to generate realistic and accurate prediction models, it is critical to use an independent dataset of measurements. Modeling hardware from C code necessitates *Quipu* to quantify the characteristic aspects of software descriptions. For this purpose, SCMs are utilized to characterize software descriptions. SCMs are indicators of different aspects of the source code under consideration. Currently, *Quipu* uses a set of 92 different SCMs, most of which are simple counts of different operations available in ANSI-C. In addition, *Quipu* includes several metrics that are related to *Software Measurement*, such as the Cyclomatic complexity, and more complex data flow metrics.

Figure 3.4 presents an overview of the *Quipu* modeling approach, which is divided into the following parts:

- ***Model Generation*** is mainly utilized when *Quipu* models need to be generated for a certain toolchain or platform. As an example, when a new version of the synthesis toolchain is released. Although the process can be time-consuming, it is only executed once for a particular combination of a tool and a platform.

- ***Hardware Estimation*** is used to provide fast and early estimates during DSE. As such, it is utilized far more often compared to *Model Generation*.

As shown in Figure 3.4, *Quipu* consists of a set of tools and a kernel library, which are described in the following:

Kernel Library - In order to generate sufficient data points for the modeling process, *Quipu* gathers SCMs and hardware performance indicators from its extensive kernel library. It is a database of 324 kernels from 66 real applications, which constitute a wide scope of different domains. This library is the main reason why *Quipu* is able to produce generally applicable models. Furthermore, this large number of kernels allows for the generation of domain-specific models as well. *Quipu* contains several scripts that traverse the library. These scripts contain the necessary hooks where target tools can be inserted. In addition, similar scripts are provided for the synthesis of the generated HDL descriptions. Detailed information about the kernels in the library is provided in [142].

Metrication Tool - The SCMs are implemented in the *Quipu Metrication Tool*, which produces an eXtensible Markup Language (XML) file containing SCM measurements for each kernel. This tool is written as an engine in the *CoSy* compiler system [59]. *CoSy* contains a large set of optimizations and is easily extensible by writing engines that can be plugged into the system.

Hardware Measurement Tool - This tool gathers necessary hardware performance indicators from outputs of different synthesis toolchains. For some toolchains, this is as simple as finding the right data in the report files, but for others this is not trivial. As an example, in order to count the number of wires in *Xilinx* designs, *Quipu* provides an Xilinx Design Language (XDL) parsing tool that extracts this and other relevant data from the XDL file.

Modeling Scripts and Prediction Tool - The gathered SCMs and hardware measurements are utilized by a set of modeling scripts that automatically evaluates the different modeling techniques described in regression. The output model XML file can be used in *Hardware Estimation*, where, based on SCM inputs, the *Quipu Prediction* tool provides estimates of any required hardware aspects. All intermediate files in the Hardware Estimation are saved in XML format for easy integration. Additionally, the results of runtime profilers, such as *QUAD*, are also integrated in the output XML file, as depicted in Figure 3.4.

The *Quipu* models can be used in several contexts. In the first place, developers can use the predictions for kernels in order to find problems with potential placement on hardware. If a kernel is predicted to occupy a large amount of slices on a particular platform, the developer might move to split the kernel in several parts, or address the primary cause of the large area (e.g., a large local array). Additionally, a tailored *Quipu* model may drive the optimization pass in a hardware compiler by predicting the hardware size of the contained basic blocks at different unroll factors. In this way, the hardware compiler can automatically choose a beneficial unroll factor. Most importantly, a hardware/software partitioning algorithm can use *Quipu* models to evaluate different partitionings at an early stage of application mapping in heterogeneous environments.

3.2.2 *QUAD* Memory Access Profiling Toolset

The *QUAD* toolset consists of several dynamic profiling tools, which are developed to demonstrate a comprehensive overview of the memory access behavior of an applica-

tion as well as to provide fine-grained detailed memory access related statistics. As depicted in Figure 3.3, ***Quad*-core** is designed and implemented as the primary profiler to extract useful quantitative information about the data dependencies between any pair of communicating functions in an application. Data dependency is estimated in the sense of producer/consumer bindings. More precisely, the *Quad*-core tool reports which function is consuming data produced by another function. It measures the exact amount of data transfer[3] in bytes and the size of required communication buffer based on Unique Memory Address (UnMA).

Quad-core contains a fast and efficient Memory Access Tracing module, which detects and traces all the memory accesses made during the execution of an application. It implements a kind of *shadow* memory mechanism for the whole access space of the application. Considering the fact that MAT structures the shadow memory in such a way to make tracing as fast as possible, the size of the space overhead can be very huge. MAT is also utilized in ***cQuad*** tool to monitor, in comprehensive detail, each and every data transfer event occurring between a pair of communicating functions. This data communication can be viewed as a dedicated virtual channel for transferring data items from the producer side to the consumer end. The Data Communication Channel Pattern Detection (DCCPD) module thoroughly analyzes the extracted raw profile data to compute several critical metrics that can classify and describe the pattern of the communication between the two functions. These metrics include the Interleaving Balance Factor (IBF), spatial and temporal localities, and data communication pattern complexity. In general, the information is extracted to reveal the coupling intensity and pattern regularity between the communicating functions in an application. These metrics can possibly be useful in understanding the behavior of the functions with regard to their data dependencies and requirements, as well as, in mapping and scheduling potential parts of the application onto heterogeneous multicore system.

The ***xQuad*** tool augments the memory access analysis of the application by providing very detailed, fine-grained intra-function information. The extended memory access information is provided based on the application source code data object granularity. In this respect, developers can utilize this information to *fine-tune* the application regarding it memory access references. The main motivation for this extension to the *Quad* toolset is that a coarse view of the memory accesses makes it difficult for application developers to attribute the extracted information to particular user-defined data objects at the source code level. Hence, source code revision/optimization becomes a burdensome task.

tQuad is another component in the *Quad* toolset. It was developed to disclose the temporal information of memory accesses at the runtime. As a result, the memory bandwidth usage of functions are estimated during the execution of the application. The information extracted by *tQuad* can lead to the recognition of the main *execution phases* within an application, which can be used subsequently to identify the related functions in each phase. Most probably, each execution phase corresponds to a specific and well-defined (sub)task based on a high-level algorithm of the application code. Even though the exact phases of the algorithm are clear for the programmers, *tQuad* enables users to

[3] Regarding the concept of *data dependency* in *Quad*, we use, when no confusion arises, the terms *'data communication'* and *'data transfer'* interchangeably, which implies the data that is produced by some function and consumed later by some function.

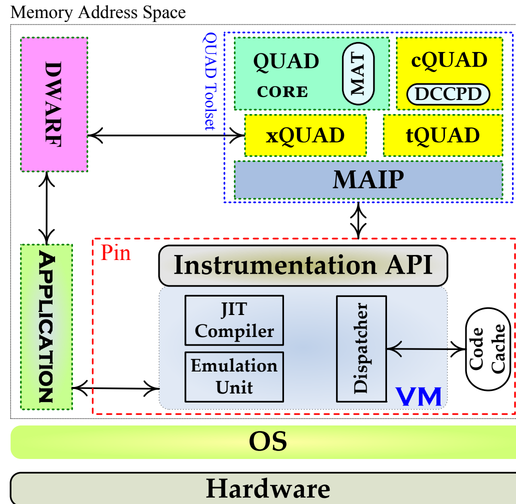


Figure 3.5: *An architectural overview of the dynamic part of the Q² profiling framework. The Q UAD toolset — consisting of QUAD-core, cQUAD, xQUAD, tQUAD, and MAIP — utilizes the Pin DBI framework to instrument the application at runtime. The xQUAD tool uses DWARF to find source-level information about the data objects in the application. Being runtime profilers, all the tools only need application binaries, nevertheless, the debugging information should be available inside the binaries.*

have a more solid view on the steps the application actually takes to perform its task. In addition, the extracted data can serve as valuable hints for the application developers to understand the *(re)usability* scope of the functions, hence, initiating code optimizations. As an example, *tQUAD* may determine that a (general-purpose) function is utilized at different phases of an application. It suggests that the function may be a good candidate for replacement with several specific-purpose customized functions.

Traditionally, a general profiler, such as *GNU gprof* [89], is employed to identify computationally-intensive functions in an application. The *gprof* profiler provides sample-based execution timing estimates in addition to an accurate call graph. On the other hand, ***MAIP***, its counterpart in the Q² profiling framework, provides *accurate* measurements for the contribution percentage of individual functions with respect to the whole execution time of an application. Unlike *gprof*, *MAIP* can distinguish between memory access operations and computational operations as well. This enables *MAIP* to provide several useful detailed statistics about the memory access behaviour of an application.

All the tools in the dynamic part of the Q² profiling framework are implemented using the *Pin* [134] DBI framework. However, *Pin* does not provide any Application Programming Interface (API) function for retrieving data object information in an application. Thus, source-level information about data objects should be extracted directly from binary file(s) of the application using the *DWARF* [67] debugging format. Figure 3.5 illustrates the architectural overview of the *QUAD* toolset along with the *Pin* DBI components. At the highest level in the *Pin* software layer, there is a Virtual Machine (VM), a code cache, and an instrumentation API. The VM consists of a Just-In-Time (JIT) compil-

er, an emulator, and a dispatcher. After *Pin* gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments the application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering (leaving) the VM from (to) the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls that require special handling from the VM.

Since *Pin* does not reside in the kernel of the operating system, it can only capture user-level code. As depicted in Figure 3.5, three binary objects are present when an instrumented application is running: 1) the to-be-profiled application, 2) *Pin*, and 3) the *QUAD* toolset. Pin is the engine that instruments the application. *QUAD* contains instrumentation and analysis routines, and it is linked with a library that allows *QUAD* to communicate with *Pin*.

3.3 Runtime Memory Access Profiling

Profiling, succinctly put, refers to measuring where programs consume resources, most notably, *the processor time* and *memory*. Runtime profilers analyse the behavior of an application during its execution in order to extract useful information related to the performance of the application. The extracted information helps developers identify the types of optimizations that can be carried out on the application and/or the target architecture. General profilers, which analyse an application at the function-level such as *gprof*, provide execution time profiles to identify application hotspots. *However, they do not distinguish between the* ***computation time*** *and* ***memory access time****.* Accordingly, they cannot efficiently be employed to locate potential memory-related bottlenecks, particularly when targeting heterogeneous reconfigurable systems. As an example, in [26, 86], the authors provide target-independent execution performance estimations. However, they lack a thorough analysis of memory access behavior, which is vital for performance optimizations in reconfigurable architectures.

The way an application interacts with the memory has a huge impact on its performance. To express it differently and more elaborately, the memory access behavior of an application, at the most basic level, depends on the intrinsic nature of the application. However, the developer still has considerable flexibility in manipulating the algorithms, data structures, and the structure of the application to modify its memory access behaviour for performance improvements [141]. In the case of mapping an application onto a heterogeneous reconfigurable system, the amount of data that is transferred between different tasks is vital. This is due to the fact that the communication overhead of the application, which is a serious performance obstacle, is mostly influenced by the amount of data that has to be communicated between the HW/SW boundary. It is clear that accurate measurements of this property can be greatly beneficial for developers to efficiently perform the DSE and mapping tasks. Nevertheless, this is not easy to achieve, since it needs a profiling tool that accurately monitors all the memory accesses in the application.

Most of the memory access profilers focus on detecting memory faults, bugs, or leaks and do not provide detailed information about the inherent data dependencies in a pro-

gram's memory access behavior [53, 213]. Notwithstanding, there are some works that perform exhaustive memory access analysis. One early profiling tool developed for understanding memory access patterns of Fortran[4] programs is presented in [44]. The tool instruments a program and produces a flat trace file of all memory accesses which can be visualized later. Similarly, a toolset is presented in [25] to reveal the pattern of memory accesses. It generates a set of histograms for each memory access in a program with respect to the strides of references. In [214], the authors present a quantitative approach to analyse parallelization opportunities in applications with irregular memory access patterns. Applications are classified into three categories with low, medium, and high dependence densities. Embla [77] allows the user to discover the data dependencies in a sequential program, thereby exposing opportunities for parallelization. It performs offline dynamic analysis and records dependencies as they arise during program execution.

What we require in the profiling stage of the *DWB* is more than just discovering the data communication relations between the functions in the application. In order to efficiently improve the application mapping, comprehensive profiling information is required to quantify the data transfers performed during the execution of the application. In particular, we are interested in the amount of data transfers and the size of the actual memory blocks needed for this purpose. The *QUAD*-core tool is able to discover the actual data dependency, which can be different from the conventional data dependency referred to in similar tools. By definition, data dependency indicates a situation in which a program segment[5] refers to the data of a preceding segment. By **actual data dependency**, we want to stress the fact that the data is indeed accessed by the succeeding segment. This situation arises, for example, when a function requires the data that is produced by another function earlier. Put in other words, the common argument passing by the caller function to the callee does not necessarily imply that the data will be indeed used in the called function.

In this section, we describe the *QUAD*-core tool in detail. *It is a runtime memory access profiler that provides a quantitative analysis of the memory access patterns of an application with the primary goal of detecting the actual data dependencies at the function-level.* *QUAD*-core abstracts away from the properties of the data dependency detection of an application on a particular architecture. Furthermore, any restriction of detecting data dependency based on hierarchies of function calls (commonly depicted with call graph) is completely relaxed, as *QUAD*-core exhaustively traces data transfers via memory addresses and does not rely on the control dependencies of tasks to detect potential data dependencies. Mostly, the prior research in data dependency detection are focused on the discovery of parallelization opportunities. However, in our profiling framework, we do not primarily target parallel application development. Even though *QUAD* is general enough and quite efficient to be used for spotting coarse-grained parallelism, it provides a more general-purpose profiling framework. This framework can be utilized in various optimization directions for application development. As an example, estimating the amount of UnMAs provides valuable information for designers to come up with efficient on-chip interconnections to maximize the application performance [167].

Accessing memory locations sequentially or within defined strides can substantially

[4] A general-purpose programming language that is especially suitable for numeric computations. *Fortran* is a blend derived from the IBM Mathematical **For**mula **Tran**slating System.

[5] *segment* in its general sense, which may refer to an instruction, a block, or a whole function.

contribute to the efficiency of applications, particularly in cache-based computing systems. In most cases, it is possible for application developers to *restructure* data or code in order to achieve better memory reference behavior. From a different perspective, *Q UAD* can also be useful in diagnosing coding problems and inefficiencies related to memory, such as referencing beyond the boundaries of allocated memory blocks, detection of unused data, etc. Furthermore, *QUAD* can be easily ported to other platforms as long as a primitive toolkit can be found to provide the basic memory access (read/write) instrumentation capabilities, such as *BIT* [126] which instruments java byte codes.

The main features of the *QUAD*-core profiler are listed in the following:

- It detects the actual data dependency at the function-level with the highest degree of accuracy. The data dependency information provided by *QUAD*-core can be different compared to the conventional data dependency detection by other similar tools.

- An accurate list of memory addresses used in data transfers is extracted by *QUAD*-core tool. This enables us to measure the actual size of required memory blocks for data communication between a pair of functions.

- It does not require the source file(s) of an application nor it requires any modification of the application binaries; it has no compiler dependence other than the availability of debugging information. Furthermore, it abstracts away from the properties of a particular architecture.

The rest of this section is organized as follows. First, in Section 3.3.1, we briefly introduce the instrumentation framework used in the development of our dynamic profiling toolset. Section 3.3.2 presents some highlights about the *QUAD*-core tool implementation. In Section 3.3.3, we describe the Memory Access Tracing module. Next, in Section 3.3.4 and Section 3.3.5, we explain the usage of the *QUAD*-core tool in practice, using two real applications. In each case study, we comment on some observations that are extracted from the profiling information.

3.3.1 Pin Dynamic Binary Instrumentation

Instrumentation is a technique for inserting extra code into an application to observe its behavior. This process can be performed at various stages either in the source code, or at compile time, or at post-link time, or at runtime. Runtime instrumentation means that the code is injected into the application during the execution. The primary advantage of this kind of instrumentation is that, in order to profile an application, only unmodified binary executable would suffice. *QUAD*-core is a Dynamic Binary Analysis (DBA) tool that analyses an application at the machine code level, as it runs. DBA tools can be built from scratch or be implemented using a Dynamic Binary Instrumentation framework. In a DBI tool, instrumentation is performed at runtime on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that *dynamically* generate code.

Quad-core is implemented as a *Pintool*[6]. By using *Pin*, we have the benefit of working transparently with unmodified Linux, Windows and MacOS binaries on Intel ARM, IA32, 64-bit x86, and Itanium architectures. Thanks to the instrumentation transparency, *Pin* preserves the original behavior of the application. Thus, the application uses the same addresses (both instruction and data) and the same values (both register and memory) as it would in an un-instrumented execution. Furthermore, *Pin* does not modify the application stack, as some applications may deliberately reference memory addresses beyond the top of the stack. This transparency, vital for correctness, results in accurate data collection.

Dynamic instrumentation is particularly beneficial for this type of tools. It captures the execution of arbitrary shared libraries in addition to the main program, and it has no dependence on the instrumented application's compiler. Requiring only a binary and being compiler-independent does not imply that the source code is not needed for program revisions. Instead, it provides flexibility for the tool to be language-independent and it can be used with any compiler toolchain that produces a common binary format. Furthermore, it does not require the user to modify the build environment to recompile the application with special profiling flags. *Pin* adopts the dynamic compilation technique that uses a JIT compiler to (re)compile and instrument the application code on the fly. This capability provides the benefits of portability, transparency, and efficiency to the end user. *In summary, the reason we adopted Pin is that it supplies a fast instrumentation framework, which is able to work with unmodified executables in addition to the preservation of the application's original un-instrumented behavior.*

Two types of routines are defined in *Pintools*, namely *instrumentation* routines and *analysis* routines. Instrumentation routines determine where, in the application code, to place calls to analysis routines. Analysis routines are customizable by the user and they are called while the program executes. The arguments to analysis routines can be, for example, the instruction pointer, the effective memory address of the instruction, the memory value, the address of branch instruction, the system calls values, and others. The actual instrumentation is performed by the JIT compiler. *Pin* intercepts the very first instruction of the application and re-compiles the executable generating *basic blocks* code starting at that instruction, and instrumenting the code according to the specified instrumentation type. The generated code sequence is almost identical to the original one, except that it runs under the control of *Pin*. When a branch exits a basic block, *Pin* generates more basic blocks code for the branch target and it continues the execution. The JIT generated code and its instrumentation are saved in a code cache for future execution of the same sequence of instructions to improve performance.

Instrumentation with *Pin* can be done at different levels of granularity. The finest level is instrumentation at the *instruction level*, i.e. instrumenting the application one instruction at a time. It is also possible to instrument code at the *trace level*[7], at the *routine level*, and at the entire *image level*. *Image* instrumentation enables the tool to inspect and instrument an entire image, when it is first loaded. Subsequently, the tool can walk the sections of the image, the routines of a section, and the instructions of a

[6] The tools created using the *Pin* DBI framework are called *Pintools*. *Pin* is proprietary software, but it is available free of charge for non-commercial use from: http://www.pintool.org/.

[7] A *trace* in *Pin* is defined as a straight-line sequence of instructions executed sequentially. *Pin* guarantees that traces only enter at the top, but may have multiple exits.

routine. In general, instrumentation routine can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed.

The execution of an instrumented application usually shows a considerable *slow-down*. This depends on the nature of the instrumented application as well as on the overhead caused by the analysis routines in the tool. It appears that most of the slow-down is caused by the execution of the code, rather than on the fly code compilation, which includes the insertion of the instrumentation code. In *Pin*, some performance improvements are done during the compilation phase of the application. Improvements are performed on register reallocation, inlining, liveness analysis, and instruction scheduling. This results in an instrumented code, which run very fast compared to other DBI frameworks.

3.3.2 *Quad*-core Development

The QUAD-core tool, is the fundamental part of the dynamic profiling framework in Q^2, which is designed to provide useful quantitative memory access information. Its main goal is to accurately measure the actual data dependencies between any pair of communicating functions in an application. Data dependency is extracted in the sense of producer/consumer bindings. More precisely, *Quad*-core reports which function is consuming data produced by another function. The exact amount of the data transfer and the number of UnMAs used in the transfer are extracted. Monitoring and keeping records of all the memory access activities is made possible with an efficient Memory Access Tracing module, which is developed from scratch based on a tree-like data structure. Utilizing the MAT module, a variety of memory access related statistics can be measured, e.g., the ratio of local to global memory accesses in a particular function call.

The interfaces to most DBI frameworks are API calls that allow developers to hook in their instrumentation routines. In *Pin*, the API call to *INS_AddInstrumentationFunction()* allows a user to instrument programs based on a single instruction while the *RTN_Add InstrumentFunction()* provides instrumentation capability at the routine granularity. *Q uad*-core uses these two API routines to set up calls to the instrumentation routines *Instruction()* and *UpdateCurrentFunctionName()*. These two instrumentation routines, in turn, call the two main analysis routines *RecordMemRef()* and *EnterFunc()* which are responsible for updating tracing information of memory accesses and maintaining an internal call graph, respectively. Since *Quad*-core relies on runtime instrumentation and it is compiler-independent, detecting the producers or consumers of the data must be performed in the absence of any kind of control/data flow information. In order words, the detection is only based on the information available during a conventional execution of the application. As a consequence, *Quad*-core implements and maintains its own customized call graph during the execution of the application. This graph also provides the flexibility to implement some of the profiling options available in *Quad*.

The details of the *Quad*-core implementation modules can be found on the project home page[8]. In the following, we only discuss some highlights. Figure 3.6 illustrates an implementation overview of the *Quad*-core tool. The initialization process in the main module includes *Pin* initialization, command line options parsing, internal call graph

[8] http://sourceforge.net/projects/quadtoolset/.

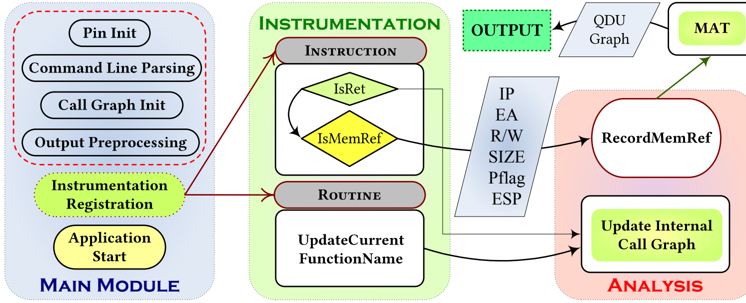


Figure 3.6: *Implementation overview of the QUAD-core tool. The main module contains the preprocessing and instrument registration routines. It also handles the DBI process. The instrumentation and analysis modules contain the memory access instrumentation and analysis routines, respectively. Instrumentation in QUAD-core is performed at two different granularity levels. The QDU graph is the primary output of the tool.*

initialization, and some output XML file preprocessing. The *Instruction()* instrumentation routine sets up the call to *RecordMemRef()* routines every time an instruction that references memory is executed. When *Pin* starts the execution of an application, the JIT compiler calls *Instruction()* to insert new instructions into the code cache. If the instruction references memory (read or write), *QUAD*-core inserts a call to *RecordMemRef()* before the instruction, passing the IP, Effective Address (EA) for the memory operation, a flag indicating whether it is a read or write operation, number of bytes read or written, and a flag showing whether or not the instruction is a prefetch. The analysis routine returns immediately upon detection of a prefetch state for an instruction. *INS_InsertPredicatedCall()* injects the analysis routine and ensures that the analysis routine is invoked only if the memory instruction is predicated true. There is also a separate *RecordMemRef()* analysis routine for the case that we are interested to trace local memory references in the stack region. In this case, the value of the Stack Pointer (SP) is also passed to the analysis routine for further investigation. *Instruction()* also monitors the *ret*[9] instruction to leave a function and upon detection, it calls a different analysis routine that updates the internal call graph, if necessary.

The main objective of *RecordMemRef()* is to identify the function responsible for the current memory reference, and to pass the required information to the MAT module. The instrumentation at the routine granularity in *QUAD*-core is responsible for *pushing* the name of the currently called function onto an internal call stack. The respective *popping* is later performed upon detection of the *ret* instruction. In particular, *QUAD*-core requires to maintain its own call graph because one may not be interested to dive into library routines or routines that are not included in the main binary image file. In these cases, *QUAD*-core assumes the most recent caller routine from the main image as the one responsible for issuing memory references.

The memory access information gathered by the *QUAD*-core tool during the execution of the application is reported in two separate formats. The producer/consumer binding information is output to a text file using standard portable XML format. This

[9] *return.*

makes it easy for third-party applications to import the data for further interpretation and processing. The actual data dependency information is also provided in the form of the QDU graph. The output graph can be easily rendered using any application which can interpret the *DOT* graph description language, such as *Graphviz* graph visualization package [90].

3.3.3 Memory Access Tracing

The main component that enables *Quad*-core to efficiently monitor memory accesses, is the Memory Access Tracing (MAT) module. It is responsible for building and maintaining dynamic data structures in order to trace memory accesses as fast as possible, while adhering to a reasonable memory footprint. The dynamic data structure is based on *trie*[10] [81], which is customized to implement a *shadow memory* for the entire address space of an application. More specifically, the *Quad*-core tool actually *shadows*, in software, every byte of the memory accessed in an application by annotating it with extra information. It is very difficult to create an efficient implementation of such a shadow memory. Nevertheless, it is a very powerful technique with various applications in profiling tools; e.g., detecting critical errors, such as, bad memory accesses, data races, and referencing un-initialized data objects. In a DBA tool which implements shadow memory, an analysis routine is responsible for querying, and if required updating, the shadow memory in reaction to each memory access. The data maintained in the shadow memory is subsequently used to report profiling information to the user. The granularity of the shadowing can vary, but usually every *used* memory byte or word has a shadow memory value, and each shadow memory value may itself be one bit, a few bits, one byte, or even more, depending on the profiler [150, 234].

Implementing an efficient and robust shadow memory is complicated since there is always a major trade off between the tracing time and the space overhead. On one side, shadow memory is inherently expensive, particularly if large amounts of extra data should be maintained. Furthermore, all (or most) read/write operations should be instrumented to keep the shadow memory state up-to-date. This unavoidably increases the total amount of executed code, increases the application's memory footprint, and degrade the locality of its memory accesses. Thus, profiling tools that implement shadow memory typically experience a slowdown factor up to 100× [150]. On the other side, shadow memory should be squeezed into the address space alongside the original memory in a way that does not interfere with it and does not modify the behaviour of the application. This requires considerable flexibility in the shadow memory structure and layout. It also inevitably reduces the amount of available main memory for the application itself.

In MAT, we define *trie* structures with base 16, which is representative of memory addresses in hexadecimal format. Each hexadecimal digit in a 32-bit memory address corresponds to one level in the trie data structure, leaving 8 levels deep in the hierarchy for complete address tracing. The same structure can be generalized to accommodate 64-bit memory addresses. This will create a trie whose depth is 16. The idea is depicted in Figure 3.7. Additionally, the trie data structure is implemented in such a way that it

[10] The term *trie* comes from re**trie**val. Following the etymology, the inventor, Edward Fredkin, pronounces it /ˈtriː/ (tree). However, it is now widely pronounced /ˈtraɪ/ (try) [115].

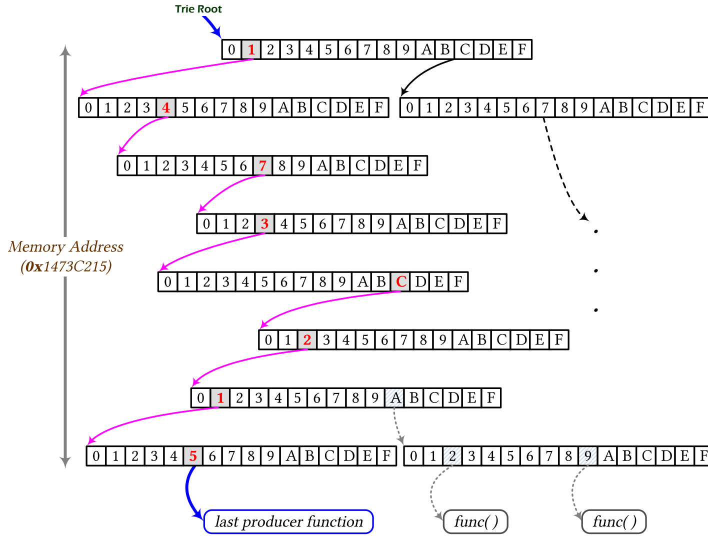


Figure 3.7: *An outline of the dynamic trie data structure of base 16 for tracing 32-bit memory accesses. Each level in trie represents a 4-bit value (a hexadecimal digit) in the memory address. Taking a memory address, the trie crawls 8 levels deep to fully trace the address as quickly as possible w/o a naive array-based method using statically-reserved space. Along the way, required levels are dynamically created on demand.*

grows *dynamically* on demand so as to reduce the memory footprint as much as possible. This means that if a particular memory address is fed to the MAT module for the very first time, the levels required to trace that particular address are created in the trie. Thus, no space is allocated for unused memory addresses. The space saving is substantial because the data structure can become gigantic, and it is very probable to run out of the main memory for its storage if it is not dynamically increased in size.

Listing 3.1 presents the source code of the *RecordMemoryAccess()* routine of the MAT module. Since this routine is a critical component of the tracing mechanism, it is implemented purely in C language and completely from scratch for optimum performance. Any deficiency in the implementation, such as the usage of any template library functions, would result in huge overhead, and hence considerably slows down the instrumentation.

The memory reference recording process is accomplished in two distinct phases. In the first phase, we trace an 8-level trie for a particular memory address. For each memory reference three different arguments are specified: the memory address, the function identifier, and the read/write flag. In the case of *write* access, the corresponding shadow memory address in the trie is labeled with the caller (producer) function identifier. When a *read* access is detected, the function identifier responsible for the most recent write access to that memory location is retrieved. Subsequently, the producer function identifier is passed along with the consumer function identifier to the second phase of tracing, where a data communication binding (record) is created.

```c
#define  trieHeight  8
struct trieNode
{
    struct trieNode * list[16];
} *trieRoot=NULL;

struct AddressSplitter
{
    unsigned int h0:4;    unsigned int h1:4;
    unsigned int h2:4;    unsigned int h3:4;
    unsigned int h4:4;    unsigned int h5:4;
    unsigned int h6:4;    unsigned int h7:4;
};

int RecordMemoryAccess(ADDRINT addy, ADDRINT func,bool writeFlag)
{
    int i,retv,currentLevel=0;
    struct trieNode* currentLP;
    struct AddressSplitter* ASP= (struct AddressSplitter *)&addy;
    unsigned int addressArray[trieHeight];

    addressArray[0]=ASP->h0;    addressArray[1]=ASP->h1;
    addressArray[2]=ASP->h2;    addressArray[3]=ASP->h3;
    addressArray[4]=ASP->h4;    addressArray[5]=ASP->h5;
    addressArray[6]=ASP->h6;    addressArray[7]=ASP->h7;

    if(!trieRoot)  /* create the first level in trie */
         if(!(trieRoot=(struct trieNode*)malloc(sizeof(struct trieNode)))) return 1;
              /* memory allocation failed */
         else  for (i=0;i<16;i++)   trieRoot->list[i]=NULL;

    currentLP=trieRoot;
    while(currentLevel<trieHeight-1)  /* proceed to the last level */
    {
         if(! (currentLP->list[addressArray[currentLevel]]) ) /* create a new level
              on demand */
              if(!(currentLP->list[addressArray[currentLevel]]=(struct trieNode*)
                   malloc(sizeof(struct trieNode))) ) return 1; /* memory allocation
                    failed */
              else  for (i=0;i<16;i++)
                        (currentLP->list[addressArray[currentLevel]])->list[i]=NULL
                             ;
        currentLP=currentLP->list[addressArray[currentLevel]];
        currentLevel++;
    }  /* reached the last level for the address tracing? */

    if(!currentLP->list[addressArray[currentLevel]]) /* create a new data bucket to
         save access info */
         if(!(currentLP->list[addressArray[currentLevel]]=(struct trieNode*)malloc(
              sizeof(ADDRINT)))) return 1; /* memory allocation failed */
         else  /* no write access has been recorded yet. */
                 *((ADDRINT*) (currentLP->list[addressArray[currentLevel]]))=0;

    if (writeFlag)       /* only record the last write access */
         *((ADDRINT*) (currentLP->list[addressArray[currentLevel]]))=func;
    else  /* record the producer -> consumer binding */
        if (RecordCommunicationInQDUGraph(*((ADDRINT*) (currentLP->list[addressArray
             [currentLevel]]) ),func,addy))  return 1; /* memory exhausted */

    return 0; /* successful tracing */
}
```

Listing 3.1: *Memory access tracing implementation in Quad-core. The implementation is based on a dynamic trie data structure to create an efficient shadow memory for each memory address used in the application.*

3.3.4 Identifying Memory-Intensive Kernels

In order to inspect and verify the extracted profiling information by the *QUAD*-core tool, we conducted several experiments using real applications. In this section, we discuss the experimental results of the *x264* [229] video codec. *The goal is to have an initial understanding of the application behavior regarding its data communication, memory usage, and memory requirements. The information provided by QUAD-core can be further used for partitioning and mapping, as well as for providing hints to developers how to optimize and tune the code for a particular platform.* For more detailed and practical usage of the profiling information extracted by the *QUAD* toolset refer to Chapter 6. *x264* is a free library for encoding H.264/AVC video streams. The version used in this work is a modified *x264 r654* encoder tailored for Molen (see Section 3.1.1) taking into account the restrictions in terms of coding rules accepted by the Delft Workbench Automated Reconfigurable VHDL generator (DWARV) hardware compiler [233].

Experimental Setup

All the experiments were executed on an Intel 64-bit Core 2 Quad CPU Q9550 @ 2.83GHz with the main memory of 8GB, running Linux kernel v2.6.18-164.6.1.el5. The *x264* source code was compiled with *gcc* v3.4.6 and with the profiling option enabled. We used *gprof* as an auxiliary tool to interpret execution timing information. The standard command line options used to run the 64-bit compiled version of *x264* was the following:

1. *−no-ssim* - to disable the computing of Structural SIMilarity (SSIM) index which is used as a video quality metric;
2. *rate control -q1* - to indicate almost lossless compression;
3. *−no-asm* - to disable all stream processing optimizations based on CPU capabilities. Using Streaming SIMD Extensions (SSE) technology enables Single Instruction Multiple Data (SIMD) to speed up streams processing. The library provided by *x264* is capable of utilizing this technology and its former MMX[11] one. As a result, the kernels called are somehow different from the normal processing functions. This option has been intentionally disabled, since it may not be applicable to the target architecture.

The 64-bit version of *QUAD*-core was used with the following command line options:

1. *ignore_stack_access* - to ignore all the memory accesses to the stack region. This gives a clear view of the data transferred via non-stack region.
2. *use_monitor_list* - to include only some critically potential functions in the report files, due to the high complexity and the size of the *x264* application.

akiyo_qcif was used as the input data file for encoding. It is a raw YUV 4:2:0 file with the resolution of 176×144 pixels, containing 300 frames. The output was in raw byte stream format. No subsequent encoding of the raw output file was performed.

[11] Officially, *MMX* is meaningless. However, it has been unofficially cited as *MultiMedia eXtension*, *Multiple Math eXtension*, or *Matrix Math eXtension*.

Table 3.1: *gprof flat profile of the x264 application on the Intel x86 architecture.*

Kernel	%time	Self seconds	Calls	Self ms/call	Total ms/call
pixel_satd_wxh	34.51	0.49	1361024	0	0
x264_cabac_encode_decision	8.45	0.12	10808084	0	0
get_ref	7.75	0.11	1165182	0	0
block_residual_write_cabac	7.04	0.1	400643	0	0
x264_pixel_sad_x4_16x16	5.63	0.08	88506	0	0
x264_frame_filter	5.63	0.08	2700	0.03	0.03
x264_pixel_sad_x4_8x8	3.52	0.05	243588	0	0
refine_subpel	2.82	0.04	151014	0	0
motion_compensation_chroma	2.11	0.03	442213	0	0

% time is the percentage of the total execution time of the program used by the function; ***Self seconds*** is the number of seconds accounted for by the function alone; ***Calls*** is the number of times a function is invoked; ***Self ms/call*** is the average number of milliseconds spent in the function per call; ***Total ms/call*** is the average number of milliseconds spent in the function and its descendants per call.

Experimental Analysis

x264 contains over two hundreds functions. The set of functions that are actually called is determined based on different options selected by the user or by the input/output file specifications. On the basis of the computationally-intensive kernels identified by *gprof*, we chose a number of functions (or series of functions) for further inspection. The main criterion was the suitability for the DWARV compilation tool. Table 3.1 presents part of the flat profile.

As specified in Table 3.1, **pixel_satd_wxh** is the main kernel of the application accounting for 35% of the total execution time. It was initially selected as the main candidate kernel for hardware mapping along with the **sad**-related functions. Although **x264_cabac_encode_decision** is the most frequently called function, each call has a smaller contribution compared to **pixel_satd_wxh**. As a result, the overall contribution of **x264_cabac_encode_decision** drops considerably. There are several **satd**-related functions defined in the form of *macros* corresponding to various block sizes. These macros, when expanded, create different functions calling the **pixel_satd_wxh**, hence, making it a very critical function on the execution path. Table 3.2 summarizes the results of memory access tracing for the **satd**- and **sad**-related functions. As expected, **pixel_satd_wxh** is the top consumer on the list (in total more than 300MB), as all the other **satd**-related functions call this kernel to perform their tasks.

It is worth noting that some of the **sad**-related functions (the ones with 4 rows and/or columns) do not exhibit any data transfers, which is an indication that they are not called. As mentioned before, this depends on the input file characteristics and options used. Although the kernels are intensely reading (writing) data from (to) memory, the number of UnMAs used in data transfers is limited (MBs data transfer vs. KBs locations). This indicates the possibility of allocating memory buffers, e.g., on FPGA BRAM to improve performance. It is worth to note that *Quad*-core can also provide a detailed map of the used memory addresses for investigating mapping opportunities on a target architec-

Table 3.2: *Summary of the data production/consumption of the **satd**- and **sad**-related kernels in the x264 application.*

Kernel	IN	IN UnMA	OUT	OUT UnMA
pixel_satd_wxh	326310528	137425	91578266	5126
x264_pixel_satd_16x16	40607660	1523	26133254	1233
x264_pixel_satd_16x8	5795852	1009	2849136	722
x264_pixel_satd_4x4	34342432	2745	18099806	2318
x264_pixel_satd_4x8	3091448	1953	1610194	1644
x264_pixel_satd_8x16	6248933	1053	3152620	650
x264_pixel_satd_8x4	3019905	1937	1568928	1594
x264_pixel_satd_8x8	91709500	3307	46203830	3049
x264_pixel_sad_16x16	59965275	103924	5704610	624
x264_pixel_sad_16x8	9159212	55626	1736556	480
x264_pixel_sad_4x4	0	0	0	0
x264_pixel_sad_4x8	0	0	0	0
x264_pixel_sad_8x16	8924130	53174	1709404	566
x264_pixel_sad_8x4	0	0	0	0
x264_pixel_sad_8x8	53730341	89155	10838624	664

IN represents the total number of bytes read by the function; ***IN UnMA*** indicates the total number of unique memory addresses used in reading; ***OUT*** represents the total number of bytes read by any function in the application from memory locations that the specified function has written to those locations earlier; ***OUT UnMA*** indicates the total number of unique memory addresses used in writing.

ture. The auxiliary functions communicating with kernels are recognized and presented in the QDU graph. These auxiliary functions can be a source of further investigation. For example, one might investigate mapping tightly-coupled functions on FPGA, and create a buffer to facilitate data transfer, or merge auxiliary function(s) with the primary kernel to cut off data transfers between the functions. In case of **pixel_satd_wxh**, **mc_copy_w16** is tightly coupled with the main kernel, and it is responsible for producing approximately 130 MB of data (75k UnMA). Further inspection of **mc_copy_w16** reveals that it belongs to the *motion compensation* library and merely calls the built-in *memcpy* routine of the C language library, in a loop, to create a block of pixels from a flat set of pixels with a predefined stride. Hinted by this, it seems feasible to rewrite the routine from scratch and to combine it with the kernel.

The **sad**-related routines are also defined in the form of macros corresponding to various block sizes. Unlike the **satd**-related functions, these macros–when expanded–create different functions with separate bodies. In order to evaluate the impact of an identical kernel routine for the **sad**-related functions, we created a new function called **pixel_sad_wxh** and revised all the **sad**-related functions to call this critical kernel. It seems to be a more suitable candidate for implementation on FPGA.

Table 3.3 depicts part of the flat profile for *x264* after the introduction of the new **pixel_sad_wxh** kernel. **pixel_sad_wxh** now gets the dominant position with the execution time contribution of about 33.5%. Note that it is also called nearly double of the times compared to the second dominant kernel, **pixel_satd_wxh**. The *gprof* flat profile of the *Quad*-instrumented binary is also provided. The considerable increase in the self-

seconds contribution of each kernel is due to the overhead introduced by the *QUAD*-core instrumentation and analysis routines. However, the ranking provided in this version is somehow more representative of the real execution time, stressing the data communication between the functions via non-local memory. Non-local memory accesses are filtered out[12], because only upon the detection of a non-local memory access, *QUAD*-core initiates the time-consuming tracing process in MAT.

Table 3.4 summarizes the memory profiling results of **pixel_satd_wxh** and the **sad**-related kernels in the revised version of the *x264* application. As expected, the communication load of **pixel_sad_wxh** dominates the former main kernel **pixel_satd_wxh**. Nevertheless, there is a substantial increase in the total amount of bytes consumed by this new kernel (about 800 MB) compared to the total amount of bytes consumed by the **sad**-related functions in the original version. This is due to the fact that the **sad**-related functions have to pass extra arguments to the new kernel. The new kernel uses the extra information to distinguish between the **sad**-related functions for different block sizes. The number of bytes consumed in **pixel_sad_wxh** can be further reduced by optimizing the code to minimize this overload. The total number of bytes produced and consumed beside the UnMAs used inside individual **sad**-related functions are significantly reduced, because the load is shifted to the new **pixel_sad_wxh** kernel.

Including the local memory accesses in the tracing would also reveal notable observations. By including stack area accesses, **pixel_satd_wxh** becomes the dominant kernel once again (22.15% of the total execution time contribution). This indicates that if one has no intention to place the local temporary memory blocks in hardware, and fetching data from external memory is expensive, most probably, mapping **pixel_satd_wxh** onto hardware is preferred than **pixel_sad_wxh**.

3.3.5 Bulk Data Flow Detection

In another case study, we examine *libdwt*[13], which is a cross-platform Discrete Wavelet Transform (DWT) library. It was primarily implemented in C, currently available in other languages as well. The library is available for common personal computing platforms (basic and optimized implementations), embedded processor platforms, hardware platforms (FPGA), and a combined environment (*EdkDSP*). *libdwt* implements a fast wavelet transform algorithm using *lifting scheme*. The fourth-order (with four vanishing moments) biorthogonal spline wavelets, also known as Cohen-Daubechies-Feauveau wavelets, were used in the experiments [57].

Here, we use the *QUAD*-core tool to pinpoint the main data flows that occur during the execution of the application. For this purpose, the primary output of the profiler, the QDU graph, is inspected to visually track the bulk of the data as it passes among the functions. It should be noted that what we discuss here is absolutely derived from the graph only, i.e. the flow detection is carried out with no review of the source code. As such, *QUAD* enables users to have an insight of the main path of data transfers in the application, particularly the ones who have no precise knowledge of the application

[12] To be more precise, their effect is considerably diminished.
[13] It is open source and the code is available from the Computer Graphics Research Group (VZ GRAPH), Department of Computer Graphics and Multimedia, FIT, Brno University of Technology, at http://www.fit.vutbr.cz/research/view_product.php?id=211.

Table 3.3: *gprof flat profile of the revised x264 application, both for un-instrumented and QUAD-instrumented binaries.*

Kernel	% time	Self seconds	Calls	Rank	% time(+QUAD)[a]	Self seconds	Rank(+QUAD)[b]
pixel_sad_wxh	33.45	0.5	2646060	1	22.94	546.53	1
pixel_satd_wxh	24.32	0.36	1361024	2	7.61	181.18	4
x264_frame_filter	8.11	0.12	2700	3	9.96	237.21	3
get_ref	7.43	0.11	1165182	4	7.49	178.41	5
motion_compensation_chroma	5.41	0.08	442213	5	3.25	77.53	7
block_residual_write_cabac	4.73	0.07	400643	6	2.64	62.96	8
x264_cabac_encode_decision	2.03	0.03	10808084	8	14.62	348.32	2
x264_macroblock_cache_load	2.03	0.03	29700	9	1.49	35.39	12
x264_cabac_encode_bypass	1.35	0.02	2234007	10	4.84	115.29	6

[a] The execution time as reported by *gprof* for the *QUAD*-instrumented version of the application.
[b] The execution times of the kernels for the *QUAD*-instrumented version of the application is somehow different from the original application. All the non-local memory accesses are now augmented by a factor on the account of the time-consuming analysis routine in *QUAD*-core. *gprof* notices this extra burden, and thus, reports substantially increased execution time. The ranks are derived based on this new observation, which somehow stress the memory access time in the whole execution time.

Table 3.4: *Summary of the data production/consumption of* ***pixel_satd_wxh*** *and the* ***sad****-related functions in the revised version of the x264 application.*

Kernel	IN	IN UnMA	OUT	OUT UnMA
pixel_sad_wxh	816414788	245781	100154164	2976
pixel_satd_wxh	326389008	137389	91580364	5108
x264_pixel_sad_16x16	16169821	885	7275066	614
x264_pixel_sad_16x8	4588984	793	2122520	510
x264_pixel_sad_8x16	4480742	843	2066828	552
x264_pixel_sad_8x8	39174841	1011	17382666	732

code. This is one of the important advantages of *QUAD* to provide a clear overview of the bulk data transfers in any application. Subsequently, the observations can be verified by inspecting the source code and performing any optimization that is hinted at by the profiling information.

The *libdwt* implementation has 31 functions. A sample application that uses the *libdwt* library was executed several times with different test cases. Each test case was based on a different input image size (256×256, 512×512, 1024×1024, 2048×2048), which is defined by the *SIZE* macro in the source code. Figure 3.8 depicts part of the output QDU graph corresponding to the image size of 2048×2048. We have omitted several nodes and some edges for the sake of clarity, as the original graph is quite dense and difficult to present here. The top critical functions are shown in the graph along with their execution time contributions.

By inspecting the graph in Figure 3.8, two types of intense data transfers are spotted in the application:

1. Several functions (*addr1_s*, *addr1_const_s*, *addr2_s*) are intensely transferring data which seems to be hard-coded (statically defined) in the application and passed through formal arguments to these functions. Although the communication is intense–which can be due to high number of calling times–the number of UnMAs used in transfers is low. This indeed means that the number of data items transferred should be limited to few. As an optimization hint, these functions should be *inlined* in the code, if possible, or constant values should not be passed as formal parameters to these functions.

2. The bulk flows of data transfers are within two paths. These may become the causes of memory bottlenecks, and should be addressed in case of any memory constraint and/or deficiency.

   dwt_util_test_image_fill_s ⟹ *dwt_cdf97_f_ex_stride_s* ⟹ *dwt_cdf97_i_ex_stride_s*

   Examining the amount of data transferred as well as the number of UnMAs reveals that the image data is passed along this path[14]. Reviewing the QDU graphs

[14] The resolution of the input image (2048×2048) makes it easy to verify the number of transferred bytes: 2048×2048×4 bytes=16 MB.

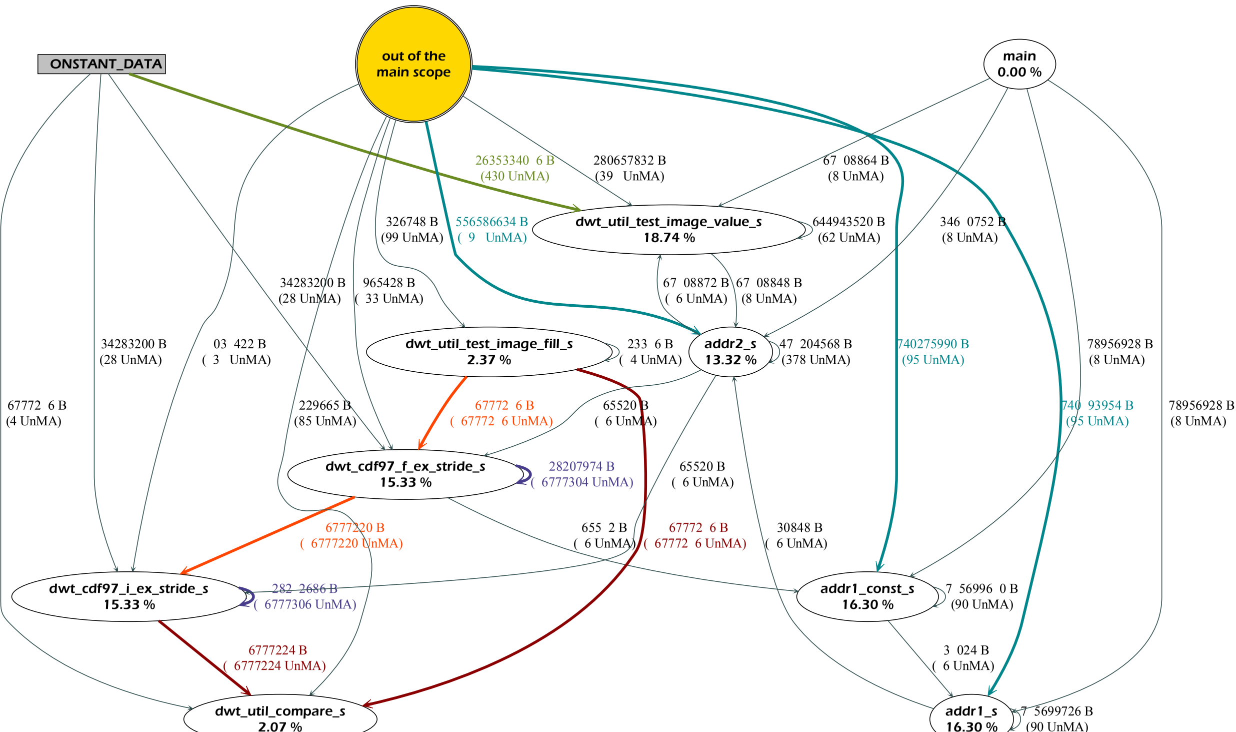


Figure 3.8: *Partial QDU graph of a sample application using the* ***libdwt*** *library. The main bulk of data transfers are highlighted in the graph. Whenever Q UAD detects that data is read from a memory address for which no function has set the value before, it is interpreted as* ***CONSTANT_DATA****.* ***Out of the main scope*** *means that not a user-defined function is responsible for the production (being set by an OS routine). To interpret the data transfers, particular attention has to be given to UnMAs. The values correspond to the test case with the input image size of 2048×2048.*

for other test cases indeed verifies this concept.

dwt_cdf97_i_ex_stride_s ⟹ *dwt_util_compare_s* ⟸ *dwt_util_test_image_fill_s*

dwt_util_compare_s receives two sets of image data, one from *dwt_cdf97_i_ex_stride_s* and the other one from *dwt_util_test_image_fill_s* in order to compare them.

Any memory optimization and tuning for the application source code should be focused around these intense data transfers because these have the potential to become bottlenecks for the overall system performance.

3.4 MAIP

Generally, profilers — including the most commonly used *gprof* — provide a cumulative execution time estimation for functions, failing to distinguish between the time spent on computations and the time spent on memory operations. This makes it inaccurate to assess the critically of the candidates in the context of heterogeneous multicore systems. The reason simply lies in dealing with the communication time overhead when data access accounts for a substantial share of the whole execution time. To address this deficiency, we developed an standalone general profiler, called *MAIP*, in the Q^2 profiling framework.

Memory Access Intensity Profiler aims to provide several unprocessed basic profiling items with the main objective of revealing the intensity of memory access operations. Furthermore, *MAIP* can be utilized as an alternative to (or in conjunction with) the common *gprof* profiler, allowing accurate measurements for programs which have only a small execution time. The problem with *gprof* is that the derived runtime figures are based on sampling process. Hence, besides being subject to statistical inaccuracy, there is a pretty good chance that *gprof* overlooks a function in action in case it runs for only a small amount of time. Only if the total execution time of an application is large enough, a small measured value by *gprof* truthfully indicates that the function's contribution is an insignificant fraction of the application's whole execution time. Otherwise, nothing valuable can be inferred from this observation at all. The extracted raw data by *MAIP* can be further processed to get valuable information, such as the ratio of memory access instructions to the total executed instructions.

The basic profiling data that are measured by *MAIP* are classified into three main classes: instructions, operands, and bytes. Each class is subsequently divided into Read-/Write and Stack/Non-stack subcategories. A variety of parameters is computed to assess the intensity of the memory access operations within an application. This is done by intercepting and inspecting every memory access instruction for detailed information. Specifically, *MAIP* extracts the following profiling data for each function:

- *The total number of instructions executed within each function call.*
  If there is more than one call instance for a particular function, the aggregate value of this parameter is used in subsequent processing to have a more accurate estimation.

- *The total number of memory access instructions executed within each function call.* As before, if there is more than one call instance for a particular function, the aggregate value of this parameter is used in subsequent processing.

- *Memory Access Ratio (MAR)*
  This is the percentual ratio of the total number of memory access instructions to the total number of instructions.

- *Non-Local Memory Access Ratio (NLOC-MAR)*
  This is similar to the previous parameter. However, here we only consider memory access instructions within the heap and global data regions. This parameter tends to provide a more accurate estimation of the MAR, when the cost of local memory access is considered to be low compared to the expensive external memory access.

- *Memory Operand Ratio (MOR)*
  This is the percentage ratio of the total number of memory access operands to the total number of operands.

- *Non-Local Memory Operand Ratio (NLOC-MOR)*
  This is similar to the previous parameter. However, here we only consider the operands of the memory access instructions referencing the heap and the global data area.

- *Stk Ratio*
  This is the percentage ratio of the total number of memory access instructions referencing the stack area to the total number of memory access instructions.

- *Flow Ratio*
  This is the total number of bytes read minus the total number of bytes written divided by the total number of bytes accessed. An extreme value of -1 means a *write-only* function, +1 represents the counterpart *read-only* function, and 0 represents a balanced R/W *inert* function.

- *NLOC-Flow Ratio*
  This is similar to the previous parameter. However, here we only consider memory accesses referencing the heap and the global data areas.

- *Byte-wise-Stk ratio*
  This is the percentage ratio of the total number of bytes accessed by the memory access instructions referencing the stack area to the total number of bytes accessed.

- *Bytes/Acc. ratio*
  This indicates the average value of the number of bytes accessed within each memory access instruction.

In the interpretation of these parameters, it should be generally noted that in some architectures, a single memory operand can be both read and written, for instance *incl (%eax)* on IA-32. In this case, instrumentation is performed once for read and once for write. The same holds for combined R/W instructions.

3.4.1 *Maip* Implementation

Maip is implemented using the *Pin* DBI framework, which provides interfaces to inspect various system resources, including the CPU registers. Like the *Quad*-core tool, *Maip* also requires instrumentation at instruction and routine granularities. However, we only utilize *RTN_AddInstrumentFunction()* to directly register the instrumentation function at the routine level. If only a routine seems to be of interest for inspection (the ones specified in the monitor list file), we set off to instrument the instructions contained in the routine. In this way, the profiling runs much faster. Instruction instrumentation is carried out via traversing every single instruction within each routine and inserting a call to an external function, which gathers necessary data for generating the profiling report file.

The main function in *Maip* initializes the *Pin* runtime system and parses the command line arguments. The command line arguments support the filtering of specific functions for further processing, and the report file specifications. Inspecting only a selected list of functions speeds up the profiling task. Otherwise, the cost of instrumenting all the functions in an application can be huge, and the execution slowdown degrades the performance of the profiler. The routine instrumentation function sets up the call to a corresponding analysis function every time a function is called. This is necessary because, *Maip* maintains its own internal call stack to avoid entering the library routines as well as undesirable functions. Furthermore, *Maip* ignores updating the internal call stack for the functions that are not in the main image file of the application. The call stack is also used to keep the required data for each function under inspection. It is vital to retain an accurate measurement as the application moves around nested function calls.

The instruction instrumentation function keeps track of the number of instructions and operands seen so far. It also iterates over each memory operand of the instruction, if any, to gather appropriate information. The task is performed for both read and write operands. A call to an analysis routine is set up every time an instruction references a memory location. The analysis routine updates the collected memory access data based on the properties of the instruction, being read or write, within stack area or non-local, etc.

The IP, the value of the SP, the EA and the size of the accessed memory location, and a flag indicating whether or not the instruction is a prefetch, are passed to the analysis routine. These arguments are necessary for the collected profiling data update. The analysis routine returns immediately upon detection of a prefetch state. A dedicated mechanism is also implemented to ensure the accuracy of the data when an instruction with a combined R/W operation is processed. Furthermore, the *ret* instruction is specifically monitored to maintain the integrity of the internal call stack. Detection of this instruction also implies the finalization of the collected profiling information for the current (to-be-terminated) function.

3.4.2 Computation Time vs. Communication Time

Finding the exact time that is spent on each memory access, if possible at all, is a very difficult task. This is because memory access time depends extensively on the intrinsic

nature of the underlying platform and its instantaneous state. It becomes even harder when targeting Complex Instruction Set Computers (CISCs), as no easy way exists for measuring the time taken for a particular memory access. Using the profiling information extracted by MAIP, we make an initial attempt to formulate — for a simplified scenario — the estimation of the time spent on memory operations in distinction of the time spent on computations.

Assume that for each kernel k, the time needed for accessing the memory system (τ_{comm}) is proportional to the total number of issued memory accesses during the execution of that kernel (η_{ma}):

$$\forall k : \tau_{comm}(k) \propto \eta_{ma}(k). \tag{3.1}$$

Furthermore, the time needed for accessing the memory can be estimated by Equation (3.2), assuming a completely primitive flat memory architecture without considering any hierarchies, caches, compiler optimizations, and other complexities:

$$\begin{aligned} \tau_{comm}(k) &\approx \xi \times \tau_{gprof}(k) \times MAR(k) & (3.2) \\ &\approx \alpha \times \tau_{stk}(k) + \beta \times \tau_{heap}(k) + \gamma \times \tau_{data}(k), & (3.3) \end{aligned}$$

where $\tau_{gprof}(k)$ is the total cumulative time reported by *gprof* for the kernel k and *MAR* is the ratio of the total number of memory access instructions (η_{ma}) to the total number of instructions. τ_{stk} denotes the communication time accessing the stack area. Accordingly, τ_{heap} refers to the heap and τ_{data} to the data areas of the memory. α, β and γ reflect cost factors for accessing data objects in stack, heap, and data areas, respectively. Using the profiling data provided by MAIP, we can revise $\tau_{comm}(k)$ as follows.

$$\tau_{comm}(k) \approx \tau_{gprof}(k) \times MAR(k) \times \left(\alpha \times \frac{\eta_{stk}(k)}{\eta_{ma}(k)} + \beta \times \frac{\eta_{heap}(k)}{\eta_{ma}(k)} + \gamma \times \frac{\eta_{data}(k)}{\eta_{ma}(k)}\right). \tag{3.4}$$

In the context of reconfigurable systems, it makes sense to consider two distinctive types of memory accesses. Local data objects, commonly limited in size, are allocated in the on-chip memory (BRAMs and/or LUTs) and all the other global and dynamically allocated data objects, rather large in size, are put on any kind of off-chip memory, e.g., SDRAMs. These can be addressed with the terms, *local* data, usually referring to the data objects in the stack area, and *non-local* data, which commonly refers to the data objects in the global scope and heap areas. In this respect, we rewrite Equation (3.4) as the following:

$$\begin{aligned} \tau_{comm}(k) &\approx \tau_{gprof}(k) \times \left(\alpha \times MAR(k) \times \frac{\eta_{stk}(k)}{\eta_{ma}(k)} + \zeta \times MAR(k) \times \frac{\eta_{heap}(k) + \eta_{data}(k)}{\eta_{ma}(k)}\right) \\ &\approx \tau_{gprof}(k) \times \left(\alpha \times MAR(k) \times \left(StkRatio\right) + \zeta \times MAR_{nloc}(k)\right), & (3.5) \end{aligned}$$

where $MAR_{nloc}(k)$ is the ratio of the *non-local* memory access instructions to the total number of instructions executed in the kernel k. α and ζ reflect cost factors for accessing

data objects in *local* and *non-local* areas, respectively. We define the Memory Access Intensity (MAI) of a kernel as a metric to distinguish between the intensity of kernels with respect to their memory access operations.

$$MAI(k) = \frac{\tau_{comm}(k)}{\eta_{ma}(k)}. \tag{3.6}$$

In order to evaluate the criticality of kernels, using the *MAI* metric is preferred over the *execution time* metric (see Section 3.6). MAI captures the communication time property of kernels, which is unavailable in the plain execution time metric.

3.5 Runtime Extraction of Source-Level Data

The *Quad*-core tool lacks the ability to relate the extracted memory access information to the individual data objects defined in an application source code. This deficiency implies that a troublesome burden is imposed on the application developers to revise the code based on the extracted information. In other words, it is complicated for developers to discretize exactly where, inside a kernel, a certain memory access behavior appears.

xQuad enhances the memory access profiling by revealing the correspondence between data objects defined in the application source code and the relevant runtime memory access information. For this purpose, it first retrieves source code representations (identifiers) of data objects from the Debugging With Attributed Record Formats (DWARF) debugging section of executables. This is necessary as *Pin* does not provide any information about data object identifiers. *xQuad* performs two types of analysis: 1) a detailed data object analysis for each function, and 2) a global memory access usage analysis of the entire application.

Performing such a detailed analysis at this fine level of granularity (data objects in the source code) can produce profiling data, which can soon become unmanageable, both in terms of the contents and the required time. Thus, it is necessary to implement the feature for selective filtering according to the user need and preference. This is implemented by feeding a text-based input file to the tool upon analysis start. The user can specify the functions to be profiled as well as the list of specific, or all, local data objects to be analysed. Furthermore, it is also possible to include global data objects in the analysis. The extracted information is output in flat text files, which allows the user to inspect these files later for desired information via postprocessing. The extracted information can also be visualized with third-party visualization tools to get a general outline of the memory usage. To facilitate postprocessing, the text files have a simple format, which makes it possible to parse them for extracting specific information, such as the frequency of accesses on a particular data object.

In the following, we first describe the DWARF debugging file format, which is used by many compilers and debuggers to support source level debugging, and then briefly talk about the implementation of *xQuad*.

```
.debug_info
COMPILE_UNIT<header overall offset = 306>:
<0><11> DW_TAG_compile_unit
            DW_AT_stmt_list     218
            DW_AT_high_pc       0x8048571
            DW_AT_low_pc        0x80483e4
            DW_AT_comp_dir      /comp/dir/dwarf
            <1>< 686> DW_TAG_subprogram
                      DW_AT_sibling   <867>
                      DW_AT_external  yes(1)
                      DW_AT_name      main
                      DW_AT_low_pc    0x804845e
                      DW_AT_high_pc   0x8048571
                      <2><715> DW_TAG_variable
                                DW_AT_name      s
                                DW_AT_decl_file 1
                                DW_AT_decl_line 53
                                DW_AT_type      <238>
                                DW_AT_location  DW_OP_fbreg -44
            <2><699> DW_TAG_subprogram
                      ------
                      DW_AT_name                  function_sample
                      <2><715> DW_TAG_variable
                                DW_AT_name          m
                                DW_AT_decl_file     1
                                DW_AT_type          <238>
```

Figure 3.9: *A sample Debugging Information Entry (DIE). DIE is the basic descriptive entity in DWARF. The DIE type is specified by a* tag. *Additionally, a DIE contains a set of attributes which describe types, names, source line numbers, location addresses, or a references to another DIE — e.g., a variable's reference to a data type specification.*

3.5.1 DWARF Debugging Information

The *Pin* DBI framework provides no API for retrieving information about source-level data objects, including identifiers referencing variables. Therefore, this information should be extracted directly from the Executable and Linkable Format (ELF) object file. By compiling the application with debugging information flag on, *gcc* augments the ELF object file with a debugging section, whose format is, unless specified otherwise, DWARF2 [67, 68]. DWARF provides debugging entries to define representations at the source code level, like, among others, information about source code types, function and object names, line numbers, instruction addresses, and EA offsets. These information are stored in different sections, all prefixed with the *debug_* keyword. The most important section is *.debug_info*, which is the DWARF core data structure containing the actual debugging information. The debugging information is, in turn, structured in tree-like Debugging Information Entry (DIE), as depicted in Figure 3.9. We modified the data layout to resemble the tree structure of the *.debug_info* section.

As most modern programming languages are block structured, DWARF also follows this model. Hence, each DIE, except the topmost root DIE representing the Compilation Unit (CU) of the source file, is contained in or owned by a parent DIE, and may have sibling DIEs or children DIEs. DIEs are tree nodes, which may represent data types, variables, functions, and everything else which participates in the formation of the object code. A DIE has a *tag*, which specifies what the DIE describes, and a list of attributes which fill in details and further describes the entity. Attributes may contain a variety of values: constants (such as a function name), variables (such as the start address for a function), or references to another DIE (such as for the type of a function's return val-

ue). Following the DWARF structure, source-level information is retrieved by using the *libdwarf* consumer library interface [9]. This library abstracts away from the DWARF low-level routines, by defining wrap functions that ease the process of retrieving debugging information from object files.

3.5.2 *xQuad* Implementation

xQuad consists of the following three parts:

- a module for retrieving DWARF debugging information;
- instrumentation functions based on the *Pin* API;
- the *Quad*-core's analysis functions, which are called in the instrumentation functions.

In the following, we provide an overview of the most important routines and peculiarities in the *xQuad* implementation. After the initial preprocessing, *xQuad* processes the debugging information that is read from the object file(s). Debugging information processing is completely *transparent*, which means the user is not aware of the process and the only requirement is that the object file(s) should include debugging information–which is indeed a compulsory prerequisite. Using the *libdwarf* library, the DWARF process is initialized and, afterwards, each CU inside the application is analysed. Following the tree structure of the DWARF format, individual CUs are examined through a depth-first traversal algorithm. During the traversal, only the information interesting for the user (according to the previously defined list of functions and variables) is further processed. The processing involves saving variable names and their offsets with respect to the current frame *base pointer* into a table, which is kept temporarily in the main memory.

After reading the debugging information, *xQuad* starts the instrumentation process by using relevant APIs from the *Pin* DBI framework. Using the instrumentation at routine granularity, *xQuad* retrieves the name of each function by using the API routine *RTN_name()*. Note that unlike data object identifiers, *Pin* is able to make the name of the functions available for the user. Function names are stored in the internal stack data structure maintained by *Quad*-core. The internal call stack is used in various aspects of the memory access analysis, such as the implementation of the *routine count functionality* and the implementation of the *memory map file*. It also enables *Quad* to find out whether or not the current executing function is called for the first time.

Using the instrumentation at the instruction level, *xQuad* catches are memory accesses. Each time an instruction references memory (read or write), the tool calls an analysis routine that inspects whether or not the current memory address belongs to a data object of interest defined earlier. This is done by checking if the examined function matches a function in the table created during the DWARF information extraction phase. When a memory access instruction is executed, various parameters are passed to the analysis routine *RecordTrace()*, such as *REG_GBP* representing the current base pointer. To calculate the actual memory address of the variable in question, the offset

stored in the table of the offset-variable name pairs, is added to the base pointer. Furthermore, the EA of the instruction is passed to the analysis routine, along with the current function name. Using these parameters, another table is built which consists of the pairs *variable name* and *variable address*. Finally, the *INS_IsPrefetch()* parameter is used to indicate if the current instruction is a prefetch instruction. Provided that it is the case, the analysis routine returns immediately, because we are only interested in the actual memory operations.

Another analysis routine is defined for the purpose of keeping a count of the executed instructions. This analysis function is called for every instrumented instruction, regardless of being memory access related or not. Keeping the record of executed instructions allows *xQuad* to have an estimation of the time a certain memory operation is executed. This temporal data can then be postprocessed to gain useful information about the lifetime of data objects defined in the application source code.

3.6 Kernel Ranking Based on MAI

The intensity of a kernel with respect to its memory access behaviour serves as a critical parameter when it comes to mapping the kernel onto the hardware. Based on the profiling information extracted by *Quad*, we develop a *ranking method* to define a memory access penalty factor for the execution time retrieved by *gprof*. This will be an indication of the memory access intensity of a kernel relative to its execution time. However, the ranking per se does not reflect any particular quantitative value of measurement. It only specifies an *index* by taking the ratio of the memory accesses over the memory access related part of the *gprof* execution time. As a result, this index is only applicable for comparison purposes between kernels. We elaborate on this through a real-world application example, *hArtes wfs*.

In the audio domain, a primary source wavefront can be created by secondary audio sources (plane of speakers) that emit secondary wavefronts, the superposition of which creates the original one. The Wave Field Synthesis (WFS) [32] is a 3D spatial audio rendering technique characterized by the creation of virtual acoustic environments. WFS is based on the *Huygens principle*, which, informally, states that each point in a wavefront can be considered as a *primary source* for the creation of new *secondary waves*, which, in turn, become a primary source for other waves. Hence, an advancing wave can be constructed by the summation of all the secondary waves arising from previously *primary source* waves. This principle is reproduced by *loudspeaker arrays* that generate a complete sound field in the listening zone, which is identical to an appropriate real sound event. Each of these speakers is activated at the exact time when the desired virtual wavefront passes through it to reproduce the original wavefront of the audio source. The *hArtes wfs* application provided by *Fraunhofer IDMT* [80] implements a self-contained wave field synthesis system.

General Observations. We used the *Quad* toolset to profile the *hArtes wfs* application. Some preliminary observation are made by inspecting the memory access profiles of the *hArtes wfs* application for the three distinctive areas, namely the stack, the heap, and the data. It turns out that from the beginning until approximately half of the execution,

Table 3.5: *Memory access statistics for the hArtes wfs application, divided in stack, heap, and data sections.*

Kernel	Stack			Heap		Data	
	# DO	# ADD	# ACC	# ADD	# ACC	# ADD	# ACC
wav_store	4	38	3160179	35935	291844	537	131932
fft1d	10	33	2192660	6	13	9	59
DelayLine_processChunk	18	41	893129	3	5	9	25
bitrev	2	26	922918	7	32443	10	64303
zeroRealVec	1	19	324462	3	281	7	504
AudioIo_setFrames	2	14	665	32	32	5	12
perm	4	22	126662	5	17	10	50
cadd	1	24	86742	5	16155	10	32378
cmult	3	24	123200	6	16173	10	32251
Filter_process	3	20	81054	5	19	9	47

#DO is the number of local data objects defined in the kernel; ***#ADD*** is the total number of distinct addresses referenced during the application execution; ***#ACC*** is the total number of accesses for data objects.

there is a sparse usage of heap memory addresses, while during the second half of the execution, this usage becomes quite intense for a certain range of addresses in the heap memory. Intensive heap usage is accounted for the *wav_store* kernel, which becomes active approximately in the middle of the execution, and it is the only active function until the end. Source code examination reveals that *wav_store* saves the output audio signals from the buffers allocated in the heap memory to an output file. To accomplish this, it uses mostly individual heap addresses, which explains the intense heap usage.

Table 3.5 summarizes all the memory references of the *hArtes wfs* application along with the number of individual memory addresses used[15]. The detailed profiling data produced by *x*Quad contains the memory addresses referenced during a kernel's execution, the corresponding data object name, the kernel's call number, and the *timestamp* when the reference is issued. The number of data objects presented in Table 3.5 refers to the local variables defined in the kernel, excluding the formal parameters. The recorded accesses are based on variables, which can have different sizes. Therefore, *x*Quad does not reveal the actual number of bytes accessed during the execution of a kernel. To have an aggregate estimation of this value, *t*Quad can be used (see Chapter 4). The detailed memory usage information of each data object defined in the source code is not presented here. The data in Table 3.5 is recorded with a *time slice* length of 500 instructions. To put in other words, these profiling data are describing *almost completely* the actual memory usage of the application through the time. Selecting a larger time slice results in the reduction of the analysis time and the output file size[16]. However, the choice of a *too large* time slice should be avoided, as this would cause the loss of some valuable information.

From Table 3.5, we can see that the number of individual memory addresses for ac-

[15] Only the top ten kernels of the application with the largest execution time contributions are listed.

[16] The output memory map file for the *hArtes wfs* application with a time slice of 500 instructions is almost 50 MB. By storing the profiling data for each instruction, the file can grow up into GBs, which may make the postprocessing of the data impractical.

Table 3.6: *The gprof profiling data for the hArtes wfs application on the Intel x86 architecture.*

Kernel	%time	Self seconds	Calls	Self ms/call	Total ms/call
wav_store	31.91	0.28	1	277.25	277.25
fft1d	28.23	0.25	984	0.25	0.25
DelayLine_processChunk	14.23	0.12	493	0.25	0.38
bitrev	8.19	0.07	2015232	0.00	0.00
zeroRealVec	7.44	0.06	15782	0.00	0.00
AudioIo_setFrames	4.01	0.03	493	0.07	0.07
perm	2.07	0.02	984	0.02	0.09
cadd	0.79	0.01	1009664	0.00	0.00
cmult	0.73	0.01	1009664	0.00	0.00
Filter_process	0.71	0.01	493	0.01	0.73

cessing the non-local memory is considerably higher for *wav_store* compared to the other functions. Examining the detailed *xQuad* profiling data reveals that, in the case of stack area, more than one third of the total accesses are due to a local variable that acts as a sentinel for the main loop which stores the specifications of wave frames to the output file[17]. Another interesting observation is that the primary load of the local accesses in *DelayLine_processChunk* originates from a rather large data object, which collects the necessary data for delay update. Should the kernel be implemented in the hardware, on-chip allocation of this data object in addition to two small counters, results in more than fifty percent reduction in the total external memory accesses.

AudioIo_setFrames is responsible for copying interleaved audio signal parts into the corresponding audio frames. The *hArtes wfs* application uses 32 secondary sources, i.e. an array of 32 loudspeakers, thus, *AudioIo_setFrames* needs 32 distinct addresses for doing this task. The data presented in Table 3.5 can be misleading per se, particularly for the stack area, as it does not take into account the number of times that a function is called. Therefore, the *gprof* profiling data is also presented in Table 3.6 to indicate the execution frequencies of the functions and their execution times. At a first glance, *bitrev* shows a high frequency of stack usage, as presented in Table 3.5. Nevertheless, this function is called over two million times, which implies that the number of local memory accesses are only a few instances per call.

MAI Ranking. Using *Maip*, it is possible to distinguish between the computation and the communication parts of the kernels in the *hArtes wfs* application. Table 3.7 presents a partial summary of the *Maip* profiling results. In case a kernel shows different behaviors in subsequent calls, an average value is calculated. Almost all the kernels had similar behaviors in recurring calls and the differences were negligible, except for *zeroRealVec* which acted significantly different in one case. As Table 3.7 shows, for most of the kernels, more than half of the whole executed instructions are due to memory access operations. This memory communication load can increase up to nearly the whole execution time for strictly memory-bound kernels, such as *AudioIo_setFrames*.

[17] The detailed *xQuad* profiling data is not included in Table 3.5.

Table 3.7: *Communication vs. computation profiling data of the hArtes wfs application on the Intel x86 architecture.*

Kernel	MAR	NLOC MAR	Total Inst.	Total MA Inst.	Stk Ratio	# Unique Exec.
wav_store	37.14	24.54	3389224874	1258788805	33.92	1
fft1d	54.43	13.06	1411388	768176	76.00	6
DelayLine_processChunk	54.24	10.30	1009733	547848	81.04	139
bitrev	51.31	5.48	264	136	89.33	4
zeroRealVec	54.59	9.16	11277	6156	83.21	6
AudioIo_setFrames	99.66	99.37	132127	131677	0.30	4
perm	65.86	11.33	70235	46260	82.81	6
cadd	60.22	18.18	86	51	69.72	4
cmult	63.46	15.40	94	60	75.75	4
Filter_process	53.08	22.45	100520	53351	57.71	7

MAR is the percentage ratio of the memory access instructions to the total instructions executed in the kernel; ***NLOC-MAR*** is the same as ***MAR*** except that only references to the non-local area are considered; ***Stk Ratio*** is the percentage ratio of the memory access instructions within the local area to the total memory access instructions; ***# Unique Exec.*** is the total number of distinct statistical data recorded for different calls of the kernel.

Using Equation (3.6), we propose a ranking to evaluate the criticality of kernels with respect to their communication time. In a simple scenario, suppose that the cost of accessing the local memory area is totally insignificant compared to the non-local memory area, i.e α is close to zero in Equation (3.5). Table 3.8 presents an order of the kernels based on the probable suitability for mapping onto hardware. Lower value for MAI indicates that the kernel is more appropriate for hardware implementation. The total execution time is retrieved from the *gprof* profiling data as presented in Table 3.6, while the total number of memory accesses is extracted from Table 3.5. In Table 3.8, the stack accesses of the kernels are not taken into account, as most time penalty is expected to be for accessing the heap and data segments of the memory. The η_{ma} column reports the total number of memory accesses for the heap and the data areas, while $\tau_{comm}(k)$ is an estimate of the time spent for executing memory operations. The ordering is based on the inverse values of the MAI for the kernels.

As seen in Table 3.8, *cadd* and *cmult* get the top positions in the ranking. These tiny frequently-used kernels are responsible to do mathematical addition and multiplication for complex numbers. A thorough inspection of the source code also justifies the placement. The reason lies in the fact that *cadd* consists of only two floating point addition operations (computation workload) and six memory access operations in total, four memory reads and two memory writes (communication workload). A similar scenario applies to *cmult* with five additions, three multiplications, and sixteen memory accesses. It should also be stressed that mapping these kernels to hardware still does not affect the overall performance considerably, because they are only responsible for a very small fraction of the whole execution time of the application. Apart from these kernels, *wav_store* still accounts as one of the most appropriate kernels for mapping onto the hardware. Interestingly, *fft1d* and *DelayLine_processChunck*, which had top positions in the *gprof* profiling list, drop down to near the last position in the ranking. The distinct case of *AudioIo_setFrames*, which is the strictly memory-bound kernel in the list, is quite interesting. This kernel somehow gets now its actual position in the ranking. It is

Table 3.8: A Ranking based on the MAI of the kernels in the hArtes wfs application.

Kernel	η_{ma}	τ_{comm}	$(\mathbf{MAI})^{-1}$	Rank
wav_store	423776	0.068712	6167423	4
fft1d	72	0.032650	2205	9
DelayLine_processChunk	30	0.012360	2427	8
bitrev	96746	0.003836	25220542	3
zeroRealVec	785	0.005496	142831	5
AudioIo_setFrames	44	0.029811	1475	10
perm	67	0.002266	29568	6
cadd	48533	0.001818	26695819	2
cmult	48424	0.001540	31444156	1
Filter_process	66	0.002245	29399	7

η_{ma} is the total number of memory accesses instructions during the execution of the kernel and τ_{comm} is the time needed for accessing the memory system.

identified as the worst candidate for the hardware mapping. Examining the source code reveals that *AudioIo_setFrames* merely copies interleaved audio signal parts to a frame by invoking the *memcpy* library function. This means that the time spent for memory accesses *relative* to the computation time is dominating, thus, making it an inappropriate candidate for hardware implementation.

3.7 Summary

The gap between the processor and the memory performance still continues to be the major challenge for computing systems. It is even intensified with the emergence of today's heterogeneous multicore systems, not to mention the rich repository of legacy applications that have been inherited from the earlier platforms. This demands the development of tools to help developers in mapping and tuning applications for the maximal performance gain of these systems. In this chapter, we presented the Q^2 profiling framework which, in general, plays a crucial role in mapping applications onto any heterogeneous multicore platform. We detailed the dynamic part of the profiling framework, the *QUAD* toolset, which aims to provide a comprehensive analysis of the memory access behavior of an application. Utilizing a Dynamic Binary Instrumentation technique to find the actual data dependencies between functions, *QUAD* is able to detect coarse-grained parallelism opportunities as well as to extract useful information regarding the memory requirements of an application.

We further demonstrated the usefulness of *QUAD* in practice by profiling several real applications. For each case study, we highlighted some major observations followed by detailed comments. Furthermore, based on the extracted profiling information, we introduced a ranking strategy which can provide a preliminary estimation of the criticality of the functions regarding their memory access intensity.

Note.

The content of this chapter is partly based on the following articles:

S. Arash Ostadzadeh, Roel Meeuws, Carlo Galuzzi, and Koen Bertels, **QUAD - A Memory Access Pattern Analyser**, *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC'10)*, Bangkok, Thailand, March 2010, pp. 269-281.

S. Arash Ostadzadeh, Marco Corina, Carlo Galuzzi, and Koen Bertels, **Runtime Extraction of Memory Access Information from the Application Source Code**, *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS'11)*, Istanbul, Turkey, July 2011, pp. 647-655.

Koen Bertels, S. Arash Ostadzadeh, and Roel Meeuws, **Advanced Profiling of Applications for Heterogeneous Multi-Core Platforms**, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'11)*, Las Vegas, USA, July 2011, pp. 171-183.

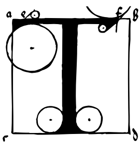
CHAPTER 4

Temporal Memory Bandwidth Analysis

"... even good programmers and language designers tended to do terrible extensions when they were in the heat of programming, because design is something that is best done slowly and carefully." †

— Alan Curtis Kay

A thorough behavioral analysis of memory accesses in an application is not feasible in the absence of timing information. Though not built in the original QUAD*-core tool, relative timing data — which indicates the progress of the application execution — is extracted during memory access instrumentation. This is of critical importance, particularly in scheduling and mapping tasks onto heterogeneous systems. In addition to be of assistance in detecting memory bandwidth problems, the extracted timing information is utilized as well to identify different phases in the application.*



EMPORAL ANALYSIS of tasks is a crucial aspect in application development and/or its mapping onto heterogeneous multicore systems. Basically, this kind of analysis is required to extract necessary information about the *life cycles* and the *time spans* of functions that are invoked during a typical execution of an application. When mapping an application onto a heterogeneous system, the most critical phase is task partitioning, which can seriously affect the efficiency of the mapped application. Based on previously defined characteristics and properties, tasks can be executed in software mode (e.g., on a GPP) or in hardware mode (e.g., on an FPGA). Additionally, there are tasks that can be implemented either in software or in hardware, depending on the availability of adequate hardware resources. These decisions are an essential part of task scheduling and mapping in the application development process. Almost in all computing systems, including those incorporating

† *A Conversation with Alan Kay*, ACM Queue, Vol. 2, No. 9, Dec./Jan. 2004-2005.

heterogeneous PEs, task scheduling is of vital importance and is a well-studied topic [14, 19, 43, 181, 198]. In the case of reconfigurable systems, some advanced features, such as dynamic partial reconfiguration, further raise the complexity of the task scheduling problem. *Task scheduling and subsequently placing designated tasks on the hardware are inseparable parts of porting applications into heterogeneous systems utilizing reconfigurable hardware. Efficient handling of this process is largely dependent on the availability of appropriate temporal information about the application tasks [4, 116, 148].*

Additionally, over the past few years, the complexity of applications has considerably increased, most notably with regard to the interactions among different functions. This is partly because of the well-structured and modular application development process, which encourages the practice of *software reuse* [79]. As revealed in practice, in heterogeneous reconfigurable systems, developers implement more functionality in software than in hardware due to its flexibility and shorter time-to-market, among others [30, 145, 197]. This, in turn, brings about the necessity of a proper analysis of tasks' interactions with a particular focus on timing aspect. It is necessary to examine the behavior of tasks individually as well as in mutual communication with others. Consequently, dedicated profiling tools targeting heterogeneous reconfigurable systems require to have an appropriate estimation of the timing measures for the called functions in an application.

To address this issue, a proper temporal analysis of tasks' execution is inevitable. *It is important to note that due to the nature of the analysis performed in our profiling framework, we care about the relative*[1] *timing of functions compared to each other, not the absolute timing. The absolute time obviously varies, depending on the hardware characteristics of the target architecture and other influential factors.* In our profiling framework, the *Quad* main tool, *Quad*-core, lacks any temporal analysis of the instrumented code. *This feature is intentionally excluded from the main module due to the imposition of extra overhead in the case of tracing memory accesses comprising temporal data.* Nevertheless, because of the criticality of this requirement, a separate module, called *tQuad*, is developed to provide detailed relative timing information for each function called during the execution of an application. As an immediate result, *tQuad* enables *Quad* to deliver temporal memory bandwidth usage information for individual functions.

Memory bandwidth information is gathered by comprehensive analysis of all memory accesses in the presence of timing estimations. The extracted information is subsequently processed to identify *virtual* phases[2] in the life span of the application. In this chapter, together with an elaborate description of *tQuad*, we examine, in detail, a real application from the multimedia domain to substantiate the potential of *tQuad*. Additionally, the virtual phases of the application are detected as a side outcome.

The remainder of this chapter is organized as follows. In Section 4.1, we briefly discuss the extraction of temporal data from applications. Then, in Section 4.2, we present an overview of practical approaches used to extract timing information in profiling tools. Implementation details of *tQuad* are discussed in Section 4.3. In Section 4.4, a real application is examined in detail to demonstrate the potential and applicability of the *tQ*

[1] Relative time is an approximation of the absolute time, which maintains the ordering of the events. However, relative time measurement cannot be accounted for, by itself, as an accurate assessment of the wall-clock time.
[2] *Virtual* phases can correspond to *actual* phases defined in the application, however, with no knowledge of the implemented algorithms at hand, it would not be always feasible to make matches with real phases.

UAD extension in the Q² profiling framework. Finally, Section 4.5 concludes the chapter.

4.1 Background

The two primary approaches for behavioral application analysis, i.e. *static* and *dynamic*, can be used for timing analysis. Static analysis has to estimate the timing progress in a system without executing any application. As a result, some kind of a *model* of the system or implementation artifact is required that is correlated directly with the timing behavior. Additionally, static analysis of an application source code can explore dimensions of timing behavior that are hard to investigate through manipulation of the application input data. Static analysis tools typically construct program execution models, potentially through *reverse engineering*. These models can then be analyzed to derive and ensure a particular behavioral characteristic [103].

Conventionally, model checking techniques are used to check exhaustively all feasible paths of execution and interleavings between different entities if relevant. This avoids leaving any behavior unchecked, and thus, ensures correctness properties [55]. Nevertheless, model checking is computationally very expensive in practice, hence, limited in its applicability. This can be partly due to the enormous number of possible interleavings a large system may exhibit. Other static analysis techniques, such as automated checking of design intent [125] and program analysis driven theorem proving [96], have been also utilized to ensure behavioral correctness. The common characteristic among all these techniques is the trade off between analytical thoroughness and computational cost. Static analysis techniques typically prove to be successful in modeling relative timing and temporal ordering of events. However, they are incapable of accurately modeling the wall-clock time; hence, they are not used for measuring the absolute time.

Dynamic analysis is the only *practical* approach to behavioral analysis that can incorporate aspects of the absolute time. Dynamic profilers inspect the behavior of a system at runtime, thus they are able to measure various aspects of the system that are exactly representative of the real scenarios. In the case of absolute timing aspect, the exactness of measurements may be diminished due to some influential factors, such as sample-based timing measurement or slowed down execution due to extra overhead. Additionally, by using a system that has somehow different architectural or platform-related properties, the accuracy of absolute timing measurements is compromised as well. Dynamic approaches are also implemented in *active* or *passive* modes. Active profiling requires that the application under inspection explicitly generate information about its execution, such as compiler-based probe insertion, where the application makes callbacks to the trace collection engine to record temporal data. Conversely, passive approaches rely on the inspection of control flow and execution state through an external entity, such as a probe or modified runtime environment. The advantage of passive profiling is that it typically does not require any modification of the examined application; however, it is hard to implement and requires specialized tracing facility.

In a nutshell, dynamic profilers can collect precise and fine-grained behavioral data from a running application, which can also be coupled with post-processing to help summarize and reason about observed results. The collected data is thus accurate and

representative of real execution as long as some critical factor, such as the overhead of the measurement, has not unduly influenced the results. Moreover, these profilers can only provide behavioral data for application paths that are actually taken during execution. Hence, successfully applying dynamic profiling tools usually depends on analyzing multiple runs of the application to test all relevant paths. Full coverage can be achieved through careful selection of representative input data or through artificial input generators. Because of this limitation, dynamic profiling is more useful for behavior analysis in those circumstances where samples of behavior are sufficient, such as determining the approximate life span or ordering of functions in an application. It is not well-suited to ensure correct behavior in a critical system where, for example, real-time deadlines shall be met. Both static and dynamic analysis have their own advantages and disadvantages. Advanced behavioral analysis solutions may require to combine both approaches to provide a comprehensive picture of the application behavior [222].

4.2 Temporal Data Extraction

Determining the execution time of functions in an application is very common in general profilers targeting various platforms. This is partly because knowing the execution time of an application is of fundamental importance for the acceptability of many systems, especially for systems delivering *real-time* services. The correct behavior of a real-time system depends both on the result of the computation and on the time at which the result is produced [75]. While for static profilers, this only means obtaining an *upper bound* on the execution time in a specified hardware model, dynamic profilers take into consideration a specific hardware and derive the execution time from the recorded results [220].

Hard real-time systems require a deterministic timing behavior of the application to guarantee the termination of tasks execution. This is guaranteed by estimating the Worst Case Execution Time (WCET) of the application, classified as a static method. WCET is defined as the longest execution time that will ever be observed when an application runs on a specific hardware, for all the possible input data [220]. Commercial and research prototype tools for WCET estimation are available, such as *aiT* [5], *Bound-T* [42], *Chronos* [127], *Heptane* [97], *SWEET* [191], and *Symta/P* [192]. Each of these tools apply common phases for the WCET calculation. Usually WCET tools work on binary executables, as binaries contain all the information needed for the analysis. In the first phase, the binary is decoded for the construction of the CFG. The graph is used to perform a control-flow analysis by determining the possible application paths. Additionally, infeasible paths are excluded and loop bounds are determined. Most tools automatically identify an upper bound on the number of loop iterations. However, it may be the case that a tool fails at retrieving these bounds. In such a case, the user has to input loop bounds via a *manual annotation* method, either in a separate text file or embedded in the source code, which is then recompiled into a new executable. The second phase is usually called *Value Analysis* or *Processor-Behavior Analysis*. This phase needs a simplified model of a microarchitecture, which gathers information on caches, memory, pipelines, branch prediction schemes, and other hardware components, to produce timings for the application paths. The model can be "built-in" inside the tool or, in case of a new target

processor, it can be constructed and fed to the tool by the user. Subsequently, by using the information acquired during the first two phases, the final bound calculation is performed. This produces WCET estimations in terms of *cycles* which could be then, if supported by the tool, converted to seconds.

The fact that WCET estimation needs a model of the target microarchitecture, even though simplified, can be quite cumbersome, as most vendors do not disclose enough information on their microarchitectures. Hence, the outcomes of tools must be validated by real measurements before considering the calculated WCET reliable. Additionally, the complexity of hardware is very high and, in most cases, it is difficult to extract an accurate model. Estimating a WCET on state-of-the-art hardware with simplified models may result in an unreliable estimation. Therefore, a measurement-based analysis beside a static analysis is a common practice when estimating WCET. As a matter of fact, when the real execution time of a certain application on a particular hardware is desired and not necessarily an upper bound on execution time, the user performs a measurement-based analysis of the system. To cope with this necessity, hybrid approaches combining measurement and static analysis are developed, such as *RapiTime* [172].

Static WCET analysis can deliver an over-pessimistic or inaccurate timing estimation. This drawback, in addition to the hardware complexity, causes static techniques to be inefficient when applied to heterogeneous reconfigurable systems. In these systems, task scheduling and mapping algorithms must strive to make the reconfigurable hardware area usage as efficient as possible. The algorithms usually work in an iterative manner to prune a huge design space, and eventually find an optimal solution for application mapping. The rather slow timing estimation inherent in static approaches reinforces the need for dynamic analysis techniques which aim at providing precise information on temporal aspects in an acceptable time. This is very critical for developers to effectively port their applications to the target platform. It even gains more relevance considering the fact that absolute individual task execution time is mostly not needed for early design space exploration. However, relative timing among different tasks is of critical importance.

Conventionally, general profilers provide estimations of the execution time of functions in an application. These estimates are valid for the platform on which the application runs and cannot be considered accurate when targeting a different platform. Profilers like *gprof* [89] are regarded as dynamic analysis tools. *gprof* performs the analysis based on source code instrumentation. The problem with *gprof* is that the derived runtime figures are based on a sampling process. As a result, besides being subject to statistical inaccuracy, if a particular function runs for only a small amount of time, it is very probable that the function be actually overlooked in action. Only if the total running time of an application is large enough, a small value shows that the function's contribution to the whole execution time of the application is insignificant. Otherwise, no valuable information can be derived from *gprof* measures at all. Additionally, it should be noted that *gprof* does not distinguish among computation time and memory access time. Deciding to map a task onto reconfigurable hardware, merely based on the total execution time, may result in an inefficient usage of the hardware resources. This eliminates the advantages of a hardware implementation because the CPU-time expensive tasks may be in reality bounded by their memory access behavior.

Some dynamic analysis tools integrate hardware event monitoring facilities such as

counters. Examples are the *Intel VTune* [215] and the *AMD CodeAnalyst* [56]. Both tools provide the user with a suite for application performance analysis which includes, among others, a time-based analyser that helps locating the application hotspots and bottlenecks as candidates for optimization. *VTune* profiles everything that is executed on the CPU, including information about the OS, third party libraries and drivers. It also locates memory hotspots and relates them to the code hotspots. The source code is not required to use VTune, however, for accuracy, debugging information are necessary, even though they are not always required. Compiling the application by including debugging information allows the displaying of the source code with its execution time against each line of the code. The *VTune* time-based sampling approach gathers information on the percentage of time by interrupting the application's execution at regular time intervals (1 *ms* by default) and recording instruction pointer addresses. The most frequently executed portions of code are reported in terms of clock ticks. In addition to reporting hotspots, *VTune* produces a detailed analysis at architectural level by specifying how the application behaves in memory and by identifying problems such as cache misses. Similar to *VTune*, *CodeAnalyst* suite performs system-wide profiling and supports the analysis of both user applications and kernel-mode software. It also collects instruction pointer addresses at predefined time intervals (the finest time resolution is 0.1ms) and it reports bottlenecks, execution penalties, and optimization opportunities. Furthermore, *CodeAnalyst* makes it possible to identify what causes certain bottlenecks on the architecture level. For a given application, it reports the number of CPU cycles needed by a code region, the Instructions Per Cycle (IPC) and its inverse Cycles Per Instruction (CPI) and statistics on data accesses. Both these tools are, however, hardware dependent.

In *tQuad*, we utilize dynamic binary instrumentation to record plain relative timing data. The raw data is subsequently processed by several tools to reason about the timing information required in support of decisions in application mapping. In our profiling framework, temporal data extraction is based on relative timings, mainly because of the following issues:

- Absolute time extraction can only be verified by running the application on the target architecture, the Molen reconfigurable machine in our case (see Section 3.1.1). However, the *Pin* [134] DBI framework is not available for this architecture. Running the application on any other platform provides absolute time measurements, which lack accuracy for the Molen architecture.
- The extracted profiling information is mainly used in *early* design stage for porting the application into a heterogeneous reconfigurable platform. The absolute timing measurements do not have an influential role in the decision making process and/or high-level optimizations, which may be derived by the profiling information.

4.3 *tQuad* Implementation

tQuad uses the *Pin* DBI framework to extract the required runtime information from the application. Figure 4.1 provides an overview of the components in the *tQuad* implemen-

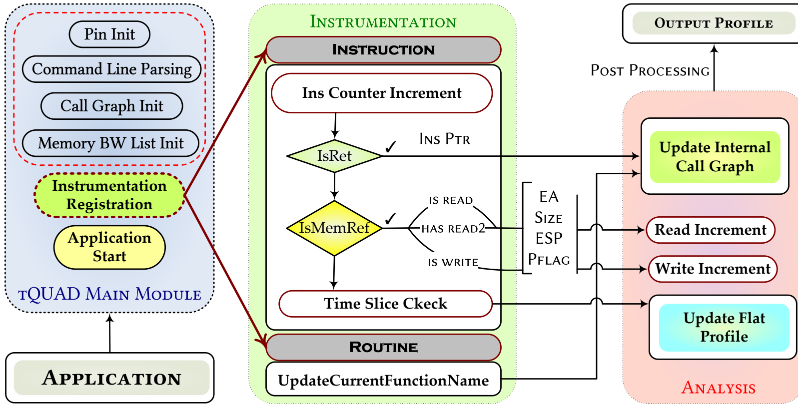


Figure 4.1: *An overview of the tQUAD implementation.*

tation. The three modules comprising *tQUAD* are the *main* module, the *instrumentation* module, and the *analysis* module. Preprocessing includes initializations and registration of instrumentation routines. For each instruction executed in the application, the instruction counter is incremented. Time slice checking is also performed after each instruction to dump the extracted information at predefined time intervals. For each instruction referencing memory addresses, the designated analysis functions are invoked to collect the timing and bandwidth usage data. Furthermore, the instruction which signals the return from a function is monitored to ensure the consistency of the internal call stack. The analysis module contains the routines to process the extracted data during the runtime in order to create the output profiling results.

The interfaces to most runtime binary instrumentation systems are API calls that allow users to hook in their instrumentation routines. Listing 4.1 shows the pseudocode of the main *tQUAD* interface in C++ style. At the beginning, there are several initializations for memory bandwidth usage data list, internal call stack and a mutual kernel-to-bandwidth data map list. *PIN_InitSymbols()* must be called to access functions by name. After initializing the *Pin* runtime system, the command line arguments of *tQUAD* are parsed to set primary parameters for the profiling process. Three options are supported, namely, the inclusion/exclusion of the local stack area memory accesses, the time slice interval setting, and the exclusion of memory bandwidth usage data caused by OS and library routine calls. When mapping a kernel on a reconfigurable device, there may be the possibility to allocate the corresponding local buffer on the hardware as well, provided that enough space is available for the size of needed memory block. In this case, all the local memory accesses should be distinguished from external memory accesses. In *tQUAD*, we provide the option to estimate the usage of the memory access bandwidth including or excluding the local stack area. Time slice interval is a key parameter that adjusts the detailing degree of the extracted memory access bandwidth information. Using large time slices, some information is lost and a coarser view of the memory bandwidth usage of functions is obtained. Library and operating system routines are usually of no interest to the user, thus, *tQUAD* has the option to exclude them from the analysis.

In *Pin*, the API call to *INS_AddInstrumentationFunction()* allows a user to instrument

```
//Data Structures
class CallStack; // ADT for internal call stack
class MBWUDataList { // Memory Bandwidth Usage Data List
    list<MACC*> mbwulst;
    // MBWUDataList members
    ...
 } MBW;
class K2BW; // ADT for kernel to MBW Usage Data mapping

//Global Variables
   string mainImg; // The main image name
   ofstream fprofile; // Temporary flat profile for snapshots data
   UINT64 TSInterval; // The time slice interval
   BOOL Uncommon_Functions_Filter=TRUE;
   BOOL No_Stack_Flag = FALSE;
   UINT64 CurrentInstructions=0; // Current number of executed instructions
   UINT64 SliceNumber=1; // Current time slice number
   CallStack CS;
   K2BW Kernels2MBWListMap;

// Command line Options
KNOB<UINT64> KnobSlice(KNOB_MODE_WRITEONCE, "pintool","slice","500000","Specify time
      slice interval in terms of the number of instructions");
KNOB<BOOL> KnobIgnoreUncommonFNames(KNOB_MODE_WRITEONCE, "pintool","
     filter_uncommon_functions","1","Filter out uncommon function names" );
KNOB<BOOL> KnobIgnoreStackAccess(KNOB_MODE_WRITEONCE, "pintool","ignore_stack_access
     ","0","Ignore memory accesses within application's stack area");

int main( int argc, char *argv[] )
{
    MBW.Init();// Memory bandwidth usage data list initialization
    CS.Init(); // Internal call stack initilization
    Kernels2MBWListMap.Init(); // Kernels <--> MBWU Map
    GetMainImg(); // Parse commandline for primary image name

    PIN_InitSymbols();
    if( PIN_Init(argc,argv) )  return Usage();
    CkeckTS(TSInterval=KnobSlice.Value());
    Uncommon_Functions_Filter=KnobIgnoreUncommonFNames.Value();
    No_Stack_Flag=KnobIgnoreStackAccess.Value();

    RTN_AddInstrumentFunction(UpdateCallStack,0);
    INS_AddInstrumentFunction(Instruction,0);
    PIN_AddFiniFunction(Fini,0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

Listing 4.1: *tQuad main interface.*

programs based on a single instruction, while the *RTN_AddInstrumentFunction()* provides instrumentation capability at routine granularity. We use these two API routines to set up calls to the instrumentation routines *Instruction()* and *UpdateCallStack()*, respectively. Listing 4.2 shows the body of the *Instruction()* instrumentation routine. The *Instruction()* instrumentation routine sets up the call to the analysis routine *IncreaseRead()* every time an instruction that references memory read is executed. A similar process is followed in the case of memory write reference. *Instruction()* also monitors instructions for the return from a function to maintain the integrity of the internal call stack. When *Pin* starts the execution of an application, the JIT compiler calls *Instruction()* to insert new instructions into the code cache. If the instruction references memory or signals the return from a function, *tQuad* inserts a call to the corresponding analysis routine before the instruction, passing the required arguments which are the IP, the number of bytes read or written, and a flag showing whether or not the instruction is a prefetch. The corresponding analysis routines return immediately upon detection of a prefetch state for an instruction. *INS_InsertPredicatedCall()* injects the analysis routine and ensures that the analysis routine is invoked only if the instruction is predicated true. When local stack area memory accesses have to be excluded, the Stack Pointer (*REG_STACK_PTR*) is also passed as an extra argument to the analysis routine for subsequent processing. Furthermore, *Instruction()* is responsible to initiate the time simulation and memory access bandwidth monitoring for taking snapshots at predefined time intervals.

The source code of the *UpdateCallStack()* instrumentation routine is presented in Listing 4.3. The *UpdateCallStack()* instrumentation routine sets up the call to the analysis routine *EnterFC()* every time a function is called during program execution. This is necessary to update the internal call stack. Since *tQuad* ignores the functions which are not in the main image file of the application, *flag* is used as a signal to indicate the location of the most recently invoked function. The name of the function, as reported by *Pin*, is also passed to *EnterFC()* for the sake of updating internal call stack.

4.4 Case Study: Wave Field Synthesis

tQuad was tested on several real applications, specifically for phase detection based on memory bandwidth usage information. Each application requires customized explanation of results to further verify the usability and accuracy of *tQuad* results. In the rest of this chapter, we present a detailed analysis of the *hArtes wfs* application (see Section 3.6). The main goal is to have a thorough understanding of the application behavior regarding the memory access bandwidth usage of the kernels. The extracted information can be further used for critical decisions, such as hardware/software partitioning, mapping, and scheduling. The information can also be useful to spot bottlenecks related to the memory usage of the application. Additionally, it can assist application developers to revise the source code in order to gain performance increase on a particular platform.

In this case study, we specifically aim to achieve the following goals:

- the extraction of information about the intensity of the memory bandwidth usage for each kernel during its execution. This information is critical in finding the potential memory access related bottlenecks when the application is executed on

```
void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)IncTotalInstCount, IARG_END);

    if (INS_IsRet(ins))  // returning from routines is monitored
       INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)Return, IARG_INST_PTR,
            IARG_END);

    if (!No_Stack_Flag)  // stack accesses is ok
    {
      if (INS_IsMemoryRead(ins) || INS_IsStackRead(ins))
         INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)IncreaseRead,
              IARG_MEMORYREAD_SIZE, IARG_UINT32, INS_IsPrefetch(ins), IARG_END);
      if (INS_HasMemoryRead2(ins))
         INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)IncreaseRead,
              IARG_MEMORYREAD_SIZE, IARG_UINT32, INS_IsPrefetch(ins), IARG_END);
      if (INS_IsMemoryWrite(ins) || INS_IsStackWrite(ins))
         INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)IncreaseWrite,
              IARG_MEMORYWRITE_SIZE, IARG_UINT32, INS_IsPrefetch(ins), IARG_END);
    } // end of Stack is ok!

    else  // ignore stack accesses
    {
      if (INS_IsMemoryRead(ins))
         INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)IncreaseReadSP,
              IARG_REG_VALUE, REG_STACK_PTR, IARG_MEMORYREAD_EA,
              IARG_MEMORYREAD_SIZE, IARG_UINT32, INS_IsPrefetch(ins), IARG_END);

      if (INS_HasMemoryRead2(ins))
         INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)IncreaseReadSP,
              IARG_REG_VALUE, REG_STACK_PTR, IARG_MEMORYREAD2_EA,
              IARG_MEMORYREAD_SIZE, IARG_UINT32, INS_IsPrefetch(ins), IARG_END);

      if (INS_IsMemoryWrite(ins) )
         INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)IncreaseWriteSP,
              IARG_REG_VALUE, REG_STACK_PTR, IARG_MEMORYWRITE_EA,
              IARG_MEMORYWRITE_SIZE, IARG_UINT32, INS_IsPrefetch(ins), IARG_END);
    } // end of ignore stack accesses

    // due time to take a snapshot?
    INS_InsertCall(ins,IPOINT_BEFORE,(AFUNPTR)Slice_checkpoint,IARG_END);
}
```

Listing 4.2: *tQuad instruction instrumentation.*

```
1  void UpdateCallStack(RTN rtn,void *v)
2  {
3      bool flag;
4      char *rNtemp;
5      string rName;
6      flag=(!((IMG_Name(SEC_Img(RTN_Sec(rtn))).find(mainImg)) == string::npos));
7      rName=RTN_Name(rtn);
8      rNtemp=new char[strlen(rName.c_str())+1];
9      strcpy(rNtemp,rName.c_str());
10     RTN_Open(rtn);
11
12     // Insert a call at the entry point of a routine to update Call Stack
13     RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)EnterFC, IARG_PTR, rNtemp, IARG_BOOL
           , flag, IARG_END);
14     RTN_Close(rtn);
15 }
```

Listing 4.3: *tQUAD routine instrumentation.*

a particular heterogeneous reconfigurable architecture;

- the detection of the relative timing and *activity span* (starting and ending points of execution) for each kernel in the application. The extracted information is subsequently used for task scheduling and mapping [133];

- the identification of the main phases in the application, based on the activity spans of the kernels. The kernels that are active at the same time interval are *possibly* relevant (communicating data). This information can be later utilized in task clustering to help in efficiently partitioning the application.

In order to achieve these goals, several experiments were carried out. First, we used *gprof*, as a general available profiler, to quickly identify the top contributing kernels in the application as well as the call graph of the application. Then, *QUAD*-core is used to get an overview of the amount of data communication between the kernels in the application. Based on the extracted data, a discussion on the potential memory access problems is subsequently presented. We also profiled the *QUAD*-instrumented version of the *hArtes wfs* application to clarify the effect of data communications in the overall contribution of each kernel in the application. Finally, *tQUAD* is used in a series of experiments to extract the timing information for each kernel and to identify the main phases in the application.

4.4.1 Experimental Setup

All experiments were executed on an Intel 64-bit Core 2 Quad CPU Q9550 @ 2.83GHz with a main memory of 8GB, running Linux kernel v2.6.18-164.6.1.el5. The *hArtes wfs* source code was compiled with *gcc* v3.4.6. To use *gprof*, the application was compiled and linked with the *-pg* profiling option enabled along with the *-g* option for the debugging information to be available. The 64-bit version of *tQUAD* was used with the command line options to include/exclude stack area memory access and to adjust the time slice interval, ranging from 5000 to 10^8 instructions per time slice. The *hArtes wfs*

runs in off-line mode. This means that the input audio source is read from files instead of audio devices. In all experiments, we used one primary wavefront source and thirty two secondary audio sources (speakers).

Instrumentation-based tools can considerably slow down the execution of an application. Since memory read and write instructions are executed frequently in multimedia applications, the overhead of intercepting and checking these instructions is very high. *tQuad* instruments every *load*, *store*, *call* and *return* instruction, which results in a slowdown of the execution of the *hArtes wfs* application, ranging from 37.2× to 68.95× compared to the native execution. The amount of introduced overhead is strongly dependent on the time slice and the option to include/exclude stack area accesses. Despite such significant slowdown, the execution time is comparable with the expected slowdown [199], hence, it is acceptable for a realistic working set data.

4.4.2 Kernels Overview

The *hArtes wfs* application consists of 64 functions. We used *gprof* to identify the top computation-intensive kernels for further inspection. As stated before, the runtime figures provided by *gprof* are based on sampling. Hence, they are subject to statistical inaccuracy, particularly if a function runs only for a small amount of time. In our experiment, the sampling period is one hundredth of a second, which is a fair indication of the accuracy of a function figure regarding its total execution time. Since the total runtime of application is rather large, a small runtime value for a function indicates that the function uses an insignificant fraction of the whole execution time. We executed the *hArtes wfs* application fifty times to gain more accuracy. The results are summarized and presented in Table 4.1.

It was determined that **wav_store** and **fft1d** are the top two kernels. These kernels together account for approximately sixty percent of the whole execution time of the application. Inspecting **wav_store** reveals that the function saves the output audio signals from buffers allocated in memory to an output file in the *.wav* format. **fft1d** implements a fast algorithm to compute the one-dimensional Discrete Fourier Transform (DFT) using the in-place (no additional memory allocation) butterfly *Danielson-Lanczos* method.

4.4.3 Quantification of Data Communication

Table 4.2 provides an overview of the amount of data communication between kernels in the form of producer/consumer bindings. The results take into consideration the inclusion and the exclusion of the stack area memory accesses. By comparing the data extracted from the individual cases, a lot of information can be derived. From Table 4.2, it can be seen that in most cases the ratio between the amount of data produced/consumed for the stack inclusion to exclusion is limited. However, it is not the case with **zeroCplxVec** and **zeroRealVec** as the ratios are greater than 750 and 300, respectively. This means that the mentioned kernels are nearly reading all the time from the local memory. In other words, they can be excellent candidates for hardware mapping provided that the corresponding input buffer is also placed on the chip. However, their intense communication with the memory for writing data into the output buffers should not

Table 4.1: *gprof flat profile for the hArtes wfs application.*

kernel	**%time**	**self seconds**	**calls**	**self ms/call**	**total ms/call**
wav_store	31.91	0.28	1	277.25	277.25
fft1d	28.23	0.25	984	0.25	0.25
DelayLine_processChunk	14.23	0.12	493	0.25	0.38
bitrev	8.19	0.07	2015232	0.00	0.00
zeroRealVec	7.44	0.06	15782	0.00	0.00
AudioIo_setFrames	4.01	0.03	493	0.07	0.07
perm	2.07	0.02	984	0.02	0.09
cadd	0.79	0.01	1009664	0.00	0.00
cmult	0.73	0.01	1009664	0.00	0.00
Filter_process	0.71	0.01	493	0.01	0.73
wav_load	0.44	0.00	1	3.80	3.80
Filter_process_pre_	0.35	0.00	493	0.01	0.35
zeroCplxVec	0.28	0.00	495	0.00	0.00
r2c	0.16	0.00	490	0.00	0.00
c2r	0.14	0.00	493	0.00	0.00
AudioIo_getFrames	0.14	0.00	489	0.00	0.00
ffw	0.08	0.00	2	0.35	0.35
vsmult2d	0.02	0.00	7026	0.00	0.00
calculateGainPQ	0.02	0.00	6994	0.00	0.00
PrimarySource_deriveTP	0.02	0.00	236	0.00	0.00
ldint	0.01	0.00	1	0.10	0.10

% time is the percentage of the total execution time of the program used by the function; ***self seconds*** is the number of seconds accounted for by the function alone; ***calls*** is the number of times a function is invoked; ***self ms/call*** is the average number of milliseconds spent in the function per call; ***total ms/call*** is the average number of milliseconds spent in the function and its descendants per call.

be ignored. The output buffers should also be instantly accessible to fully exploit the performance gain.

Approximately half of the giga bytes read by **wav_store** originates from the stack memory. This indicates that a substantial portion of the data has been produced inside the function for internal processing. However, the size of the used memory in almost the same (around 60 MB) for both cases, including or excluding stack accesses. This means that a small area is locally allocated inside the function for temporary storage. The need to fetch data out of sixty five millions distinct locations into **wav_store** can pose a serious bottleneck. By examining the QDU graph of the *hArtes wfs*, which is produced by *Quad*-core, other useful information can be derived. For example, it turns out that nearly all the data produced by **wav_store** are used internally and the kernel discloses very limited amount of data for the other kernels. This remark can also be verified by the small number of UnMAs used as output buffers compared to the huge amount of data produced (hundreds of addresses per GBs). The **fft1d** case is somehow different as the ratio of stack inclusion to exclusion is approximately ten. This indicates that most of the computations are performed inside the kernel. It is also worth noting that, the size of the locally allocated memory used for temporary results is rather nominal due to the fact that the UnMAs reported in the two cases remain identical. The immediate outcome of this observation is that **fft1d** is a better candidate than **wav_store** for hardware mapping onto a reconfigurable device. This is particularly true if there is an intention to map the corresponding local buffers as well.

Table 4.2: *Summary of the data produced/consumed by the kernels in the hArtes wfs application.*

	Stack area accesses excluded				Stack area accesses included			
kernel	**IN**	**IN UnMA**	**OUT**	**OUT UnMA**	**IN**	**IN UnMA**	**OUT**	**OUT UnMA**
AudioIo_getFrames	2082977	2003143	2030924	4178	2193001	2003319	2132616	4290
AudioIo_setFrames	65642447	131797	64790862	64618668	66910617	131955	65875370	64618788
DelayLine_processChunk	136426363	187911	130079532	162800	1207848481	188349	1199055238	163146
Filter_process	76962891	65853	8367732	16562	166795095	66075	113578568	16744
Filter_process_pre_	8159527	16623	8288564	16480	8310811	16807	8428110	16614
PrimarySource_deriveTP	28658	271	9504	248	102558	785	81336	750
bitrev	147305084	145	64488030	86	1092514838	397	991569196	214
c2r	2062775	4231	2019224	4180	22360399	4433	22271396	4310
cadd	73825250	129	32309436	82	203213962	377	153474676	194
calculateGainPQ	654672	305	223904	270	2977380	1151	6046220	1384
cmult	73767500	137	32309306	74	235522840	393	185786118	194
fft1d	541111698	115143	348733474	86182	3377052372	115439	3178842792	86370
ffw	571706	4863	177374320	16640	832298	5496	177633766	17151
ldint	81	73	72	64	399	231	336	168
perm	15747216	55745	31271422	47762	190358486	55985	221582640	47914
r2c	2048600	4331	8028298	8458	26181770	4571	32117142	8600
vsmult2d	513564	159	224864	152	1414418	705	1807246	690
wav_load	73166075	5606	118994504	2000393	148386954	6668	194027099	2001719
wav_store	3407275698	64941803	1754503491	392	5946326334	64942676	4282480373	1115
zeroCplxVec	48499	171	8151616	41130	36631679	417	44664318	41282
zeroRealVec	1257818	219	65398908	140194	391633848	537	454905252	140406

IN represents the total number of bytes read by the function; ***IN UnMA*** indicates the total number of unique memory addresses used in reading; ***OUT*** represents the total number of bytes read by any function in the application from memory locations that the specified function has previously written to; ***OUT UnMA*** indicates the total number of unique memory addresses used in writing.

Table 4.3: *gprof flat profile for QUAD-instrumented version of the hArtes wfs application.*

kernel	% time	self seconds	rank	trend
wav_store	33.69	346.93	1	↔
fft1d	30.35	312.46	2	↔
DelayLine_processChunk	10.85	111.75	4	↓
bitrev	0.42	4.33	11	⇊
zeroRealVec	3.14	32.30	5	↓
AudioIo_setFrames	11.19	115.18	3	⇈
perm	1.52	15.69	7	↔
cadd	0.39	0.01	13	↓
cmult	2.12	21.80	6	↑
Filter_process	0.67	7.04	8	↔

% time is the percentage of the total execution time of the program used by the function; ***self seconds*** is the number of seconds accounted for by the function alone; ***rank*** is the position of the function among all the kernels; ***trend*** shows the intensity to increase or decrease the function's contribution compared to the initial flat profile.

As a general remark from Table 4.2, although all the kernels are intensely communicating with memory, which is common for A/V processing applications, the size of the memory addresses used for data transfer is limited (100MB-1GB of data vs. KBs of UnMAs). However, a thorough analysis of the data in Table 4.2 discloses a critical potential bottleneck arising from the memory access pattern of **AudioIo_getFrames** and **AudioIo_setFrames**. In these kernels, the data transfer is carried out via separate memory addresses. This is the reason why the number of bytes and UnMAs are almost identical in the corresponding columns. The case is quite critical for the data written into the memory addresses in **AudioIo_setFrames** (more than 60 MB of data are saved in distinct memory addresses). This behavior, undoubtedly, will surpass any performance gain that can be achieved by running the kernel in hardware, e.g., on an FPGA. As a matter of fact, **AudioIo_setFrames** is responsible for copying interleaved audio signal parts into relevant audio frames in the memory. This is the reason why it is saving data in completely separate locations. The detailed information in the QDU graph can even allow us to trace back the source of the data which is originating from **DelayLine_processChunk**. Later, **AudioIo_setFrames** passes the data to **wav_store** to be processed. By examining Table 4.1, we can see that **AudioIo_setFrames** is only contributing to four percent of the whole execution time. Nevertheless, with the huge impact of the memory communication problem, it seems underestimated. Unfortunately, general profilers, such as *gprof*, are not able to provide an accurate estimation of the memory access overhead impact on the overall kernel performance when a program is profiled. In fact, the timing information estimated by *gprof* can not precisely describe the behavior of an application in practice, particularly when there is an extreme interaction with the memory system whose response time is influenced by some critical parameters.

QUAD-instrumented profiling data. We also profiled the QUAD-instrumented version of the *hArtes wfs* to have a genuine overview of the application's behavior. Certainly, this version tends to reveal the data communication overhead introduced by accessing individual memory addresses. Furthermore, it stresses costly global memory accesses in

contrast to the less expensive local memory references during the execution of the program. Table 4.3 summarizes the results for the previously identified top ten kernels. The considerable increase in the timing contribution of each kernel is accounted to the overhead introduced by the instrumentation and analysis routines. Meanwhile, the ranking of kernels in this version is somehow more representative of a real execution, particularly on systems that have a very expensive access cost for external memory compared to on-chip local buffers. The reason is that the instrumentation routine simply discards the local stack area accesses and only upon detection of a non-local memory access, an analysis routine is called to handle a tracing process.

It is worth noting that, due to the long execution time of the instrumented program (a couple of hours) the statistics extracted from the flat profile show a high level of accuracy. Only very slight deviations can be detected in different runs. As expected, there is a substantial increase in the contribution of **AudioIo_setFrames** from 4 to 11 percent. By profiling the QUAD-instrumented version of the program, we can distinguish between local and global memory accesses. We can also take into account the size of the memory blocks used in the data transfers. For instance, **bitrev** and **DelayLine_processChunk** have more or less the same ratio of including to excluding stack area accesses. **bitrev** shows a severe drop on the execution time contribution (from 8.19 to 0.42 percent). However, this is not the case with **DelayLine_processChunk**. This observation is justified by looking at the reported UnMA usage for the two kernels in Table 4.2. **bitrev** only uses around one tenth of a KB as buffer, whereas **DelayLine_processChunk** accesses about 180 KB of memory locations.

The kernels in the *hArtes wfs* show a huge diversity in the number of times they are called, ranging from one to millions of calls. The huge number of calls does not necessarily mean that the corresponding kernel has a large contribution to the total execution time. Instead, the frequently-invoked kernels have often quite a simple body. Anyhow, the case of the top kernel in *hArtes wfs* is quite interesting: **wav_store** is called only once and it has the contribution of about one third for the whole execution time. It clearly indicates that the kernel must be active in a large time span during the execution of the application.

4.4.4 Temporal Information Extraction

We utilized *tQuad* to have a clear view of the running times of the kernels in the program. The extracted information is depicted in the form of running time graphs in Figure 4.2. The x-axis is the execution time. Each unit represents the time slice which is set to 10^8 instructions span. The y-axis represents the intensity of the memory accesses for each kernel at a specified time slice. The graphs for different kernels are shown along the z-axis. This makes it easy to compare the memory access behaviors of the kernels in each time slice. As expected, **wav_store** is called approximately in the middle of the execution time. It is silent in the first half and it is the only kernel active in the second half.

Figure 4.2 also shows the memory bandwidth usage of the top ten kernels in the *hArtes wfs* related to the memory read accesses including the stack area. Memory write accesses have almost similar figures but the intensity of the data transfers is less by at least a factor of two in most kernels. The time slice interval is set to 10^8, i.e. a snap-

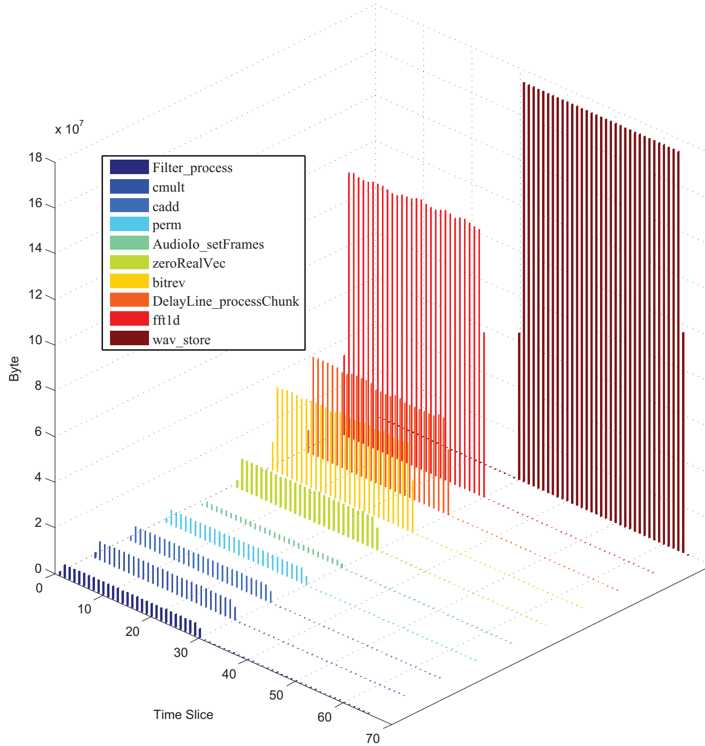


Figure 4.2: *Memory bandwidth usage of the kernels in the hArtes wfs, considering only the read accesses including the stack area.*

shot of the memory bandwidth usage is recorded every hundred millions instructions. In total, 64 time slices are counted representing the execution of more than six billion instructions for the completion of the program. Setting the time slice interval to a large number causes the loss of detailed information. This is evident in the density of the produced graphs. Small time slice intervals are preferable for more accurate estimations. Figure 4.3 depicts the relevant graphs for the last ten kernels. Here, the time slice interval is set to 25×10^6, which provides a more detailed view of the running times of the kernels. The second half of the total 255 time slices is cut off, as no kernel but **wav_store** is active during this period. The graphs depict the memory bandwidth usage of the kernels regarding the memory write accesses excluding the stack area. The memory access patterns of all kernels are strictly regular in *hArtes wfs*. This is common in nearly all the applications from multimedia domain as the processing algorithms are well-formulated to work on predefined data blocks.

4.4.5 Phase Detection

tQUAD recognized five different phases in the whole execution span of the *hArtes wfs* via thorough examination of different graphs. Several experiments were carried out to extract the required measurements for each phase. A summary of the results is presented

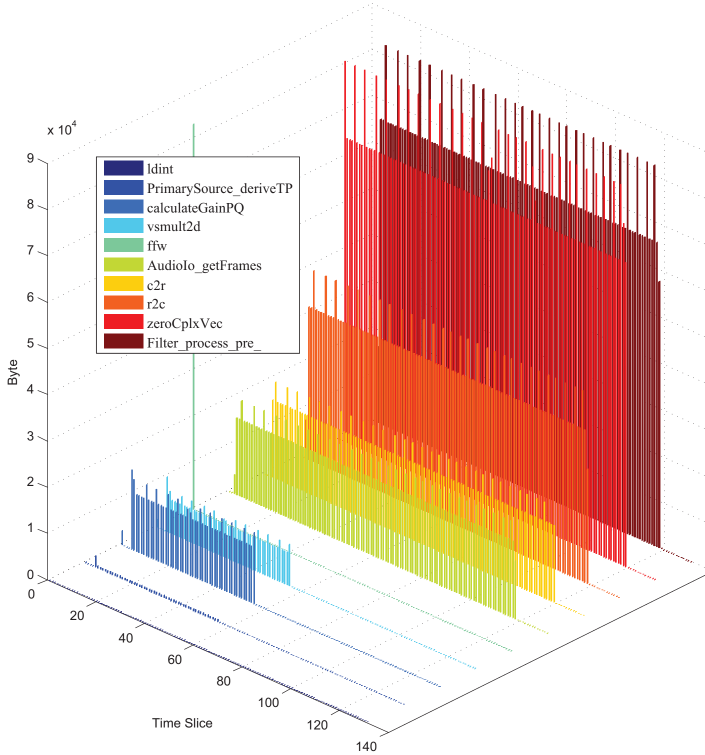


Figure 4.3: *Memory bandwidth usage of kernels in hArtes wfs, considering only the write accesses excluding the stack area.*

in Table 4.4. This depicts a clear image of the related active kernels in each phase including their self contributions and memory access patterns during the execution time of the program. The phases identified are mainly based on the role of the active kernels in that particular time span. Nevertheless, some kernels, such as **bitrev**, are utilized in a more general way, which causes the phases to overlap if we only consider the activity time span of the kernels. We set the time slice interval to 5000 in order to have accurate estimations of the memory bandwidth usage. However, the data presented in Table 4.4 are still prone to slight statistical inaccuracy. Nevertheless, this inaccuracy should, in no sense, distort the overall behaviors of the kernels. 1270684 time slices were measured in total.

The *phase span* column in Table 4.4 indicates the earliest starting point and the latest ending point in which a kernel in the phase is communicating with the memory. It should be noted that, we only consider the kernels previously selected and not all the functions in the *hArtes wfs*. Moreover, there are cases in which kernels are activated in a short period of time outside the identified span. We merely ignore these cases with respect to the overall memory access pattern of the kernels in a phase. As an example, **r2c** gets active in the 145th time slice for a very short time and then becomes silent until the 14663th time slice. Furthermore, the phase span does not necessarily mean that all the kernels within that phase are active through the whole time slices.

Table 4.4: *Identified phases in the execution path of the hArtes wfs application.*

phase	phase span	% phase span	kernel	activity span	average memory bandwidth usage: read access, stack incl.	read access, stack excl.	write access, stack incl.	write access, stack excl.	maximum memory bandwidth usage (R+W): stack incl.	stack excl.	aggregate MBW
initialization	53-144	0.007	ffw	92	1.8071	1.2422	0.4807	0.1811	2.4704	1.6376	2.6018
			ldint	1	0.0798	0.0162	0.0516	0.0176	0.1314	0.0338	
wave load	552-14660	1.1103	wav_load	14109	2.0993	1.0358	1.0355	0.9929	3.1566	2.0664	3.1566
wave propagation	540-274868	21.5891	vsmult2d	1570	0.1799	0.0655	0.1182	0.0503	0.3996	0.1548	1.4530
			calculateGainPQ	1600	0.3708	0.0815	0.2633	0.0847	0.7714	<0.2116	
			PrimarySource_deriveTP	235	0.0870	0.0240	0.0547	0.0208	0.2820	0.0980	
WFS main processing	14663-592803	45.4983	fft1d	278781	2.4179	0.3876	0.3501	0.1331	2.8738	<0.6428	<84.1862
			DelayLine_processChunk	115546	2.0859	0.2356	0.2339	0.1180	<3.3316	1.7050	
			bitrev	116755	1.8677	0.2521	0.7457	0.1934	<2.8778	0.4966	
			zeroRealVec	36304	2.1529	0.0145	0.7233	0.3610	<2.9386	<0.4028	
			AudioIo_setFrames	616	21.5553	21.8035	21.0646	21.5860	<53.2686	<52.7330	
			perm	116776	0.3252	0.0280	0.0956	0.0545	<0.6556	<0.1466	
			cadd	41076	0.9882	0.3590	0.6686	0.2753	1.6946	0.6514	
			cmult	41073	1.1456	0.3590	0.7080	0.2753	1.8946	0.6594	
			Filter_process	42583	0.7789	0.3609	0.1332	0.1141	<0.9768	0.5064	
			Filter_process_pre_	1487	1.1113	1.6384	1.6267	1.6290	<3.3862	<3.3302	
			zeroCplxVec	4132	1.7693	0.0141	0.5913	0.3926	2.6874	<0.4710	
			r2c	2716	1.9250	0.1510	0.4474	0.2983	<2.9386	<0.5642	
			c2r	2318	1.9251	0.1774	0.3549	0.1769	2.9138	0.4658	
			AudioIo_getFrames	502	0.8701	0.8296	0.8268	0.8137	1.7482	1.6866	
wave save	592804-1270674	53.3469	wav_store	677871	1.7492	1.0033	0.8064	0.7765	2.7244	1.9044	2.7244

phase span indicates the starting and ending time slices for the phase; ***% phase span*** is the percentage of the phase time interval to the program whole execution time span; ***activity span*** represents the number of time slices in which the kernel is active (accesses memory); ***memory bandwidth usage*** is measured in bytes per instruction; ***aggregate MBW*** represents the summation of all kernels' maximum memory bandwidth usage in the phase including the stack area accesses.

They can be quite active, such as **fft1d** or less active, such as **AudioIo_getFrames**. This behavior has nothing to do with the intensity of the memory communications of a kernel. As an example, **perm** is moderately active in the fourth phase. However, the memory communication is not intense at all. On the other hand, **AudioIo_setFrames** is only active in rather small time intervals but acts truly intensive in memory referencing. *tQ* *UAD* is capable of providing the detailed information about the exact time intervals in which a kernel is communicating with the memory. It also analyzes the data to identify the boundaries of potential phases.

The average memory bandwidth usage is calculated over several passes with different time slices. The data are normalized as number of bytes-per-instruction. In this way, it is possible to have a general estimation of the kernel's architecture-independent intensity. If a more specific unit of measurement is needed, additional parameters for the target architecture should be provided for *tQuad*, such as the number of PE cycles required to execute each instruction. It is also possible to derive different measurement units, such as bytes-per-cycle or bytes-per-second. The maximum memory bandwidth usage represents the maximum bytes-per-instruction measured in the peak of the communication with memory counting both read and write accesses. For some of the kernels in Table 4.4, the upper bounds are specified. This is due to the fact that slight inconsistencies in the measurements of the overall time slices were detected in the experiments.

The initialization phase runs only for a very short time interval, which makes it rather uninfluential in the overall analysis. The second phase contains only one kernel which is active throughout the whole time span. The kernels appearing in the third phase are related to the implementation of a Multiple-Input and Multiple-Output (MIMO) delay line and wave propagation computations for an array of speakers. Although the activity span of the kernels in the this phase cover more than one fifth of the whole execution time, they have a nominal share of the memory bandwidth traffic. The main WFS processing is carried out in the fourth phase, during which, fourteen kernels are active. As expected, this phase has the biggest share of the whole memory bandwidth traffic. **Filter_process_pre_** has almost identical amount of memory bandwidth usage in the cases of including and excluding the stack area accesses. **AudioIo_setFrames** and **AudioIo_getFrames** have similar traits. This also conforms to the information presented in Table 4.2 for the mentioned kernels. As mentioned before, **AudioIo_setFrames** shows a completely unique attribute among all the kernels in the *hArtes wfs* application. The intensity of the maximum memory bandwidth usage for this particular kernel reaches over 50 bytes per instruction, while for all the others, it is at most 3 bytes per instruction. Some kernels, such as **DelayLine_processChunk**, show a severe drop in the memory bandwidth usage by a factor of 10 when excluding stack area accesses. In the cases of **zeroRealVec** and **zeroCplxVec**, the factor is more than 125. Further investigation of the information in the flat profile (not presented here) also reveals that, by excluding the stack area accesses, the activity spans of these kernels are reduced by a factor of 2 and 8, respectively. It is observed that **wav_store** is the only kernel in the last phase. Despite being active for more than half of the whole execution time span, it cannot be exclusively as influential as the main WFS processing phase.

The information about the phases in an application and the active kernels in each phase along with the extracted quantitative memory access data provide valuable clues for partitioning an application with respect to certain criteria. We investigate applica-

tion partitioning in detail in Chapter 5. Although not all the information extracted by the *Quad* toolset is inclusively used in the partitioning process, we formulate a general application partitioning problem and discuss the procedure to carry out partitioning based on some memory access profiling data. More precisely, relevant kernels are clustered together in a sense that the intra-cluster communication is maximized, whereas the inter-cluster communication is minimized.

4.5 Summary

In this chapter, we presented *tQuad*, an extension for the *Quad* dynamic profiling toolset, that provides relative timing information for the individual functions in an application. It equips *Quad* to deliver temporal memory bandwidth information, which was lacking initially in the *Quad*-core tool. Memory bandwidth information is gathered by detailed analysis of all memory accesses in the presence of the timing data. In addition to identifying the ordering of function calls and their rather accurate execution durations, the extracted information is subsequently utilized to discover the *virtual* phases inherent in an application. To substantiate the correctness and potential of the developed extension, a real audio processing application, namely the *hArtes wfs*, was examined in detail.

Note.

The content of this chapter is based on the following article:

S. Arash Ostadzadeh, Marco Corina, Carlo Galuzzi, and Koen Bertels, ***tQuad* - Memory Bandwidth Usage Analysis**, *Proceedings of the 39th International Conference on Parallel Processing (ICPP'10)*, San Diego, USA, September 2010, pp. 217-226.

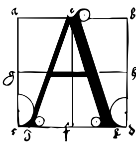
CHAPTER 5

Task Clustering: A Greedy Approach

"... tractability reaches far beyond the racetrack where computing competes for speed. It literally forces us to think differently. The agent of change is the ubiquitous Algorithm. Let's look over the horizon where its disruptive force beckons, shall we?" †

— Bernard Chazelle

The problem of partitioning an application is discussed in this chapter. The goal is to divide the functions (tasks) in an application into a predefined number of disjoint sets such that some objectives are satisfied. Besides the formulation of the general application partitioning problem, a heuristic approach is proposed as a solution to the problem, which focuses on the minimization of inter-cluster and the maximization of the intra-cluster[1] data communication as the primary objective. The QDU graph, provided by the QUAD-core tool, is utilized as the input data model to drive the partitioning. Experimental results are presented for a real application as well as for synthetic data in comparison with the optimal solution.



PPLICATION PARTITIONING[2] is a broad term referring to splitting an application into (smaller) parts which then run on different PEs, while preserving the semantics of the original application [17]. These PEs can be integrated into a single chip or be distributed among different processors, which are not necessarily in close proximity. The purpose of the partitioning can vary from speeding up the execution of the application to allowing better power management, to offering predictable and reliable behaviour, to ensuring the security of the application. Furthermore, the partitioning process itself is performed based on various criteria. It can be inclined more towards balanced workloads or based on the functionalities of different

† ***The Algorithm: Idiom of Modern Science***, Sept., 2006.

[1] *Cluster*, in this context, refers to a set of tasks that are grouped together.

[2] It is also referred to as ***code partitioning*** or ***source code partitioning*** in literature.

tasks constituting the application algorithm. Irrespective of its purpose and methodology, there should always be some mechanisms to ensure that partitioning is performed in the right direction and its main goal is fulfilled. As a result, thorough inspection of the application behavior, and more specifically the interactions among different application parts, is crucial in any efficient partitioning scheme.

The ability of parallel computing systems to execute more than one task simultaneously makes application partitioning a necessity to fully exploit the processing potential of these systems. Several critical performance factors, such as the degree of parallelism and the amount of overhead involved in parallel execution of an application, depend directly on the application partitioning scheme [82, 94, 105]. An effective partitioning scheme must take into account all these factors in order to provide performance improvement, which is mostly the primary goal of partitioning regarding parallel application execution.

In this chapter, we present a heuristic solution for the general application partitioning problem which, in most cases, is considered to be *NP-hard*, or in specific cases, *NP-complete* [11, 17, 109, 194]. Thus, an exhaustive search is likely to be impractical. Our proposed heuristic solution is based on the *greedy algorithm* [58], which yields locally optimal solutions that approximate a global optimality in a reasonable time. ***The main difference between our approach and other solutions is the fact that we utilize the QDU graph, provided by the Quad-core tool, as the input data model to direct the partitioning. This graph contains the actual data dependencies between the tasks, which is not available in other approaches. As discussed in detail in Section 5.4.1, the QDU graph proves to be superior to other input data models when the partitioning objective targets the minimization of the data communication between the different parts of an application.***

To describe the partitioning approach in practice, we present a case study of the Motion Joint Photographic Experts Group (MJPEG) encoding application. Additionally, to substantiate the quality of the solutions obtained from the proposed partitioning algorithm, we compare the results of synthetic graphs against the optimal solution determined by an exhaustive search. The remainder of this chapter is organized as follows. In Section 5.1, we give an overview of the general application partitioning. Section 5.2 presents a concise survey of the methods used for application partitioning. Before discussing our proposed solution, in Section 5.3, we first formulate the application partitioning problem in a very general and flexible manner. In Section 5.4, we propose a task clustering algorithm for the application partitioning, which is based on the greedy approach. The algorithm can be easily customized to accommodate different objectives. Section 5.5 presents a thorough complexity analysis of the proposed algorithm. To substantiate the potential of the proposed task clustering algorithm, a summary of results, using synthetic graphs, is presented in Section 5.6. Section 5.7 presents the case study of a real application partitioning and eventually, Section 5.8 concludes the chapter.

5.1 Application Partitioning

For quite a long time, high performance computing in large-scale systems was centered around the client/server model and also, to some extend, around parallel/concurrent

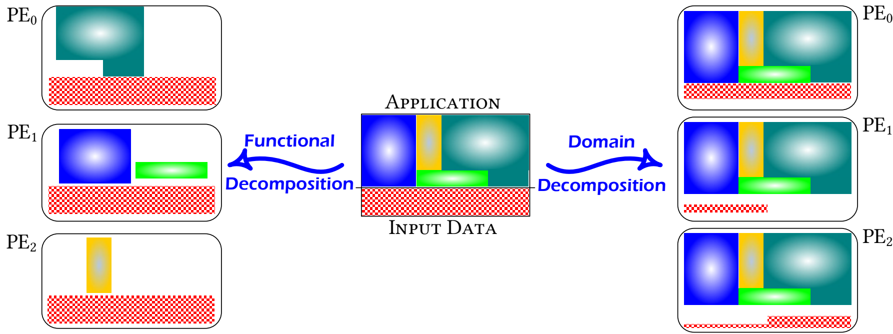


Figure 5.1: *A schematic diagram of domain vs. functional decomposition. The input data is the target of partitioning in domain decomposition, while functional decomposition targets the application itself.*

programming paradigms [232]. However, with today's advent of heterogenous parallel systems, where complex applications are executed on different PEs using various resources, distributed execution of an application is more ostensible. At the same time, creating such distributed applications is not easy indeed. Traditionally, most of the applications have been developed with the assumption that they would be executed on a single processor. As a result, in an ideal case, it is desirable to have an intelligent system that automatically partitions and distributes such applications while respecting critical settings and metrics of the target platform. Considering a more practical point of view, application developers need effective tools to help them in this respect, where a fully automated solution seems to be hard to attain.

One of the primary steps in developing a distributed application is to break the whole problem into discrete chunks of work that can be distributed among multiple PEs. This is generally known as *partitioning* or *decomposition*. The computational work can be partitioned among parallel PEs following two basic approaches:

Domain Decomposition - The data associated with a problem is divided into smaller compounds and, then, each PE actually performs the same task, but on a different portion of data.

Functional Decomposition - The focus is on the work itself to be performed rather than on the data. Therefore, the problem is decomposed according to the work that must be done. Each PE is assigned (most probably) a (different) task in order to perform a portion of the overall work. This approach is called *application partitioning* because it necessitates the computational work to be partitioned. Figure 5.1 depicts a schematic representation of the two approaches.

Several fundamental issues must be considered for application partitioning. Deciding what part of an application constitutes a potential executable unit (task) is not trivial nor straightforward. Mostly, an optimal solution would be application- and/or architecture-dependent. Conventionally, the granularity of the application partitioning is in accordance with the granularity level of parallelism on the target architecture. The granularity level can range from fine-grained instructions to different iterations of loops,

to coarse-grained functions. There is always a trade off between the granularity level of the partitioning, flexibility, and performance of the application. Fine grained partitioning can be in favor of flexibility, however, the performance gain in current heterogeneous multiprocessors can be negatively influenced by dividing the application into too many small chunks. Commonly, the overhead associated with the coordination (communication and synchronization) of too many small partitions surpasses the performance gain offered by the distributed execution of an application.

In addition to the granularity of the partitioning, it is also very important to decide on the placement of different tasks on available PEs. There are several critical factors that can severely affect task allocation decisions, some of which can only be dealt with at runtime and/or by having some dynamic information about the application's behavior a priori. Most of the existing partitioning mechanisms are restricted in the sense that they statically decide on the allocation of tasks, with no (or very limited) dynamic information available beforehand. These mechanisms are mostly dependent on user interactions than extracted information from the application itself.

Generally, an application partitioning scheme is characterized based on the following factors:

Static vs. Dynamic Allocation - **Static** application partitioning refers to the separation of application workload at design time. Client/Server applications are typical examples of this class of partitioning scheme. Static partitioning leads to inflexibility that refrains from task migration to where they could have been executed most efficiently. On the other hand, in **dynamic** application partitioning, the workload of the PEs may vary during the execution of the application. In such scheme, based on some predefined criteria, tasks may be offloaded to other (more suitable) PEs at runtime. Consequently, dynamic partitioning becomes more challenging compared to static partitioning. Similarly, it cannot benefit from a better programming language support like static partitioning [123].

Since this factor is concerned with the allocation of tasks to PEs (mapping), it is sometimes referred to as ***Offline*** (static) or ***Online*** (dynamic) task partitioning or *task mapping* in literature [117]. Offline task mapping can be specified in the form of a configuration file or as annotations within the source code of the application. Online task mapping is postponed until the execution time; mapping decisions are normally made by some kind of runtime scheduler or by the operating system.

Explicit vs. Implicit Implementation - Application partitioning can be implemented utilizing different mechanisms which, in turn, exhibit different levels of abstraction. On one hand, in an **explicit mechanism**, partitioning is expressed within the programming language itself. It prohibits the provision of a full abstraction level, since the programmer is somehow aware of the complexities of the coordination between different PEs. Explicit mechanisms can partially achieve transparency by using traditional object-oriented programming techniques, such as inheritance and the separation of interface and implementation. When applied consequently, this means that task distribution is invisible to the application programmer, although behind the scenes, it may be expressed using the same language. Moreover, explicit mechanisms can allow varying degrees of task distribution transparency, depending on how the abstraction is actually realized. On

the other hand, instead of utilizing the programming language for partitioning, it can be handled implicitly by the execution environment. Consequently, **implicit mechanisms** have the advantage that they achieve complete transparency from the programmer's point of view. The problem, however, is the lack of efficiency because the abstraction is made below the programming language. To some extent, utilizing higher degrees of automatic optimizations may compensate for this inefficiency [183].

Coarse vs. Fine Granularity - The granularity level of a partitioning algorithm is an important factor that substantially affects the performance of the partitioning algorithm itself as well as the application for which the partitioning is performed. Conventionally, the granularity level is decided by the estimated size of each potential task that is to be mapped on the PEs of a target system. In general, tasks which consist of loops, blocks or whole functions of code are assumed to be coarse-grained. A coarser granularity can be advantageous with respect to reduced partitioning effort and lower communication overhead during the execution of the application. The partitioning algorithm is characterized as operating on fine granularity when the tasks comprising the partitions of the application consist of several instructions (or even individual instructions) of source code. Fine granularity results in a huge solution space, which makes the partitioning difficult and, mostly, inefficient. An increase in the size of the solution search space increases the execution time of the partitioning algorithm as well as the number of design choices.

When an application is implemented using an object-oriented approach rather than a procedural one, partitioning based on *class* or *object* granularity is common. Based on the assumption that all the objects instantiated from the same class exhibit similar behavior and should be mapped on the same PE, partitioning is performed on an individual class or on a group of closely interacting classes. On the other hand, partitioning based on objects comes from the fact that objects from the same class may exhibit different behaviors when interacting with different objects. Moreover, assigning all objects of a class to a PEs can be restricting. It should be noted that only in object-based partitioning, the source code is not actually divided, instead, the aim is on the distribution of objects across different PEs.

Source Code vs. Runtime Analysis - Partitioning is mainly performed based on some behavioral analysis of an application. As an example, a dependency relationship analysis can direct the identification of closely interacting components and, as a result, they can be grouped together and placed in the same part. The analysis can be carried out at the application **source code** level, which is conventionally very fast; it can be performed on some kind of intermediate representation of the source code, such as Sun Microsystem's Java bytecode; it can even be performed on the binary code of the application, although it is not common. Alternatively, **runtime** analysis can be used to study the application behavior, including subsequent profile data processing.

Conventionally, if application analysis is performed without actually executing the application, it is called **SCA**, in contrast to the **DCA**, where it is dependent

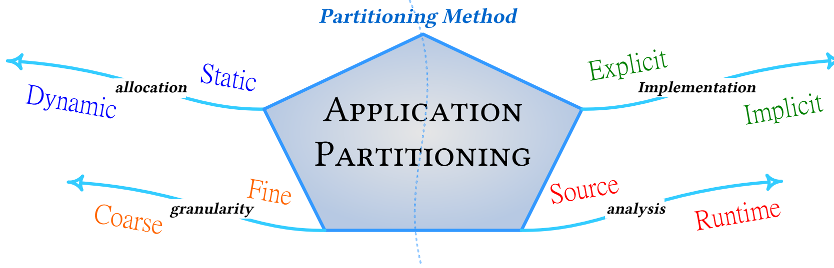


Figure 5.2: *Different application partitioning factors. Application partitioning is commonly classified by the method utilized for the partitioning. Furthermore, partitioning methods are distinguished by different factors which characterize the partitioning from various aspects. These aspects include: the allocation of parts, the implementation of the partitioning, the granularity level of the partitioning, and the analysis of the application to partition.*

on the application execution on a real or virtual processor. For DCA to be effective, the target application must be executed with sufficient and representative input data sets to disclose a true and thorough behavior of the application. As a result, it is difficult to practically guarantee the comprehensiveness and exactness of the application's behavior. It should be noted that SCA formally refers to the analysis performed by (semi)automated tools, as opposed to human analysis of code, which is conventionally addressed with the terms: *program understanding*, *program comprehension* or *code review*.

Figure 5.2 summaries the important factors which somehow categorize the partitioning mechanisms. Apart from the four factors discussed here, the principal parameter, which is widely used to classify application partitioning, is the utilized partitioning method. Different partitioning methods and their implications are investigated in detail in the following section.

5.2 Partitioning Methods

Programming multiprocessors has always been a very difficult and complicated problem. High-level programming abstractions for these systems are almost non-existent, leaving the programmers with the tedious burden of explicitly programming them using platform-dependent, low-level abstractions. Nonetheless, this process is error-prone and forces the programmer to deal with many details outside of the application domain. More precisely, the programmer has to deal with all parallel processing tasks required to program a multiprocessor system. These tasks include explicit partitioning of the program into parallel tasks, scheduling these tasks on the PEs, synchronization, explicit distribution of data among the PEs, and proper handling of data transfers [17].

Much effort is being done to automate the parallel processing tasks mentioned above by the compiler. Thus, the user would be relieved of explicitly handling the architectural details of the system. *Code partitioning* and *scheduling* are assumed to be two main phases of a compiler for parallel architectures [176, 177]. However, general code par-

titioning has failed to attract researchers in comparison with the scheduling problem [17, 36, 62, 120].

The application to be partitioned is generally given in the form of a task-/data-flow graph or a set of directed/undirected cyclic/acyclic graphs, which represent some kind of dependencies/relations between the tasks/data in the application. In a graph $G = (N, E)$, the set N of nodes represents the computational tasks (at fine or coarse level of granularity) and the set E of arcs represents the data and/or control dependencies between the nodes. Different cost parameters can be associated with each node i, such as execution time for HW and/or SW implementations, area required for hardware implementation, etc. Parameters associated with each arc (i, j) may include the amount of data transferred from node i to node j, the communication time calculated for node i and j, when mapped into the same/different PE(s), etc. Design constraints may also be defined, such as latency, the capacity of the hardware resources, etc. The partitioning problem is to find a mapping (assignment) of nodes to PEs based on one or more objective functions, subject to the defined constraints. In case the partitioning involves scheduling as well, the starting time of each node should also be determined.

Ordinarily, the target architecture or execution environment must be determined. In some approaches, the partitioning problem is solved independently from the target architecture by estimating the costs. If the target architecture is specified, appropriate estimates for design parameters can be determined in advance. At a high level, regardless of specific details, the code partitioning problem falls into two classes. In one class, partitioning is addressed for a specific application, which is assumed to be more efficient (**manual** or **semi-automatic** partitioning). In the other class, a general solution (**automatic** partitioning) is sought. The latter is proven to be hard to attain or the solution is far too simple and, hence, inefficient [15, 17, 29, 39, 40]. Some early works addressing the code partitioning problem for multiprocessor systems can be found in [15–17, 84, 176–179].

Apart from the general application partitioning, a specific class of this problem, called **harware/software (HW/SW) partitioning** (see Section 6.1), has gained extensive attention recently. HW/SW partitioning is basically a *bipartitioning* problem, which divides the application into two disjoint HW and SW parts. The requirements of today's computing systems necessitate the realization of both hardware and software components. In this context, hardware means application-specific hardware units, i.e. hardware designed and implemented specifically for a given system, whereas software means a program running on a general-purpose hardware unit, such as a microprocessor. Traditionally, HW/SW partitioning was performed manually. However, as the systems to design have become more and more complex, this method has become infeasible, and many research efforts have been undertaken to automate partitioning as much as possible. Most presented methods are *heuristic*, but there also exists *non-heuristic*[3] algorithms [138].

The majority of the previously proposed partitioning algorithms is heuristic. This is due to the fact that partitioning is a hard problem and, hence, exact algorithms tend to be quite slow for moderate to large problem sizes. More specifically, as mentioned before, most formulations of the partitioning problem are NP-hard [11, 17, 109, 194], and the

[3] Non-heuristic methods are also known as *exact*, *deterministic* or *optimal* in literature.

exact algorithms for them have exponential runtimes. Heuristic methods are generally classified as hardware-oriented or software-oriented. In a hardware-oriented approach, a complete hardware solution is initially realized and, subsequently, parts of the application are iteratively moved to the software as long as the performance constraints are still met [93, 154]. The software-oriented approach starts with a software implementation, moving pieces to hardware for performance improvement until the time constraint is satisfied [27, 207, 208]. In many research works, general-purpose heuristic methods are applied to solve the partitioning problem. In particular, *Genetic Algorithm (GA)* has been extensively used [12, 64, 65, 143, 170, 185, 195], as well as *Simulated Annealing (SA)* [27, 73, 74, 76, 95, 132]. Other, less popular heuristic algorithms in this group are *tabu search* [73, 74, 219], *greedy algorithm* [51, 91], and *Particle Swarm Optimization (PSO)* [34]. In a comparative study [219], tabu search is reported to outperform GA and SA. Some researchers used *custom* heuristics. This includes the Global Criticality/Local Phase (GCLP) algorithm [107, 108], expert systems [129, 131], as well as other heuristics [11, 93, 224]. There are also some *well-established* classes of heuristic methods that are usually applied to partitioning problems. *Hierarchical Clustering* is one such class [1, 28, 130, 202, 206, 207, 231]. The other class of partitioning-related heuristic method is known as *Kernighan-Lin (KL)* [111]. The method has been subsequently improved in [61, 78, 174, 201, 209][4]. [131] presents a comprehensive comparison between their proposed expert system and three other techniques, namely hierarchical clustering, SA, and KL heuristic. It has been concluded that the expert system achieves the best partitioning results, both qualitatively and quantitatively.

Although heuristic partitioning methods are typically very fast and produce near-optimal or even optimal results for small problem sizes, their effectiveness (in terms of the quality of the found solution) degrades drastically as the size of the problem increases [12]. This is because, to quickly find a solution, these heuristic methods evaluate only a small fraction of the whole search space. As the size of the problem increases, the search space grows exponentially[5], which means that the ratio of evaluated points of the search space must decrease rapidly, leading to worse results. Moreover, if the to-be-partitioned application is large in scope and tight system constraints (performance, cost, etc.) must be met (which is usually the case), then chances are high that a heuristic partitioner will find no valid solution. What is even worse, the designer will not know if this is due to the weak performance of the partitioning algorithm or because there is no valid partition at all. This shows that in practice, it makes sense to go for an optimal solution whenever possible [138].

The above issues can be addressed by using exact partitioning methods. These methods include Branch and Bound (BB) [35, 52, 138], Dynamic Programming [135, 156], and Integer Linear Programming (ILP) [153, 154, 169]. Due to large runtimes, exact methods are usually used either for small problem sizes [35, 135, 156], or combined with heuristic methods, which can sacrifice their optimality [153, 154]. This also makes it difficult to verify their scalability for larger problem sizes. Appropriate scalability up to several hundreds components is also reported in exact methods [138]. Arató et al. [11] investigate the algorithmic aspect of the partitioning problem and evaluates the capability

[4] The Kernighan-Lin partitioning heuristic, also known as *min-cut* or *group migration*, has been successfully applied in circuit partitioning for a long time, and it is somehow adopted as a standard in this respect. The extended version of the algorithm proposed in [78] is known as Kernighan-Lin/Fiduccia-Matheyes (KLFM).

[5] There are 2^n different ways to partition n components.

of deterministic partitioning for obtaining high quality solutions with less search time. They claim the superiority of the algorithmic approach compared to heuristic methods, such as KL and GA. In a following study, Tahaee et al. [194] pursues the theoretical approach to further elaborate the algorithmic properties of the partitioning problem. The authors propose a method with polynomial complexity to find the global optimum of an NP-hard model partitioning problem for 75% of occurrences under some practical conditions. The problem modeling and categorization correspond to those used in [11, 12, 137]. For a more detailed survey of primary HW/SW partitioning methods refer to [69, 70, 137].

Additionally, the proposed methods vary significantly regarding the granularity of partitioning. There are works on fine granularity, where graph nodes represent single instructions or short sequences of instructions [28, 102, 180], as well as basic blocks [104, 113, 162]. Coarse granularity, where nodes represent functions, is also studied [3, 93, 154, 207]. In addition, there are studies with flexible granularity, where nodes can represent any of the above [11, 95, 202]. Concerning the scope of partitioning, further distinction can be made. In particular, many studies consider scheduling as part of partitioning [51, 65, 107, 131, 143, 154], whereas others do not [73, 91, 135, 156, 202, 209]. Some even include the problem of assigning communication events to links between hardware and/or software units [65, 143]. In some studies, the target architecture is assumed to consist of just a SW and a HW PE [11, 73, 91, 93, 95, 131, 135, 143, 156, 180, 185, 187, 209], while on the contrary, some do not impose such restriction [51, 65, 154].

Table 5.1 presents a summary of several partitioning methods, which have been proposed so far. As discussed earlier, the majority of research work is based on heuristic methods. In addition, the simple case of application partitioning, which assumes only a SW PE and a HW PE as the target architecture, is mostly investigated. ***Considering the data communication analysis among multiple PEs makes the general application partitioning even more intractable. As a results, mainly the approaches which address general partitioning are either restricted to theoretical formulation of the problem or do not consider such communication in a practical environment.***

In the following sections, we present a formulation of the general application partitioning problem and, subsequently, describe a practical heuristic method to solve it using the QDU graph introduced in Chapter 3.

5.3 Problem Formulation

As stated in Section 5.2, we use the (directed weighted) QDU graph as the input for application partitioning. In its most general form, the *graph partitioning problem* deals with the division of a graph's vertices into a predefined number of subsets in such a way that: 1) the number of vertices per subset is equal, and 2) the number of edges straddling the subsets is minimized. Graph partitioning has important applications in different fields of Computer Science, including image processing, solving linear systems, VLSI circuit layout, and distributing workloads for parallel computation [50]. Graph partitioning is known to be NP-hard [83], hence, all known optimization algorithms utilizing graph partitions merely return approximations to the optimal solution for large problem sizes. In spite of this theoretical limitation, numerous algorithms for graph partitioning have

Table 5.1: *An overview of various application partitioning approaches previously appeared in literature.*

	Ref.	Method	Grain	Input[a]	Target	Lang. Spec.[b]	Sch.[c]
Exact Methods	[35]	BB	ins.	data lists	ASIP (PEAS-I)	C	✓
	[169]	MILP	n.s.	TDFG	multiprocessor[d]	n.s.	✓
	[104, 156]	dynamic programming	bb[e]	data lists	S SW + M HW[f]	C/C++	n.s.
	[180]	algebraic partitioning	op.[g]	n/a	S SW + S HW	occam	✓
	[52]	BB	task	task graph	S SW + S HW	C, synthetic	✓
	[138]	ILP, BB, GA, min-cut, hierarchical clustering	task	communication graph	S SW + S HW	C, synthetic	×
	[153, 154]	MILP, custom heuristic, GA	func.	internal syntax graph	multiprocessor	C, DFL, VHDL	✓
	[225]	dynamic programming, tabu search	n/a	task graph	S SW + S HW	synthetic	×
	[11]	min-cut, custom heuristic	flex.[h]	communication graph	S SW + S HW	n/a	×
Heuristic Methods	[91, 113, 135]	greedy algorithm, dynamic programming	bb	CDFG	S SW + S HW	C, VHDL	✓
	[201, 209]	KL, greedy algorithm, SA, hierarchical clustering	process	SLIF access graph	S SW + S HW	VHDL, SpechCharts	n.s.
	[176–179]	custom heuristic	task	IF1 (DFG)	multiprocessor	SISAL	✓
	[107, 108]	GCLP	task	CDFG	S SW + M HW	C, Silage/VHDL	✓
	[129, 131]	expert system vs. SA, hierarchical clustering, KL	task	EFG	S SW + M HW	HLL (n.s.)	✓
	[51]	greedy algorithm	task	HCDFG	multiprocessor	n.s.	✓
	[34]	PSO vs. ILP, GA, ACO	task	task graph	S SW + S HW	n.s.	×
	[12]	GA, ILP	flex.	task graph	S SW + S HW	n.s.	×
	[93]	custom heuristic, KL	ins.	DFG	S SW + S HW	hardwareC	✓
	[132]	SA, KL	flex.	exec. flow graph	S SW + M HW	HLL (n.s.)	✓
	[65]	GA	flex.	task graph	multiprocessor	synthetic	✓
	[219]	GA vs. tabu search vs. SA	task	task graph	S SW + S HW	synthetic	✓
	[187]	runtime profiling	loop	n/a	S SW + S HW	n/a (binary code)	✓
	[105]	greedy algorithm, custom heuristic	flex.	communication graph	S SW + S HW	n/a	×
	[27]	SA vs. KLFM	task	call graph	S SW + S HW	synthetic	×
	[195]	GA	task	task graph	S SW + S HW	synthetic	×
	[73, 74]	SA, tabu search	process	process graph	S SW + S HW	VHDL	×
	[202]	hierarchical clustering	dyn.[i]	call graph	multiprocessor	VHDL	n.s.
	[206, 207]	hierarchical clustering	func.	SLIF access graph	multiprocessor	VHDL	n.s.
	[3]	SA	bb	behavior desc. (SSA)	S SW + CCUs	Verilog HDL	✓
	[208]	meta-algorithm (BCS), SA	flex.	n.s.	S SW + S HW	VHDL	×
	[143]	GA	flex.	task graph	S SW + S HW	synthetic	✓
	[76]	SA	bb	ESG	S SW + S HW	C	✓
	[170]	GA	dyn.	HTG	S SW + S HW	n.s.	n.s.
	[130]	hierarchical clustering	flex.	EFG	S SW + M HW	n.s.	✓
	[185]	GA	task	task graph	S SW + S HW	C, VHDL	✓
	[28, 231]	hierarchical clustering, min k-way cut	ins.	Unity HDL, occam	S SW + M HW	C, Unity, occam	✓
	[1]	hierarchical clustering	task	attraction graph	dist. het. RT[j]	n.s.	✓
	This Thesis	**greedy algorithm, hierarchical clustering**	**task**	**QDU graph**	**multiprocessor**	**n/a (binary code)**	×

[a] The data model used to input information for the partitioning algorithm.
[b] Programming language specification utilized as input and/or output for the partitioner.
[c] Indicates whether or not the scheduling problem is also investigated along with partitioning.
[d] A general system with multiple PEs, no restrictions.
[e] bb refers to *basic block*, for which the granularity is not well-defined. It is also referred to as *basic scheduling block* in some references.
[f] A system consisting of either *S (single)* or *M (Multiple)* software/hardware processors.
[g] Operations, the finest granularity level.
[h] *Flexible* means the method is general enough to be applied with different granularity levels.
[i] The granularity for the partitioning is dynamically determined by the algorithm.
[j] Distributed heterogeneous real-time systems

been developed during the past three decades that can generate high-quality partitions in very little time. [50] presents an overview of different graph partitioning algorithms, focusing on workload distribution in parallel systems. Since the QDU graph is weighted and the extracted data communication information is supposed to drive the partitioning process, the general graph partitioning scheme seems to be inadequate for this purpose. Other variants, namely *weighted graph partitioning* and *δ-partitioning*, are more pertinent as they can be customized based on weighting values.

Weighted graph partitioning allows weights to be associated with both the vertices and the edges of an input graph. A good partition is determined by the rough equality of the total weight of vertices in each subset and the minimization of the total weight of cut-edges. In the context of task graphs, node weights can be used to encode different computational costs (execution times) of tasks. Similarly, edge weights can signify the volume of data communication required between two tasks. In δ-partitioning, imbal-

ance in subset weights is somewhat tolerated in the hope of significantly reducing the number of cut-edges. In this partitioning, a fixed tolerance value δ is supplied as input, in contrast to *skewed partitioning*, where user supplied weights are associated with each subset to specify a desired imbalance in the partition. These generalizations add more flexibility for the characteristics of a heterogeneous system to have some bearing on the partitioning. For example, if a machine's inter-processor communication overhead is sufficiently high, it might be worthwhile to tolerate some degree of load imbalance in order to drastically reduce the communication volume. This tradeoff can be described using δ-partitioning. As another example, in a heterogeneous system, the relative computational performance of PEs can be described by specifying appropriate target weights in skewed partitioning. It goes without saying that the closer one gets to modeling the behavior of actual machines, the more complicated it becomes. Further generalizations of the graph partitioning problem can also be assumed, e.g., minimizing the number of PEs with which each PE communicates, or modeling non-uniform communication costs that occur in clusters of multiprocessors [50].

The model that is defined and used in this work is a further extension of the application partitioning with the focus on the minimization of data communication and task merging. More specific models, such as δ-partitioning or the usual HW/SW partitioning discussed in Section 5.2 and Section 6.1, can be derived from our general model. The model proposed here *only* deals with application partitioning. It does not include task scheduling/mapping, data distribution or other tasks, which are common for application development in heterogeneous environments. We assume that decoupling the specific tasks of the system design allows the development of more effective solutions. Certainly, this decoupling necessitates some simplifications in the problem formulation and may result in precision loss. By utilizing the profiling information extracted by Q^2, the partitioning algorithm has more or less accurate estimations of several cost metrics. However, missing data, such as architectural properties of the communication system, can affect the quality of the output results. Conversely, simplification drastically reduces the complexity of the partitioning problem, and thus, a broader part of the search space can be examined. In this way, even better results may be achieved than by considering all the modeling details and aspects of a design, but scanning only a small part of the huge search space.

In our model, the target application is described as an annotated QDU graph, whose nodes represent the functions (tasks) of the application to be mapped onto PEs. The edges represent quantitative information of the actual data communications (transfers) between the functions. We have to stress that in the QDU graph, edges do not convey the usual meaning as depicted in common task or communication graphs seen in previous work [105, 137, 194]. Furthermore, the acyclicity restriction and even the graph being directed are relaxed in our model[6]. These, in turn, contribute to the advantages of our model, because it is more representative of the real world scenarios.

Each node is assigned a cost vector called *execution cost* that represents the cost values of executing a certain task on each of the available PEs. The cost is assumed to be distinctive for different PEs in the support of heterogeneity of the target architecture. However, for the sake of simplicity, one may decide that only one or several value(s)

[6] Though the original QDU graph is directed, this is not a requirement in our partitioning model and will be clarified later in the text.

suffice for each node. No explicit restriction is imposed on the semantics of this cost. It may represent any cost metric, such as the usual execution time or the percentage of the execution contribution, the size of the utilized hardware resources in the case of hardware PEs, or power consumption. It is also possible to have a combination of several metrics. However, it should be clear for the designer how various metrics should be integrated to determine the value asserting the execution cost of each task. Additionally, the designer has to know how the cost of a part should be calculated with regard to the total nodes in that part. In the most simple way, a linear summation of the values combined with weight factors would do fine. While these assumptions are restrictive, they generally pose no critical problem that substantially degrade the results. Most important cost metrics fall within the simple summation strategy. For example, power consumption is usually assumed to be additive. The same holds, to some extent, for the hardware area resource usage and the execution time. Although the QDU graph is based on the coarse function granularity, this is not a restriction within our partitioning model. Finer grained levels, such as basic blocks, can also be used as long as the input graph is adapted accordingly.

The edges in the QDU graph represent the actual data flow between functions, and hence, are directed. However, this is not a requirement, since graph edges are utilized with a different role in our partitioning model. In case a pair of communicating functions are mapped to distinctive PEs, their data communication incurs a communication penalty, the value of which is determined, for each edge, as a *communication cost*. Depending on the nature of each PE and architectural properties (communication means, memory hierarchies, etc.), the communication cost can be different in different contexts (heterogeneous PEs). In the general case, a vector of communication costs (or relevant parameters, based on which the actual cost is computed) is maintained for each edge. For homogeneous architectures, this can be reduced to just one value. The communication cost is conventionally assumed to be independent of the communication direction. Otherwise, one can decide to use the directed QDU graph to compensate for different communication costs based on the data flow direction. Primarily, the communication cost depends on the volume of the data transferred between the two functions, which is available in the QDU graph. For simplicity, if data communication stays internal within a PE, the communication cost is neglected. This is due to the fact that, on most architectures, inter-context communication is orders of magnitude more costly compared to intra-context one.

Similar to the execution cost mentioned above, no explicit restriction is imposed on the semantics of the communication cost. The value of this cost can be determined based on any parameter according to the architectural properties and design objectives. Furthermore, the summation strategy to estimate the total communication cost of a part shall remain applicable. The edges that cross the boundary of a part are called *cut-edges*, and these are the ones which contribute to the communication cost of a PE. On the other hand, the edges which remain internal inside a part, i.e. the edges which represent the data communication among the functions mapped into a single PE, represent the *coupling degree* inside a cluster. Though not used for the estimation of the communication costs inside clusters, our model still considers it as the main critical factor to hint at merging functions in one cluster. In this respect, our partitioning model extends the prior definitions by introducing the concept of merging tightly coupled functions, which

are mapped onto a PE. A realization of this practice is investigated in details in Chapter 6. Converting the directed QDU graph into an undirected one is easily accomplished by replacing each mutual (a pair of inward/outward) edges between two nodes by a single edge, whose value equals the sum of the connecting edges. Note that the QDU graph may also contain self-loops, which, at this time, have no role in our partitioning model. These self-loops are simply ignored in the analysis.

We formalize the k-way weighted graph partitioning as follows. An undirected weighted graph $G=(V,E)$ is given as input, where the vertex set $V=\{v_1,v_2,\cdots,v_n\}$ represents the set of n functions in an application, and the edge set $E=\{e_{ij}=(v_i,v_j)\colon 1\le i,j\le n; i\neq j\}$ denotes the actual data communication between pairs of cooperating functions. For each node v_i, a set of *Execution Costs* $EC(v_i)$, or simply $EC_i=\{ec_i^1,ec_i^2,\cdots,ec_i^k\}$, $EC\colon V\mapsto(\mathbb{R}_{\ge 0},\mathbb{R}_{\ge 0},\cdots,\mathbb{R}_{\ge 0})$, is defined, which specifies the execution cost of node v_i when mapped to either of the k available parts[7]. The value of the actual execution cost is determined by picking up one of the k available values in EC_i, based on the selected part.

The *Communication Cost (CC)* for each edge e_{ij} is defined as the set of non-negative values $CC(e_{ij})$, or simply $CC_{ij}=\{cc_{ij}^1,cc_{ij}^2,\cdots,cc_{ij}^m\}$, $CC\colon E\mapsto(\mathbb{R}_{\ge 0},\mathbb{R}_{\ge 0},\cdots,\mathbb{R}_{\ge 0})$, when v_i and v_j are mapped into distinct clusters. The value of the actual communication cost is determined by picking up one of the $m=\frac{1}{2}k(k-1)$ available values in CC_{ij}, based on the selected clusters into which v_i and v_j fall. The value of m corresponds to all the different possible ways that an edge can connect two clusters from a set of k clusters. In case the graph is directed[8], the total number of values in CC_{ij} would be doubled ($m=k^2-k$) and the actual communication cost is determined based on the source and destination clusters of the edge. Similarly, the *Coupling Degree (CD)* for each edge e_{ij} is defined as the set of non-negative values $CD(e_{ij})$, or simply $CD_{ij}=\{cd_{ij}^1,cd_{ij}^2,\cdots,cd_{ij}^k\}, CD\colon E\mapsto(\mathbb{R}_{\ge 0},\mathbb{R}_{\ge 0},\cdots,\mathbb{R}_{\ge 0})$, when v_i and v_j are mapped into the same cluster. The value of the actual coupling degree is determined by picking up one of the k available values in CD_{ij}, based on the selected cluster to which v_i and v_j belong. In case the graph is directed and there exists a pair of inward/outward edges between v_i and v_j, we simply assume that the coupling degree value is the aggregation of the two values of the edges.

P is defined to be a *k-way partition* of an application if it is a partition of $V: P=\{C_1,C_2,\cdots,C_k\}$, where $C_i\neq\emptyset$; $\bigcup P=V$; $C_i\cap C_j=\emptyset$; $1\le i,j\le k; i\neq j$. Note that the clusters cannot be empty. The set of edges within cluster C_i is defined as $E_{C_i}=\{(v_x,v_y)\colon v_x,v_y\in C_i; 1\le x,y\le n; 1\le i\le k\}$. The set of cut-edges $CUT_P\subset E$, with respect to partition P, is defined as:

$$\begin{aligned}CUT_P &= E\setminus\bigcup E_{C_i}\,,\ 1\le i\le k\\ &=\{(v_i,v_j)\colon v_i\in C_x, v_j\in C_y; 1\le i,j\le n; 1\le x,y\le k; x\neq y\},\end{aligned}$$

and the execution cost of each cluster C_i is defined as:

$$EC_{C_i}=\sum_{v_j\in C_i} ec_j,\quad 1\le i\le k; 1\le j\le n.$$

[7] Each *part* represents a subset of V that includes the set of functions to be mapped onto one PE. When there is no confusion, we use the terms "cluster" and "part" interchangeably in the text.

[8] As an example, when different communication channels for sending and receiving data exist.

For each cluster C_i, $1 \leq i \leq k$, we define the *Balancing Penalty (BP)* to be the distance between EC_{C_i} (the execution cost of cluster C_i) and the average execution cost of the clusters in P, as follows:

$$BP_{C_i} = \left| \frac{\sum_{j=1}^{k} EC_{C_j}}{k} - EC_{C_i} \right|, \quad 1 \leq i \leq k,$$

and the balancing penalty of partition P is simply calculated as $BP_p = \sum_{i=1}^{k} BP_{C_i}$. Similarly, the coupling degree of cluster C_i is defined as:

$$CD_{C_i} = \sum\nolimits_{(v_x,v_y)\in E_{C_i}} cd_{xy}\,,\ 1 \leq x, y \leq n\,;\ 1 \leq i \leq k\,,$$

and the coupling degree of partition P is defined as the following:

$$CD_p = \sum_{i=1}^{k} CD_{C_i} = \sum\nolimits_{(v_x,v_y)\in E\setminus CUT_P} cd_{xy}\,,\ 1 \leq x, y \leq n\,.$$

The communication cost of partition P is defined as:

$$CC_p = \sum\nolimits_{(v_x,v_y)\in CUT_P} cc_{xy}\,,\ 1 \leq x, y \leq n\,.$$

Thus, in our model, an application partition is primarily characterized by three metrics: 1) the balancing penalty, 2) the communication cost, and 3) the coupling degree. These metrics are defined in an abstract way in order to be general enough for various model customizations. Furthermore, they can be inherently conflicting. As an example, decreasing the data communication between PEs, may contribute towards unbalanced workloads and vice versa. Based on the combination of these metrics, there are several ways to formulate a well-defined optimization or decision problem. Furthermore, different constraints on these metrics and possibly other parameters, such as resource utilization, can be formulated in the problem definition. In this work, we only formulate the most straightforward, yet comprehensive, partitioning problem as follows.

Problem AP[9]. Given an application and its undirected weighted QDU graph G as an input, along with the cost functions BP, CC, CD, and the constants α, β, $\gamma \geq 0$, find the *k-way partition P* of the application that minimizes $TC_P = \alpha\, BP + \beta\, CC + \gamma\, CD^{-1}$.

This basic model captures the main important properties of general application partitioning in a heterogeneous environment. Its compactness helps designers to understand and focus on the essence of the partitioning problem. A possible concise interpretation for **AP** would be the following. Assume that the communication cost between PEs accounts for the extra execution delay imposed on the application execution. We would like to know how to partition the application such that each PE gets an equal share of the total computational workload, while related (tightly-coupled) functions are mapped onto the same PE. Mapping related function onto the same PE implies that there would be ample opportunities for local code optimizations. That is the reason why we are interested in such a solution to the partitioning problem.

[9] **AP** stands for **Application Partitioning**.

It should be noted that the aim of the partitioning is to minimize the weighted sum of the cost metrics. These weights (α, β, and γ) are specified by the designer, and only define the relative significance of the three metrics. This means that the solution of the problem **AP** is dependent on the ratio of the weighting factors, not on their absolute values. As mentioned previously, many different versions of application partitioning can be derived from our general model and problem definition. For instance, if the execution cost of the functions mapped onto a PE captures the amount of required memory, one may decide to solve the following constrained optimization problem: among all the possible partitions of an application, find the one that minimizes the communication cost, while meeting the constraint that the total required memory size for each PE is less than or equal to a predefined value.

All the previously examined HW/SW partitioning problems can also be modeled with our definition. As an example, suppose that the number of clusters is two ($k = 2$), where the first cluster is labeled *Hardware PE*, and the second cluster, *Software PE*. We define the execution cost of a function to be an aggregation of two different cost values, one for *software* (EC_S) and one for *hardware* (EC_H). The total execution cost is simply defined by $EC = \zeta\, EC_S + (1 - \zeta)EC_H$. ζ is the parameter, which switches between 0 and 1, depending on the function being mapped onto the hardware or the software cluster, respectively. One may assume that hardware cost (EC_H) denotes the occupied area on the chip, and software cost (EC_S) specifies code size. Considering that γ is zero or its value is very small compared to α and β (thus, it can simply be ignored), we can easily model the HW/SW partitioning problem defined in [137].

5.4 Multi-Objective Task Clustering

In the context of mapping applications onto systems with multiple (heterogeneous) PEs, one key problem that severely limits the performance gain of the whole system is the size of distributed tasks. This problem emerges from the fact that, when distributing tasks, we have to take care of the extra overheads (communication, synchronization, etc.), regardless of the properties of the underlying architecture. To tackle this problem, coarse grained partitioning of an application is generally considered to improve the performance of an implementation by decreasing the costs involved, which can also result in increased parallelism [184]. Grouping tightly-coupled (related) functions is a natural step to contribute to the coarseness of the partitioning granularity. Furthermore, after grouping, the size of the task graph (problem size) is reduced, and this can be beneficial in subsequent stages, such as DSE for mapping and scheduling, and the synthesis process in reconfigurable systems.

In this section, we present our task clustering algorithm which groups functions into clusters based on the dynamic profiling information provided by *Quad*. It creates a partition of an application that is assumed to satisfy an objective function defined by the designer. Task clustering is used as the primary technique to contribute to the coarseness of the granularity of tasks in our application partitioning solution.

The main motivation for task clustering is to suppress undesirable partitions, which turn to be inappropriate, unnecessary, and non-optimal in the huge DSE of the application mapping onto a heterogeneous multiprocessor environment. The flexible multi-

objective clustering algorithm presented in this work allows us to examine various criteria in order to come up with near-optimal (or even optimal) solutions suited for different scenarios. Since system- and architectural-specific details are peeled off from the core of our partitioning solution, the proposed approach is considered general enough to be applicable to different heterogeneous multiprocessor systems. Furthermore, the outcome of our clustering algorithm can provide hints to developers for customized design optimizations and revisions in the application, such as possible coarse grained parallelism detection.

5.4.1 Input Data Model

In our partitioning model, data communication information is provided at function level, which implies coarse granularity, yet delivering sufficient detailed overview of quantitative data dependencies within an application. Since the clustering algorithm generally aims to form task groups as primary *building blocks* for application development and execution in heterogeneous multiprocessor environments, we only need to go through the analysis at the coarse level. Therefore, we do not consider instructions or small basic blocks within functions as the unit of partitioning. However, in case there is a requirement to analyze the application at finer granularity, the proposed model still proves to be applicable to a great extent. As a quick and simple hack for the basic block or loop level partitioning, one may opt to extract critical regions in the source code of the application to create new (dummy) functions. Subsequently, the extracted profiling information based on this modified version of the application can be fed as the input to the clustering algorithm. Tackling the problem at coarse level benefits from the fact that it is more likely the partitioning algorithm ends up with loosely-coupled parts.

A call graph reveals the relations between the caller and callee functions in an application. Various profiling tools, such as *GNU gprof* [89], can report this information based on source code or runtime analysis. The procedure involves tracking all function calls and returns. The call graph represents primarily the *control dependencies* of an application, since it shows the execution sequence on a coarse scale. However, regarding the fact that a caller function typically passes input data to the callee, which, in turn, produces data as (a) return value(s), indeed initiates the concept of data dependency among functions. Nevertheless, the data dependency presented by call graph is by no means a complete representation of the actual data dependency in the application. Though partly demonstrating the data transfers among functions (with direct calling relationship), it does not account for indirect data production/consumption. Even worse, it can be misleading, in the sense that there is no actual data dependency between the functions that have a direct calling relation in the call graph. All partitioning approaches which only rely on this graph or its variants may suffer from inefficiency if data communication is the primary concern in the partitioning process[10], which turns out to be mostly the case.

Certainly, what is needed as an input for a clustering algorithm is more than just the function call relation. Customarily, an annotated call graph not only visualizes these relations, but also provides other information, such as the number of times a particular function is called, a function's self-contribution to the whole execution time, and

[10] As a simple rule of thumb, input data incompleteness causes lack of accuracy, which, in turn, can result in an inaccurate partitioning solution.

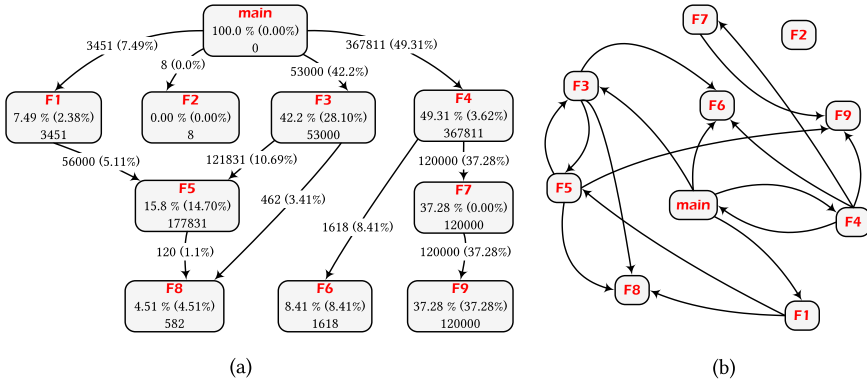


Figure 5.3: *A typical example of the data dependency among functions in an application based on (a) an annotated call graph (provided by GNU gprof) vs. its corresponding (b) QDU graph. As seen in the figure, the actual data dependency is not captured accurately in the call graph, e.g., F3 and F6 are indeed communicating data, however, it is not directly identified by the call graph.*

a function's entire contribution as an aggregate of its descendants. The execution time contribution is used as the main parameter for workload balancing in our proposed clustering algorithm.

Up to the present time, data communication as the input model is either specified by *synthetic* graphs or in the form of *task graphs*. Task graphs integrate the concepts of *control dependency* between tasks, i.e. which task has to start after the execution of which task(s), with *data dependency*, i.e. the ordering of execution is imposed by the data requirement. This is still reliant on *direct* dependence. Contrary to other approaches, we use the QDU graph to address this defect. The QDU graph exhaustively identifies all (direct or indirect) actual data transfers occurring during the execution of the application, regardless of any restriction which may result from the structure of the application.

Figure 5.3 (a) demonstrates a typical example of an annotated call graph. In each node, the number in parentheses specifies a function's self-contribution to the whole execution time. The number appearing on the left is the entire contribution of the node including all its descendants. On each edge, the aggregate contribution out of that branch is stated along with the number of times that link is crossed. The number at the bottom of each node represents the total number of times a function is called. Compared with the actual data dependency extracted by the corresponding QDU graph (as depicted in Figure 5.3 (b)), we clearly spot the differences. A control dependency does not necessarily imply an actual data dependency (e.g., the case of *main* and *F2*). Additionally, the call graph fails to report the data dependency between some functions (e.g., the case of *F3* and *F6*).

5.4.2 Greedy Algorithm

Algorithms for optimization problems typically carry out a sequence of steps, with a set of choices at each step. For many optimization problems, an exhaustive search to determine the best choice simply exceeds the needed effort. In these cases, there are

efficient algorithms which will do just fine in decision-making. A ***greedy algorithm*** obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes a choice which seems best at that moment. In other words, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution [58]. This heuristic strategy does not always yields optimal solutions, however, sometimes it does. In fact, the greedy strategy is quite powerful and works well for a wide range of areas, such as task scheduling and data compression. The proposed algorithm in this work also utilizes a greedy strategy to create task clusters. In the following, we briefly describe the general properties of a greedy strategy for problem solving. Furthermore, we discuss why it can be appropriate to solve the application partitioning problem defined in Section 5.3.

It is not always easy to prove that a greedy algorithm will produce an optimal solution. However, the suitability of the greedy strategy for a particular problem is commonly related to two properties: the ***greedy-choice property*** and the ***optimal substructure***. We say that a problem lends to the *greedy-choice property* if we can make a decision that looks best in the current problem, without considering results from its subproblems. It is worth to note that a greedy algorithm never reconsiders its choices. This is the main difference between a greedy algorithm and dynamic programming. In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, we typically solve dynamic programming problems in a bottom-up fashion, progressing from smaller subproblems to larger subproblems. Alternatively, they can be solved top-down using memoization[11]. Even though the code works top-down, we still must solve the subproblems before making a choice.

In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. Nevertheless, beneath every greedy algorithm, there is almost always a more burdensome dynamic programming solution. A problem exhibits *optimal substructure* if an optimal solution to the problem contains, within it, optimal solutions to subproblems. This property is a key element in assessing the suitability of a greedy algorithm as well as dynamic programming [58].

It is probable for greedy algorithms not to consistently find optimal solutions for NP-complete problems with different input data. Nevertheless, they are still effective because of their performance and often provide good approximations of optimal solutions. Greedy algorithms can suffer from the limitation that they are easily trapped in a local optimum, thus, fail to find the global optimal solution. They can make premature commitments to certain choices, which prevent them from reaching the optimal solution in the end. As an example, in the proposed clustering algorithm (see Section 5.4.3), if the initial functions in the clusters are not properly selected (let's assume they are heavily communicating), there will never be a chance for them to be merged in one cluster. Merging those functions can be part of the optimal solution, however, the algorithm fails to reach this solution.

[11] A technique used primarily for optimization by preventing the function calls to recalculate the results of previously processed inputs.

One may argue that the flaw mentioned above is not only relevant to the initial selection of functions, but it can degrade the quality of the solution, as we proceed to augment the clusters with subsequent decisions. Although this is generally true and it may happen, we are not expecting it to be dominant at all. The reason simply lies in the fact that the QDU graph used as the input data model, is very dense. This means all the functions which are *somehow* related to each other (regardless of their coupling degrees) get the chance to be part of the examined candidates for inclusion in the related clusters. In other words, if a function initially gets no priority to be checked for inclusion in a particular cluster, it is unlikely that it belongs to that cluster anyway. Therefore, it will be included in another cluster if it is assessed to be the top candidate, or its placement will be decided upon later. This is partly ensured by even gradual growth of all clusters — to respect a fair chance of inclusion for all clusters — in our algorithm. The experimental results presented in Section 5.6.2 further substantiate this claim. Should other input data models, such as call graph, be used for clustering, greedy algorithm would be seriously susceptible to this flaw, thus, producing low quality solutions.

Apart from finding optimal solutions to some problems, greedy algorithms can also be used as auxiliary selection criteria to prioritize options within other decision-making algorithms, such as Branch and Bound [88, 163]. Furthermore, it is usually possible to make greedy choices more efficiently by performing some kind of preprocessing. For example, in our clustering algorithm, the candidate functions are sorted based on the ranks assigned by a predefined function. This helps in the quick selection of substitutes in the *conflict resolution* phase of the algorithm. By preprocessing the list of candidates or even using an appropriate data structure (a priority queue in our case), greedy choices can be made quickly, thus improving the efficiency of the algorithm. It should be noted that in our algorithm, the list of options (function candidates) for each iteration of the main loop of the algorithm is updated based on recent selections. As a result, there is a requirement to sort the list repeatedly or maintain the list elements in such a way that facilitates selecting the top candidate.

5.4.3 Application Partitioning Algorithm

As discussed in the problem formulation (Section 5.3), our general partitioning model focuses on three primary goals, namely workload balancing, maximization of the intra-cluster communication, and minimization of the inter-cluster communication. These goals address the most critical aspects of application development and execution in a multiprocessor system. Needless to say that the load balancing between clusters can affect the performance of such a system, and the total communication time required by the tasks is mainly dependent on the amount of inter-cluster data transfers. Additionally, the flexible clustering algorithm proposed in this work is capable of considering system and/or architectural constraints as well as user-defined heuristic rules to adjust the partitioning outcome based on different preferences and objectives. These can include, for example, FPGA hardware area constraints, in reconfigurable systems.

The purpose of the clustering algorithm is to find (strongly) interconnected functions in order to group them together in a cluster. As discussed subsequently, the term *interconnected* may refer to more than just functions which extensively exchange data, though this characteristic is considered to have a critical role in clustering. As the main

component of the partitioning, the role of the clustering algorithm is crucial in subsequent stages of application development. The outcome clusters constitute the building blocks for mapping/scheduling tasks on the target heterogeneous architecture. Moreover, cluster information can be used as hints to a parallelization unit, where application revision is done to exploit possible coarse grained parallelism.

Grouping functions into clusters divides the data communication channels[12] in two classes: ***Intra-cluster*** (connecting functions within a cluster), and ***Inter-cluster*** (connecting functions in different clusters). In order to perform clustering, a well-defined objective is essential, which directs the formation of clusters. Generally, an objective falls into one or more of the following categories:

1. **Balanced clusters** - forming clusters that are nearly balanced with respect to the workload (mostly specified with the total required executing time). Balance does not necessarily imply equality of the amount of workload. More precisely, it is better defined by the uniformity of the required execution time among all the clusters, particularly in the case of heterogeneous PEs.

2. **Loosely-coupled clusters** - as few inter-cluster data communication channels as possible contributes to the overall performance and facilitates parallel tasks execution.

3. **Tightly-coupled functions within clusters** - functions having tight interactions are likely to be related, and hence it is beneficial to be integrated as one entity. The coupling may be attributed to bidirectional data communication channels, huge amount of data exchange, the workhorse relation, etc.

4. **Balanced data communication** - the data communication loads for inter-cluster channels are expected to be nearly balanced. This also implicitly implies uniformity within the number of required channels between clusters and the channel bandwidth specifications.

5. **Resource constraints satisfaction** - some system and/or architectural constraints may be specified in the clustering objective, e.g., on-chip memory mapping constraints, power consumption constraints, etc.

Apart from the defined objective, *heuristic* rules may also be used as adjustments in candidate selection procedure. These rules may surpass or superimpose the objective function. Anyhow, they are supposed to improve the quality of the prospective clusters. In the following, we present some sample heuristic rules.

- If a function only communicates data with another function, it is preferable to put them inside the same cluster.

- If the examined cluster is relatively small in terms of the overall execution time, it is desirable to favor the node with the largest execution time in the selection criterion and vice versa. Definition of a small or a large cluster is totally subjective.

[12] By data communication channel, we mean a virtual path along which data is communicated between two functions.

Nevertheless, it should be fairly easy to just estimate a contribution percentage based on the number of clusters. As an example, for 5 clusters, we expect an execution time contribution of about 20% for each cluster. Thus, a cluster with a contribution of 4% is considered too small.

- It is beneficial to limit the number of bidirectional data communication channels as much as possible. These channels impose severe restrictions on any possible subsequent parallelization[13].

These heuristic rules are only presented here as examples. They may be revised or augmented for adaptation to individual system design, architectural or user preferences and requirements.

Figure 5.4 depicts the outline of the proposed task clustering algorithm. According to our problem formulation described in Section 5.3, clusters cannot be empty. As a result, we opt to select one function for each cluster to form the initial populations in clusters. The initial user-defined selection has a substantial impact on the final clusters and should be done cautiously[14]. We adopt the following criteria for the initial selection. To form clusters around computationally-intensive kernels, first we spot those functions which pass a minimum contribution threshold. This value is predefined based on the number of clusters. Just as an example, if we plan to end up with 4 clusters, a minimum amount of about 10% might be appropriate. As a rule of thumb, in each cluster with approximately 25% contribution to the total execution time, we probably find a kernel with a contribution around 10%. This is quite dependent on the application, and does not necessarily hold in every situation. The selected boundary value can be increased if too many kernels are selected or decreased if not sufficient kernels meet the condition. We also follow another strategy to discard excessive picked out kernels and to ensure not finally end up with surplus small clusters. In the preprocessing step, pairs of kernels that are topologically close to each other (at most within a predefined distance in the QDU graph) are compared and the kernel with the lower contribution is discarded, unless the kernel has a contribution percentage higher than a threshold value, making it illogical to discard the kernel. As an example, when we are heading for five clusters, two kernels with 15% and 18% contributions are both assumed to form their own clusters, no matter how topologically close they are. Following this step, initial clusters are created with selected kernels as first populations. The clusters gradually grow during subsequent steps.

In the main body of the algorithm, at each iteration, clusters are augmented with the best interconnected neighbor nodes. For each cluster, the candidates are evaluated by a *ranking function* ($\mathscr{R}$). This function inspects all the candidates for each examined cluster and assigns a value to each candidate indicating the suitability of that candidate for merging. It is clear that the selection process is strongly dependent on the definition of the ranking function, providing high flexibility for the clustering algorithm. In order to comply with different policies, individual terms and weights in the definition of the ranking function can be customized. The ranking function should not be confused with the definition of the *total cost function* (TC_P), as defined in Section 5.3. Although they can be defined with comparable terms and weights, the latter is related to the cost estimation

[13] This requires that we retain the direction information in the QDU graph.
[14] This initial selection can be done *manually* by an expert user or, *automatically*, based on a defined strategy.

INITIAL SELECTION	1	select the initial kernels
SELECTION REVISION	2	revise the initial kernels if necessary (k kernels are selected out of n total functions in the input application)
CLUSTER CREATION	3	create the initial clusters $V_1, V_2, .., V_k$
CLUSTER REVISION	4	**repeat**
	5	{ // phase 1: candidate selection for merging
	6	**for** (all ***unfinished*** clusters)
	7	{
	8	**if** (no neighbor node left)
	9	mark current cluster as ***finished*** and **continue**;
	10	**for** (all neighbor nodes of current cluster)
	11	{
	12	**if** (candidate node does not satisfy any constraint) **continue**;
	13	evaluate with the ranking function;
	14	}
	15	**if** (all candidates failed to pass constraints)
	16	mark current cluster as ***finished*** and **continue**;
	17	sort candidates in decreasing order based on ranks;
	18	mark the candidate with the highest rank for merging;
	19	}
		// phase 2: conflict resolution
	20	**while** (there exists conflict between ***undecided*** clusters)
	21	{
	22	**flag** *iter_dec*=false;
	23	**for** (all clusters $V_1, V_2, \ldots, V_m$ which marked C as the top candidate)
		// C refers to the node for which maximum value of m is obtained
	24	{
	25	**if** (*iter_dec* && C has the highest score in the current cluster)
	26	{
	27	*iter_dec*=true;
	28	mark the current cluster as ***decided***;
	29	}
	30	**else**
	31	{
	32	unmark C in the current cluster;
	33	**if** (any candidate left) mark the next candidate in list;
	34	**else** mark the current cluster as ***decided*** and ***finished***;
	35	}
	36	}
	37	}
		// phase 3: merge and update
	38	**while** (there exists unmerged marked candidates)
	39	{
	40	merge marked candidate with its respective cluster;
	41	update cluster properties;
	42	}
	43	} **until** (all clusters are marked as ***finished***);
COMPLETION	44	resolve the problem of unassigned nodes

Figure 5.4: *The outline of the task clustering algorithm. The output of the algorithm is a k-way partition of the input application.*

of a partition. The ranking function is only used to choose the winner candidate which is indeed expected to contribute to the optimality of the final solution. It should be stressed that the terms and weighting factors used in the ranking function do not necessarily have to cover all the cost functions defined for nodes and edges in the input model. In

particular, the user-defined heuristic rules play vital roles in the assignment of ranks to candidates (an ordering of candidates), yet these rules are not defined in the context of the cost functions.

As an initial attempt in the proposed clustering algorithm, we define the following simple ranking function, which considers only two terms in ranking candidates:

$$\mathscr{R}(v_i) = \sigma_1 \mathscr{R}_{exe}(v_i) + \sigma_2 \mathscr{R}_{cou}(v_i),\ 1 \le i \le n, \tag{5.1}$$

where $\mathscr{R}(v_i)$ denotes the final ranking value of the function v_i. $\mathscr{R}_{exe}(v_i)$ refers to the *execution rank*[15] of v_i among all the candidates to be merged with the examined cluster. As mentioned before, the final value of the $\mathscr{R}_{exe}(v_i)$ is expected to be determined in accordance with the heuristic rules if relevant. Higher rank implies better chance for selection. The *coupling rank* ($\mathscr{R}_{cou}$) is calculated based on the data communication intensity of v_i with respect to the functions currently residing in the examined cluster. Tighter data communication (quantitative value of the total bytes transferred) infers greater probability for inclusion in the current cluster. σ_1 and σ_2 are weighing factors to adjust individual ranking terms in the final multi-objective ranking function. Heading for a fair share of each metric requires an equal value of weight factors for these two parameters, e.g., setting both to 0.5.

Should constraints be defined that by definition disqualify candidate(s), they are applied in the candidate selection prior to the evaluation of the ranking function. Other (non-disqualifying) heuristic rules can be optionally applied as either pre or post to the ranking function in order to modify ranks in favor or against certain candidate(s). Alternatively, these rules can be implemented directly in the definition of the ranking function. In the following phase, *Conflict resolution* is addressed for clusters competing to draw inward an identical candidate. As clusters grow larger, this condition happens when a particular candidate appears on the top of the lists of two (or possibly more) clusters. In case of a conflict, we favor the cluster in which the examined candidate has achieved the highest score. For all the other competing clusters, new substitutes — if any exists — residing right next to the top candidates are selected. The same process is repeated until all conflicts[16] are resolved.

When we include a candidate within a cluster, the properties and parameters of the cluster should be updated accordingly, in order to represent the current status of the cluster. These parameters are particularly used in checking the conditions for finalizing clusters as well as in the ranking function and constraints. For example, if a cluster is growing big with respect to the total execution time contribution, exceeding a predefined threshold, we opt to mark the cluster as finished to prevent further growth. Generally, the *stopping point*, where we decide not to continue with a particular cluster, is subjective and should be declared in constraints.

The last stage of the algorithm deals with the nodes which had no chance to end up inside any cluster, during the execution of the main loop of the algorithm. This rarely happens provided that we define appropriate and reasonable constraints complying with the properties of the target system and the application. Nevertheless, we propose two solutions to handle this situation. In the first, we may opt to put all the remained functions in an extra cluster or more. The same procedure as described in the *Cluster Revision*

[15] The rank of a function among the ordered list of functions based on the execution time contribution.
[16] New conflicts may appear during substitutions.

stage of the algorithm can be used for this purpose. Another solution would be to relax the constraint which prevents the functions to be included in the currently defined number of clusters. It should also be noted that this is not always feasible. In case there are uncompromising constraints for clusters, e.g., hardware area restriction, it is not possible to go for this solution. It should be noted that if *isolated* functions exist in the QDU graph, it is very probable[17] that these functions do not end up in any cluster.

The clustering algorithm depicted in Figure 5.4 is written in a general form and excludes some implementation details. Furthermore, we have intentionally structured and formulated the algorithm in such a way to facilitate its parallelization. A parallel version of the algorithm can be easily derived from the general outline, which can even further reduce the time (complexity) needed to solve the problem.

5.5 Complexity Analysis

Measuring the time and/or space needed to solve a computational problem requires an appropriate model. Conventionally, the *deterministic Turing machine* is adopted as a standard computational model for this purpose. The time required by a deterministic **Turing** machine *TM* on the input *I* is denoted with the total number of state transitions[18] the machine makes before it stops. *TM* is said to operate within time $f(n)$, if the time required by *TM* on each input *I* of length n is at most $f(n)$. In other words, a problem can be solved in time $f(n)$, if there exists a *TM* operating in time $f(n)$ that solves the problem. A similar definition can also be stated for the space requirement of an algorithm. Although time and space are the most widely-used complexity measurements, any computational resource can be used in this respect. In summary, the theory of complexity allows the classification of problems based on their difficulty. Furthermore, it enables us to compare different solutions for a given problem. In the following, we discuss the time and the space complexities of the proposed task clustering algorithm.

5.5.1 Time Complexity

Time complexity analysis of algorithms is based on the assumption that performing a basic operation in computing takes a fixed amount of time. Thus, the amount of time taken by an algorithm is proportional to the number of basic operations performed by the algorithm. As a result, time complexity of an algorithm is commonly estimated by counting the number of basic operations performed. Since it is not possible nor necessary to quantify exactly the wall-clock time needed to run an algorithm, the time complexity of the algorithm is evaluated as a function of the input size of the problem.

The time complexity of an algorithm may vary based on the different inputs of the same size, simply because some inputs may be faster to solve compared to others. To address this issue, three cases for the complexity measurement are common, namely best-, worst- and average-case complexity. Time complexity in the best and worst cases

[17] It is for sure to happen if no initial kernel is selected which resides among the functions in an isolated cluster. This is simply because these function get no chance to be examined as potential candidates for other clusters in our algorithm.

[18] State transitions are also called *steps* in some literature.

are computed with the best and worst inputs of a certain problem size. Average-case time complexity refers to solving the problem on an average. It can only be defined with respect to a probability distribution over the inputs. For example, if all inputs of the same size are assumed to be equally probable to appear, the average-case complexity is defined with respect to the uniform distribution over all inputs of a certain size.

As a standard, the time complexity of an algorithm is expressed using the *big $\mathcal{O}$ notation*, which suppresses multiplicative constants and lower order terms [58]. When expressed this way, the time complexity is said to be described *asymptotically*, i.e. as the input size tends to infinity. The worst-case time complexity of an algorithm, $T(n)$, is the most commonly used measure of the time complexity, which is the maximum amount of time taken on any input of size n.

In our task clustering algorithm, it is assumed that n functions have to be partitioned into k clusters. These parameters (n and k) constitute the sizes of the inputs to the algorithm. Based on the initially selected kernels for each cluster, the number of remaining functions that go through the *Cluster Revision* stage is $(n-k)$. The time taken for the *Initial Selection* and *Selection Revision* stages is dependent on the user implementation. We assume that it is constant and simply ignore it in our time complexity analysis. The time complexity of the third stage, *Cluster Creation*, is clearly $\mathcal{O}(k)$. To find the time complexity of the main loop in the *Cluster Revision* stage (repeat/until), we shall determine the number of times it is iterated. In each iteration of the loop, clusters are expected to take only one candidate in, unless they cannot grow larger due to defined constraints or no connectivity with neighbor nodes. In the best case, we can assume that, in each iteration, all the k clusters evenly take one candidate in. Hence, the loop is repeated $\left\lceil \frac{n-k}{k} \right\rceil$ times. Worst case happens when from the beginning, $(k-1)$ clusters can grow no more, and only one cluster takes in one element at each time, i.e. $\mathcal{O}(n-k)$. Assuming a uniform distribution of the probabilities of different inputs between these two extremes, the average-case complexity can be computed as follows:

$$T(n) = \frac{\frac{n-k}{1} + \frac{n-k}{2} + \frac{n-k}{3} + \cdots + \frac{n-k}{k}}{k} \tag{5.2}$$

$$= \frac{(n-k)\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}\right)}{k} \tag{5.3}$$

$$= \frac{(n-k)\mathcal{H}_k}{k}, \tag{5.4}$$

where $\mathcal{H}_k$ denotes the k-th *harmonic number*[19]. The values of the sequence $(\mathcal{H}_k - \ln(k))$ decreases monotonically towards the following limit:

$$\lim_{k\to\infty} (\mathcal{H}_k - \ln(k)) = \gamma, \tag{5.5}$$

where γ is the *Euler-Mascheroni* constant (0.57721...) and $\ln(k)$ is the natural logarithm of k[20]. $\mathcal{H}_k$ grows as fast as the natural logarithm of k, i.e. $\mathcal{H}_k = \theta(\ln k)$. The reason is that $\mathcal{H}_k$ can be approximated by the integral $\int_1^k \frac{1}{x} dx$, which is equal to $\ln(k)$, as depicted in Figure 5.5. As a result, the average-case time complexity of the loop can be stated as

[19] The sum of the reciprocals of the first k natural numbers.
[20] Logarithm to the base e=2.718281828, also known as Euler's number.

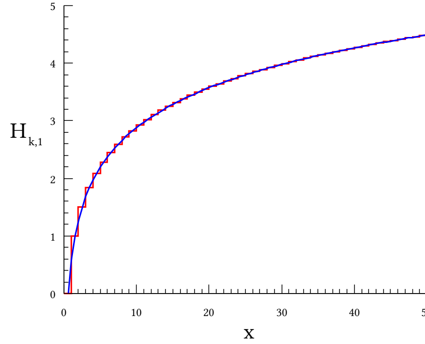


Figure 5.5: The generalized harmonic number of order k of 1 ($\mathcal{H}_{k,1}$ or simply $\mathcal{H}_k$) with $k = \lfloor x \rfloor$ (red line) and its asymptotic limit $\gamma + \ln(x)$ (blue line). Adapted from the Wikimedia Commons file "HarmonicNumbers.svg" (http://commons.wikimedia.org/wiki/File:HarmonicNumbers.svg).

follows:

$$\mathcal{O}\left(\frac{(n-k)\ln k}{k}\right). \tag{5.6}$$

In the worst case, the *for* loop at lines 6-19 in Figure 5.4 is checked for each of the k clusters in each iteration of the outer main loop. Its time complexity is thus $\mathcal{O}(k)$. The time needed to check the condition in line 8 is dependent on the implementation of the algorithm. However, in the simplest case, the use of an adjacency matrix to represent the input QDU graph incurs checking at most $(n-k)$ entries in the matrix. This is to determine the connectivity of each remaining $(n-k)$ nodes to the examined cluster. With the same reasoning, the *for* loop at lines 10-14 is repeated $(n-k)$ times in the worst case (where all the remaining nodes, which are not assigned to any cluster, are directly connected to one of the nodes inside the examined cluster). We assume that the evaluation with the ranking function, as well as constraint checking, only takes constant amount of time, i.e. $\mathcal{O}(1)$. It is also straightforward to infer that evaluating the condition in line 15 takes only constant time, provided that we update a simple *status flag* while processing the candidates in the loop at lines 10-14.

Optimal sorting is known to have *linearithmic*[21] time complexity. Thus, sorting the candidates for each cluster will take $\mathcal{O}\left((n-k)\log(n-k)\right)$ steps in the worst case[22]. The time complexity inside the *for* loop at lines 6-19 is thus bounded by the fastest growing term, namely $\mathcal{O}\left((n-k)\log(n-k)\right)$. At first glance, it seems that the sorting in line 17 is simply redundant, because when we are evaluating the candidates with the ranking function (line 13), we can tag the top candidate at the same time. In other words, no extra time is needed for sorting the candidates, or simply we can scan once more the list of candidates to pick the top one, i.e. *linear* complexity. While this assumption is true, it is not applicable in our solution, because the clusters may compete for certain candidates and this has to be addressed later. The sorting would compensate for the effort that we have to do in order to find appropriate substitutes. This will be detailed in the following discussion.

[21] Portmanteau of linear and logarithmic

[22] Any sort algorithm that runs in linearithmic time in the worst case can be used, such as heap sort, merge sort, smooth sort, etc.

The second phase of *Cluster Revision* (lines 20-37) deals with the conflicts between the clusters for the selected candidates. In the worst case, suppose that all the k clusters have tagged the same candidate for inclusion. Determining the number of conflicting clusters (m in line 23) requires a linear scan of the top candidates in all the k lists. This denotes the number of times the *for* loop at lines 23-36 is repeated, which is bounded by k. The statements at lines 25-34 only need a constant amount of time to be executed. It is obvious that during each iteration of the outer *while* loop at lines 20-37, one conflict between the clusters is cleared. As result, in the worst case, a maximum number of k conflicts should be cleared. Note that in each conflict resolution, the number of clusters that can possibly compete for a candidate is decreased by at least one. Thus, in the worst case, the *for* loop at lines 23-36 can be executed with less number of iterations at each time ($k,\ k-1,\ k-2,\ \cdots,\ 2$). As a consequence, the time complexity required to resolve all the conflicts between the clusters is bounded by the following:

$$\mathcal{O}\left(k+(k-1)+(k-2)+\cdots+2\right)=\mathcal{O}\left(\frac{k(k-1)}{2}-1\right)=\mathcal{O}\left(k^2\right). \tag{5.7}$$

In the last phase of *Cluster Revision*, we go through all the *unfinished* clusters to include the selected candidates in their respective clusters, which is, in the worst case, $\mathcal{O}(k)$.

The last stage of the algorithm, *Completion*, is dependent on the defined strategy and its implementation. In the worst case, there can be $(n-k)$ unassigned nodes left for this stage. Regardless of the strategy used (a linear scan of the nodes or any other kind of processing), we assume that the time for this stage is not going to be the dominant term (or at most as large as the Cluster Revision stage). In conclusion, the worst-case time complexity of the clustering algorithm is computed as follows.

$$\begin{aligned} T(n,k) &= \mathcal{O}\left(k+\left(\frac{n-k}{k}\right)\left(\left(k\times(n-k)\times\log(n-k)\right)+k^2+k\right)+(n-k)\right) && (5.8)\\ &= \mathcal{O}\left((n-k)^2\log(n-k)+k(n-k)\right)\ . && (5.9)\end{aligned}$$

The value of k is bounded by the value of n, i.e. $1\leq k\leq n$. Thus, we can logically assume that k is a constant, or in the worst case, to have the maximum value for $T(n,k)$, k should be $\frac{n}{2}$, which results in the following time complexity for the algorithm:

$$T(n)=\mathcal{O}\left(n^2\log n\right)\ . \tag{5.10}$$

In Section 5.6.1, we will prove that the exhaustive search of all the k-way partitions for a given application with n functions requires *superpolynomial* (more specifically, *exponential*[23]) time. This means that in order to find the optimal solution in any case, the time required to solve the partitioning problem grows exponentially with the increase in the input size n. As n grows, the time needed to determine the optimal solution with the exhaustive search soon turns to be huge and the method simply becomes infeasible. It is not very difficult to see why exponential time algorithms are impractical for moderate to large input sizes. Assume that a program executes 2^n basic operations on a computer that

[23] An algorithm is said to run in exponential time, if $T(n)$ is upper bounded by $2^{poly(n)}$, where $poly(n)$ is some polynomial in n.

performs 10^{12} operations per second. For a moderate input size n, say 100, the program would run for about 4×10^{10} years, which is roughly the age of the universe. Even with a much faster computer, the program would only be useful for very small instances and in that sense the intractability of a problem is somewhat independent of technological progress. To address the intractability of the application partitioning problem, heuristic methods, such as our proposed solution, which offer polynomial time complexities, are inevitable.

5.5.2 Space Complexity

The space complexity of the clustering algorithm can only be accurately determined when the implementation details are clear, since it is tightly dependent on the utilized data structures. The algorithm presented in Figure 5.4 does not discuss the implementation details. Nevertheless, we provide an estimation of its space complexity. To this purpose, we assume straightforward implementations of the required data structures without considering any optimization, revision or whatsoever.

For the main input data model, the QDU graph, we use an *adjacency matrix* to store the information of the data transfers among the functions. Since we expect the QDU graph to be dense, the matrix is preferred over a list representation, which is usually used for sparse graphs. It should be noted that, besides the space tradeoff, the two data structures also facilitate different operations. Finding all the functions communicating with a given function in an adjacency list simply requires reading the list. With an adjacency matrix, an entire row must be scanned instead, which takes $\mathcal{O}(n)$ time. Whether there is data communication between a pair of given functions can be determined immediately with an adjacency matrix, while, in the case of the adjacency list, it requires time proportional to the minimum length of the lists of the two functions.

The graph used in the clustering algorithm is undirected and has no self-loops. As a result, it is possible to cut off the upper or lower triangular elements in the matrix as well as the elements stored in the main diagonal. This means it is only required to store $\left(n^2 - \left(\frac{n^2}{2} - \frac{n}{2} + n\right)\right) = \left(\frac{n(n-1)}{2}\right)$ elements. The compensation for this space saving is nominal, i.e. a simple check on the indices of elements in the matrix to swap them before lookup, where pertinent. This is done in constant time. Furthermore, for each function in the application, a set of data values can be maintained, such as the execution time contribution, the required hardware area on FPGA, etc. The space needed is, thus, always linear with the number of functions, i.e. $\mathcal{O}(n)$.

In order to keep track of the functions in each of the k clusters, a *list* data structure can be used. In case a fixed-length array is selected as for the list implementation, the maximum space for accommodating $(1 + n - k)$ elements should be reserved for each cluster. That would result in a total space complexity of $\mathcal{O}(k \times (n - k + 1))$. A constant amount of space is also required to maintain the integrity of each cluster during *cluster revision* stage of the algorithm. The extra space accommodates the required housekeeping flags, such as *finished* and *decided*, to enable the update process on clusters. Using a *linked* structure to implement the lists has the advantage that we do not have to reserve the maximum possible space for each list. Note that the total number of elements that can be distributed among all the k clusters, is n. In other words, the space complexity to

implement the lists would be $\mathcal{O}\left(k + \mu \times n\right)$. μ denotes a constant factor to compensate for the implementation of the required links in the data structure.

Implementing the for loop at lines 10-14 necessitates that we evaluate and keep a list of all candidates for each cluster. This is because we need to sort these candidates later, as stated in line 17. To accomplish this task, sufficient space should be reserved for each cluster independently to accommodate the maximal number of candidates, which is $(n - k)$. As a result, the total space complexity would be $\mathcal{O}(k \times (n - k))$.

To efficiently[24] find the conflicting top candidate C, for which the maximum number of clusters are competing, we have to keep a list for each potential top candidate. In order to instantly access each function as a potential candidate, an array of the length $(n - k)$ is maintained, which enables us to refer to each function based on its index in the array. It is worth noting that we do not necessarily have to reserve space for the maximum number of clusters in each list to be linked to each of $(n - k)$ elements of the array. If cluster CL_x marks element f_y in the array, it has to appear in the list linked to f_y, thus, it will not be linked to another element f_z. In other words, the whole space required for all the formed lists would be linear with k, the number of clusters. Consequently, this will lead to an extra space requirement of $\mathcal{O}((n - k) + k) = \mathcal{O}(n)$.

The space complexity of the task clustering algorithm with the input sizes of n and k, denoted by $S(n, k)$, is then calculated as follows:

$$S(n,k) = \mathcal{O}\left(\left(\frac{n(n-1)}{2}\right) + (n) + (k+n) + (k \times (n-k)) + (n)\right) = \mathcal{O}\left(n^2 + k(n-k)\right). \quad (5.11)$$

As discussed previously in Section 5.5.1, the value of k is bounded by the value of n. Compared to n, we can consider k as a constant, or in the worst case, k is assumed to be equal to $\left(\frac{n}{2}\right)$. In any case, the upper bound for the space complexity of the algorithm would be $\mathcal{O}\left(n^2\right)$.

5.6 Synthetic Analysis

In order to evaluate the quality of the results found by the proposed heuristic algorithm, a proper test bench is needed. In our case, it is not relevant to compare the results with our approaches, because they simply use different input data models. However, the ideal way would be to compare the solution found by our algorithm against the optimal one which is determined by an exhaustive search of all possible solutions. This allows us to precisely assess how close we can get to the optimal solution. Of course, as already discussed in Section 5.5.1, exhaustive search would only be feasible for limited problem sizes. Nevertheless, it is the prefect verification to indicate whether or not the selected partitioning determined by our heuristic algorithm is near-optimal or even the optimal one itself.

Before setting up a randomly generated synthetic test bench, it is required to have a clear insight of all the possibilities to come up with a partitioning. It is also needed to have an estimation of the problem size that is still tractable based on the problem

[24] By ***efficient***, here we mean imposing no extra time penalty than a single scan of all the k clusters.

formulation. In this section, we first elaborate on a detailed analysis of the application partitioning problem from a combinatorial mathematics perspective, and then present the experimental results based on synthetically generated data.

5.6.1 Exhaustive Application Partitioning

In a general sense, *application partitioning* can be considered as a *set partitioning* problem, since we aim to form non-empty groups of the entire tasks in an application, such that a particular task belongs to exactly one group (the groups are pairwise disjoint). By definition, the number of different partitioning cases corresponds to n-th *Bell* number (B_n), where n denotes the number of tasks in a program. Bell numbers satisfy the recursion $B_{n+1} = \sum_{i=0}^{n} \binom{n}{i} B_i$, starting with $B_0 = B_1 = 1$.

Due to the fact that the number of groups is pre-decided in our problem, the partitioning cases are reduced. In this respect, the number of different possibilities to create exactly k clusters out of n tasks, TC_k^n complies with the following recurrence formula:

$$TC_k^n = TC_{k-1}^{n-1} + k \times TC_k^{n-1} \tag{5.12}$$

To justify this recurrence, we assume that the partitioning of n tasks $(T_1, T_2, ..., T_n)$ into k clusters can only be achieved in one of these cases:

- Partitioning $T_1, T_2, ..., T_{n-1}$ into $(k-1)$ clusters and putting T_n into the kth cluster.
- Partitioning $T_1, T_2, ..., T_{n-1}$ into k clusters and then adding T_n into one of the existing clusters, which could be accomplished in k ways.

It is also trivial to observe that $TC_1^n = TC_n^n = 1$. The expansion of Equation (5.12) results in explicit values. Suppose we plan to come up with only two clusters (TC_2^n):

$$\begin{aligned}
TC_2^n &= TC_1^{n-1} + 2 \times TC_2^{n-1} && (5.13)\\
&= 1 + 2(TC_1^{n-2} + 2 \times TC_2^{n-2}) && (5.14)\\
&= 1 + 2 + 2^2 + 2^3 \times TC_2^{n-3} && (5.15)\\
&= 1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-3} + 2^{n-2} \times TC_2^2 && (5.16)\\
&= \frac{2^{n-1} - 1}{2 - 1} = 2^{n-1} - 1 && (5.17)
\end{aligned}$$

To verify the number of clusters in another way, we can say that there are 2^n ordered pairs of complementary clusters (CL_1 and CL_2). By discarding the two illegal cases where CL_1 and CL_2 are empty and considering only unordered pairs, which sets aside half of the possibilities, we simply reach the $(2^{n-1} - 1)$ partitioning cases. Other identities

can also be computed with the same spirit:

$$TC_2^n = 2^{n-1} - 1 = \frac{\frac{1}{1}(2^{n-1} - 1^{n-1})}{0!} \tag{5.18}$$

$$TC_3^n = \frac{3^{n-1} - 2^n + 1}{2} = \frac{\frac{1}{1}(3^{n-1} - 2^{n-1}) - \frac{1}{2}(3^{n-1} - 1^{n-1})}{1!} \tag{5.19}$$

$$\begin{aligned} TC_4^n &= \frac{4^{n-1} - 3^n + 3 \times 2^{n-1} - 1}{6} \\ &= \frac{\frac{1}{1}(4^{n-1} - 3^{n-1}) - \frac{2}{2}(4^{n-1} - 2^{n-1}) + \frac{1}{3}(4^{n-1} - 1^{n-1})}{2!} \end{aligned} \tag{5.20}$$

$$\begin{aligned} TC_5^n &= \frac{5^{n-1} - 4^n + 2 \times 3^n - 2^{n+1} + 1}{24} \\ &= \frac{\frac{1}{1}(5^{n-1} - 4^{n-1}) - \frac{3}{2}(5^{n-1} - 3^{n-1}) + \frac{3}{3}(5^{n-1} - 2^{n-1}) - \frac{1}{4}(5^{n-1} - 1^{n-1})}{3!} \end{aligned} \tag{5.21}$$

$$\vdots$$

As a result, the following explicit formula for TC_k^n is derived:

$$\begin{aligned} TC_k^n &= \sum_{i=1}^{k} (-1)^{k-i} \frac{i^{n-1}}{(i-1)!(k-i)!} \\ &= \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n . \end{aligned} \tag{5.22}$$

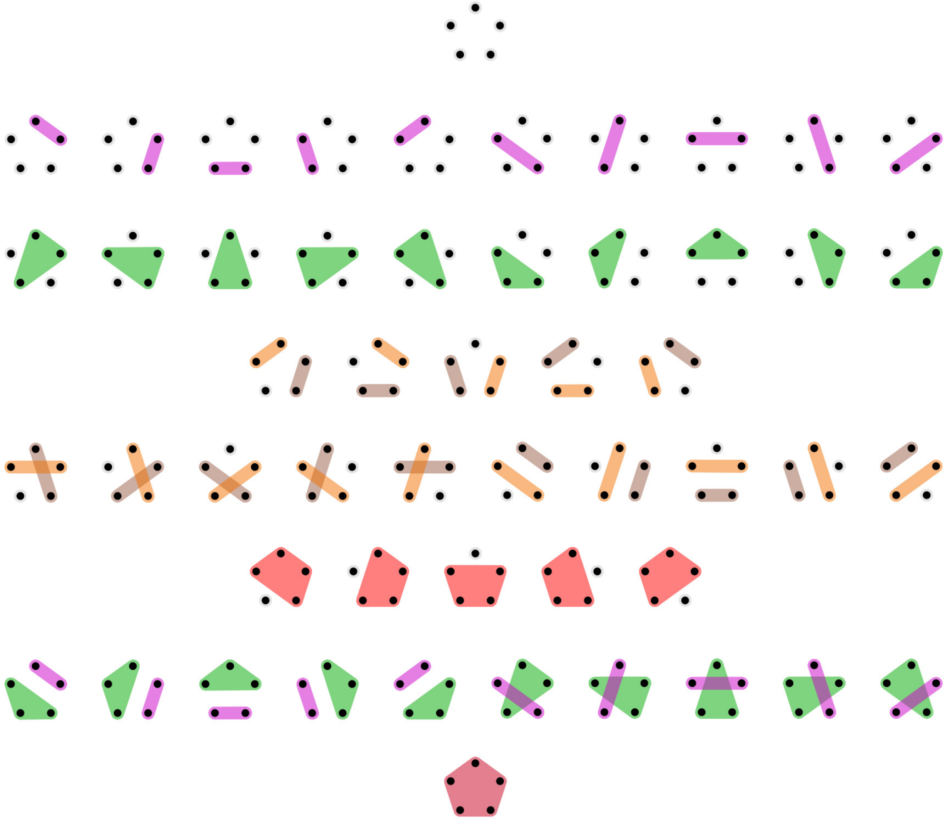
Equation (5.22) corresponds to the *Stirling* number of the second kind, which specifies the number of ways to partition a set of n elements into k non-empty groups. There are various notations for this number. Here, we adopt the following notation used by *Donald Knuth* [114]:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{i} \binom{k}{i} (k-i)^n , \tag{5.23}$$

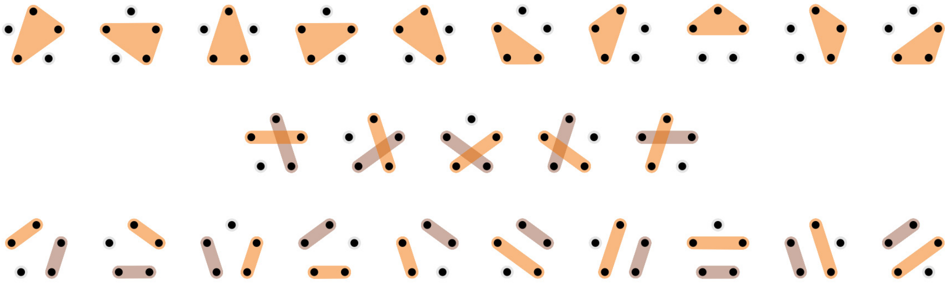
where $\binom{k}{i}$ denotes the *binomial coefficient*, which is the number of ways of picking i unordered outcomes from k possibilities, also known as a combination or combinatorial number. Figure 5.6(a) depicts all the possible ways to partition a set of five elements into different number of groups. The *Stirling* number of the second kind for $\left\{ {5 \atop 3} \right\}$ along with the possibilities is shown in Figure 5.6(b).

The way we partition an application consisting of n functions into k cluster is somehow different from the standard set partitioning problem. In our solution, we first put aside k elements (initial members for clusters are selected a priori), i.e. only $(n-k)$ elements are left to be partitioned into k clusters. Using Equation (5.23), the number of possible ways to partition the remaining elements would be:

$$\left\{ {n-k \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^{n-k} . \tag{5.24}$$



(a) All possible partitions for a 5-element set.



(b) Partitioning a 5-element set into exactly 3 groups.

Figure 5.6: *An example of partitioning a set of five elements. (a) All the possible ways of partitioning the set into one, two, three, four and five groups. With just one way to partition the set into one or five groups, ten ways for four groups, twenty five ways for three groups, and fifteen ways for two groups, there are, in total, fifty two possibilities. (b) All the possible ways of partitioning the set into exactly three groups, which corresponds to the Stirling number of the second kind* $\left\{ {5 \atop 3} \right\}$*. Adapted from the Wikimedia Commons file "Image:Set_partitions_5;_circles.svg" (http://commons.wikimedia.org/wiki/File:Set_partitions_5;_circles.svg).*

The initial k elements can be distributed in $k!$ different ways in the k clusters. It should be noted that each cluster only gets one element and the clusters are not considered identical anymore. The clusters are differentiated based on the function which is initially put in each cluster. As a result, the total number of possibilities would be:

$$(k!) \times \left\{ {n-k \atop k} \right\} = \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^{n-k} \,. \tag{5.25}$$

As mentioned before in the set partitioning definition, we stress again that the created groups are *non-empty*, i.e. the value calculated for $\left\{ {n \atop k} \right\}$ (Equation (5.23)) only accounts for the possibilities when no part is left empty at the end. This is not what we want for our application partitioning problem, because the clusters are initially filled with one element, i.e. the final clusters simply cannot be empty at all. As a result the remaining $(n-k)$ elements should be distributed into clusters accounting for the possibility that a cluster gets no new element. Considering this fact, we know for sure that we would have a higher number of possibilities than the one computed with Equation (5.25). The exact number of possibilities for our partitioning problem is calculated as follows.

Using Equation (5.25), the number of possibilities that only one of k clusters gets no share of partitioning $(n-k)$ elements into them would be calculated as:

$$\binom{k}{1} \times \left\{ {n-k \atop k-1} \right\} \times (k-1)! \,. \tag{5.26}$$

This is justified by the fact that the only cluster which gets no new element of the distributed elements, can be chosen in $\binom{k}{1} = k$ ways and the $(n-k)$ elements are partitioned among the remaining $(k-1)$ clusters. Similarly, if exactly two clusters are left aside (get no share of the distributed elements), the number of possibilities would be calculated as follows:

$$\binom{k}{2} \times \left\{ {n-k \atop k-2} \right\} \times (k-2)! \,. \tag{5.27}$$

Other cases, where some clusters get no new element out of the partitioning, are computed in the same manner. In the last case, only one of the clusters gets all the $(n-k)$ elements, i.e. $(k-1)$ clusters are completely left aside. This case is stated with the following equation:

$$\binom{k}{k-1} \times \left\{ {n-k \atop k-(k-1)} \right\} \times (k-(k-1))! = k \times \left\{ {n-k \atop 1} \right\} = k \times 1 = k \,. \tag{5.28}$$

Hence, the total number of possibilities that are not accounted for in Equation (5.25), is computed as follows:

$$\sum_{j=1}^{k-1} \binom{k}{j} \times \left\{ {n-k \atop k-j} \right\} \times (k-j)! \,. \tag{5.29}$$

Equation (5.29) specifies the total number of possibilities in which, during the partitioning of $(n-k)$ elements into k clusters, at least one of the clusters gets no share of the distributed elements. Using Equation (5.25) and Equation (5.29), the total number of possibilities to partition an application consisting of n functions into k clusters, $\mathscr{AP}(n,k)$,

is calculated as follows:

$$\begin{aligned}
\mathscr{AP}(n,k) &= (k!) \times \left\{ \begin{matrix} n-k \\ k \end{matrix} \right\} + \sum_{j=1}^{k-1} \binom{k}{j} \times \left\{ \begin{matrix} n-k \\ k-j \end{matrix} \right\} \times (k-j)! \\
&= \binom{k}{0} \times \left\{ \begin{matrix} n-k \\ k-0 \end{matrix} \right\} \times (k-0)! + \sum_{j=1}^{k-1} \binom{k}{j} \times \left\{ \begin{matrix} n-k \\ k-j \end{matrix} \right\} \times (k-j)! \\
&= \sum_{j=0}^{k-1} \binom{k}{j} \times \left\{ \begin{matrix} n-k \\ k-j \end{matrix} \right\} \times (k-j)! \\
&= \sum_{j=0}^{k-1} \left[\frac{k!}{j!\,(k-j)!} \times \left(\frac{1}{(k-j)!} \sum_{i=0}^{k-j} (-1)^{k-j-i} \binom{k-j}{i} i^{n-k} \right) \times (k-j)! \right] \\
&= \sum_{j=0}^{k-1} \binom{k}{j} \left(\sum_{i=0}^{k-j} (-1)^{k-j-i} \binom{k-j}{i} i^{n-k} \right) \\
&= \sum_{j=0}^{k-1} \frac{k!}{j!} \left(\sum_{i=0}^{k-j} \frac{(-1)^{k-j-i}}{i!\,(k-j-i)!} i^{n-k} \right).
\end{aligned} \tag{5.30}$$

Using Equation (5.30) gives an indication of the time needed to exhaustively search all the partitions of an application as described in Section 5.4.3. As an example, if we assume that a conventional computer can examine 100 million partitions per second, an exhaustive search for the optimal solution of partitioning an application with 30 functions over 3 clusters would take approximately 21 hours to complete. This time would be 143 years when we have 40 functions instead of 30.

5.6.2 Experimental Results

We used a synthetic QDU graph generator as a test bench to investigate the quality of the solutions provided by the proposed clustering algorithm. In order to generate synthetic graphs and cost values, an efficient random number generator is utilized, which is based on the Ziggurat Method [139] implementing the algorithm presented in [140] for generating *Gamma* variables. The developed random number tool generates high-quality random numbers that are able to pass all the commonly-used tests for randomness [139]. In the implemented synthetic QDU graph generator, the user has the flexibility to define the number of nodes, the density[25] and the connectivity degree of the graph, and the range of the cost values for nodes and edges.

In our experiments, we selected two different scenarios for the number of functions, $n = 18$ and $n = 20$. In each scenario, the number of parts (k) is set to 4, 5, and 6. Furthermore, we set the density parameter to 75%, which means that the total number of edges in the generated graphs would lie between 70-80% of the total number of possible edges in its *fully-connected* counterpart graph. The connectivity of the graph is set to full, i.e. no isolated node exists in the generated graphs (the graph is connected). The ranking function for the clustering algorithm is defined in complete accordance with the cost functions defined in **Problem AP** (Section 5.3). This means that we consider all

[25] As discussed in Section 5.4.1, the QDU graph is generally very dense.

Table 5.2: *Total number of possible partitions in an exhaustive search of the solution space regarding different problem sizes.*

		$k = 4$	$k = 5$	$k = 6$
$n = 18$	**# partitions**	268,435,456	1,220,703,125	2,176,782,336
	5 %	13,421,773	61,035,156	108,839,117
$n = 20$	**# partitions**	4,294,967,296	30,517,578,125	78,364,164,096
	5 %	214,748,365	1,525,878,906	3,918,208,205

the three metrics exactly as defined in **AP** to calculate the final ranks of the candidate functions. No extra heuristic rule is applied to the ranking function. The final rank of a candidate function f, $R_t(f)$, for each examined cluster, is computed as follows:

$$R_t(f) = R_{exe}(f) + R_{cou}(f) + R_{com}(f), \quad (5.31)$$

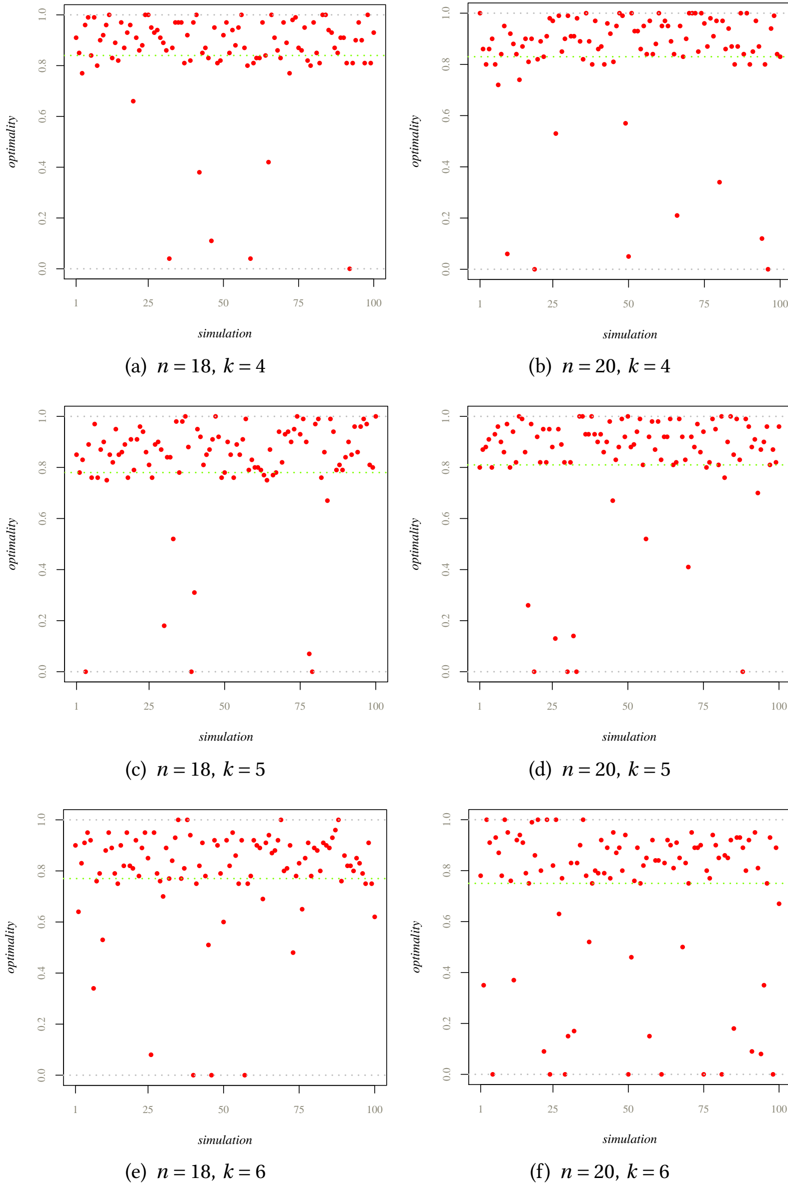
where $R_{exe}(f)$ denotes the *execution cost* rank of function f among all the candidates of the examined cluster. Similarly, $R_{cou}(f)$ and $R_{com}(f)$ denote, respectively, the *coupling degree* rank of function f regarding the functions already in the examined cluster, and $R_{com}(f)$, the *communication cost* rank of f with respect to all the other clusters in the partition. The values of all weight factors (such as σ_1 and σ_2 stated in Equation (5.1)) are considered as 1, so that there is an equal effect for all the three metrics.

We generate all the possible partitions and compare the optimality of the solution found by the proposed greedy approach to the optimal solution. In order to be fair in comparison, the initial functions selected by the clustering algorithm is marked and kept fixed for the exhaustive search as well. Since it does not make sense to compare the absolute cost value of the optimal partition with the cost of the solution in our algorithm, we only take the top 5% of the different partitions from the exhaustive search and find the standing of our heuristic solution among them. Table 5.2 depicts the number of all partitioning possibilities regarding different combinations for n and k. For each combination of the n and k, we have conducted 100 different experiments. The graphs depicted in Figure 5.7 summarizes the results. The optimality values of the found solutions are normalized within the interval [0..1]. An optimal value of 1.0 denotes that the found solution is the optimal partition, and 0.0 indicates that the found solution was not at all in the top 5% of the possible partitions[26]. The experimental results look promising, as in most cases, acceptable near-optimal solutions are found. Even though it is not feasible to evaluate the optimality of the solutions for large problem sizes, it is still a concrete proof of the effectiveness of the proposed heuristic algorithm.

5.7 MJPEG Case Study

To evaluate the proposed clustering algorithm in a real application, we used an implementation of the MJPEG algorithm. The application contains 33 functions. The QUAD toolset was used to profile the application and to identify the critical kernels. We opted to

[26] In this case, the optimality value is simply ignored though it may still be much better than the worst or the average partitions.



(a) $n = 18,\ k = 4$ (b) $n = 20,\ k = 4$

(c) $n = 18,\ k = 5$ (d) $n = 20,\ k = 5$

(e) $n = 18,\ k = 6$ (f) $n = 20,\ k = 6$

Figure 5.7: *Summary of experimental results for the synthetic data compared with the optimal partitions in different problem sizes, (a) n=18, k=4, (b) n=20, k=4, (c) n=18, k=5 , (d) n=20, k=5, (e) n=18, k=6, (f) n=20, k=6.*

partition the application into 4 clusters. Based on this assumption, the top contributing Discrete Cosine Transform (DCT) kernel (*DoubleReferenceDct1D*), which is responsible for nearly half of the whole execution time, was selected to form its own cluster. Other selected kernels were: *UseHuffman*, *ReferenceDct*, and *Quantize*. In a balanced partition-

Table 5.3: *Clusters in the MJPEG application.*

Cluster	Main Kernel	# Functions	Kernel's contribution	Cluster's contribution
1	*DoubleReferenceDct1D*	1	45.63%	45.63%
2	*UseHuffman*	5	19.50%	25.62%
3	*Quantize*	4	4.76%	9.01%
4	*ReferenceDct*	23	7.26%	19.74%

ing, an execution time contribution of about 25% is expected for each cluster. Hence, we defined this as the *stopping point* condition for the cluster growth. With this respect, the cluster corresponding to the top computationally-intensive kernel *DoubleReferenceDct1D* was immediately concluded with only one function, while other clusters grew gradually.

At the end of the partitioning, the *ReferenceDct* cluster stands as the largest cluster regarding the number of functions, yet its total contribution is not substantially high. The reason is that it contains all the I/O-related functions in the application with nearly no extra contribution to the whole execution time of the application. The *UseHuffman* cluster includes five functions with the contribution near the expected optimum value. Only the *Quantize* cluster gets notably less than the expected contribution share, mainly for two reasons. First, the *DoubleReferenceDct1D* cluster initially got approximately double of its share, hence, not much left to be equally distributed to the other clusters. Second, *Quantize* is limited in (or has no) data communication with the other clusters. This also prevented the clustering algorithm to lend more functions to *Quantize*, as they could fit better in *UseHuffman* and *ReferenceDct*.

The partitioning results are summarized in Table 5.3. An outline of the final partitioning of the QDU graph is also depicted in Figure 5.8. In order to make the figure less dense, we omitted some of the nominal functions, mostly performing I/O operations. As inferred from Figure 5.8, *UseHuffman* is correctly identified and isolated from the irrelevant *ReferenceDct* and *DoubleReferenceDct1D*. However, *ReferenceDct*, *DoubleReferenceDct1D*, and *Quantize* are somehow communicating sizable data. This was inevitable with respect to the partitioning parameters specified in the initial settings. It is worth to note that the solution found by the clustering algorithm indeed corresponds to the optimal solution.

5.8 Summary

In this chapter, we defined and formulated a general application partitioning problem, which can cover all the prior related partitioning models, including the specific case of HW/SW partitioning. Furthermore, we proposed a heuristic solution to the problem utilizing a greedy approach. As the primary objective of the partitioning, we considered the minimization of inter-cluster, the maximization of the intra-cluster, and the uniformity of the execution time distribution. The QDU graph, provided by the *Quad*-core tool, is utilized as the input data model to drive the partitioning. For synthetically generated graphs, experimental results showed that the heuristic algorithm is able to provide comparable solutions with optimal partitions. Additionally, we examined the case of a real application to further substantiates the efficiency of the proposed algorithm in practice.

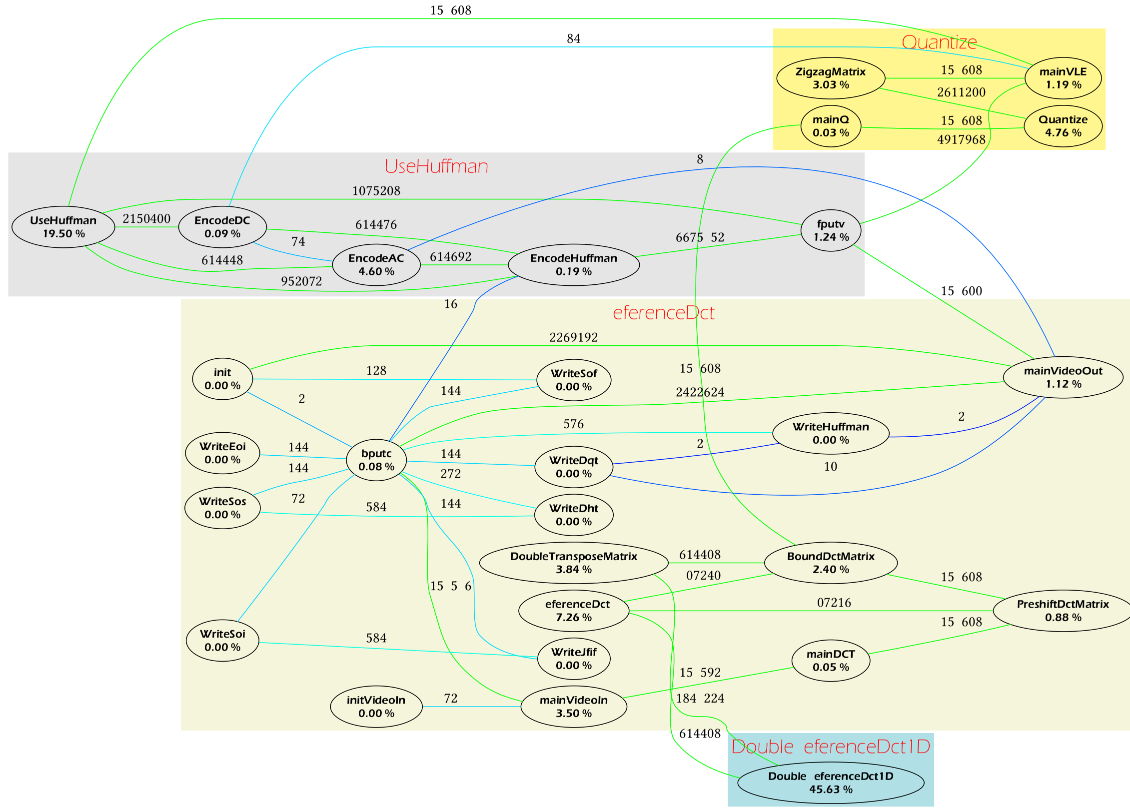


Figure 5.8: A partitioned QDU graph of the MJPEG application. The application is partitioned into four clusters. The execution time contribution of each function–as reported by MAIP–is also stated inside the nodes. The original QDU graph is converted into an undirected version. Some unimportant functions as well as all self-loops are removed from the graph. The final identified clusters, DoubleReferenceDct1D, UseHuffman, ReferenceDct, and Quantize, have total execution time contributions of 45.63, 25.62, 19.13, and 9.01, respectively.

Note.

The content of this chapter is partly based on the following article:

S. Arash Ostadzadeh, Roel Meeuws, Kamana Sigdel, and Koen Bertels, **A Multipurpose Clustering Algorithm for Task Partitioning in Multicore Reconfigurable Systems**, *Proceedings of International Workshop on Multi-Core Computing Systems (MuCoCoS'09)*, Fukuoka, Japan, March 2009.

CHAPTER 6

Utilizing Q² in HW/SW Partitioning: Case Studies

"An algorithm must be seen to be believed." †

— Donald E. Knuth

The QUAD *profiling information plays an important role in hardware/software co-design. To confirm its applicability in real scenarios, a partitioning approach is proposed which divides an application into hardware and software parts. Two realistic case studies are described: a well-known edge detector and a standard voice codec. These applications are analyzed using the Q^2 profiling framework and, subsequently, partitioned for the Molen heterogeneous reconfigurable platform. Detailed discussions and results are presented.*

HETEROGENEOUS system design can be subject to different types of constraints, including performance, power consumption, and cost. HardWare/SoftWare (HW/SW) co-design addresses the development of complex heterogeneous systems seeking the best trade offs among different solutions. The most critical task in this system-level co-design is *HW/SW partitioning*. The decisions made during partitioning directly influence the performance of the final implementation. Broadly speaking, HW/SW partitioning deals with the assignment of parts of a system description to heterogeneous implementation units: FPGAs or ASICs (hardware), standard or embedded processors (software), memories, and so forth [131].

Software implementation is generally used for *flexibility* and *feature-richness*, while hardware implementation is used for *performance*. The aim of the partitioning is to find a design implementation that fulfils all the specification requirements (functionality, goals, and constraints) at a minimum cost. In traditional design strategies, the system designer decides which blocks of the system could be implemented in hardware and which could be realized as software running on a standard processor, taking into

† *The Art of Computer Programming*, Vol. I, Fundamental Algorithms, Section 1.1, 1968.

account his/her own knowledge as an expert in the field. To automate such a difficult task, several algorithms and techniques have been developed in different co-design environments [105, 131, 136, 138, 194, 226]. *Most proposed approaches are tailored to work within their own environments. As a result, there are substantial inconsistencies between these approaches. Hence, a precise and fair comparison between them is nearly impossible.*

In this chapter, we present a heuristic-based approach to address the HW/SW partitioning problem for which gaining the optimal solution bears a very high cost [194]. Conventionally, performing an exhaustive search of the design space to find the optimal solution is unlikely to be practical even for a moderate problem size. We do not necessarily aim to provide the (theoretically) optimal solution, but a near-optimal practical solution in a reasonable amount of time. ***Our partitioning approach is driven by the QDU graph which makes it stand out among other heuristic approaches [17, 105, 131, 136, 138, 194] when put into practice. Irrespective of the heuristic rules utilized, HW/SW partitioning in the absence of the actual data dependencies at the coarse function-level fails to obtain effective partitions. In real scenarios, the performance gained by hardware execution can be adversely affected by the cost of data communication between hardware and software parts. A function-merging strategy is proposed that aims to reduce or even eliminate the data communication between hardware and software parts.Driven by accurate actual data dependencies between functions, this strategy aims to merge tightly-coupled functions, which heavily communicate with hardware kernels, in order to internalize the communication.*** To demonstrate the applicability of our approach, we utilize the profiling information extracted by the Q² profiling framework to map real applications onto the Molen heterogeneous reconfigurable platform. For this purpose, two applications have been selected, one from the image processing domain and the other from the speech processing domain.

The remainder of this chapter is organized as follows. Section 6.1 establishes the background for HW/SW partitioning, which is a crucial step in porting sequential applications into heterogeneous reconfigurable platforms. Our proposed HW/SW partitioning methodology is presented in Section 6.2. In Section 6.3 and Section 6.4, we present detailed descriptions and analyses of two applications to demonstrate the potential and the applicability of the Q² profiling framework in HW/SW partitioning. Finally, Section 6.5 summarizes this chapter.

6.1 HW/SW Partitioning

HW/SW partitioning is the problem of dividing an application's computations into a part that executes as sequential instructions on a (general-purpose) microprocessor (the "software") and a part that runs as parallel circuits on some Integrated Circuit (IC) fabric like an FPGA or an ASIC (the "hardware"), so as to achieve design goals like performance, power, size, and cost [204]. Commonly, the hardware part of the application is executed on PEs that act as coprocessors for the microprocessor, as seen in the Molen machine organization (see Section 3.1.1).

As an example, a brute-force DES cracking application [60] to analyze the entire DES 56-bit keyspace may be partitioned in such a way that the iterative block handling

operations are executed on a microprocessor, while the actual decrypting of blocks is offloaded to hardware [100]. Block decryption is performed to find keys that decrypt some portions of data into sequences of ASCII numbers. This technique is often used for recovering the keys of encrypted files containing known types of data, for example encrypted documents that include financial or military information.

DES cracking applications running on current-generation CPU cores can process approximately 16 million DES key operations per second. A modern Graphical Processing Unit (GPU) card such as the NVIDIA Tesla® [196] can handle more than 15× that number, or approximately 250 million DES operations per second. When using an FPGA cluster[1] based on a single off-the-shelf motherboard, each FPGA is able to perform 1.6 billion DES operations per second. Using the cluster, the highest-known benchmark speed was achieved for the 56-bit DES decryption, with the throughput exceeding 280 billion keys per second. This means that a key recovery that would take years to perform on a PC, even with GPU acceleration, could be accomplished in less than three days on an FPGA cluster [100].

The increase in the performance of hardware execution is attributed to the circuits that execute some computations thousands of times faster than software sequential instructions. This is largely because of the parallel nature of hardware circuits. As an example, if an instruction contains 100 multiplications of independent data items, a general microprocessor would have to execute them one (or a few) at a time, thus, requiring hundreds of clock cycles. This is not the case with a dedicated hardware circuit, which can potentially execute all of them in parallel using 100 multipliers, requiring only one or few clock cycles. Moreover, such execution results in energy efficiency.

The existing concept of HW/SW partitioning appeared in the early 90s, shortly after the advent of High-Level Synthesis (HLS) in the late 1980s [93, 205, 223]. During the early days of computing, the designers had to manually perform such partitioning. Despite the fact that the idea has been around for two decades, HW/SW partitioning that involves some degree of automation, informally called *automated HW/SW partitioning*, has gained popularity in the last decade due to substantial advancements in the semiconductor technology and the emergence of Programmable Logic Devices (PLDs) as commodity processors.

While formal compilers convert program source code into sequential machine instructions, HLS shall automatically analyzes, architecturally constrains, and schedules source code to create a register transfer level HDL. It is then synthesized to the gate level by the use of a logic synthesis tool. With both HLS and compilers, the ultimate goal is to write a single program (or "executable specification") from which both instructions and circuits could be *automatically* generated. However, this is not fully achieved yet.

Today, HW/SW partitioning encompasses not only dividing the computations per se, but also involves utilizing some well-established concepts, such as application profiling, developing target architectures with efficient communication, microprocessor and circuits simulation, system debugging, developing customized tools and languages, and so on [205, 223]. Due to the heterogeneity of target architectures, different means of data communication exist between the microprocessor and coprocessors (FPGA, DSP, etc.).

[1] The FPGA cluster includes an SC4 containing 176 Spartan3 FPGA devices, housed in a single 4U chassis drawing under 1400 W.

Loosely coupled coprocessors may communicate with the microprocessor via a shared memory or DMA, while more tightly coupled coprocessors may have more equal access to a shared memory and the system bus. In an Application Specific Instruction-set Processor (ASIP), the coprocessor circuit is integrated directly into a microprocessor's datapath, where partitioning occurs during compilation. On the contrary, partitioning can generate processing circuits that execute as peers to microprocessors rather than as coprocessors. Different target architectures require tools that partition at different levels of granularity, including the instruction level (for the ASIP scenario) and the loop or subroutine level (for coprocessor scenarios).

The advent of FPGAs has made HW/SW partitioning even more relevant and important in the field of computing [223]. There is a resemblance between the way FPGAs implement circuits and software. Synthesis tools generate bitstreams, just like compilers generate binaries, and the bitstream is fed into the FPGA's program memory, just like a binary is fed into a microprocessor's program memory. FPGAs have started to appear alongside with microprocessors for computing purposes on nearly all types of platforms, including desktop computers (e.g., Intel's QuickAssist [171], and AMD's Opteron [159]), supercomputers (e.g., SGI's Altix [7]), and even in mobile and handheld devices [121]. As such, HW/SW partitioning is addressed extensively by the reconfigurable computing community. This includes the development of compilers targeting FPGAs, handling key obstacles degrading performance, such as the memory bottleneck problem, and applying technological advancements, such as smart buffers that actively fetch and maintain the data required by coprocessors [92].

In fact, in view of FPGAs implementing circuits as software, the term HW/SW partitioning seems to be misleading today [203]. *Instruction/circuit* partitioning looks like more appropriate, with the output of partitioning being the software intended for microprocessors and FPGAs. Partitioning may eventually be considered just a step during compilation, along with existing steps, such as code parsing and code generation. Furthermore, just as instruction software today is commonly translated JIT by computing platforms from one instruction set to another (e.g., Java bytecode JIT-compiled to a native instruction set, or x86 code JIT-compiled to a VLIW instruction set), it is feasible to JIT-partition instruction software to circuit software, a process known as *warp processing* [203, 210].

The rest of this section is organized as follows. In Section 6.1.1 and Section 6.1.2, we discuss the main objectives of the integrated HW/SW co-design in the context of reconfigurable systems, as well as the important challenges which the users of these systems face to make efficient use of their potential. In Section 6.1.3, we briefly survey some relevant HW/SW partitioning approaches that utilize profiling information in order to drive the partitioning process.

6.1.1 HW/SW Co-design: Research Directions and Challenges

HW/SW co-design is a multi-disciplinary research theme. The study of computing systems' architectures directs performance and/or energy consumption analysis of CPUs or multiprocessors; the theory of real-time systems drives the analysis of deadline-driven applications, while computer-aided design (customized tools) helps in evaluating the hardware and/or software cost, as well as, provides methods for DSE. The intrinsic di-

versity in this research area, in turn, brings about the fundamental problem of HW/SW partitioning to be driven by various objectives.

Co-synthesis was basically established to target FPGA-based platforms. Nevertheless, several critical challenges should be addressed before co-synthesis becomes a commonplace approach for FPGA design. In general, the efficient use of an FPGA platform requires identifying applications that map well onto it. Nowadays, many applications require distributed computations to deal with fast response rates. As the most important requirement, the PEs in these systems should be integrated in a way to ensure that they jointly satisfy the application's performance requirements.

Conventionally, a HW/SW partition is determined a priori (and is adhered to as much as possible) because any changes in this partition may necessitate extensive redesign. Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet performance constraints. This will not always result in efficient designs. In general, at a high level abstract, major problems with the HW/SW co-design are:

- Lack of a unified HW-SW representation, which leads to difficulties in verifying the entire system, and hence to incompatibilities across the HW/SW boundary;
- A priori definition of partitions, which can lead to sub-optimal designs;
- Lack of a well-defined exact design flow, which can make the design process complicated and revisions difficult.

A key problem that remains, as always, is the communication between the GPP and the coprocessor(s). Several delay sources can annul any performance gain achieved by accelerating computationally-intensive parts on these coprocessors. These include *communication delay*, which is due to architectural limitations, and *tasks' synchronization delay*. An appropriate application for these kind of platforms would usually have intense computations that can be moved to coprocessor(s) ensuring rather small communication interactions with the GPP, enabling efficient work overlapping.

Creating suitable interfaces for both the FPGA fabric and GPP sides of the system is indeed indispensable. On the GPP side, drivers are required to turn software operations into instruction sets for the hardware. On the FPGA fabric side, interfaces to manage the interconnect systems should be built. Until now, no appropriate unified language is established as a standard to describe HW/SW partitioning specifications. Software languages, like C, bias the implementation in favor of software, while hardware languages, like Verilog, bias results towards hardware. An alternative might be to describe the system in two languages: describe some obvious hardware functions in a HDL and describe the rest of the functions in a software language. As a result, when operations are moved across the partition, only a relatively small part of the total specification is translated.

Currently, system design, software design, and semiconductor design are converging in order to address the increasing challenges to create complex ICs and SoCs. This convergence has brought to the forefront the need for organizations to facilitate the creation of system-level, semiconductor design, and verification standards. Leading industry standards associations Accellera and Open SystemC Initiative (OSCI) merged recently in 2011 to form a single organization, called *Accellera Systems Initiative* [2], to

address the needs of the system and semiconductor designers. With the recent addition of Transaction-Level Modeling (TLM) to the IEEE-1666 SystemC Standard, the design community has expressed interest in exploring technical synergy with the Universal Verification Methodology (UVM) and other languages, as well as, a more transparent flow from SystemC to the standard unified Hardware Description and Verification Language (HDVL) SystemVerilog [193]. System-Level Design Languages (SLDLs), such as SystemC and SpecC [182], must consider computational models, system design methodologies, simulation, and many other factors to be widely adopted in electronics industry.

Despite considerable effort to model joint hardware and software systems, the goal still remains as one of the most difficult challenges. System-level performance analysis is a complex problem that must be studied under a variety of operating conditions suitable for various application types. Furthermore, researchers have already set off to investigate more advanced methods, such as heuristics and genetic algorithms, to apply in the DSE in HW/SW co-design.

On the hardware side, memory systems still continue to be a prominent subject of research, since their design profoundly influences the performance of the system and the energy consumption [119]. Cache models are one aspect of particular importance in understanding memory systems. The better the cache model, the easier it is to predict how changes to hardware or software influence system performance and power. Software optimizations let designers to implement programs in the best possible way on the available architectures. With cache synthesis, it becomes possible to choose a cache configuration for a particular application. Several alternatives to traditional cache models have appeared in literature, including *scratch-pad* memory [223] and *Connected RAM* (CoRAM) [54]. System-level power management [31] is well suited to co-design because designers can use the characteristics of an application to optimize the management strategy and its implementation in hardware and software. Given the prime importance of power management in digital systems, more work is expected in this area in the future.

6.1.2 HW/SW Co-design Objectives

HW/SW partitioning can be performed in various directions in order to fulfil different goals. The way the partitioning process is performed can considerably influence the behaviour of the whole system. Figure 6.1 depicts a schematic overview of major objectives which can direct the partitioning process. In the following, we describe these objectives:

Performance One of the main goals of partitioning is to improve the performance of the system in terms of the execution time. In order to achieve this goal, several parameters have to be taken into account, most importantly, the intra- and inter-communication costs in the hardware and software partitions, the internal degree of parallelism in each partition, as well as the overall parallelism of the synthesized system. As the primary consideration, designers opt to map *critical* parts[2] of the application to the hardware partition, and execute the rest as software. By *critical*, we mean that the execution latency is high, mostly, but not necessarily, because of computational intensiveness. In other words, the execution time contribution

[2] Part refers to process, task, subroutine, loop, basic blocks, and in general, any block(s) of statements.

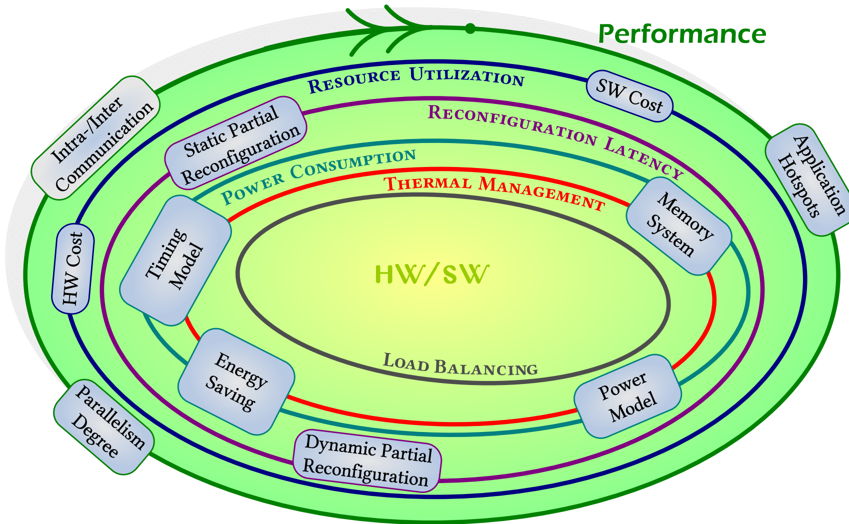


Figure 6.1: *HW/SW co-design objectives.*

of that part compared to the whole execution time of the application is substantial, hence, an accelerated run of that part tends to be a considerable time saver for the application execution. Alternatively, minimizing the data communication between hardware and software partitions plays an influential role in achieving overall performance gain. On the other hand, parallel execution can be another fundamental concept to boost the performance of the system. In this respect, parallelism is usually considered at three different levels:

- internal parallelism of each hardware task (during high-level synthesis, operations are scheduled to be executed in parallel by the available CCUs);
- parallelism between hardware tasks;
- parallelism between the CCUs running the hardware tasks and the GPPs executing the software tasks.

Resource Utilization Efficiently utilizing the available resources is another key objective in HW/SW partitioning. This becomes more severe in case of hardware resources, as they are more limited and costly. Taking into account the limited hardware area available on FPGAs, designers have to find a trade off between the performance and hardware cost. This may mean that not all critical tasks can be placed on the reconfigurable fabric to be accelerated. Hence, by an appropriate selection, only those tasks which are considered to be more beneficial are mapped to the hardware and the rest has to be accommodated in the software part.

Power Consumption Undoubtedly, power consumption has become one of the major design concerns in today's systems. A designer must guarantee that his design does not exceed the power constraints of a target platform. In addition, due to the proliferation of portable and battery-dependent devices, low-energy consumption

has become one of the key features for the success of a design. HW/SW partitioning can drastically affect the power and energy consumption in a system. It could be that a particular partitioning, not necessarily the one that provides the fastest execution, can save energy while still meeting performance constraints. Regarding energy consumption, an exceptional attention has to be given to the buses and the memory system. Evaluating the impact of the memory system in the overall energy consumption is only possible if accurate timing and power models are available, which is not often the case. Ideally, the hardware vendors should provide these models or at least some estimates. Even having accurate models would not suffice, because data dependencies between the tasks and accesses to the shared resources should also be accurately analyzed which is indeed not trivial.

Thermal Management A tight correlation can exist between temperature and application behavior. Adapting to each task's thermal demands can optimize system performance, while keeping the PEs thermally saturated [112]. Thermal monitoring and response management is a critical part of systems that incorporate multiple PEs. As a core requirement, the manager senses the impact of temperature on various components in a multiprocessor environment, and subsequently responds by configuring the system in such a way that maximum performance is achieved without exceeding thermal boundaries. When the heat removal ability is less than the heat dissipation requirement, thermal runaway happens, which introduces vital concerns for the system performance and reliability [99]. Generally, in a temperature-aware design, reducing the power consumption, i.e. heat generation at the source, is the main plan to avoid thermal runaway. To achieve this goal, the designer must make sure that temperature hotspots are avoided when the computational load is divided among the PEs.

Reconfiguration Latency Current dynamic reconfigurable systems allow the mapping of large tasks that exceed the available area on the hardware. To achieve this, the reconfigurable device should support *partial* reconfigurability, so that the distinct parts of hardware tasks[3] are mapped onto the hardware in different time intervals. Partial reconfiguration allows the continuous running of critical parts of hardware tasks while a controller loads a partial design into a reconfigurable module. It is useful for hardware space saving by only storing the partial designs that change between several designs. In this case, one of the key objectives for the partitioning is the minimization of reconfiguration latency. For static partial reconfiguration, the reconfigurable device is not active during the reconfiguration process. This means while partial reconfiguration data is sent into the FPGA, the rest of the device is stopped and brought up after the reconfiguration is completed. The partial reconfiguration bitstream contains only information about the differences between the current design structure (that resides in the FPGA) and the new content of the FPGA. As a result, the partitioning decision has a huge impact on the overall system performance.

Load Balancing Partitioning an application workload among several PEs can be biased towards unfairness regarding the amount of work assigned to each PE. The designer can make sure that PEs which have the same or similar capabilities get

[3] These parts are called *configurations* in literature.

more or less comparable workloads. This objective normally lends itself to the same direction as resource utilization, particularly in the cases where multiple PEs exist in software or hardware domains. Load balancing does not always mean that everyone should get an equal amount of work. Based on the differences in the processing nature of PEs, ensuring that each PE gets the task for which it is intended makes a balanced (fair) distribution of workload. Improper application partitioning can result in resource waste (idle processing capacity and inefficiency) which can severely affect the performance of a system.

The objectives defined for a partitioning problem are often conflicting. Therefore, it is unlikely that each one by itself solves the problem. For example, an objective function that minimizes the execution time drives the solution towards feasibility from the viewpoint of deadline constraint or performance gain. This solution is likely to be suboptimal regarding the resource utilization (increased hardware area cost). On the other hand, if the application is partitioned in such a way that hardware area consumption is minimized, the solution is quite unlikely to meet performance constraints. As a result, a fixed objective function is most probably incapable of solving the partitioning problem.

Application partitioning in general, and HW/SW partitioning in particular, is considered as a constrained optimization problem [105, 131, 194]. Traditional approaches, such as ILP, or more advanced heuristic and evolutionary techniques can be used to solve such multi-objective optimization problems. However, there can always be limitations and inadequacies which prevent guaranteeing the optimality of a solution, if any reached at all.

6.1.3 Profile-guided HW/SW Partitioning

The widespread utilization of heterogeneous reconfigurable systems relies on the availability of appropriate tools to assist developers in mapping existing applications onto these systems by reducing the time and effort, as well as efficiently exploiting the provided flexibility. As discussed earlier in Section 5.1, the HW/SW partitioning process can be carried out based on different levels of granularity, such as, loops or functions. *Profiling*, the process of monitoring an application to spotlight specific code region(s) that intensely use(s) up resources, is an essential preliminary step within the design process for many software and hardware systems.

HW/SW partitioning has been an active field of research in the last decades. Many different approaches have been proposed to solve the problem [105, 131, 136, 138, 194, 226]. For a concise survey of related work regarding HW/SW partitioning methods, see Section 5.2. Because the main focus of the work in this thesis is on profiling, in the following, we briefly describe several related practical works that consider partitioning based on profiling data.

In [128], a HW/SW partitioning approach is proposed for dynamically reconfigurable architectures consisting of a GPP and an FPGA. The main focus of the authors is on the temporal aspect of an application not on the spatial one. The partitioning is carried out at a fine-grained level, i.e. at loop and basic block level, with the goal of optimizing the total application execution time, which includes the software and hardware execution times, communication time and datapath reconfiguration time. In the compilation

flow, the source code of an application (in C) is preprocessed to extract loops as hardware candidates. Furthermore, multiple optimized versions of the loops are created by compiler transformations. Each extracted loop candidate is profiled to estimate the total software time, execution frequency, memory bandwidth requirement and the trace behavior. A quick synthesis is also performed to estimate the delay and the area needed for the hardware implementations. The loop entry trace profiling is used to find out the exact runtime sequence of all hardware candidate loops. The details of the various profiling processes are not presented. In the end, the extracted information are fed to the HW/SW partitioner to decide which loops should go to the hardware, and which versions of the loops should be used.

A HW/SW partitioning and code generation flow for reconfigurable platforms is presented in [20]. Given the C code of a target application, the user has to manually identify and tag computation blocks in the source code to be extracted for implementation on FPGA. Since in their target architecture, the reconfigurable array is part of the GPP's datapath, there is a severe restriction of potential code that can be moved to the hardware. Kernel candidates can only contain small computation blocks with few inputs and outputs. After that, a simulator evaluates the code using the cycle counts specified by the user for the tagged blocks. A profiler returns the number of cycles used to execute each line of code. To estimate the cycle count of the FPGA code blocks, the authors use heuristics to have a quick performance evaluation. As a result, the estimation may not be accurate enough. The partitioning problem is addressed by exploring different HW/SW trade offs based on the performance profiles, with the objective of maximizing the overall performance, while satisfying FPGA mapping size constraints. This objective is formulated as a boolean programming problem.

Santambrogio et al. [175] proposed a methodology based on the Adaptive Programming technique to evaluate and subsequently perform the HW/SW partitioning for a SoC that employs dynamically reconfigurable hardware and software programmable cores. They developed quantitative evaluation metrics to determine the reconfigurable system performance and to represent the performance of software in a SoC from an application-specific, input-oriented point of view. A performance model is built with the associated evaluation metric to identify application specific input behavior of software modules. This general performance model is then embedded along with hardware performance models to yield a flexible mean to evaluate the performance impact of different partitioning and allocation decisions. A profiler enhanced by implementing *Adaptive Metrics* is used to reveal the potential in functions for performance improvement as a result of transformation into an adaptive form.

Apart from the traditional partitioning methods, different heuristic and evolutionary methods are also investigated to solve this problem. In [217], a heuristic searching approach is presented based on the Ant Colony Optimization (ACO) algorithm. Both global and local heuristics are combined in a stochastic decision making process to effectively explore the search space. As authors state, profiling information can also be utilized in the decision strategies to assign tasks to different resources. However, no reported study exists for such a work.

In [87], a design methodology for application partitioning is presented, which targets a reconfigurable Multi-Processor System on Chip (MPSoC). The methodology supports the partitioning of an application between several processing elements (SW/SW parti-

tioning) at the function level, as well as, HW/SW partitioning. A combination of a dynamic profiler (*CodeAnalyst* [56]), and a developed tracing tool, is utilized together with manual code analysis to analyze the data communication between functions. The parameters used for the partitioning decision are extracted from the results obtained in the code analysis step, such as, the execution contribution of each function, the call graph, and the communication graph. For the SW/SW partitioning, hierarchical clustering is utilized, which is based on heuristics and, thus, it is faster than ILP. The partitioning algorithm can consider some critical issues, such as, workload balancing and minimal inter-processor communication. A tool is also developed to analyze the output of the detailed *CodeAnalyst* profile to automatically calculate the block and loop nesting, the timing of the loops and functions, the function affiliation of the loops and the calculation of the threshold, which defines the hotspots in code fragments. This information can be used for HW/SW partitioning in systems incorporating reconfigurable fabrics.

Although [87] is similar to our proposed approach with respect to addressing the application partitioning on the coarse function level, the way we address the application partitioning is somehow different. *In particular, we utilize the QDU graph, instead of the commonly used call graph, as the primary reference for the partitioning process.* The data communication between functions in the application is extracted *automatically* by our advanced profiling framework. This process is performed manually in other relevant works. In cases where the application source code is complex, such as the case studies presented in this chapter, manual processing of data communication is very tedious, or even impossible to quantify data and make a conclusion.

Furthermore, application partitioning driven by call graph does not necessarily result in efficient clusters, specially when tight bindings exist between those functions that do not exhibit direct calling links. This phenomena is investigated in detail in the case studies. As the size of available area on FPGAs increases, it is no longer a restriction nor efficient in terms of performance, to map small code segments of an application onto the hardware. *This means that HW/SW partitioning based on finer granularity levels, despite being in the center of attention so far, will not be the focus of research work in future reconfigurable systems.* As a result, mapping a complete function or, in general, a combination of several coupled functions, will not be elusive anymore. *In our partitioning methodology, we initiate the concept of function merging based on accurate profiling information. Merging tightly communicating functions as one not only presents a solid view on a whole task but also allows developers to perform optimizations, particularly for memory requirements, in a feasible and efficient way.*

6.2 The Q^2 Partitioning Methodology

Traditionally, the primary objective of HW/SW partitioning in heterogeneous reconfigurable systems is to improve the execution performance, hence, to gain speedup. In order to achieve this goal, the program parts with higher execution time contributions[4] are usually mapped to the hardware, while the parts with lower contributions are exe-

[4] By *program part*, we mean, in general, any collection of instructions that forms a computational task. It can refer to a very small function in an application source code, which, due to numerous calls, contributes substantially to the whole execution time of the application.

cuted on GPPs. Nevertheless, it has turned out that the key hurdle to limit the speedup is the well-known *memory bottleneck* problem. The problem is even intensified in case the potential hardware kernel candidates are heavily communicating with the kernels in the software part [221, 228].

To address this critical problem, we utilize the profiling information provided by the *Q^2 profiling framework* to minimize, as much as possible, the data communication between the hardware and the software boundary. This is mainly achieved by *merging* tightly coupled kernels, respecting the Molen machine restrictions, and kernel *splitting*, in case the hardware area limitation prevails. We have applied our partitioning strategy, in particular, the function-merging approach, on several real applications. At this point in time, parts of the partitioning analysis are performed *manually*. Hence, the investigation is not exhaustive and a limited set of merging possibilities is considered based on the user expertise. It should be noted that the actual application source code modification to merge functions is *essentially* a manual process, due to the required human intelligence in performing a sophisticated merging other than simple inlining.

The existing partitioning approaches that are carried out at the coarse-grained function level by utilizing profiling data, commonly, take advantage of some kind of CDFGs, e.g., a call graph, as the primary reference for the partitioning process. The way we investigate HW/SW partitioning is in some way different from these approaches, mainly because the detailed profiling data extracted by our profiling framework has not been considered in existing approaches. To be more precise, we base our partitioning methodology on the QDU graph, which is used as the main reference to reveal the actual *coupling degree* between the functions in an application.

Application partitioning based on some kind of conventional CDFGs — which is the only practiced strategy up to now — will not yield proper and efficient partitions in case of reconfigurable systems. This is due to the fact that these graphs do not contain the necessary information to correctly guide the developers in order to identify related functions. By related functions, we mean functions that have tight data communication bindings. Normally these functions are separated in the implementation stage of an algorithm to support and maintain the modularity of the application. Nevertheless, this separation is the main threat to the efficiency of application mapping in heterogeneous reconfigurable systems. As an example, there might be a very strong relationship between a pair of functions that do not exhibit any direct calling link, when the call graph for the corresponding application is examined. It can be quite misleading to identify related functions, if the application is reasonably large in scope (see Section 6.4.3 for a detailed example).

The problem with mapping small functions to the hardware is the marginal potential for application speedup, which severely limits the efficiency of the partitioning and, hence, mapping process. As a result, merging the tightly coupled functions to improve the performance of mapping applications to reconfigurable systems turns out not to be only an option, but a necessity. Furthermore, with the continuous increase in the size of available area on FPGAs, mapping larger and larger code segments of an application onto the hardware becomes feasible. The obvious implication would be that course-grained partitioning approaches will gain more popularity in future heterogeneous systems using different PEs.

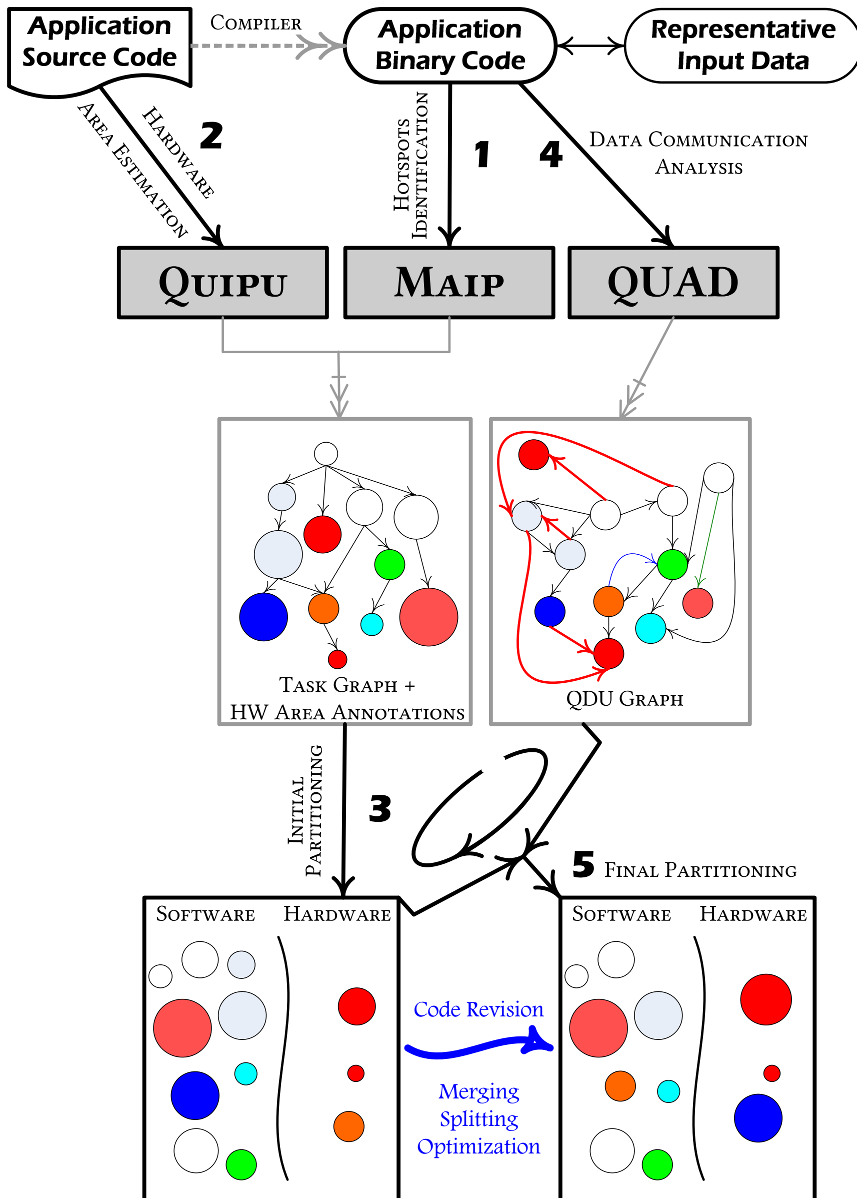


Figure 6.2: The Q² partitioning approach. The five steps defined in the Q² HW/SW partitioning algorithm are specified with bold numbers in the figure.

Figure 6.2 depicts the major steps taken in our proposed HW/SW partitioning approach utilizing the extracted information provided by the *Q² profiling framework*. In summary, the following partitioning methodology is defined to partition the tasks in an application:

- ***Hotspots Identification*** - The *Maip* determines the application hotspots. Although *Maip* extracts a rich set of profiling information from an application, which is partly related to its memory accesses, here, we consider the execution time as the main parameter to determine hotspots in the application. Nevertheless, using any other parameter does not interfere with the proposed partitioning methodology.

- ***Hardware Area Estimation*** - All functions in the application are annotated with FPGA hardware area estimates, as predicted by the respective *Quipu* models (see Section 3.2.1).

- ***Initial Partitioning*** - With the knowledge of the application hotspots and the respective area predictions, an initial partitioning is determined. In this respect, as many computationally-intensive kernels as possible are moved to the hardware, so as to speed up their execution, while satisfying the area constraints.

- ***Data Communication Analysis*** - The data communication of the kernels in the initial partitioning set is then analyzed using *Quad*. Because the set of functions that is analyzed has been reduced, *Quad* can run much faster. Additionally, it helps the developer to focus on the main data communication bottlenecks.

- ***Final Partitioning*** - Certain kernels in the initial partitioning can still heavily communicate with other functions in software, implying a heavy communication overhead. Therefore, an additional set of kernels may be moved to hardware, if possible, so as to reduce the amount of data communicated between hardware and software. Furthermore, based on the extracted profiling information, source code revisions and optimizations may also be carried out. These include depriving part of a kernel to adhere to restrictions, merging tightly-coupled functions, and optimizing memory accesses.

Dividing the functions into two parts per se does not guarantee its applicability nor its effectivity in practice. As discussed earlier, a critical obstacle remains the data communication overhead, which can neutralize or even aggravate the performance resulted from accelerated hardware execution. What makes our partitioning approach different from others, is the fact that the partitioning is primarily based on data communication, i.e. communication-aware partitioning. This means that the focus of our approach is on tackling the main obstacle which limits the performance of the mapped application. Contrary to the existing theoretical HW/SW partitioning approaches [17, 131, 194], in this chapter, we examine the proposed HW/SW partitioning methodology in practice to substantiate its feasibility in a real environment. In the following case studies, we specifically aim to demonstrate the following qualities of the Q² partitioning methodology:

- comprehensive application analysis regarding the extracted profiling information,

- performing various memory access related optimizations derived by thorough inspection of the profiling information,
- examining the effects of different application modifications regarding possible performance improvements,
- preparing executable versions of the applications which can run on the Molen platform with respect to the restrictions described previously.

A fully automated partitioning process is still difficult to grasp and implement. This is because each application has its own characteristics and complexities, hence, it requires human intelligence to handle some tasks. This can be time-consuming, particularly the source code inspection for performing efficient function merging.

6.3 Canny Edge Detection

In this section, we present a real case of an image processing application, Canny Edge Detection (CED) [48], to demonstrate the potentials of the advanced tools developed in the Q^2 profiling framework. It serves as the direct approach to validate the usefulness, efficiency, and applicability of the profiling tools and the proposed partitioning methodology. *The main objective is to map the application onto the Molen heterogeneous reconfigurable platform. To accomplish this objective, we shall have a comprehensive understanding of the application behavior, in particular of its memory access behavior and requirements.* Although the focus of the analysis is on the runtime attributes, we also examine the application source code to extract some valuable information. The result of this detailed profiling is primarily utilized to spot bottlenecks and deficiencies related to the application memory usage. Furthermore, it provides hints for code revisions in order to improve the execution performance. Throughout the experimental analysis, we conduct several phases of source code optimizations as a means of verification of the extracted profiling information.

In the following, we first describe the CED algorithm in Section 6.3.1. Then, in Section 6.3.2, we present the experimental setup used for the analysis of the application. The results of the different profiling data analysis are discussed in Section 6.3.3. By inspecting the profiling information, some source code optimizations are performed on the application, and their effects are investigated. Finally, in Section 6.3.4, we summarize our observations and results.

6.3.1 Edge Detection Overview

Canny [48] is a well-known edge detection algorithm, which aims to achieve the following three principal goals:

- *good detection* - the detection of as many of the real edges as possible, while also not falsely detecting non-existing edges as much as possible.

- *good localization* - the detected edges are as close as possible to the actual edges, i.e. the distance between the edge pixels as found by the detector and the actual edge is to be minimal.

- *unique detection of edges* - real edges should be detected only once. This aspect of the detection method was added because the previous criteria do not imply that edges are only identified once.

Based on these criteria, the Canny edge detector first smoothes the image to eliminate any noise. It then finds the image gradient to highlight regions with high spatial derivatives. The algorithm then tracks along these regions and suppresses any pixel that is not at the maximum (non-maximal suppression). The gradient array is then further reduced by hysteresis, which tracks along the remaining pixels that have not been suppressed so far. Hysteresis uses two thresholds to accomplish this task. If the magnitude is below the first threshold, it is set to zero (made a non-edge). If the magnitude is above the high threshold, it is made an edge. In case the magnitude is between the two thresholds, then it is set to zero, unless there is a path from this pixel to a pixel with a gradient above the second threshold.

For our experiments, we have used the implementation provided by the Computer Vision Laboratory at the University of South Florida [200]. This CED application has the following steps:

- **Step 1.** *Filtering out any noise in the original image.* A Gaussian filter is used exclusively due to its simplicity. Once a suitable mask has been calculated, the Gaussian smoothing can be performed using standard convolution methods. A convolution mask is usually much smaller than the actual image. As a result, the mask is slid over the image, manipulating a square of pixels at a time. The larger the width of the Gaussian mask, the lower is the detector's sensitivity to noise. The localization error in the detected edges also increases slightly as the Gaussian width is increased. The width of the Gaussian mask used in the implementation is determined based on the standard deviation of the Gaussian smoothing filter that should be input by the user.

- **Step 2.** *Finding the edge strength.* This is done by taking the gradient of the image. The Sobel operator performs a 2D spatial gradient measurement on an image. Then, the approximate absolute gradient magnitude (edge strength) at each point is found. The Sobel operator uses a pair of 3×3 convolution masks, one estimating the gradient in the x-direction and the other estimating the gradient in the y-direction.

- **Step 3.** *Applying non-maximal suppression.* After finding the edge directions using the gradient values, non-maximal suppression is used to trace along the edges and suppress any pixel value that is not considered to be an edge. This will result in a thin line in the output image.

- **Step 4.** *Performing hysteresis.* Hysteresis is used to eliminate the breaking up of an edge contour caused by the edge pixels fluctuating above and below a threshold. Thresholding with hysteresis requires a low and a high threshold. Making the

assumption that important edges should be along continuous curves in the image allows to follow a faint section of a given line and to discard a few noisy pixels that do not constitute a line but have produced large gradients. The hysteresis process begins by applying the high threshold. This marks out the edges that are fairly guaranteed to be genuine. Starting from these edges, and by using the directional information derived earlier, all the edges can be traced through the image. While tracing an edge, the lower threshold is applied, which allows to trace faint sections of edges as long as a starting point is found. The low and high threshold values for hysteresis should be specified as input parameters by a user.

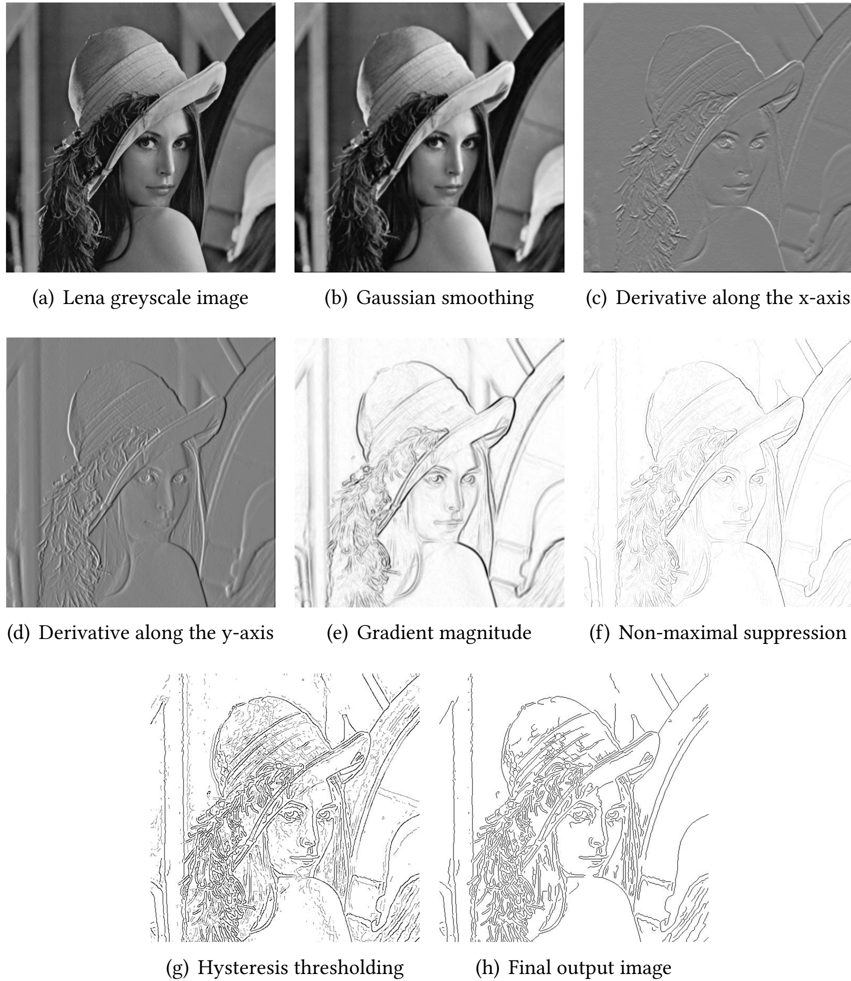
Figure 6.3 demonstrates the result of applying consecutive phases of the CED algorithm on the standard greyscale *Lena* photo. The standard deviation of the Gaussian filter is set to 1. The low and high thresholding values for hysteresis are set to 0.4 and 0.6, respectively.

6.3.2 Experimental Setup

The used CED implementation consists of three source files containing fifteen functions in total. For the experiments, we used a sample Portable GrayMap (PGM) image with a resolution of 800×600 pixels and 8 bpp. The standard deviation of the Gaussian filter was set to 2.0. The values of low and high thresholds for hysteresis were both set to 0.5.

The partitioning of the CED application is intended for the Molen Polymorphic Processor [212], which is an implementation of the MAL (see Section 3.1.1). The particular version that we used was implemented on a Xilinx Virtex-5 FX 200T FPGA with 2 MB Block RAM (BRAM) and 30720 slices. Due to the platform restrictions, at most 256 KB of BRAM can be used for each CCU. We used the *QUAD* toolset on an Intel 32-bit Core2 Duo E8500 @3.16 GHz, running Linux kernel v2.6.34. The *GNU* profiler, *gprof*, is also used on the same machine to get the call graph of the application.

The *CED* application source code was compiled with *gcc* v4.5.0, using level two (*-O2*) optimizations and without function inlining. We also utilized *gprof* on the embedded PowerPC (PPC) 440 @400 MHz with 512 MB DRAM, which is integrated in a Xilinx ML510, Virtex5 FX 200T with 2.0 MB BRAM FPGA board. In order to profile the application using the *QUAD* toolset, *Pin* [134] DBI framework is needed which does not support PPC architecture. As a result, the *QUAD* profiling information on Intel x86 can demonstrate some level of inaccuracy for the architecture-specific data when targeting a different architecture. However, the overall behavior of the application should stay similar. The utilized *Quipu* prediction models were generated for the DWARV C-to-VHDL compiler and the Xilinx ISE 13.2 synthesis tools targeting the same Virtex5 FPGA containing a Molen machine implementation. For other FPGA devices, conversion formulas should be used based on the authentic published data-sheets. The Molen implementation requires 7283 slices, leaving 23437 slices available for accelerating application kernels. All the simulations were performed using *Modelsim 6.5f*.



(a) Lena greyscale image (b) Gaussian smoothing (c) Derivative along the x-axis

(d) Derivative along the y-axis (e) Gradient magnitude (f) Non-maximal suppression

(g) Hysteresis thresholding (h) Final output image

Figure 6.3: *The steps of the CED implementation, (a) original greyscale Lena image, (b) the output image after performing Gaussian smoothing ($\sigma = 1$), (c) X-derivative of the Gaussian smoothed image, (d) Y-derivative of the Gaussian smoothed image, (e) gradient magnitude, (f) the result image after applying non-maximal suppression, (g) the application of hysteresis thresholding with thresholds at low/high levels to get weak/strong edge pixels, (h) the final output image showing the edges.*

6.3.3 Experimental Analysis

We follow the steps defined in the Q^2 partitioning methodology (Section 6.2) to analyze the CED application. The results of analyses are presented in the following. Based on the extracted information and detailed analyses, we make some conjectures on revising the CED source code in several phases to tailor and optimize the code for mapping onto the Molen platform.

Table 6.1: *gprof flat profile for the CED application on the Intel x86 architecture.*

Kernel	%time	Self seconds	Calls	Self ms/call	Total ms/call
gaussian_smooth	70.11	0.0985	1	98.50	98.50
non_max_supp	12.46	0.0175	1	17.50	17.50
magnitude_x_y	6.41	0.0090	1	9.00	9.00
apply_hysteresis	4.63	0.0065	1	6.50	11.50
follow_edges	3.56	0.0050	1433	0.00	0.00
derrivative_x_y	2.85	0.0040	1	4.00	4.00
canny	0.00	0.0000	1	0.00	140.50
make_gaussian_kernel	0.00	0.0000	1	0.00	0.00
read_pgm_image	0.00	0.0000	1	0.00	0.00
write_pgm_image	0.00	0.0000	1	0.00	0.00

% time is the percentage of the total execution time of the program used by the function; ***Self seconds*** is the number of seconds accounted for by the function alone; ***Calls*** is the number of times a function is invoked; ***Self ms/call*** is the average number of milliseconds spent in the function per call; ***Total ms/call*** is the average number of milliseconds spent in the function and its descendants per call.

Hotspots Identification. General profilers can provide an overview of an application. In an initial attempt, we used the conventional *GNU* profiler, *gprof*, to further substantiate the added value of its counterpart, *Maip*, in the Q^2 profiling framework. Using *gprof*, the reported numbers for a typical run of the application demonstrate considerable amount of error. This is due to the fact that the application runs for a quite short time (approximately 150 milliseconds). Considering the sampling period which is 10 milliseconds, functions do not get much chances of being examined by the profiler. As mentioned previously, this is the main problem with *gprof*. To alleviate the problem, we run the application 20 times and recorded the average values. Table 6.1 summarizes these results. As seen in Table 6.1, all the functions are called once with the exception of *follow_edges*, which is a recursive function. The number of times *follow_edges* is called depends on the image itself and on the values of input parameters. The primary share of the total execution time is attributed to *gaussian_smooth*. It is also interesting to note that there is no self contribution[5] for *canny*, while the total contribution of this function and its descendants is around 140 milliseconds, which is equal to the sum of contributions from *gaussian_smooth*, *non_max_supp*, *magnitude_x_y*, *apply_hysteresis*, and *derrivative_x_y*. It indirectly implies that *canny* is the main function doing all the processing by just calling individual functions to carry out different phases in the edge detection algorithm described before. The time taken for reading the input image file and writing the output data is negligible.

Table 6.2 presents the *gprof* profiling results on the embedded PPC. As seen in Table 6.2, the total execution time of the application is increased by approximately 60× due to the decrease in the processing speed and simulated floating point arithmetic. Hence, the length of the application execution is large enough to get somehow accurate

[5] Execution contribution of the function itself, excluding the functions it calls.

Table 6.2: *gprof flat profile for the CED application on the embedded PPC.*

Kernel	%time	Self seconds	Calls	Self s/call	Total s/call
gaussian_smooth	69.09	5.79	1	5.79	5.79
non_max_supp	19.81	1.66	1	1.66	1.66
magnitude_x_y	5.61	0.47	1	0.47	0.47
derrivative_x_y	3.70	0.31	1	0.31	0.31
apply_hysteresis	0.95	0.08	1	0.08	0.14
follow_edges	0.72	0.06	1433	0.00	0.00
canny	0.00	0.00	1	0.00	8.37
make_gaussian_kernel	0.00	0.00	1	0.00	0.00
read_pgm_image	0.00	0.00	1	0.00	0.00
write_pgm_image	0.00	0.00	1	0.00	0.00

% time is the percentage of the total execution time of the program used by the function; ***Self seconds*** is the number of seconds accounted for by the function alone; ***Calls*** is the number of times a function is invoked; ***Self s/call*** is the average number of seconds spent in the function per call; ***Total s/call*** is the average number of seconds spent in the function and its descendants per call.

data with the sampling technique. Some changes are evident in the contribution percentages and the ordering of the functions. *follow_edges* and *apply_hysteresis* each now contributes less than 1 percent of the whole execution time.

A brief inspection of the application reveals the links between the main conceptual steps described in the CED algorithm and the corresponding functions defined in the source code. The top kernel, *gaussian_smooth*, utilizes the Gaussian filter to blur the input image. The filter itself is created in *make_gaussian_kernel*, which only allocates and fills a one-dimensional floating array. The size of the array is dependent on the input parameter *sigma* (standard deviation of the filter). However, for a predefined sigma value, there is a possibility of hard-wiring the individual calculated values. It is clear that the first step of the algorithm is the most time consuming task. The blurring procedure is performed on each pixel in the input image. *derrivative_x_y* computes the first derivatives (gradient) of the image along both the x and y directions. Subsequently, *magnitude_x_y* calculates the magnitude of the gradient. These functions together relate to the second step of the CED algorithm. Step 3 of the algorithm is implemented by *non_max_supp*, which applies non-maximal suppression to the magnitude of the gradient image. Finally, *apply_hysteresis* implements the last step in the CED algorithm. Basically, the function finds edges that are above a high threshold or connected to a high pixel by a path of pixels greater than the low threshold. This is done by first initializing the edge map with all the possible edges that the non-maximal suppression suggested, except for the borders. Then, when a pixel is located above the high threshold, the function calls the recursive function *follow_edges* to continue tracking the edge along all paths.

In order to have accurate execution time estimates and also an overview of some memory access related statistics, we used *Maip* to profile the application. Table 6.3 presents a summary of the results. The most accurate values for the execution contribution of each function is calculated with *Maip*, as it accounts for each single instruction within a function, contrary to the sampling technique used in *gprof*. As seen in Ta-

Table 6.3: *MAIP flat profile for the CED application.*

Kernel	% time	MAR	NLOC-MAR	MOR	NLOC-MOR	Stk Ratio	Flow Ratio	NLOC-Flow Ratio	Bytes/Acc.
gaussian_smooth	69.89	29.05	11.75	9.56	3.74	61.54	0.7376	0.9213	3.3090
non_max_supp	14.22	51.89	10.36	20.58	3.84	89.79	0.3635	0.8712	3.4318
magnitude_x_y	5.62	50.00	9.37	17.98	3.37	87.51	0.4168	0.3333	3.0007
apply_hysteresis	4.97	23.21	21.71	8.26	7.54	27.10	0.3288	0.4026	1.2998
derrivative_x_y	2.99	64.74	35.22	26.70	13.32	66.79	0.5557	0.3334	3.0028
follow_edges	2.24	48.33	7.79	18.25	4.41	94.85	-0.0030	0.8225	3.4357
read_pgm_image	0.03	46.81	22.56	18.78	9.39	59.02	0.5881	0.7547	3.4548
write_pgm_image	0.00	60.61	28.72	24.32	15.41	43.57	0.1258	0.1652	3.8105
make_gaussian_kernel	0.00	45.14	10.72	15.34	5.20	80.06	0.2529	0.8504	4.7549
canny	0.00	52.26	12.22	20.23	7.34	75.84	0.1889	0.6508	3.9658

% time is percentage contribution of the execution time; ***MAR*** is the percentage ratio of the memory access operations to the total instructions executed in the application; ***NLOC-MAR*** is the same as ***MAR*** except that only references to the non-local region are considered; ***MOR*** is the percentage ratio of the memory access operands to the total number of operands; ***NLOC-MOR*** is the same as ***MOR*** except that only references to the non-local region are considered; ***Stk Ratio*** is the percentage ratio of the memory access instructions within the local region to the total memory access instructions; ***Flow Ratio*** is an indication of a function being more memory reader or writer. -1 means that the function only writes to the memory and +1 means that the function only read from memory; ***NLOC-Flow Ratio*** is the same as ***Flow Ratio*** except that only references to the non-local region are considered.

ble 6.3, the values more or less conform to the numbers that were calculated previously. *gaussian_smooth* is not only the top contributor for the CED application, but also it demonstrates a relatively low percentage of memory access related workload compared to the computational burden. The *NLOC-MAR* and *Stk ratio* columns in Table 6.3 can reveal valuable hints regarding the tendency of the function to go for local or non-local memory accesses. This is an important attribute if we opt to maintain the dynamically allocated memory on external sources (off-chip memory) in reconfigurable devices [161]. For example, we would rather map *gaussian_smooth* or *non_max_supp* on FPGA than *apply_hysteresis*, because a higher portion of memory accesses are intended for external memory regarding the latter function. In the case of the Molen platform, however, this is not relevant, as Molen does not currently support off-chip memory. As such, all the memory space required for the execution of the application should be allocated and managed on-chip. In this respect, a memory management module particularly takes care of all the dynamic memory allocations in the application.

Hardware Area Estimation. In the continuing work with the CED application, we want to be able to map different kernels to hardware. An important caveat for mapping kernels to hardware is that within the *DWB*, some restrictions apply to the kernels that will be mapped to hardware. For this reason, we have modified the CED application where necessary, so that the intensive kernels could be mapped to hardware. There were two main issues that were solved:

- **Memory allocation**
  The DWARV compiler currently does not support allocating memory blocks at runtime from hardware. Therefore, all memory allocations were moved from the kernels into function stubs that allocate the required memory blocks first and then call the real kernels.

- **(Recursive) function calls**
  Furthermore, at this time function calls are not supported in DWARV. For most cases, manually inlining the code solved the problem. However, there was one instance of recursion in the *follow_edges* function, which inhibited simple inlining in the *apply_hysteresis* function. Therefore, the recursive function call was moved to an appropriate function stub.

Of course, these changes required new profiling results. In Table 6.4, the top 5 kernels are listed with their associated new time contributions, as reported by *Maip*.

Evidently, in order to partition the application over hardware and software components, the evaluation of the computational and memory hotspots is not sufficient. As there is only a limited amount of reconfigurable hardware area available, an investigation of the size of potential hardware designs is warranted. For this purpose, we used a *Quipu* prediction model for the Virtex-5 FX 200T FPGA. The results of the area prediction of the top five contributing kernels are presented in Table 6.4. The table lists the actual number of slices, as well as the percentual area with respect to the target FPGA. The kernels in the table are in order of execution in the CED application. As such, we can evaluate subsequent merging options by providing cumulative area figures as well. Note that the *hw_apply_hysteresis* kernel is exceedingly resource-intensive, consuming

Table 6.4: Area predictions and theoretical speedups for the kernels in the CED application.

Kernel	Area [a]			Exec. time [c]	Speedup [b]	
	Slices	% of area	Cum.[d] % of area		Single kernel	Cum.
hw_gaussian_smooth	1951	8.32%	8.32%	70.59%	3.40×	3.40×
hw_derrivative_x_y	510	2.18%	10.50%	2.49%	1.03×	3.71×
hw_magnitude_x_y	1442	6.15%	16.65%	5.14%	1.05×	4.59×
non_max_supp	2132	9.10%	25.75%	14.36%	1.17×	13.48×
hw_apply_hysteresis	765209	**3264.96%**	**3290.71%**	2.68%	1.03×	21.10×

[a] Area predicted by a *Quipu* prediction model for the Virtex-5 FX 200T.
[b] Theoretical application speedup, assuming 0 s execution time for each kernel. Single is for the case when each kernel is accelerated by itself, Cum. is for the case when all kernels in the sequence including the current kernel are accelerated together.
[c] Percentage contribution of the execution time as reported by *MAIP*
[d] Cumulative

3265% of the target FPGA area. This large area requirement can be traced back to a local array of 32K 32-bit integers. The used *Quipu* model targets the DWARV C-to-VHDL compiler, which converts such local arrays to registers, resulting in a large number of slices[6]. When considering to merge several kernels together, the predictions suggest the first four kernels will easily fit together on the target FPGA.

Initial Partitioning. In order to find a candidate set of functions to be moved to the FPGA hardware, we opt to have an idea of the speedup that might be achieved. Therefore, in addition to the area predictions, the theoretical application speedups for the kernels in the application are also reported in Table 6.4. These speedups are calculated using Amdahl's law [8] assuming an unlimited speedup for the kernels in question, as follows:

$$\lim_{p \to \infty} \frac{p}{1 - f(p-1)} = \frac{1}{f} = \frac{1}{1-s}, \qquad (6.1)$$

where p stands for the speedup factor that is achieved in the accelerated part of the application, f stands for the percentual contribution of the remaining sequential part of the application, and s stands for the percentual execution contribution of the accelerated part of the application before acceleration. Table 6.4 lists both the speedup when one kernel is accelerated and the cumulative speedup, where each subsequent kernel is accelerated together with the previously accelerated kernels. Observe that as large parts of the application are accelerated, the contribution of the remaining kernels becomes more significant. For example, *apply_hysteresis* has a contribution of 2.68%, but with much of the application already accelerated, the difference in theoretical speedup is 56.8%.

As we have mentioned before, merging the first four kernels in Table 6.4 would yield a hardware block that would fit in the target FPGA. The maximum speedup of the application using that block would be 13.48×. Of course, the efficiency of accelerating this block will never be 100%. And the actual speedup will be lower. The designer would have to decide whether to go for the larger kernel or to instantiate several blocks of smaller

[6] It translates to 1M (1,048,576) flip-flops with additional logic and wiring.

size in parallel. This last option can especially be interesting when edge detection is performed on a video stream.

Data Communication Analysis. To have a clear insight into the CED application behavior regarding the data communication between different functions, we utilized *Quad* to profile the application. The QDU graph of the CED is depicted in Figure 6.4. The graph makes it possible to visually follow the journey of data objects through the sequences of function calls. In this way, we can trace what is actually happening to the input image as it moves along different phases of the CED algorithm. The detailed information presented in the graph also helps to understand what are the memory requirements of each function to accomplish its task. Furthermore, it plainly identifies the actual data dependencies between the functions. The critical data path is highlighted in the graph to distinguish it from all the subordinate data communications between functions.

Quad starts instrumenting the application as soon as the main function gains control. In other words, the data communication associated with the operating system calls and libraries are somehow overlooked. Although this normally does not contain much valuable information about the critical data path of the application, it may cause missing starting or ending points for communication edges, i.e. the producer or consumer of the data. Currently, for every byte with an unidentified producer, a manual investigation has to be conducted to verify the source of data. In the CED application, the main part of the input image file (800×600 bytes) has been tracked down to be part of the *unknown producer* entity. We revised the graph to replace it with a dummy *Image* node. A small part of the input image (4864 bytes) is recognized to be attributed to *read_pgm_image*. Since the file reading process is considered to be I/O, the operating system will take care of it, and how the process is implemented is completely platform-dependent. Hence, slight inconsistencies appear in numbers reported by *Quad* for such cases. Nevertheless, this is not affecting the overall picture of the data communication in the CED application.

make_gaussian_kernel is responsible for creating the Gaussian filter array. Based on the standard deviation used in our case study, an array of eleven elements is allocated. This is verified by the 44 UnMAs reported on the edge from *make_gaussian_kernel* to the *hw_gaussian_smooth*. It should be noted that the array of eleven floating point values is accessed repeatedly throughout the smoothing process. As a result, huge data communication is reported between the two functions (about 40 MB). *hw_gaussian_smooth* produces a temporary image data object filled with calculated intermediate floating point values. This is clearly reported by a self edge of 800×600×4 bytes UnMAs in the graph. *hw_derrivative_x_y* uses the smoothed image data object as an input. The smoothed image used in *hw_derrivative_x_y* is of the short integer type, which is appearing as a corresponding edge of 960000 UnMAs between the *hw_gaussian_smooth* and *hw_derrivative_x_y*. It can also be derived that the smoothed image data object is accessed four times in total (3840000 bytes), two times for calculating the derivative in the X direction and two times for the Y direction.

The calculated derivatives are stored separately in two arrays of the short integer type. They serve as inputs to *hw_magnitude_x_y*, which will compute edge strength values based on the gradient of the image. The result will be saved in an array called *magnitude*. It is also clear from the graph that the input arrays are scanned only once to compute the magnitudes. The third step of the CED algorithm needs the computed

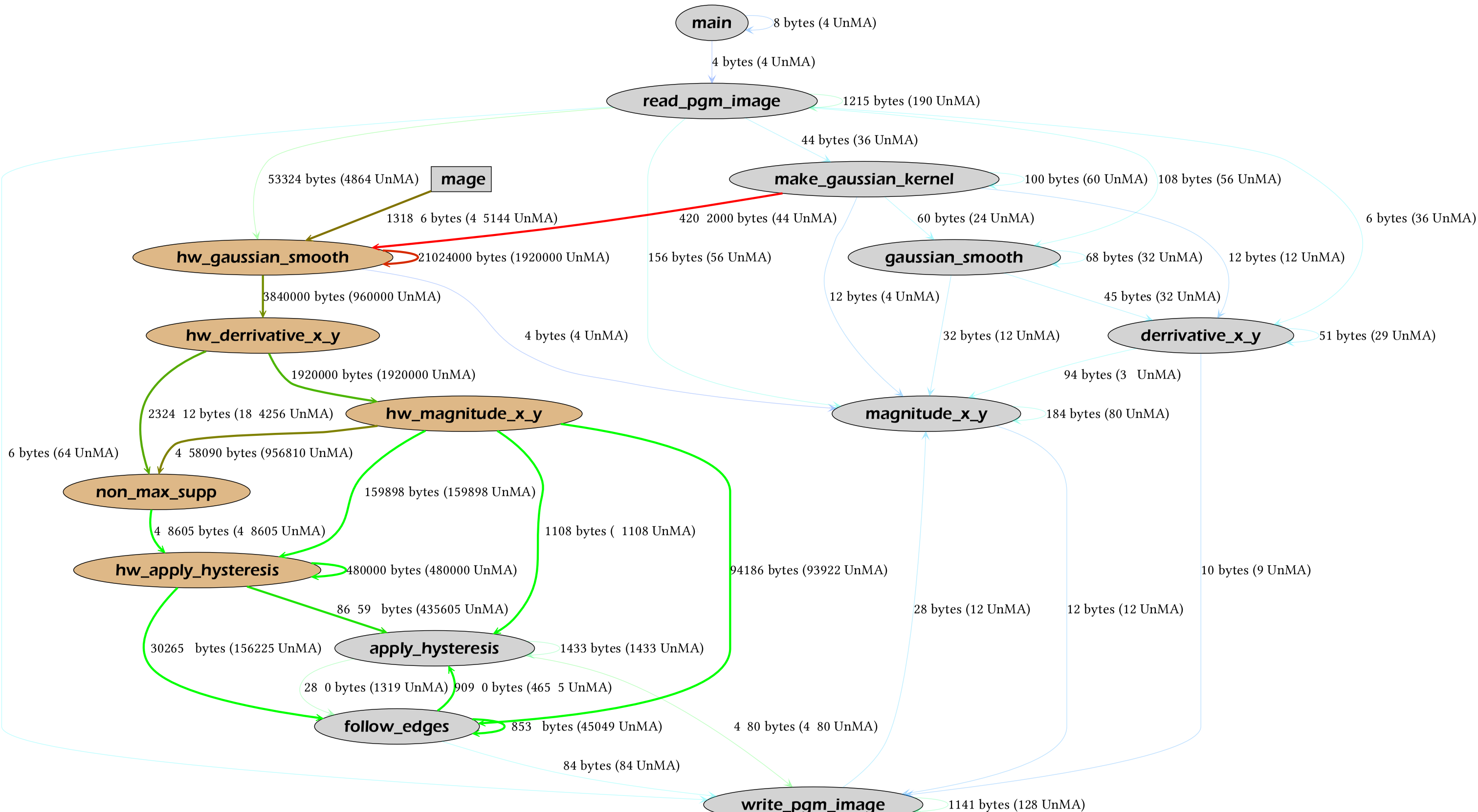


Figure 6.4: QDU graph for the hardware version of the CED application.

magnitude array and also the derivative arrays. This can be verified by the graph edges connecting *hw_derrivative_x_y* and *hw_magnitude_x_y* to the *non_max_supp*. For each pixel, the values of neighboring pixels in some directions are also examined. This results in a large number reported for memory accesses (approximately 4.5 MB). The resulting binary image, also known as *thin edges*, is output into an array with the same size of the input image.

Regarding the final step, *hw_apply_hysteresis* (excluding the last recursive part to trace along the identified edges) uses the *magnitude* array and the output array from non-maximal suppression. As shown in the extracted data, the usage of *magnitude* is conditioned to only those pixels which indicate a possible edge. This is dependent on the input image. In our case, about one sixth of the total pixels are identified as possible edge pixels. The self edge of 480000 bytes in *hw_apply_hysteresis* is attributed to the initialization of the *edge map* array. The array is subsequently processed in *hw_apply_hysteresis* for the computation of the histogram of magnitude values. *apply_hysteresis* finalizes the *edge map* by using the output of *hw_apply_hysteresis* and the *magnitude* array. To accomplish this task, it in turn utilizes *follow_edges*. All the corresponding data communications to perform the hysteresis process is identified and presented in Figure 6.4. *write_pgm_image* is responsible for writing the *edge map* array to the output file. As explained before, no major consumer of the array is identified by *QUAD* due to the I/O process.

Final Partitioning. Modifying the CED application to comply with the hardware mapping restrictions, in turn, causes some changes in the data communication patterns of the application. As an obvious result, new data communication channels are formed between the introduced function stubs and the corresponding hardware-compliant kernels. However, this characteristic will not reinforce the data communication problem, as the connections between the stubs and their corresponding kernels are limited to providing the starting address of allocated memory blocks and some related basic data elements. There is only one exception in the case of *apply_hysteresis*, whose body can not be moved entirely to hardware, due to the invocation of a recursive function, *follow_edges*. Therefore, a considerable amount of data transfer is established between the extracted *hw_apply_hysteresis* and the corresponding function stub, *apply_hysteresis* (850 kB using 425k UnMAs). The newly formed communication channel may be considered as a source of potential memory bottleneck and needs proper handling. Primarily, *apply_hysteresis* is dependent on the data that is provided by the *nms* and *magnitude* arrays. The dataflow originating from *magnitude_x_y* is now divided into separate flows for *apply_hysteresis* and *hw_apply_hysteresis*. From the total amount of 225K memory accesses, approximately 70% is accounted for *hw_apply_hysteresis* and the rest for *apply_hysteresis*. Nevertheless, both functions strictly access a whole part of the *magnitude* array. In essence, *magnitude* should be made available for both functions as a whole, regardless of the number of accesses carried out on the data residing in the array.

Not every heavy data communication yields a potential memory bottleneck. A more detailed investigation is required to pinpoint problems related to memory accesses. Special attention has to be given to the size of the accessed memory blocks, the locality, the reusability, and, most significantly, to the placement of the data (on-/off-chip data allocation [161]) where applicable. For our experiments, there was no off-chip data allocation due to the Molen restrictions. However, this property must be considered in

Table 6.5: Area predictions and theoretical speedups for the merged and optimized versions of the CED application.

Instance	Area[a]	Exec. time[b]	Speedup	
cumulative	6035	92.58%	13.48×	Theoretical
merged	5874	92.58%	13.48×	
optimized (block reuse)	5808	92.57%	13.46×	
optimized (loop merge)	5576	92.68%	13.66×	
Instance	**Area**	**Time[c] (ms)**	**Speedup**	
simulation	7307[d]	88.5	2.92×	Real

[a] Area predicted by a *Quipu* prediction model for the Virtex-5 FX 200T.
[b] Percentage contribution of execution time as reported by Maip.
[c] Execution time as calculated by the *Modelsim 6.5f* simulator.
[d] Actual area for the optimized version (block reuse) of the merged kernel.

the general case. A review of the critical data path reveals several potential problematic memory access bottlenecks, which limit the performance of the application. Loading the image from an external source is the first obstacle.

Beginning with *hw_derrivative_x_y*, there is a series of data communication via different memory blocks, which is responsible for the main performance bottleneck of the application. In each (sub)phase, one or more memory blocks are used as input to produce an output block. The critical data path goes through *hw_magnitude_x_y*, *non_max_supp*, *hw_apply_hysteresis*, *apply_hysteresis*, and *follow_edges*. Optimizing the CED application should be centered around the block processing. This, in a subsequent analysis, requires thorough examination of the exact life span of each block, the data dependencies between them, and the possible merging/reusing of the relevant data.

After a careful analysis of the results, we observed that all hardware-compliant kernels up to *hw_apply_hysteresis* fit together on the reconfigurable fabric. Furthermore, together, they would exhibit a significant potential speedup of 13.46×. Finally, the bulk of the communication occurred in this group of functions. In the following, we continue our evaluation by merging these kernels, providing additional profiling results, and implementing certain optimizations that are suggested by the results.

The initial merging process consisted of the concatenation of the subsequent function calls. In the case of CED, this process is trivial. However, in case the to-be-merged functions have no direct connections in the call graph, the process may become more complex. In Table 6.5, we observe that, prior to the merging, the predicted area consumption was 6035 slices with a theoretical speedup of 13.48×. After the merging, the potential speedup remains the same, but the predicted area decreases to 5874 slices. The reason for this behavior is very likely the increased reuse of calculations and variables. The used *Quipu* model was generated for the DWARV compiler and Xilinx ISE synthesizer, which use common compiler front-end optimizations and resource sharing, which can significantly reduce the required area. In particular, the optimizations include the reduction in the number of required unique variables, which, in turn, results in the reduction of the hardware area consumption when the number of registers is lessened.

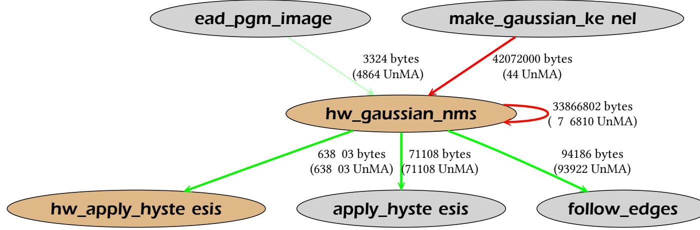


Figure 6.5: *Partial QDU graph for the hardware version of the CED application after merging.*

Now that we have a merged kernel, the *Quad* results also change accordingly. In Figure 6.5, we see that most of the memory locations and accesses are now internal to the merged *hw_gaussian_nms* kernel. When we carefully investigate these new results, we see that the number of UnMAs is roughly 12 times as large as our input image (800×600×12×1 byte = 5760000 bytes). As subsequent phases of CED use different temporary data objects, we analyzed how the corresponding memory blocks might be reused. Figure 6.6 depicts the different memory blocks that are allocated over time through the critical data path of the CED application. Memory size is indicated on the vertical axis in units of one image block, and the time on the horizontal axis is represented in terms of phases of the algorithm. By determining the live ranges of different memory blocks allocated throughout the main phases of the CED implementation, we observed that the maximum amount of memory needed at the same time is 7 times the size of one image block. Therefore, we optimized the merged kernel to reuse memory blocks when they are no longer needed. Table 6.5 indicates that this optimization (block reuse) does not influence the potential speedup, but the required amount of memory is reduced significantly. Furthermore, the required area remains effectively the same at 5808 slices.

Apart from required memory size, the number of memory accesses in the merged kernel is also significant. In order to reduce this amount, we performed loop merging and introduced temporary variables to reduce redundant memory accesses in subsequent iterations, effectively reducing the number of accesses by 8.5% from 33866802 to 30986802 accesses. Again the theoretical speedup is not significantly affected, although the actual speedup of the hardware implementation is expected to improve in case of using off-chip memory [161]. The same situation holds for hardware area prediction as calculated by *Quipu*. Due to minor modifications in the source code, we estimate a slight reduction in the required hardware area for the merged kernel.

6.3.4 Observations and Results

In order to evaluate the proposed adjustments, VHDL code was generated for the merged *hw_gaussian_nms* kernel and simulated for the target platform using *Modelsim 6.5f*. Using exactly the same kernel input data as the one used on the PPC, the simulation took 177 million cycles. The synthesis of the kernel suggested a maximum clock speed of 235 MHz. Assuming a conservative 200 MHz results in an execution time of 88.5 ms. On the PPC, the execution took 292 ms, accounting for a kernel speedup of 3.44× and an application speedup of 2.92×. Because of the merging we performed, the kernel speedup has

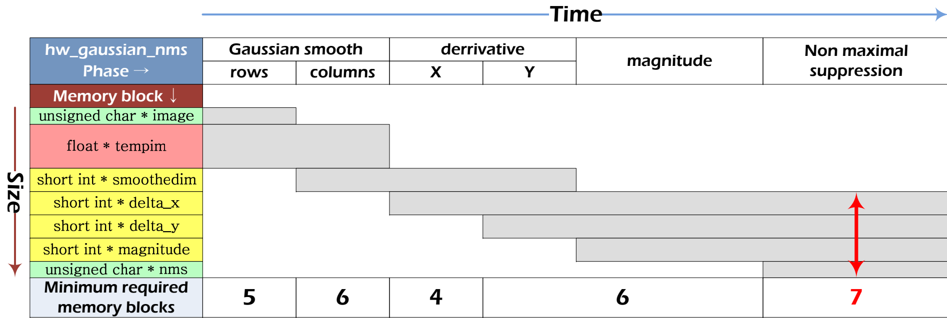


Figure 6.6: *Overview of the memory blocks used throughout the critical data path of the CED application.*

a big impact on the overall application speedup, as the merged kernel represents most of the computational work.

6.4 Mixed Excitation Linear Prediction

Following the first case study, we present the case of a complex application from the speech processing domain, namely the Mixed Excitation Linear Prediction (MELP) vocoder [189]. As before, we perform some revisions at the source code level based on the extracted profiling information to tailor the application for execution on our target platform. In this case study, in addition to thorough behavioral analysis of the application, we particularly address the following issues in our partitioning methodology:

- the concept of **nontrivial** merging of coupled functions based on detailed data communication;
- the detailed examination of merging possibilities, whereas the application comprises different potential kernel candidates for hardware mapping.

In the following sections, we first introduce the MELP algorithm in Section 6.4.1. Subsequently, in Section 6.4.2, we briefly mention the experimental setup in this case study. Section 6.4.3 includes detailed results of profiling examinations as well as concise discussions regarding the extracted data. Kernel merging opportunities along with relevant source code modifications are also investigated in this section. Finally, in Section 6.4.4, we present the empirical results.

6.4.1 MELP Overview

The MELP vocoder [189] is a standard low rate speech coder selected by the United States Department of Defence (USDoD) Digital Voice Processing Consortium (DDVPC) and used mainly in military/satellite communications, and secure voice/radio devices. Among several candidates, MELP was selected mainly based on the following features:

voice quality, talker distinguishability, intelligibility, and communicability. The selection criteria also included hardware parameters such as processing power, memory usage and delay. The initial MELP algorithm was invented by *Alan McCree* in the mid 90s at the Center for Signal and Image Processing at Georgia Institute of Technology in Atlanta. Later, the MELP vocoder was developed by a team from Texas Instruments Corporate Research in Dallas and Atlanta Signal Processors.

Traditional pitched-excited Linear Predictive Coding (LPC) vocoders use either a periodic pulse train or white noise as the excitation for an all-pole synthesis filter. These vocoders produce intelligible speech at very low bit rates, but they sometimes sound mechanical or buzzy, and are prone to annoying thumps and tonal noises. These problems arise from the inability of a simple pulse train to reproduce all kinds of voiced speech. The MELP vocoder uses a mixed-excitation model that can produce more natural sounding speech due to the representation of a richer ensemble of possible speech characteristics. MELP is particularly robust in difficult background noise environments such as those frequently encountered in commercial and military communication systems. Furthermore, it is very efficient in its computational requirements. As a result, it has low power consumption, which is a crucial consideration in portable, embedded and dedicated hardware systems.

The enhanced version of MELP vocoder (MELPe) surpasses that of the old version in terms of voice quality. MELPe vocoder implementation operates at 2400, 1200 and 600 bps. It also includes compressed bitstream transcoding between the rates, optional Noise Pre-Processor (NPP), Dual Tone Multi Frequency (DTMF) detection and regeneration, tone signal generation and jitter-buffering. Moreover, MELPe uses extensive lookup tables and models of the human voice to extract and regenerate speech and it is tuned to regenerate the English language.

Algorithm Description. The MELP vocoder is based on the traditional LPC parametric model, but also includes four extra features, namely *mixed-excitation*, *aperiodic pulses*, *pulse dispersion*, and *adaptive spectral enhancement*. These additional features mainly improve the excitation structure of LPC along with an accurate simulation of the natural speech.

The mixed-excitation is implemented using a multi-band mixing model to reduce the buzz usually associated with LPC vocoders, especially in broadband acoustic noise. This multi-band mixing model can simulate frequency dependent voicing strength using a novel adaptive filtering structure based on a fixed filterbank.

When the input speech is voiced, MELP can synthesize speech using either periodic or aperiodic pulses. Aperiodic pulses are most often used during transition regions between voiced and unvoiced segments of the speech signal. This feature allows the synthesizer to reproduce erratic glottal pulses without introducing tonal noises.

The pulse dispersion filter, based on a spectrally flattened triangle pulse, is used in the final stage to spread the excitation energy with a pitch period. This, in turn, reduces the harsh quality of the synthetic speech.

The adaptive spectral enhancement filter is based on the poles of the LPC vocal tract filter and is used to enhance the formant structure in the synthetic speech. This filter improves the match between synthetic and natural bandpass waveforms, and introduces

a more natural quality to the speech output.

In the MELP algorithm, first, the speech is filtered into five frequency bands, then, a linear prediction analysis is performed on the input speech using a hamming window centered on the last sample of the current frame. Afterwards, the LPC residual signal is calculated by filtering the input speech signal with the prediction filter, whose coefficients are determined by the linear prediction analysis. At this point, the final pitch estimate is calculated using the low-pass filtered residual signal. The gain is estimated subsequently. The first ten Fourier magnitudes are obtained by picking peaks in the Fast Fourier Transform (FFT) of the residual signal. The information embodied in these coefficients improves the accuracy of the speech production model at the perceptually important lower frequencies. This increases the quality of the coded speech, particularly for males and in the presence of background noise. To put the speech data (LPC coefficients, pitch, gain, and bandpass voicing) into a smaller representation, Vector Quantization (VQ) is used.

The encoded data, which are formed into packets, along with the VQ codebook, constitute the MELP bitstream. Following that, channel coupling is generated. The decoding phase is less complex than the encoding. The packets are decomposed out of the bitstream and are processed for factors extraction and channel decoupling. Initially, the speech signal in each frequency band is recovered, and then the output speech signal is regenerated by applying the voicing filter and adaptive spectral enhancement post filter.

The encoding and decoding phases of the MELP algorithm can be classified into two stages in each phase. Encoding begins with the ***Analysis*** stage, which analyses the speech data to find the optimal representation, and continues with the ***Coding*** stage to actually encode the speech data into smaller data representation. Decoding, on the other hand, starts with the ***Decoding*** stage to reconstruct the sound signal representation from the data packets, and subsequently, in the ***Synthesis*** stage, the initial speech signal is regenerated. To elucidate the sequence of different steps in the MELP algorithm, a block diagram is shown in Figure 6.7.

6.4.2 Experimental Setup

For our experiments, we used the Proposed U.S. Federal Standard 2.4 kbps MELP vocoder implementation (version 1.2) [144], which is developed jointly by Texas Instruments and Atlanta Signal Processors, Inc. The target platform contains an implementation of the Molen Polymorphic Processor, as described in Section 6.3.2. Due to the platform-dependent restriction imposed by the *Pin* DBI framework, we used the *Quad* toolset on an Intel 32-bit Core2 Duo E8500 @3.16 GHz, running Linux kernel v2.6.34. Hence, the *Quad* profiling data on Intel x86 can exhibit some measure of inaccuracy for the architecture-specific parameters, when targeting a different architecture (For example, PPC, in our case). However, we assume that the overall behavior of the MELP application remains sufficiently similar.

The MELP application was also executed on the embedded PPC 440 @400 MHz, which is integrated in a Xilinx ML510, Virtex-5 FX 200T with 2.0 MB BRAM FPGA board. On this platform, we instrumented the functions, counting the number of cycles for each function call using the PPC Time Base (TB) register which is incremented each clock

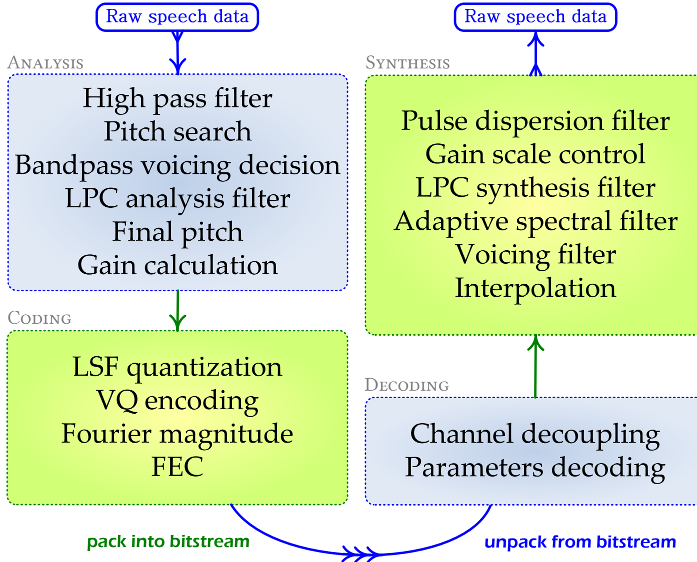


Figure 6.7: *The MELP vocoder block diagram.*

cycle. The MELP application source code was compiled with *gcc* v4.1.1, with *-O2* compiler optimizations on both platforms. The MELP implementation consists of 25 source files with, in total, 70 functions. For the experiments, we encoded a sample voice recording of male voice fragments in 8000 sample/s raw Pulse-Code Modulation (PCM) format. The *Quipu* model that we utilize in this case study was generated for a combination of the DWARV C-to-VHDL compiler and Xilinx Integrated Software Environment (ISE) 13.2 for the Virtex-5 FX 200T FPGA.

6.4.3 Experimental Analysis

In the following, we report the results of different analysis steps regarding the MELP application.

Hotspots Identification. In Table 6.6, we list the top ten kernels according to the *Maip* profiling tool. The top kernel *vq_ms4*, the main part of the vector quantization step in the MELP algorithm, accounts for 23.28% of the total execution time. The subsequent four kernels in the top five are part of the LPC and together account for 56.15% of the total execution time. Then, there is *fft* accounting for 8.15% and some remaining smaller kernels accounting for 7.21%. In the first column, the number of calls to each function is listed. At the first glance, it can be observed that *vq_ms4*, *vq_enc* and *autocorr* are called exactly the same number of times. It easily strikes the mind with the impression that they are somehow related. They are used in the VQ encoding phase.

From the NLOC-MAR column, we can observe that nearly all kernels spend roughly 20% to 30% of their execution times in accessing the non-local memory region (most probably the heap area). One exception is the *lsp_g* kernel, which contains one loop

Table 6.6: *MAIP flat profile for the MELP application.*

Kernel	Calls	% time	MAR	NLOC-MAR	MOR	NLOC-MOR	Stk Ratio	Flow Ratio	NLOC-Flow Ratio	Bytes/Acc.
vq_ms4	2268	23.28	47.39	18.99	16.11	6.28	61.04	0.8184	0.8071	4.0000
zerflt	22303	19.56	37.69	19.40	12.75	6.34	50.31	0.8594	0.8597	4.0000
find_pitch	9072	14.88	32.04	31.21	9.68	9.42	2.72	0.9831	1.0000	4.0003
polflt	18144	11.53	30.27	26.20	9.39	8.13	13.44	0.8713	0.8533	4.0000
frac_pch	32097	10.18	37.27	22.42	11.71	7.06	39.78	0.8796	0.9996	3.9992
fft	1891	8.15	39.92	23.35	12.25	7.09	43.03	0.2977	0.1864	4.0640
lsp_g	687115	2.16	25.64	7.69	7.41	2.59	70.00	0.4000	1.0000	4.0000
vq_enc	2268	2.09	23.69	22.47	7.35	6.98	5.18	0.9945	0.9957	4.0000
envelope	9072	1.77	26.78	26.59	7.73	7.68	0.69	0.5019	0.5016	4.0062
autocorr	2268	1.19	33.22	32.95	9.98	9.90	0.83	0.9931	1.0000	4.0000

Calls indicates the number of times the function is called; ***% time*** is percentage contribution of the execution time; ***MAR*** is the percentage ratio of the memory access operations to the total instructions executed in the application; ***NLOC-MAR*** is the same as ***MAR*** except that only references to the non-local region are considered; ***MOR*** is the percentage ratio of the memory access operands to the total number of operands; ***NLOC-MOR*** is the same as ***MOR*** except that only references to the non-local region are considered; ***Stk Ratio*** is the percentage ratio of the memory access instructions within the local region to the total memory access instructions; ***Flow Ratio*** is an indication of a function being more memory reader or writer. -1 means that the function only writes to the memory and +1 means that the function only read from memory; ***NLOC-Flow Ratio*** is the same as ***Flow Ratio*** except that only references to the non-local region are considered.

Table 6.7: Area predictions and theoretical speedups for the kernels in the MELP application.

Kernel	Area [a]			Exec. time [c]	Speedup [b]	
	Slices	% of area	Cum.[d] % of area		Single kernel	Cum.
vq_ms4	36337	**155.04%**	-	23.28%	1.30×	-
zerflt	540	2.30%	2.30%	19.56%	1.24×	1.24×
find_pitch	2202	9.40%	11.70%	14.88%	1.17×	1.53×
polflt	610	2.60%	14.30%	11.53%	1.13×	1.85×
frac_pch	4760	20.31%	34.61%	10.18%	1.11×	2.28×
fft	2903	12.39%	47.00%	8.15%	1.09×	2.80×
lsp_g	1053	4.49%	51.49%	2.16%	1.02×	2.98×
vq_enc	557	2.38%	53.87%	2.09%	1.02×	3.18×
envelope	810	3.46%	57.33%	1.77%	1.02×	3.37×
autocorr	560	2.39%	59.72%	1.19%	1.01×	3.51×

[a] Number of slices predicted by a *Quipu* model generated for DWARV C-to-VHDL and the Virtex-5 FX 200T FPGA.
[b] Theoretical application speedup, assuming 0 s execution time for each kernel. Single is for the case when each kernel is accelerated by itself, Cum. is for the case when all kernels in the sequence except the top kernel, vq_ms4, which does not fit on the target FPGA, are accelerated together.
[c] Percentage contribution of the execution time as reported by *Maip*
[d] Cumulative, excluding the top kernel (vq_ms4), which does not fit on the target FPGA.

that performs one heap load but executes several other operations per iteration. A high NLOC-MAR indicates that the increased performance of hardware implementation is counteracted by latencies in accessing the memory, if we assume that such accesses can not be further reduced. In the case of these top ten kernels, the NLOC-MAR is low enough to allow for reasonable speedups.

Hardware Area Estimation. The *Quipu* area estimates for each kernel are listed in Table 6.7. Given that the Virtex 5 FPGA on the ML510 board has 20480 slices available, notice that all kernels except *vq_ms4* are predicted to fit. The reason why *vq_ms4* exceeds the size of the FPGA is the corresponding large size of the function itself. It consists of 210 lines of code, contains 12 loops, and has a nesting depth of 9. Although this function, in its current form, can not be mapped to hardware, ample room remains for speedup in the remaining kernels.

Initial Partitioning. In addition to the area predictions, the theoretical application speedups are also reported in the sixth column. The theoretical speedups are calculated using Amdahl's law, assuming unlimited speedups for the examined kernels, as described in Equation (6.1). We can see that without merging some kernels together the maximum possible theoretical speedup would be 1.30×. It should be noted that Amdahl's theoretical speedup is an estimation of the maximum achievable speedup for an application, when an accelerated kernel takes absolutely no time for execution. Doubtlessly, this does not happens in reality. It can only serve as an indication of the highest imagined speedup one conceives in case a kernel is accelerated in the most desirable way. Certainly, the real speedup is bound to be much lower. As such, it is desirable to find a combination of kernels that has a larger computational contribution and, therefore, allows for a larger speedup. This is where the idea of merging coupled kernels becomes

valuable as proposed by the Q² partitioning methodology.

In an initial partitioning, one may decide to have the top kernels, namely, *zerflt*, *find_pitch*, *polflt*, and *frac_pch*, in hardware. Of course, this has to adhere to the restrictions of the target platform, including the available slices on the board and maximum number of parallel CCUs that can be instantiated. *vq_ms4* does not fit on our FPGA board.

Data Communication Analysis. We utilized *QUAD* to reveal the data communications between different functions of the MELP application. The resulting QDU graph is huge and does not fit here. We removed some subordinate functions, which were of lesser importance, and a partial QDU graph is depicted in Figure 6.8. The percentage of execution time contributions of the top kernels are also included in the graph to highlight the application hotspots. Regarding the data communication hotspots, a brief examination of the source code is required to accurately determine the origins of the heavy communications. Intense data communication may be originated from repeated memory reads of the same value(s) from a particular memory address or a limited memory block. These intense memory communications may be easily fixed by modifying the source code. Conventionally, the location of the communicated data for producer/consumer bindings (mainly, the on-/off-chip data allocation), is the primary cause of memory bottleneck.

Final Partitioning. In the top part of Table 6.8, we see an overview of the different candidates for merging. The candidates were selected by looking at functions that would fit on the FPGA and had intense communication. As expected, by inspecting the source code, we noticed that these functions were almost always called together and in similar sequences. In the analysis phase, Q² provides the computational intensity, area estimates, and insight to the communication intensity. As the QDU graph suggests, an obvious merging option would be to combine *polflt* and *zerflt*. It should be noted that examining the call graph for the application displays no calling link between these two function, although they are tightly coupled. These two functions together contribute 31.09% to the computation time. They are intensively communicating, and after inspecting the code, they appear to be called in sequence in almost all cases. Row o_1 lists the tentative data for this merging option. Note that the merged kernel has a larger theoretical speedup than the individual kernels.

The QDU graph also reveals that two other functions, from the top 10, are intensely communicating with *polflt* and *zerflt*, namely *find_pitch* and *frac_pch*. Again, Q² helps us identify these kernels, as the predicted area for each one is relatively small, and the communication with other functions is quite intense. However, when we investigate the code, we notice that these four functions are not called in a consistent order throughout the code. Sometimes only the filters are executed, sometimes *find_pitch* is also included, or *frac_pch*, other times the filters are omitted. In total, we identified four scenarios. We merged the four kernels and added proper *if*-statements to switch on the correct code regions. The combined predictions for this second option are listed in row o_2.

After the identification of the merging candidates utilizing the Q² profiling results, we merged the kernels and performed a new analysis iteration. The results are presented in rows m_1 and m_2 of Table 6.8. As mentioned before, we identified four scenarios

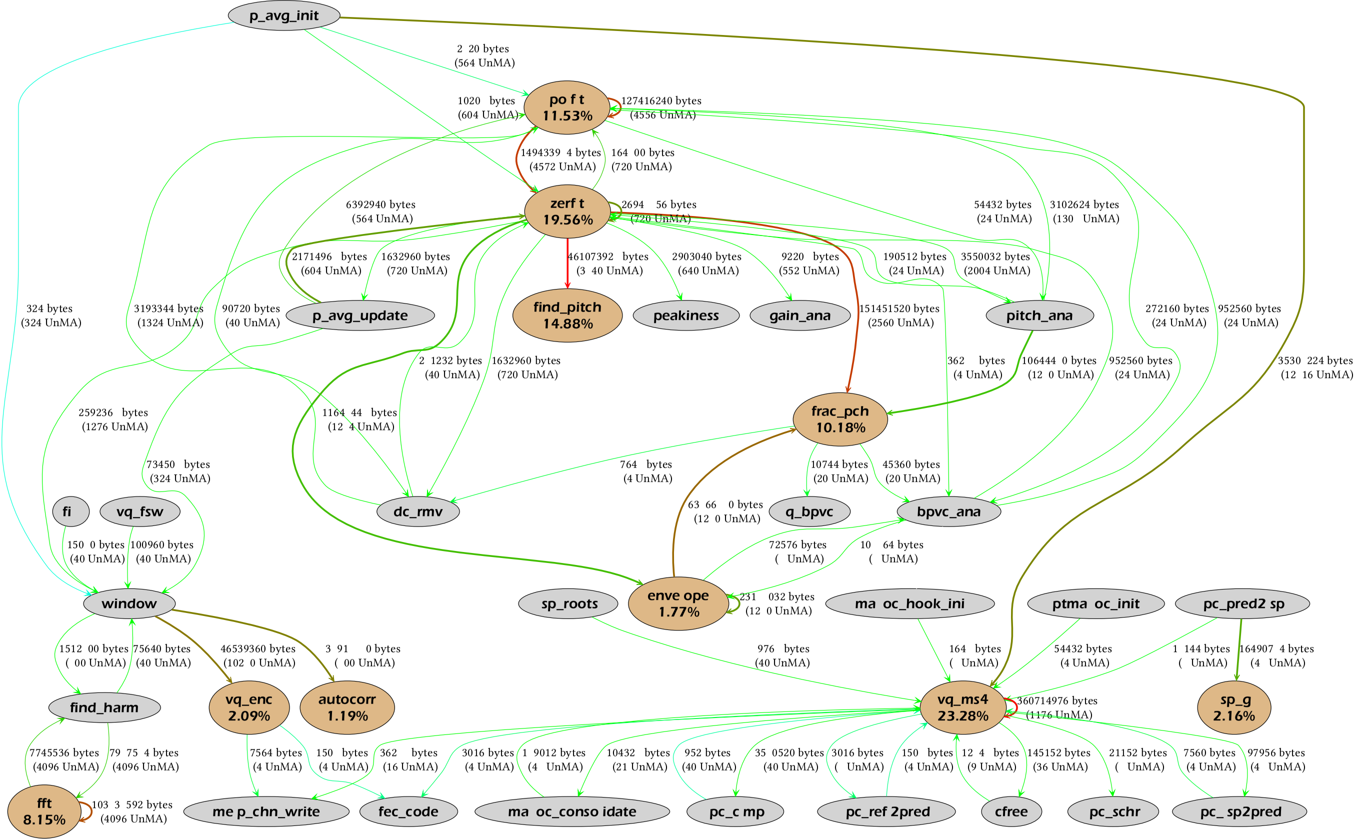


Figure 6.8: Partial QDU graph of the MELP application before merging. The top ten kernels are marked in the graph along with their corresponding execution time contributions as reported by MAIP.

Table 6.8: *Results of the analysis of the merging options, final merged kernels, and the actual synthesis results for the MELP application.*

id	Kernel	Area (slices)		Exec. time[a]	t_{sw}(µs)[b]	t_{hw}(µs)[c]	Speedup		
		Predicted	Actual				Kernel	Application	
								Theoretical	Actual
k_1	zerflt	522	401	19.56	381	181	1.24	2.10	1.11
k_2	find_pitch	1900	2205	14.88	3370	1824	1.17	1.85	1.07
k_3	polflt	661	421	11.53	513	152	1.13	3.38	1.09
k_4	frac_pch	4760	n.a.	10.18	n.a.	n.a.	1.11	n.a.	n.a.
o_1	$k_1 + k_3$	1183	n.a.	31.09	894	333	1.45	2.68	1.24
o_2	$o_1 + k_2 + k_4$	8112	n.a.	56.15	n.a.	n.a.	2.28	n.a.	n.a.
m_1	pol_vequ_zerflt	869	706	26.81	894	365	1.38	2.44	1.19
m_2	filters_plus_pitch	7111	6722	51.80	n.a.	n.a.	2.07	1.80	1.30
m_{2a}	“	“	“	14.84	4259	2505	2.07	1.70	1.07
m_{2b}	“	“	“	1.54	349	181	2.07	1.92	1.01
m_{2c}	“	“	“	28.33	1094	575	2.07	1.90	1.15
m_{2d}	“	“	“	7.09	547	336	2.07	1.63	1.03

[a] Percentage contribution of execution time reported by MAIP.
[b] Time measured on PPC on the ML510.
[c] Time measured with Modelsim 3.5f simulator targeting ML510 Virtex 5 FPGA.

of ordering the individual kernels merged in m_2. For each scenario, we list detailed information in rows m_{2a}-m_{2d}. Observe that the predicted area consumption has been reduced. The reason for this is the additional opportunities for the compiler to optimize the code. We have also included parts of the new QDU graphs representing the first and second merging options in Figure 6.9 and Figure 6.10, respectively. Clearly, the data communication channels between the corresponding kernels have been internalized by the merging process.

6.4.4 Observations and Results

We synthesized the kernels and merged them in order to validate our predictions and to determine the kernel and the application speedups. The results are also listed in Table 6.8. The table includes the execution times in percentage and in *μs* for the PPC and the FPGA. The FPGA execution times were determined using the *Modelsim 6.5* simulator with the same input as on the PPC. Furthermore, we also include the actual area for each of the kernels. In case of the *frac_pch* function, no actual area is reported, as this kernel contains a function call, which DWARV does not support. Apart from the theoretical speedup, we also include the kernel speedup and the corresponding application speedup. It can be seen that the *Quipu* estimates exhibited an error of 5.8% to 57%. This is an acceptable error rate for early analysis of hardware resource consumptions. From the analysis data, it can be observed that the potential for speedup has increased to 2.07×. The actual speedup achieved by merging the kernel candidates was 1.3×.

6.5 Summary

We have seen in this chapter how the Q² profiling framework can be used in real scenarios of partitioning an application into hardware and software parts for a heterogeneous reconfigurable platform. The Canny Edge Detection (CED) and Mixed Excitation Linear Prediction (MELP) applications were presented. A detailed analysis of these two applications was presented that resulted in speedups of 2.92× and 1.30×, respectively. The *Quad* toolset played an important role in extracting the actual data communication bindings, guiding kernels merging, investigating the effect of code transformations, and performing DSE.

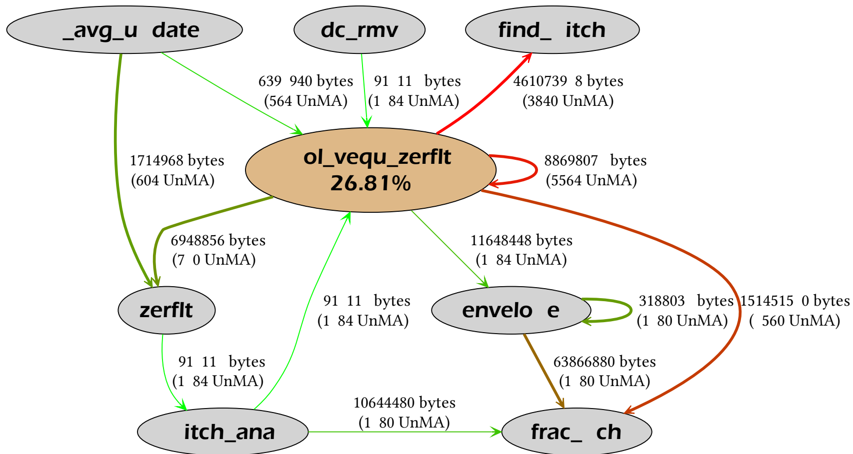


Figure 6.9: *Partial QDU graph of the MELP application after the first merging step.*

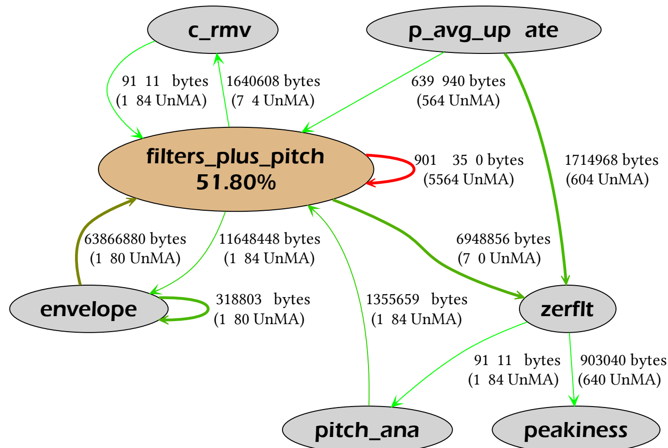


Figure 6.10: *Partial QDU graph of the MELP application after the second merging step.*

Note.

The content of this chapter is partly based on the following articles:

Koen Bertels, S. Arash Ostadzadeh, and Roel Meeuws, **Advanced Profiling of Applications for Heterogeneous Multi-Core Platforms**, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'11)*, Las Vegas, USA, July 2011, pp. 171-183.

S. Arash Ostadzadeh, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, and Koen Bertels, **The Q^2 Profiling Framework: Driving Application Mapping for Heterogeneous Reconfigurable Platforms**, *Proceedings of the 8th International Symposium on Applied Reconfigurable Computing (ARC'12)*, Hong Kong, March 2012, pp. 76-88.

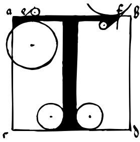
S. Arash Ostadzadeh, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, and Koen Bertels, **Profile-Guided Application Partitioning for Heterogeneous Reconfigurable Platforms**, *Proceedings of the 16th International Symposium on Computer Architecture and Digital Systems (CADS'12)*, Shiraz, Iran, May 2012, pp. 37-43.

CHAPTER 7

Conclusions

"We can only see a short distance ahead, but we can see plenty there that needs to be done." †

— Alan M. Turing



HIS dissertation has presented a dynamic memory access profiling toolset, called *Quad*, which provides quantitative profiling information about the memory accesses in an application. The profiling toolset is able to detect the actual data dependencies that occur, at runtime, between the functions in the application. In addition to the detection of data dependencies, accurate quantitative measurements of the transferred bytes and the size of required memory for communication are provided as well. The extracted profiling information can be utilized to identify critical parts[1] of the application, highlight coarse-grained parallelism opportunities, and direct code optimizations, among others. Although it was developed in the context of the *Delft Workbench* — which targets the Molen heterogeneous reconfigurable platform — our dynamic profiling framework is general and not restricted to any particular platform, application, or purpose. To substantiate the usefulness of the extracted profiling information, we have utilized the main output of the toolset, the Quantitative Data Usage (QDU) graph, as the input data model for general application partitioning. The solutions found by the proposed heuristic clustering algorithm looked promising in comparison to the optimal solutions identified by an exhaustive search. Furthermore, we have investigated in details the utilization of the extracted memory access profiling information in practice using two real scenarios. In this chapter, we first summarize the contents of each chapter in the dissertation in Section 7.1. Then, in Section 7.2, we evaluate the main contributions of our work and draw conclusions. We close the chapter with an outlook to the future by identifying open issues and future research directions in Section 7.3.

† The last sentence of Turing's pioneering article, ***Computing Machinery and Intelligence***, Mind - A Quarterly Review of Psychology and Philosophy, Vol. 59, No. 236 (Oct., 1950), pp. 433-460, well-known as the *Turing Test*

[1] Criticality with a specific focus on the data communication in the application via the main memory.

7.1 Summary

In this dissertation, we have presented a dynamic profiling framework for extracting quantitative memory access information during the execution of an application. We employ Dynamic Binary Instrumentation (DBI) to inject the required analysis code into the application in order to track its runtime behaviour. To this purpose, we utilize an efficient approach to trace every single access to the main memory.

In Chapter 2, we introduced the concept of profiling as the most common method of application analysis. We set out stressing the importance of such analysis in understanding the behaviour of applications, which, in turn, is of great value to application developers and computer architects. Profiling tools are a necessity to evaluate how well applications perform on different platforms as well as to identify the critical parts which pose potential bottlenecks for the whole system performance. We also explained the idea of application tracing in contrast to profiling, and how their objectives differ in the context of application analysis. Furthermore, we elaborated on the different aspects in which profiling is useful. Subsequently, we listed different types of data structures that are used in various profiling and tracing techniques. The choice of proper data structures substantially affects the performance of the profiler in addition to the execution time of the profiled application itself. We described the two main categories of profiling, namely the static analysis and the dynamic analysis. From a different perspective, we subsequently looked at the differences between software and hardware profiling. In particular, we further explored instrumentation, which is used in our profiling toolset. The chapter ended with a brief account of several existing profiling tools used in analysing applications.

The *Quad* memory access profiling toolset was introduced in Chapter 3. In this chapter, we initially presented the project context of our work, focusing on the Molen Abstraction Layer (MAL), the *Delft Workbench* tool platform, and the Q^2 profiling framework. The chapter continued with a detailed description of the development issues in the dynamic part of the profiling framework. In particular, we elaborated on the description of the *Pin* DBI framework, and the implementation of the *Quad*-core tool, the Memory Access Tracing (MAT) module, Memory Access Intensity Profiler (*Maip*), and the *xQuad* tool. Furthermore, using the profiling information extracted by *Maip*, we set out to estimate the time spent on memory operations in distinction of the time spent on computations. Based on this estimation, we proposed a ranking strategy that provides a preliminary assessment of the criticality of a function regarding its memory access intensity. In order to demonstrate how the profiling information can be interpreted and used in different aspects, we presented three case studies with real world applications. In each case study, we highlighted the major observations followed by detailed comments.

Chapter 4 presented the *tQuad* tool that further extends the *Quad* toolset by enabling it to extract relative timing information from the application's execution. This is of critical significance, particularly with regard to scheduling and mapping tasks onto heterogeneous multicore systems. The original *Quad*-core tool provides no track of any temporal information, mainly due to instrumentation performance issues. The *tQuad* tool collects the relative timing profiles as an indication of the progress of the application. We presented a concise overview of how this functionality was implemented in *tQ uad*. In the presence of the memory access data, the extracted temporal profiles by *tQ*

UAD give an account of the memory bandwidth usage of functions in the application over time. Additionally, we utilized the extracted temporal information to discover the different (virtual) phases of the application. The chapter ended with a detailed case study of a real application to demonstrate the potential and the applicability of *tQUAD* in practice.

The focus of Chapter 5 was on the problem of the coarse-grained partitioning of an application in its general sense. We presented a detailed investigation into the factors that characterize a partitioning scheme and the methods that are utilized to perform partitioning. In addition to a comprehensive formulation of the general application partitioning problem, we proposed a heuristic approach to tackle this intractable problem with the aim of working out a near-optimal (or optimal) solution in a feasible amount of time. The proposed approach was based on the well-known greedy algorithm with the primary objective of minimizing (and maximizing) the inter-cluster (intra-cluster) data communication, and the uniformity of the workload in different Processing Elements (PEs). The proposed partitioning algorithm was bound to fail without a proper input model to fully capture the data transfers in the application. To address this critical issue, we utilized the QDU graph, provided by the *QUAD*-core tool, to drive the partitioning procedure. As customary in Computer Science, we gave an account of the detailed complexity analysis of the proposed partitioning, both in terms of time and space. Moreover, we presented a thorough analysis of the application partitioning problem from a combinatorial mathematics perspective. This was required to conduct an exhaustive search of the solution space in order to have a strictly accurate assessment of how close we can get to the optimal solution. The chapter was concluded with experimental results for a real application as well as for synthetic data in comparison with the optimal solution.

Finally, we demonstrated how the dynamic profiling framework can be applied in real scenarios by investigating two realistic cases in Chapter 6. For this purpose, we proposed the Q^2 partitioning approach which divides an application into hardware and software parts. We evaluated the Canny Edge Detection (CED) application, a well-known edge detection algorithm, and the Mixed Excitation Linear Prediction (MELP) application, a high-grade voice coder targeting very low bit rates. Both applications were mapped onto the Molen heterogeneous platform. To this purpose, an elaborate analysis of the applications was performed beforehand. During this analysis, memory access profiling information provided by *QUAD* was utilized for source code modifications and optimizations. We employed the QDU graph as the main reference to analyse the data transfers between functions, find memory bottlenecks and deficiencies, and spot opportunities to merge functions.

7.2 Main Contributions

In this dissertation, we have presented diverse results, ranging from practical usage of the developed toolset, to theoretical analysis of partitioning, and to simulations. In the following, we review the main contributions of our work.

- We developed an efficient memory access profiling framework that can be utilized to extract useful behavioral information from applications. Several profilers were

designed and implemented based on the *Pin* DBI framework. The QDU graph, as the primary output of the *QUAD* toolset, is introduced in Chapter 3. The graph demonstrates comprehensiveness with respect to the amount of information it provides to the application developers. The *QUAD* toolset contrasts with similar data dependency tools in various aspects: 1) the scope of the extracted information, 2) the accuracy of the profiling data, and 3) the flexibility and the potential for extension. These properties were validated in different scenarios and experiments throughout this work.

- In Chapter 4, we showed how relative timing data can be extracted in addition to the memory access profiles during the execution of the application. The extracted temporal information was used to give an overview of the memory bandwidth requirements of the application during its execution and, subsequently, guide the identification of different phases in the application.

- In this dissertation, we have utilized the QDU graph for different purposes. In Chapter 5, we discussed that the existing application partitioning approaches — which account for data communication — either use inappropriate input data models or are restricted to theoretical analysis of the partitioning problem. Instead, we formulated a general application partitioning problem, which can be used to model different objectives and, subsequently, proposed a heuristic approach as a solution.

- In addition to a thorough analysis of the exhaustive search for our partitioning model, we presented extensive simulation results using synthetically generated graphs to further substantiate the quality of solutions found by our partitioning algorithm. As demonstrated in Chapter 5, for the majority of cases, the solutions found by our algorithm lie in the top 5% of the solution space, either as the optimal or near-optimal solution.

- We have also demonstrated that the extracted profiles can be valuable in scenarios where legacy applications are mapped onto a heterogeneous platform. In Chapter 6, two realistic cases were given where the *QUAD* toolset played a crucial role in directing the application mapping. For this purpose, we proposed a HW/SW partitioning methodology based on the static and dynamic parts of the profiling framework. The obtained partitions resulted in a speedup of 2.92× for CED and 1.30× for MELP, respectively.

7.3 Research Opportunities

We envision different areas where our profiling framework can be used. Furthermore, the work presented in this dissertation can be improved in many ways. In the following, we list several research opportunities related to our work and we comment on how these opportunities may be addressed in the future.

Opportunity 1 — ***The adaptation of the QUAD dynamic memory access profiling toolset to analyse parallel applications.***

The current *QUAD* toolset is only capable to work with sequential applications. With a large repository of legacy applications, which are developed based on the *Von Neumann* model, our profiling framework still proves to be useful for coarse-grained parallelism identification, code optimizations, and application mapping onto heterogeneous architectures. Nevertheless, with multicore systems becoming the preferred technology for building powerful computers, it is anticipated that there will be an increasing availability of parallel applications. In this respect, software will be critical to leveraging parallel systems, as the time with hardware exploitation of parallelism is almost over. It is also clear that the main issue with parallel programming is data dependency. Multiple tasks work on the same data, in an unpredictable way, and that is why these data dependencies become difficult to handle in parallel applications. Instrumenting a parallel application is not fundamentally different from instrumenting sequential applications. As an example, the *Pin* DBI framework provides the required facilities for this purpose [18]. There are mechanisms to signal that a new thread or process is created. Analysis routines can be provided with a thread ID, so it is possible to attribute the extracted data, e.g., a memory access, to the thread that performed the operation. We envision the possibility of implementing our efficient MAT module as well as adapting the proposed QDU graph to accommodate parallel applications. However, instrumenting a multi-threaded application indeed has its own difficulties and requires some special care. In particular, the analysis routines can be called by multiple threads simultaneously, thus one has to make sure that the analysis routines are thread-safe.

Opportunity 2 — ***The extension of the QUAD toolset in terms of the extracted profiling information, the inspection granularity, and the performance optimizations.***

Although we have tried to make *QUAD* perform its tasks as efficient as possible, there should still be several opportunities to further revise the toolset. Considering the fact that *QUAD* was expected to be used as an extremely *heavyweight* Dynamic Binary Analysis (DBA) tool — incomparable with other tools utilizing the shadow memory concept due to the huge amount of data and processing involved in shadowing each address — several features were left out of the initial design. As an example, *QUAD* is only able to tell the user how much data is communicated between a pair of functions and what would be the actual size of memory needed for this communication, however, it is not able to reveal anything about the communicated data values. This can be indeed important when the user needs to fine-tune an application. An initial attempt to extract some useful information based on the concept of Unique Data Values (UnDVs) is presented in [13]. The extension enables *QUAD*-core to quantify the uniqueness of data values and how often they are accessed. Furthermore, the data dependency detection in *QUAD* is performed at the coarse granularity of function level. If finer granularity, such as loop or basic block level, is required, *QUAD* cannot be of much use. The investigation of implementing a flexible granularity level would be an interesting feature for the toolset. Another hint would be the detection of shared data objects in an application. Finally, it is worth to mention that some profiling data are extracted in the *QUAD*-core during the instrumentation process which simply are not post-processed to be considered as deliverable outputs, e.g., the complete range

of memory addresses in data transfers. The runtime overhead of *Quad* is still a critical performance issue. As a heavyweight DBA tool — which involves large amounts of profiling data that is accessed and updated in irregular patterns — the user experiences slowdowns on the scale of hundreds to thousands of times, depending on the nature of the application. This can be undoubtedly a major point of research in order to cut off extra overhead and boost the performance of the toolset.

Opportunity 3 — ***The investigation of utilizing other extracted profiling information in addition to the QDU graph for application partitioning.***
The information extracted by the *Quad* toolset is not limited to the data that is included in the QDU graph. There exists various memory access related data, such as Interleaving Balance Factor (IBF) and the phase information, which is not directly used in our application partitioning nor in mapping an application onto a target heterogeneous platform. The extraction of this diverse profiling information was made possible because of the powerful tracing mechanism that *Quad* provides for analyzing memory accesses. Nevertheless, the effect of this extensive collection of profiling information has not been investigated in practice, though we foresee an immense potential to positively affect the partitioning results. Regarding the proposed Q^2 partitioning methodology, there is an interest to examine how this information can help the user find better merging candidates or carry out function splitting — which was not yet addressed in our HW/SW partitioning. On a more general scale, the multi-objective clustering algorithm can benefit from an extended cost and ranking function definition based on this more detailed information. It will require a thorough investigation to assess the effect of each modification on the quality of the found partitions as well as the time needed to reach the solution. Moreover, this additional information can be used to define heuristic rules which in turn affect the clustering procedure. As an example, one may opt to prefer clustering two functions which are communicating in a tight interleaved production-consumption manner above functions where the whole production precedes the consumption. How this decision affects the performance of the mapped application in practice is one aspect that needs further investigation.

Opportunity 4 — ***The comparison of various heuristic application partitioning algorithms in terms of the result quality and the execution time.***
As mentioned in Chapter 5, one major problem regarding different application partitioning algorithms is the lack of a robust and fair basis of comparison. To this day, no approach has been proposed in literature that enables the comparison of the results of different partitioning strategies. This is due to the different input models, objective functions, assumptions, test cases, and target architectures that are used in each research work. The diversity of critical factors in these works is such that one would never be able to prefer one over another. Even worse, there is no standard metric in order to assess the quality of the results. Apart from limited works that propose deterministic methods to find the optimal solution[2], for the heuristic methods no solid proof is given to validate the quality of solutions. The

[2] Only feasible when the problem size is small or the problem formulation makes it possible to work out a polynomial time complexity solution, such as ILP.

synthetic test bench that we provided in this work can be used as a starting point in this respect.

Bibliography

[1] *Abdelzaher, T. F., and Shin, K. G.* Period-based load partitioning and assignment for large real-time applications. *IEEE Transactions on Computers 49*, 1 (January 2000), 81–87.

[2] Accellera systems initiative. http://www.accellera.org/.

[3] *Adams, J. K., and Thomas, D. E.* Multiple-process behavioral synthesis for mixed hardware-software systems. In *Proceedings of the 8th international symposium on System synthesis* (New York, NY, USA, 1995), ISSS '95, ACM, pp. 10–15.

[4] *Ahmadinia, A., Bobda, C., Koch, D., Majer, M., and Teich, J.* Task scheduling for heterogeneous reconfigurable computers. In *Proceedings of the 17th symposium on Integrated circuits and system design* (New York, NY, USA, 2004), ACM, pp. 22–27.

[5] aiT Worst-Case Execution Time Analyzers. http://www.absint.com/ait/.

[6] Altera corporation. http://www.altera.com/.

[7] SGI Builds World's Largest FPGA Supercomputer. http://www.sgi.com/company_info/newsroom/press_releases/2007/november/fpga.html.

[8] *Amdahl, G. M.* Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies Conf.* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[9] *Anderson, D.* A Consumer Library Interface to DWARF, rev. 2.02.

[10] *Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E.* Continuous profiling: where have all the cycles gone? In *Proceedings of the 16th ACM symposium on Operating systems principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 1–14.

[11] *Arató, P., Mann, Z. A., and Orbán, A.* Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 10*, 1 (January 2005), 136–156.

[12] *Arató, P., Mann, Z. A., Orbán, A., and Papp, D.* Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing* (September 2003), pp. 197–202.

[13] *Ashraf, I., Ostadzadeh, S. A., Meeuws, R., and Bertels, K.* Communication-aware HW/SW co-design for heterogeneous multicore platforms. In *Proceedings of the 2012 Workshop on Dynamic Analysis* (New York, NY, USA, 2012), WODA 2012, ACM, pp. 36–41.

[14] *Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A.* Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience 23*, 2 (2011), 187–198.

[15] *Ayed, M.* *Automatic Code Partitioning for Distributed-Memory Multiprocessors (DMMs).* PhD thesis, University of Southern California, Los Angeles, CA, USA, 1996.

[16] *Ayed, M., and Gaudiot, J.-L.* Analysis of a heuristic for code partitioning. *The Journal of Supercomputing 12*, 3 (May 1998), 191–226.

[17] *Ayed, M., and Gaudiot, J.-L.* An efficient heuristic for code partitioning. *Parallel Computing 26*, 4 (2000), 399 – 426.

[18] *Bach, M. M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.-K., Lyons, G., Patil, H., and Tal, A.* Analyzing Parallel Programs with Pin. *Computer 43*, 3 (March 2010), 34–41.

[19] *Balarin, F., Lavagno, L., Murthy, P., and Sangiovanni-vincentelli, A.* Scheduling for embedded real-time systems. *IEEE Design and Test of Computers 15*, 1 (January 1998), 71–82.

[20] *Baleani, M., Gennari, F., Jiang, Y., Patel, Y., Brayton, R. K., and Sangiovanni-Vincentelli, A.* Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the tenth international symposium on Hardware/software codesign* (New York, NY, USA, 2002), CODES '02, ACM, pp. 151–156.

[21] *Ball, T., and Larus, J. R.* Branch prediction for free. *ACM SIGPLAN Notices 28*, 6 (June 1993), 300–313.

[22] *Ball, T., and Larus, J. R.* Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*, 4 (July 1994), 1319–1360.

[23] *Ball, T., and Larus, J. R.* Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture* (Washington, DC, USA, 1996), MICRO 29, IEEE Computer Society, pp. 46–57.

[24] *Ball, T., Mataga, P., and Sagiv, M.* Edge profiling versus path profiling: the showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 134–148.

[25] *Balle, S., and Steely, S.* Memory access profiling tools for alpha-based architectures. In *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, B. Kågström, J. Dongarra, E. Elmroth, and J. Wasniewski, Eds., vol. 1541 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998, pp. 28–37.

[26] *Bammi, J. R., Kruijtzer, W., Lavagno, L., Harcourt, E., and Lazarescu, M. T.* Software performance estimation strategies in a system-level design tool. In *Proceedings of the 8th international workshop on Hardware/software codesign* (New York, NY, USA, 2000), CODES '00, ACM, pp. 82–86.

[27] *Banerjee, S., and Dutt, N.* Efficient search space exploration for hw-sw partitioning. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis* (New York, NY, USA, 2004), CODES+ISSS '04, ACM, pp. 122–127.

[28] *Barros, E., and Sampaio, A.* Towards provably correct hardware/software partitioning using OCCAM. In *Proceedings of the 3rd International Workshop on Hardware/Software Codesign* (September 1994), pp. 210–217.

[29] *Barua, R., and Kotha, A.* Automatic parallelization using binary rewriting, April 2010. US Patent App. 12/771,460.

[30] *Beck, A. C. S., Rutzig, M. B., Gaydadjiev, G., and Carro, L.* Transparent reconfigurable acceleration for heterogeneous embedded applications. In *Proceedings of the conference on Design, automation and test in Europe* (New York, NY, USA, 2008), DATE '08, ACM, pp. 1208–1213.

[31] *Benini, L., Bogliolo, A., and De Micheli, G.* A survey of design techniques for system-level dynamic power management. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 8*, 3 (june 2000), 299–316.

[32] *Berkhout, A. J., de Vries, D., and Vogel, P.* Acoustic control by wave field synthesis. *The Journal of the Acoustical Society of America 93*, 5 (1993), 2764–2778.

[33] *Bertels, K., Sima, V., Meeuws, R., Ostadzadeh, S. A., Galuzzi, C., Nane, R., Yankova, Y., Mariani, G., and Kuzmanov, G.* Hardware/Software Co-design for Polymorphic Processors : The Delft Workbench. Tech. rep., Computer Engineering, Delft University of Technology, Delft, the Netherlands, January 2012.

[34] *Bhattacharya, A., Konar, A., Das, S., Grosan, C., and Abraham, A.* Hardware software partitioning problem in embedded system design using particle swarm optimization algorithm. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems* (Washington, DC, USA, 2008), CISIS '08, IEEE Computer Society, pp. 171–176.

[35] *Bình, N. N., Imai, M., Shiomi, A., and Hikichi, N.* A hardware/software partitioning algorithm for designing pipelined asips with least gate counts. In *Proceedings of the 33rd annual Design Automation Conference* (New York, NY, USA, 1996), DAC '96, ACM, pp. 527–532.

[36] *Błażewicz, J., Drabowski, M., and Weglarz, J.* Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers 35*, 5 (May 1986), 389–393.

[37] *Bond, M. D., and McKinley, K. S.* Continuous path and edge profiling. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2005), MICRO 38, IEEE Computer Society, pp. 130–140.

[38] *Bond, M. D., and McKinley, K. S.* Practical path profiling for dynamic optimizers. In *Proceedings of the international symposium on Code generation and optimization* (Washington, DC, USA, March 2005), CGO '05, IEEE Computer Society, pp. 205–216.

[39] *Bondhugula, U.* Automatic distributed-memory parallelization and code generation using the polyhedral framework. Tech. Rep. IISc-CSA-TR-2011-3, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India, September 2011.

[40] *Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P.* A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices - PLDI '08 43*, 6 (June 2008), 101–113.

[41] *Borkar, S. Y., Dubey, P., Kahn, K. C., Kuck, D. J., Mulder, H., Pawlowski, S. S., and Rattner, J. R.* Platform 2015: Intel processor and platform evolution for the next decade. *Intelligence/SIGART Bulletin* (2005).

[42] Bound-T Time and Stack Analyser. http://www.bound-t.com/.

[43] *Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F.* 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 (June 2001), 810–837.

[44] *Brewer, O., Dongarra, J., and Sorensen, D.* Tools to aid in the analysis of memory access patterns for fortran programs. *Parallel Computing 9*, 1 (1988), 25 – 35.

[45] *Buck, B., and Hollingsworth, J. K.* An api for runtime code patching. *International Journal of High Performance Computing Applications 14*, 4 (November 2000), 317–329.

[46] *Burger, D., and Austin, T. M.* The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News 25*, 3 (June 1997), 13–25.

[47] *Calder, B., Feller, P., and Eustace, A.* Value profiling. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (Washington, DC, USA, December 1997), MICRO 30, IEEE Computer Society, pp. 259–269.

[48] *Canny, J.* A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence 8*, 6 (November 1986), 679–698.

[49] *Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H.* Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), ATEC '04, USENIX Association, pp. 2–2.

[50] *Chamberlain, B. L.* Graph partitioning algorithms for distributing workloads of parallel computations. Tech. Rep. UW-CSE-98-10, University of Washington, 1998.

[51] *Chatha, K. S., and Vemuri, R.* Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *Proceedings of the 9th international symposium on Hardware/software codesign* (New York, NY, USA, 2001), CODES '01, ACM, pp. 42–47.

[52] *Chatha, K. S., and Vemuri, R.* Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 10*, 3 (June 2002), 193–208.

[53] *Choudhury, A. N. M. I., Potter, K. C., and Parker, S. G.* Interactive visualization for memory reference traces. *Computer Graphics Forum 27*, 3 (2008), 815–822.

[54] *Chung, E. S., Hoe, J. C., and Mai, K.* CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (New York, NY, USA, 2011), FPGA '11, ACM, pp. 97–106.

[55] *Clarke, E.* The birth of model checking. In *25 Years of Model Checking*, O. Grumberg and H. Veith, Eds., vol. 5000 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 1–26.

[56] AMD CodeAnalyst Performance Analyzer. http://developer.amd.com/tools/CodeAnalyst/Pages/default.aspx.

[57] *Cohen, A., Daubechies, I., and Feauveau, J.-C.* Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics 45*, 5 (1992), 485–560.

[58] *Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.* *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[59] CoSy Compiler Development System. http://www.ace.nl/compiler/cosy.html.

[60] *Curtin, M.* *Brute Force: Cracking the Data Encryption Standard.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[61] *Dasdan, A., and Aykanat, C.* Two novel multiway circuit partitioning algorithms using relaxed locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16*, 2 (feb 1997), 169–178.

[62] *Davis, R. I., and Burns, A.* A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys 43*, 4 (October 2011), 35:1–35:44.

[63] *Denning, P. J.* The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333.

[64] *Dias, M. A., Sales, D. O., and Osorio, F. S.* A profile-based method for hardware/-software co-design applied in evolutionary robotics using reconfigurable computing. In *Proceedings of the 2010 IEEE Electronics, Robotics and Automotive Mechanics Conference* (Washington, DC, USA, 2010), CERMA '10, IEEE Computer Society, pp. 463–468.

[65] *Dick, R., and Jha, N.* Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17*, 10 (October 1998), 920–935.

[66] *Dmitriev, M.* Design of jfluid: a profiling technology and tool based on dynamic bytecode instrumentation. Tech. rep., Mountain View, CA, USA, 2003.

[67] The DWARF Debugging Standard. http://dwarfstd.org/.

[68] *Eager, M. J.* Introduction to the DWARF Debugging Format.

[69] *Edwards, S., Lavagno, L., Lee, E., and Sangiovanni-Vincentelli, A.* Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE 85*, 3 (March 1997), 366 –390.

[70] *Edwards, S., Lavagno, L., Lee, E. A., and Sangiovanni-Vincentelli, A.* Design of embedded systems: formal models, validation, and synthesis. In *Readings in hardware/software co-design*, G. De Micheli, R. Ernst, and W. Wolf, Eds. Kluwer Academic Publishers, Norwell, MA, USA, 2002, pp. 86–107.

[71] *Eeckhout, L., and Bosschere, K. D.* Efficient simulation of trace samples on parallel machines. *Parallel Computing 30*, 3 (2004), 317–335.

[72] *Eeckhout, L., de Bosschere, K., and Neefs, H.* Performance analysis through synthetic trace generation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (Washington, DC, USA, 2000), ISPASS '00, IEEE Computer Society, pp. 1–6.

[73] *Eles, P., Kuchcinski, K., Peng, Z., and Doboli, A.* Hardware/software partitioning of VHDL system specifications. In *Proceedings of the Conference on European Design Automation* (Los Alamitos, CA, USA, 1996), EURO-DAC '96/EURO-VHDL '96, IEEE Computer Society Press, pp. 434–439.

[74] *Eles, P., Peng, Z., Kuchcinski, K., and Doboli, A.* System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems 2* (1997), 5–32. 10.1023/A:1008857008151.

[75] *Emerson, M., Neema, S., and Sztipanovits, J.* *Handbook of Real-Time and Embedded Systems*. CRC Press, 2006. ISBN: 1584886781.

[76] *Ernst, R., Henkel, J., and Benner, T.* Hardware-software cosynthesis for micro-controllers. *IEEE Design Test of Computers 10*, 4 (October 1993), 64–75.

[77] *Faxén, K.-F., Popov, K., Jansson, S., and Albertsson, L.* Embla - Data Dependence Profiling for Parallel Programming. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems* (Washington, DC, USA, 2008), CISIS '08, IEEE Computer Society, pp. 780–785.

[78] *Fiduccia, C. M., and Mattheyses, R. M.* A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference* (Piscataway, NJ, USA, 1982), DAC '82, IEEE Press, pp. 175–181.

[79] *Frakes, W., and Kang, K.* Software reuse research: status and future. *IEEE Transactions on Software Engineering 31*, 7 (July 2005), 529–536.

[80] Fraunhofer Institute for Digital Media Technology. `http://www2.idmt.fraunhofer.de/eng/research_topics/wave_field_synthesis.htm`.

[81] *Fredkin, E.* Trie memory. *Communications of the ACM (CACM) 3*, 9 (September 1960), 490–499.

[82] *Gajski, D. D., Vahid, F., Narayan, S., and Gong, J.* Specsyn: an environment supporting the specify-explore-refine paradigm for hardware/software system design. In *Readings in hardware/software co-design*, G. De Micheli, R. Ernst, and W. Wolf, Eds. Kluwer Academic Publishers, Norwell, MA, USA, 2002, pp. 108–124.

[83] *Garey, M., Johnson, D., and Stockmeyer, L.* Some simplified np-complete graph problems. *Theoretical Computer Science 1*, 3 (1976), 237–267.

[84] *Gerasoulis, A., and Yang, T.* On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems 4*, 6 (June 1993), 686–701.

[85] *Gesellensetter, L., and Glesner, S.* Interprocedural speculative optimization of memory accesses to global variables. In *Euro-Par 2008 - Parallel Processing*, E. Luque, T. Margalef, and D. Benítez, Eds., vol. 5168 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 350–359.

[86] *Giusto, P., Martin, G., and Harcourt, E.* Reliable estimation of execution time of embedded software. In *Proceedings of the conference on Design, automation and test in Europe* (Piscataway, NJ, USA, 2001), DATE '01, IEEE Press, pp. 580–589.

[87] *Gohringer, D., Hubner, M., Benz, M., and Becker, J.* A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip. In *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Los Alamitos, CA, USA, May 2010), IEEE Computer Society, pp. 259–262.

[88] *Gonen, R., and Lehmann, D.* Optimal solutions for multi-unit combinatorial auctions: branch and bound heuristics. In *Proceedings of the 2nd ACM conference on Electronic commerce* (New York, NY, USA, 2000), ACM, pp. 13–20.

[89] *Graham, S. L., Kessler, P. B., and Mckusick, M. K.* Gprof: A call graph execution profiler. *ACM SIGPLAN Notices - Proceedings of the SIGPLAN symposium on Compiler construction 17*, 6 (June 1982), 120–126.

[90] Graphviz - Graph Visualization Software. http://www.graphviz.org/.

[91] *Grode, J., Knudsen, P. V., and Madsen, J.* Hardware resource allocation for hardware/software partitioning in the lycos system. In *Proceedings of the conference on Design, automation and test in Europe* (Washington, DC, USA, 1998), DATE '98, IEEE Computer Society, pp. 22–27.

[92] *Guo, Z., Buyukkurt, B., and Najjar, W.* Input data reuse in compiling window operations onto reconfigurable hardware. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2004), LCTES '04, ACM, pp. 249–256.

[93] *Gupta, R. K., and De Micheli, G.* Hardware-software cosynthesis for digital systems. *IEEE Design Test of Computers 10*, 3 (July 1993), 29–41.

[94] *Henkel, J.* A low power hardware/software partitioning approach for core-based embedded systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference* (New York, NY, USA, 1999), DAC '99, ACM, pp. 122–127.

[95] *Henkel, J., and Ernst, R.* An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9*, 2 (April 2001), 273–290.

[96] *Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G.* Software verification with blast. In *Proceedings of the 10th International Conference on Model Checking Software* (Berlin, Heidelberg, 2003), SPIN'03, Springer-Verlag, pp. 235–239.

[97] Heptane WCET Analysis Tool. http://www.irisa.fr/alf/index.php?option=com_content&view=article&id=29&Itemid=&lang=en.

[98] *Hollingsworth, J. K., Niam, O., Miller, B. P., Xu, Z., Goncalves, M. J. R., and Zheng, L.* Mdl: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, November 1997), PACT '97, IEEE Computer Society, pp. 201–212.

[99] *Huang, X., Yu, H., and Zhang, W.* Nems based thermal management for 3D many-core system. In *Proceedings of the IEEE/ACM International Symposium on Nanoscale Architectures* (Washington, DC, USA, 2011), NANOARCH '11, IEEE Computer Society, pp. 218–223.

[100] *Hulton, D., and Pellerin, D.* Accelerating cryptography with FPGA clusters. online. http://www.mil-embedded.com/articles/id/?4724, June 2010.

[101] *Hundt, R.* Hp caliper: A framework for performance analysis tools. *IEEE Concurrency 8*, 4 (October 2000), 64–71.

[102] *Hwang, E., Vahid, F., and Hsu, Y.-C.* Fsmd functional partitioning for low power. In *Proceedings of the conference on Design, automation and test in Europe* (New York, NY, USA, 1999), DATE '99, ACM.

[103] *Jackson, D., and Rinard, M.* Software analysis: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (New York, NY, USA, 2000), ICSE '00, ACM, pp. 133–145.

[104] *Jantsch, A., Ellervee, P., Hemani, A., Öberg, J., and Tenhunen, H.* Hardware/-software partitioning and minimizing memory interface traffic. In *Proceedings of the Conference on European Design Automation* (Los Alamitos, CA, USA, 1994), EURO-DAC '94, IEEE Computer Society Press, pp. 226–231.

[105] *Jigang, W., Srikanthan, T., and Chen, G.* Algorithmic aspects of hardware/software partitioning: 1D search algorithms. *IEEE Transactions on Computers 59*, 4 (April 2010), 532–544.

[106] *Joshi, R., Bond, M. D., and Zilles, C.* Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (Washington, DC, USA, March 2004), CGO '04, IEEE Computer Society, pp. 239–250.

[107] *Kalavade, A., and Lee, E. A.* The extended partitioning problem: Hardware/software mapping, scheduling, and implementation-bin selection. *Design Automation for Embedded Systems 2* (1997), 125–163. 10.1023/A:1008872518365.

[108] *Kalavade, A., and Subrahmanyam, P.* Hardware/software partitioning for multi-function systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17*, 9 (September 1998), 819–837.

[109] *Kalavade, A. P.* *System-level codesign of mixed hardware-software systems.* PhD thesis, University of California, Berkeley, 1995.

[110] *Karuri, K., Al Faruque, M. A., Kraemer, S., Leupers, R., Ascheid, G., and Meyr, H.* Fine-grained application source code profiling for ASIP design. In *Proceedings of the 42nd annual Design Automation Conference* (New York, NY, USA, 2005), DAC '05, ACM, pp. 329–334.

[111] *Kernighan, B. W., and Lin, S.* An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal 49*, 2 (1970), 291–307.

[112] *Khan, O., and Kundu, S.* Hardware/software co-design architecture for thermal management of chip multiprocessors. In *Proceedings of the Conference on Design, Automation and Test in Europe* (3001 Leuven, Belgium, Belgium, 2009), DATE '09, European Design and Automation Association, pp. 952–957.

[113] *Knudsen, P. V., and Madsen, J.* Pace: A dynamic programming algorithm for hardware/software partitioning. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design* (Washington, DC, USA, 1996), CODES '96, IEEE Computer Society, pp. 85–92.

[114] Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[115] Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[116] Kooti, H., Bozorgzadeh, E., Liao, S., and Bao, L. Transition-aware real-time task scheduling for reconfigurable embedded systems. In *Design, Automation Test in Europe Conference (DATE)* (march 2010), pp. 232–237.

[117] Kornaros, G. *Multi-Core Embedded Systems*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 2010.

[118] Kufrin, R. Measuring and improving application performance with perfsuite. *Linux Journal 2005*, 135 (July 2005), 62–64.

[119] Kuon, I., Tessier, R., and Rose, J. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation 2*, 2 (February 2008), 135–253.

[120] Kwok, Y.-K., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys 31*, 4 (December 1999), 406–471.

[121] LaPedus, M. Plds jockey to set new lows in cost, power budgets. online. http://www.eetimes.com/news/semi/showArticle.jhtml?articleID=208401267, June 2008.

[122] Larus, J. Efficient program tracing. *Computer 26*, 5 (May 1993), 52–61.

[123] Larus, J. Programming clouds. In *Compiler Construction*, R. Gupta, Ed., vol. 6011 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 1–9.

[124] Larus, J. R. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation* (New York, NY, USA, 1999), PLDI '99, ACM, pp. 259–269.

[125] Leavens, G., Leino, K., and Müller, P. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing 19*, 2 (2007), 159–189.

[126] Lee, H. B., and Zorn, B. G. BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (Berkeley, CA, USA, 1997), USITS'97, USENIX Association, pp. 7–16.

[127] Li, X., Liang, Y., Mitra, T., and Roychoudhury, A. Chronos: A timing analyzer for embedded software. *Science of Computer Programming 69*, 1-3 (2007), 56–67.

[128] Li, Y., Callahan, T., Darnell, E., Harr, R., Kurkure, U., and Stockwood, J. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the 37th Annual Design Automation Conference* (New York, NY, USA, 2000), DAC '00, ACM, pp. 507–512.

[129] *López-Vallejo, M., Iglesias, C. A., and López, J. C.* A knowledge-based system for hardware-software partitioning. In *Proceedings of the conference on Design, automation and test in Europe* (Washington, DC, USA, February 1998), DATE '98, IEEE Computer Society, pp. 914–915.

[130] *López-Vallejo, M., and López, J. C.* Multi-way clustering techniques for system level partitioning. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference* (2001), pp. 242–247.

[131] *López-Vallejo, M., and López, J. C.* On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 8*, 3 (July 2003), 269–297.

[132] *López-Vallejo, M. L., Grajal, J., and López, J. C.* Constraint-driven system partitioning. In *Proceedings of the conference on Design, Automation and Test in Europe* (New York, NY, USA, 2000), DATE '00, ACM, pp. 411–416.

[133] *Lu, Y., Marconi, T., Bertels, K., and Gaydadjiev, G.* A Communication Aware Online Task Scheduling Algorithm for FPGA-Based Partially Reconfigurable Systems. In *Processidings of 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (May 2010), pp. 65–68.

[134] *Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K.* Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[135] *Madsen, J., Grode, J., Knudsen, P., Petersen, M., and Haxthausen, A.* Lycos: the lyngby co-synthesis system. *Design Automation for Embedded Systems 2* (1997), 195–235. 10.1023/A:1008884219274.

[136] *Mann, Z., OrbáN, A., and Farkas, V.* Evaluating the kernighan-lin heuristic for hardware/software partitioning. *International Journal of Applied Mathematics and Computer Science 17*, 2 (June 2007), 249–267.

[137] *Mann, Z. A.* *Partitioning algorithms for hardware/software co-design.* PhD thesis, Budapest University of Technology and Economics, Department of Control Engineering and Information Technology, 2004.

[138] *Mann, Z. A., Orbán, A., and Arató, P.* Finding optimal hardware/software partitions. *Formal Methods in System Design 31*, 3 (December 2007), 241–263.

[139] *Marsaglia, G., and Tsang, W.* The Ziggurat Method for Generating Random Variables. *Journal of Statistical Software 5*, 8 (2000), 1–7.

[140] *Marsaglia, G., and Tsang, W. W.* A simple method for generating gamma variables. *ACM Transactions on Mathematical Software (TOMS) 26*, 3 (September 2000), 363–372.

[141] Martonosi, M., Gupta, A., and Anderson, T. MemSpy: analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (New York, NY, USA, 1992), SIGMETRICS '92/PERFORMANCE '92, ACM, pp. 1–12.

[142] Meeuws, R. J. *Quantitative Hardware Prediction Modeling for Hardware/Software Co-design*. PhD thesis, Delft University of Technology, Delft, the Netherlands, July 2012.

[143] Mei, B., Schaumont, P., and Vernalde, S. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proceedings of the Annual Workshop on Circuits, Systems and Signal Processing (ProRISC '10)* (November 2000).

[144] 2.4 kbps MELP Proposed Federal Standard speech coder. http://www.data-compression.com/melp1.2.tar.gz.

[145] Mencer, O., Platzner, M., Morf, M., and Flynn, M. Object-oriented domain specific compilers for programming FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9*, 1 (February 2001), 205–210.

[146] Merten, M. C., Trick, A. R., George, C. N., Gyllenhaal, J. C., and Hwu, W.-m. W. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th annual international symposium on Computer architecture* (Washington, DC, USA, 1999), ISCA '99, IEEE Computer Society, pp. 136–147.

[147] Merten, M. C., Trick, A. R., Nystrom, E. M., Barnes, R. D., and Hmu, W.-m. W. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proceedings of the 27th annual international symposium on Computer architecture* (New York, NY, USA, 2000), ISCA '00, ACM, pp. 59–70.

[148] Nahapetian, A., Brisk, P., Ghiasi, S., and Sarrafzadeh, M. An approximation algorithm for scheduling on heterogeneous reconfigurable resources. *ACM Transactions on Embedded Computing Systems (TECS) 9*, 1 (October 2009), 5:1–5:20.

[149] Narayanasamy, S., Sherwood, T., Sair, S., Calder, B., and Varghese, G. Catching accurate profiles in hardware. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2003), HPCA '03, IEEE Computer Society, pp. 269–280.

[150] Nethercote, N., and Seward, J. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)* (June 2007), pp. 65–74.

[151] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[152] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[153] *Niemann, R.* *Hardware/Software CO-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[154] *Niemann, R., and Marwedel, P.* An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems 2* (1997), 165–193. 10.1023/A:1008832202436.

[155] *Noordergraaf, L., and Zak, R.* Smp system interconnect instrumentation for performance analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 2002), Supercomputing '02, IEEE Computer Society Press, pp. 1–9.

[156] *O'Nils, M., Jantsch, A., Hemani, A., and Tenhunen, H.* Interactive hardware-software partitioning and memory allocation based on data transfer profiling. In *International Conference on Recent Advances in Mechantronics (ICRAM '95)* (August 1995), pp. 447–452.

[157] The OpenCL Specification, version 1.2 (revision 15), Khronos OpenCL Working Group. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, November 2011.

[158] The OpenMP API specification for parallel programming, version 3.1, OpenMP Architecture Review Board. http://www.openmp.org/mp-documents/OpenMP3.1.pdf, July 2011.

[159] Celoxica Unveils FPGA Acceleration Solution for AMD Opteron Processor-Based Systems. http://www.soccentral.com/results.asp?EntryID=18790.

[160] *Panait, V.-M., Sasturkar, A., and Wong, W.-F.* Static identification of delinquent loads. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 303–314.

[161] *Panda, P. R., Dutt, N. D., and Nicolau, A.* On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 5*, 3 (July 2000), 682–704.

[162] *Parameswaran, S., Parkinson, M. F., and Bartlett, P.* Profiling in the ASP codesign environment. *Journal of Systems Architecture 46*, 14 (2000), 1263 – 1274.

[163] *Pardalos, P. M., and Rodgers, G. P.* A branch and bound algorithm for the maximum clique problem. *Computers & Operations Research 19*, 5 (1992), 363–375.

[164] *Patterson, J. R. C.* Accurate static branch prediction by value range propagation. *ACM SIGPLAN Notices 30*, 6 (June 1995), 67–78.

[165] *Perelman, E., Chilimbi, T., and Calder, B.* Variational path profiling. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2005), PACT '05, IEEE Computer Society, pp. 7–16.

[166] *Peri, R., Jinturkar, S., and Fajardo, L.* A novel technique for profiling programs in embedded systems. In *ACM Workshop on Feedback-Directed and Dynamic Optimization* (1999).

[167] *Pham, C., Al-Ars, Z., and Bertels, K.* Rule-based data communication optimization using quantitative communication profiling. In *Proceedings of the International Conference on Field-Programmable Technology* (Seoul, Korea, December 2012).

[168] *Popeea, C., Xu, D. N., and Chin, W.-N.* A practical and precise inference and specializer for array bound checks elimination. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (New York, NY, USA, 2008), PEPM '08, ACM, pp. 177–187.

[169] *Prakash, S., and Parker, A. C.* Sos: synthesis of application-specific heterogeneous multiprocessor systems. In *Readings in hardware/software co-design*, G. De Micheli, R. Ernst, and W. Wolf, Eds. Kluwer Academic Publishers, Norwell, MA, USA, 2002, pp. 324–337.

[170] *Quan, G., Hu, X., and Greenwood, G.* Preference-driven hierarchical hardware/-software partitioning. In *International Conference on Computer Design (ICCD '99)* (1999), pp. 652–657.

[171] Intel QuickAssist Accelerator Technology for Embedded Systems. http://www.intel.com/p/en_US/embedded/hwsw/technology/quickassist.

[172] RapiTime - Rapita Systems). http://www.rapitasystems.com/products/RapiTime.

[173] *Rose, J., El Gamal, A., and Sangiovanni-Vincentelli, A.* Architecture of field-programmable gate arrays. *Proceedings of the IEEE 81*, 7 (July 1993), 1013–1029.

[174] *Saab, Y. G.* A fast and robust network bisection algorithm. *IEEE Transactions on Computers 44*, 7 (July 1995), 903–913.

[175] *Santambrogio, M. D., Memik, S. O., Rana, V., Acar, U. A., and Sciuto, D.* A novel soc design methodology combining adaptive software and reconfigurable hardware. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design* (Piscataway, NJ, USA, November 2007), ICCAD '07, IEEE Press, pp. 303–308.

[176] *Sarkar, V. Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989.

[177] *Sarkar, V.* Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev. 35*, 5-6 (September 1991), 779–804.

[178] *Sarkar, V., and Hennessy, J.* Compile-time partitioning and scheduling of parallel programs. *ACM SIGPLAN Notices 21*, 7 (July 1986), 17–26.

[179] *Sarkar, V., and Hennessy, J.* Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM conference on LISP and functional programming* (New York, NY, USA, 1986), LFP '86, ACM, pp. 202–211.

[180] *Shengchao, Q., and Jifeng, H.* An algebraic approach to hardware/software partitioning. In *Proceedings of the 7th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (2000), vol. 1, pp. 273–276.

[181] *Sinnen, O.* *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, 2007.

[182] SpecC system. http://www.cecs.uci.edu/~specc/.

[183] *Spiegel, A.* *Automatic distribution of object-oriented programs.* PhD thesis, Freie Universität Berlin, Universitätsbibliothek, 2002.

[184] *Srinivasan, V., Govindarajan, S., and Vemuri, R.* Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9*, 1 (February 2001), 140–159.

[185] *Srinivasan, V., Radhakrishnan, S., and Vemuri, R.* Hardware/software partitioning with integrated hardware design space exploration. In *Proceedings of the conference on Design, automation and test in Europe* (Washington, DC, USA, 1998), DATE '98, IEEE Computer Society, pp. 28–35.

[186] *Srivastava, A., and Eustace, A.* ATOM: a system for building customized program analysis tools. *ACM SIGPLAN Notices - Best of PLDI 1979-1999 39*, 4 (April 2004), 528–539.

[187] *Stitt, G., Lysecky, R., and Vahid, F.* Dynamic hardware/software partitioning: a first approach. In *Proceedings of the 40th annual Design Automation Conference* (New York, NY, USA, 2003), DAC '03, ACM, pp. 250–255.

[188] *Sun, Q., Zhao, J., and Chen, Y.* Probabilistic points-to analysis for java. In *Proceedings of the 20th International Conference on Compiler Construction: part of the joint European conferences on theory and practice of software* (Berlin, Heidelberg, 2011), CC'11/ETAPS'11, Springer-Verlag, pp. 62–81.

[189] *Supplee, L. M., Cohn, R. P., Collura, J. S., and McCree, A. V.* MELP: the new federal standard at 2400 bps. *IEEE International Conference on Acoustics Speech and Signal Processing* (1997), 1591–1594.

[190] *Suresh, D. C., Najjar, W. A., Vahid, F., Villarreal, J. R., and Stitt, G.* Profiling tools for hardware/software partitioning of embedded applications. In *Proceedings of the ACM SIGPLAN conference on Language, Compilers, and Tools for Embedded Systems* (New York, NY, USA, 2003), LCTES '03, ACM, pp. 189–198.

[191] SWEET (SWEdish Execution Time tool). http://www.mrtc.mdh.se/projects/wcet/sweet.html.

[192] SymTA/P Tool). http://www.ida.ing.tu-bs.de/forschung/projekte/symtap/.

[193] Systemverilog. http://www.systemverilog.org/.

[194] *Tahaee, S.-A., and Jahangir, A. H.* A polynomial algorithm for partitioning problems. *ACM Transactions on Embedded Computing Systems (TECS) 9*, 4 (April 2010), 34:1–34:38.

[195] *Tahaee, S. A., Jahangir, A. H., and Habibi-Masouleh, H.* Improving the performance of heuristic searches with judicious initial point selection. In *Proceedings of the 5th IEEE International Symposium on Embedded Computing* (Washington, DC, USA, 2008), SEC '08, IEEE Computer Society, pp. 14–19.

[196] Supercomputing with nvidia tesla gpus. http://www.nvidia.com/object/tesla_computing_solutions.html.

[197] *Todman, T., Constantinides, G., Wilton, S., Mencer, O., Luk, W., and Cheung, P.* Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings - 152*, 2 (March 2005), 193–207.

[198] *Topcuouglu, H., Hariri, S., and Wu, M.-y.* Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst. 13*, 3 (March 2002), 260–274.

[199] *Uh, G.-R., Cohn, R., Yadavalli, B., Peri, R., and Ayyagari, R.* Analyzing dynamic binary instrumentation overhead. In *Proceedings of the Workshop on Binary Instrumentation and Application (WBIA) at International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2007)* (San Jose, CA, USA, October 2007).

[200] Canny Edge Detector, Image Analysis Research Lab., USF. http://marathon.csee.usf.edu/edge/edge_detection.html.

[201] *Vahid, F.* Modifying min-cut for hardware and software functional partitioning. In *Proceedings of the 5th International Workshop on Hardware/Software Co-Design* (Washington, DC, USA, 1997), CODES '97, IEEE Computer Society, pp. 43–48.

[202] *Vahid, F.* Partitioning sequential programs for cad using a three-step approach. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 7*, 3 (July 2002), 413–429.

[203] *Vahid, F.* It's time to stop calling circuits "hardware". *Computer 40*, 9 (sept. 2007), 106–108.

[204] *Vahid, F.* What is hardware/software partitioning? *SIGDA Newsl. 39*, 6 (June 2009), 1–1.

[205] *Vahid, F., and Gajski, D. D.* Specification partitioning for system design. In *Proceedings of the 29th ACM/IEEE Design Automation Conference* (Los Alamitos, CA, USA, 1992), DAC '92, IEEE Computer Society Press, pp. 219–224.

[206] *Vahid, F., and Gajski, D. D.* Closeness metrics for system-level functional partitioning. In *Proceedings of the Conference on European Design Automation* (Los Alamitos, CA, USA, 1995), EURO-DAC '95/EURO-VHDL '95, IEEE Computer Society Press, pp. 328–333.

[207] VAHID, F., AND GAJSKI, D. D. Clustering for improved system-level functional partitioning. In *Proceedings of the 8th international symposium on System synthesis* (New York, NY, USA, 1995), ISSS '95, ACM, pp. 28–35.

[208] VAHID, F., GAJSKI, D. D., AND GONG, J. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *Proceedings of the Conference on European Design Automation* (Los Alamitos, CA, USA, 1994), EURO-DAC '94, IEEE Computer Society Press, pp. 214–219.

[209] VAHID, F., AND LE, T. D. Extending the kernighan/lin heuristic for hardware and software functional partitioning. *Design Automation for Embedded Systems 2* (1997), 237–261. 10.1023/A:1008836303344.

[210] VAHID, F., STITT, G., AND LYSECKY, R. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer 41*, 7 (july 2008), 40–46.

[211] VASSILIADIS, S., GAYDADJIEV, G., BERTELS, K., AND MOSCU PANAINTE, E. The Molen Programming Paradigm. In *Computer Systems: Architectures, Modeling, and Simulation*, vol. 3133 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 1–10.

[212] VASSILIADIS, S., WONG, S., GAYDADJIEV, G., BERTELS, K., KUZMANOV, G., AND PANAINTE, E. M. The Molen Polymorphic Processor. *IEEE Transactions on Computers 53*, 11 (November 2004), 1363–1375.

[213] VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., AND PRVULOVIC, M. Memtracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization (TACO) 6*, 2 (July 2009), 5:1–5:33.

[214] VON PRAUN, C., BORDAWEKAR, R., AND CASCAVAL, C. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 185–196.

[215] Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune.

[216] WAGNER, T. A., MAVERICK, V., GRAHAM, S. L., AND HARRISON, M. A. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 1994), PLDI '94, ACM, pp. 85–96.

[217] WANG, G., GONG, W., AND KASTNER, R. Application partitioning on programmable platforms using the ant colony optimization. *Journal of Embedded Computing - Embeded Processors and Systems: Architectural Issues and Solutions for Emerging Applications 2*, 1 (January 2006), 119–136.

[218] WATTERSON, S., AND DEBRAY, S. Goal-directed value profiling. In *Compiler Construction*, R. Wilhelm, Ed., vol. 2027 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 319–333.

[219] *Wiangtong, T., Cheung, P. Y. K., and Luk, W.* Comparing three heuristic search methods for functional partitioning in hardware-software codesign. *Design Automation for Embedded Systems 6* (2002), 425–449. 10.1023/A:1016567828852.

[220] *Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P.* The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS) 7*, 3 (April 2008), 36:1–36:53.

[221] *Williams, J., Massie, C., George, A. D., Richardson, J., Gosrani, K., and Lam, H.* Characterization of fixed and reconfigurable multi-core devices for application acceleration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 3*, 4 (November 2010), 19:1–19:29.

[222] *Wiseman, Y., Jiang, S., Wiseman, Y., and Jiang, S. Advanced Operating Systems and Kernel Applications: Techniques and Technologies.* Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2009.

[223] *Wolf, W.* A decade of hardware/software codesign. *Computer 36*, 4 (April 2003), 38–43.

[224] *Wolf, W. H.* An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 5*, 2 (June 1997), 218–229.

[225] *Wu, J., Sun, Q., and Srikanthan, T.* Algorithmic aspects for multiple-choice hardware/software partitioning. *Computers & Operations Research 39*, 12 (December 2012), 3281–3292.

[226] *Wu, J.-G., Srikanthan, T., and Zou, G.-W.* New model and algorithm for hardware/software partitioning. *Journal of Computer Science and Technology 23*, 4 (July 2008), 644–651.

[227] *Wu, Y., and Larus, J. R.* Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (New York, NY, USA, 1994), MICRO 27, ACM, pp. 1–11.

[228] *Wulf, W. A., and McKee, S. A.* Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News 23*, 1 (March 1995), 20–24.

[229] x264 - A free H.264/MPEG-4 AVC video codec. http://www.videolan.org/developers/x264.html.

[230] Xilinx, Inc. http://www.xilinx.com/.

[231] *Xiong, X., Barros, E., and Rosenstiel, W.* A method for partitioning unity language in hardware and software. In *Proceedings of the Conference on European Design Automation* (Los Alamitos, CA, USA, 1994), EURO-DAC '94, IEEE Computer Society Press, pp. 220–225.

[232] *Yang, L., and Guo, M. High-performance computing: paradigm and infrastructure*, vol. 44. John Wiley & Sons, 2006.

[233] *Yankova, Y., Kuzmanov, G., Bertels, K., Gaydadjiev, G., Lu, Y., and Vassiliadis, S.* DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL'07)* (August 2007), pp. 697–701.

[234] *Zhao, Q., Bruening, D., and Amarasinghe, S.* Umbra: efficient and scalable memory shadowing. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 22–31.

List of Publications

Publications related to this thesis

[1] Roel Meeuws, **S. Arash Ostadzadeh**, Carlo Galuzzi, Vlad Mihai Sima, Razvan Nane, and Koen Bertels. **Quipu: A Statistical Model for Predicting Hardware Resources**. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*. Accepted for publication.

[2] Imran Ashraf, **S. Arash Ostadzadeh**, Roel Meeuws, and Koen Bertels. **Communication-Aware HW/SW Co-design for Heterogeneous Multicore Platforms**. In *Proceedings of the 10th International Workshop on Dynamic Analysis*, WODA'12, pages 36–41, Minneapolis, MN, USA, July 2012. ACM. ISBN 978-1-4503-1455-8. doi:10.1145/2338966.2336806.

[3] **S. Arash Ostadzadeh**, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, and Koen Bertels. **Profile-Guided Application Partitioning for Heterogeneous Reconfigurable Platforms**. In *Proceedings of the 16th CSI International Symposium on Computer Architecture and Digital Systems*, CADS'12, pages 37–43, Shiraz, Iran, May 2012. ISBN 978-1-4673-1481-7. doi:10.1109/CADS.2012.6316416.

[4] **S. Arash Ostadzadeh**, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, and Koen Bertels. **The Q^2 Profiling Framework: Driving Application Mapping for Heterogeneous Reconfigurable Platforms**. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 7199 of *Lecture Notes in Computer Science (LNCS)*, pages 76–88. Springer Berlin / Heidelberg, March 2012. ISBN 978-3-642-28364-2. doi:10.1007/978-3-642-28365-9_7.

[5] Ferruccio Bettarelli, Emanuele Ciavattini, Ariano Lattanzi, Giovanni Beltrame, Fabrizio Ferrandi, Luca Fossati, Christian Pilato, Donatella Sciuto, Roel Meeuws, **S. Arash Ostadzadeh**, Zubair Nawaz, Yi Lu, Thomas Marconi, Mojtaba Sabeghi, Vlad Mihai Sima, and Kamana Sigdel. **Extensions of the hArtes Tool Chain**. In Koen Bertels, editor, *Hardware/Software Co-design for Heterogeneous Multi-core Platforms: The hArtes Toolchain*, chapter 6, pages 193–227. Springer Verlag, November 2011. ISBN 978-9-4007-1405-2.

[6] **S. Arash Ostadzadeh**, Roel Meeuws, and Koen Bertels. **Advanced Profiling in the Delft Workbench: The Q^2 Framework**. In *The 2011 Annual ICT.OPEN Con-*

ference, Embedded Systems, Poster Session, Veldhoven, The Netherlands, November 2011.

[7] Koen Bertels, **S. Arash Ostadzadeh**, and Roel Meeuws. **Advanced Profiling of Applications for Heterogeneous Multi-Core Platforms**. In *Proceedings of the 2011 International Conference on Engineering of Reconfigurable Systems & Algorithms*, ERSA'11, pages 171–183, Las Vegas, Nevada, USA, July 2011. CSREA Press. ISBN 1-60132-177-5.

[8] **S. Arash Ostadzadeh**, Marco Corina, Carlo Galuzzi, and Koen Bertels. **Runtime Extraction of Memory Access Information from the Application Source Code**. In *Proceedings of the 2011 International Conference on High Performance Computing and Simulation*, HPCS'11, pages 647–655, Istanbul, Turkey, July 2011. ISBN 978-1-61284-380-3. doi:10.1109/HPCSim.2011.5999888.

[9] **S. Arash Ostadzadeh** and Koen Bertels. ***Quad*: A Sophisticated Memory Access Profiling Toolset**. In *Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing Workshop, Poster Session at Design, Automation and Test in Europe (DATE)*, Grenoble, France, March 2011.

[10] **S. Arash Ostadzadeh** and Koen Bertels. **Dynamic Profiling Framework in the Delft Workbench**. In *The 21st Annual Workshop on Circuits, Systems and Signal Processing (ProRISC), Poster Session*, Veldhoven, The Netherlands, November 2010.

[11] **S. Arash Ostadzadeh**, Marco Corina, Carlo Galuzzi, and Koen Bertels. **t*Quad* - Memory Bandwidth Usage Analysis**. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, ICPPW'10, pages 217–226, San Diego, USA, September 2010. IEEE Computer Society. ISBN 978-0-7695-4157-0. doi:10.1109/ICPPW.2010.39.

[12] **S. Arash Ostadzadeh**, Roel Meeuws, Carlo Galuzzi, and Koen Bertels. ***Quad* - Quantitative Usage Analysis of Data**. In *Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications Workshop, Poster Session: Applications & Architectures at Design, Automation and Test in Europe (DATE)*, Dresden, Germany, March 2010.

[13] **S. Arash Ostadzadeh**, Roel Meeuws, Carlo Galuzzi, and Koen Bertels. ***Quad* - A Memory Access Pattern Analyser**. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science (LNCS)*, pages 269–281. Springer Berlin / Heidelberg, March 2010. ISBN 978-3-642-12132-6. doi:10.1007/978-3-642-12133-3_25.

[14] **S. Arash Ostadzadeh**, Roel Meeuws, kamana Sigdel, and Koen Bertels. **A Multipurpose Clustering Algorithm for Task Partitioning in Multicore Reconfigurable Systems**. In *Proceedings of the 2009 International Workshop on Multi-Core Computing Systems at the International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS'09, pages 663–668, Fukuoka, Japan, March 2009. doi:10.1109/CISIS.2009.127.

[15] **S. Arash Ostadzadeh**, Roel Meeuws, Kamana Sigdel, and Koen Bertels. **A Clustering Framework for Task Partitioning Based on Function-Level Data Usage Analysis**. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Poster Session: Processors & CAD Tools*, FPGA'09, pages 279–279, Monterey, California, USA, February 2009. ACM. ISBN 978-1-60558-410-2. doi:10.1145/1508128.1508183.

[16] **S. Arash Ostadzadeh** and Koen Bertels. **Parallelism Utilization in Embedded Reconfigurable Computing Systems: A Survey of Recent Trends**. In *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2007.

Other publications

[1] M. Faisal Nadeem, Imran Ashraf, **S. Arash Ostadzadeh**, Stephan Wong, and Koen Bertels. **Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements**. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW'12, pages 79–90, Shanghai, China, May 2012. IEEE Computer Society. ISBN 978-1-4673-0974-5. doi:10.1109/IPDPSW.2012.6.

[2] M. Faisal Nadeem, **S. Arash Ostadzadeh**, Muhammad Nadeem, Stephan Wong, and Koen Bertels. **A Simulation Framework for Reconfigurable Processors in Large-Scale Distributed Systems**. In *Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) at the 40th International Conference on Parallel Processing*, ICPPW'11, pages 352–360, Taipei City, Taiwan, September 2011. ISBN 978-1-4577-1337-8. doi:10.1109/ICPPW.2011.50.

[3] M. Faisal Nadeem, **S. Arash Ostadzadeh**, Stephan Wong, and Koen Bertels. **Task Scheduling Strategies for Dynamic Reconfigurable Processors in Distributed Systems**. In *Proceedings of the 2011 International Conference on High Performance Computing and Simulation*, HPCS'11, pages 90–97, Istanbul, Turkey, July 2011. ISBN 978-1-61284-380-3. doi:10.1109/HPCSim.2011.5999811.

[4] M. Faisal Nadeem, **S. Arash Ostadzadeh**, Mahmood Ahmadi, Muhammad Nadeem, and Stephan Wong. **A Novel Dynamic Task Scheduling Algorithm for Grid Networks with Reconfigurable Processors**. In *Proceedings of the 5th HiPEAC Workshop on Reconfigurable Computing*, WRC'11, Heraklion, Greece, January 2011.

[5] M. Faisal Nadeem, Fakhar Anjam, **S. Arash Ostadzadeh**, Mahmood Ahmadi, and Stephan Wong. **Towards the Utilization of Reconfigurable Processors in Grid Networks**. In *Proceedings of the 21st Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, pages 29–35, Veldhoven, The Netherlands, November 2010.

[6] S. Shervin Ostadzadeh, Jafar Habibi, and **S. Arash Ostadzadeh**. **A Framework for Decision Support Systems Based on Zachman Framework**. In *Advanced Techniques in Computing Sciences and Software Engineering*, pages 497–502. Springer Netherlands, 2010. ISBN 978-90-481-3660-5. doi:10.1007/978-90-481-3660-5_86.

[7] Mahmood Ahmadi, **S. Arash Ostadzadeh**, and Stephan Wong. **Rule-set Database Inspection: Towards Data Utilization in Packet Processing**. In *Proceedings of the International Conference on the Latest Advances in Networks*, ICLAN'08, Toulouse, France, December 2008.

[8] Behnaz Pourebrahimi, **S. Arash Ostadzadeh**, and Koen Bertels. **Resource Allocation in Market-based Grids Using a History-based Pricing Mechanism**. In *Advances in Computer and Information Sciences and Engineering*, pages 97–100. Springer Netherlands, 2008. ISBN 978-1-4020-8741-7. doi:10.1007/978-1-4020-8741-7_18.

[9] S. Shervin Ostadzadeh, Fereidoon Shams Aliee, and **S. Arash Ostadzadeh**. **An MDA-Based Generic Framework to Address Various Aspects of Enterprise Architecture**. In *Advances in Computer and Information Sciences and Engineering*, pages 455–460. Springer Netherlands, 2008. ISBN 978-1-4020-8741-7. doi:10.1007/978-1-4020-8741-7_81.

[10] Mahmood Ahmadi, **S. Arash Ostadzadeh**, and Stephan Wong. **An Analysis of Rule-set Databases in Packet Classification**. In *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2007.

[11] **S. Arash Ostadzadeh**, B. Maryam Elahi, Zeinab Zeinalpour Tabrizi, M. Amir Moulavi, and Koen Bertels. **A Two-phase Practical Parallel Algorithm for Construction of Huffman Codes**. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA'07, pages 284–291, Las Vegas, USA, June 2007.

[12] S. Shervin Ostadzadeh, Fereidoon Shams Aliee, and **S. Arash Ostadzadeh**. **Employing MDA in Enterprise Architecture**. In *Proceedings of the 12th Annual CSI Computer Conference*, CSICC'07, pages 1646–1653, Tehran, Iran, February 2007.

[13] S. Shervin Ostadzadeh, Fereidoon Shams Aliee, and **S. Arash Ostadzadeh**. **A Method for Consistent Modeling of Zachman Framework Cells**. In *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, pages 375–380. Springer Netherlands, 2007. ISBN 978-1-4020-6264-3. doi:10.1007/978-1-4020-6264-3_65.

[14] S. Shervin Ostadzadeh, Fereidoon Shams Aliee, and **S. Arash Ostadzadeh**. **Applying MDA to Zachman Framework for Information Technology Systems**. In *Proceedings of the 2nd Conference of Information and Communication Technology Management*, ICTM'06, Tehran, Iran, February 2006.

[15] **S. Arash Ostadzadeh**, B. Maryam Elahi, M. Amir Moulavi, and Zeinab Zeinalpour. **A Level-based Practical Parallel Algorithm for Huffman Encoding**. In *Proceedings of the 11th Annual CSI Computer Conference*, CSICC'06, pages 559–562, Tehran, Iran, January 2006.

[16] **S. Arash Ostadzadeh**, Zeinab Zeinalpour, and M. Amir Moulavi. **A Parallel Algorithm for Reconstruction of B-tree Structure after Substantial Key Removals**. In *Proceedings of the 11th Annual CSI Computer Conference*, CSICC'06, pages 592–595, Tehran, Iran, January 2006.

[17] **S. Arash Ostadzadeh**, M. Amir Moulavi, Zeinab Zeinalpour, and B. Maryam Elahi. **Parallel Construction of Huffman Codes**. In *Advances in Systems, Computing Sciences and Software Engineering*, pages 17–23. Springer Netherlands, 2006. ISBN 978-1-4020-5262-0. doi:10.1007/1-4020-5263-4_4.

[18] **S. Arash Ostadzadeh**, M. Amir Moulavi, and Zeinab Zeinalpour. **Massive Concurrent Deletion of Keys in B*-Tree**. In *Parallel Processing and Applied Mathematics*, volume 3911 of *Lecture Notes in Computer Science (LNCS)*, pages 83–91. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-34141-3. doi:10.1007/11752578_11.

Publications in Persian

[1] **S. Arash Ostadzadeh** and S. Shervin Ostadzadeh. **Parallel IPR Tree: Dictionary Operations in a Shared Memory Parallel System**. In *Proceedings of the 13th Annual CSI Computer Conference*, CSICC'08, Kish, Iran, March 2008.

[2] S. Shervin Ostadzadeh, Fereidoon Shams Aliee, and **S. Arash Ostadzadeh**. **The Adaption of MDA in Extended Enterprise Architecture Framework**. In *Proceedings of the 13th Annual CSI Computer Conference*, CSICC'08, Kish, Iran, March 2008.

[3] **S. Arash Ostadzadeh** and S. Shervin Ostadzadeh. **A Multi-stage Offline Handwritten Signature Verification System for Persian Signature Authentication**. *Technical Engineering Journal of IAUM*, 1(1):1–15, September 2006.

[4] S. Shervin Ostadzadeh, Fereidoon Shams Aliee, and **S. Arash Ostadzadeh**. **The Role of the Software Development Process in Enterprise Architecture**. In *Proceedings of the 2nd Conference of Information and Communication Technology Management*, ICTM'06, Tehran, Iran, February 2006.

[5] **S. Arash Ostadzadeh** and S. Shervin Ostadzadeh. **An Optimal Parallel Algorithm for Searching Keys in IPR Trees**. In *Proceedings of the 11th Annual CSI Computer Conference*, volume 1 of *CSICC'06*, pages 951–955, Tehran, Iran, January 2006.

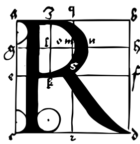
[6] **S. Arash Ostadzadeh**, S. Shervin Ostadzadeh, and Mohammad Hassan Shenasa. **A Multi-stage Offline Signature Verification System Based on Elastic Matching With Multi Resolution Segmentation**. In *Proceedings of the 7th National Conference on Intelligent Systems*, CIS'05, Tehran, Iran, December 2005.

[7] **S. Arash Ostadzadeh**. **The Design and Implementation of an Offline Handwritten Signature Verification System Using Local Correspondence**. Msc thesis, Ferdowsi University of Mashhad, Mashhad, Iran, September 2001.

[8] **S. Arash Ostadzadeh** and Hossein Deldari. **Offline Tracing and Representation of Handwritten Signatures**. Technical report, Ferdowsi University of Mashhad, Mashhad, Iran, August 2001.

[9] **S. Arash Ostadzadeh**. **Handwritten Signature Verification: A Literature Survey**. Technical report, Ferdowsi University of Mashhad, Mashhad, Iran, April 2000.

Samenvatting



Recente ontwikkelingen tonen een gestadige groei van het gebruik van heterogene multicore architecturen om de steeds groeiende vraag naar computerprestaties. Deze ontluikende architecturen stellen specifieke uitdagingen ten aanzien van hun programmeerbaarheid. Bovendien vereisen ze dat applicaties efficiënt worden toegewezen om de prestaties ten volle te benutten en knelpunten te voorkomen. In dit opzicht is het van cruciaal belang om applicatiegedrag en, in het bijzonder, de datacommunicatie tussen taken te analyseren.

In deze dissertatie presenteren we een profileringsraamwerk dat ontwikkelaars helpt om inzicht te verwerven in het gedrag van een applicatie. Het gepresenteerde profileringsraamwerk is generiek en niet beperkt tot een bepaald platform, applicatie of doelstelling. We gebruiken dit raamwerk voornamelijk om applicaties toe te wijzen aan een heterogene multicore architectuur. Het raamwerk bevat een set hulpgereedschappen voor het profileren van geheugentoegang, *Quad* genaamd, welke kwantitatieve informatie verschaft met betrekking tot de geheugentoegangen binnen een applicatie. *Quad* gebruikt Dynamic Binary Instrumentation (DBI) om de werkelijke data dependencies die zich tijdens het uitvoeren voordoen tussen taken van een applicatie. Daarenboven verschaft het ook nauwkeurige metingen van de geheugenstoegang, zoals de hoeveelheid aan data dat tussen taken is uitgewisseld en de geheugengrootte die voor deze communicatie benodigd was. Dergelijke informatie kan gebruikt worden om kritieke delen van een applicatie te identificeren, mogelijkheden voor grof-gekorreld parallelisme te markeren en code-optimalisaties te sturen.

Om de bruikbaarheid van de afgeleide profielinformatie te onderbouwen hebben we, als proef concept, het voornaamste resultaat van *Quad*, de Quantitative Data Usage (QDU) graaf, aangewend als basis voor een generieke formulering van het applicatiepartitioneringsprobleem. De formulering van dit onhandelbare probleem is flexibel en voegt zich makkelijk naar verschillende ontwerpdoelen en -eisen. Voorts stellen we een heuristieke algoritme voor om partitioneringen van hoge kwaliteit te vinden binnen een aanvaardbare termijn. Behalve de complexiteitsanalyse van het voorgestelde algoritme presenteren we ook een uitvoerige theoretische analyse van het applicatie-partitionering probleem. Om de kwaliteit van de oplossingen te evalueren ontwikkelden we een test bench om synthetische QDU grafen te genereren en vergeleken de resultaten met de optimale resultaten behaald met een uitputtend zoekalgoritme. Deze vergelijking toont aan dat het voorgestelde algoritme in staat is optimale of bijna-optimale oplossingen te

bepalen.

Om de toepasbaarheid van het profileringsraamwerk verder aan te tonen onderzoeken we, in detail, hoe het raamwerk in de praktijk werkt door twee realistische applicaties toe te wijzen op een heterogene herconfigureerbare architectuur. Om dit doel te bereiken stellen we een hardware/software partitioneringsmethodologie voor, die nauwgekoppelde taken op basis van datacommunicatieanalyse samenvoegt. Daarenboven wordt de profielinformatie gebruikt om de applicaties af te stemmen en hun data flow te optimaliseren. De verkregen resultaten tonen een prestatiewinst van 192% en 30%.

Stellingen behorende bij het proefschrift

Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures

S. Arash Ostadzadeh

1. Als je iedere programmeer probleem aanpakt door hergebruik van legacy code en klaagt over de prestatie, dan kun je niet echt een programmeur zijn.
2. In wetenschappelijk onderzoek is het citeren van een artikel zonder concreet bewijs van haar beweringen net als het verwijzen naar "Nautilus" in Jules Verne's "Twenty Thousand Leagues Under the Sea" als de eerste oceaan onderzeeër!
3. Je moet je geen zorgen maken over datgene wat anderen over je zeggen, zolang het incongruent is waar je geweten je stoort.
4. Motivatie staat centraal in creativiteit, productiviteit en geluk. De zoektocht naar idealisme is de meest noodlottige demotivatie.
5. Het is onrechtvaardig om te klagen dat geschoolde programmeurs die parallel programeren zijn schaars, terwijl traditionele concepten bij de ontwikkeling van programma's nog steeds onderwezen worden als fundamenten voor bachelor studenten in de informatica en engineering.
6. Het maken van hypothesen op basis van wat anderen alleen hebben beweerd is absoluut waardeloos.
7. De verafgoding van gerenommeerde for-profit bedrijfsleiders alsof zij wetenschappers leidden is symptomatisch voor een ernstig probleem met wat mensen waarderen in de wereld.
8. De meest logische manier om een einde te maken aan een zinloze onophoudelijke argument is te verklaren: ik kan het niet bewijzen, maar je kunt het ook niet als onjuist bewijzen!
9. Er is een fundamenteel lijn tussen "novel research" en "research novel"! Alleen "novel research" moet worden beschouwd als wetenschappelijke waarde, terwijl "research novelists" in het meest optimistische geval als ambitieuze dromers moeten worden beschouwd.
10. Een korte leven met breedte heeft de voorkeur boven een smalle met lengte. — *Pur Sina*, ook bekend met de Latijnse naam "Avicenna".
11. Het programmeren is een kunst, want het past opgebouwde kennis aan de wereld toe, want het vergt vaardigheid en vindingrijkheid, en voornamelijk omdat het objecten van schoonheid produceert. Een programmeur die zichzelf onbewust beschouwt als een kunstenaar zal genieten van wat hij doet en zal het beter doen. — *Donald E. Knuth.*

Deze stellingen worden opponeerbaar en verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor, prof. dr. K.L.M Bertels.

Propositions accompanying the PhD dissertation

Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures

S. Arash Ostadzadeh

1. If you tackle every programming problem by reusing legacy code and complain about its performance, then you cannot be much of a programmer.
2. In scientific research, citing an article without a concrete evidence of its claim is like referring to "Nautilus" in Jules Verne's "Twenty Thousand Leagues Under the Sea" as the first ocean-spanning submarine!
3. You should not care about what others say about you as long as it is incongruent with what your conscience indeed bothers you for.
4. Motivation is central to creativity, productivity and happiness. The quest for idealism is the most fatal demotivation.
5. It is inequitable to complain that skilled parallel programmers are scarce while traditional concepts in developing programs are still taught as fundamentals to undergrads in computer science and engineering.
6. Making any assumption based upon what others have only claimed is absolutely worthless.
7. The idolization of renowned for-profit company managers as though they were leading scientists is symptomatic of a severe problem with what people value in the world.
8. The most logical way to put an end to a pointless incessant argument is to declare: I cannot prove it, but you cannot prove it false either!
9. There is a fundamental line between "novel research" and "research novel"! Only novel research should be considered as having scientific value, while research novelists, in the most optimistic case, should be considered ambitious dreamers.
10. A short life with width is preferable to a narrow one with length. — *Pur Sina*, also known with the Latinized name "Avicenna".
11. Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. — *Donald E. Knuth.*

These propositions are regarded as opposable and defendable, and have been approved as such by the promoter, prof. dr. K.L.M Bertels.

Curriculum Vitae

S. Arash Ostadzadeh was born in Mashhad, Iran, 1976. He followed his secondary eduction at the Malek Ashtar high school in Mashhad, Iran, where he got his diploma in 1994. He received a Bachelor of Science degree in Computer Engineering from Ferdowsi University of Mashhad, Mashhad, Iran, in 1998. Afterwards, he continued his studies towards a Master of Science degree in Computer Engineering, majoring Software Engineering, at the same university. He was awarded the M.Sc. degree in 2001, as the top postgraduate in Computer Engineering program. Starting from 1999, for a continuous eight years, he was lecturing several Computer Science and Engineering courses for undergraduates in major universities and higher education institutes in Mashhad, Iran. During 2001-2006, he served as a faculty member of the Computer Engineering Department at IAUM.

In 2007, he joined the Computer Engineering Group, Department of Software and Computer Technology, Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS/EWI), Delft University of Technology, the Netherlands, to pursue his Doctoral Research under the supervision of Prof. dr. Koen Bertels. His research focused on application profiling and partitioning, utilizing dynamic binary instrumentation, which resulted in the development of a memory access profiling toolset, called *Quad*. At the same time, he developed a simulation framework, named DReAMSim, which was designed for modeling homogeneous and/or heterogeneous (re)-configurable processing elements in a large-scale distributed environment. His research work was mainly funded by the European Commission FWP6 hArtes (holistic Approach to reconfigurable real-time embedded systems) and the SMECY (Smart Multicore Embedded Systems) projects. He has also contributed to the EU-ICT FWP7 REFLECT and the iFEST projects.

His research interests include parallel and distributed computing, heterogeneous multicore systems, data structures and algorithms, application profiling, program analysis and performance evaluation, and hardware/software co-design.