Automating FPGA-based CSoC platform generation

Siriwardena Wijesundera, Deshya Senelie

2019

Siriwardena Wijesundera, D. S. (2019). Automating FPGA-based CSoC platform generation. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/90141>

<https://doi.org/10.32657/10220/48432>

AUTOMATING FPGA-BASED CSOC PLATFORM GENERATION

**DESHYA SENELIE SIRIWARDENA
WIJESUNDERA**

School of Computer Science and Engineering

*A thesis submitted to the Nanyang Technological University in
fulfilment of the requirement for the degree of Doctor of Philosophy*

January 2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

17/01/2019
.....
Date

.....
Deshya Senelie Siriwardena Wijesundera

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

.....17/1/2019
Date

.....
Dr Thambipillai Srikanthan (Prof.)

Authorship Attribution Statement

This thesis contains material from 6 papers published in the following peer-reviewed journals/conferences where I was the first and corresponding author.

Chapter 3 is published as Deshya Wijesundera, Alok Prakash, Thambipillai Srikanthan, and Achintha Ihalage. 2018. Framework for Rapid Performance Estimation of Embedded Soft Core Processors. *ACM Trans. Reconfigurable Technol. Syst.* 11, 2, Article 9 (July 2018), 21 pages. DOI: <https://doi.org/10.1145/3195801>.

The contributions of the co-authors are as follows:

- Prof Srikanthan provided the initial project direction
- Dr Prakash participated in idea generation and brain storming sessions
- I prepared the manuscript drafts. The manuscript was revised by Dr Prakash
- I designed the study, performed all experiments and analyzed the data
- Achintha Ihalage assisted in the experiments

Chapter 3 is published as D. Wijesundera, A. Prakash and T. Srikanthan, "Rapid design space exploration for soft core processor customization and selection," *2016 International Conference on Field-Programmable Technology (FPT)*, Xi'an, 2016, pp. 185-188. doi: 10.1109/FPT.2016.7929529

The contributions of the co-authors are as follows:

- Prof Srikanthan provided the initial project direction
- Dr Prakash participated in idea generation and brain storming sessions
- I prepared the manuscript drafts. The manuscript was revised by Dr Prakash
- I designed the study, performed all experiments and analyzed the data.

Chapter 4 is published as Deshya Wijesundera, Achintha Ihalage, Alok Prakash, and Thambipillai Srikanthan. 2017. High Speed Performance Estimation of Embedded Hard-core Processors in FPGA-based SoCs. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable*

Technologies (HEART2017). ACM, New York, NY, USA, Article 2, 6 pages. DOI: <https://doi.org/10.1145/3120895.3120902>

The contributions of the co-authors are as follows:

- Prof Srikanthan provided the initial project direction
- Dr Prakash participated in idea generation and brain storming sessions
- I prepared the manuscript drafts. The manuscript was revised by Dr Prakash
- I designed the study, performed all experiments and analyzed the data
- Achintha Ihalage assisted in the experiments and data analysis

Chapter 6 is published as Deshya Wijesundera, Alok Prakash, Thilina Perera, Kalindu Herath, and Thambipillai Srikanthan. 2018. Wibheda+: Framework for Data Dependency-aware Multi-constrained Hardware-Software Partitioning in FPGA-based SoCs for IoT Applications. In Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2018). ACM, New York, NY, USA, Article 3, 6 pages. DOI: <https://doi.org/10.1145/3241793.3241796>

The contributions of the co-authors are as follows:

- Prof Srikanthan provided the initial project direction
- I prepared the manuscript drafts. The manuscript was revised by Dr Prakash
- I designed the study, performed all experiments and analyzed the data
- Thilina Perera assisted in the experiments and data analysis
- Kalindu Herath assisted in the experiments

Chapter 7 is published as Deshya Wijesundera, Alok Prakash, Siew Kei Lam, and Thambipillai Srikanthan. 2016. Exploiting Configuration Dependencies for Rapid Area-efficient Customization of Soft-core Processors. In Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES '16), Sander Stuijk (Ed.). ACM, New York, NY, USA, 163-172. DOI=<http://dx.doi.org/10.1145/2906363.2906385>

The contributions of the co-authors are as follows:

- Prof Srikanthan provided the initial project direction
- Dr Prakash participated in idea generation and brain storming sessions
- Dr Lam participated in idea generation and brain storming sessions
- I prepared the manuscript drafts. The manuscript was revised by Dr Prakash
- I designed the study, performed all experiments and analyzed the data

Chapter 8 is published as D. Wijesundera, A. Prakash, T. Perera, K. Herath and T. Srikanthan, "Wibheda: Framework for Data Dependency-Aware Multi-Constrained Hardware-Software Partitioning in FPGA-Based SoCs for IoT Devices," *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Boulder, CO, 2018, pp. 213-213. doi: 10.1109/FCCM.2018.00047

The contributions of the co-authors are as follows:

- Prof Srikanthan provided the initial project direction
- I prepared the manuscript drafts. The manuscript was revised by Dr Prakash
- I designed the study, performed all experiments and analyzed the data
- Thilina Perera assisted in the experiments and data analysis
- Kalindu Herath assisted in the experiments

17/01/2019
.....
Date

.....
Deshya Senelie Siriwardena Wijesundera

Acknowledgements

I would like to take this opportunity to thank many people who have directly or indirectly helped me find my foothold in research. First and foremost, I would like to express my heartfelt gratitude to my supervisor and mentor, Prof. Thambipillai Srikanthan, for his unceasing support, time, valuable guidance and constant encouragement. I am extremely grateful to him for the constructive suggestions and ideas, that have immensely contributed to the unfolding of the research work.

I am very much grateful to Dr Alok Prakash for helping me throughout this work. His support, advice and insights truly helped me at all stages of this thesis. I would like to thank Asst. Prof. Siew Kei Lam who has always provided valuable advice.

My gratitude extends to my colleague and husband Thilina who has played a vital role at every stage of this thesis. The discussions with him has greatly contributed to this research. More importantly, it was his support and encouragement which made this thesis a reality. My good friend and colleague Kalindu helped me immensely throughout this journey. The technical discussions I have had with him as well as his encouragement during difficult times was invaluable.

The technical discussions with Dr Yan Lin Aung have been extremely helpful. My sincere appreciation goes to Kisaru and Achintha who worked with me as interns in HESL. My gratitude extends to Jeremiah for his invaluable administrative support.

I would like to thank my loving mother if not for whose single-handed effort I will not be what I am today. Thank you for all the support and encouragement which saw me through the toughest of times.

Table of contents

List of figures	xv
List of tables	xvii
Abstract	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Research Objective	2
1.3 Organization	3
1.4 Publications	5
2 Literature Review	7
2.1 Introduction	7
2.2 Embedded Computing Architectures	8
2.2.1 Heterogeneous Architectures	8
2.2.2 Reconfigurable Architectures	9
2.2.3 Configurable System on Chip	11

2.3	Design Methodologies for CSoC Customization	13
2.3.1	Software Performance Estimation	14
2.3.2	Application-specific Soft Core Processor Customization . . .	20
2.3.3	Hardware Area-Time Estimation	23
2.3.4	Hardware-Software Partitioning	26
2.4	EDA Tools for FPGA-based CSoC	30
2.4.1	Academic Tools	31
2.4.2	Commercial Tools	32
2.5	Summary	33
3	Performance Estimation of Soft Core Processors	37
3.1	Introduction	37
3.2	Methodology	38
3.2.1	Processor Characterization	40
3.2.2	Application Profiling	43
3.2.3	Performance Estimation	45
3.3	Performance Evaluation	51
3.4	Summary	59
4	Performance Estimation of Hard Core Processors	61
4.1	Introduction	61
4.2	Methodology	62
4.2.1	Processor Characterization	64

4.2.2	Application Profiling	67
4.2.3	Instruction Analysis	67
4.2.4	Performance Estimation	70
4.2.5	Support for Other Hard Core Processors	75
4.3	Performance Evaluation	75
4.4	A Unified Framework for Performance Estimation of Hard and Soft Core Processors	78
4.5	Summary	80
5	Hardware Area-Time Estimation	83
5.1	Introduction	83
5.2	Methodology	84
5.2.1	Basic Block Wrapper	86
5.2.2	RTL Conversion	89
5.2.3	Synthesis on Vendor Tool	92
5.2.4	Area-Time Extraction	93
5.2.5	Incorporation of Proprietary Soft IP Cores	94
5.2.6	Ensuring Functional Correctness	97
5.3	Performance Evaluation	98
5.4	Summary	102
6	Data Dependency-aware Hardware-Software Partitioning	103
6.1	Introduction	103

6.2	Methodology	105
6.2.1	System Level Model	105
6.2.2	Hardware-Software Partitioning	106
6.3	Performance Evaluation	115
6.4	Summary	123
7	Instruction Subsetting for Soft Core Processor Customization	125
7.1	Introduction	125
7.2	Methodology	126
7.2.1	Processor Characterization	127
7.2.2	Architecture-aware Dependency Analysis	129
7.2.3	Application Profiling & Analysis	130
7.2.4	Selection Heuristic	131
7.3	Performance Evaluation	133
7.4	Summary	147
8	Wibheda+: Integrated Framework for Automating CSoC Platform Generation	149
8.1	Introduction	149
8.2	Methodology	153
8.2.1	Input Parameters	154
8.2.2	C to IR Conversion (Section A)	155
8.2.3	Profiling (Section A)	155

8.2.4	Processor Performance Estimation - Soft Core (Section A) . . .	155
8.2.5	Processor Performance Estimation- Hard Core (Section B) . . .	156
8.2.6	Data Dependency Analysis	156
8.2.7	Hot Block Identification	156
8.2.8	Hardware Area-Time Estimation (Section C)	157
8.2.9	Hardware-Software Partitioning (Section D)	157
8.2.10	Iterative Customization	157
8.2.11	Soft Core Processor Subsetting and Area Optimization (Section E)	159
8.3	Performance Evaluation	161
8.4	Summary	169
9	Conclusions and Future Directions	171
9.1	Conclusions	171
9.2	Future Research Directions	175
	References	179

List of figures

3.1	Framework for Soft Core Processor Performance Estimation	39
3.2	Sample LLVM Profile	44
3.3	Sample LLVM Tracer Tool Output	46
3.4	Distribution of Estimation Results Across the Design Space	57
4.1	Framework for Hard Core Processor Performance Estimation	63
4.2	Sample ISA Mapping File	66
4.3	Comparison of Results	77
4.4	Unified Framework for Performance Estimation of Hard and Soft Core Processors	79
5.1	Hardware Area-Time Estimation - A	85
5.2	Block Diagram of Generic Divider	90
5.3	Block Diagram of Generic BRAM	90
5.4	Hardware Area-Time Estimation - B	95
5.5	Inputs and Output of Hardware-Software Partitioning	102
6.1	System Level Model	105

6.2	Criteria 1: Runtime Comparison	116
6.3	Criteria 2: Comparison of Estimated Performance	121
6.4	Criteria 3: Accuracy of System Evaluation	122
7.1	Framework for Design Space Pruning and Soft Core Processor Customization	128
7.2	Ordered tree for 1-greedy search heuristic	132
7.3	Comparison of Actual Area and Estimated Area	137
7.4	Distribution of CFU Impact	139
7.5	Results of Exhaustive Estimation Technique	143
7.6	Comparison of Accuracy of Estimated Performance	146
8.1	Wibheda+ Framework	150
8.2	GUI of Wibheda+	153
8.3	GUI of Wibheda+ - Input Parameters	154
8.4	Technique for Soft Core Processor Subsetting and Area Optimization	160
8.5	Scenario 1	163
8.6	Scenario 2	164
8.7	Scenario 3	165
8.8	Scenario 4	165

List of tables

3.1	Classification of LLVM Instructions	42
3.2	Configurations in Experimental Setup	52
3.3	Configurable Functional Units	53
3.4	Equivalent LE Calculation	53
3.5	Estimation Errors (%)	58
3.6	Comparison of Runtime	59
4.1	Classification of LLVM Instructions	65
4.2	Estimation Errors and Runtime	78
5.1	Generic RTL for Floating Point Cores	91
5.2	Examples of Incorporated Vendor Proprietary IP cores	96
5.3	Area-Time Estimation Values for ADPCM	100
5.4	Area-Time Estimation Values for DFSIN	101
6.1	Criteria 2: Estimation Errors	117
6.2	Criteria 3: Constraints for System Evaluation	121

7.1	Equivalent ALMs of CFUs	134
7.2	Timing Characteristics of Configurations	135
7.3	Area Characteristics of Configurations	137
7.4	Equivalent ALMs of CFUs	137
7.5	Instruction Counts of Test Applications	138
7.6	Comparison of Results	144
8.1	Sample LLVM profile of DFMUL	158
8.2	Recommendations of Wibheda+ for DFMUL	159
8.3	FPGA Device Families and Processors	162
8.4	Average Runtime (min:sec) of Wibheda+	162
8.5	Experimental Scenarios	163
8.6	Percentage Difference in Performance (smallest devices)	166
8.7	Percentage Difference in Performance (largest devices)	167
8.8	Implementation of Wibheda+ Recommendations	168

List of Abbreviations

ADL Architecture Description Language

ALM Adaptive Logic Module

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction set Processor

AST Abstract Syntax Tree

BLS Binary Level Simulation

CAGR Compound Annual Growth Rate

CDFG Control Data Flow Graph

CFG Control Flow Graph

CFU Configurable Functional Unit

CLB Configurable Logic Block

CSoC Configurable System-on-Chip

DMA Direct Memory Access

DoE Design of Experiments

DSP Digital Signal Processing block

EDA Electronic Design Automation

FPGA Field Programmable Gate Array

GPP General Purpose Processor

GPU Graphics Processing Unit

HLS High-Level Synthesis

HVP Hardware Virtual Platform

ILP Integer Linear Programming

IP Intellectual Property

IR Intermediate Representation

IRLS Intermediate Representation Level Simulation

ISA Instruction Set Architecture

ISSB Instruction Set Simulator Based

ITRS International Technology Roadmap for Semiconductors

LUT Look-Up Table

MILP Mixed Integer Linear Programming

MPSoC Multi Processor System-on-Chip

NEB Native Execution Based

NRE Non-Recurring Engineering

OVP Open Virtual Platform

REG Register

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SLS Source Level Simulation

SVP Software Virtual Platform

TTM Time-to-Market

VPB Virtual Platform Based

Abstract

Field Programmable Gate Array (FPGA) based configurable system-on-chip (CSoC) platforms have become a preferred choice for embedded computing systems to meet the increasing demand for shorter time-to-market (TTM) and lower non-recurring engineering (NRE) costs, due to both high density and myriad of on-chip hardware and software compute resources. However, the inability of existing tools to effectively exploit these resources to satisfy design constraints, especially from high level specifications such as C/C++, remains a bottleneck for meeting the TTM pressures. In this research, techniques for the automatic generation of a FPGA-based CSoC platform have been proposed to satisfy the area-time design constraints by taking user preferences into account.

A rapid technique has been proposed to estimate application performance (runtime) on soft core processors. The proposed methodology relies on the target independent intermediate representation (IR) of the LLVM compiler, without necessitating application execution on the target processor or instruction set simulators, thereby making it applicable to other soft core processors and corresponding FPGA architectures. Further, the approach is scalable to the large number of configuration options available in modern soft core processors. The technique takes into account both data hazards and control hazards within the processor pipeline in order to obtain high estimation accuracy. Experimental results using applications from the CHStone benchmark suite on two commercial soft core processors, Xilinx MicroBlaze and Altera Nios show an error of only 5% averaged across the full design space.

Noting that modern FPGA platforms typically also integrate hard core processors, the technique proposed for soft core processors has been extended to support the performance estimation for hard core processors. This necessitated the introduction of models for addressing performance-centric features such as dual-issue, out-of-order and superscalar. The proposed technique takes into account data hazards, control hazards and structural hazards within the processor pipeline in order to obtain high estimation accuracy. The technique has been tested using applications from the CHStone benchmark suite for the ARM Cortex-A9 processor in a Xilinx Zynq SoC FPGA and has been shown to be accurate with an average estimation error of only 5.84%. The estimation accuracy compares well with that of soft core processor performance estimation. Moreover, a unified framework to facilitate the performance estimation on both soft core and hard core processors has been proposed and described.

A novel technique for hardware area-time estimation of applications on FPGA has been proposed. The application C code was first converted to the target independent LLVM IR prior to wrapping the basic blocks as functions using a LLVM transformation pass. The LegUp tool's 'LLVM IR functions to RTL modules' conversion was carried out to facilitate RTL synthesis using the Altera Quartus tools. In order to support FPGAs other than Altera, the soft intellectual property (IP) cores generated by LegUp were replaced as generic RTL components. This approach, together with methods for incorporating vendor-specific basic IP cores, has made it possible to support FPGAs from other vendors with high area-time estimation accuracy. The proposed technique relies on the free versions of the vendor tools and LegUp. Moreover, the proposed approach does not necessitate time consuming post Synthesis steps such as Place

& Route and Bit Stream Generation in order to obtain reasonably accurate area estimation measures.

A technique for data dependency-aware hardware-software partitioning has been proposed. The complex data dependencies between the basic blocks amenable for hardware acceleration as well as all the memory components are identified to facilitate the hardware-software partitioning. The partitioning problem is then solved as an Oregon Trail Knapsack where the value of an item of type x depends on the presence of another item of type y in the knapsack. In order to overcome the exponential complexity of traditional hardware-software partitioning problems, an integer linear programming based heuristic with a time complexity of $\mathcal{O}(n^2)$ has been employed to solve the Oregon Trail Knapsack model. The proposed heuristic is capable of recommending the most suitable hardware-software partitioning under multiple user specified area constraints. The experimental results using applications from the CHStone benchmark suite show that the performance of the recommended accelerators from the proposed technique is within 99% of that obtained from an exhaustive approach. When compared with the existing state-of-the-art, the proposed approach reduced the runtime by orders of magnitude.

A novel application-specific instruction subsetting technique for soft core processor customization has been proposed to minimize the area utilized by the soft core processor. The design space is first pruned by exploiting dependencies between configurations before invoking a systematic subsetting of the microarchitecture by referring to the LLVM IR of the application. Two impact ordered tree based selection heuristics with time complexity of $\mathcal{O}(n)$ have been proposed to perform application-specific instruction subsetting. The proposed technique has been tested using applications from the CHStone benchmark suite and 2 handwritten applications on the Altera Nios soft core processor. The methodology provides an average area reduction of 47.58% compared to a processor with all configuration options enabled. Further, the selection heuristics provide a significant speed up in the order of 10^6X as opposed to an existing approach.

The various methods proposed earlier have been integrated to facilitate the automatic generation of constraint-aware platforms for application specific FPGA-based CSoC. The automated framework, called Wibheda+, strives to obtain the best performance under user specified area constraint and soft core/hard core processor (if any). Pre-defined code segments can be excluded from hardware acceleration to accommodate design flexibility. Wibheda+ has been tested extensively for a comprehensive evaluation with ARM Cortex A9, Altera Nios and Xilinx MicroBlaze processors. Applications from CHStone benchmark suite have been employed to evaluate Wibheda+ on Cyclone V, Cyclone II and Arria II FPGA devices from Altera as well as Kintex 7 and Artix 7 from Xilinx. Experimental results show that Wibheda+ can consistently identify near-optimal solutions in a few minutes. Wibheda+ can also be relied upon to recommend the most suitable CSoC platform from the supported FPGA devices and processors. Finally, the proposed automatic generation of constraint-aware platforms for application specific FPGA-based CSoC has made it possible to facilitate the rapid design space exploration of complex applications without violating the stringent TTM requirements.

Chapter 1

Introduction

1.1 Motivation

The complexity of designing embedded systems continue to escalate with the increasing complexity in applications and stringent design requirements. The time-to-market (TTM) pressure has reached a level where a delay of a single quarter to reach the market, results in a loss of 1/3 of the expected revenue [1]. At the same time, the non-recurring engineering (NRE) cost is rising exponentially with shrinking process technology [2]. Hence, meeting the myriad of conflicting design requirements in terms of performance, area and power limitations combined with the TTM pressure and NRE cost make customized solutions the only viable option in the domain of embedded systems design [3]. As such, embedded systems designed on Application-Specific Integrated Circuits (ASIC) and platform Field Programmable Gate Arrays (FPGA) are fast becoming the main competitors at the forefront of hardware customization.

ASICs provide the optimal configuration based on the requirements for performance, area or power for a given application but incur long TTM and high NRE cost. In contrast, FPGAs provide configurability and flexibility with shorter TTM and low NRE cost, even though the performance is comparatively low. For example, FPGAs typically reach the production stage within 3 – 5 months, whereas ASICs require

12 – 18 months for the same [4]. Further, modern FPGAs constituting reconfigurable logic, processors, memory subsystems, etc. have become the ideal platform to perform acceleration for critical parts of an application along with support for realizing an entire System-on-Chip (SoC). Thus, owing to the shorter development cycles, re-usability and heterogeneity of FPGAs, they are considered as an attractive alternative over ASICs for customized hardware designs. This is exemplified by the 7.23 billion USD FPGA market expected in 2022, with a Compound Annual Growth Rate (CAGR) of 7.41% from 2016 onwards [5].

However, the design of FPGA-based Configurable SoC (CSoC) is a challenging task due to the conflicting design constraints of performance, power, area and cost. Further, the flexibility of heterogeneity introduces the cost of added complexity. As such, the productivity of electronic design automation (EDA) tools is of paramount importance in meeting the aforementioned criteria. However, the productivity of EDA tools utilized in the development of embedded systems has significantly lagged behind the improvements achieved in hardware process technologies, which is referred to as the *design productivity gap* [6]. The widening of this design productivity gap with improvement in hardware process technologies, has created a major challenge for modern embedded systems designers. Thus, this research discusses techniques to address the widening design productivity gap by introducing an intelligent customization and exploration framework for automating FPGA-based CSoC platform generation.

1.2 Research Objective

To develop a methodology for rapid recommendation of application-specific FPGA-based CSoC platforms to provide the best performance under user-specified area constraints by proposing novel techniques for:

Processor Performance Estimation: A rapid and reliable technique for performance estimation of applications on embedded processors. The techniques should

support both soft and hard core embedded processors and is intended to be scalable to different embedded processor architectures.

Hardware Area-Time Estimation: A reliable technique for hardware area-time estimation of applications on FPGAs. The technique is intended to have the ability to support different FPGA vendors and device families.

Hardware-Software Partitioning: Novel technique for optimal partitioning of an application between hardware and software components of the FPGA-based CSoC while taking into account their complex data-dependencies.

Soft Core Processor Subsetting: Rapid and reliable techniques for instruction subsetting for soft core processor customization, in order to minimize area utilization. The saved area can be deployed to port additional portions of the application into hardware for further performance improvement.

Intelligent Framework for CSoC Platform Generation: Framework to generate application-specific FPGA-based CSoC platforms that achieves the best performance under user-specified area constraints. The framework is intended to recommend the best partitioning of the components of the application between hardware and software as well as the most appropriate processor (subsetting in the case of a soft core).

1.3 Organization

The rest of the thesis is organized as follows.

- **Chapter 2** presents an extensive discussion on existing literature. It initially discusses the available embedded computing architectures. A comparison of the advantages and limitations of the architectures moves the focus to CSoCs. Next,

existing design methodologies to generate such CSoCs are examined. Various research work done in the area of selecting and customizing the individual components of CSoC are described. The chapter concludes with specific objectives and motivations for the rest of this thesis.

- **Chapter 3** proposes techniques for rapid performance estimation of applications on soft core processors. The proposed technique is scalable to the large number of configuration options available in modern soft core processors by relying on rapid and accurate estimation models instead of time consuming FPGA synthesis and execution-based techniques. The proposed approach can be targeted towards different soft core processor architectures and underlying FPGA architectures.
- **Chapter 4** further extends the work on processor performance estimation to dual-issue, out-of-order, superscalar hard core processors, typically integrated in FPGA-based CSoCs. The approach can also be adapted to other embedded processor architectures.
- **Chapter 5** proposes a scalable technique that can be applied for hardware area-time estimation of different FPGA vendors and device families. The proposed technique seamlessly integrates with existing open source tools and free versions of vendor tools in order to provide area-time estimates. Further, the utilization of existing tools to obtain the final area-time estimation guarantees accuracy of results.
- **Chapter 6** proposes a methodology for rapid fine-grained data dependency- and area constraint- aware hardware-software partitioning. The technique has been shown to be extremely fast with runtime in the order of milliseconds as opposed to minutes/hours taken in an exhaustive approach as well as the state-of-the-art for the same applications.
- **Chapter 7** presents methodologies for rapid customization of soft core processors through instruction subsetting. The proposed methodology is scalable

and can be applied to different soft core processors with similar configurable functional units on a FPGA.

- **Chapter 8** proposes Wibheda+, a fully automated integrated framework for application-specific FPGA-based CSoC generation. The proposed framework incorporates all methodologies proposed in the earlier chapters of the thesis.
- **Chapter 9** concludes the thesis by summarizing the important contributions made in this research followed by recommended directions for future work.

1.4 Publications

The work presented in this thesis have been communicated in international journals and conferences as follows:

International Refereed Journal Papers

[J-1] **Deshya Wijesundera**, Alok Prakash and Thambipillai Srikanthan. “Framework for Rapid Performance Estimation of Embedded Soft Core Processors.” ACM Transactions on Reconfigurable Technology and Systems (TRETs), July 2018.

International Refereed Conference Papers

[C-1] **Deshya Wijesundera**, Alok Prakash and Thambipillai Srikanthan. “Rapid Design Space Exploration for Soft Core Processor Customization and Selection.”, The 15th International Conference on Field-Programmable Technology (FPT), 2016.

[C-2] **Deshya Wijesundera**, Alok Prakash, Siew-Kei Lam and Thambipillai Srikanthan. “Exploiting Configuration Dependencies for Rapid Area-efficient

Customization of Soft-core Processors.” The 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES), 2016.

[C-3] **Deshya Wijesundera**, Achintha Ihalage, Alok Prakash and Thambipillai Srikanthan. “Performance Estimation of Out-of-Order Superscalar Embedded Processors.” The 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART), 2017.

[C-4] **Deshya Wijesundera**, Alok Prakash, Thilina Perera, Kalindu Herath and Thambipillai Srikanthan. “Wibheda: Framework for Data Dependency-aware Multi-constrained Hardware-Software Partitioning in FPGA-based SoCs for IoT Devices.” The 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018.

[C-5] **Deshya Wijesundera**, Alok Prakash, Thilina Perera, Kalindu Herath and Thambipillai Srikanthan. “Wibheda+: Framework for Data Dependency-aware Multi-constrained Hardware-Software Partitioning in FPGA-based SoCs for IoT Applications.” The 9th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART), 2018 (**Best Paper Nominee**).

Chapter 2

Literature Review

2.1 Introduction

Modern embedded systems designers are constantly faced with the challenges of ever-increasing pressure of reducing the NRE cost and TTM coupled with the inability to harness the full potential of immense growth in silicon [6] [7]. The growth in silicon as predicted by the Moore's law has led to a *Design Productivity Gap* due to the difficulties of making effective use of devices, calling for efficient and effective design methodologies. Also, the contradictory set of design constraints of higher performance, lower area and power along with the NRE cost and TTM requirements has led to designers exploring the possibilities of introducing novel architectures and fast techniques for design space exploration.

In the architecture space, homogeneous architectures provide ease of programmability, flexibility and simplicity. However, in contrast, heterogeneous architectures could achieve better performance, power efficiency and product differentiation [8] [9] [10]. Heterogeneous architectures could be ASICs or reconfigurable platforms such as FPGAs. ASICs provide power and performance efficiency but the requirements for low NRE costs and TTM have made them a less attractive choice while, FPGAs offer

much lower NRE costs, TTM and superior flexibility. Thus, FPGAs have become a popular alternative to ASICs [11] [12].

This chapter will present characteristics of contemporary FPGAs constituting heterogeneous on-chip resources and discuss the inadequacy of existing design methodologies in exploring and capitalizing on the abundance of resources. The rest of the chapter is organized as follows. Section 2.2 discusses different embedded computing architectures, followed by a detailed discussion on design methodologies for CSoC in section 2.3. Section 2.4 discusses EDA Tools for FPGA-based CSoC while, conclusions are presented in section 2.5.

2.2 Embedded Computing Architectures

Contradictory design requirements and fierce competition has led to the emergence of different architectures in the domain of embedded systems. The diversity is such that on one hand it ranges from General Purpose Processors (GPPs) and Application-Specific Instruction set Processors (ASIPs) to ASICs and on the other hand to reconfigurable devices like FPGAs and Graphics Processing Units (GPUs). Embedded architectures can also be classified as homogeneous and heterogeneous. Homogeneous architectures would typically carry multiple instances of a single type of processing core. In contrast, heterogeneous architectures contain different types of processing cores and can contain single or multiple instances of each type. However, the proliferation of embedded systems and the myriad of design constraints has made heterogeneous architectures the preferred choice for many designers due to the immense flexibility offered [6]. As such, the next section discusses heterogeneous architectures in detail.

2.2.1 Heterogeneous Architectures

While homogeneous platforms are relatively easier to design and program, the increasing performance demand as well as stringent power budget requirements necessitates

the usage of heterogeneous platforms, especially in embedded systems. Different heterogeneous architectures have been proposed in both academia and industry in order to meet design constraints.

Lee et al. proposes, MorphoSys a heterogeneous system with MIPS-like reduced instruction set computer (RISC) processor core, a reconfigurable array, context memory with a direct memory access (DMA) controller and frame buffer for high speed memory transactions [13]. Similarly, Zhang et al. uses a heterogeneous architecture for baseband wireless applications to achieve high power efficiency [14]. The heterogeneous commercial platforms in [15] and [16] combine a GPP and other customized architectures. Even though, they are not necessarily part of embedded systems, the concepts used can be useful for embedded system designers. These architectures allow customers to benefit from FPGAs placed alongside traditional x86_64 based processors. It allows to accelerate part of an application on the high-end FPGA connected typically via the PCIe bus using the familiar desktop environment.

Even though, different application- and domain- specific off-the shelf heterogeneous platforms have been proposed, the increasing complexity of embedded applications demand configurability [17]. Hence, the ability to tweak the designs and reprogram over and over again at all stages of the design cycle offers immense flexibility and convenience. Thus, flexible and programmable reconfigurable architectures have gained momentum. Therefore, the next section presents these reconfigurable architectures.

2.2.2 Reconfigurable Architectures

Reconfigurable architectures have relatively shorter TTM as well as lower NRE cost, especially for designs which require flexibility and custom designs in low to mid volume products. However, these architectures can be customized for applications or domains such that it is possible to achieve high performance within low power budgets while providing flexibility for design changes at all stages of the design cycle. This

has led to the proliferation of reconfigurable architectures in spite of the lower power efficiency and performance compared to ASICs [18].

In the recent past, FPGA and GPU based hardware accelerators have become popular choices in the domain of reconfigurable architectures. However, both FPGAs and GPUs have an associated set of pros and cons. Da Silva et al. compared a Nvidia GPU and a FPGA in the image processing domain and concludes that both perform equally well [19]. The authors in [20] [21] conclude that GPUs are more beneficial in designs with high parallelism while FPGAs are extremely useful for relatively low power designs. Also, the modularity of FPGAs provide flexibility to implement any type of logic compared to the GPU's fixed structure which only allows acceleration of certain types of applications [20]. Thus, FPGAs are more suitable for a wider range of applications.

Further, modern FPGAs also provide numerous hard intellectual property (IP) blocks such as multipliers, dividers, digital signal processing blocks (DSP) etc. that provides the ability for design re-use, saving both time and effort making them attractive as a readily configurable system component. Further, the flexibility FPGAs provide even in implementing multiple applications from different domains make them extremely popular for space and military applications [22] [23].

At the same time, a large number of components (soft core processors, soft IP cores etc.) can be synthesized on current high-end FPGAs from vendors such as Xilinx, Altera etc. Xilinx and Altera have also incorporated ARM based hard core processors to their FPGA platforms, in addition to the soft core processors [24] [25]. Therefore, given the multitude of features offered by FPGA platforms both the research community and industry are exploring the possibilities of creating FPGA-based CSoC platforms in order to maximize the full potential of the available features.

Typically, CSoC is a computing platform constituting a processor and memory hierarchy using custom logic for acceleration [26]. A CSoC also allows for design and implementation changes for future proofing a commercial product. In the next section, the CSoC platforms are discussed in more detail.

2.2.3 Configurable System on Chip

A CSoC is a configurable or programmable heterogeneous platform on a single chip such as a modern FPGA [27]. Such CSoC architectures have been developed both commercially and in academia [26]. These CSoCs typically consist of a processor, memory and on-chip reconfigurable hardware which could be customized to a particular application [26] [28]. The main components constituting a modern FPGA-based CSoC platform are discussed below.

Processor

A CSoC can typically constitute of a hard core processor, a soft core processor, both or even multiples of each. A hard core processor is a hardwired processor core like the ARM processor in FPGA platforms [24]. Hard core processors provide optimum performance even though they are not configurable like the soft core processor counterparts. On the other hand, soft core processors, which are configurable can be customized using software tools for numerous configurations. Soft core processors provide flexibility for highly customized processor designs. A multitude of soft core processor architectures, both commercial and open source exist in literature [29]. For example, Altera Nios [30] and Xilinx MicroBlaze [31] are two leading commercial soft core processors, while Leon, OpenRISC, Amber etc. are open source soft core processors. Further, several leaner and domain specific soft core architectures have also been proposed in research [32] [33] [34]. However, these processors offer a myriad of configuration options. Thus, customization of the processors to obtain the best area-time trade-off is a challenging task to the designer.

Hardware Accelerators

These are custom units designed to accelerate sections of code based on performance and area constraints of the design. Hardware accelerators can be custom co-processors or IP cores. When there is an increase in demand for performance more parts of the

application code that perform better on hardware is ported to hardware. These hardware accelerators are connected outside the processor's data path on the CSoC and hence can be executed in parallel with the processor and thus, are called co-processors. The identification of such co-processors are usually done based on profiling results while the selection process considers the performance-area constraints in the design [28]. On the other hand, IP blocks are also a type of hardware accelerator typically implementing commonly used functionality. These are dedicated hardware blocks which are pre-verified, typically provided by FPGA manufactures [35] [36].

Memory subsystem

The memory subsystem constitutes on-chip memory such as block RAM, logic resources configured as distributed RAM, off-chip main memory and various levels of cache. Thus, it is important that applications are carefully analyzed in order to determine the size and configuration of the different types of memory. Also, in the case of hardware-software partitioned designs, it should be ensured that data shared between the hardware accelerators and the processor are synchronized accordingly in memory.

However, as already discussed, shifting towards FPGA-based CSoC poses several challenges to the designer in terms of making optimal use of the real estate and meeting the trade-off between performance, area and power under increasing NRE costs and TTM pressures. Modern CSoC architectures provide flexibility and lower cost of implementation but the complexity involved in design poses a significant challenge. Thus, is it inevitable that there is a need for methodologies which assist the designer by generating architectures that make optimal use of the CSoC platform while meeting design constraints.

As such, efficient design methodologies are of paramount importance in designing CSoCs for better performance, area and power efficiency [37]. Issues such as optimal

use of resources and achieving power-performance efficiency has opened up scope for research in design methodologies for CSoC platforms. Further, the International Technology Roadmap for Semiconductors (ITRS) in its reports emphasizes the need for suitable design methodologies to increase the productivity of designers [6] [7]. Hence, design methodologies for CSoC are discussed in detail in the next section.

2.3 Design Methodologies for CSoC Customization

As discussed in the previous sections a CSoC typically constitutes soft core or/and hard core processors, hardware accelerators and a memory subsystem. Hence, the designer has to select the most suitable parts of the application to be implemented in each of these components (hardware-software partitioning) in order to meet design constraints. Further, to perform rapid hardware-software partitioning, the designer has to estimate the performance of each component of the application on different heterogeneous architectures. This necessitates rapid performance estimation techniques for soft core processors, hard core processors, hardware accelerators etc. Further, the provision of customization in soft core processors introduce additional complexity to this process.

Currently, FPGA-based CSoC development predominantly relies on extensive manual tuning and hand-coding register transfer level (RTL) code [38]. This is typically done using hardware description languages such as Verilog and VHDL, which take significant amounts of effort. Also, state-of-art performance estimation techniques consume a significant amount of time, which calls for rapid and accurate estimation based techniques [39]. Further, most FPGA designers continue to perform hardware-software partitioning manually due to the absence of an efficient and effective methodology to facilitate rapid design space exploration [40]. Thus, this section will first discuss design methodologies for rapid performance estimation of soft core and hard core processors, collectively known as software performance estimation. Next, the focus will move to application-specific soft core processor customization.

Subsequently, techniques for hardware area-time estimation will be discussed. Finally, methodologies for constraint aware hardware-software partitioning will be presented.

2.3.1 Software Performance Estimation

Due to critical performance constraints in the design of CSoC based systems, processor performance estimation has been a hot topic of research [41]. It is important to make informed decisions at early design stages in order to meet stringent TTM and NRE cost requirements. Thus, performance evaluation is introduced early in the design cycle. Patel et al. discuss directions for embedded processor performance estimation and their salient features [42]. In this work, performance estimation techniques for embedded processors are grouped into two main categories namely, analytical models and simulation.

Analytical models identify numerical features of the application and extracts the features through static or dynamic analysis. Then, the feature set is used to correlate with application performance. Analytical models could be categorized as linear or non-linear models. Linear models utilize regression techniques and thus require a large number of applications to train the model in order to obtain high accuracy in estimation. Giusto et al. concluded that their model based on virtual instruction set and linear regression techniques is accurate only when considerably large benchmarks are used for training or when characteristics of the application is closely related to that of the training set [43]. Lattuada et al. presented a methodology to automatically build an analytical estimation model based on regression techniques and provides experimental results for ARM926EJ-S and LEON3 processors [44]. However, the correlation between the architectural features of the target processor and the application is strongly non-linear. Oyamada et al. presented a neural network based estimation model in [45]. However, this non-linear model requires not only the trained neural network but also the compiler and methods to extract the dynamic instruction count of the application [46].

Simulation based estimation techniques can be classified into 3 main categories, namely instruction set simulator based (ISSB), native execution based (NEB) and virtual platform based (VPB) simulation [42]. NEB methods can be further classified as binary(assembly) level, intermediate representation (IR) level (e.g. GIMPLE, LLVM) and Source level (e.g. C, C++).

Instruction Set Simulator Based Simulation

ISSB simulation models are written in a high level programming language to implement the instruction set architecture (ISA) of a target processor on the host machine. ISSB simulators could be instruction or cycle accurate [47]. Instruction accurate simulators ensure functional correctness while cycles accurate simulators emulate exact behavior of the target processor including its micro-architectural features like branch prediction, pipelining etc. Most ISSB simulators are interpretive, that is they mimic the target processor in fetching, decoding and executing instructions one after another at runtime. Such interpretive ISSB simulators behave closely to real hardware providing cycle accurate estimation of performance. However, this comes at the cost of low simulation speeds of a few hundred kiloinstructions per second to a few million instructions per second, due to run-time interpretation of target instructions. This is a major drawback of ISSB simulators in addition to the high development time and the inability to customize or modify the models [48]. SimpleScalar's *simfast* [49] and ArchC simulator *acsim* [50] are examples of popular ISSB simulators.

Native Execution Based Simulation

In the NEB simulation approach, the application under consideration is first instrumented with timing information of the target processor, re-compiled and then executed on the host machine. Instrumentation of the code is performed either at binary level, IR level or source level.

Binary Level Simulation (BLS) This approach translates the target binary of an application into host instructions or a high level language ensuring functionality is preserved. Timing information of the target binary is captured in the high level code which is compiled and executed on the host machine. Also, referred to as compiled ISSB simulation this approach performs time-consuming instruction fetch-decode-execute operations at compile-time by means of a simulation compiler thus, significantly improving the simulation performance. *PSIM* [51] and *accsim* [50] are popular compiled simulators incorporating BLS. Mills et al. [52] and Zivojnovic et al. [53] were among the first to propose the BLS technique. Bammi et al. described object-level performance estimation approaches by translating the assembly code generated by the target compiler to assembly-level functionally equivalent C and annotating with timing information [54]. Experimental results are presented for an image filtering application on a MIPS3000 processor. Nakada et al. proposed a workload specific simulator, which consists of a set of C functions corresponding to each basic block of the workload binary and other simulator functions including an ordinary emulator [55]. Using SPEC CPU95 benchmarks this method achieves a simulation speedup of 19 times compared to the *sim-fast* simulator of SimpleScalar.

Intermediate Representation Level Simulation (IRLS) In this approach, the target independent IR of the source code of the application is annotated with timing information of the target processor, followed by recompilation and execution on the host machine. In IRLS, instrumentation is host independent as it is performed prior to the host processor specific backend. Also, annotation is independent from the used high-level languages. Further, the control flow graph (CFG) can be extracted from the IR. Finally, application programming interfaces (APIs) are well defined to realize algorithms that operate on the IR [56].

Lee et al. presented a time-delta generation method for performance estimation and architecture exploration. The RTL IR of the GCC compiler is augmented with a counter field to count cycles of the IR operations and an indicator field to indicate the locations of memory operations in the IR [57]. Experimental results were presented for

three target processors, namely ARM7TDMI, MCORE and PowerPC 601 using nine benchmarks. Kempf et al. proposed a fine-grained software instrumentation applied on three address code IR level where all C operators and majority of memory accesses are visible [58]. Further, authors present experimental results for MIPS32 using blowfish and G.729 benchmarks. The authors have noted that conventional profiling tools such as *gprof* [59] and *gcov* [60] in GCC provide users with function level execution statistics for identification of program hot-spots and such coarse-grained information is not suitable for performance evaluation of embedded software. The approach has not taken the timing effects of the compiler backend and target processor into account thus, the timing values mentioned are not accurate [61] [41]. Cheung et al. presents a method where a software process written in C is compiled into an IR and annotated with timing information of the target processor [62]. This method uses a SystemC backend that emulates the backend of the cross-compiler and optimizes both IR operations and their associated latencies instead of generating an executable. The method uses the Tensilica Xtensa LX2 processor with MPEG2 decoder application for verification. Bouchhima et al. extends the LLVM compiler infrastructure to allow instrumentation of embedded software at the basic block level using a cross annotation approach [56]. The authors obtained a perfect match between basic block traces generated by annotated native execution when compared to the modified ISSB simulation for an ARM processor. However, performance estimation was not incorporated in this work.

The performance estimation tool TotalProf is introduced in [63]. Even though, the authors mention the technique as a source level one it is an IRLS technique since performance estimation and instrumentation was performed on the modified LLVM IR rather than on the input source code. TotalProf performs several IR-to-IR transformations, which consist of a virtual backend, a performance estimator and an instrumentor before the host backend takes place. The method is verified using the MIPS-4K architecture. Wang et al. proposed iSciSim for performance simulation of compiler-optimized software [48]. In this method, the CFG was significantly altered at binary level due to target architecture specific optimizations, and hence errors were introduced during back-annotation [42].

Source Level Simulation (SLS) This approach annotates timing information into the source code for a native execution that can generate performance information of the target processor. An automated timing back-annotation at the source code level is presented in [64]. This method requires the application source code, corresponding assembly and disassembled object files and execution traces generated by the virtual prototype as input. A delay is calculated for each line of the original source code and added to a timing annotated version of the application. This is then compiled and run on the SystemC/C++ timed functional simulator in order to obtain timing information. Schnerr et al. proposed an approach where instead of using the virtual prototype, the method performed back-annotation of timing values determined statically at basic block level in the binary code of the input C source code [65]. This method also considers architectural properties such as branch prediction. Experimental results were presented for Infineon TriCore processor with four applications. A cycle-approximate source-level performance estimation technique which relies on the C/C++ front-end of LLVM compiler infrastructure to parse the application into control data-flow graphs (CDFGs) is proposed in [66]. The performance estimation engine was evaluated with a MP3 decoder application using the processing unit model of the Xilinx *MicroBlaze* processor. Zhonglei et al. proposed a similar approach considering mapping between source level and binary level control flows as the basis for timing annotation which was verified on PowerPC for six benchmarks [67].

Virtual Platform Based Simulation

Virtual platforms are models of key components in a processor platform that can be assembled to form an executable subsystem [68]. Virtual platforms are of two types, hardware virtual platforms (HVPs) and software virtual platforms (SVPs). HVPs closely model the target hardware whereas SVPs provide enough functionality to execute the application code correctly and unmodified while retaining a higher-level of design abstraction. Modelling of HVPs is complex, time consuming and requires specialized knowledge. Further, HVPs are also very slow [64]. Even though,

instruction accurate, SVPs represent only functional aspects and thus is used mainly for early software development, integration and testing. QEMU is a well known open source SVP for processor emulation/virtualization [69]. This SVP can emulate several target processor architectures through dynamic translation. Also, it provides a set of device models and supports several guest operating systems. Xilinx has leveraged on QEMU to provide models for Zynq devices and MicroBlaze [70].

Open virtual platforms (OVPs) are also a type of SVP [71]. OVPs constitute of three main components: (1) open source models comprise of several model categories such as processor models, system components, peripheral models and pre-built platforms, (2) OVPsim simulator and (3) modelling APIs. OVP provides a comprehensive methodology to easily incorporate existing platforms or to configure and create new application-specific platforms utilizing virtual components compared to QEMU. Commercial plug-ins which work with OVP tool suites, provide advanced features such as profiling, coverage, memory checking etc. [72]. Moreover, Xilinx exploited Virtual System Platform from Cadence [73] for Zynq SoCs whereas Altera employed Synopsys's Virtual Prototyping Solution [74] for its SoC FPGAs.

The level of complexity in porting the estimation approach to a new processor depends on the degree of association of the approach with the underlined target architecture. ISSB simulators mimics the complete micro-architectural features of the target making them very less portable. In VPB approaches, porting to a new target involves efforts in relating target instructions with virtual instructions. NEB approaches (BLS, IRLS and SLS) are easily portable compared to ISSB approaches as the compilation is target independent. Even though back annotating timing details in the source code is architecture dependent this is a one time process. However, BLS requires the backend of the processor under consideration compared to IRLS which utilizes the target independent IR.

Aung et al. [75] utilize the binary level CFG to extract information and back annotate the details to estimate the performance using IR level CFG for the PowerPC

405 hard core processor. However, the technique is not readily transferable to new processor architectures. Powell et al. [76] present an execution time estimation framework for FPGA-based soft core processors. However, the method is limited to image compression applications. Eyerman et al. [77] proposed a method to model various characteristics of processors to estimate its performance. But, their approach requires the user to possess a deep understanding of the micro-architectural details that might not be feasible for commercially available processors. Ivošević et al. [78] proposed a model for function-level performance estimation for heterogeneous multi processor system-on-chip (MPSoC) platforms. However, they only showed results for a few functions. Furthermore, it is observed that existing methodologies for performance estimation are focused on single-issue, in-order processors or are limited to specific processor architectures [75].

2.3.2 Application-specific Soft Core Processor Customization

A CSoC can constitute hard core processors or/and soft core processors. Even though, typically hard core processors supersede soft core processors in terms of power and performance, works by Lysecky et al. [79] proved the competitiveness of soft core processors against similar low power hard core processors. Here, authors create a customized multi-core processor configuration using Xilinx MicroBlaze soft core processor on a low power Spartan 3 FPGA. The architecture proposed in this work shows competitive performance and power efficiency when compared to low power hard core processors such as ARM9, ARM10, ARM11 etc. Furthermore, according to recent research, soft core processors can be configured to offer performance that is comparable to specialized hard core processors [80]. A hard core processor, unlike a soft core does not allow the flexibility of adding a processor to an existing FPGA design or adding an additional processor to provide more processing capabilities [81]. While an un-utilized hard core processor is typically a wastage of FPGA area, a soft core could be added to a design only when necessary and the FPGA area occupied could be reduced further by appropriately customizing the core [80]. Additionally,

the design flexibility obtained through partial reconfiguration also makes soft core processors the preferred choice over their hard core counterparts in certain applications [82]. Thus, low cost and competitive performance of soft core processors which are especially optimized for FPGA makes soft cores a popular choice in FPGA-based CSoC platforms [83]. At the same time, the myriad of configuration options available in a typical soft core processor also poses an additional challenge to the designer in tailoring the processor.

Thus, methods for micro-architectural modification in soft core processors have been proposed by researchers in view of exploiting the benefits of a soft core processor. Architecture description languages (ADL) such as LISA [84] and EXPRESSION [85] have also been proposed for early design space exploration in processor modelling. However, for a given processor, micro-architectural modifications are typically carried out as instruction subsetting or instruction set extension [86]. Instruction subsetting sheds unwanted instructions from the processor ISA, where as in instruction set extension, novel instructions are added to the processor ISA as custom instructions. However, instruction set extension has been a broadly studied area in contrast to instruction subsetting. Hence, the next subsection discusses work on methodologies to customize processors using instruction subsetting.

Instruction subsetting

Instruction subsetting is defined as creating an application-specific instruction set processor from a more general processor, by removing support for unused instructions [87]. Dougherty et al. [87] introduced and demonstrated the advantages of instruction subsetting as a means for reducing power consumption by manually optimizing the assembly codes of applications. Further, instruction subsetting can provide significant improvements in performance while reducing area utilization [88] [89]. Also, the myriad of configuration options in modern commercial FPGA-based soft core processors, necessitates instruction subsetting [90] [31].

Thus, application-specific customization of soft core processors has been a hot research topic in the recent past due to its inherent benefits. Methodologies for soft core processor customization for specific types of applications are proposed in [91] [92] [34]. Yiannacouras et al. [93] and Padmanabhan et al. [91] perform application-specific customization on Nios and LEON processors respectively with significant gain in performance and area. However, these approaches necessitate synthesis and execution of the code thus, incurring exponentially increasing time with increasing number of configurable parameters. For example, an exhaustive search for a suitable configuration in MicroBlaze could take up to 11 hours [94] [89]. Sheldon et al. propose a synthesis-in-the-loop approach using impact ordered tree search heuristic [89]. This approach reduces the search to a fraction of the solution space but requires synthesis and execution of the code. The application-specific impact ordered tree based search requires a run time of 3.33 hours. The fixed ordered tree based approach can run in 1.5-2 hours but behaves poorly in certain scenarios. Further, this approach does not consider dependencies between configurations. Sheldon et al. propose improvements on the work in [89] using a design of experiments (DoE) approach [95]. However, this approach also requires synthesis runs of 2-3 hours and manual analysis and exclusion of infeasible configurations. Since the DoE tool does not analyze infeasible combinations, this methodology can have scenarios where the considered design space has a significant component of infeasible combinations which can have a direct negative impact on the results. Further, the DoE approach is implemented through manual intervention of the DOE PRO XL tool [96].

In contrast, Prakash et al. [94] claim to significantly reduce run time using an estimation-based approach. However, this method uses exhaustive estimation for each possible configuration and thus, will require significant effort with increasing number of configurations, even though only 12 configurations of Nios II are used for analysis purposes in this work. Also, the paper mentions that the LEON processor has $5 * 2^{10}$ configurations. Therefore, such a processor would increase the run time claimed in [94] by approximately 29 times. Moreover, the work does not analyze dependencies between configuration options, which can result in the selection of configurations

which have repetitive functionality or are not feasible. For example, the work mentions that dependency between the Floating Point unit and Floating Divider unit is not considered in the analysis.

2.3.3 Hardware Area-Time Estimation

FPGA compilation flow typically constitutes time-consuming logic synthesis, design placement and routing. The need to evaluate hardware designs at early stages in the design cycle in order to trade-off between design options has resulted in research efforts for hardware area-time estimation in FPGA-based CSoC designs. Existing work in hardware area-time estimation for FPGAs can be broadly categorized as high-level estimation, high-level synthesis (HLS) based estimation and RTL/logic-level estimation [97]. Some of the work combines two or all of these strategies. This section discusses these hardware area-time estimation strategies in detail.

High Level Estimation

In high level estimation, hardware area (resources) and time (latency) is estimated prior to scheduling. Typically, this category of approaches rely on probabilistic models and integer linear programming (ILP) models in predicting cycle counts and functional units of applications described in high level languages. Most of the early research efforts in hardware area-time estimation rely on high level estimation [41]. Hardware area-time estimation for efficient design space exploration by obtaining the lower bound execution time of a DFG, subject to area constraints is discussed in [98] [99]. Chaudhuri et al. proposed an algorithm to estimate the minimum area required to execute a DFG within a control step constraint [100]. Nayak et al. and Shi et al. suggested techniques to estimate hardware area using Matlab [101] [102].

A parametric area estimation methodology modeling low-level metrics as a function of high-level variables using high-level and low-level parameters derived through high-level model tuning (with synthesized RTL) and low-level model tuning (with the

circuit netlist) using 20 SystemC designs is proposed in [103]. A high level area estimation approach for Single Assignment C based designs is proposed in [104]. This method incorporates approximation formula for various arithmetic operators together with heuristics that take into account logic optimizations typically applied by the synthesis tool. Todman et al. presented a high-level design exploration framework, which features a hardware performance estimator using area-time constraints and a C application as input [105]. The methodology employs an abstract syntax tree (AST) to represent the application. Then the transformed AST is used to estimate the area and time. Meeuws et al. uses a statistical model in estimating hardware area introducing 92 software complexity metrics and several software measurement techniques such as cyclomatic complexity as indicators of different aspects of software descriptions [106]. Zhong et al. propose an estimation framework, to evaluate area/time metrics of various accelerator options for an application at an early design phase [107].

High Level Synthesis (HLS) Based Estimation

The advent of HLS tools which could translate algorithmic descriptions in high level languages to RTL has led to the development of interest on HLS based estimation. HLS tools typically perform scheduling and resource binding based on the IR of the application. The RTL generated by the tools typically consist of a data path and a control path. An estimation technique based on execution traces generated from simulation of Matlab programs to estimate area and time of a FPGA implementation, incorporating greedy scheduling and binding algorithm is proposed in [108]. Schumacher et al. uses force-directed scheduling to determine the number of operators in execution time steps of the input Matlab code [109].

A methodology to estimate area and time on FPGA for applications written in Java is proposed in [110]. Bilavarn et al. performs area, time and power estimation by transforming C functions to hierarchical control data flow graphs [111]. Lieu et al. rely on the Trimaran compiler infrastructure in the area-time estimation framework proposed in [97]. The proposed technique targeted modified very long instruction

word architecture as an architecture template for FPGA implementation and employed a hybrid approach incorporating bit-width sensitive estimation models and a set of pre-characterized values in the estimation process for applications written in ANSI-C. Simplified floor planning was also proposed in this approach in order to predict the placement of hardware functional units on FPGAs to improve the accuracy of time estimation. The experimental results use FPGAs with 4-input Look-up Tables (LUTs).

Further, existing HLS based hardware area-time estimation techniques use the addition of the area of individual operators in estimating the area required for the data path while using analytical models for the control path [97] [101]. Also, most of the existing techniques for estimating the critical path delay are based on the longest latency of the execution unit. However, the work in [101] [97] incorporates the interconnect delays in estimation.

RTL/Logic-Level Estimation

The input RTL description or the netlist of the circuit is used for estimation at RTL/logic-level. The techniques proposed in [109] use a commercial HDL parser [112] in resource estimation. An area-time estimation technique using an input netlist is proposed in [113]. The authors in this work, decompose the netlist into LUTs and allocate the LUTs with the most common inputs to the same configurable logic block (CLB). In estimating the routing resources, the work also predicts the placement and shape of CLBs. Both CLB delays and wire delays are used in calculating the path delays.

Hardware area consumption and time is dependent on the architecture of the FPGA device. The architectural features of a FPGA device differs across vendors and also FPGA device families. As such, existing techniques for hardware area-time estimation are limited to a single FPGA device or a few devices from a single vendor. Extending such techniques to support different devices require individual modelling of each of the devices which is cumbersome and time consuming. Also, due to this factor, scaling

such techniques to support new devices will require modelling of each of those device as well. However, in order to make optimal use of hardware area-time estimation by providing the user the option to select between devices, it is important to support different devices.

2.3.4 Hardware-Software Partitioning

Stringent design constraints have resulted in hardware-software partitioning being a critical part in the design of CSoC based systems. Typically the designer is vested with the responsibility of deciding whether an application component should be implemented in hardware or software to meet specified system constraints. It has been shown that efficient techniques for partitioning can achieve results in performance or energy efficiency superior to a pure software solution [41]. Traditional approaches to hardware-software partitioning could be hardware-oriented or software-oriented. The hardware-oriented approach starts with a complete hardware solution and iteratively moves parts of the system to the software, within performance constraints. The software-oriented approach starts with a software program and moves parts of the system to hardware until the performance constraint is satisfied [114].

However, partitioning the design into its hardware and software components is one of the biggest challenges in architecting an embedded system, as badly partitioned designs can lead to inefficiencies due to data dependencies between components [115]. Partitioning decisions must typically be made early in the design of a product. However, the consequences of lack of proper analysis at this stage can result in higher NRE costs and TTM delays, or even an inability to meet design requirements [116]. Hence, it is important for the partitioning methodology to carefully analyze the applications at this early design stage and accurately model the system while considering the plethora of different implementation possibilities [117].

Many algorithms have been proposed for hardware-software partitioning. The approaches can be mainly categorized as exact and heuristic. However, finding the op-

timal partitioned solution is an NP-complete problem [118]. As such exact approaches such as branch and bound, integer linear programming [119] [120] and dynamic programming [121] [122] are not feasible for complex applications. Thus, heuristic approaches such as Tabu Search [123] [124], Simulated Annealing [123] [125] [126], Knapsack [124] [127], Particle Swarm Optimization [128] and Genetic Algorithm [151] [128] have been proposed. Further, several approaches have been proposed where two heuristics have been combined, in view of obtaining solutions that are close to those obtained by exact techniques while minimizing run-time overheads. Weijia et al. propose a method combining clustering algorithm and genetic algorithm [129]. Jing et al. [130] and Garg et al. [131] proposed approaches combining knapsack with simulated annealing. The latter work claims to achieve a performance significantly better than that of simulated annealing. Wu et al. proposed an approach to combine knapsack with a tabu search algorithm [124]. The works in [132] [133] incorporate simulated annealing with genetic algorithms.

Yousuf et al. present a methodology for design automation for partial reconfiguration (DAPR) design flow, for hardware-software co-designed systems [134]. DAPR's design flow isolates low-level partial reconfiguration design complexities involved in analyzing partial reconfiguration designs with different performance parameters to make partial reconfiguration more amenable to designers. Zhang et al. propose a new mixed integer linear programming (MILP) formulation for hardware-software partitioning targeting a MPSoC platform with a dynamic partial reconfigurable fabric [135].

Tang et al. formulate the optimization of hardware-software partitioning aiming at maximizing streaming throughput with predefined area constraint, targeted for multi-processor system with hardware accelerator sharing capability [136]. Authors propose software-oriented and hardware-oriented greedy heuristic algorithms for hardware-software partitioning and verifies them on several random graphs and one multimedia application (MP3 decoder). Results show that the best result from both proposed

greedy algorithms produce 93.6% accuracy compared to brute force ground truth with faster hardware-software partitioning time.

Govil et al. present a multi-dimensional global optimization approach with the aim of minimizing the area utilized, power dissipated, time of execution and system memory requirement for an implementation in hardware-software partitioning [137]. The authors also perform a comparison of the execution time of the proposed algorithm with other major partitioning algorithms. The time of execution is 0.011 seconds, with deviation from the optimal solution by less than 2%, for partitioning an algorithm consisting of 2500 blocks.

However, it should be noted that the above works do not take into the data dependencies arising as a result of hardware-software partitioning. Thus, data manipulated by hardware has to be updated in the appropriate memory location so that software uses the updated data value and vice versa. This results in an overhead for the partitioning step and must be carefully accounted for. Thus, the next subsection looks at memory-aware hardware-software partitioning.

Memory-Aware Hardware-Software Partitioning

As mentioned in the previous section, usage of memory blocks in CSoC based systems necessitates data sharing and communication as well as synchronizing between the hardware and software components of the application. Thus, it is important that a partitioning algorithm facilitates efficient memory sharing between components. The authors in [138] [139] [140] proposed transferring the relevant data from the main memory to the local memory of the FPGA by a DMA transfer in a basic block before the custom hardware block executes. Once the hardware block executes on the data in its local memory, the data is transferred back to main memory using another DMA transfer cycle. A significant amount of time saved by executing the code on hardware is eventually lost due to DMA transfers [138] [140]. The authors also proposed

techniques to minimize the effect of these multiple DMA transfers but these have considerable limitations.

Henkel et al. proposed copying of relevant data in shared memory space each time before executing either software or hardware components [125]. Sun et al. proposed a shared memory approach requiring data transfer between different memory blocks during execution [141]. Prakash et al. propose a model to estimate the penalty incurred due to memory dependencies between the program running on the processor and the local memories in the FPGA hardware accelerators [142]. Authors claim that this model can be used in an automated design exploration framework for heterogeneous FPGA platforms to select profitable hardware accelerators with local memories.

The granularity at which to partition an application poses yet another challenge for the designer during the partitioning step. The merits and de-merits of different granularities of partitioning are discussed in the next section.

Granularity of Hardware-Software Partitioning

Different approaches have been proposed to partition an application at a fine or coarse granularity [125] [143]. Compared to a fine-grained approach, a coarse-grained approach implements large sections of code where only a small fraction of the code may actually execute frequently enough to provide any meaningful acceleration in execution time, resulting in reduced return on the investment of the hardware area [115]. On the other hand, a fine-grained approach provides much greater control on the accelerated code segments resulting in higher acceleration in execution time for the same hardware space, but leads to a much larger design space for the designer to select the best application code segments for acceleration [125]. In addition, the larger design space, results in increase in data communication between components, incurring higher data communication cost. Therefore, it is important to accurately model data communication costs in a fine-grained approach.

As mentioned the problem of finding the optimal partition under a given constraint is NP-complete, resulting in exponential increase in runtime with increase in design space for an exact formulation. As such, heuristic based approaches have been proposed. However, the solution quality of these approaches is highly dependent on the initial solution seed. Further, it is important that a partitioning algorithm facilitates accurate modeling of data communication. However, the current state of practice offers only limited accuracy at modeling time [117]. Also, literature reveal that existing techniques are not able to support designs of varying size and complexity. Some of the approaches are limited to coarse grained partitioning as a result of which smaller designs cannot be supported [117]. On the contrary, Prakash et al. [144] propose a rapid methodology for memory-aware hardware-software partitioning at finer granularity and accurate modelling of the communication penalty. However, the authors only consider the number of basic blocks as the design constraint which results in large errors for an area-constrained design. As such, there is a lack of reliable methodologies for memory-aware hardware-software partitioning which can be applied to designs of varying size and complexity.

The next section looks at EDA tools proposed in academia as well as available commercially, which can be used for design space exploration and hardware-software partitioning in CSoCs.

2.4 EDA Tools for FPGA-based CSoC

A myriad of EDA tools for FPGA could be found both in academia and commercially. The tools support different high-level languages, optimization features and target devices. However, ANSI-C is the most commonly used programming language in the domain of embedded systems and thus, a significant amount of legacy code is written in C [145] [146]. Further, software tool chains like compilers and debuggers

are readily available for ANSI-C and most embedded systems designers are familiar with the C language. Therefore, this reduces the learning curve in the developer's perspective. Moreover, many applications have C reference implementations serving as a golden description of functionality. Therefore, many tools support functionality for C applications.

2.4.1 Academic Tools

GAUT is a well-known academic tool mainly suited for domain-specific digital signal processing applications targeting FPGAs [147]. ROCC is another such tool focused on domain specific streaming applications for FPGA [148]. GAUT and ROCC generate highly pipelined, parallel and optimized VHDL code from a subset of C input algorithms. However, they do not have provision for application profiling, thus the designer is responsible for application analysis.

The Panda framework for hardware software co-design includes the Bambu tool for high level synthesis of memory implementation in complex applications [22] [149]. Bambu can be used to automatically infer and implement suitable memory hierarchies during the high level synthesis of complex applications. This methodology is used to classify the variables according to their viable placement in internal or external memory blocks and fully supports all constructs in the standard C language.

Luk et al. proposed a high level compilation tool chain for mapping applications on a heterogeneous architecture constituting a GPP, DSPs and FPGAs [150]. This tool chain takes a C application annotated with partitioning information as input and maps to different functional units for efficient execution. The hArtes "Holistic Approach to Reconfigurable Real Time Embedded Systems" project is aimed at creating a complete design flow for reconfigurable heterogeneous platforms [151] [152] [153]. Bluebee technologies commercializes the outcomes of the hArtes project especially for Big Data processing systems [154].

LegUp is a more recent open source HLS framework for FPGA-based embedded systems [28]. The design flow in this tool first compiles and runs an application on a standard MIPS processor, performs hardware-assisted application profiling, selects code segments to target hardware and then recompiles the application to a hybrid system. However, LegUp necessitates availability of the target processor for application profiling. Further, in the LegUp [155] tool, hardware-software partitioning is limited to Altera FPGAs, and it does not support Altera's own Nios soft core processor. Moreover, Legup provides full support only for Altera FPGAs, offering only limited support for other FPGA vendors. Also, automated application-specific soft core processor customization is outside the scope of LegUp.

2.4.2 Commercial Tools

Leading vendors in EDA such as Cadence [156], Synopsys [157] and Mentor Graphics [158] have released their own system level design tools to leverage on the platform-based design concepts. The Synopsys tool chains include tools for architecture design, processor design, platform architecture etc. These tools can be used to model and design systems at various levels of abstraction and granularity. The Vista tool chain by Mentor Graphics constitutes solutions for architectural design space exploration, virtual prototyping etc which could be useful in hardware software partitioning [158]. Cadence's Tensilica XPRES compiler is a HLS tool specifically targeted for the Xtensa processor based systems by Tensilica itself [156] [159]. Design framework development for FPGA-based embedded systems starting at design specification written in C, is carried out by several commercial tools. CoDeveloper from Impulse Accelerated Technologies [160], NEC's CyberWorkBench [161], eXCite [162], SpaceStudio [163], Triton tool suite [164] are some of the commonly used commercial tools.

SDSoC by Xilinx [165] and SpaceStudio [163] by Space Codelsign are design exploration tools where the designer can select the code segments for hardware implementation using pragmas while the tool performs automated system connectivity generation. However, SDSoC supports only Xilinx Zynq Devices while SpaceStudio

supports Xilinx Zynq and Altera Stratix 10 and Arria 10 devices. Furthermore, the hardware-software partitioning process is a manual one in both the tools where the optimal partition can only be found by trial and error. SDSoC also lacks support for accommodating a soft core processor.

In a nutshell, existing literature reveal the lack of fast and accurate estimation based techniques for selection of the best FPGA-based SoC device from multiple vendors and subsequent partitioning of the design.

2.5 Summary

As discussed in this chapter, FPGA-based CSoCs have been gaining traction in the recent past due to heterogeneity, reprogrammability, lower NRE cost and faster TTM. However, both academic and commercial tools, which are currently available for application-specific customization of FPGA-based CSoCs require expert manual intervention. Furthermore, these techniques are mostly limited to a single FPGA vendor or tightly coupled to device families. Thus, the need for rapid and intelligent techniques for customization of FPGA-based CSoCs from different vendors and device families is identified.

A FPGA-based CSoC typically entails software components such as soft core processors and hard core processors and hardware in the form of the FPGA fabric. Thus, rapid and intelligent customization of such a CSoC requires the area and time of the application on these components. Further, in order to partition the design between the different components, it is also important to obtain the optimal partitioning of the application between the hardware and software. Thus, a unified intelligent approach to CSoC customization entails performance estimation of software, hardware area-time estimation and hardware-software partitioning techniques.

The literature survey reveals performance estimation is preferred over software-intrusive and hardware-assisted techniques to rapidly extract software performance

metrics. However, the existing techniques for performance estimation are tightly coupled to specific processor architectures. Also, modern FPGA platforms not only support soft core processors which are synthesizable on the FPGA fabric, but also hard core processors which have been pre-fabricated. Thus, the need for scalable performance estimation techniques for both soft core processors and hard core processors is identified.

Furthermore, literature reveal that existing hardware area-time estimation techniques rely on architectural analysis and modelling of individual devices. Thus, existing hardware area-time estimation techniques are limited to specific FPGA vendors and device architectures. Extending such techniques to different FPGA vendors and devices require detailed analysis and modeling of individual FPGA devices, which is cumbersome and time consuming. As such, the need for scalable hardware area-time estimation techniques that can be applied across different FPGA vendors and devices is identified.

Another important factor in customization of a CSoC is the partitioning of the design between the software and hardware components. As discussed earlier in this chapter, there is a lack of a single methodology for hardware-software partitioning that can support applications of varying size and complexity. Course granularities of partitioning lead to inability to support smaller applications while, inefficient analysis of communication cost in fine-grained partitioning inhibits supporting complexities of applications. Obtaining the optimal partition is of paramount importance in order to improve the performance-area ratio. Thus, the need for hardware-software partitioning techniques that can be applied to designs of varying size and complexity is identified.

Soft core processors provide different configuration options, thus it is important to ensure that only configurations useful for the application under consideration are enabled in order to save area which can be used for hardware acceleration in a hardware-software partitioned design. The techniques proposed in literature for soft processor customization are either based on synthesis or exhaustive estimation. Thus, the techniques are time consuming specially with increasing number of configurations.

Thus, the need for rapid techniques for customization of soft core processors is identified.

Chapter 3

Performance Estimation of Soft Core Processors

3.1 Introduction

The sheer number of soft core processors available in the market, both commercial and open source, makes it a daunting task to select the best processor that achieves optimal performance within a given FPGA area constraint. This, in combination with the large number of configuration options available, even in a single soft core processor, makes the design space of selecting the *best processor with its best configuration* for a given application, prohibitively large. Hence, a designer cannot rely on traditional time consuming FPGA synthesis-based techniques. Further, it should be noted that the traditional synthesis-based techniques rely on the access to the necessary FPGA boards and the corresponding commercial tools to identify the best processor with its best configuration. Thus, it prevents designers from using them at an early design stage prior to the availability of boards and commercial tools which directly add to the TTM. Also, the requirement to obtain a large number of costly development boards and the necessary tools adds to the NRE costs.

Thus, rapid software-based performance estimation techniques that can be employed repeatedly and reliably to customize the processor is of paramount importance in not only obtaining better performance but also to evaluate the necessity of configuring the processor within the design. This chapter proposes a rapid design space exploration methodology for performance estimation of soft core processors that relies solely on estimation based techniques instead of performing time consuming FPGA synthesis steps. It is adaptable to different embedded soft core processor and target FPGA architectures. The main contributions of this chapter are:

- A methodology for characterization of processor instruction set architecture
- Analysis of data hazards and control hazards in the processor pipeline
- Analysis of the variation of performance with operand size for *multiplication* and *shift* instructions
- A methodology for rapid soft core processor performance estimation

The rest of the chapter is organized as follows. Section 3.2 discusses the methodology and the results are presented in section 3.3. The conclusion is presented in section 3.4.

Portions of the work discussed in this chapter have been published in [166] and [167].

3.2 Methodology

This section presents the methodology for performance estimation of soft core processors using the open source LLVM compiler infrastructure shown in Figure 3.1. The proposed methodology is scalable and can be extended to other embedded soft core processors. Additionally, it can be easily re-targeted for different FPGA devices and specifications. As shown in Figure 3.1, the methodology can be abstracted into

Figure 3.1 Framework for Soft Core Processor Performance Estimation

three phases, namely processor characterization, application profiling and performance estimation. These phases are described in detail in the subsequent sections. However, prior to the processor characterization step needed for any target processor, the LLVM instructions are classified into 4 predefined categories as follows:

- Configurable Functional Unit (CFU) Instructions: LLVM instructions corresponding to the processor instructions that can be accelerated on the CFU
- Standard C Library Functions: LLVM instructions corresponding to standard C library functions
- Custom Instructions: Instructions defined for custom functionality in order to obtain higher performance
- Remaining Instructions

The 4 main categories of instructions, specified above, have also been further divided into subcategories based on other properties such as bit width and their specific hardware implementation option, if any. After classifying the LLVM instructions, the processor characterization step is performed.

3.2.1 Processor Characterization

Processor characterization is a one-time semi-automated step for a given target processor. This includes classification of the target processor ISA (by analyzing the processor datasheets) and simulation of the base processor for a set of scripts, created in this work, using free editions of the software. The detailed analysis of processor characterization is provided subsequently.

This phase first maps the ISA of the relevant processor to the instruction set of the LLVM ISA. Given a new processor, the processor ISA is also classified under the same categories/subcategories and mapped to the LLVM ISA by matching the equivalent instructions in the two ISAs. Table 3.1 depicts the classification of LLVM instructions and an example mapping of these instructions to processor instructions for the case of

the *Nios* processor. Next, the methods used to obtain the execution time for each class of instructions are discussed.

CFU Instructions For CFU instructions, the execution time is obtained when the CFU is enabled, which is called the hardware time. Similarly, the execution time when the CFU is disabled (i.e. the instruction is executed in software) is called the software time. The hardware time in all cases can be directly obtained from the data sheet. The software times too can be obtained from data sheets except for *multiplication* and *shift* instructions, where the software times have to be obtained experimentally by simulating the base processor. For this simple scripts running on free editions of the software have been written. Further, it was observed that the software time of *multiplication* and *shift* CFU instructions depend heavily on the input bit width of the operands. Thus, the variation of software times of these instructions based on the bit width of the operands (e.g. 5-bit, 62-bit etc.) is obtained. The software time of the remaining CFU instructions are independent of the operands.

C Library Functions LLVM supports the standard C library intrinsics as fundamental instructions [168]. Thus, micro-benchmarks are used to obtain the execution time of the C library functions. C library functions are initially separated into two distinct categories, namely functions with fixed execution time and functions with varying execution time. For functions with fixed execution time, the function is simulated thousands of times to obtain an average execution time. However, for functions with varying execution time the method proposed in [169] where a variable component is added to the execution time depending on the input characteristics is used. For example, in `printf()` function depending on the input string length a variable component is added to the execution time. The variable component is extracted based on micro-benchmarks similar to fixed execution time functions. However, it should be noted that irrespective of the classification, micro-benchmarks used above can be simulated once on a base-processor with free editions of the software.

Table 3.1 Classification of LLVM Instructions

Type of Instructions			Processor Instructions	LLVM Instructions
CFU Instructions	Configuration On (Implemented on hardware),		Mult. I64	mul i64
	Configuration Off (Implemented on Software)		Arith. Right Shift I32	ashr i32
			Floating Point Division	fdiv
			...	...
C Library Functions	Fixed Execution Time		Sine	llvm.sin*
			Rounding Numbers	llvm.round.*
			...	...
	Varying Execution Time		Print	@printf
			Square Root	llvm.sqrt.*
			...	...
Custom Instructions	One to Sequence		Multiply	mul i32,
			Accumulate I32	mul i32,
				add i32
			...	...
Remaining Instructions	Fixed Execution Time		Add I32	add i32
			Subtract I32	sub i32
			Casting I32	zextb i32
			...	...
	Varying Execution Time	Control Hazards	Switch Case	switch
			Branch	br
		Data Hazards	Load	load
			Store	store

Custom Instructions A processor can be customized by bolstering its base instruction set architecture with specific custom instructions for higher performance. Such an instruction typically constitutes a sequence of instructions from the base instruction set of the processor and maps to a sequence of instructions in the LLVM IR. The proposed framework identifies such custom instructions by matching this sequence. This process is carried out automatically, when the sequence is provided as input to the framework. The execution times corresponding to the instructions can be obtained from data sheets.

Remaining Instructions Initially, these instructions are classified into two categories, namely fixed execution time and varying execution time. The execution time of fixed execution time instructions can be obtained directly from the data sheet. The instructions with varying execution times are further sub categorized based on the reason for the variation. Thus, instructions are classified into control hazards and data hazards. The variation of execution time in these instructions occur due to the micro-architectural features of the underlying processor. In the proposed approach, models have been devised in order to account for these behaviors. These models are described in detail in subsection 3.2.3.

3.2.2 Application Profiling

This is a one-time phase for a given application. The application is profiled using the LLVM open source compiler. The profiled output obtained from LLVM is used to extract basic blocks, instructions in these basic blocks and execution frequency of basic blocks. Additionally, metadata required for the analysis of the varying execution time instructions is extracted in this phase. For example, the profile output is used to obtain the number of bits to be shifted of *shift* instructions, branch weight metadata (to extract the edge weights of *branch* instructions), the number of cases and presence of default in *switch* instructions, etc. A sample LLVM profile output is shown in Figure 3.2. Details on interpreting the LLVM profile can be found in [168].

```

1  LLVM Profile
2
3  ;;; %gsm_add called 79 times.
4  ;;;
5  ; Function Attrs: nounwind uwtable
6  define signext i16 @gsm_add(i16 signext %a, i16 signext %b) #0 {
7  entry:
8
9      %a.addr = alloca i16, align 2
10     %b.addr = alloca i16, align 2
11     %sum = alloca i64, align 8
12     store i16 %a, i16* %a.addr, align 2
13     store i16 %b, i16* %b.addr, align 2
14     %0 = load i16* %a.addr, align 2
15     %conv = sext i16 %0 to i64
16     %1 = load i16* %b.addr, align 2
17     %conv1 = sext i16 %1 to i64
18     %add = add nsw i64 %conv, %conv1
19     store i64 %add, i64* %sum, align 8
20     %2 = load i64* %sum, align 8
21     %cmp = icmp slt i64 %2, -32768
22     br i1 %cmp, label %cond.true, label %cond.false
23     ;;; Out-edge counts: [7.900000e+01 -> cond.false]
24
25     cond.true:                                ; preds = %entry
26     ;;; Never executed!
27     br label %cond.end7
28

```

Figure 3.2 Sample LLVM Profile

The *LLVM Tracer* tool proposed in [170] is used to obtain the bit width and sign of operands for all *multiplication* instructions. This tool first creates a dynamic trace of the application using the LLVM profile output. This dynamic trace is then used to obtain the bit width of all the *multiplication* instructions present in the application profile. Thus, the model first checks for the presence of *multiplication* instructions in the LLVM profile output. If *multiplication* instructions are present in the instruction profile, the application is profiled using the *LLVM Tracer* tool to obtain a dynamic instruction trace. Then the operand bit widths and signs for *multiplication* instructions are extracted from this trace. Figure 3.3 shows a sample of the *LLVM Tracer* tool output. Multiplication instructions can be identified in lines 10 and 27 (last comma separated word is "mul"). The operands are depicted in the 2 preceding lines as the 3rd comma separated field. For example the operands for the multiplication indicated in line 10 are given as the 3rd comma separated field in lines 8 and 9 (operand values are 2048 and 81).

3.2.3 Performance Estimation

This phase uses the target processor information from the Processor Characterization phase (3.2.1) and the information regarding the application from the Application Profiling phase (3.2.2). These information are fed into a rapid estimation based technique to calculate the performance (total execution time) of the application on the given processor. The subsequent sub-sections describe this estimation process in detail by elaborating the steps used to obtain the execution time of different categories of instructions present in the given application.

Input Operand Size Dependent CFU Instructions

As mentioned earlier, the software time of certain instructions are dependent on the bit width of operands. Thus, these types of instructions are specifically identified to

1	<i>LLVM Tracer Tool Output</i>
2	
3	0,59,gsm_mult_r,if.else,conv6,35,3290
4	1,16,2048,1,3
5	r,64,2048,1,conv6
6	
7	0,59,gsm_mult_r,if.else,mul,12,3291
8	2,64,2048,1,conv6
9	1,64,81,1,conv5
10	r,64,165888,1,mul
11	
12	0,59,gsm_mult_r,if.else,add,8,3292
13	2,64,16384,0
14	1,64,165888,1,mul
15	r,64,182272,1,add
16	
17	0,59,gsm_mult_r,if.else,if.else-0,28,3293
18	2,64,140733230557360,1,prod
19	1,64,182272,1,add
20	0,59,gsm_mult_r,if.else,conv6,35,3290
21	1,16,2048,1,3
22	r,64,2048,1,conv6
23	
24	0,59,gsm_mult_r,if.else,mul,12,3291
25	2,64,2048,1,conv6
26	1,64,81,1,conv5
27	r,64,165888,1,mul
28	0,59,gsm_mult_r,if.else,conv6,35,3290

Figure 3.3 Sample LLVM Tracer Tool Output

analyze their specific dependencies. *Multiplication* and *shift* instructions fall under this category.

It is observed that the software time of *multiplication* instructions are dependent on type of operands, sign of operands and the bit width of operands. Once again, it was experimented using micro-benchmarks to obtain the pattern of variation of execution time with varying types, signs and bit width of operands. For example, initially software multiplication operation is classified as 64-bit multiplication or 32-bit multiplication. Then, the sign of the operand is checked and finally the operands are classified based on the corresponding bit width of the binary representation of the operand. This data is used in conjunction with the profiling information in the estimation model.

Similar to the *multiplication* instructions, it was observed that the software time of *shift* instructions is dependent on the number of bits to be shifted. Thus, this data was obtained experimentally for both 32-bit and 64-bit *shift* instructions and incorporated in estimation models.

The execution time of the remaining instructions have also been obtained experimentally. The time in the case of these instructions are not dependent on the operands.

Instructions leading to Control Hazards

Control hazards are pipeline hazards that arise from the pipelining of branches and other instructions that change the *program counter*. The instructions in Table 3.1 which lead to these control hazards are discussed below.

Switch Instructions The execution time of *switch* instructions corresponding to switch case statements is dependent on the number of case statements and the presence of a default statement. Eq. 3.1 is used to compute the execution time of the *switch* instruction.

$$T_{switch} = (t_{case} * n_{case}) + (t_{default} * d) \quad (3.1)$$

where T_{switch} = Execution time of *switch* instruction,

t_{case} = Execution time of a single case statement,

$t_{default}$ = Execution time of default statement,

n_{case} = Number of case statements,

d = Presence of a default statement ($d = 1, 0$)

Branch Instructions Branches can be conditional or unconditional. The estimation of unconditional branches is trivial as the time could be directly extracted from the data sheet. However, estimation of conditional *branch* instructions requires information on branch prediction and misprediction. It is not possible to obtain this information at the LLVM IR level. Thus, a model based on branch entropy is used to calculate the execution time of branches [170].

Branch Entropy

Branch Entropy is defined as a measure of the randomness of branch behavior, representing branch predictability [170]. This can be used to calculate branch prediction and misprediction penalties at IR level for conditional branches. Edge frequencies of conditional branches obtained from LLVM metadata is used in the prediction of these branches. The model for entropy is shown in Eq. 3.2. The clock cycles to execute conditional *branch* instructions is calculated using Eq. 3.3.

$$Entropy = - \sum_{i=0}^n P_i * \log_2 P_i \quad (3.2)$$

$$\text{where } P_i = \frac{f_i}{\sum_{i=0}^n f_i},$$

$$n = \text{Number of branches,}$$

$$f_i = \text{Edge frequency of branch } i$$

$$cc_{cond_br} = f_{bb}(Entropy * cc_{mp} + (1 - Entropy) * cc_p) \quad (3.3)$$

$$\text{where } cc_{cond_br} = \text{Clock cycles to execute a conditional branch,}$$

$$cc_{mp} = \text{Clock cycles to execute a mispredicted branch,}$$

$$cc_p = \text{Clock cycles to execute a predicted branch}$$

Instructions leading to Data Hazards

Data hazards are pipeline hazards that arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. The instructions in Table 3.1 which lead to these data hazards are described below.

Load-Use Dependencies Load-use dependencies occur when the data to be fetched is used by a subsequent instruction. In this case, even though the IR level profile does not indicate any instructions between the load instruction and the instruction using the loaded data, the actual implementation of a processor would stall the processor pipeline to compensate for the latency in loading data. The model identifies such load-use dependencies and compensates for the latency.

Consecutive Load Instructions Pipelining reduces the latency of *load* instructions when there are consecutive *load* instructions. The proposed model analyzes the instruction profile to identify consecutive *load* instructions and adjusts the latency according to the usage of pipelining in the processor under consideration. However, it should be noted that the model contains a switch which indicates the presence of consecutive *load* instructions, as it is an architecture specific feature. Thus, consecutive *load* instructions will be considered in the estimation only when this switch is enabled.

The proposed methodology accumulates the estimation of all categories of instructions present in the LLVM instruction profile of the application in order to estimate the performance of the full application. The execution time, in terms of clock cycles, of each basic block is computed using this information as given in Eq. 3.4. The total number of clock cycles for execution is computed using Eq. 3.5. The framework uses Eq. 3.6 to compute the estimated execution time of the full application.

$$bb_i = \sum_{j=1}^m c_j \quad (3.4)$$

$$N_{cc} = \sum_{i=1}^n (bb_i * f_{bb_i}) \quad (3.5)$$

$$T_{ex} = N_{cc} * \frac{1}{f} \quad (3.6)$$

where m = Number of instructions in basic block i ,
 n = Number of basic blocks,
 f = Frequency of the processor,
 c_j = Clock cycles to execute instruction j
 bb_i = Estimated clock cycles to execute basic block i ,
 f_{bb_i} = Execution frequency of basic block i ,
 N_{cc} = Total estimated clock cycles for execution,
 T_{ex} = Estimated execution time of the application

3.3 Performance Evaluation

This section presents the results of the proposed methodology for performance estimation of soft core processors and compare them with actual values obtained through synthesis and implementation on the selected FPGA platforms.

Two commercial soft core processors, Altera's *Nios II Gen2 fast* and Xilinx's *MicroBlaze* are used for evaluation purposes. These 2 processors are selected as they are commonly used processors with the most advanced micro-architectural features [29] [171]. Altera provides 2 variants of its *Nios II Gen2* processor, *economy* and *fast*, both of which support the same instruction set architecture but with different micro-architectural features. Hence, the two processors offer heterogeneity in terms of performance. The *fast* core, hereafter referred to as *Nios* in this chapter for brevity, is used in experiments. Also, the maximum possible cache size is used when generating both processors to avoid cache related performance variations. Further, both processors are run at 100 MHz and hence use that value in Eq. 3.6 to obtain the total execution time.

Table 3.2 Configurations in Experimental Setup

Feature	MicroBlaze	Nios
Device	Kintex 7	Cyclone V
FPGA chip	XC7K325T-2FFG900C	5CGXFC7F23C8
Speed grade	-2	8
Tool chain	Vivado v2015.3	Quartus v15.0

In order to ensure a fair comparison, comparable FPGA platforms, *Kintex 7* and *Cyclone V* from Xilinx and Altera respectively, have been selected to implement and evaluate the processors. Both FPGA devices are based on 28nm process technology and are mid-range devices from the respective vendors [172], [173]. Configurations of the FPGA devices used during the experimental setup for the two processors are shown in Table 3.2. In addition, default optimization flags have been used during the compilation step to enable a consistent comparison across different processors and configurations. Also, most modern FPGAs contain relatively large amounts of on-chip BRAM which enables them to retain significantly large portions of data within chip boundaries. Hence, the impact of external off-chip memory has not been considered in this work. All experiments were carried out on a 6-core virtual machine with 8GB RAM, running OpenSUSE 13.2 on an Intel Xeon E5-1650V2 CPU host at 3.5 GHz.

Applications from the popular CHStone [174] benchmark suite were used to verify the proposed techniques. CHStone benchmarks contain applications from different domains such as arithmetic, media processing, security etc., which enables to evaluate the proposed techniques on different domains. Furthermore, existing work such as [41] has used CHStone benchmarks for performance estimation of processors, similar to the work proposed. In addition, in their work on soft core processor customization, Prakash et al. [94] also used applications from MiBench and Mediabench benchmark suites. It is noteworthy that several applications in the CHStone benchmark suite are directly ported from MiBench and Mediabench suites [174].

Table 3.3 Configurable Functional Units

MicroBlaze	Nios
Int. Divider	Int. Divider
64-bit Int. Multiplier	64-bit Int. Multiplier
32-bit Int. Multiplier	LE based 32-bit Int. Multiplier 3*16 Multiplier based 32-bit Int. Multiplier 1*32 Multiplier based 32-bit Int. Multiplier
Barrel Shifter	Pipelined Shifter Non-pipelined Shifter
Basic Floating Point Unit	Floating Point Unit
Extended Floating Point Unit	

Table 3.4 Equivalent LE Calculation

Device	Total ALM/Slice LUT	Total LE	Equivalent LE Factorization Value
			$(\frac{TotalLE}{TotalALM/SliceLUT})$
Cyclone 5	56480	150,000	2.65
Kintex 7	203800	326,080	1.6

Various CFUs available in these soft core processors, for example, shifter, integer divider, integer multiplier (both 32-bit and 64-bit), etc. are considered. *Nios* has a total of 8 CFUs while *MicroBlaze* provides 6 such CFUs as shown in Table 3.3. Since, Xilinx and Altera report the area consumption of their platforms in different units of measurement, the techniques proposed in [175] are used to obtain the equivalent *logic elements* (LE). This enabled to analyze the results of the proposed approach with variation in area for both processors, under a common unit of measurement. Table 3.4 presents the method for equivalent LE factor calculation.

11 applications from the CHStone benchmark suite were executed on the *Nios* processor to obtain the actual runtime of these applications. These results serve as a baseline for comparison for this processor. Similarly, these applications were also executed on the *MicroBlaze* processor to obtain their baseline results (actual runtime) on the processor. However, only 8 applications could be successfully executed on

MicroBlaze due to various runtime errors. Since, debugging these applications is out of the scope of the proposed techniques, the estimation results are compared with actual results for 11 applications for *Nios* and 8 applications for *MicroBlaze*. It should be noted that this is not a limitation of the proposed framework, since the application performance can be estimated without the need for running the application on the target processor.

Also, redundant and infeasible configurations from the processor are removed and the design space is manually pruned such that repetitive configurations are removed to a total of 36 and 20 configurations for *MicroBlaze* and *Nios*, respectively. Even though, this process is done manually at this stage, it has been automated and explained in detail in chapter 7. The estimated application execution time results of the benchmark applications for this pruned design space along with the actual results are presented in Figure 3.4. Here each data point represents performance of a processor configuration (horizontal axis indicates the area consumption of the configuration). Figures 3.4a-3.4h show the 8 applications that were executed on both processors, while Figures 3.4i-3.4k show the remaining applications that could only be run on the *Nios* processor. It can be clearly observed that the estimated execution time values for all configurations is extremely close to the actual results obtained by executing the applications on the processor. Table 3.5 shows the estimation errors for these applications, averaged across all configurations. The average estimation errors for *MicroBlaze* and *Nios* are **5.70%** and **4.75%** respectively.

One important factor that is noteworthy is the sensitivity of the accuracy of the results proposed in this work to different compilers and their versions. Different compilers can generate different output instructions based on the backend architecture to which they are targeted. Considering this, it was specifically chosen to use the LLVM compiler to generate the IR, since LLVM assumes a RISC based architecture [168] backend, similar to the soft core processor architectures that are also mostly based on RISC, [80] [171]. This ensures to a large extent that the output produced by LLVM matches closely to the executables produced by the compiler for the target processor.

(a) AES

(b) BLOWFISH

(c) DFADD

(d) DFDIV

(e) DFMUL

(f) DFSIN

(g) GSM

(h) MOTION

(i) ADPCM

(j) JPEG

(k) MIPS

Figure 3.4 Distribution of Estimation Results Across the Design Space

Table 3.5 Estimation Errors (%)

Application	MicroBlaze	Nios
AES	4.39	4.50
BLOWFISH	9.59	4.09
DFADD	3.92	2.75
DFDIV	4.79	3.67
DFMUL	5.01	1.84
DFSIN	9.92	9.19
GSM	2.46	3.38
MOTION	5.54	4.75
ADPCM	-	9.82
JPEG	-	6.24
MIPS	-	2.02
Average	5.70	4.75

With respect to the compiler version, the results presented in this work are based on LLVM v3.3. However, it was also tested with LLVM v3.0 and obtained similar results. The only difference between the outputs of these two versions of LLVM were observed in basic block and instruction names, which the scripts are able to adapt to while classifying the LLVM instructions prior to the processor characterization step.

Next, the savings in terms of runtime from the proposed technique when compared to an exact method based on synthesis has been assessed. In a synthesis-based method, a designer first needs to synthesize the processor for all configurations identified in the pruned design space. Next, the application is executed on these synthesized processors to obtain the actual runtime on the corresponding configuration. On the other hand, the proposed methodology relies entirely on estimation based techniques to customize and recommend a suitable processor, along with its best configuration, without a need for synthesizing the processor. Hence, the time taken to synthesize the processor for all configurations in the pruned design space was measured using the respective tools of the processors. *MicroBlaze* requires 9 hours and 37 minutes to compile its 36 configurations and *Nios* requires 3 hours and 20 minutes to compile

Table 3.6 Comparison of Runtime

Application	MicroBlaze		Nios	
	Actual	Estimation	Actual	Estimation
	hr:min:sec	hr:min:sec	hr:min:sec	hr:min:sec
AES	9:37:00	0:25:49	3:20:00	0:14:51
BLOWFISH	9:37:01	0:08:48	3:20:00	0:04:56
DFADD	9:37:00	0:04:36	3:20:00	0:02:36
DFDIV	9:37:00	0:04:24	3:20:00	0:02:28
DFMUL	9:37:00	0:03:24	3:20:00	0:01:52
DFSIN	9:37:00	0:10:38	3:20:01	0:06:14
GSM	9:37:00	0:07:37	3:20:00	0:04:19
MOTION	9:37:00	0:04:24	3:20:00	0:02:28
ADPCM	-	-	3:20:00	0:06:55
JPEG	-	-	3:20:03	0:51:29
MIPS	-	-	3:20:00	0:02:08

20 configurations. Next, the application run time was measured by running on the synthesized processor. Since, the application runtime is significantly small, the total time (synthesis time and application execution time) is dominated by the synthesis time. Columns 2 and 4 in Table 3.6 show this total runtime for the *MicroBlaze* and *Nios* processors respectively. The total time for estimation for all configurations is obtained from the proposed methodology. Columns 3 and 5 in Table 3.6 show the total time for the proposed methodology for the *MicroBlaze* and *Nios* processors, respectively. Thus, it can be clearly observed that the proposed method runs significantly faster than the synthesis-based approach, while producing low estimation errors.

3.4 Summary

This chapter proposes a rapid technique for performance estimation of applications on soft core processors. The proposed methodology relies on the target-independent IR of the LLVM compiler, without necessitating application execution on the target processor

or instruction set simulators, thereby making it applicable to other processors and underlying FPGA architectures. The technique takes into account both data hazards and control hazards within a processor in order to obtain high estimation accuracy. The proposed methodology is capable of supporting different soft processor configurations, as it can estimate performance of CFU instructions, either with or without the CFU hardware. The approach, based on target independent LLVM IR, makes it scalable to the large number of configuration options available in modern soft core processors. The estimation process is fully automated to estimate the performance on a soft core processor within a few minutes.

The proposed technique has been tested using applications from the CHStone benchmark suite on two commercial soft core processors, Xilinx MicroBlaze and Altera Nios. The testing is carried out considering all the configurations (36 in MicroBlaze and 20 in Nios) available for both processors. The tests show an error of only 5% averaged across the full design space.

Although, the proposed methodology can be effectively deployed for performance estimation of applications on soft core processors, it is not capable in its present form to support hard core processors found in modern FPGA platforms. Hence, the proposed technique is further extended in the next chapter to incorporate the additional features that must be supported to facilitate the performance estimation for hard core processors.

Chapter 4

Performance Estimation of Hard Core Processors

4.1 Introduction

With a variety of FPGAs in the market there are several choices of hard core processors used in FPGA-based SoCs, varying across vendors or even from a single vendor that provides different hard cores on different FPGA platforms. Hence, the task of selecting the best processor that provides optimal performance under user constraints for an application is a daunting task for the designer as executing the application on the multitude of processors is practically infeasible. Alternatively, a designer could rely on simulators to estimate performance. However, simulators are typically slow, complex and require specialized knowledge. Hence, work has been focused on estimating the performance on hard core processors.

However, estimating the application performance on an embedded hard core processor is itself challenging due to control, data and structural hazards. The advent of superscalar architectures and out-of-order processing further exacerbates this problem. Existing work in this area has mostly been focused on single-issue, in-order processors such as *PowerPC 405* or need significant effort for every new target processor. Hence,

this chapter proposes a methodology for rapid performance estimation of applications on dual-issue, out-of-order, superscalar embedded hard core processors that relies solely on estimation based techniques. The proposed methodology further extends the methods proposed in chapter 3 to support hard core processors performing dual-issue out-of-order processing. Further, it can be extended to other similar processor architectures. Also, the methods can be used along with existing estimation tools for FPGAs, e.g. [176], to enable rapid design space exploration on FPGA-based SoCs at an early design stage during the hardware-software co-design step.

The rest of the chapter is organized as follows. Section 4.2 discusses the proposed methodology followed by the results in section 4.3. Section 4.4 proposes and describes a unified framework for performance estimation of hard and soft core processors. Conclusions are presented in section 4.5.

Portions of the work discussed in this chapter have been published in [177].

4.2 Methodology

This section discusses the proposed methodology, shown in Figure 4.1, to estimate application performance on dual-issue out-of-order superscalar processors. The technique is based on the application's IR obtained from the LLVM compiler. The clock cycle count of each instruction, hereafter referred to as execution count, is estimated considering data hazards, control hazards and structural hazards. The execution count of each instruction is then accumulated to obtain the full application execution time. The method constitutes of 4 phases, Processor Characterization, Application Profiling, Instruction Analysis and Performance Estimation.

Figure 4.1 Framework for Hard Core Processor Performance Estimation

4.2.1 Processor Characterization

This is a one-time semi-automated phase for each processor. Similar to chapter 3, this phase maps the ISA of the target processor to the LLVM ISA used in the IR. These LLVM instructions are classified into 4 categories for clarity:

- Data Processing Instructions: Instructions corresponding to ALU operations
- Data Transfer Instructions: Instructions corresponding to loading/storing data between memory and registers
- Flow Control Instructions: Instructions which determine the value of the program counter
- Remaining Instructions: Instructions which could not be mapped to any category mentioned above

However, the process of mapping instructions under each category is non-trivial. The following types of mapping between processor ISA and LLVM ISA have been identified:

- One-to-one mapped instructions: One processor instruction maps to one instruction in LLVM ISA
- One-to-many mapped instructions: One processor instruction maps to many instructions in LLVM ISA
- Many-to-many mapped instructions: Many processor instructions map to many instructions in LLVM ISA
- Other instructions: Type of mapping which does not belong to the categories above. For example, in the case of an ARM Cortex-A9, any number of subsequent *alloca* instructions in the LLVM ISA maps to a single sequence of *push*, *move*, *sub*, *bic* in the processor ISA

Table 4.1 Classification of LLVM Instructions

Instruction Category	Processor Instruction	LLVM Instruction
Data Processing Instructions	add	add
	mls	[mul,sub]
	[bfc,movw,movt]	getelementptr
	cmp	icmp
	[cmp,movne]	icmp
	[bic,pop]	[srem,ret]
	[smmul,mov]	[sext,mul]
	[push,mov,sub,bic]	alloca x M
Data Transfer Instructions	...	...
	ldr	load
	ldrb	load
	str	store
	stm	store
Flow Control Instructions	...	...
	b	br
	bl	call
	[b,blt]	br
	[beq,b]	br
Remaining Instructions	...	...
	vadd	fadd
	vcvt	sitofp
	vmla	[fmul,fadd]
	vfma	[fmul,fadd]
Remaining Instructions	...	...

1	ISA Mapping File: ARM Cortex-A9		
2			
3	***** Data Processing Instructions *****		
4			
5	[Instruction]	[Type]	[Clock Cycle Count]
6	add	i32	1
7	add	i64	1
8	mul	i32	3
9	mul	i64	4
10	icmp	n/a	1
11	alloca	i32	1
12	alloca	i64	2
13	sdiv	i32	4
14	-----	----	--
15			
16	***** Data Transfer Instructions *****		
17			
18	[Instruction]	[Type]	[Clock Cycle Count]
19	load	i32	1
20	load	i64	2
21	store	i32	1
22	store	i64	2
23	-----	----	--
24			
25	***** Flow Control Instructions *****		
26			
27	[Instruction]	[Type]	[Clock Cycle Count]
28	br	n/a	1
29	br (mis-predicted)	n/a	11
30	call	n/a	1
31	-----	----	--
32			
33	***** Remaining Instructions *****		
34	sitofp	float	4
35	fptosi	float	4
36	-----	----	--
37			

Figure 4.2 Sample ISA Mapping File

An example instruction mapping under the classified categories for the case of the ARM Cortex-A9 processor [178] is shown in Table 4.1, where M is a positive integer. The execution count of each instruction is obtained from data sheets or by running micro-benchmarks. The LLVM reference manual is used to create micro-benchmarks [179]. Figure 4.2 shows a sample output ISA mapping file.

4.2.2 Application Profiling

In this phase, the application is first profiled using the LLVM compiler to obtain an IR of the application. The profiling information is used to extract basic blocks, instructions in each basic block and execution frequency of basic blocks in the application. This information is used during scheduling in phase 4.

4.2.3 Instruction Analysis

The IR and other profiling information from section 4.2.2 have been used to identify the pipeline hazards between instructions and to annotate each instruction with this information. Also, multi-cycle instructions are identified as this is a special case of processor pipeline behavior. Further, each instruction is annotated with the execution count obtained in section 4.2.1.

As mentioned in section 4.1, modern hard core processors incorporate sophisticated execution mechanisms and advanced micro-architectural features, which makes the process of estimating performance of an application on a given processor significantly challenging. Thus, analysing the inter-dependencies between the instructions is of utmost importance to improve the accuracy of the estimation process. Therefore, the proposed methodology analyzes the instruction profile by incorporating pipeline hazard analysis and multi-cycle analysis.

Pipeline Hazard Analysis

A modern processor typically integrates deep pipelines within the processor execution path. However, due to the interaction of instructions with each other leading to data, control and structural hazards, the execution count of a given application varies significantly in each scenario. These hazards affecting the processor pipeline could result in flushing or stalling of the processor pipeline. With deep pipelines (for an example the ARM Cortex-A9 processor pipeline has 8-11 stages) the number of clock cycles required to re-fill the pipeline would incur a significant penalty adding to the execution count of the application [178]. Thus, identifying the aforementioned hazards from the IR level instruction profile is a critical step in the estimation process. Analytical techniques are proposed in order to identify pipeline hazards in the processor. The subsequent sections explain these techniques.

Data Hazard: A data hazard is created whenever there is a dependency between instructions and the overlap caused by pipelining of spatially localized instructions would change the order of access to an operand [180]. The proposed methodology analyses data hazards using the dependency analysis technique.

Dependency analysis is the process of analysing the instruction profile to identify instructions which are dependent on the data produced by the previous instructions or subsequent instructions which have the same destination address [180] [178]. A single-issue processor would only execute a single instruction at any given time. Also, an in-order processor would only consider dependencies with the subsequent instruction, and hence does not require extensive dependency analysis. However, an out-of-order processor requires an extensive dependency analysis within the instruction window. This is further exacerbated by the dual-issue feature compared to a single-issue processor. Hence, since the proposed model is focused on dual-issue, out-of-order processors, the dependencies among instructions are analyzed in order to identify suitable candidate instructions that could be processed in parallel and also to identify

their order of execution. Instructions are annotated with identified dependencies and this information is later used in the scheduling phase.

For example, load-use dependency is a common data dependency occurring in processors. Load-use dependencies occur when the data to be loaded is used in subsequent instructions prior to the correct data being written to the registers. The proposed model analyses such dependencies and compensates the execution count in the scheduling phase.

Control Hazard: Control hazards occur due to the instructions that alter the control flow of a program by modifying the *program counter*. Therefore, instructions such as conditional branches are analyzed in order to calculate the additional latency incurred due to branch mis-predictions.

Branches could be either conditional or unconditional. Unconditional branches do not stall or flush the pipeline. In the case of conditional branches, the execution count depends on whether or not the branch was correctly predicted. Correctly predicted branches behave similar to unconditional branches. However, mis-predicted branches cause pipeline flushing. Thus, it is important to identify branch mis-predictions. In chapter 3, the technique of Branch Entropy is used to estimate the performance of conditional branch instructions. However, as the branch prediction scheme used in hard cores is different, the branch prediction accuracy for the relevant processor is used to compensate for branch mis-prediction. This value is obtained from the datasheet of the relevant processor [181]. An approximation is made by multiplying the frequency of the basic block by the accuracy of prediction to obtain the number of times an instruction is predicted correctly and the remaining number is considered as the number of times it is mis-predicted.

Structural Hazard: Structural hazards occur due to the limited resources in the processor. For example, due to instruction-level parallelism there could be multiple

instructions requesting the service of a common resource. In such a scenario, the processor pipeline will be stalled while the hazard is resolved.

The proposed method analyses load/store instructions to identify structural hazards. Load/store analysis identifies load and store instructions in the instruction profile as some processors could have limitations in dual issuing load/store instructions [178]. The proposed method analyses the load/store instructions and annotates the information into the instruction trace. The scheduler in the subsequent phase analyses the annotated information in order to compensate, whenever necessary, during the scheduling process.

Multi-cycle Instructions

Multi-cycle instructions may or may not be handled in parallel, depending on the processor architecture [178]. For example, for a processor which does not support parallel handling of multi-cycle instructions, another instruction can be only scheduled at the last clock cycle. Thus, the models identify these instructions, and annotate them for identification by the scheduler.

4.2.4 Performance Estimation

In this phase, the annotated instruction trace from section 4.2.3 is used to model the schedule of instructions. Firstly, the model checks if the pipeline is free and if so, the instructions are scheduled. The instruction schedule is later used to estimate the performance of the application.

Scheduling and Execution Cycle Estimation

Scheduling is performed on each basic block obtained from the application profiling phase. Algorithm 1 is used for scheduling the application. Here, *PQ* stands for priority

Algorithm 1 Scheduling Algorithm

Input: Annotated Instruction Trace**Output:** Instruction Schedule of Basic Blocks

```

1: loop:
2: if initial or last issue is a dual issue then
3:   if PQ is non-empty then
4:     issue 1st instruction of PQ;
5:     stall other pipeline for ( $t_i-1$ );
6:     issued pipeline = busy;
7:     other pipeline = non-busy;
8:     dual-issue  $\leftarrow 0$ ;
9:     remove issued instruction from PQ;
10:   endif
11: else
12:   issue next instruction from instruction window;
13:   stall other pipeline for ( $t_i-1$ );
14:   issued pipeline = busy;
15:   other pipeline = non-busy;
16:   dual-issue  $\leftarrow 0$ ;
17: end if
18: endif
19: if issued instruction = load or store then
20:   load/store  $\leftarrow 1$ 
21: end if
22: endif
23: else if last issue is a single issue then
24:   if PQ is non-empty then
25:     while read PQ do
26:       check for dependencies;
27:       check previous issue for a load/store;
28:       check previous issue for multi-cycle;
29:       if current instruction is dual issuable then
30:         issue instruction;
31:         stall other pipeline for ( $t_i-1$ );
32:         issued pipeline = busy;
33:         other pipeline = non-busy;
34:         remove issued instruction from PQ;
35:         dual-issue  $\leftarrow 1$ ;

```

```

36:         end if
37:     endif
38: end while
39: done
40: end if
41: endif
42: if no instruction from PQ is issued then
43:     for i in next N instructions do
44:         check for dependencies;
45:         check previous issue for a load/store;
46:         check previous issue for a multi-cycle;
47:         if current instruction is dual issuable then
48:             issue instruction;
49:             stall other pipeline for ( $t_i-1$ );
50:             issued pipeline = busy;
51:             other pipeline = non-busy;
52:             remove issued instruction from PQ;
53:             dual-issue  $\leftarrow I$ ;
54:             break
55:         else
56:             add current instruction to PQ;
57:         end if
58:     endif
59: end for
60: done
61: if not issued from next N instructions then
62:     stall non-busy pipeline;
63: end if
64: endif
65: end if
66: endif
67: end if
68: endif
69: goto loop.

```

queue, N is the size of the instruction window and t_i is the execution count of the instruction under consideration.

Typically, out-of-order processing is performed within a specified instruction window which is dependent on the processor [182]. The proposed implementation utilizes a configurable instruction window size. During dual issue, if the subsequent instruction is not feasible for scheduling, the scheduler checks the next instruction and continues this process within the specified instruction window until an independent schedulable instruction is identified. The proposed model implements a priority queue to store the dependent instructions as obtained from the previous phase. This process is repeated until the end of the basic block.

The schedule provides the execution count of each basic block on the two pipelines. However, the higher execution count is selected as the execution count of the basic block, since execution requires a minimal time equivalent to the pipeline which consumes the larger amount of time.

The method proposed in section 4.2.3 is used to estimate the execution count of a basic block incorporating branch prediction. According to the definition of a basic block [179], a basic block could contain only one terminator instruction such as a branch instruction. Thus, Equation 4.1 is used to compute the total execution count of a basic block depending on the presence of a branch instruction. Here, the value of br represents the presence of a branch instruction.

$$N_{bb} = \begin{cases} t_p * f_{bb}, & br = 0 \\ t_p * f_{bb} * P_p + t_{mp} * f_{bb} * P_{mp}, & br = 1 \end{cases} \quad (4.1)$$

where f_{bb} = Execution frequency of the basic block,

P_p = Branch prediction probability,

P_{mp} = Branch mis-prediction probability,

$= 1 - P_p$,

t_p = Execution count of the mis-predicted basic block,

t_{mp} = Execution count of the mis-predicted basic block,

$= t_p + \text{branch mis-prediction penalty}$,

N_{bb} = Total execution count of the basic block

Next, the total execution count of all the basic blocks is accumulated to obtain the execution count of the full application using Equation 4.2. Equation 4.3 is used to obtain the estimated execution time of the application in seconds.

$$N_{cc} = \sum_{i=1}^n N_{bb_i} \quad (4.2)$$

$$T_{ex} = N_{cc} * \frac{1}{f} \quad (4.3)$$

where f = Frequency of the processor,

N_{bb_i} = Total execution count of basic block i ,

N_{cc} = Total execution count of the application,

T_{ex} = Estimated execution time of the application

4.2.5 Support for Other Hard Core Processors

FPGA-based CSoC, typically constitute the 32-bit ARM Cortex-A9 processor or the 64-bit ARM Cortex-A53 processor [183]. As such, the suitability of the proposed techniques for these two processors are discussed. The experiments and analysis in this chapter are performed considering the ARM Cortex-A9 processor. However, as mentioned the techniques can be adopted to different hard core processors. Thus, this section discusses incorporating the proposed techniques for the ARM Cortex-A53 processor.

While the ARM Cortex-A9, is an out-of-order processor, the ARM Cortex-A53 is an in-order processor. However, both processors are dual-issue processors and possess a superscalar pipeline. Thus, incorporating the techniques to the ARM Cortex A-53 processor will not require support for out-of-order processing. As such, the features for out-of-order processing provided in the scheduling algorithm in Algorithm 1 in this chapter will not be necessary in the case of the Cortex A-53 processor, rather simplifying the scheduling process. Thus, the scheduling algorithm can be modified only to incorporate the features for dual-issue of instructions and the superscalar pipeline. Therefore, the techniques proposed in this chapter can be extended to the ARM Cortex A-53 processor by incorporating the aforementioned modification to Algorithm 1.

4.3 Performance Evaluation

This section presents the results obtained by the proposed methodology for performance estimation of dual-issue, out-of-order, superscalar hard core processors and compare them with actual values obtained through implementation on the selected FPGA platform. The AVNET ZedBoard development kit [184] with Xilinx Zynq-7000 All Programmable SoC platform and the Xilinx Vivado 2015.3 development environment are used for the experiments. This FPGA-based SoC contains an ARM Cortex-A9

hard core processor, which is a dual-issue, out-of-order, superscalar processor running at $667MHz$ [184].

A superscalar processor is a CPU that implements instruction-level parallelism within a single processor. In contrast to a scalar processor that can execute at most a single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor, allowing for more throughput. Here, each execution unit is not a separate processor, but an execution resource within a single CPU. A dual-issue superscalar processor can execute 2 instructions at any given time. The proposed technique supports such superscalar dual-issue processing as discussed in section 4.2.

An out-of-order processor will execute instructions in an order of availability of data or operands instead of the original order of the instructions in the program. The purpose of this is to minimize idle time while data is retrieved for the next instruction in a program. The proposed techniques incorporate support for out-of-order processing, by executing instructions in order of availability. However, executing instructions in order of availability requires analysis of data dependencies between instructions. As such, the proposed technique takes into account the dependencies between instructions. The process adopted for this purpose is explained in detail in section 4.2.3.

In the ARM Cortex-A9 processor, level 1 instruction and data caches are $32kB$ in size and the level 2 cache is $1MB$ [185]. The instruction window is set to hold 50 instructions [182]. A branch prediction accuracy of 95% similar to [181] is used. Similarly, branch mis-prediction penalty is set to 11 clock cycles. Applications from the CHStone benchmark suite are executed in bare-metal mode on the processor [174]. As mentioned in chapter 3, CHStone benchmark suite contain applications from different domains such as arithmetic, media processing, security etc., which enables us to evaluate the proposed techniques on different domains. The *gprof* profiler of Vivado SDK is used to measure the runtimes of applications [186]. All experiments

Figure 4.3 Comparison of Results

were carried out on a 6-core virtual machine with 8GB RAM, running OpenSUSE 13.2 on an Intel Xeon E5-1650V2 CPU host at 3.5 GHz.

Figure 4.3 shows a comparison between the actual and estimated results. Both axes of the graph are represented in $\log_2$ scale for clarity. Here, blue circles represent actual results depicted in the horizontal axis while the orange triangles represent estimated results depicted in the vertical axis. The distance of the data points from the $(y = x)$ line shown in green indicates the accuracy of results. A lower deviation of the data points from the line depicts higher accuracy. The proposed technique has an average estimation error of **5.84%** averaged across 9 applications from the CHStone suite. The estimated results for the *MOTION* application initially exhibited a relatively higher error. Upon closer inspection of the actual results by accessing the processor performance counters, it was identified that this was due to the branch prediction accuracy for the application being lower than the value stipulated in the data sheets and not due to a short coming of the approach. Thus, for the *MOTION* application the branch prediction accuracy was adjusted to reflect the actual numbers.

Table 4.2 Estimation Errors and Runtime

Application	Estimation Error (%)	Runtime (min:sec)
AES	0.10	2:09
DFDIV	9.84	0:42
DFMUL	4.90	0:31
DFSIN	4.57	1:25
GSM	9.04	1:11
MOTION	1.14	0:39
ADPCM	7.03	1:20
JPEG	7.18	1:46
SHA	8.76	0:33
Average	5.84	1:08

The runtime of the proposed technique averaged across all these applications is **1** minute and **8** seconds. Table 4.2 shows the individual estimation errors and runtime for all applications. This shows the suitability of the proposed technique to rapidly estimate the performance of hard core processors with significant accuracy.

Further, it can be observed that compared to a state-of-art work by Aung et al. [75], it is not necessary to obtain the binary level information for each application, that necessitates a LLVM backend for the target processor. Instead, the proposed method relies only on the target-independent LLVM IR information and yet obtains rapid and accurate performance estimation for more complex and widely prevalent dual-issue, out-of-order processors.

4.4 A Unified Framework for Performance Estimation of Hard and Soft Core Processors

This chapter proposes a technique for performance estimation on hard core processors while in chapter 3 a technique for performance estimation on soft core processors was proposed. These techniques could be incorporated in a unified framework such as the

Figure 4.4 Unified Framework for Performance Estimation of Hard and Soft Core Processors

one shown in Figure 4.4, that could perform both soft core and hard core processor performance estimation. Such a framework facilitates a single interface to obtain performance estimation on both hard core and soft core processors.

The processor ISA, application C code and processor configuration will be input to the framework. The processor configuration will be provided only in the case of a soft core. Along with the processor configuration the user will indicate that the processor under consideration is a soft core. If these inputs are not provided the processor will be considered as a hard core. The key at the bottom of Figure 4.4 indicates whether the processes in the proposed technique are common to both soft and hard core processors or whether it is specific to either soft core or hard core. Depending on whether a soft core or a hard core processor is selected the relevant processes will be inferred.

The flow of the *Processor Characterization* and *Application Profiling* phase is identical to that proposed for the respective technique (soft core processor performance estimation or hard core processor performance estimation). The flow of the *Performance Estimation* phase is similar to that of the same phase in the technique proposed for soft core processor performance estimation. However, in the case of hard core processor performance estimation, the flow is a combination of *Instruction Analysis* (section 4.2.3) and *Performance Estimation* (section 4.2.4) phases. At the end of the proposed technique, the estimated performance of the application for the processor under consideration is provided.

4.5 Summary

This chapter proposes a rapid technique for performance estimation of dual-issue, out-of-order, superscalar hard core processors, that are typically integrated in modern FPGA-based SoCs. The technique proposed in chapter 3 was further extended by introducing models for accommodating performance-centric features such as dual-issue, out-of-order and superscalar. The proposed technique takes into account data hazards, control hazards and structural hazards within the dual-issue pipeline in order

to obtain high estimation accuracy. It is noteworthy that the proposed technique can be adapted to in-order processors, such as the ARM Cortex-A53, available in the latest FPGA platforms as discussed in section 4.2.5.

The proposed technique has been tested using applications from the CHStone benchmark suite for the ARM Cortex-A9 processor in a Xilinx Zynq SoC FPGA and has been shown to be accurate with an average estimation error of only 5.84%. The estimation accuracy compares well with that of soft core processor performance estimation in chapter 3. The average runtime of the approach is just over 1 minute.

A unified framework has also been proposed to support both hard and soft core processors by intelligently combining the technique proposed in chapter 3 for soft core processor performance estimation with the technique in this chapter for hard core processor performance estimation. This makes it possible to realize a common interface for estimating performance on hard core and soft core processors that are often made available on modern FPGA platforms. Finally, the proposed framework is scalable to support the future variations of both soft core and hard core processors.

Chapter 5

Hardware Area-Time Estimation

5.1 Introduction

Hardware area consumption and performance depend on the architecture of the FPGA device. The architectural features of a FPGA differs across vendors and devices. As such, existing techniques for hardware area-time estimation are limited to a single FPGA device or a few devices from a single vendor, even though it is an area of research that has been extensively studied. These techniques are discussed in chapter 2. Extending such techniques to support different devices requires modelling of the architecture of each FPGA device, entailing detailed analysis of the architectural features of the device. However, in order to make optimal use of hardware area-time estimation by providing the user the option to select between devices, it is important to support different devices.

This chapter proposes a technique to modify the existing open source LegUp tool [155] and subsequently use vendor synthesis tools to perform hardware area-time estimation across different FPGA vendors and devices, without the need for detailed architectural analysis and modelling. The main contributions of this chapter are,

- A technique to encapsulate basic blocks within a wrapper such that the area consumption of the basic blocks can be obtained using existing free versions of vendor tools
- A technique for generation of generic RTL code that can be synthesized on different vendor tools
- A technique for vendor specific RTL generation, in order to uphold performance optimization
- A unified methodology that can be applied for hardware area-time estimation across different FPGA vendors and device families

The rest of the chapter is organized as follows. The methodology is discussed in section 5.2, the results in section 5.3 and conclusion in section 5.4.

5.2 Methodology

This section presents the technique for hardware area-time estimation, which can be adopted across different FPGA vendors and devices. Initially, a hardware area-time estimation technique as depicted in Figure 5.1 is proposed and further improved by incorporating soft IP cores as shown in Figure 5.4. Later, the performance values obtained from the two techniques are compared in order to ensure that the incorporation of soft IP cores results in a performance gain. If it does not result in a performance gain, the process in Figure 5.1 is used for area-time extraction. This is explained in detail in the subsequent sections.

First, the technique depicted in Figure 5.1 will be explained. In this technique, initially the LLVM compiler is used to convert the application C code provided as input, to the LLVM IR. This LLVM IR is used by the subsequent Basic Block Wrapper as explained in the next section.

Figure 5.1 Hardware Area-Time Estimation - A

5.2.1 Basic Block Wrapper

The proposed technique uses an RTL code to obtain the hardware area (FPGA resource consumption) estimation using the synthesis flow of vendor tools. The RTL code is obtained by converting the LLVM IR, using ‘LLVM IR functions to RTL modules’ conversion of LegUp. In the LegUp LLVM IR to RTL conversion process, functions in the LLVM IR (functions in C code map to function in LLVM IR) are converted to modules in the Verilog RTL description. As such, it is possible to obtain area estimation at function level granularity by synthesizing the RTL code obtained by converting the LLVM IR using LegUp (as functions are mapped into RTL modules by LegUp). However, in contrast to the existing SoA, the method proposed in this work, relies on a basic block level granularity (The selection of basic block level granularity is justified later in chapters 6 and 8). Hence, it is necessary to obtain area estimates at basic block level granularity. However, during the synthesis flow, it does not provide area estimation of basic blocks of the C code. Thus, it is proposed to wrap basic blocks as functions which in turn map to RTL modules.

Since the RTL code is obtained by converting the LLVM IR using LegUp, it is necessary to provide a modified version of the LLVM IR such that LegUp can generate the RTL at the required granularity. To this end, a LLVM Transformation Pass is developed to wrap basic blocks inside functions. The pseudo code of the LLVM Transformation Pass is shown in Algorithm 2. The inputs to the pass consist of the LLVM IR and the list of basic blocks which need to be wrapped as functions. The basic steps of wrapping a basic block inside a function are given below.

1. Create a new function
2. Moving all instructions except terminator instructions to the new function
3. Resolve data dependencies using global memories

Initially, for each basic block j , a new function j' is created. Next, the pass moves all the instructions (in sequence) in the basic block j except the terminator instruction

Algorithm 2 Pseudocode of LLVM Transformation Pass

Input: LLVM IR, List of basic blocks to be wrapped and their respective functions

Output: Modified LLVM IR

```

1: GV : List of global variables
2: for Each Basic Block j do
3:   Create new function j'
4:   Add terminator instruction to function j'
5:   for Each Instruction i in Basic Block j do
6:     if i = terminator instruction of j then
7:       Continue
8:     else
9:       Move i to new function j' before terminator instruction
10:    end if
11:    for Each source operand n in i do
12:      if  $n \in k, k \neq j'$  then
13:        if  $n \notin GV$  then
14:          Create new global variable n_gv
15:          Add store instruction to j
16:          Add load instruction to j'
17:        end if
18:      end if
19:    end for
20:  end for
21:  for Each destination operand m in j' do
22:    if  $m \in j'', j'' \neq j'$  then
23:      Create new global variable m_gv
24:      Add store instruction to j'
25:      Add load instruction to j
26:    end if
27:  end for
28:  Add call instruction to j
29: end for

```

to the new function j' . Thereafter, a *call* instruction is added to the basic block j to invoke the new function j' . In the next stage, the pass resolves data dependencies. Here, data dependencies are defined as cases where a variable modified inside one basic block is used by another basic block. To resolve data dependencies, all the operands created or used outside the new function j' are converted to global variables. The *store* instruction represents preserving data on created global variables and *load* instruction represents retrieving the stored data to where it is needed. For source operands, *store* instructions are added inside the basic block j while *load* instructions are added in the new function j' . For destination operands, *store* instructions are added inside the new function j' , while *load* instructions are added to the other basic blocks j that use the destination operand.

The output of this pass is a modified version of the LLVM IR, where the input basic blocks are wrapped inside functions. However, this process incurs an overhead due to the addition of the mentioned *load* and *store* instructions. Later, an analytical technique is proposed to account for this overhead.

Thereafter, the modified LLVM IR obtained from the transformation pass is provided as input to LegUp to perform IR to RTL conversion using the ‘LLVM IR functions to RTL modules’ conversion. Thus, the RTL code obtained from LegUp will contain Verilog RTL modules for each of the new basic blocks, which were transformed into functions in the previous step. However, it was observed that the RTL code generated by LegUp is not generic as it contains code specific to Altera FPGAs (This is due to the fact that LegUp offers full support only for Altera FPGA devices). Hence, the RTL code cannot be synthesized on other vendor tools to obtain hardware area-time estimates. Therefore, a technique to generate a RTL code that can support different FPGA vendors has been proposed. This technique is explained in detail in sections 5.2.2 and 5.2.6.

5.2.2 RTL Conversion

Initially during this stage, the LLVM IR modified in the previous stage is converted to RTL code using LegUp. As previously mentioned in section 5.2.1, the RTL code provided by LegUp is specific to Altera FPGAs. In order to use this RTL code for area estimation in FPGAs belonging to other vendors, the code has to be modified either as a generic one or as one specific to the respective vendor under consideration. Thus, if the specified FPGA vendor is not Altera, the code is modified as a generic RTL representation, which can be used for any FPGA vendor. This section presents the technique used to modify the RTL code generated by LegUp as a generic RTL code such that it could be synthesized on FPGA devices from other vendors.

Analysis of the LLVM IR to RTL conversion process in the LegUp tool reveals that the process uses soft IP cores proprietary to Altera along with generic RTL code. It is the use of the proprietary soft IPs that make the code viable only for Altera devices. Besides the soft IPs the remaining part of the code is in generic RTL (Verilog) representation, which can be synthesized on any FPGA vendor tool. Thus, replacement of the proprietary soft IP cores with generic code will make the RTL code suitable for synthesis on other vendor tools.

The documentation of LegUp mentions that changing the Altera IP cores used for division and local memory will make the code suitable for other (e.g. Xilinx) FPGA devices [187]. However, analysis of the source code of LegUp and the LLVM IR to RTL conversion process revealed that it also uses floating point IP cores which are proprietary to Altera. Suitable replacement of the functionality of these components in the RTL code generated by LegUp will provide the ability to synthesize the code on other vendor tools. Thus, the Altera soft IP cores are replaced with generic RTL code. The replacement of these soft IP cores is discussed below.

Divider: LegUp uses Altera soft IP cores for signed and unsigned division in its LLVM IR to RTL conversion process. A modified version of a generic pipelined RTL

Figure 5.2 Block Diagram of Generic Divider

(Verilog) divider is used to replace the divider modules in the LegUp RTL code [188]. The code is modified such that the functionality is identical to that of the corresponding Altera IP core. It is observed that the original code does not provide the remainder as an output and only supports unsigned division. Thus, the code is modified to incorporate features to produce the remainder of division as an output and to perform both signed and unsigned division. The block diagram of the generic divider is shown in Figure 5.2.

Figure 5.3 Block Diagram of Generic BRAM

Block RAM: LegUp uses Altera soft IP cores for block RAM (local memory) in its LLVM IR to RTL conversion process. A generic RTL (Verilog) RAM model is used to replace the block RAM modules in the LegUp RTL code. Initially, this generic RAM model is obtained by coding the simple RAM model (coded in VHDL) [189] in Verilog. The RTL code in this model implements memory as a 2-dimensional array. The size of the array will be automatically configured depending on the size of the memory. This model is modified in order to support *true dual port* functionality as that is supported by the Altera soft IP core. The block diagram of the generic block RAM (BRAM) is shown in Figure 5.3. The BRAM constitutes a 2-D array which is

used for data storage and a controller used to read and write addresses. Further, since the BRAM supports true dual port functionality, both ports are used for read as well as write operations.

Altera Soft IP	Generic RTL
Adder	Adder [190]
Adder 64	Double Adder [190]
Divider	Divider [190]
Divider 64	Double Divider [190]
Float to Int32	Float to Int [190]
Float to Int64	Double to Long [190]
Multiplication	Multiplier [190]
Multiplication 64	Double Multiplier [190]
Int to Float32	Int to Float [190]
Int to Float64	Long to Double [190]
Subtract	Adder [190] (Modified)
Subtract 64	Double Adder [190] (Modified)
Compare 32	Altera Compare 32 + Adder [190] (Modified for Sub.)
Compare 64	Altera Compare 64 + Adder 64 [190] (Modified for Sub.)
Truncate	Altera Truncate + Adder [190] (Modified for Sub.)
Extend 64	Altera Extend 64 + Adder [190] (Modified for Sub & Add.)

Table 5.1 Generic RTL for Floating Point Cores

Soft IP Cores for Floating Point Operations: LegUp uses Altera proprietary soft IP cores for floating point operations, as revealed in the analysis of the LegUp source code. Thus, the replacement technique of the Altera floating point soft IP cores with generic RTL code is discussed here. Some of the soft IP cores for floating point operations are directly replaced using the generic Verilog RTL code provided for IEEE compliant floating point operations in [190]. It does not provide compatible generic code for *truncate*, *compare* and *extend* soft IP cores. However, these soft IP cores can be supported with a combination of other floating point cores along with generic code. Thus, for these soft IP cores, the Verilog code of the Altera cores (that is used in LegUp) is modified such that the Altera proprietary components are replaced by generic code. The details of the code that has been replaced are provided in Table 5.1. Column 1 in Table 5.1 depicts the Altera soft IP core incorporated in the LegUp generated RTL

code. Column 2 shows the corresponding generic RTL code obtained from [190] that has been used to replace the Altera soft IP core. However, some generic code obtained from [190] have been modified to incorporate the necessary functionality. These have been specified as '*Modified*' in Table 5.1. This way, Subtract and Subtract 64 soft IP cores have been replaced by modifying the generic RTL Adder in [190]. In addition, some of the Altera soft IP cores have been replaced by a combination of code in [190]. For example, in the Compare 32, Compare 64, Truncate and Extend 64 soft IP cores, the Altera specific components within the code have been replaced without altering the generic RTL code in the Altera IP core. The modified version of the Adder used for Subtract and Subtract 64 have been combined with the generic part of the Altera IP core for Compare 32, Compare 64 and Truncate, while the Adder is modified to support both Adder and Subtract in Extend 64. All other cores have been replaced with the generic RTL code in [190] with minimal modifications.

Micro-benchmarks are used to ensure that the replaced codes provide functionally accurate results in comparison to the Altera specific soft IP cores. Further, it is also important to ensure that timing synchronization is maintained to ensure functional correctness of the application. For example, if the performance (number of clock cycles required for execution) of the newly incorporated code is not identical to those in the original LegUp converted RTL code, the application schedule is modified. Similar to the case of testing the individual functionality of the new code, once this has been completed, micro-benchmarks are used to ensure that timing synchronization is maintained.

5.2.3 Synthesis on Vendor Tool

Typically, synthesis tools provide area estimates of the RTL code. Therefore, after generating the generic RTL code suitable for synthesis on the given FPGA vendor tool, the synthesis tool of the relevant FPGA vendor is inferred. The synthesis report generated by the tool is used to extract the area estimates in the subsequent stage. It

should also be noted that non-commercial versions of the synthesis tools are used for this purpose.

5.2.4 Area-Time Extraction

The hardware area is obtained in terms of LUT, DSP and register (REG) resources on the FPGA. This data is extracted from the synthesis report of the vendor tool. However, the process of wrapping basic blocks in the LLVM IR into functions in the LLVM IR presented in section 5.2.1 introduces an overhead due to the additional *load* and *store* instructions. This overhead is accounted for by deducting the area required for these instructions from the actual extracted area estimates for each basic block. In order to do this, the area consumption of *load* and *store* instructions in terms of LUT, DSP and REG resources have been experimentally obtained.

LegUp scheduling report provides the execution time in clock cycles at basic block level granularity for the RTL code. Hence, the process of wrapping basic blocks in the LLVM IR into functions in the LLVM IR presented in section 5.2.1 has no effect on the execution time estimates. As such, the execution time of the basic blocks are extracted from the LegUp scheduling report, except in cases where the newly incorporated code to replace the Altera specific soft IP cores change the original version of LegUp in terms of execution clock cycles. In such a scenario, the change when extracting execution clock cycles is compensated. The execution clock cycle count extracted using the LegUp scheduling report is for a single execution of each basic block. Thus, this value is multiplied by the frequency of the basic block, which could be obtained from the LLVM IR, in order to obtain the total execution cycles of the basic block.

At the end of the hardware area-time estimation process the area and time estimates at basic block level is provided. It is noteworthy that all the techniques discussed have been fully automated. Shell and TCL scripts have been used in the automation process. While, Shell scripts are used to control the overall flow, TCL is used to infer vendor specific tools.

The technique proposed above replaces the Altera specific soft IP cores in the RTL code generated by LegUp by generic RTL code. However, there can be other vendor specific IP cores which can be used to replace the Altera specific soft IP cores or even additional soft IP cores for Altera that may improve the performance of the proposed technique. Hence, the next section proposes a technique to incorporate proprietary soft IP cores.

5.2.5 Incorporation of Proprietary Soft IP Cores

For FPGA vendors other than Altera, the Altera proprietary soft IP cores are replaced with compatible generic code as discussed in section 5.2.2. Typically, proprietary soft IP cores are optimized for performance/area in the devices belonging to the respective FPGA vendor. Thus, a technique to identify possibilities of replacing generic code with proprietary soft IP cores as shown in Figure 5.4 has been adopted. In the case of Altera, additional soft IP cores other than the ones which have been incorporated by LegUp, are inserted while for other FPGA vendors all relevant IP cores are inserted. The soft IP cores which have been incorporated in this work are shown in Table 5.2. Even though, IP cores typically provide better performance than generic RTL, there could be cases where application-specific RTL performs better than the IP cores which are typically intended for a predefined set of inputs. In such cases, even though, the requirement for the core is identified, the RTL is not replaced with the IP core. Further, it is ensured that timing synchronization is maintained during this process using the same technique as in section 5.2.2.

Automatic replacement of RTL code with soft IP cores necessitates the identification of the requirement for the IP core. In the subsequent sections, this identification process is discussed. It is also important to note that the proposed technique is able to select the IP core and also configure the core to obtain optimal area/performance.

Figure 5.4 Hardware Area-Time Estimation - B

Type	IP core
Memory	Block RAM single port
	Block RAM dual port
	Shift Register
Floating Point	...
	Adder
	Divider
	Multiplier
	Subtract
	Accumulate
	...

Table 5.2 Examples of Incorporated Vendor Proprietary IP cores

Memory

RTL code generated by LegUp only contains local memory elements. Thus, only soft IP cores pertaining to local memory are considered. Local memory (Block RAM) is included in the LegUp RTL generation process as a Altera soft IP core. However, in the generic RTL code, the BRAM modules are replaced. Hence, the generic code segments are used for Block RAM to identify the requirement to insert Block RAM IPs. Shift Registers are used to replace the register stages inserted for synchronization as well as other register stages prevalent in the RTL code. Shift registers are identified by the shift operators and included for Altera devices as well (This is due to the fact that LegUp does not include the IP cores for shift operators).

Floating Point

Altera IP cores are replaced by generic RTL code for floating point operations for vendors other than Altera as discussed in section 5.2.2. Thus, these code segments are identified in order to replace them by the vendor specific IP cores such as the ones shown in Table 5.2

However, in this work, IP cores with high levels of complexity in identification are not considered. This is due to the fact that this process involves high possibility of error,

which will alter the functionality of the application. Even though, failure to identify the requirement to insert the IP core will not lead to errors, a wrong match would alter the functionality of the application. Correct functionality of the RTL is of paramount importance since this RTL is ultimately intended to be used for actual implementation. Also, the RTL code has to be synthesized without errors in order to obtain the area estimation. Thus, only IP cores with high accuracy of correct identification have been incorporated. Further, it is important to note that even though this work supports only Altera and Xilinx FPGA devices the technique can be adopted to support any FPGA vendor.

5.2.6 Ensuring Functional Correctness

In the proposed methodology, the RTL code is modified in 2 stages. Initially, the original RTL code generated by LegUp is modified to obtain a generic code. Thereafter, the RTL code is modified to include vendor proprietary soft IP cores. Thus, it is important to verify the modifications to the RTL code for functional correctness in order to ensure that the functionality of the application remains unaltered. Thus, the functionality of the new soft IP cores/code has to be identical to that of the proprietary soft IP cores used in the original RTL code provided by LegUp.

In order to ensure functional correctness, the individual functionality of the newly incorporated IP cores/compatible code are tested using micro-benchmarks. Further, it is also important to ensure that timing synchronization is maintained to ensure functional correctness of the application. If the number of clock cycles required for execution of the code of the newly incorporated IP cores/code is not identical to those in the original LegUp converted RTL code, the scheduling process is modified in the RTL code such that timing synchronization is maintained. Similar to the case of testing the individual functionality of the new IP core/compatible code, this process is tested using micro-benchmarks to ensure that timing synchronization is maintained.

Verification of Performance Gain due to Introduction of Soft IP Cores

As initially mentioned, the technique in Figure 5.1 is used to ensure that the incorporation of IP cores results in a gain in performance. Typically, IP cores are intended to improve performance. As such, hardware area-time estimation is performed separately for both the processes in Figure 5.1 (without incorporation of IP cores) and Figure 5.4 (with incorporation of IP cores). If the incorporation of IP cores as in Figure 5.4 results in an increase in performance in comparison to that without IP cores as in Figure 5.1, the area and time are extracted from the process in Figure 5.4, if not the process in Figure 5.1 is used for area-time extraction.

5.3 Performance Evaluation

This section explains the experimental setup followed by an analysis of results for hardware area-time estimation. All experiments are carried out on a virtual machine running Ubuntu on an Intel Xeon CPU host at 3.5 GHz with 8 GB RAM. Applications from the popular CHStone [191] benchmark suite are used to validate the proposed techniques. CHStone benchmarks contain applications from different domains such as image processing, encryption etc. Hence, it allows to validate the proposed techniques for applications from different domains. This section only presents the results obtained for the Xilinx Kintex 7 and Altera Cyclone V FPGA devices. However, the technique is tested for 5 device families commonly used in FPGA-based SoC applications, Altera Cyclone V, Cyclone II and Arria II and also Xilinx Kintex 7 and Artix 7. Even though, the full benchmark suite is used for experimentation, 2 applications are selected for analysis of results. Further, since an application contains a large number of basic blocks, for ease of representation only the results for the 25 most frequently executed blocks are depicted in each application. It is important to note that the proposed technique can be used either for all the basic blocks in the application or for any given subset of basic blocks.

Table 5.3 and Table 5.4 show the area-time estimation values obtained from the proposed approach for ADPCM and DFSIN applications respectively. The results are presented for Xilinx Kintex 7 and Altera Cyclone V FPGA devices. The area is shown in terms of LUT, REG, DSP and BRAM resources. The time value indicates the number of clock cycles the basic block requires to execute on hardware. It should be noted that the 25 most frequently executed basic blocks in each application are selected solely for analysis purposes. The techniques could be adopted to all or any selected subset of basic blocks in an application.

As can be clearly identified from Table 5.3 and Table 5.4, the area and time for the applications differ across the selected FPGA devices. The reason for the difference in the resource consumption is due to the size and architectural differences of the FPGA resources. Further, it can be observed that there are changes in the type of resources utilized across devices. For example, basic blocks 1 and 2 in ADPCM only use block RAM in the Xilinx Kintex 7 device, while in the case of Altera it uses only LUT and REG resources. It is also interesting to note that basic blocks 9 and 13 in ADPCM utilizes DSPs in the Xilinx Kintex 7 while it does not use DSPs in Altera Cyclone V. While, in the case of basic block 21 in the DFSIN application, DSPs are utilized in Altera Cyclone V device while no DSPs are utilized in the Xilinx Kintex 7. This could be attributed to the difference in architecture of the resources in the 2 devices. The difference in time for the same basic block across the 2 FPGAs is due to the incorporation of vendor proprietary IP cores.

Basic Block	Xilinx Kintex 7					Altera Cyclone V				
	LUT	REG	DSP	BRAM	Time	LUT	REG	DSP	BRAM	Time
1	0	0	0	1	3600	29	18	0	0	3600
2	0	0	0	1	3450	12	9	0	0	3450
3	69	161	0	0	3300	77	107	0	0	3300
4	5	67	0	0	4400	41	56	0	0	3300
5	12	134	0	0	3000	82	112	0	0	3000
6	82	384	6	0	3000	233	318	4	0	4000
7	0	0	0	1	2268	0	0	0	1	2268
8	10	134	0	0	1944	82	112	0	0	1944
9	144	402	4	0	1944	238	304	0	0	1944
10	70	12	6	0	3240	116	86	4	0	3240
11	0	0	0	1	1932	29	18	0	0	1932
12	0	0	0	1	608	12	10	0	0	608
13	140	274	4	0	1656	214	226	0	0	1656
14	10	134	0	0	1656	82	112	0	0	1656
15	0	0	0	1	1650	13	9	0	0	1650
16	0	0	0	1	1650	12	9	0	0	1650
17	0	0	0	1	1650	11	9	0	0	1650
18	83	318	6	0	1500	188	281	4	0	1500
19	74	318	0	0	1500	97	211	0	0	1500
20	6	67	0	0	1500	41	56	0	0	1500
21	6	67	0	0	1500	41	56	0	0	2000
22	101	349	6	0	2000	170	296	4	0	1500
23	5	67	0	0	1500	41	56	0	0	1500
24	0	0	0	1	1263	0	0	0	1	1263
25	29	101	3	0	2100	89	113	2	0	2100

Table 5.3 Area-Time Estimation Values for ADPCM

Basic Block	Xilinx Kintex 7					Altera Cyclone V				
	LUT	REG	DSP	BRAM	Time	LUT	REG	DSP	BRAM	Time
1	10	20	0	0	11080	49	30	0	0	11080
2	8	16	0	0	13296	29	24	0	0	11080
3	10	20	0	0	10080	41	30	0	0	8400
4	240	897	0	0	5540	896	949	0	0	5540
5	460	160	0	0	4180	173	208	0	0	4180
6	76	176	0	0	4180	84	200	0	0	4180
7	0	0	0	0	3344	0	0	0	0	5016
8	759	860	0	0	5016	444	908	0	0	3344
9	1052	1200	0	0	2508	133	1224	0	0	2508
10	0	0	0	2	4180	0	0	0	2	4180
11	8	12	0	0	2139	24	20	0	0	2139
12	642	2427	12	0	1782	2261	2508	96	0	1782
13	39	36	0	0	2255	47	36	0	0	2255
14	0	0	0	2	2255	0	0	0	0	2255
15	160	352	0	0	2255	208	384	0	0	2255
16	0	0	0	2	1520	0	0	0	2	1520
17	0	0	0	2	1520	0	0	0	2	1520
18	0	0	0	2	1520	0	0	0	2	1520
19	258	523	0	0	1520	232	554	0	0	1520
20	0	0	0	2	1510	0	0	0	2	1510
21	64	449	0	0	1510	152	472	32	0	1510
22	72	300	0	0	906	192	306	0	0	906
23	464	726	0	0	876	581	740	0	0	876
24	0	0	0	1	804	0	0	0	1	804
25	447	487	0	0	1072	474	502	0	0	1072

Table 5.4 Area-Time Estimation Values for DFSIN

Figure 5.5 Inputs and Output of Hardware-Software Partitioning

It is also important to note that the results of hardware area-time estimation is used as an input to the hardware-software partitioning technique as shown in Figure 5.5. The methodologies used for hardware-software partitioning are discussed in chapter 6.

5.4 Summary

This chapter proposed a technique for hardware area-time estimation of applications on FPGAs that can be applied across different FPGA vendors and device families. The application C code was converted to the target independent LLVM IR prior to wrapping the basic blocks as functions using a LLVM transformation pass. The LegUp tool's 'LLVM IR functions to RTL modules' conversion was then carried out to facilitate RTL synthesis using the Altera Quartus tool. The approach to rely on vendor tools for obtaining the final area estimation effectively guarantees accuracy of estimation.

In order to support FPGAs other than Altera, the Altera soft IP cores generated by LegUp were replaced as generic RTL components. This has made it possible to extend support to FPGAs from other vendors. Also, vendor-specific basic soft IP cores are incorporated for performance improvement.

The proposed technique relies on the free versions of the vendor tools and LegUp. Moreover, the proposed approach does not necessitate time consuming post synthesis steps such as Place & Route and Bit Stream Generation in order to obtain reasonably accurate area estimation measures.

Chapter 6

Data Dependency-aware Hardware-Software Partitioning

6.1 Introduction

Modern FPGAs are not only suited for accelerating critical parts of an application, but also for realizing an entire SoC, constituting processors, programmable logic, memory subsystems, etc. Hence, while the critical section of the applications can still be accelerated on the programmable logic, the rest of the application can be executed on the accompanying on-chip processors. An efficiently partitioned design effectively combines the flexibility of software with the performance acceleration of hardware.

The granularity of partitioning an application poses yet another challenge for the designer during the partitioning step. Compared to a fine-grained approach, a coarse-grained approach implements large sections of code where only a small fraction of the code may actually execute frequently enough to provide any meaningful acceleration in execution time, resulting in reduced return on the investment of the hardware area. On the other hand, a fine-grained approach provides much greater control on the accelerated code segments resulting in higher acceleration in execution time for the same hardware space, but leads to a much larger design space for the designer to select

the best application code segments for acceleration. In addition, the larger design space, results in increase in data communication between components, incurring higher data communication cost. Therefore, it is important to accurately model data communication costs in a fine-grained approach.

Thus, it can be observed that there is a necessity for hardware-software partitioning methodologies at fine granularity that provide optimal performance within stringent area constraints while taking into account data dependencies between components. This chapter proposes methodologies for rapid data dependency- and area constraint-aware hardware-software partitioning at the basic block level that can be applied to applications of varying size and complexity. The main contributions of this chapter are,

- A methodology for analysis of data communication cost between basic blocks and memory components
- A rapid technique to model the application hardware-software partitioning problem to an Oregon Trail inspired knapsack problem using the proposed system level model
- A scalable ILP-based heuristic formulation to select the most profitable hardware-software partitioning considering (i) data communication cost of basic blocks and memory components, (ii) Area constraints in terms of LUT, DSP and REG resources

The proposed methods compared against the state-of-the-art (SoA) [144], not only achieves better accuracy but also advances the SoA by incorporating multiple design constraints without compromising the runtime.

The rest of the chapter is organized as follows. The methodology is discussed in section 6.2, with the results in section 6.2.2 and conclusion in section 8.4.

Portions of the work discussed in this chapter have been published in [192].

6.2 Methodology

This section discusses the system level model and the details of the proposed methodology for hardware-software partitioning. As discussed in chapter 2, a fine grained partitioning approach at basic block level, provides the opportunity to maximize the utilization of FPGA area. Efficient utilization of FPGA area is essential in order to achieve the optimal performance in highly resource constrained designs. Thus, hardware-software partitioning is performed at basic block level.

6.2.1 System Level Model

Figure 6.1 shows the proposed system level model for application characterization, comprising of c computation units denoted by $\{C_1, C_2, C_3, \dots, C_c\}$ and m memory blocks denoted by $\{M_1, M_2, M_3, \dots, M_m\}$ interconnected via a data communication network. Here, C_i represents computation unit i and M_j represents memory block j . Also, the set of all computation units is denoted by Δ , and the set of all memory blocks by Γ as shown in Eq. 6.1 and Eq. 6.2 respectively.

Figure 6.1 System Level Model

$$\Delta = \{C_1, C_2, C_3, \dots, C_c\} \quad (6.1)$$

$$\Gamma = \{M_1, M_2, M_3, \dots, M_m\} \quad (6.2)$$

It should be noted that, in this work, an application is represented using the proposed model where basic blocks are mapped to computation units and statically assigned variables to memories. Further, each computation unit is characterized by a set of features, defined by (i) Software Time (ST): Given that the full application is executed in the processor space (SW), the execution time of a computation unit; (ii) Hardware Time (HT): Similar to ST, provided that the full application is executed in the FPGA space (HW), the execution time of a computation unit (iii) Area (L,D,R): resource consumption of a computation unit in terms of LUT, DSP, REG in HW; and (iv) Frequency (α): number of times a computation unit is executed during application execution. Thus, each computation unit (C_i) is represented by a tuple as shown in Eq. 6.3.

$$C_i = \{ST_i, HT_i, L_i, D_i, R_i, \alpha_i\} \quad (6.3)$$

Further, each computation unit (C_i) accesses the memory blocks (M_j) for data communication. Thus, M_{ij} represents the number of accesses to memory block M_j from computation unit C_i during application execution.

6.2.2 Hardware-Software Partitioning

During application characterization, the input C application is mapped to the system level model presented in subsection 6.2.1. At the same time, all features of computation units (C_i) and memory accesses (M_{ij}) are obtained using the techniques proposed in chapter 3 (soft core), 4 (hard core) and 5. The system level model and design constraints are used as the inputs in the *Oregon Trail inspired knapsack* model. Next, it is formulated into an *ILP-based* heuristic and solved using an ILP solver. Finally, a post optimization process of the obtained solution is performed to further enhance accuracy.

Application Characterization

The C application is first profiled using the LLVM profiler to obtain the most frequently executed basic blocks, henceforth referred to as ‘hot blocks,’. These hot blocks are considered as computation units while each assignment in the LLVM profile is considered as a memory block. Even though, the user has the flexibility to select the number of computation units for analysis, 25 hot blocks are selected to verify the techniques proposed in this chapter. However, it should be noted that the method proposed in this chapter is independent of the number of selected computation units.

Feature Extraction The LLVM profile is used to extract area and time features of computation units. The estimation based methodology proposed in chapter 3 is used to extract the ST, while HT and area extraction is performed using the techniques proposed in chapter 5.

Dependency Analysis The data flow information between the computation units and the memory blocks are extracted using a control data flow graph. The process of obtaining the data dependency cost between a single computation unit and a memory block is as follows. Let d_{uk} be data produced by instruction i_k in C_u . Also, assume that d_{uk} is consumed by β instructions of C_v , where $u \neq v$. Thus, the data dependency cost of C_v , when accessing d_{uk} is defined as β . Since, the execution frequency of C_v is known to be α_v (section 6.2.1), the total data dependency cost is $(\beta * \alpha_v)$. This procedure is repeated for all nodes of the control flow graph using an LLVM analysis pass. Based on the data obtained from this pass, the data flow graph (DFG) is developed, with nodes and edges representing computation units and dependency costs respectively. It should be noted that there can be multiple edges between two nodes of the DFG as two computation units may share multiple data between each other. Thus, each edge in the DFG is mapped to the corresponding M_{ij} values explained in section 6.2.1.

It should be noted that the extracted features and data dependencies are purely used as inputs. The methodology proposed for hardware-software partitioning in this chapter is independent of the tools/methods used for application characterization.

Design Partitioning

After obtaining the system level model, the partitioning algorithm strives to find the optimal set of computation units for hardware acceleration to minimize the overall application execution time within the design constraints. Formally, the problem is defined as “*given a finite set of computation units and the data transfer costs between computation units and memory blocks, find the optimal set of units and memories to be implemented as hardware accelerators such that the application execution time is minimized within area constraints*”.

Architecture Model In the proposed work, the analysis is based on the architecture model described in [115]. The model is summarized for clarity below.

1. Placement of memory blocks: When a computation unit is implemented in hardware, all memory blocks that are accessed by the computation unit are also placed alongside it in the FPGA BRAMs as un-cached local memories. This is shown in Eq. 6.4, where $C_i = HW$ means that computation unit i is implemented in hardware. This assumption is required to exploit potential benefits of hardware acceleration [193].

$$C_i = HW \Rightarrow M_j = HW \iff M_{ij} > 0, \quad (6.4)$$

$$\forall C_i \in \Delta, \forall M_j \in \Gamma$$

2. Data communication penalty (Ω): There will be a data communication overhead incurred by computation units implemented in software while accessing data stored in un-cached local memories (δ). In contrast, when computation units

and memory blocks are both in software the access time will be the cache access time (γ). Thus, there will be a data communication penalty as shown in Eq. 6.5. Further, it should be noted that the data communication penalty effectively encapsulates the speeds of communication interfaces. On the other hand, based on assumption (i), the penalty incurred by computation units in hardware to access the un-cached local memories is encapsulated in the HT, obtained during feature extraction.

$$\Omega = \begin{cases} \delta, & \text{if } M_{ij} > 0; \forall C_i = SW, \forall M_j = HW \\ \gamma, & \text{if } M_{ij} > 0; \forall C_i = SW, \forall M_j = SW \end{cases} \quad (6.5)$$

3. Static data allocation: The model assumes that all data will be initialized prior to the execution of the application. Thus, it should be noted that applications with dynamic data input are not considered in this chapter.
4. Atomicity of computation units: Since basic blocks are mapped as computation units, it is guaranteed that the control flow of the application will not deviate from one computation unit to another while the former computation unit is executed. However, data flow requires each computation unit to access the memories at appropriate times.
5. Resource sharing: It is assumed that none of the computation units share hardware resources in terms of LUT, DSP and REG. As discussed in [194], this assumption is based on the fact that resource sharing negates the objective (minimizing execution time) of the partitioning problem.

Based on the architecture model and assumptions, it is evident that even though certain computation units might be profitable candidates for hardware acceleration, due to data dependencies, they might have a negative impact on the overall execution time of the application. This effect is analogous to the 0-1 Oregon Trail knapsack problem presented in [195]. The 0-1 Oregon Trail knapsack problem discusses filling a knapsack with a set of items where the value of an item of type x depends on the

presence of another item of type y in the knapsack. Inspired by this similarity, the proposed HW-SW partitioning problem is mapped such that, in the partitioned design space, the execution time of a computation unit placed in SW ($T_{C_i,SW}$), depends on itself (C_i) and the penalty incurred due to presence of other computation units placed in HW ($\Delta \setminus C_i$) that have overlapping data dependencies with C_i . The proposed model is shown in Eq. 6.6, which shows that the execution time of a computation unit in SW is a function (f_i) of itself (C_i) and other computation units in HW ($\Delta \setminus C_i$). However, it should be noted that based on assumption (i), the execution time of a computation unit placed in HW ($T_{C_i,HW}$) is not affected by this model and is still equal to HT_i . This is shown in Eq. 6.7.

$$T_{C_i,SW} = f_i(C_i, \Delta \setminus C_i) \quad (6.6)$$

$$T_{C_i,HW} = HT_i \quad (6.7)$$

ILP-based Heuristic Formulation The next step in the framework is to propose a runtime efficient algorithm to solve the problem while providing near optimal results. It has been shown that ILP-based formulations can optimally solve the hardware-software partitioning problem [118]. However, the time complexity of ILP-based exact solutions is exponential, which in-turn reduces their usefulness for larger design spaces. To alleviate this problem, an ILP-based heuristic formulation as explained in detail in the subsequent sections is proposed.

The input to the ILP solver consists of the set of computation units (Δ), set of memory blocks (Γ) and their dependencies (M_{ij}). The design constraints are provided in terms of the LUT (ζ), DSP (ξ) and REG (λ) resources where $\zeta, \xi, \lambda \in \mathbb{Z}^+$. Next, a binary decision variable x_i , as shown in Eq. 6.8, is defined for each C_i which indicates

whether the computation unit is placed in hardware ($x_i = 1$) or software ($x_i = 0$).

$$x_i = \begin{cases} 1, & \text{if } C_i = HW, \\ 0, & \text{otherwise.} \end{cases} \quad \forall i = 1, 2, 3, \dots, c \quad (6.8)$$

The objective of the partitioning problem is to minimize application execution time. While the execution time of a computation unit placed in HW is directly obtained through feature extraction (section 6.2.2), its execution time during SW execution is modeled in Eq. 6.6. Since, this SW execution time depends on whether other computation units are implemented in HW or SW, an exact ILP formulation models all possible design points in the full design space (2^n combinations for n computation units), which in turn leads to the exponential increase in runtime of the partitioning algorithm. To alleviate this combinatorial explosion, in the proposed heuristic, the total data communication penalty of a computation unit in SW, is assumed to be the summation of individual communication penalties of computation units in HW on the SW computation unit in consideration. This limits the search space to all possible combinations of only two computation units (for n computation units and m memory blocks $m * (n - 1)$ combinations). Therefore, the time complexity of the problem is effectively reduced to $\mathcal{O}(n^2)$. This is explained in detail later in this chapter.

This assumption will be explained using an example of 5 computation units. Consider finding the execution time of the application when computation units 1 and 3 are implemented in hardware. Equation 6.9 shows the total execution time (T_{ex}) of the application and Eq. 6.10, the individual execution time of the computation units. Here, the execution time of computation units in hardware and software are modelled as in Eq. 6.7 and Eq. 6.6 respectively. Next, in Eq. 6.11 the proposed algorithm, in which the total data communication penalty of computation units 1 and 3, on computation unit 2 (similarly for 4 & 5) is assumed to be equal to the summation of individual data

communication penalties, is applied.

$$T_{ex} = \sum_{i=1}^5 T_{C_i} \quad (6.9)$$

$$T_{C_1} = HT_1, T_{C_3} = HT_3, T_{C_2} = f_2(C_2, \Delta \setminus C_2), \quad (6.10)$$

$$T_{C_4} = f_4(C_4, \Delta \setminus C_4), T_{C_5} = f_5(C_5, \Delta \setminus C_5)$$

$$f_2(C_2, \Delta \setminus C_2) = ST_2 + f_2(C_2, C_1) + f_2(C_2, C_3) \quad (6.11)$$

Thus, based on the proposed algorithm, the execution time model for C_n is generalized in Eq. 6.12. Here, the three terms refer to Hardware Time, Software Time and data communication penalty of C_n due to computation units which are placed in hardware. It should be noted that in the last term of the data communication penalty of Eq. 6.12, the “ $\wedge$ ” refers to the logical AND operator and thus implies, if $M_{nj} > 0$ and $M_{ij} > 0$, then M_{nj} is considered in the summation. The impact of the terms of Eq. 6.12 for the two cases of C_n in hardware ($x_n = 1$) and software ($x_n = 0$) are shown in Eq. 6.13 and Eq. 6.14 respectively.

$$f_n(C_n, \Delta \setminus C_n) = \underbrace{(HT_n * x_n)}_{\text{Hardware Time}} + \underbrace{(ST_n * (1 - x_n))}_{\text{Software Time}} + \underbrace{(\Omega * (1 - x_n) * (\sum_{C_i \in \Delta \setminus C_i} x_i * \sum_{M_j \in \Gamma} (M_{nj} \wedge M_{ij}) * M_{nj}))}_{\text{Data Communication Penalty}} \quad (6.12)$$

$$f_n(C_n, \Delta \setminus C_n) = HT_n + 0 + 0 \quad (6.13)$$

$$f_n(C_n, \Delta \setminus C_n) = 0 + ST_n + \Omega * (\sum_{C_i \in \Delta \setminus C_i} x_i * \sum_{M_j \in \Gamma} (M_{nj} \wedge M_{ij}) * M_{nj}) \quad (6.14)$$

Thus, the objective function can be expressed as Eq. 6.15. With the substitution of Eq. 6.12, as shown in Eq. 6.16, the final objective function is derived in Eq. 6.17. Equation 6.18 - 6.20 represent the design constraints.

$$\text{minimize } \sum_{C_i \in \Delta} T_{C_i} \quad (6.15)$$

$$\sum_{C_i \in \Delta} T_{C_i} = \sum_{C_i \in \Delta} f_i(C_i, \Delta \setminus C_i) \quad (6.16)$$

Objective function:

$$\begin{aligned} \text{minimize } & \left(\sum_{C_n \in \Delta} (HT_n * x_n) + \sum_{C_n \in \Delta} (ST_n * (1 - x_n)) + \right. \\ & \left. \sum_{C_n \in \Delta} \Omega * (1 - x_n) * \left(\sum_{C_i \in \Delta \setminus C_i} x_i * \sum_{M_j \in \Gamma} (M_{nj} \wedge M_{ij}) * M_{nj} \right) \right) \\ & \text{where } i, n = 1, 2, 3, \dots, c; j = 1, 2, 3, \dots, m \quad (6.17) \end{aligned}$$

Subject to:

$$\text{LUT Constraints: } \sum_{C_i \in \Delta} x_i * L_i \leq \zeta \quad \forall C_i \in \Delta \quad (6.18)$$

$$\text{DSP Constraints: } \sum_{C_i \in \Delta} x_i * D_i \leq \xi \quad \forall C_i \in \Delta \quad (6.19)$$

$$\text{REG Constraints: } \sum_{C_i \in \Delta} x_i * R_i \leq \lambda \quad \forall C_i \in \Delta \quad (6.20)$$

It should be noted that as a result of the heuristic formulation, the execution time obtained (T_{ex}) from solving Eq. 6.17 contains an additional penalty component due to the summation of individual data communication costs. Thus, a post optimization process is performed on the obtained solution. The post optimization is carried out using Algorithm 3 to calculate the post optimization penalty (T_{opt}). Initially, in Algorithm 3, for all combinations of two computation units in hardware, the overlapping memory accesses are identified and the overlapping memory access vector

Algorithm 3 Post Optimization Algorithm**Input:** computation units (Δ), memory blocks(Γ), placement vector (x_i)**Output:** post optimization penalty T_{opt}

```

1: for  $i \in \Delta$  do
2:   for  $n = i + 1 \in \Delta$  do
3:     if  $x_i$  &  $x_n$  then
4:       for  $j \in \Gamma$  do
5:         if  $M_{ij} \wedge M_{nj}$  then
6:           for  $v \in \Delta$  do
7:             if  $(1 - x_v) \wedge M_{vj}$  then
8:                $penalty = penalty + M_{vj}$ 
9:             end if
10:          end for
11:        end if
12:      end for
13:    end if
14:  end for
15: done
16: end for
17: done
18:  $T_{opt} = \Omega * penalty;$ 

```

(lines 1 - 5) is populated. Next, for each computation unit in software, the overlapping memory access vector is compared with the memory access pattern of the computation unit and if an overlap is identified the corresponding memory access value (M_{vj}) is added to the penalty. This process is repeated for all the computation units in software (lines 6 - 8). Finally, T_{opt} is calculated as shown in line 11 and the final execution time (T_f) using Eq. 6.21.

$$T_f = T_{ex} - T_{opt} \quad (6.21)$$

Proof of complexity: Considering a design space with n computation units and m memory blocks ($n, m \in \mathbb{Z}^+$), assuming that C_1 is placed in software, the execution time is calculated for the worst case using Eq. 6.22.

$$\begin{aligned}
& ST_1 + \\
& \Omega * [(((M_{11} \wedge M_{21}) * M_{11}) + ((M_{12} \wedge M_{22}) * M_{12}) \cdots ((M_{1m} \wedge M_{2m}) * M_{2m})) + \\
& \quad (((M_{11} \wedge M_{31}) * M_{11}) + ((M_{12} \wedge M_{32}) * M_{12}) \cdots ((M_{1m} \wedge M_{3m}) * M_{2m})) \\
& \quad \cdots \\
& \quad (((M_{11} \wedge M_{n1}) * M_{11}) + ((M_{12} \wedge M_{n2}) * M_{12}) \cdots ((M_{1m} \wedge M_{nm}) * M_{2m}))] \quad (6.22)
\end{aligned}$$

Thus, the worst case calculations consider $1 + (n - 1) * m$ computations. Thus the complexity is equal to $\mathcal{O}(1 + nm - m)$. Given that $(n, m \in \mathbb{Z}^+)$ the complexity of the proposed algorithm is $\mathcal{O}(n^2)$.

6.3 Performance Evaluation

All experiments are carried out on a virtual machine running Ubuntu on an Intel Xeon E5-1650V2 CPU host at 3.5 GHz with 8 GB RAM. Applications from the popular CHStone [191] benchmark suite are used to verify the proposed techniques. The proposed approach is compared with the techniques presented in the SoA [144], as well as an exhaustive approach. Design points (LUT constraints), distributed across the design space in each application, are provided as input constraints. Altera Cyclone V FPGA is used as the target device for feature extraction. To test the scalability and accuracy of the proposed technique, 10 benchmarks from the CHStone benchmark suite were used. As mentioned in section 6.2.2, the 25 most frequently executed computation units of each benchmark were analyzed.

Evaluation Criteria

The evaluation of the proposed approach is performed under three criteria; Criteria 1: Runtime and speed-up; Criteria 2: Performance in terms of the accuracy of results

under stringent design constraints; and Criteria 3: A holistic system evaluation of a SoC design under stringent LUT, DSP and REG constraints.

Criteria 1: The runtime of the proposed approach and the SoA work [144] is shown in Figure 6.2. Here, the horizontal axis represents the benchmarks while the vertical axes on left and right represent the $\log_{10}$ value of runtime and the number of memory blocks respectively. It should be noted that the logarithmic value is used for clarity of representation. As can be observed, the runtime of the proposed approach is in the order of milliseconds while that of the SoA is in the order of minutes. Also, the speedup obtained from the proposed approach with respect to an exhaustive approach for each benchmark is indicated in Figure 6.2. Here, the significant speed-up with increasing number of memory blocks due to the reduction in time complexity from exponential in the exhaustive approach to quadratic in the proposed approach can be seen.

Figure 6.2 Criteria 1: Runtime Comparison

Criteria 2: Criteria 2 assesses the performance in terms of the accuracy of results under stringent design constraints in comparison with the SoA work [144] normalized against the optimal values obtained from an exhaustive approach. Figure 6.3 shows the obtained results for 10 applications under consideration. Here, the vertical axis depicts the normalized performance while the horizontal axis indicates the area constraint, distributed across the design space in each application. As can be seen in Figure 6.3a - Figure 6.3j the proposed approach shows insignificant errors in most cases with a

Table 6.1 Criteria 2: Estimation Errors

Application	Estimation Error (%)	
	SoA [144]	Proposed approach
ADPCM	12.70	1.06
AES	4.02	0.00
BLOWFISH	32.31	0.89
DFADD	51.88	3.72
DFMUL	23.77	4.43
DFSIN	29.53	0.00
GSM	11.04	0.00
JPEG	1.64	0.09
MOTION	0.43	0.21
SHA	0.41	1.36
Average	16.77	1.18

maximum error of only 13% at an area constraint of 1000 in Figure 6.3j. However, the SoA work shows significant errors at multiple constraints in most applications with the maximum error exceeding 100% as can be seen in Figure 6.3d and Figure 6.3f. This is due to the approach failing to directly consider FPGA area constraints. The average estimation errors for each application for both approaches are shown in Table 6.1. Averaging across all applications the proposed approach shows an average estimation error of only just over 1% in comparison to the SoA work which has an error of almost 17%. Thus, the accuracy of estimation under such stringent design constraints reflect the suitability of the proposed approach across the full design space of an application.

(a) ADPCM

(b) AES

(c) BLOWFISH

(d) DFADD

(e) DFMUL

(f) DFSIN

(g) GSM

(h) JPEG

(i) MOTION

(j) SHA

Figure 6.3 Criteria 2: Comparison of Estimated Performance

Criteria 3: In order to evaluate the usefulness of the proposed approach in system level design, 3 different (LUT, DSP and REG) constraints, each representing a latest FPGA device from iCE40 Ultra [196], LatticeECP2/M [197] and Cyclone 10 LP [198], targeted for applications which have strict area constraints have been used. The exact constraints used in the experiments are shown in Table 6.2. Similar to the results obtained in Table 6.1 for **Criteria 2** above, Figure 6.4 plots the percentage accuracy (on vertical axis) in performance after using the proposed approach relative to the exhaustive approach, for the 3 experiments. It should be noted that the comparison is performed against an exhaustive approach, due to the inability of the SoA to consider multiple design constraints as used by the proposed approach.

Table 6.2 Criteria 3: Constraints for System Evaluation

Experiment No.	Constraint		
	LUT (ζ)	DSP (ξ)	REG (λ)
1	1000	4	1000
2	5000	8	5000
3	10000	40	10000

(a) Experiment 1

(b) Experiment 2

(c) Experiment 3

Figure 6.4 Criteria 3: Accuracy of System Evaluation

It is evident that performance achieved by accelerators recommended by the proposed approach is nearly identical to ones from the exhaustive approach for most of the benchmarks in all experiments. The DFMUL application exhibits slightly lower performance in Experiments 1 and 2. The accelerators selected by the proposed approach in this case are slightly different than the ones selected by the exhaustive approach due to the overestimation of the communication penalty as discussed in Eq. (6.9) - (6.11), where the ILP formulation assumes the overall penalty to be equal to the sum of individual penalties. Thus, when multiple computation units access the same memory block the proposed approach erroneously incurs an additional penalty resulting in an error. However, it should be noted that the accelerators recommended by the proposed approach for this application still achieves over 97% of the performance achieved by the exhaustive approach, but in a fraction of time. Overall, the average difference in performance across the 3 experiments is only 0.16%. This affirms the suitability of the fine-grained acceleration achieved by the proposed approach in system level design targeted for applications with severely strict design constraints.

6.4 Summary

This chapter proposes a novel methodology for data dependency-aware hardware-software partitioning. The proposed methodology strives to find profitable basic blocks for hardware acceleration while accounting for the complex data dependencies between the basic blocks and memory components of the application. It has been shown to be capable of fine-grained partitioning, which is desirable in the presence of basic block level accelerators.

A system level approach was adopted to model an application using computation units and memory blocks. Novel techniques have also been developed to extract features and data dependencies of the application using the target independent LLVM IR. The partitioning problem was then solved as an Oregon Trail Knapsack where the value of an item of type x depends on the presence of another item of type y in the

knapsack. In order to overcome the exponential complexity of traditional hardware software partitioning problems, an ILP-based heuristic with a time complexity of $\mathcal{O}(n^2)$ is proposed to solve the Oregon Trail Knapsack model. The proposed heuristic is capable of recommending the most suitable hardware-software partitioning under multiple user specified area constraints.

The proposed technique has been evaluated using applications from the CHStone benchmark suite on a Altera Cyclone V FPGA device. Experiments are carried out considering multiple design constraints, including LUT, DSP and REG constraints. The experimental results show that the performance of the recommended accelerators from the proposed technique is within 99% of that obtained from an exhaustive approach. This is a significant improvement compared to a SoA work which has an accuracy of only 83%. Additionally, the runtime of the approach is in the order of milliseconds as opposed to several minutes or hours in an existing SoA work and an exhaustive approach respectively.

The next chapter explores techniques for soft core processor customization by instruction subsetting to remove unused functional units. The area saved in this manner can contribute to further hardware acceleration.

Chapter 7

Instruction Subsetting for Soft Core Processor Customization

7.1 Introduction

A soft core processor, unlike the hard core counterparts allow the flexibility of adding a processor to an existing FPGA design or removing configurations from an existing processor that are not beneficial. The latter technique known as instruction subsetting, allows to save more FPGA area by removing unwanted functionality, which could be used for hardware acceleration. Obtaining the optimal partition of the design between the hardware and software components is of paramount importance to achieve the best performance under a given area constraint in a FPGA-based CSoC. Hence, customizing the processor by instruction subsetting is an important step to optimally use the FPGA real estate.

However, as discussed in chapter 3, soft core processors constitute a prohibitively large design space due to the availability of a large number of configuration options. Thus, the process of finding the optimal configuration tailored for a particular application is often tedious and time consuming. Hence, a methodology that reduces the runtime and effort for application specific customization of a soft core processor is a

necessity in the embedded systems domain with stringent TTM constraints. Further, tight area constraints imposed on embedded devices require the embedded designer to create area-efficient designs that achieve high performance. At the same time, growth of a multitude of soft-processor architectures makes it imperative that the customization methodology remains adaptable to a given processor architecture. However, existing work in the area of instruction subsetting for soft core processor customization is limited to work that requires synthesis and execution of the application [95] or exhaustive estimation for all the configurations in the design space [94]. Hence, this chapter proposes a novel application-specific instruction subsetting technique for soft core processor customization under area constraints without performing lengthy synthesis and simulation runs. The main contributions of this chapter are,

- An analysis of the dependencies for a given processor architecture
- A tree based heuristic to prune the design space in a time-efficient manner
- A methodology for rapid application specific soft core processor customization through instruction subsetting, under area constraints

The rest of the chapter is organized as follows. Section 7.2, discusses the proposed methodology followed by the results in section 7.3. The conclusion is presented in section 7.4.

Portions of the work discussed in this chapter have been published in [86].

7.2 Methodology

This section presents the methodology of rapid application-specific instruction subsetting for soft core processor customization shown in Figure 7.1. Soft core processors offer CFUs, which accelerate the performance by implementing a given set of instructions directly on hardware. If the mentioned CFU is not selected for the processor, the corresponding instruction generates an exception which is then served and executed

by the processor similar to other instructions from its ISA. The proposed approach leverages this fact to identify the CFUs which would benefit the given application and as a further step, tries to optimize the performance-area usage by finding the most suitable CFUs in an area-constrained design. The approach can be abstracted into 4 phases namely, Processor Characterization, Architecture-aware Dependency Analysis, Application Profiling & Analysis and Selection Heuristic. Even though, the *Nios II Gen2 fast* soft core processor is used for verification (henceforth referred to as *Nios II*), the proposed methodology is generic and can be extended to other soft core processors.

7.2.1 Processor Characterization

Processor characterization is a one-time semi-automated process for a given processor. This phase involves identification of the CFUs of the target processor, the corresponding instructions and the relevant performance-area characteristics for each CFU. For example, in the *Nios II* processor, 5 CFUs, namely 32-bit Integer Multiplier (IM32), 64-bit Integer Multiplier (IM64), Integer Divider (ID), Shifter (S) and the Floating Point unit (FPU) are identified. Even though, the FPU is provided as a custom instruction interface in the *Nios II* processor design, it could be analyzed as an additional CFU [199]. In addition, some CFUs can be implemented with different configurations. This further increases the complexity of the analysis in phase 2 as each configuration provides a different performance-area metric.

It is necessary to analyze the area occupied by each CFU in the FPGA real estate as the aim is to identify the configuration which maximizes performance, given an area constraint. Initially, the area of each configurable unit in hardware is derived using the Altera Quartus [200] tool. It was observed that certain configurations utilize the on-chip DSPs. In order to compare the results across all the configurations, DSPs are replaced with the equivalent adaptive logic modules (ALMs). To derive the equivalent number of ALMs for a DSP, the functionality of DSPs are switched off in the Altera Quartus tool when generating the processor.

Figure 7.1 Framework for Design Space Pruning and Soft Core Processor Customization

The next step identifies corresponding instructions which benefit from the relevant CFUs. This data is extracted from the datasheet or obtained through simulations performed under test conditions. A pre-defined set of scripts are used for simulations. This process is required as corresponding LLVM [201], [202] instructions are later mapped to the CFUs during application profiling in phase 3. Also, it is important to identify the execution time of these instructions for both hardware and software based implementations. The approach utilizes the gain of these instructions in order to predict the best possible configuration of a processor tailored for an application.

7.2.2 Architecture-aware Dependency Analysis

This is a one-time phase for a given processor. Initially, performance-area data is analyzed for all possible configurations for each CFU and the best suited configuration/configurations are identified. This step removes repetitive functionality and selects the configuration with optimal performance and minimal area in the case of configurations with comparable area.

Then, the dependencies between CFUs are analyzed to eliminate configurations which are either impossible combinations or have repetitive functionality. For example, the 64-bit Integer Multiplier cannot be configured as a standalone CFU as it necessitates the presence of the 32-bit Integer Multiplier. Also, the functionality of the Shifter CFU is automatically included when the 64-bit Integer Multiplier is configured. Such architecture-specific dependencies are analyzed in this phase of the framework.

The algorithm analyzes all possible configurations for the processor to derive the best configuration. Hence, it is necessary to accurately estimate the area utilization of configurations with multiple CFUs. Based on the results of exhaustive synthesis runs for different configurations in phase 1, it is postulated that area utilization for such configurations has an additive effect. For example, area estimation for a configuration with the base processor, Integer Divider and 32-bit Integer Multiplier should be equal

to the addition of the individual areas for the base processor, Integer Divider and 32-bit Integer Multiplier.

The benefits of the analysis are twofold. It prevents the designer from analyzing unnecessary configurations and also identifies the most profitable configuration among configurations with repetitive functionality. This step is critical as it provides significant gain to the runtime of the technique by pruning the design space.

7.2.3 Application Profiling & Analysis

The application C code is initially compiled using LLVM to produce the LLVM IR. The LLVM IR is used to extract function names, basic blocks, instructions in the basic blocks, the operands and data types. A LLVM Analysis Pass is written for this purpose. The LLVM-profiler is used to extract execution frequencies of basic blocks. The extracted details and the information obtained in phase 1 are further processed to derive the execution count of each type of instruction which is then aggregated to the abstraction level of CFUs. The extracted instruction counts as well as the area information are utilized to compute the gain and impact of each CFU for a given application using Eq. 7.1 and Eq. 7.2 respectively.

$$Gain_{CFU} = \sum_{i=1}^n (SW_{cc_i} - HW_{cc_i}) * N_i \quad (7.1)$$

$$Impact_{CFU} = \frac{Gain_{CFU}}{Area_{CFU}} \quad (7.2)$$

where N_i = Instructions of type i ,

SW_{cc_i} = Clock cycles to execute instruction i in software,

HW_{cc_i} = Clock cycles to execute instruction i in hardware

It is important that the derived dependencies among the CFUs from phase 2 are used in this phase of the methodology. For example, the 64-bit Integer Multiplier when instantiated, performs 32-bit integer multiplication and shifting in addition to the 64-bit multiplication. Similarly, the other dependencies and redundancies are utilized in analysis. Finally, the impact and area of each CFU are provided to the selection heuristic.

7.2.4 Selection Heuristic

The final phase of the proposed methodology is the area constrained selection heuristic. The output of phase 3 and user specified area constraints are the inputs to this phase. In this phase, the main goal of optimizing performance under area constraints for a given application is approached using a greedy search heuristic which is based on a sorted tree structure. This methodology is simple to implement yet has considerable accuracy with minimal time complexity.

Experiments are performed with two variants of the greedy search heuristic. The first approach is referred as the 1-greedy approach while the second approach is called the 2-greedy approach. In the 1-greedy approach, a tree sorted by the impact is pruned if the local area constraint is not met by a CFU. For example, in Figure 7.2 for the ADPCM algorithm, if the area constraint is 1700 ALMs the element at the root of the tree IM64, cannot be accommodated. The base processor accommodates 1304 ALM and the remaining area of 396 ALM is insufficient to incorporate the IM64 CFU, which requires 434 ALM. Thus, the root is pruned and moved to the next node of the tree that has the second highest impact. This element, S could be accommodated as the area constraint is met. This leaves a remainder of 352 ALMs and the next node of the tree that has an area constraint of 324 can also be accommodated. Similarly, it moves to the bottom of the tree until either the area constraint is met or the leaf node is reached. Finally, the chosen configurations are analyzed with the provided data from previous phases to eliminate any impossible configurations and derive the final output.

Algorithm 5 Pseudocode for 2-greedy Search Heuristic

Input: Remaining Area, Area of Base Processor, Area of Nodes Generate all possible combinations of 2 configurations

- 1: Sort the new combined node array according to impact
- 2: Eliminate combinations with 0 impact
- 3: Remaining Area = Remaining Area – Area of Base Processor
- 4: **for** each node j **do**
- 5: **if** Area of node corresponding to combination $j <$ Remaining Area **then**
- 6: Accommodate node j
- 7: Remaining Area = Remaining Area – Area of node j
- 8: break;
- 9: **else** Prune node j
- 10: **end if**
- 11: **end for**
- 12: Apply 1-greedy for the rest of the individual nodes

7.3 Performance Evaluation

This section presents the results of the proposed rapid application-specific instruction subsetting technique for soft core processor customization. As mentioned earlier, the Altera *Nios II* soft core processor [201] is used for the experiments. In order to verify the Processor Characterization phase of the algorithm, *Nios II* processor was exhaustively synthesized with various combinations of CFUs using the Altera Quartus development environment and Qsys tool [200]. Applications from the popular CHStone benchmark suite [174] are used to validate the proposed technique. However, CHStone applications do not include single precision floating point operations. Hence, 2 hand-coded applications that contain several single precision floating point operations have been used to exercise the single precision floating point CFU. The benchmark applications were simulated on the *Nios II* processor using ModelSim [203] to extract the execution counts. Altera Cyclone V [30] FPGA is used as the target platform of the proposed technique. However, it should be noted that the proposed technique in this work is independent of the selected FPGA platform. All experiments were carried out on a 6-core virtual machine with 8GB RAM, running OpenSUSE 13.2 on an Intel Xeon E5-1650V2 CPU host at 3.5 GHz.

Table 7.1 Equivalent ALMs of CFUs

CFU	Implementation Methods	Selected Method
32-bit Integer Multiplier	Logic elements	3 * 16 Multiplier
	3 * 16 Multiplier	
	1 * 32 Multiplier	
Shifter	Pipelined	Pipelined
	Non-pipelined	

Dependency Analysis

As mentioned in section 7.2.2, some CFUs provide several methods of implementation, thereby increasing the complexity of exploring the design space. In phase 2, one/several methods are selected for each CFU, based on performance optimization under area constraints. The respective CFUs, implementation methods and selected method are presented in Table 7.1. The observations in phase 2 of the methodology are given below.

- Nios II provides the 64-bit Integer Multiplier unit as a separate CFU. The implementation of this unit requires the presence of the 32-bit Integer Multiplier as a 3 * 16-bit or 1 * 32-bit multiplier.
- The execution time of each CFU depends on the method used for implementation. For example, the 32-bit Integer Multiplier implemented as 3 * 16-bit multipliers, 1 * 32-bit multipliers or using logic elements will have different performance metrics.
- Area utilization depends on the configuration.
- Double precision floating point instructions are always implemented in software.

The proposed selection heuristics prune the design space based on the gain achieved by implementing instructions on dedicated hardware. Table 7.2 presents the execution time in clock cycles and the corresponding gain for each configuration, considering the selected method of implementation in Table 7.1. All execution times in software

Table 7.2 Timing Characteristics of Configurations

FU	Execution Time		Gain (<i>SW</i> - <i>HW</i>)	
	Software (<i>SW</i>)	Hardware (<i>HW</i>)		
ID	56	4-66	24	
IM32	140	1	139	
IM64	150	1	149	
S	1-32	1	15	
FP	add	196	5	191
	sub	218	5	213
	div	517	16	501
	mul	774	4	770
	sqrt	494	4	490
	float to int	69	2	67
	int to float	115	4	111
	min	151	1	150
	max	160	1	159
	abs	11	1	10
	compare	91	1	90

have been obtained through experimental results, except for the Shifter CFU. The execution time in hardware and the software execution time for Shifter CFU have been obtained from datasheets available online for Nios II [8]. However, it should be noted that the execution times represent average values and does not employ the comprehensive estimation techniques proposed in chapter 3. The gain is calculated as a difference between the software and hardware execution time and is used for calculation of impact in phase 3.

Dependency analysis has the ability to rapidly prune the design space. A total of 8 independent implementation methods are considered. Theoretically, there are 256 different configurations. The design space is first pruned to 96 configurations by removing the infeasible combinations. For example, multiple instances of the same CFU using different methods of implementation cannot be configured in the same configuration. Selecting the best configuration for the remaining combinations as given in Table 7.1, further prunes the design space to 32 configurations. Dependency and redundancy analysis further reduce this to 20 configurations in the case of the selected

processor. A 79.17% reduction is achieved in the total number of configurations which require analysis.

Soft Core Processor Area Estimation

Area estimation is performed in phase 2 of the methodology and the estimated area is utilized in calculating the impact of CFUs in phase 3. Further, the selection heuristic in phase 4 also utilize the area in pruning the ordered tree. The area of the CFUs are calculated in terms of ALMs and DSPs. In order to use a common metric for calculations the area is derived in equivalent ALMs as described next.

Firstly, the area utilization of each CFU is obtained from the synthesis results after synthesizing the processor twice for every CFU, once with DSP and once without DSP support. In the case of DSP disabled configuration, the corresponding functionality is implemented in the ALMs. The area results are shown in Table 7.3. Next, the area for each CFU is deduced, by deducting the area of the base processor from the relevant configurations. The results for this process are given in Table 7.4. Comparing the area results for the DSP enabled and disabled runs, the equivalent ALMs required per DSP for each CFU is obtained as shown in Table 7.4. It is interesting to note that the equivalent ALM utilization per DSP is dependent on the CFU. This fact is attributed to organization of the DSPs and allocation of resources within the blocks. The numbers in Table 7.4 are used to estimate the area utilization of configurations with multiple CFUs.

An additive approach is used to estimate the area of configurations with multiple CFUs. Figure 7.3 presents the results of area estimation for the pruned design space in phase 2 compared to the actual values. The maximum error in area estimation is less than 1.2% and the average error in estimation is 0.55%. Hence, the additive method could be reliably deployed to estimate the configurations with multiple CFUs.

Table 7.3 Area Characteristics of Configurations

Configuration	Area		
	DSP Enabled		DSP Disabled
	ALM	DSP	ALM
Base Processor (BP)	1304	0	1304
BP + Integer Divider (ID)	1417	0	1417
BP + 32-bit Integer Multiplier (IM32*)	1349	3	1628
BP + 64-bit Integer Multiplier (IM64)	1330	3	1737
BP + Shifter*	1348	0	1348
BP + Floating Point Hardware 2 (FP2)	2369	5	2750

*Values for the selected method of implementation is mentioned.
Similar analysis has been done for all methods of implementation.

Table 7.4 Equivalent ALMs of CFUs

Configuration	Area			
	DSP Enabled		DSP Disabled	
	ALM	DSP	ALM	ALM per DSP
			<i>(Configuration – Base)</i>	<i>(Configuration – Base)/DSP</i>
ID	113	0	113	0
IM321	45	3	324	93
IM64	26	3	433	136
S1	44	0	44	0
FP2	1065	5	1446	76

Figure 7.3 Comparison of Actual Area and Estimated Area

Impact of CFUs

The results of profiling in phase 3, for the benchmarks and the hand-coded applications are presented in Table 7.5. It is observed that applications such as ADPCM and JPEG contain all types of instructions under consideration except floating point operations. Also, it is important to note that the benchmark suite consists of applications that use floating point instructions. However, it was revealed that all of them are double precision instructions. As discussed earlier in section 7.3, the Nios II core processes double precision floating point operations in software, thus these instructions were not accelerated using the Floating Point CFU. Hence, two hand-coded algorithms are used to prove the scalability of the methodology for single precision floating point arithmetic.

Table 7.5 Instruction Counts of Test Applications

Application	ID	Instruction Count			
		IM32	IM64	S	FP
ADPCM	3776	3306	56229	36590	0
MIPS	0	0	0	2926	0
GSM	0	0	1868	1955	0
BLOWFISH	0	0	0	70302	0
SHA	1044	0	0	96349	0
MOTION	0	0	0	141	0
AES	259	7023	0	5892	0
JPEG	86884	812018	1232352	1902149	0
DFADD	0	0	0	6325	0
DFDIV	27	0	1104	3471	0
DFMUL	0	0	836	2552	0
DFSIN	802	804	50184	225435	0
FLOAT_HW1	0	0	0	0	105
FLOAT_HW2	3776	3306	56229	36590	46725

Figure 7.4 presents the impact of each CFU for the test applications, computed in phase 3. The graph indicates that the impact of each CFU varies with the application and as a designer, the complexity of manually choosing the best possible configuration is tedious. Further, an area constrained design space increases the complexity as the optimization goal is based on a performance-area trade-off.

Figure 7.4 Distribution of CFU Impact

Figure 7.5 presents the results obtained for the pruned design space using the exhaustive simulation technique [94]. Here the blue dots represent the design space while the red triangles represent the pareto points. These results indicate that the configuration which gives the best performance depends on the application. Exhaustive runs will be time consuming and the results have to be further analyzed to find the optimal configuration meeting specified area constraints.

The final goal of the proposed approach is to find the optimal processor configuration under given area constraints. The exhaustive estimation technique in [94] is used to obtain the performance and area values for verification of the proposed methodology. Results of the proposed methodology and those of exhaustive technique, given area constraints of 1700, 1800 and 3400 ALMs are presented in Table 7.6. The area constraints have been selected such that 1700, 1800 and 3400 ALM could accommodate 1 CFU, 2-3 CFUs and all CFUs respectively. This represents a balanced distribution of the design space. It should be noted that in the case when all CFUs can be accommodated all CFUs where the gain (Eq. 7.1) is non-zero will be enabled as discussed in section 7.2.4.

(a) ADPCM

(b) MIPS

(c) GSM

(d) BLOWFISH

(e) SHA

(f) MOTION

(g) AES

(h) JPEG

(i) DFADD

(j) DFDIV

(k) DFMUL

(l) DFSIN

Figure 7.5 Results of Exhaustive Estimation Technique

Table 7.6 Comparison of Results

Application	ALM Constraint < 1700			ALM Constraint < 1800			Selected FUs			ALM Constraint < 3400		
	E*	1-G*	2-G*	E*	1-G*	2-G*	E*	1-G*	2-G*	E*	1-G*	2-G*
ADPCM	S+IM32	S+IM32	S+IM32	S+IM32	IM64	IM64	IM32+S	IM64 +ID	IM64 +ID	IM32+S	IM64 +ID	IM64 +ID
MIPS	S	S	S+ID	S	S	S+ID	S	S	S	S	S	S+ID
GSM	S+IM32	S	S+ID	IM32+S	IM64	IM64	IM32+S	IM64	IM64	IM32+S	IM64	IM64
BLOWFISH	S	S	S+ID	S	S	S+ID	S	S	S	S	S	S+ID
SHA	S	S+ID	S+ID	S	S + ID	S+ID	S	S	S	S	S+ID	S+ID
MOTION	S	S	S+ID	S	S	S+ID	S	S	S	S	S	S+ID
AES	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S	IM32+S
JPEG	IM32+S	IM32+S	IM32+S	IM64	IM64	IM64	IM64	IM64	IM64 +ID	IM64	IM64 +ID	IM64 +ID
DFADD	S	S	S+ID	S	S	S+ID	S	S	S	S	S	S+ID
DFDIV	S+IM32	S+ID	S+ID	S+IM32	IM64	IM64	S+IM32	IM64	IM64+ID	S+IM32	IM64+ID	IM64+ID
DFMUL	S+IM32	S	S+ID	IM64	IM64	IM64	IM64	IM64	IM64	IM64	IM64	IM64
DFSIN	S+IM32	S+IM32	S+IM32	S+IM32	IM64	IM64	S+IM32	IM64	IM64+ID	S+IM32	IM64+ID	IM64+ID
FLOAT_HW1	S+IM32	BP	BP	S+IM32	BP	BP	FP2+S	FP2	FP2+S	FP2+S	FP2	FP2+S
FLOAT_HW2	S+IM32	S+IM32	S+IM32	S+IM32	IM64	IM64	FP2+IM32	IM64+FP	IM64+FP	FP2+IM32	IM64+FP	IM64+FP
							+S	+ID	+ID			+ID
E* = Exhaustive, 1-G* = 1-Greedy, 2-G* = 2-Greedy												

The results in phase 4 prove that the 1-greedy search heuristic produces accurate results in most scenarios. The results are accurate for MIPS, BLOWFISH, DFADD, MOTION and AES. The exhaustive approach indicates that the impact of the Integer Divider is negligible in most cases. However, the datasheet indicates a significant gain. Further investigation revealed that the gain is dependent on the bit width of the operands as presented in chapter 3. The deviation of results in Table 7.6 can be attributed to this behavior of the 32-bit Integer Multiplier and Integer Divider CFUs. Changing the methodology to consider these two facts yields nearly 100% optimal results as shown in Figure 7.6. The instruction profile of FLOAT_HW1 does not map to any of the integer instructions indicated in the datasheets. However, exhaustive results reveal that performance improves for some integer configurations. Further exploration with different handwritten codes provided similar results. Even though it is not mentioned in the datasheets this indicates that Floating Point Hardware 2 unit utilizes some functionality of the integer units if these CFUs are configured. Thus, the presence of these CFUs inevitably improves performance for most applications but this will be at the cost of area.

2-greedy heuristic produces sub optimal results in most scenarios for the test applications. This is due to the fact that the algorithm initially checks for combinations of two configurations in selecting the best configuration. Several applications do not utilize all CFUs. Thus, 2-greedy approach produces poor results. However, when multiple CFUs are utilized by the algorithm the 2-greedy method also produces accurate results. Thus, it is expected that the accuracy of the 2-greedy approach to increase with the complexity of the algorithm. FLOAT_HW2 justifies this understanding.

The proposed methodology provides maximum, median and average area reductions of 59.25%, 49.52% and 47.58% respectively, compared to a design where all configurable options are enabled. Figure 7.6 depicts the reduction in area for each application compared to the configuration that has all configurable options enabled.

Another factor that needs to be considered with the exhaustive estimation approach in [94] is the time complexity of the algorithm. Considering a scenario with n possible

Figure 7.6 Comparison of Accuracy of Estimated Performance

configurable methods, using dependency analysis eliminates several configurations which effectively yields m configurations ($m \ll n$). The approach has a time complexity of $O(m)$ while exhaustive estimation approach claims $O(2^n)$. The 1-greedy and 2-greedy approaches have runtime of 204 μ s and 224 μ s respectively for the Nios II processor under consideration. This is a significant improvement compared to the runtime of 512 s for the exhaustive estimation approach discussed in [94]. Further, when the number of configurable units increase the runtime of exhaustive estimation increases considerably as each addition of a configurable unit doubles the runtime. For the proposed algorithm, runtime analysis reveals that the overhead is less than 20% for the Nios II fast processor. For example, a processor like LEON which has $5 * 2^{10}$ configurations, will require a runtime of almost 3 hours in the exhaustive approach whereas the proposed method would take less than a second. Thus, considering the significant gain in runtime and time complexity the proposed approach is superior to the exhaustive estimation approach.

7.4 Summary

A novel application-specific instruction subsetting technique for soft core processor customization has been proposed in this chapter. Subsetting of the processor microarchitecture aims to remove hardware resources for functional units that are not utilized by the application.

The design space is first pruned by exploiting dependencies between configurations before invoking a systematic subsetting of the microarchitecture by referring to the LLVM compiler generated intermediate representation of the application. Impact ordered tree based selection heuristics have been proposed to perform application-specific instruction subsetting. It is noteworthy that the proposed methodology can also be used for customizing other soft core processors with similar configurable functional units on FPGA.

The proposed technique has been tested using applications from the CHStone benchmark suite and 2 handwritten applications on the Altera Nios soft core processor. The methodology provides maximum, median and average area reductions of 59.25%, 49.52% and 47.58% respectively compared to a processor with all configuration options enabled. Further, the runtime of the proposed 1-greedy and 2-greedy selection heuristics are 204 μ s and 224 μ s respectively, as opposed to 512 s taken in an existing approach with a linear runtime complexity.

Chapter 8

Wibheda+: Integrated Framework for Automating CSoC Platform Generation

8.1 Introduction

This chapter proposes Wibheda+, an integrated framework for automating CSoC platform generation shown in Figure 8.1. Wibheda+ unifies support for the popular FPGA vendors Altera [204] and Xilinx [205] while having the ability to support different soft and hard core processors. In addition, Wibheda+ can be extended to other FPGA vendors, soft core and hard core processor architectures. It should be noted that to the best of our knowledge, this is the first attempt at devising a holistic tool which caters to these requirements. Thus, the main contribution of this chapter is *an automated framework that intelligently recommends the most suitable FPGA-based CSoC device, on-chip soft core processor and configuration/hard core processor and partitioning of the design between hardware and software, in order to obtain optimal performance for a given application under user specified LUT, REG and DSP constraints. Further, the framework also provides a user the choice to specify functions*

Figure 8.1 Wibheda+ Framework

mandatory to be implemented on the on-chip processor (e.g. when iterative updates are necessary), preferred FPGA-based SoC device/devices, vendor and processors.

Wibheda+ unifies the techniques proposed in chapters 3-7, in proposing an integrated framework for automating CSoC platform generation. This chapter elaborates on the techniques used in realizing this framework. The rest of the chapter is organized as follows. Initially, the contribution of each of the chapters to Wibheda+ is summarized. Next, the methodology is discussed in section 8.2 followed by the results in section 8.3 and conclusion in section 8.4.

Portions of the work discussed in this chapter have been published in [206].

Section A in Figure 8.1

Section A in Figure 8.1 depicts the contribution from chapter 3. Chapter 3 proposed a technique for soft core processor performance estimation. The proposed technique uses the application C code to estimate the performance for any configuration on the soft core processor. Initially, the C code is converted to the LLVM IR. Next, the processor ISA is mapped to the instructions in the LLVM IR. Later, analytical and mathematical models are used to estimate the performance of different types of instructions and pipeline hazards in the IR of the application. The sum of the performance of all the instructions provide the performance of the application on the soft core processor for a given configuration. The techniques proposed in chapter 3 have been tested on Xilinx MicroBlaze and Altera Nios soft core processors.

Section B in Figure 8.1

Section B in Figure 8.1 depicts the contribution from chapter 4. Chapter 4 proposed a technique for performance estimation in dual-issue, out-of-order, superscalar hard core processors. Similar to chapter 3, the proposed technique uses the application C code to estimate performance on the hard core processor by converting it to the LLVM IR and subsequently mapping the processor ISA to the instructions in the LLVM IR. Next, the work proposed in chapter 3 is extended by incorporating analytical models and a scheduling algorithm to estimate the performance of different types of instructions and

pipeline hazards in the IR of the application. Finally, the sum of all the instructions provide the performance of the application on the hard core processor. The technique proposed in chapter 4 has been tested on a ARM Cortex-A9 hard core processor. Further, the process of incorporating the given techniques to other hard core processors is explained.

Section C in Figure 8.1

Section C in Figure 8.1 depicts the contribution from chapter 5. Chapter 5 proposed a technique for hardware area-time estimation of FPGAs that can be applied across different FPGA vendors and device families. The proposed technique uses the application C code to estimate hardware area-time consumption on the FPGA by converting it to the LLVM IR and subsequently seamlessly integrating with the LegUp tool and free versions of commercial tools. The proposed approach does not necessitate time consuming post synthesis steps such as Place & Route and Bit Stream Generation in order to obtain reasonably accurate area estimation measures. The technique proposed in chapter 5 has been verified for multiple FPGA device families from Altera and Xilinx.

Section D in Figure 8.1

Section D in Figure 8.1 depicts the contribution from chapter 6. Chapter 6 proposed a technique for fine-grained data dependency- and area- constraint aware hardware-software partitioning. The proposed technique uses the application C code, which is converted to the LLVM IR in order to obtain the fine-grained computation units for hardware-software partitioning. Next, the technique models the hardware-software partitioning problem as an Oregon Trail Knapsack. Subsequently, the Oregon Trail Knapsack is solved using a proposed ILP-based heuristic which has a time complexity of $\mathcal{O}(n^2)$.

Section E in Figure 8.1

Section E in Figure 8.1 depicts the contribution from chapter 7. Chapter 7 proposed an instruction subsetting technique for soft core processor customization under user

specified area constraints by exploiting dependencies between the various configuration options. The proposed technique uses the application C code, which is converted to the LLVM IR for analysis. The method recommends the most suitable processor configuration for a given application. The technique proposed in chapter 7 has been verified for the Altera Nios soft core processor. Further, it can be applied to other soft core processors with similar configurable functional units on a FPGA.

8.2 Methodology

Figure 8.2 GUI of Wibheda+

This section discusses the architecture of Wibheda+ shown in Figure 8.1. It should be noted that Wibheda+ is fully automated and can be used at the click of a button using the graphical user interface (GUI) shown in Figure 8.2.

8.2.1 Input Parameters

Wibheda+ accepts a C application and the area constraints in-terms of LUT, REG and DSP which are the typical resources available on a FPGA. The user can also specify the FPGA vendor, device family and preferred processor. However, the framework can operate without these options as well. Further, if the user prefers certain functions of the application (henceforth referred as mandatory SW blocks) implemented on the on-chip processor, this too can be specified. The input parameters in the GUI have been highlighted in red boxes in Figure 8.3. In this case, the C application is *dfmul* and one function (*float64_is_signalling_nan*) has been selected as mandatory for software implementation. The selected FPGA platform is Altera Cyclone V and the processor is Nios II.

Figure 8.3 GUI of Wibheda+ - Input Parameters

8.2.2 C to IR Conversion (Section A)

Initially, the C application is converted to an IR using the LLVM compiler [168] (The process of converting the C code to LLVM IR is initially introduced in chapter 3). Thereafter, Wibheda+ identifies and annotates user specified mandatory SW blocks to be implemented on the on-chip processor. The user specifies these blocks as functions mandatory to be implemented in software. Wibheda+ extracts these functions and annotates the blocks belonging to each of these functions within the LLVM IR.

8.2.3 Profiling (Section A)

Next, the application is profiled using the LLVM profiler, similar to the method described in chapter 3. The LLVM profile contains the basic blocks of the application, the instructions in each basic block, and the execution frequencies of each of these basic blocks which are used in the subsequent stages.

After profiling is completed Wibheda+ performs the soft core processor performance estimation (section 8.2.4), hard core processor performance estimation (section 8.2.5), data dependency analysis (section 8.2.6) and hot block identification (section 8.2.7). Processor performance estimation is performed only for the soft/hard core processor specified by the user. However, if the processor is not specified, Wibheda+ only performs hard core processor performance estimation. This is based on the fact that hard core processors are superior in performance compared to soft cores.

8.2.4 Processor Performance Estimation - Soft Core (Section A)

Performance is estimated for all basic blocks of the application on a soft core processor using the technique proposed in chapter 3. Also, it should be noted that soft core performance estimation is performed for a soft core processor with all configuration options enabled.

8.2.5 Processor Performance Estimation- Hard Core (Section B)

Similar to the case of soft cores, software performance on hard core processors is estimated for all basic blocks of the application using the technique proposed in chapter 4.

8.2.6 Data Dependency Analysis

The purpose of this stage is to identify the communication costs between the basic blocks arising as a result of data sharing. The data flow information between hot blocks and memories are extracted using a control data flow graph through an LLVM analysis pass as discussed in chapter 6 (section 6.2.2).

8.2.7 Hot Block Identification

During this stage, Wibheda+ uses the outputs of data dependency analysis in section 8.2.6, in order to identify the hot blocks, i.e. the blocks which are most amenable for hardware acceleration. The list of basic blocks ordered in terms of frequency obtained from section 8.2.3 is used for this purpose. The list also contains the data flow information of each basic block. The following procedure is adhered to in selecting the hot blocks.

1. Remove all blocks with 0 frequency.
2. Select the top 10% of blocks in order of frequency as hot blocks.
3. Obtain the blocks that interact with the already selected blocks. In these blocks, select the blocks which communicate most with the top 10% of blocks such that the total number of selected hot blocks is 20% of the blocks (This is based on the fact that the application spends most of its time in executing these basic blocks [207]). It should be noted that the results from the dependency analysis process in section 8.2.6 is used to identify the communication cost between blocks.

4. In cases where the blocks with interactions do not amount to constitute 20% of the blocks, the next set of blocks in order of frequency will be selected.

As mentioned above, in grouping hot blocks, the blocks that communicate the highest with the top 10% of most frequently executed blocks is given priority for selection. As such, the overhead caused by communicating between hardware and software can be minimized. This adds to the effectiveness of the communication model adopted in the hardware-software partitioning stage (as discussed in chapter 6) to mitigate the communication overhead of fine grained partitioning, while still providing the ability to exploit its advantages over a coarse grained approach.

8.2.8 Hardware Area-Time Estimation (Section C)

Hardware area-time (hardware area in terms of LUT, REG, DSP and execution clock cycles in hardware) is estimated for all hot blocks of the application. This is performed based on the technique proposed in chapter 5.

8.2.9 Hardware-Software Partitioning (Section D)

At this stage, Wibheda+ partitions the hot blocks between the hardware (reconfigurable logic) and software (on-chip processor) with the objective of maximizing application performance (minimizing application execution time). The technique proposed in chapter 6 that uses an Oregon trail based Knapsack problem modeled as an ILP-based heuristic is used for this purpose. The outcome of this stage recommends the optimal partitioning of hot blocks between the hardware and software in the SoC.

8.2.10 Iterative Customization

Next, Wibheda+ strives to optimize the FPGA area by iteratively partitioning the design further with the remaining basic blocks (non hot blocks), which is termed as

iterative customization. During each iteration, the next set of hot blocks is selected using the techniques proposed in section 8.2.7. Next, hardware area-time estimation (section 8.2.8) is performed for the new set of blocks. Thereafter, hardware-software partitioning (section 8.2.8) is performed considering all the blocks which have been considered as hot blocks in all iterations so far. This ensures the selection of most suitable candidate blocks for hardware implementation from all the new hot blocks under consideration. The iterative process terminates if the remaining area is less than the minimum area occupied by a basic block so far. It should be noted that the iterative approach can reduce the design space as well as the runtime of Wibheda+ by progressively increasing the number of basic blocks instead of considering all basic blocks in a single partitioning stage.

Table 8.1 Sample LLVM profile of DFMUL

Rank	Basic Block #	Execution %	Function Name
1	bb16	9.20	extractFloat64Sign()
2	bb15	9.20	extractFloat64Exp()
3	bb14	9.20	extractFloat64Frac()
4	bb88	4.83	main()
5	bb43	4.60	float64_mul()
6	bb90	4.60	main()
7	bb18	4.60	float64_mul()
8	bb89	4.60	main()
9	bb26	3.68	float64_mul()
10	bb17	3.45	packFloat64()
11	bb32	2.99	float64_mul()
12	bb36	2.30	float64_mul()
13	bb69	1.84	roundAndPackFloat64()
14	bb42	1.84	float64_mul()
15	bb41	1.84	float64_mul()
16	bb57	1.84	roundAndPackFloat64()
17	bb40	1.84	float64_mul()
18	bb85	1.84	roundAndPackFloat64()
19	bb84	1.84	roundAndPackFloat64()
20	bb82	1.84	roundAndPackFloat64()
21	bb80	1.84	roundAndPackFloat64()
22	bb13	1.84	mul64To128()
23	bb5	1.38	float64_is_signaling_nan()
24	bb3	1.38	float64_is_signaling_nan()
25	bb2	1.38	float64_is_nan()
26	bb19	0.92	float64_mul()
27	bb20	0.69	float64_mul()
28	bb44	0.69	propagateFloat64NaN()
29	bb33	0.69	float64_mul()
30	bb34	0.69	float64_mul()

The process of iterative customization will be further explained with an example considering the DFMUL application. The LLVM profile of DFMUL contains 54 basic blocks that are executed. Table 8.1 provides the ranking of these basic blocks based on

execution frequency for the top 30 blocks along with the corresponding basic block number, the execution percentage and the name of the function that contains each basic block. During the hardware-software partitioning and iterative customization stages, initially the first set of hot blocks will be partitioned and subsequently, the rest of the basic blocks will be considered based on the design constraints. Table 8.2 shows the recommendations of Wibheda+ for 3 different design constraints. Analysis reveals that basic blocks such as bb3 and bb2 are not selected as hot blocks in the initial iteration. However, since iterative customization strives to select blocks for hardware implementation in order to maximise performance, it can be observed that even at tighter design constraints (e.g 1000 LUT, REG) where the total number of selected blocks is less than 20% of the blocks in the application (which is the maximum number of hot blocks that could be selected in the first iteration) the recommended hardware blocks consist of bb3 and bb2. This verifies the benefits of the iterative approach as opposed to a fixed size method.

Table 8.2 Recommendations of Wibheda+ for DFMUL

Design Constraints			Recommended HW Blocks (bb #)	Estimated HW Area			Estimated Clock Cycles
LUT	REG	DSP		LUT	REG	DSP	
1000	1000	2	16,15,14,90,18,17,41,3,2	686	947	0	4756
2000	2000	10	16,15,14,18,17,69,57,82,13,3,2	1230	1972	8	3166
4500	4500	30	16,15,14,90,18,89,17,69,42,41,57,40,84,82,13,3,2	2205	3638	24	1886

8.2.11 Soft Core Processor Subsetting and Area Optimization (Section E)

After Wibheda+ exits the iterative customization process, it checks if the selected processor is a hard core processor. In this case, the tool recommends the suitable FPGA device and the partition. However, if the selected processor is a soft core processor, it further customizes by subsetting the processor and strives to maximally utilize area.

Figure 8.4 Technique for Soft Core Processor Subsetting and Area Optimization

The technique for soft core processor subsetting and area optimization is depicted in Figure 8.4. Initially, the processor is customized by subsetting based on the instructions in the remaining blocks recommended for software implementation (hot blocks in software and non-hot blocks). The technique proposed chapter 7 is used for instruction subsetting. However, the goal at this stage is to subset the processor in order to disable any unused configuration such that additional area could be obtained for hardware acceleration. As such, in this case only the configurable functional units of the soft core processor where the gain obtained by a configurable functional unit (Equation 7.1 in chapter 7) is equal to zero will be removed. Next, Wibheda+ checks whether there is a change in processor configuration from the previous configuration. If the configuration does not change the process terminates recommending the current partition and processor configuration.

However, since the customization is performed for the remaining blocks, a change of configuration will result in a leaner processor, creating more area for hardware

acceleration. As such, if there is a change in processor configuration, the remaining area is calculated, and more hardware blocks are allocated based on an ordered list. The ordered list is generated from the set considering the hot blocks recommended for software implementation in section 8.2.9. In this list, the blocks recommended for software implementation are sorted based on the gain in hardware implementation (hardware performance - software performance). Once the new hardware blocks have been allocated the framework checks as to whether the processor could be customized any further, i.e. whether the base processor configuration has been reached. If this is the case, the process terminates recommending the current partition and processor configuration, if not it continues iteratively.

8.3 Performance Evaluation

This section, initially explains the experimental setup and analyze the runtime of Wibheda+. Next, a case study is discussed to demonstrate the features of Wibheda+ and finally compare the accuracy with respect to an exhaustive approach.

All experiments are carried out on a virtual machine running Ubuntu on an Intel Xeon CPU host at 3.5 GHz with 8 GB RAM. Applications from the popular CHStone [191] benchmark suite are used to validate the proposed techniques. CHStone benchmarks contain applications from different domains such as image processing, encryption etc. which enables to validate Wibheda+ on different application domains. Furthermore, Altera Cyclone V, Cyclone II and Arria II as well as Xilinx Kintex 7 and Artix 7 device families are used in the experiments as they are commonly used devices in FPGA-based SoC applications. Similarly, Nios and MicroBlaze soft core processors have been considered based on popularity and micro-architectural features of the processor. The ARM Cortex A9 hard core processor is selected for the experiments as this is available in both Altera and Xilinx FPGAs. The FPGA device families used in the validation process and the respective soft core and hard core processors supported by the devices are given in Table 8.3.

Table 8.3 FPGA Device Families and Processors

Vendor	Device Family	Soft Core	Hard Core
Altera	Cyclone V	Nios II(Nios)	ARM Cortex A9
	Cyclone II	Nios	Not Applicable
	Arria II	Nios	Not Applicable
Xilinx	Kintex 7	MicroBlaze(MB)	Not Applicable
	Artix 7	MB	ARM Cortex A9

Table 8.4 Average Runtime (min:sec) of Wibheda+

On-chip Processor	Application			
	DFMUL	GSM	MOTION	JPEG
ARM	6:32	8:34	9:55	23:21
Nios	7:11	8:29	8:34	26:21

Runtime

Runtime of Wibheda+ is analyzed for scenarios with different design constraints on an Altera Cyclone V device. Four applications, namely DFMUL, GSM, MOTION and JPEG, which possess different numbers of basic blocks and data dependency characteristics are selected. The average runtime for the cases of soft core and hard core processors are presented in Table 8.4. As can be seen the average runtime of Wibheda+ is in the order of minutes. Further, the trend of runtime increase in applications due to the number of basic blocks was observed. For example, DFMUL which contains a total of 100 basic blocks has an average runtime in the range of 6 - 7 minutes in comparison to JPEG with 23 - 26 minutes, which has 352 basic blocks. However, it is noteworthy that even then the runtime of Wibheda+ is significantly superior compared to a process that requires manual intervention or an exhaustive approach.

An Experimental Analysis

This section experimentally analyses the recommendations of Wibheda+ for the GSM application on an Altera Cyclone V FPGA. The experiment is conducted with the two processors, ARM hard core and Nios soft core. Further, for each processor, two

Table 8.5 Experimental Scenarios

Scenario #	Processor	Mandatory SW Blocks
1	ARM	N/A
2	ARM	<i>gsm_mult_r</i>
3	Nios	N/A
4	Nios	<i>gsm_mult_r</i>

Figure 8.5 Scenario 1

scenarios, namely without any mandatory SW blocks provided by the user and for the case of function *gsm_mult_r* specified as a mandatory SW block were considered. Scenarios used for analysis are classified in Table 8.5. A common constraint of 15 DSP, 5000 LUT, 5000 REG is selected across all scenarios. Figures 8.5, 8.6, 8.7 and 8.8 show the input and output of Wibheda+ for scenarios 1, 2, 3 and 4 respectively. Outputs specifically discussed in the subsequent sections have been highlighted in red boxes in each figure.

Comparing scenario 1 and scenario 2, it was observed that the change in the two scenarios is the addition of an extra design constraint in terms of a mandatory SW block (*gsm_mult_r*) in scenario 2. Here, it was observed that none of the basic blocks from the function *gsm_mult_r* that were previously recommended for hardware implementation

Figure 8.6 Scenario 2

in scenario 1 (highlighted in red box in Figure 8.5) are recommended here. As a result, the performance estimation for scenario 2 is relatively lower compared to scenario 1 for the corresponding test cases. Further, it was also observed that area consumed is also less due to the selection of a lesser number of blocks for hardware implementation.

In scenario 3 and scenario 4, the experiments were conducted with the Nios soft core processor instead of the ARM hard core processor used in scenario 1 and scenario 2. Here, it was observed that the area consumption was relatively large compared to scenario 1 and scenario 2. This is due to increased area requirement as a result of implementing the soft core processor on the reconfigurable logic. Further, it was also noticed that the estimated performance of the designs are lower compared to scenario 1 and 2. This is due to the superiority of performance of hard core processors over soft cores. In scenario 4, the experiments were performed with the Nios soft core processor with the additional constraint of the mandatory SW block (*gsm_mult_r*) in addition to the constraints in scenario 3. Thus, it was observed that hardware area consumption of the design points are relatively lower compared to corresponding test cases in scenario 3. Further, similar to scenario 2 when the ARM hard core processor was used with a

(a) Input

(b) Output

Figure 8.7 Scenario 3

(a) Input

(b) Output

Figure 8.8 Scenario 4

mandatory SW block, estimated performance of these test cases are lower compared to scenario 3.

Validation

As mentioned in section 8.1, to the best of our knowledge, existing work do not contain methodologies which could be directly compared with Wibheda+. Therefore, the recommendation of Wibheda+ was compared, with those obtained through an exhaustive approach. The difference of performance of the recommendation for each processor and respective FPGA devices, are shown in Tables 8.6 and 8.7. The values in Table 8.6 and 8.7 have been obtained for the smallest and the largest device in each FPGA device family in Table 8.3. Here, the FPGA resources available in each selected device is considered as the design constraint. Further, mandatory SW blocks are not specified.

Table 8.6 Percentage Difference in Performance (smallest devices)

FPGA Vendor	Device Family	On-chip Proc.	Application											Average
			adpcm	aes	blowfish	dfadd	dfdiv	dfmul	dfsin	gsm	jpeg	mips	motion	
Altera	Arria II	Nios	0	0	0	0	0	0	0	3.908	0	0	0	0.355
	Cyclone II	Nios	0	3.623	1.929	2.51	0.517	0	1.21	0	0	0	0.937	0.975
	Cyclone V	Nios	0	0	0	0	0.015	0	0	3.908	0	0	0	0.357
		ARM	0	0	-	-	0.041	0	0	0	0	0	-	0
0.0403														
Xilinx	Artix 7	MB	-	0	0	0	0	0	-	0	-	-	0	0
		ARM	0	0	-	-	0	0	0	0	0	0	-	0
	Kintex 7	MB	-	0	0	0	0	0	-	0	-	-	0	0
		ARM	0	0	-	-	0	0	0	0	0	0	-	0
0														

It is evident that performance achieved by the recommendation made by Wibheda+ is nearly identical to ones from the exhaustive approach in all experiments. Overall, the average difference in performance across all experiments is only 0.012% for Altera and there is no difference for Xilinx. Thus, the overall average accuracy of Wibheda+

Table 8.7 Percentage Difference in Performance (largest devices)

FPGA Vendor	Device Family	On-chip Proc.	Application											Average
			adpcm	aes	blowfish	dfadd	dfdiv	dfmul	dfsine	gsm	jpeg	mips	motion	
Altera	Arria II	Nios	0	0	0	0	0	0	0	0	0	0	0	0
	Cyclone II	Nios	0	0	0	0	0	0	0	0	0	0	0	0
	Cyclone V	Nios	0	0	0	0	0.015	0	0	0	0	0	0	0.0014
		ARM	0	0	-	-	0.041	0	0	0	0	-	0	0.005
														0.0014
Xilinx	Artix 7	MB	-	0	0	0	0	0	-	0	-	-	0	0
		ARM	0	0	-	-	0	0	0	0	0	-	0	0
	Kintex 7	MB	-	0	0	0	0	0	-	0	-	-	0	0
		ARM	0	0	-	-	0	0	0	0	0	-	0	0
														0

is 99.99%. It is noteworthy that even for the smallest device in Cyclone II which has only 4608 LUT, REG and 13 DSP, a significantly small FPGA, the average difference in performance is only 0.9%. This shows the suitability of Wibheda+ across different FPGA devices and sizes. Further, as the runtime of Wibheda+ is in the order of minutes compared to hours or days for an exhaustive approach, it highlights the ability of Wibheda+ in providing both fast and reliable solutions. This affirms the suitability of Wibheda+ in generating application-specific FPGA-based SoCs.

An Example Implementation

This section provides a description of the setup for implementation of recommended designs. The approach proposed in [39] is used for this purpose. The proposed techniques are implemented and verified on the Altera SoC development kit containing a Cyclone V FPGA device. Altera provides provision for incorporating various types of custom instructions into their Nios II soft core processor including support for multi-cycle custom instructions. This is a useful feature since it allows the addition of arbitrary size custom instruction without significantly affecting the processor's operating frequency. However, in other processors where such an option is not available a pre-defined memory space could be accessed to trigger the accelerator execution.

On starting execution, the accelerator can block the bus access for other operations by asserting a wait state on the bus.

The GSM application has been implemented as a hardware-software partitioned design on the Cyclone V SoC development kit. The Verilog RTL code for the recommended hardware blocks have been obtained from the RTL code generated in the proposed hardware area-time estimation techniques in section 8.2.8. Interfaces are generated using Altera's Qsys and Quartus tools.

Table 8.8 depicts the actual performance (in clock cycles) and the one estimated by Wibheda+. Results indicate that the estimated performance of Wibheda+ is significantly close to the actual values obtained by implementing the recommended partition on FPGA. However, due to the cumbersome manual intervention involved, implementation was performed only for 2 scenarios purely as justification of the practical implementability of Wibheda+'s recommended application-specific customization of a suitable FPGA platform. Further, due to the trivial research component of the automation process, the process was not automated as it is out of the scope of this work.

Table 8.8 Implementation of Wibheda+ Recommendations

Scenario	Constraints			Recommended HW blocks (bb #)	Performance (Clock Cycles)		Performance Difference (%)
	LUT	DSP	REG		Wibheda+	Actual	
1	2400	10	2400	17,16,13	79914	74388	7.43
2	2500	20	2500	17,16,13,24,18	78241	74701	4.74

Processors with Similar Performance

As mentioned in this chapter, Wibheda+ can be scaled to support any number of soft or hard core processors. Furthermore, as discussed in section 8.2, Wibheda+ selects to use a hard core processor when the user does not specify to use a soft core. This is based

on the fact that hard core processors offer superior performance compared to soft core processors. However, when supporting multiple processors either soft core or hard core, two or more processors could have similar performance for a given application. Due to the similarity of processor performance, hardware-software partitioning could be performed for all processors under consideration and the most suitable processor and FPGA platform can be selected thereafter based on the performance of the partitioned design. This will ensure that the processor and FPGA providing the best performance for a given application is the one that is selected.

8.4 Summary

The various methods proposed in the earlier chapters have been integrated to facilitate the automatic generation of constraint-aware platforms for application specific CSoC. The automated framework, called Wibheda+, strives to obtain the best performance under user specified area constraint and soft core/hard core processor (if any).

The area constraints are specified in terms of LUT, REG and DSP. It also allows users to instruct a portion of the software code to be excluded from consideration for hardware-software partitioning. This is especially useful for large applications where certain sections of the code are still under development and hence easy to update in software instead of hardware.

The techniques in chapter 3 and chapter 4 are used for processor performance estimation in order to obtain the software time of the application on soft and hard core processors. This is followed by the identification of the hot blocks or the blocks most amenable for hardware acceleration based on the execution frequency of blocks and dependencies among associated blocks. The hardware area-time is obtained using the technique discussed in chapter 5 while hardware-software partitioning is based on the technique in chapter 6. Finally, the proposed technique performs instruction subsetting as proposed in chapter 7 in order to make optimal usage of hardware area.

Wibheda+ has been automated and tested extensively for a comprehensive evaluation using applications from the CHStone benchmark suite on Cyclone V, Cyclone II and Arria II FPGA devices from Altera as well as Kintex 7 and Artix 7 from Xilinx. Also, ARM Cortex A9, Altera Nios and Xilinx MicroBlaze processors have also been used for evaluation. Experimental results show that Wibheda+ can consistently identify near-optimal solutions in a few minutes. Wibheda+ can also be relied upon to recommend the most suitable FPGA platform and processor (and configuration) among the supported FPGA devices and processors.

Chapter 9

Conclusions and Future Directions

9.1 Conclusions

In this thesis, several methods for automatic generation of FPGA-based CSoC platforms to satisfy the increasingly strict NRE and TTM costs have been proposed. The proposed techniques support scalability and efficiency to facilitate the automatic generation of CSoC platforms that are based on FPGAs from different device families and vendors.

In order to select the most appropriate soft core processor to execute the software components of the application, a rapid performance estimation technique has been proposed in Chapter 3. Unlike existing techniques that are typically developed for a single processor architecture, the proposed technique represents the processor ISA using the ISA of the open source LLVM compiler to make it easily deployable for different processor architectures. The proposed technique accounts for both control hazards and data hazards by using mathematical models and pre-defined instruction sequences, respectively, to improve the overall estimation accuracy. In order to cater for the presence or absence of various CFUs in a soft core processor, the proposed methodology estimates the execution time of instructions, pertaining to these CFUs, both with and without the CFU instantiated. The proposed technique has been eval-

uated on two widely used commercial soft core processors, Altera Nios and Xilinx MicroBlaze while executing several applications from the CHStone benchmark suite. The tests show an error of mere 5% averaged across the various benchmark runs on the two processors. While the proposed technique achieves high estimation accuracy on current soft core processors that rely on in-order execution, they can also be extended to consider for out-of-order pipelines that may be incorporated into future soft core processors.

In order to leverage on the high performance hard core processors available in latest FPGA devices, the technique proposed in Chapter 3 was extended to model advanced architectural features such as dual-issue, out-of-order and superscalar processing found in such processors. Additionally, models for representing structural hazards due to the dual-issue pipeline while modelling the data and control hazards have been introduced to realize high estimation accuracy. A scheduling algorithm was also devised to generate the final execution sequence of the instructions using these models as well as the architectural features. The execution time, obtained from either processor datasheets or through microbenchmarks, of each instruction of the instruction execution sequence was then computed to estimate the total execution time. The proposed technique has been evaluated using the popular ARM Cortex-A9 processor, used in both Altera SoC and Xilinx Zynq FPGA devices, by executing applications from the CHStone benchmark suite and has been shown to be highly accurate with an average estimation error of less than 6%. The estimation techniques for soft and hard core processors have been integrated into a comprehensive and novel framework that not only captures the features of current processors, but also provides a mechanism for incorporating performance centric features of future processors.

A rapid and reliable technique for estimating the hardware area-time requirements of applications on FPGAs has been proposed in Chapter 5 that is capable of supporting different FPGA vendors and device families. The technique first converts the application C code to the target independent LLVM IR and then modifies the IR to leverage on the open source LegUp tool to generate the corresponding RTL code. In

order to overcome the limitation of the LegUp tool that is primarily geared towards Altera FPGAs, Altera specific IP cores in the generated RTL have been replaced with generic RTL code. To ensure high area estimation accuracy for FPGAs from different manufacturers, vendor-specific soft IP cores are represented as generic RTL code before synthesizing it on the vendor tools. The reliance on vendor tools has made it possible to be easily portable for different FPGA manufacturers and device families. It is also noteworthy that the area estimation is obtained directly after the Synthesis step and does not necessitate the time-consuming post synthesis steps such as Place & Route and Bit Stream Generation. The estimated execution time, extracted from the LegUp scheduling report, is amended to accommodate variations due to the conversion to generic RTL and incorporation of soft IP cores. The techniques proposed in this chapter have resulted in a rapid area-time estimation on a variety of FPGAs with high estimation accuracy.

Chapter 6 presents a data communication-aware hardware-software partitioning technique to rapidly select profitable hardware accelerators. The accurate modeling of data dependencies makes the proposed technique suitable for applications of different size and complexity. The partitioning problem is modelled as a 0-1 Oregon Trail Knapsack problem, which accurately captures the behavior of hardware accelerators whose performance not only depends on their own selection but also the inclusion of other accelerators. In order to avoid the otherwise exponential complexity of the partitioning problem, an ILP-based heuristic was devised to solve the Oregon Trail Knapsack model with a time complexity of $\mathcal{O}(n^2)$. Using applications from the CHStone benchmark suite on an Altera Cyclone V FPGA device, the technique has been demonstrated to achieve an accuracy of nearly 99% as opposed to 83% accuracy achieved by an existing state-of-the-art technique. Unlike existing works that do not consider data communication cost between hardware-software components, the proposed method has resulted in a more realistic hardware-software partitioning solution.

In order to minimize area required for the soft core processor, a rapid subsetting technique was proposed in Chapter 7 to facilitate the systematic removal of unutilized configurable functional units. The area saved was utilized to incorporate additional hardware accelerators in order to further improve compute performance. Unlike existing techniques for instruction subsetting that either rely on time consuming synthesis-based approaches or exhaustive estimation-based techniques, a design space pruning phase has been introduced to analyse the inherent dependencies between the various CFUs to eliminate redundant options. Impact ordered tree-based heuristics, which use the ratio between performance gain and area of the CFU pertaining to the instruction as the metric, were designed to further speed up the processor subsetting step. The proposed methodology was evaluated using applications from the CHStone benchmark suite running on the Altera Nios soft core processor and led to an average area reduction of 47.58% without any loss in performance when compared to the un-subsetted processor. Additionally, the proposed heuristics achieve a speedup of over 10^6X when compared to an existing approach. Finally, the linear runtime complexity of the proposed approach makes it ideally suited for the rapid subsetting of soft core processors with a large number of configurable options.

All the methodologies presented in this thesis have been integrated into a comprehensive framework called Wibheda+ that can be leveraged upon for automatic generation of application-specific FPGA-based CSoC platforms. The integrated framework takes various user-defined inputs such as area constraints, desired FPGA vendor/device and processor options to perform extensive yet rapid design space exploration in order to recommend the most suitable FPGA-based CSoC. The recommended CSoC includes hardware and software components at the basic block level, selected hard or soft core processor (subsetting in case of soft core) and appropriate memory blocks. The fully automated tool, complete with a user-friendly GUI, has been extensively evaluated with applications from the CHStone benchmark suite on Cyclone V, Cyclone II and Arria II FPGA devices from Altera as well as Kintex 7 and Artix 7 from Xilinx. In addition, ARM Cortex A9, Altera Nios and Xilinx MicroBlaze processors have also been used during the evaluation process to demonstrate the effectiveness of Wibheda+.

It is envisaged that the high levels of accuracy and runtime efficiency of the proposed framework will contribute to closing the design productivity gap in FPGA-based CSoC. Finally, the contributions in this thesis have made it possible to automate the complex problem of FPGA-based CSoC platform generation for large applications, given the area-time constraints.

9.2 Future Research Directions

1. Extending Soft Core Processor Performance Estimation to Out-of-Order Soft Core Processors

Out-of-order execution improves processor performance through instruction-level parallelism to execute independent instructions in parallel and to tolerate instructions with higher latency, such as memory operations. With the emergence of new architectures being proposed for out-of-order soft core processors [208], it will be useful to provide support for performance estimation of out-of-order soft core processors. It is envisaged that the techniques proposed in Chapter 4 to support out-of-order processing on hard core processors can be incorporated to provide support for performance estimation of out-of-order soft core processors.

2. Hard Core Processor Performance Estimation for 64-bit Processor Cores

The performance estimation for dual issue, out-of-order hard core processors has been tested on the dual-issue, out-of-order, superscalar ARM Cortex-A9 32-bit processor. Noting that modern FPGA-based SoCs also support 64-bit ARM Cortex-A53, an in-order hard core processor, it will be of interest to further extend the proposed techniques for 64-bit in-order processors such as the ARM Cortex-A53. Techniques proposed in Chapter 3 for in-order soft core processors should provide valuable insights while estimating application performance on the in-order ARM Cortex-A53 hard core processor.

3. Automating Performance-aware Memory Allocation

The methodology proposed in this work takes into account memory dependency analysis to recommend the most suitable blocks for hardware acceleration in a given application. However, implementation on the CSoC devices require the creation of a memory map and allocation of memory. The memory map provides the addresses for the software blocks to access the data stored in local memory components of the FPGA. At present, these components are obtained manually to facilitate implementation. It will be of interest to develop methods to automatically recommend memory block allocations along with the most suitable blocks for hardware acceleration.

4. Automating Platform Interface Generation

The methodology proposed in this work is capable of recommending the most suitable blocks for hardware acceleration for a given area constraint and soft/hard core processor. However, the communication interface between hardware and software is being determined manually at present. It would be of interest to devise suitable methods to generate the hardware-software interface details automatically.

5. Accelerating Hardware Area Estimation Process for Large Applications

Even though, the hardware area estimation methodology described in Chapter 5 can be applied across different FPGA vendors and devices, it is the most time consuming step in the Wibheda+ framework proposed in this work due to its reliance on vendor specific synthesis tools. Noting that increase in application size can notably hinder the responsiveness of the automated framework, it is worth exploring if the hardware area estimation techniques based on mathematical models can be considered to accelerate this area estimation process without loss of estimation accuracy. The models should be developed in a vendor-agnostic manner to ensure its portability across different FPGA vendors and devices.

6. Incorporating Complex IP cores for Hardware Area-Time Estimation

Chapter 5, proposed a generic hardware estimation methodology by incorporating different vendor proprietary soft IP cores for floating point and memory functions. It would be of interest to extend this technique to support more complex and diverse IP cores. It is envisaged that machine learning based techniques could be considered for the automatic incorporation of complex IP cores. A set of candidate IP cores with different performance-area trade-offs could also be presented to the developer during design analysis to help make informed decisions.

7. Extending the Proposed Framework for MPSoC Platforms

Due to the emergence of MPSoC platforms, heterogeneous MPSoCs comprising different types of processors and accelerators are commonly employed in FPGA-based multiprocessor systems to obtain the best performance for an application [209]. It will be of interest to rely on the current design exploration framework Wibheda+ to support the constraint-aware mapping of applications on MPSoC. Noting that future CSoCs will consist of both hard and soft core processors, novel techniques must be devised to manage the efficient utilization of heterogeneous processor cores.

References

- [1] Handover. M. “Time = Money: Faster Time to Market with Formal Verification”, Mentor Graphics. [http://www.testandverification.com/DVClub/23_may_2013/MarkHandover\(Mentor\).pdf](http://www.testandverification.com/DVClub/23_may_2013/MarkHandover(Mentor).pdf), 2013. visited on 2016-09-20.
- [2] Vincent Tong. Opportunities and Challenges:28nm and 2.5/3-D IC Design and Manufacturing. http://www.arena-international.com/journals/2012/03/16/s/i/l/3_xilinx.pdf, 2012. visited on 2017-08-21.
- [3] Manfred Broy. Requirements Engineering for Embedded Systems. https://www4.in.tum.de/publ/papers/femsys_boesswet_1997_Conference.pdf, 1997. visited on 2016-09-20.
- [4] S2C FPGA PROTOTYPING SOLUTIONS. FPGA Market by Type, Verticals, Architecture, Technology Node, and Geography - Forecast to 2022. <http://www.s2cinc.com/company/news/2016/fpga-market-by-type-verticals-architecture-technology-node-and-geography-forecast-to-2022-prnewswire>, 2016. visited on 2017-01-20.
- [5] Anand Joshi. Embedded systems: Technologies and markets - IFT016E. <http://www.bccresearch.com/market-research/information-technology/embedded-systems-report-ift016e.html>, 2014. visited on 2016-06-20.
- [6] International Technology Roadmap for Semiconductors (ITRS). https://www.dropbox.com/sh/r51qrus06k6ehrc/AADDsRHdJICL7QH0_frSAU7Ha?n=390448867&oref=e, 2011. visited on 2017-09-20.
- [7] Christina A Hacker and Alain C Diebold. 2012 Updates to the International Technology Roadmap for Semiconductors (ITRS) Metrology. <https://www.nist.gov/node/581881>, 2013. visited on 2016-10-14.
- [8] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Design & Test of Computers*, 28(2):6–15, 2011.
- [9] Xilinx Corporation. cell-customer-innovation. <https://www.xilinx.com/publications/archives/xcell/Xcell-customer-innovation-2010.pdf>, 2010. visited on 2015-12-20.

- [10] Sergio Saponara and Luca Fanucci. Homogeneous and heterogeneous mp soc architectures with network-on-chip connectivity for low-power and real-time multimedia signal processing. *VLSI Design*, 2012:16:16–16:16, 2012.
- [11] WinterGreen Research. Programmable logic ICs by WinterGreen research. http://wintergreenresearch.com/reports/Programable_Logic_ICs.htm", 2010. visited on 2015-12-20.
- [12] Ty Garibay. Silicon Convergence: system solutions by merging coarse and fine grained programmable hardware. <http://sites.ieee.org/scv-cs/files/2012/09/Garibay-IEEE-0910121.pdf>, 2012. visited on 2015-12-20.
- [13] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M. C. Filho, and Vladimir Castro Alves. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. *J. VLSI Signal Process. Syst.*, 24(2-3):147–164, March 2000.
- [14] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. M. Rabaey. A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing. *IEEE Journal of Solid-State Circuits*, 35(11):1697–1704, Nov 2000.
- [15] Micron Technology. Micron Technology, Inc.: About Convey Computer Accelerator Products. <https://www.micron.com/about/about-convey-computer-accelerator-products>, 2018. visited on 2018-12-01.
- [16] Micron. Micron Technology, Inc. <https://www.micron.com/products/mx-series/>. visited on 2016-10-15.
- [17] Andreas Weichslgartner, Stefan Wildermann, Johannes Götzfried, Felix Freiling, Michael Glaß, and Jürgen Teich. Design-Time/Run-Time Mapping of Security-Critical Applications in Heterogeneous MPSoCs. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES '16*, 2016.
- [18] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, Feb 2007.
- [19] B. da Silva, A. Braeken, E. H. D'Hollander, A. Touhafi, J. G. Cornelis, and J. Lemeire. Comparing and combining GPU and FPGA accelerators in an image processing context. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, Sept 2013.
- [20] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Proceedings of the 2008 Symposium on Application Specific Processors, SASP '08*, Washington, DC, USA, 2008. IEEE Computer Society.

- [21] Rainer Keller, David Kramer, and Jan-Philipp Weiss. *Facing the Multicore-Challenge: Aspects of new paradigms and Aspects of New Paradigms and Technologies in Parallel Computing, volume 1*. Springer Science & Business Media, 10 2010.
- [22] Christian Pilato, Fabrizio Ferrandi, and Donatella Sciuto. A Design Methodology to Implement Memory Accesses in High-level Synthesis. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 49–58, New York, NY, USA, 2011. ACM.
- [23] Francisco Fons, Mariano Fons, Enrique Cantó, and Mariano López. Deployment of run-time reconfigurable hardware coprocessors into compute-intensive embedded applications. *Journal of Signal Processing Systems*, 66(2):191–221, 2011.
- [24] Xilinx Corporation. Zynq-7000 all programmable SoC. <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm>. visited on 2016-10-14.
- [25] Intel Corporation. Intel FPGA and SoC. <https://www.altera.com/>, 2017. visited on 2018-11-20.
- [26] J. Becker, A. Thomas, M. Vorbach, and V. Baumgarte. An industrial/academic configurable system-on-chip project (CSoC): coarse-grain XPP-/Leon-based architecture integration. In *Design, Automation and Test in Europe Conference and Exhibition*, 2003.
- [27] Sebastian Wallner. Design Methodology of a Configurable System-on-Chip Architecture. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '04, 2004.
- [28] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.
- [29] Rui Jia, Colin Yu Lin, Zhenhong Guo, Rui Chen, Fei Wang, Tongqiang Gao, and Haigang Yang. A survey of open source processors for FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014.
- [30] Intel Inc. Nios II Processor Reference Guide. https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf, 2016. visited on 2018-11-20.
- [31] Xilinx Corporation. MicroBlaze Soft Processor Core. <http://www.xilinx.com/products/design-tools/microblaze.html>. visited on 2017-11-22.

- [32] Cheah Hui Yan, Suhaib Fahmy, and Nachiket Kapre. On Data Forwarding in Deeply Pipelined Soft Processors. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, 2015.
- [33] Hui Yan Cheah, Fredrik Brosser, Suhaib A. Fahmy, and Douglas L. Maskell. The iDEA DSP Block-Based Soft Processor for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3), September 2014.
- [34] S. Vakili, J. M. P. Langlois, and G. Bois. Accuracy-aware processor customisation for fixed-point arithmetic. *IET Computers Digital Techniques*, 10(1):1–11, 2016.
- [35] Xilinx Corporation. Intellectual Property. <https://www.xilinx.com/products/intellectual-property.html>, 2016. visited on 2016-11-15.
- [36] Altera. Cyclone IV Device Handbook. https://www.altera.com/en_US/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf, 2016. visited on 2016-10-12.
- [37] David H. Jones, Adam Powell, Christos-Savvas Bouganis, and Peter Y. K. Cheung. GPU Versus FPGA for High Productivity Computing. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, 2010.
- [38] Andrew Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42, 10 2014.
- [39] Alok Prakash. *Design Methodologies for Generating Constraint-aware Configurable System-on-Chip*. PhD thesis, Nanyang Technological University, 2013.
- [40] K. Pocek, R. Tessier, and A. DeHon. Birth and adolescence of reconfigurable computing: a survey of the first 20 years of field-programmable custom computing machines. In *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013.
- [41] Yan Lin Aung. *Rapid design exploration framework for realizing custom computing systems on FPGAs*. PhD thesis, Nanyang Technological University, 2015.
- [42] Rajendra Patel and Arvind Rajawat. Recent Trends in Embedded System Software Performance Estimation. *Des. Autom. Embedded Syst.*, 17(1), March 2013.
- [43] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, 2001.

- [44] Marco Lattuada and Fabrizio Ferrandi. Performance modeling of embedded applications with zero architectural knowledge. *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES/ISSS '10*, 2010.
- [45] Marcio Seiji Oyamada, Felipe Zschornack, and Flávio Rech Wagner. Applying neural networks to performance estimation of embedded software. *Journal of Systems Architecture*, 54, 01 2008.
- [46] M. Lattuada and F. Ferrandi. Performance modeling of embedded applications with zero architectural knowledge. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010.
- [47] Brian Bailey. System Level Virtual Prototyping becomes a reality with OVP donation from Imperas. 2008.
- [48] Zhonglei Wang. *Software performance estimation methods for system-level design of embedded systems*. PhD thesis, Technical University of Munich, 2010.
- [49] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.
- [50] Team ArchC. <http://www.archc.org/>. visited on 2018-10-14.
- [51] Andrew Cagney. PSIM - Model of the PowerPC Architecture. <http://www.sourceware.org/psim/index.html>. visited on 2015-11-20.
- [52] Christopher Mills, Stanley C. Ahalt, and Jim Fowler. Compiled instruction set simulation. *Software: Practice and Experience*, 21, 08 1991.
- [53] V. Zivojnovic and H. Meyr. Compiled HW/SW co-simulation. In *33rd Design Automation Conference Proceedings*, pages 690–695, 1996.
- [54] Jwahar R. Bammi, Wido Kruijtzter, Luciano Lavagno, Edwin Harcourt, and Mihai T. Lazarescu. Software Performance Estimation Strategies in a System-level Design Tool. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, CODES '00, 2000.
- [55] T. Nakada, T. Tsumura, and H. Nakashima. Design and implementation of a workload specific simulator. In *39th Annual Simulation Symposium (ANSS'06)*, 2006.
- [56] Aimen Bouchhima, Patrice Gerin, and Frederic Petrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. *2009 Asia and South Pacific Design Automation Conference*, 2009.
- [57] Jong-Yeol Lee and In-Cheol Park. Timed compiled-code functional simulation of embedded software for performance analysis of soc design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(1):1–14, Jan 2003.

- [58] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A SW Performance Estimation Framework for Early System-level-design Using Fine-grained Instrumentation. In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings, DATE '06*, 2006.
- [59] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof. *ACM SIGPLAN Notices*, 17, 06 1982.
- [60] Inc. Free Software Foundation. GCC, the GNU compiler collection - GNU project - free software foundation (FSF). <https://gcc.gnu.org/>. visited on 2016-06-10.
- [61] Zhonglei Wang and Andreas Herkersdorf. Software performance simulation strategies for high-level embedded system design. *Performance Evaluation*, 67, 08 2010.
- [62] E. Cheung, H. Hsieh, and F. Balarin. Framework for fast and accurate performance simulation of multiprocessor systems. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, 2007.
- [63] Lei Gao, Jia Huang, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. TotalProf: A Fast and Accurate Retargetable Source Code Profiler. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS*, pages 305–314, 2009.
- [64] Trevor Meyerowitz, Alberto Sangiovanni-Vincentelli, Mirko Sauermaun, and Dominik Langen. Source-level timing Annotation and simulation for a heterogeneous Multiprocessor. *Design, Automation and Test in Europe*, 2008.
- [65] Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, and Wolfgang Rosenstiel. High-performance timing simulation of embedded software. *Proceedings of the 45th annual conference on Design automation - DAC '08*, 2008.
- [66] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate Retargetable Performance Estimation at the Transaction Level. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '08*, 2008.
- [67] Z. Wang, K. Lu, and A. Herkersdorf. An approach to improve accuracy of source-level TLMs of embedded software. In *2011 Design, Automation Test in Europe*, 2011.
- [68] Imperas Software Ltd. Why use virtual platforms? <http://www.imperas.com/why-use-virtual-platforms>, 2008. visited on 2016-06-10.
- [69] OSUOSL. QEMU. <http://www.qemu.org/>, 2016. visited on 2016-06-10.
- [70] Xilinx Wiki - QEMU. <http://www.wiki.xilinx.com/QEMU?responseToken=0f8b04787559f57a3bb2459b9ec916fcf>. visited on 2016-06-10.

- [71] Open Virtual Platforms. Virtual Platforms | Open Virtual Platforms. http://www.ovpworld.org/about_virtualplatforms. visited on 2016-06-10.
- [72] Imperas Software Ltd. Revolutionizing Embedded Software Development. <http://www.imperas.com/>. visited on 2016-06-10.
- [73] Cadence Design Systems. Virtual system platform. http://www.cadence.com/products/sd/virtual_system/pages/default.aspx, 2016. visited on 2016-06-10.
- [74] Synopsys. Synopsys virtual prototyping solutions for embedded software development, 2016. visited on 2016-10-14.
- [75] Y. L. Aung, S. K. Lam, and T. Srikanthan. Compiler-assisted technique for rapid performance estimation of FPGA-based processors. In *SOC Conference (SOCC), 2011 IEEE International*, 2011.
- [76] A. Powell, C. S. Bouganis, and P. Y. K. Cheung. Early performance estimation of image compression methods on soft processors. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.
- [77] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):3:1–3:37, May 2009.
- [78] D. Ivošević, N. Frid, and V. Sruk. Function-level performance estimation for heterogeneous mpsoc platforms. In *2016 Zooming Innovation in Consumer Electronics International Conference (ZINC)*, pages 38–41, June 2016.
- [79] Roman Lysecky and Frank Vahid. Design and Implementation of a MicroBlaze-based Warp Processor. *ACM Trans. Embed. Comput. Syst.*, 8(3), April 2009.
- [80] Cheah Hui Yan. *The iDEA Architecture-Focused FPGA Soft Processor*. PhD thesis, Nanyang Technological University, 2016.
- [81] Sven-Ake Andersson. Four soft-core processors for embedded systems. http://www.eetimes.com/document.asp?doc_id=1280290, August 2013. visited on 2016-12-01.
- [82] R. Shendi. CUSTOMIZATION OF A SOFT-CORE CPU ON AN FPGA. Master's thesis, University of Manchester, 2015.
- [83] Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan. The Microarchitecture of FPGA-based Soft Processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '05*, 2005.
- [84] Anupam Chattopadhyay, Heinrich Meyr, and Rainer Leupers. Chapter 5 - lisa: A uniform adl for embedded processor modeling, implementation, and software toolsuite generation. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 95 – 132. Morgan Kaufmann, Burlington, 2008.

- [85] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: a language for architecture exploration through compiler/simulator retargetability. In *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, pages 485–490, March 1999.
- [86] Deshya Wijesundera, Alok Prakash, Siew Kei Lam, and Thambipillai Srikanthan. Exploiting Configuration Dependencies for Rapid Area-efficient Customization of Soft-core Processors. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES '16*, 2016.
- [87] W. E. Dougherty, D. J. Pursley, and D. E. Thomas. Instruction subsetting: Trading power for programmability. In *Proceedings IEEE Computer Society Workshop on VLSI'98 System Level Design (Cat. No.98EX158)*, pages 42–47, April 1998.
- [88] Bryan Lewis and Ganesh Ramamoorthy. Market trends: ASIC design starts. <https://www.gartner.com/doc/919712/market-trends-asic-design-starts>, 2009. visited on 2015-12-20.
- [89] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen. Application-Specific Customization of Parameterized FPGA Soft-Core Processors. In *IEEE/ACM International Conference on Computer Aided Design*, 2006.
- [90] Intel Corporation. Nios II Classic Processor Reference Guide. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf. visited on 2017-03-20.
- [91] S. Padmanabhan, R. K. Cytron, R. D. Chamberlain, and J. W. Lockwood. Automatic application-specific microarchitecture reconfiguration. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006.
- [92] S. Rajotte, D. Carolina Gil, and J. M. P. Langlois. Combining ISA extensions and subsetting for improved ASIP performance and cost. In *IEEE International Symposium of Circuits and Systems (ISCAS)*, 2011.
- [93] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Exploration and customization of fpga-based soft processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26, 02 2007.
- [94] A. Prakash, S. K. Lam, A. K. Singh, and T. Srikanthan. Rapid design exploration framework for application-aware customization of soft core processors. In *International Conference on Field Programmable Logic and Applications*, pages 539–542, Aug 2009.
- [95] David Sheldon, Frank Vahid, and Stefano Lonardi. Interactive Presentation: Soft-core Processor Customization Using the Design of Experiments Paradigm. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '07*, 2007.

- [96] SigmaZone. DOE PRO. <http://www.sigmazone.com/doepro.htm>, 2013. visited on 2016-11-15.
- [97] Lieu My Chuong, Siew Kei Lam, and Thambipillai Srikanthan. Rapid area-time estimation technique for porting c-based applications onto fpga platforms. *Scalable Computing: Practice and Experience*, 8, 01 2007.
- [98] M. Rim and R. Jain. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13, 04 1994.
- [99] Minjoong Rim and R. Jain. Estimating performance characteristics of loop transformations. In *Proceedings of IEEE International Symposium on Circuits and Systems - ISCAS '94*, 1994.
- [100] S. Chaudhuri and R.A. Walker. Computing lower bounds on functional units before scheduling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4, 06 1996.
- [101] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Accurate area and delay estimators for FPGAs. In *Design, Automation and Test in Europe Conference and Exhibition. Proceedings*, 2002.
- [102] Changchun Shi, James Hwang, Scott McMillan, Ann Root, and Vinay Singh. A system level resource estimation tool for fpgas. In *Field Programmable Logic and Application*. Springer Science + Business Media, 2004.
- [103] C. Brandolese, W. Fornaciari, and F. Salice. An area estimation methodology for FPGA based designs at systemc-level. In *41st Design Automation Conference.*, pages 129–132, July 2004.
- [104] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Compile-time area estimation for lut-based fpgas. *ACM Transactions on Design Automation of Electronic Systems*, 11, 01 2006.
- [105] T. Todman and W. Luk. Reconfigurable Design Automation by High-Level Exploration. In *IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 185–188, July 2012.
- [106] Roel Meeuws, S. Arash Ostadzadeh, Carlo Galuzzi, Vlad Mihai Sima, Razvan Nane, and Koen Bertels. Quipu: A Statistical Model for Predicting Hardware Resources. *ACM Trans. Reconfigurable Technol. Syst.*, 6(1), 2013.
- [107] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2017.
- [108] Per Bjurus, Mikael Millberg, and Axel Jantsch. FPGA Resource and Timing Estimation from Matlab Execution Traces. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign, CODES '02*, 2002.

- [109] P. Schumacher and P. Jha. Fast and accurate resource estimation of RTL-based designs targeting FPGAs. In *International Conference on Field Programmable Logic and Applications*, 2008.
- [110] J. M. P. Cardoso. On estimations for compiling software to FPGA-based systems. In *IEEE International Conference on Application-Specific Systems, Architecture Processors*, 2005.
- [111] S. Bilavarn, G. Gogniat, J.-L. Philippe, and L. Bossuet. Design space pruning through early estimations of area/delay tradeoffs for fpga implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25, 10 2006.
- [112] Brad Quinton et al. Verific Design Automation: System Verilog, VHDL and UPF Parser Platforms. <http://www.verific.com/>, 2016. visited on 2016-06-10.
- [113] Min Xu and Fadi Kurdahi. Area and Timing Estimation for Lookup Table Based FPGAs. In *Proceedings of the European Conference on Design and Test*, EDTC '96, 1996.
- [114] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for micro-controllers. *IEEE Design & Test of Computers*, 10, 12 1993.
- [115] A. Prakash, S. K. Lam, T. Srikanthan, and C. T. Clarke. Modelling communication overhead for accessing local memories in hardware accelerators. In *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 31–34, June 2013.
- [116] Tom Brooks. Hardware-Software Partitioning in Embedded Systems|Barr Group. <http://bit.ly/2fdqtbL>, 2015. visited on 2016-06-10.
- [117] W. Zuo, L. N. Pouchet, A. Ayupov, T. Kim, Chung-Wei Lin, S. Shiraishi, and D. Chen. Accurate high-level modeling and automated hardware/software co-design for effective SoC design space exploration. In *54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.
- [118] P. Arato, S. Juhasz, Z. A. Mann, A. Orban, and D. Papp. Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing*, 2003, pages 197–202, Sept 2003.
- [119] Markus Weinhardt. Integer programming for partitioning in software oriented codesign. In *Field-Programmable Logic and Applications*. Springer Science + Business Media, 1995.
- [120] Ralf Niemann. *Hardware/Software CO-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
- [121] Jigang Wu and Thambipillai Srikanthan. Low-complex dynamic programming algorithm for hardware/software partitioning. *Inf. Process. Lett.*, 98(2):41–46, April 2006.

- [122] J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen, and A. Haxthausen. LYCOS: the Lyngby Co-Synthesis System. *Design Automation for Embedded Systems*, 2(2):195–235, Mar 1997.
- [123] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. *Design Automation for Embedded Systems*, 2(1):5–32, Jan 1997.
- [124] Jigang Wu, Pu Wang, Siew-Kei Lam, and Thambipillai Srikanthan. Efficient heuristic and tabu search for hardware/software partitioning. *The Journal of Supercomputing*, 66, 02 2013.
- [125] J. Henkel and R. Ernst. 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, 04 2001.
- [126] Theerayod Wiangtong, Peter Y. K. Cheung, and Wayne Luk. Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign. *Design Autom. for Emb. Sys.*, 6:425–449, 2002.
- [127] Axel Jantsch, Peeter Ellervee, Ahmed Hemani, Johnny Öberg, and Hannu Tenhunen. Hardware/Software Partitioning and Minimizing Memory Interface Traffic. In *Proceedings of the Conference on European Design Automation*, EURO-DAC, pages 226–231, 1994.
- [128] M. B. Abdelhalim and S. E.-D. Habib. An integrated high-level hardware/software partitioning methodology. *Design Automation for Embedded Systems*, 15(1):19–50, 02 2011.
- [129] Weijia Li, Lanying Li, Jianda Sun, Zhiqiang Lv, and Fei Guan. Hardware/Software Partitioning of Combination of Clustering Algorithm and Genetic Algorithm. In *International Journal of Control and Automation*, volume 7, pages 347–356, 01 2014.
- [130] Yiming Jing, Jishun Kuang, Jiayi Du, and Biao Hu. Application of improved simulated annealing optimization algorithms in hardware/software partitioning of the reconfigurable system-on-chip. In *Communications in Computer and Information Science*. Springer Science + Business Media, 2014.
- [131] K. Garg, Y. L. Aung, S. K. Lam, and T. Srikanthan. KnapSim - Run-time efficient hardware-software partitioning technique for FPGAs. In *28th IEEE International System-on-Chip Conference (SOCC)*, 2015.
- [132] Xibin Zhao, Hehua Zhang, Yu Jiang, Songzheng Song, Xun Jiao, and Ming Gu. An effective heuristic-based approach for partitioning. *Journal of Applied Mathematics*, 2013:1–8, 2013.
- [133] Y. Jiang, H. Zhang, X. Jiao, X. Song, W. N. N. Hung, M. Gu, and J. Sun. Uncertain Model and Algorithm for Hardware/Software Partitioning. In *IEEE Computer Society Annual Symposium on VLSI*, 2012.

- [134] S. Yousuf and A. Gordon-Ross. An Automated Hardware/Software Co-Design Flow for Partially Reconfigurable FPGAs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016.
- [135] C. Zhang, Y. Ma, and W. Luk. HW/SW Partitioning Algorithm Targeting MP-SOC with Dynamic Partial Reconfigurable Fabric. In *14th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics)*, 2015.
- [136] J. W. Tang, Y. W. Hau, and M. Marsono. Hardware/software partitioning of embedded System-on-Chip applications. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2015.
- [137] Fadi Obeidat. *Algorithms for High Performance Hardware Software Partitioning*. PhD thesis, International Institute of Information Technology, 2016.
- [138] Theo Kluter, Philip Brisk, Paolo Ienne, and Edoardo Charbon. Speculative DMA for Architecturally Visible Storage in Instruction Set Extensions. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '08, pages 243–248, New York, NY, USA, 2008. ACM.
- [139] Theo Kluter, Samuel Burri, Philip Brisk, Edoardo Charbon, and Paolo Ienne. Virtual ways: Low-cost coherence for instruction set extensions with architecturally visible storage. *ACM Transactions on Architecture and Code Optimization*, 11, 07 2014.
- [140] Partha Biswas, Nikil D. Dutt, Laura Pozzi, and Paolo Ienne. Introduction of architecturally visible storage in instruction set extensions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26, 03 2007.
- [141] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26, 11 2007.
- [142] A. Prakash, S. K. Lam, T. Srikanthan, and C. T. Clarke. Modelling communication overhead for accessing local memories in hardware accelerators. In *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013.
- [143] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel. mRTS: Run-time system for reconfigurable processors with multi-grained instruction-set extensions. In *Design, Automation Test in Europe*, 2011.
- [144] A. Prakash, C. T. Clarke, S. Lam, and T. Srikanthan. Rapid Memory-Aware Selection of Hardware Accelerators in Programmable SoC Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):445–456, March 2018.

- [145] Michael Barr and O ' Reilly. Programming Embedded Systems in C and C++ | Embedded Systems Experts. <http://www.barrgroup.com/Embedded-Systems/Books/Programming-Embedded-Systems>. visited on 2016-06-10.
- [146] B Moyer. *Real World Multicore Embedded Systems*. Newnes, 01 2013.
- [147] GAUT. <http://www-labsticc.univ-ubs.fr/www-gaut/>. visited on 2018-10-14.
- [148] Apache 2. <http://roccc.cs.ucr.edu/>. visited on 2016-11-15.
- [149] mminutoli 84. Panda.dei.polimi.it. http://panda.dei.polimi.it/wp-content/uploads/posterub_date.pdf, 2014. visited on 2018-10-14.
- [150] W. Luk, J. G. F. Coutinho, T. Todman, Y. M. Lam, W. Osborne, K. W. Susanto, Q. Liu, and W. S. Wong. A high-level compilation toolchain for heterogeneous systems. In *IEEE International SOC Conference (SOCC)*, 2009.
- [151] M. Rashid, F. Ferrandi, and K. Bertels. hArtes design flow for heterogeneous platforms. In *10th International Symposium on Quality Electronic Design*, 2009.
- [152] Georgi Kuzmanov, Raffaele Tripiccion, Giacomo Marchiori, and Immacolata Colacicco. The hartes platform. In *Hardware/Software Co-design for Heterogeneous Multi-core Platforms*. Springer Science + Business Media, 2012.
- [153] F. Capman, S. Thabuteau, and A. M. Fouillart. In-car speech and audio processing - some experiments within hartes project. In *International Conference on Networking, Sensing and Control (ICNSC)*, 2009.
- [154] BlueBee Multi-Core Technologies The Gateway to Multi-Core Heterogeneous Computing Platforms | STW. <http://www.stw.nl/en/content/bluebee-multi-core-technologies-gateway-multi-core-heterogeneous-computing-platforms>. visited on 2016-11-15.
- [155] University of Toronto. LegUp High-Level Synthesis. <http://legup.eecg.utoronto.ca>. visited on 2018-12-01.
- [156] Cadence. http://www.tensilica.com/uploads/pdf/xpres-triple-threat_solution.pdf. visited on 2016-8-13.
- [157] Synopsys. Synopsys.com, 2016. visited on 2016-10-14.
- [158] Mentor Graphics. The EDA Technology leader, 10 2016. visited on 2016-10-14.
- [159] R.E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20, 2000.
- [160] Impulse Accelerated Technologies - Home. <http://www.impulsec.com/>. visited on 2016-11-16.
- [161] NEC. Cyberworkbench: Products. <http://www.nec.com/global/prod/cwb/index.html>. visited on 2016-10-14.

- [162] Y Explorations. Yxi products - excite. <http://www.yxi.com/products.php>. visited on 2016-06-10.
- [163] Space Codesign Systems Inc. Introducing the SpaceStudio development environment. <http://www.spacecodesign.com/>, 2018. visited on 2018-12-20.
- [164] Xilinx Triton Solutions. Triton Tool Suite. <http://www.poseidon-systems.com/XilinxTriton.htm>, 2015. visited on 2016-06-10.
- [165] Xilinx Corporation. SDSoc Development Environment. <https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html>, 2018. visited on 2018-12-20.
- [166] Deshya Wijesundera, Alok Prakash, Thambipillai Srikanthan, and Achintha Ihalage. Framework for Rapid Performance Estimation of Embedded Soft Core Processors. *ACM Trans. Reconfigurable Technol. Syst.*, 11(2):9:1–9:21, July 2018.
- [167] D. Wijesundera, A. Prakash, and T. Srikanthan. Rapid design space exploration for soft core processor customization and selection. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 185–188, Dec 2016.
- [168] LLVM. The LLVM Compiler Infrastructure. <http://llvm.org/>. visited on 2018-11-20.
- [169] Abhijit Ray. *Methods for rapid selection of processors for constraint-aware embedded systems*. PhD thesis, Nanyang Technological University, 2008.
- [170] Yakun Sophia Shao and David Brooks. ISA-Independent Workload Characterization and its Implications for Specialized Architectures. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.
- [171] Jan Gray. ReConFig Keynote; The Past and Future of FPGA Soft Processors. <http://fpga.org/wp-content/uploads/2014/12/reconfig-14-the-past-and-future-of-fpga-soft-processors.pdf>, December 2014. visited on 2016-12-01.
- [172] Xilinx Corporation. 7 Series FPGAs Overview. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. visited on 2017-04-19.
- [173] Intel Corporation. Cyclone V Device Overview. https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf. visited on 2018-09-20.
- [174] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *IEEE International Symposium on Circuits and Systems*, 2008.
- [175] Intel Corporation. Stratix II vs. Virtex-4 Density Comparison. https://www.altera.com/en_US/pdfs/literature/wp/wpstxiixlnx.pdf, 2005. visited on 2017-09-20.

- [176] Guanwen Zhong et al. Design Space Exploration of FPGA-based Accelerators with Multi-level Parallelism. In *DATE*, 2017.
- [177] Deshya Wijesundera, Achintha Ihalage, Alok Prakash, and Thambipillai Srikanthan. High Speed Performance Estimation of Embedded Hard-core Processors in FPGA-based SoCs. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART2017, 2017.
- [178] ARM Ltd. Cortex-A9 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf. visited on 2016-12-01.
- [179] LLVM Language Reference Manual. <http://llvm.org/docs/LangRef.html>. visited on 2018-12-01.
- [180] Gurpur Prabhu. Data Hazard Classification. <https://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/dataHazClass.html>. visited on 2017-12-01.
- [181] James A. Langbridge. *Professional Embedded ARM Development*. Wiley, 2013.
- [182] Yoav Etsion et al. Computer Architecture - Out-of-order Execution. <https://webcourse.cs.technion.ac.il/234267/Spring2014/ho/WCFiles/5-ca-oooe.pdf>, 2014. visited on 2016-12-01.
- [183] Intel Corporation. Nios SOC FPGAs. <https://www.intel.com/content/www/us/en/products/programmable/soc.html>. visited on 2017-11-10.
- [184] Zed Board Hardware User's Guide. http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf. visited on 2016-12-01.
- [185] Christopher Celio. Kayla (1.4 GHz ARM Cortex A9). [https://github.com/ucbar/ccbench/wiki/Kayla-\(1.4-GHz-ARM-Cortex-A9\)](https://github.com/ucbar/ccbench/wiki/Kayla-(1.4-GHz-ARM-Cortex-A9)). visited on 2018-12-01.
- [186] Xilinx Inc. EDK Profiling User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/edk_prof.pdf, 2009. visited on 2016-12-01.
- [187] University of Toronto. Frequently Asked Questions — LegUp 3.0 documentation. <http://legup.eecg.utoronto.ca/docs/3.0/faq.html>. visited on 2018-12-01.
- [188] Verilog Coding Tips and Tricks. Verilog Coding Tips and Tricks: Synthesisable Verilog code for Division of two binary numbers. <http://verilogcodes.blogspot.com/2015/11/synthesisable-verilog-code-for-division.html>. visited on 2018-08-28.
- [189] Doulos. Simple RAM Model. https://www.doulos.com/knowhow/vhdl_designers_guide/models/simple_ram_model/. visited on 2018-08-28.
- [190] J Dawson. Synthesiseable IEEE 754 floating point library in Verilog. <https://github.com/dawsonjon/fpu>. visited on 2018-12-01.

- [191] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17, 2009.
- [192] Deshya Wijesundera, Alok Prakash, Thilina Perera, Kalindu Herath, and Tham-bipillai Srikanthan. Wibheda+: Framework for Data Dependency-aware Multi-constrained Hardware-Software Partitioning in FPGA-based SoCs for IoT Applications. In *Proceedings of the 9th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART2018, 2018.
- [193] Partha Biswas, Vinay Choudhary, Kubilay Atasu, Laura Pozzi, Paolo Ienne, and Nikil Dutt. Introduction of Local Memory Elements in Instruction Set Extensions. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 729–734, New York, NY, USA, 2004. ACM.
- [194] Peter Wilson. Chapter 2 - An FPGA Primer. In *Design Recipes for FPGAs*. Newnes, Oxford, second edition edition, 2016.
- [195] Jennifer J. Burg, John Ainsworth, Brian Casto, and Sheau-Dong Lang. Experiments with the “oregon trail knapsack problem”. *Electronic Notes in Discrete Mathematics*, 1:26 – 35, 1999. CP98, Workshop on Large Scale Combinatorial Optimisation and Constraints.
- [196] Lattice Semiconductor. iCE40 Ultra/UltraLite/UltraPlus-Lattice Semiconductor. <http://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40Ultra.aspx>, 2017. visited on 2018-12-01.
- [197] Lattice Semiconductor. LatticeECP2/M-Lattice Semiconductor. <http://www.latticesemi.com/Products/FPGAandCPLD/LatticeECP2M.aspx>, 2017. visited on 2018-12-01.
- [198] Intel. Intel Cyclone 10 LP FPGAs Product Table. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/pt/cyclone-10-lp-product-table.pdf, 2017. visited on 2018-12-01.
- [199] Intel Corporation. Nios ii floating point hardware 2 component user guide. Technical report, 2015. visited on 2016-10-20.
- [200] Intel Corporation. Altera Design Software. <https://www.altera.com/support/support-resources/support-centers/sof-index.html>, 2016. visited on 2016-10-22.
- [201] Intel Corporation. The world’s most versatile embedded processor. <http://www.semiconductorstore.com/pages/asp/item.asp?itemnumber=NIOS%20II>, 2016. visited on 2017-11-20.
- [202] Christian Plessl. Introduction to the LLVM Compiler Framework. Technical report, High-Performance IT Systems group, Paderborn University, 2014. visited on 2018-11-20.

- [203] Mentor Graphics. ModelSim ASIC and FPGA Design. <https://www.mentor.com/products/fv/modelsim/>. visited on 2017-11-20.
- [204] Intel. Intel FPGA and SoC. <https://www.altera.com/products/soc/overview.html>, 2018. visited on 2018-12-01.
- [205] Xilinx. Expanding the All Programmable SoC Portfolio. <https://www.xilinx.com/products/silicon-devices/soc.html>, 2018. visited on 2018-12-01.
- [206] D. Wijesundera, A. Prakash, T. Perera, K. Herath, and T. Srikanthan. Wibheda: Framework for data dependency-aware multi-constrained hardware-software partitioning in fpga-based socs for iot devices. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018.
- [207] Trishul M. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01*, pages 191–202, New York, NY, USA, 2001. ACM.
- [208] Henry Ting-Hei Wong. *A Superscalar Out-of-Order x86 Soft Processor for FPGA*. PhD thesis, University of Toronto, 2017.
- [209] Taho Dorta, Jaime Jiménez, José Luis Martín, Unai Bidarte, and Armando Astarloa. Overview of FPGA-Based Multiprocessor Systems. In *Proceedings of the 2009 International Conference on Reconfigurable Computing and FPGAs, RECONFIG '09*, 2009.