FPGA-AWARE CUSTOM INSTRUCTIONS FOR RECONFIGURABLE INSTRUCTION SET PROCESSORS

LAM SIEW KEI

School of Computer Engineering

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

2011

ACKNOWLEDGEMENTS

First and foremost, I wish to express my heartfelt gratitude to my supervisor, Prof. Thambipillai Srikanthan, for his guidance and understanding that has made it possible for me to complete my Ph.D. on a part-time basis. I have gained much from his invaluable insights and ideas that are instrumental to the contributions made in this thesis. I am also grateful for all the opportunities, advice and encouragement that he has given me, which has helped me immensely in my research and profession.

I would like to direct my sincere appreciation to Dr. Christopher T. Clarke who always took the time to understand the problems that I faced during my Ph.D. and provide sound suggestions. He is always willing to share his knowledge and I have learnt much from the discussions we had during his short trips here.

I would also like to express my appreciation to all the staff and students that I have worked with at the Centre for High Performance Embedded Systems (CHiPES). The exchange of ideas during our discussions has led to numerous interesting research ideas. Special thanks to Dr. Wu Jigang for spending the time to explain and proof-read the mathematical formulations and proofs in my research publications.

I would also like to extend my appreciation to Ms. Nah Kiat Joo, Ms. Merilyn Yap Lee Peng and Mr. Chua Ngee Tat for their technical and logistic assistance. I also thank the staff and students at CHiPES for making my work here a fulfilling experience.

Finally, I am forever grateful to my family and close friends for always being there for me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADD	Addition
ALU	Arithmetic Logic Unit
AMT	All templates with data-path merging
ASIC	Application-Specific Integrated Circuit
AT	All templates without data-path merging
BLE	Basic Logic Element
BRL	Branch and Link
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CFT	Cluster Frequency Table
CFU	Custom Functional Unit
CGAU	Coarse Grained Arithmetic Unit
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
DFG	Data-Flow Graph
DFS	Depth First Search
EDA	Electronic Design Automation
eFPGA	Embedded FPGA
FF	Flip Flop
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
ILP	Integer Linear Programming
IP	Intellectual Property
IR	Intermediate Representation
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
LBAT	Logic Block Activity Table
LFF	Largest-Fit-First

LUT	Look-Up Table
MFF	Most-Frequently-Fit-First
MIS	Maximum Independent Set
MLFF	Most-Frequent-Largest-Fit-First
MSB	Most Significant Bit
MT	Merged Templates
MUL	Multiplication
MUX	Multiplexer
Non-MT	Non-Merged Templates
NRE	Non-Recurring Engineering
PE	Processing Element
RFU	Reconfigurable Functional Unit
RLFF	LFF algorithm with refinements
RISP	Reconfigurable Instruction Set Processor
RMT	Reduced templates with data-path merging
RT	Reduced templates without data-path merging
RTL	Register Transfer Level
RTR	Runtime Reconfiguration
SHL	Logical left shift
SHR	Logical right shift
SHRA	Arithmetic right shift
SOC	System-On-a-Chip
SUB	Subtraction
TTM	Time-To-Market
VHDL	Very-High-Speed Integrated Circuit HDL
VLIW	Very Long Instruction Word

ABSTRACT

It is evident that future embedded systems will continue to demand a higher degree of customization and design flexibility without compromising the Time-To-Market (TTM), and lower Non Recurring Engineering (NRE) costs. In this thesis, techniques for the automatic generation of profitable custom instructions for FPGA based Reconfigurable Instruction Set Processors (RISPs) have been proposed. A detailed literature review was undertaken to establish the shortcomings in the existing work on RISPs. In particular, challenges in the selection, hardware estimation, area-time optimization and runtime reconfiguration of custom instructions for FPGA based RISPs have been established.

A method for the selection of custom instructions has been proposed and compared with the existing ones reported in the literature. The proposed technique based on Largest-Fit-First (LFF) has been shown to yield large custom instructions that are capable of representing a number of frequently executed ones. It was shown that the outputs generated using LFF can be further refined by considering the overlapping templates that were previously ignored. Performance evaluations show that the proposed technique outperforms the existing methods by up to 32%. Moreover, the proposed selection process can be realized in the order of milliseconds.

Techniques for the rapid estimation of critical path delays and area measures of custom instructions implemented on LUT based FPGAs have been devised. The proposed high level estimation technique relies on partitioning the custom instructions into a set of basic clusters to facilitate the systematic mapping onto FPGA logic blocks. Investigations using 150 custom instructions from sixteen applications show that the average critical paths can be estimated to be within 3% of those obtained using hardware synthesis. The average area measures have also been shown to be within 1% of those obtained using hardware synthesis. The proposed estimation process can be realized within a few milliseconds, thereby making it suitable for addressing area-time measures during the selection process.

An efficient technique for the selection of most profitable custom instructions to eliminate the least profitable ones without the need to rely on hardware synthesis has been devised by leveraging proposed rapid high-level estimation process. It has been shown that the rapid selection of a reduced set of custom instructions leads to high area efficiency with minimal performance gain penalties. Simulations based on sixteen applications from benchmark suites show that the proposed framework provides, on average, an area reduction of over 25% with negligible loss in compute performance. The average area-delay product gain can also be up to 86% as a direct result of choosing the most profitable custom instructions.

A technique was proposed for the FPGA architecture-aware merging of custom instructions to further reduce the area-delay product. The proposed strategy employs a heuristic based cluster merging process to maximize the utilization of FPGA logic blocks. Unlike the popular resource sharing method, the reliance on multiplexers is reduced as resource sharing is applied sparingly only at the final stage. We show that the proposed technique leads to an average area reduction of more than 34% for Spartan-3, Virtex-4 and Virtex-5 architectures when compared to the optimized outputs of the Xilinx synthesis tool. Moreover, it outperforms the most efficient resource sharing based method as average area-delay product reductions of more than 27%, 34% and 19% for Spartan-3, Virtex-4 and Virtex-5 respectively can be evidenced.

A hierarchical loop-aware partitioning strategy has also been proposed to reduce the complexity of the search space for determining the runtime configuration of custom instructions for RISPs. Experimental results based on simulations show that runtime reconfiguration can lead to an average performance gain of over 45% and 52% for the full reconfiguration and partial reconfiguration model respectively. In addition, it provides for over 39% average reduction in the runtime overhead for the partial reconfiguration model.

Finally, a framework to generate profitable custom instructions for the runtime configurations on RISPs has been presented. The proposed framework integrates all the salient techniques developed in this thesis for the generation of runtime reconfigurable custom instructions for high profitability and low reconfiguration overhead. An approach for the runtime management custom instructions on RISP has also been proposed to cater to the dynamic execution profile of the application, thereby highlighting the significance of the contributions made in this thesis.

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Embedded systems are becoming more pervasive and complex. While application-centric customization is widely recognized as a central if not essential driver for the proliferation of embedded systems, concomitant Non-Recurring Engineering (NRE) costs[1] and Time-To-Market (TTM)[2] concerns remain significant impediments. Enterprises offering products and services in embedded systems, in particular small and medium sized organizations, will succumb to technological and market challenges unless new design automation tools emerge to fill the design productivity gap. This thesis aims to devise solutions for bridging the design productivity gap[3] in embedded systems.

In recent years, customizable processors, in the form of configurable processors and Reconfigurable Instruction Set Processors (RISPs)[4], have emerged to bridge the full hardware and processor based implementation gap[5] in System-On-a-Chip (SOC)[6] design [4]. These customizable processors can be modified or extended to address specific

[1] NRE cost is the one-time monetary cost of designing the system [1]
[2] TTM indicates the duration to develop a system to the point that it can be sold to customers [1]
[3] The design productivity gap arises when the number of available transistors grow faster than the ability to meaningfully design them [2]
[4] RISPs are programmable processors that are tightly coupled with reconfigurable logic. At runtime, instructions are issued to the standard functional units of the processor and the reconfigurable unit [3]
[5] Implementation gap refers to the performance and power efficiency of dedicated hardware and the flexibility of software-based solutions
[6] SOC is a single chip that integrates all the necessary components of a computer or electronic circuitry for a complete system

design issues by changing the processor’s feature set to suit the application and constraints.

Configurable processor technologies require a one-time customization of the processing engine prior to manufacturing. Hence, this necessitates the need for new configurable processor architecture to be generated for each application[7] domain, or when new functionalities and updates are required in existing applications. Substantial NRE costs are incurred for the generation of new configurable processor architectures. In comparison, RISPs support post-manufacturing flexibility.

Similar to configurable processors, RISPs offer the possibility of extending the basic instruction set of the microprocessor by introducing custom functional units on the reconfigurable space (e.g. Field Programmable Gate Arrays (FPGAs)[8]) to implement custom instructions. A custom instruction typically encapsulates multiple primitive operations that constitute the critical portion of the application. The corresponding code segments associated with the custom instructions are implemented in hardware (e.g. Configurable Functional Unit (CFU) in configurable processors or Reconfigurable Functional Unit (RFU) in RISPs). During application runtime, the native instructions of the processor execute on the processor data-path, while the custom instructions invoke the execution of the hardware in order to accelerate the application's performance.

It is envisioned that future SOC platforms will incorporate RISP technology to leverage the computational power of hardware while providing for high instruction set programmability to meet the increasingly tight TTM requirements. In addition, these future platforms will need to incorporate dynamic reconfiguration strategies for increased

[7] An application is a program that performs a specific set of tasks/functions for the end-user or, in some cases, for another application
[8] Refer to Appendix A for an introduction to FPGAs

performance and cost efficiency by adapting to the application runtime characteristics. One of the key challenges to increase the proliferation of RISPs lies in the development of supporting compilation and design automation tools that enable rapid design exploration and efficient mapping of applications on such platforms.

Future embedded systems will require a higher degree of customization to manage the growing complexity of the applications. At the same time, they must continue to facilitate a high degree of flexibility to meet the shrinking TTM window. In light of this, there exists an urgent need to develop techniques that focus on increasing the design productivity of highly customized embedded systems, in order to improve product life cycles, and reduce TTM pressure and NRE costs. To this end, computing platforms such as RISPs provide a promising solution to realize a balanced trade-off between flexibility and customization. In this research, we focus on developing novel techniques to ensure that RISP becomes a viable alternative despite TTM and NRE pressures.

1.2 AIM OF THE PROJECT

The main aim of this research work is to develop novel techniques and design methodologies for automatically generating efficient custom instructions that are capable of optimizing the utilization of RISPs. In particular, the proposed methods must incorporate architecture-aware strategies to generate area-time efficient custom instructions without the need to undergo time consuming hardware design iterations. The proposed methods target RISPs with runtime reconfigurable support, which enables the reconfigurable resources to be reconfigured with the most profitable[9] custom instructions at runtime to maximize the utilization of the restricted reconfigurable space. It is envisioned that runtime reconfiguration of custom instructions can lead to quality design alternatives to satisfy the workload characteristics of embedded computations, while meeting the tight non-functional requirements of embedded systems.

1.3 MAIN CONTRIBUTIONS OF THIS THESIS

In this thesis, we have proposed novel techniques and design methodologies for realizing area-time efficient custom instructions for RISPs. In addition, we have also devised strategies to reduce the runtime reconfiguration overhead of custom instructions for area-constrained RISPs. The following lists the main contributions that have been made during the course of this research, which are documented in this thesis:

1. *An efficient strategy for the rapid selection of custom instructions based on a graph covering approach*. The proposed strategy is motivated by our investigations which reveal that a majority of frequently executed custom instructions are consumed by larger custom instructions. Comparisons with previously reported approximate strategies show that the proposed technique selects custom instructions with highest performance gain.
2. *A novel cluster generation technique to estimate critical path delays and area utilization of custom instructions on Look-Up Table (LUT) based FPGAs*. Unlike existing estimation approaches that do not incorporate architecture-aware

[9] Profitable custom instructions provide higher performance for a given reconfigurable area

strategies, the proposed high level estimation technique can reliably predict the mapping of custom instructions onto the basic logic elements of the target FPGA.

3. *A design exploration framework that can rapidly identify a reduced set of profitable custom instructions without the need for actual hardware synthesis*. The proposed framework leverages the cluster generation technique for rapid selection of a reduced set of custom instructions that leads to high area efficiency with negligible loss in the performance gain. The proposed strategy outperforms an existing area-optimization method that is based on maximizing the regularity of the custom instruction data-paths.
4. *A novel cluster merging strategy that takes into account the architectural constraints of the FPGA device in order to realize custom instructions with low area-delay product*. We show that the commonly used resource sharing approaches for area optimization may not lead to the best results for FPGA designs. The proposed cluster merging method outperforms the area optimization capabilities of the commercial tool and one of the most efficient methods reported in the literature for resource sharing.
5. *A hierarchical loop partitioning strategy that reduces the complexity of the search space for determining the runtime custom instruction configurations in area-constrained RISPs*. We show that the proposed hierarchical loop partitioning strategy, which utilizes the cluster merging results, can maximize the utilization of the reconfigurable resources in each configuration and significantly reduce the runtime reconfiguration overhead.

6. *An integrated framework to generate profitable custom instruction for RISPs with runtime reconfiguration support*. The proposed framework incorporates a systematic approach consisting of multiple optimization strategies to produce a set of configurations that can be implemented on the RISP with runtime reconfiguration support in an area-time efficient manner.
7. *A scheme for managing the runtime reconfiguration of custom instructions on a partially reconfigurable architecture*. The proposed scheme relies on the dynamic execution profile of the application to replace the functionality of the FPGA logic blocks at runtime with the goal of minimizing the overall reconfiguration overhead.

1.4 ORGANIZATION OF THIS THESIS

The rest of the thesis is organized as follows:

- **Chapter 2**: We first provide a literature review on the design challenges faced by current and future embedded system designers to manage the growing complexity of applications, and to meet the TTM and NRE pressures. We also provide detailed literature review to highlight the limitations of existing work on custom instruction generation for RISPs.
- **Chapter 3**: We present a method for selecting custom instructions from the application code. The proposed method relies on graph covering heuristics to select large custom instructions that typically consume smaller but frequently used custom instructions. The proposed method is compared with existing

techniques to demonstrate its superiority in selecting high performance custom instructions.

- **Chapter 4**: We propose a novel cluster generation strategy to estimate critical path delays of custom instructions on LUT based FPGAs. The proposed high level estimation technique partitions the custom instructions into smaller clusters that can be efficiently mapped onto the FPGA logic block. We demonstrate the necessity and effectiveness of the proposed delay estimation technique for evaluating the performance gain of custom instructions that are selected using the approaches discussed in Chapter 3. Detailed evaluations are also undertaken to demonstrate the accuracy of the proposed method for estimating area utilization of custom instructions on FPGA.
- **Chapter 5**: We focus on the selection of area-time efficient custom instructions, which provide higher performance for a given reconfigurable area. We propose a design exploration framework to rapidly identify a reduced set of profitable custom instructions without the need for actual hardware synthesis. The framework incorporates the cluster generation technique discussed in the previous chapter to facilitate rapid area-time estimation of custom instruction implementations on FPGA. We also show that the proposed strategy outperforms an existing area-optimization approach that relies on exploiting the regularity of custom instruction data paths.
- **Chapter 6**: We propose a novel strategy that takes into account the architectural constraints of the FPGA device in order to realize custom instructions with low area-delay product. The proposed cluster merging strategy leverages on the

cluster generation technique discussed in Chapter 4. We perform comparisons to show that the proposed technique leads to significantly higher area-time efficiency than a commercial tool and one of the most efficient area optimization methods reported in the literature.

- **Chapter 7**: We first present a target runtime RFU that incorporates coarse granularity FPGA logic blocks with low reconfiguration latency. We then introduce the proposed hierarchical loop partitioning strategy to generate runtime custom instruction configurations for area-constrained RISPs. We also show the significance of cluster merging (discussed in Chapter 6) for maximizing the performance benefits of runtime reconfiguration in RISPs.
- **Chapter 8**: We present a complete framework that integrates all the components discussed in the previous chapters to generate profitable custom instructions for RISPs with runtime reconfiguration support. A scheme for managing the runtime reconfiguration of custom instructions on a partially reconfigurable architecture that incorporates multi-bit FPGA logic blocks is also proposed in this chapter.
- **Chapter 9**: We conclude the thesis and identify some future directions in this work.

1.5 LIST OF PUBLICATIONS RESULTED FROM THIS THESIS

International Refereed Journals

[J-1] Lam S.K., Srikanthan T. and Clarke C.T., "Architecture-Aware Technique for Mapping Area-Time Efficient Custom Instructions onto FPGAs", IEEE Transactions on Computers, Vol. 60, No. 5, May 2011, pp. 680-692

[J-2] Li T., Wu J., Lam S.K. and Srikanthan T., "Selecting Profitable Custom Instructions for Reconfigurable Processors", Journal of Systems Architecture, Vol. 56, No. 8, August 2010, pp. 340-351

[J-3] Lam S.K., Srikanthan T. and Clarke C.T., "Selecting Profitable Custom Instructions for Area-Time-Efficient Realization on Reconfigurable Architectures", IEEE Transactions on Industrial Electronics, Vol. 56, No. 10, October 2009, pp. 3998-4005

[J-4] Lam S.K., Huang F., Srikanthan T. and Wu J., "Run-Time Management of Custom Instructions on a Partially Reconfigurable Architecture", International Journal of Information and Communication Technology, Vol. 2, No. 1/2, 2009, pp. 50-59

[J-5] Lam S.K and Srikanthan T., "Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing", Journal of Systems Architecture, Vol. 55, No. 1, January 2009, pp. 1-14

[J-6] Lam S.K., Srikanthan T. and Clarke C.T., "Rapid Generation of Custom Instructions Using Predefined Dataflow Structures", Microprocessors and Microsystems (Special Issue on FPGA-based Reconfigurable Computing), Vol. 30, No. 6, September 2006, pp. 355-366

International Refereed Conferences

[C-1] Lam S.K., Deng Y., Hu J., Zhou X. and Srikanthan T., "Hierarchical Loop Partitioning for Rapid Generation of Runtime Configurations", Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin Heidelberg: Reconfigurable Computing: Architectures, Tools and Applications (6th International Symposium on Applied Reconfigurable Computing (ARC)), Vol. 5992/2010, March 2010, pp. 282-293

[C-2] Prakash A., Lam S.K., Singh A.K. and Srikanthan T., "Architecture-Aware Custom Instruction Generation for Reconfigurable Processors", Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin Heidelberg: Reconfigurable Computing: Architectures, Tools and Applications (6th International Symposium on Applied Reconfigurable Computing (ARC)), Vol. 5992/2010, March 2010, pp. 414-419

[C-3] Li T., Wu J., Lam S.K., Srikanthan T. and Lu X., "Efficient Heuristic Algorithm for Custom-Instruction Selection", IEEE/ACIS International Conference on Computer and Information Science (ICIS), June 2009

[C-4] Lam S.K., Huang F., Srikanthan T. and Wu J., "Run-Time Management of Custom Instructions on a Partially Reconfigurable Architecture", IEEE International Conference on Electronic Design (ICED), December 2008 (*Awarded Certificate of Merit for Best 15 Papers out of 170 Accepted Papers*)

[C-5] Lam S.K. and Srikanthan T., "Selection of Area-Time Efficient Custom Instructions for FPGA Realization", IEEE International Symposium on Industrial Electronics (ISIE), June/July 2008, pp. 1704-1709

[C-6] Lam S.K., Li W. and Srikanthan T., "High Level Area Estimation of Custom Instructions for FPGA-based Reconfigurable Processors", Sixth International Conference on Information, Communications and Signal Processing (ICICS), December 2007, pp. 1-5

[C-7] Lam S.K. and Srikanthan T., "Estimating Area Costs of Custom Instructions for FPGA-based Reconfigurable Processors", IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), July 2007, pp. 89-94

[C-8] Lam S.K., Krishnan B.N. and Srikanthan T., "Efficient Management of Custom Instructions for Run-Time Reconfigurable Instruction Set Processors", IEEE International Conference on Field-Programmable Technology (FPT), December 2006

[C-9] Lam S.K., Shoaib M. and Srikanthan T., "Modeling Arbitrator Delay-Area Dependencies in Customizable Instruction Set Processors", IEEE Third International Workshop on Electronic Design, Test and Applications (DELTA), January 2006, pp. 237-242

[C-10] Lam S.K., Deng Y. and Srikanthan T., "Morphable Structures for Reconfigurable Instruction Set Processors", Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin Heidelberg: Advances in Computer Systems Architecture (Tenth Asia-Pacific Computer Systems Architecture Conference (ACSAC)), Vol. 3740, October 2005, pp. 450-463

CHAPTER 2

LITERATURE REVIEW

The pervasiveness of embedded systems in human life today is unprecedented. We are constantly using embedded devices in every aspect of our life, often being unaware of the underlying technology. This phenomenon will continue to grow with consumers being exposed to new embedded devices with innovative end-applications in the global market on a daily basis. According to a recent technical market research report**,** "Embedded Systems: Technologies and Market" from BCC Research, the global market for embedded systems technologies which was worth $92.0 billion in 2008 will increase to $112.5 billion in 2013, with a compound annual growth rate of 4.1% [5].

Embedded applications continue to demand higher performance and lower power consumption despite increased design complexity and market expectations for lower unit costs [6]. Design flexibility and configurability have also become crucial, particularly due to market uncertainties and shorter product life cycles. Application-Specific Integrated Circuit (ASIC) designs have been severely impacted by continuous falling of design-start numbers due to increased NRE costs and TTM pressures. Embedded systems design is therefore increasingly moving towards programmable hardware in an attempt to satisfy both functional and non-functional design constraints. A major barrier to this shift in embedded systems design trend is the lack of appropriate design automation tools to support rapid customization of applications on the programmable hardware such that they can meet both the functional and non-functional constraints of embedded computing requirements.

In this chapter, we first provide a literature review on the design challenges faced by current and future embedded system designers. A recent report from International Technology Roadmap for Semiconductors (ITRS) shows that there is a need for a higher degree of customization in future embedded systems to manage the growing complexity of the applications. At the same time, they must continue to facilitate a high level of flexibility to meet the shrinking TTM window and lower NRE costs. In light of this, a large amount of research work has been conducted in recent years in the area of customizable processors, e.g. configurable processors and RISPs. In particular, RISPs are expected to play an important role in future embedded SOC platforms due to their promising ability to overcome the technological and market challenges. The majority of the existing research work in customizable processors focuses on efficient generation of custom instructions. We will provide a detailed literature review to highlight the limitations of this existing work, which give rise to the motivation of this dissertation.

2.1 DESIGN CHALLENGES IN EMBEDDED SYSTEMS

The design of embedded systems is, to a large extent, constrained by both technological and economic factors. Embedded systems designers are now under severe pressure to meet multiple and often contradictory design constraints of high-performance embedded systems without violating TTM and NRE pressures. In this section, we will provide a literature review on the design challenges faced by current and future embedded system designers that can only be resolved through Electronic Design Automation (EDA) innovations.

2.1.1 The Need for Customization

ITRS has projected the design complexity trends for future embedded applications in portable and wireless consumer devices such as smart media-enabled telephones, digital camera chips, etc., which shows that the number of custom Processing Elements (PEs) in SOC will grow rapidly in subsequent years [7]. As the introduction of new technology solutions is increasingly application-driven, this trend indicates the importance of customization to manage the growing complexity of future embedded applications. In addition, the SOC processing performance trends for future embedded applications in portable and wireless consumer devices reveals a super-linear growing gap between the processing performance requirement and the available device performance [7]. This implies that the computational complexity of future embedded applications is projected to grow faster than the computational capabilities of silicon. Hence, it is evident that customization will continue to be essential in future embedded systems for product differentiation as well as to satisfy the functional requirements (e.g. power, performance, area cost).

2.1.2 ASIC and Its Limitations

SOCs are becoming increasingly heterogeneous due to the need for increased functionality at lower costs. These heterogeneous SOCs include a myriad of computing elements such as ASIC devices, FPGA devices as well as domain-specific and general-purpose programmable processors [8]. It has been shown that there is an order of magnitude gap between the performance of dedicated hardware implementations using ASICs and pure software implementations on microprocessors [9]. Dedicated hardware implementations using ASICs have high power efficiency as they do not suffer from the overhead of instruction fetch and decode, as well as operand load and store operations, which are prevalent in general purpose and domain-specific processors [10][11]. In addition, the work in [12] has shown that circuits implemented on equivalent standard-cell implementations are about 3 to 5 times faster, and about 35 times smaller than FPGA implementations. Hence, large organizations commonly resort to dedicated hardware implementations using ASICs for achieving customization in their embedded system products.

One possible solution to reduce the gap between the required processing performance and the device performance is to increase the number of ASICs in SOC platforms. However, as discussed in [7], this approach towards customization of the SOC is subjected to: 1) an upper bound constraint on power to maintain battery lifetime, and 2) design effort. These factors will become a major hindrance to increasing the complexity of SOC as the processing power is projected to rise by 1000 times in the next decade [7]. In addition, due to the short life cycles of embedded products, design effort must remain at current levels in the foreseeable future [7]. ITRS has projected that a 10 times design productivity improvement is required for newly designed logic over the next ten years to 2019 in order to maintain constant SOC design effort [7].

The advances in Integrated Circuit (IC) technology have made it possible for embedded system designers to incorporate increasingly complex ASICs on SOCs [13]. However, the NRE costs associated with design, verification, and test of ASICs have been rising with each new process technology, and this trend is expected to continue with subsequent feature sizes [14]. ITRS asserts that the design cost is the greatest threat to the continuation of the semiconductor roadmap as it is often the deciding factor of the eventual computing platform (e.g. hardware or software, FPGA or ASIC) [2]. NRE manufacturing costs are further multiplied due to design shortfalls that entail silicon re-spins. Therefore, it is becoming increasingly difficult for ASIC vendors to recoup design costs through revenue gains.

TTM, which is the required design time for the release of a new generation of products, has become a critical design issue with the rapid advancement in technology that leads to short product life cycles [15]. Consumers are expecting more frequent emergence of new products with a larger functionality set due to the constant availability of increasingly sophisticated devices in the market. As discussed in [16], the turn-around-time for high-end ASICs increases with the shrinking feature sizes. The increase in the turn-around-time is mainly due to the increasing design productivity gap, and the time it takes for the design to be manufactured. It was reported in 2003 that a typical ASIC development time is generally around 18 months [17]. A relatively minor design shortfall can lead to a catastrophic delay in TTM of as much as six to nine months. The lifetime of an application-specific electronic product is projected to decrease to about one month in this decade, and the TTM induced by existing design solutions is well beyond this time frame [18].

ASIC design-starts are dropping more than 50% from 7749 in 2000, to 3196 in 2007 [19]. Further decline in the number of ASIC design-starts is expected [20]. Designing an ASIC in today's deep sub-micron geometries is becoming more difficult with the shrinking geometries. In addition, ASIC manufacturing costs continue to rise. Design tools are finding it difficult to handle the complexity and electrical design challenges posed by new technology generation. The rigid structure of ASIC also restricts the flexibility of the architecture, and excludes any post-design optimizations and upgrades in features and algorithms. This limits the utilization of ASIC for embedded devices which require the flexibility to handle a variety of applications, services and standards.

2.1.3 The Need for Customization and Flexibility Trade-Off

The number of application domains on an SOC platform has been projected to increase to about 100 in this decade [18]. The rapidly shrinking TTM window in today's fast-paced electronic markets is putting considerable pressure on embedded system companies to introduce their products early in the market. At the same time, they have to contend with increasing NRE costs. Hence, future platform designers will need to identify the right degree of programmability or flexibility that can lead to economically feasible solutions and at the same time satisfy the tight non-functional constraints of embedded computing [16]. Flexibility, however, is very often achieved by trading-off power and performance. The success of future platforms therefore depends on the availability of technology, design methodologies and techniques that facilitate rapid customization of a programmable SOC platform to meet the application needs without incurring high NRE costs.

The increasing complexity of embedded applications coupled with the incapability of future processing systems to meet the performance requirements create a design productivity challenge that can only be tackled through platform and EDA innovations. Potential innovations discussed in ITRS [7] include appropriate hardware-software partitioning in high-level design stages (e.g. [21]-[25]), architecture optimization during high-level synthesis (e.g. [26][27]), and customized PE realization (e.g. customizable processors).

2.2 CUSTOMIZABLE PROCESSORS

Customizable processors are a class of emerging computing platforms that offer a promising solution for meeting the customization and flexibility trade-off requirements in future embedded systems. Customizable processors have been described as the next evolutionary phase in the microprocessor business [28]. These processors can be categorized into: 1) *configurable processors*, and 2) *RISPs*.

2.2.1 Configurable Processors

Configurable processors [4][29]-[31] have emerged to bridge the full-hardware and processor based implementation gap in the design continuum. These processors allow designers to tune the processor's instruction set architecture to the application's characteristics. This includes altering the cache configurations, register file size, data paths and bus architecture widths.

In addition, many commercial configurable processors now offer the possibility of extending their instruction set for a specific application by introducing customized functional units within the processor architecture. For example, a Custom Functional Unit (CFU) can be included to optimize the processor pipeline or data path components for critical code segments in specific application domains. This application-specific instruction set extension to the computational capabilities of a processor provides an efficient mechanism to meet the growing performance and TTM demands of embedded systems. The resulting processors are optimized for a targeted application domain but remain programmable for any application [4].

Each new generation of configurable processors requires the redesign of the CFU, which incurs high NRE costs and is subjected to stringent TTM pressure. Hence, while configurable processors have been proven successful for features sizes below 90 nm, rising development costs for ASIC designs tend to favor reconfigurable approaches [32].

2.2.2 Reconfigurable Instruction Set Processors

In light of the economic and technological complexities associated with the design and manufacturing of ASICs, reconfigurable hardware such as FPGAs have evolved into an enabling technology that allows embedded system designers to minimize NRE costs and TTM in developing a new product [33]. Appendix A.1 provides some preliminaries of FPGA architectures. The ability of FPGAs to accelerate a wide variety of applications through hardware programmability renders them a cost-effective, flexible and lower risk alternative to their ASIC counterparts. In addition, the flexibility of FPGAs facilitates the extension of product lifetime in the market, thereby decreasing the threat of obsolescence by its competitors [17].

FPGAs have progressively dominated the IC market as the increasing NRE costs of ASIC began to outweigh the per-unit-cost of FPGAs for high-volume applications [20][34]. This is corroborated by the increasing adoption of reconfigurable technologies such as FPGAs in high-volume designs [35][36]. FPGA technology can now be found in a multitude of applications that are subjected to tight market and technological constraints [37][38]. The FPGA market is expected to grow by over 30% in 2010 to reach US$4 billion. This growth is projected to continue steadily to US$6 billion by the end of 2015 [39].

Platform FPGAs are modern FPGAs that are often characterized by their high configurable logic density and a multitude of soft and hard Intellectual Property (IP) cores [40]. As shown in [41], the availability of open-source and proprietary soft processor cores in platform FPGAs has increased notably in recent years. In addition to these soft processor cores, some FPGA vendors like Xilinx, incorporate 32-bit PowerPC hard processor cores in their high-end Virtex devices. It has been projected that by 2010, more than 40% of all FPGA designs will contain a microprocessor [38].

RISPs are a class of platform FPGA, where the reconfigurable fabric or RFU is placed within the processor (see Appendix A.2). The instruction decoder issues instructions in the form of custom instructions to the RFU as if it were one of the standard functional units in the processor [3]. Similar to configurable processors, RISPs offer the possibility of extending the basic instruction set of the microprocessor by introducing custom functional units on the RFU to implement custom instructions. These custom instructions can be commonly found in most applications, and hence unlike the attached processor and co-processor systems, the RISP is well suited to a wider range of applications. As opposed to loosely coupled schemes where data are communicated between the processor and RFU through a shared memory, the tightly coupled scheme utilizes internal register files for data transfer. Hence, as discussed in [3], the communication overhead does not pose a major bottleneck to the system performance in RISPs. RISPs can also lead to a cost effective solution as the amount of embedded reconfigurable logic in the processor is kept small.

The authors in [42] have reported that including generic memory accesses in RISP creates a twofold problem. Firstly, the resulting instruction has a non-deterministic latency, an undesirable characteristic especially in compile-time scheduled machines. Secondly, the architectural design of RISP becomes complicated with a synchronization mechanism to memory. Hence, although provision for the RFU to access the memory hierarchy can be made in order to implement memory (e.g. load and store) and stream-based operations, this is usually undesirable.

Existing commercial RISP architectures include the FPGA soft-core Xilinx MicroBlaze [43] and Altera NIOS II processors [44]; and the Stretch processor [45], which incorporates the Xtensa RISC processor core [29] and proprietary programmable logic. RISP architectures proposed in academia include the Dynamic Instruction Set Processor (DISC) [46]; the CPLD-Based Instruction Set Accelerator (ConCISe) [47], which is based on enhancing a Reduced Instruction Set Computer (RISC) with a Complex Programmable Logic Device (CPLD); and the Configurable and

Reconfigurable Instruction Set Processor (CRISP) [48], which is a Very Long Instruction Word (VLIW) based RISP.

2.3 TACKLING MARKET AND DESIGN CHALLENGES WITH RISPS

There has been an increasing amount of research work in the area of reconfigurable computing [34][49][50] and customizable processors [51]. This implies that RISPs will play an important role in future embedded SOC platforms due to their promising ability to overcome the technological and market challenges. In addition, recent reports often discuss the potential of reconfigurable platforms to tackle the various aspects of market and design challenges of embedded systems [20][36][38][39]. This section provides a literature review of the benefits/potential of RISPs in lowering NRE costs and TTM pressures, as well as providing flexibility and customization trade-offs.

2.3.1 Lowering NRE Costs and TTM Pressure with RISPs

FPGAs have become sufficiently competitive for designers to be used in high volume designs [35] due to the eroding cost per FPGA gate coupled with the escalating design costs for ASICs. The RISP's ability to customize applications with diverse functionality on the same reconfigurable fabric can further lead to amortization of the mask costs over high platform reusability. In addition, the reconfigurable capability of RISPs also significantly reduces the risk of erroneous products. These factors contribute to the reduction of NRE costs. Furthermore, the logistic costs of RISP based products can be significantly reduced as the cost associated with the maintenance and support of large families of different customized products are obviated.

In addition, even though RISPs have lower performance and power efficiency than their configurable counterparts, the design flexibility of RISPs in the presence of reconfigurable logic leads to off-the-shelf products that can be customized for each application. This avoids ASIC tape-out for each design, thereby eliminating the need to manage exorbitant NRE costs of configurable processors. As discussed in [50], the design flexibility of reconfigurable logic is especially attractive for applications in ubiquitous computing with evolving standards, which require frequent functionality updates. This is increasingly preferred by designers who develop products for uncertain markets and shorter product life cycles.

Logic suppliers are driven towards embedding FPGA cores in SOC designs to address TTM and mitigate design risk issues [33]. A high degree of reuse and direct availability of RISP based products enable a fast response to the market needs. This leads to faster TTM, which will remain as a much coveted advantage in the competitive market. At the same time, product differentiation is achievable through customization on the reconfigurable fabric.

2.3.2 Providing Flexibility and Customization Trade-Offs With RISPs

The RISP is more flexible than the ASIC and configurable processor implementation, which precludes post-manufacturing design-changes. Rapid redesigning of the software programs due to functionality updates and changing standards is possible. In addition, the built-in flexibility of reconfigurable logic in RISPs reduces the risk associated with market uncertainty.

The work in [53] reveals that a single-chip platform consisting of a microprocessor with a small amount of configurable logic can result in significant increase in performance and energy savings over pure software based implementations on a single microprocessor. As customization of RISPs is achieved by mapping critical code segments of the application onto the reconfigurable fabric to leverage their inherent computational parallelism, this can lead to significant performance acceleration and power savings. Moreover, the ability of RISP to perform runtime reconfiguration obviates the need for a large reconfigurable area by providing for hardware reusability during runtime. This not only facilitates cost-efficient products based on RISPs but also potentially reduces static power dissipation due to leakage current [54].

The authors in [53] have suggested that embedded microprocessor platform vendors should seriously consider including configurable logic for improved software execution as well as increased energy savings. A recent startup, Menta [55], now offers soft customizable embedded FPGA (eFPGA) cores that can be readily integrate into SOCs. The work in [56] investigates how the power consumption of a soft-core processor will be affected by integrating a small eFPGA in the processor pipeline. The experimental results show that when dynamic frequency scaling is employed, the soft-core processor can achieve 10 times savings in dynamic power with relatively insignificant power overhead incurred by the eFPGA. This implies that the cost of future RISP based solutions may be dominated by the hardware cost (as opposed to design costs) due to the flexibility of the system. If this becomes a reality, the market tendency for product differentiation will place a greater importance on lowering the hardware cost for FPGA based systems.

In order to realize the potential of FPGAs, supporting design methodologies and tools that will enable system designers to achieve short TTM which is comparable to pure software based solutions must be established [57][58]. In particular, given a fixed reconfigurable logic capacity, a set of candidates for hardware implementation that can lead to area-time-efficient realizations must be determined rapidly in order to meet the tight TTM pressures. In RISPs, the candidates for hardware implementations refer to custom instructions and the process to automatically generate custom instructions from an application is defined as *instruction set customization*.

2.4 INSTRUCTION SET CUSTOMIZATION FOR RISPS: A NEED FOR INNOVATIONS IN EDA

Instruction set customization is defined as the process to automatically generate custom instructions from an application in order to meet certain design objectives [51]. Custom instructions can be found in almost all applications as they impose fewer restrictions on the characteristics of the application [3]. The work in [59][60] has shown that in addition to hardware acceleration, custom instructions can also lead to further increases in performance and power savings of the application by reducing the traffic to the instruction memory. For power sensitive solutions, mapping operations to hardware enables the entire system to be clocked at a lower frequency, while still meeting the performance requirement [61]. This results in direct savings in dynamic power consumption.

While commercial FPGA design flows are readily available, none of them have the provision for instruction set customization that would lead to efficient solutions for RISPs. As demonstrated in [62], manually identifying custom instructions from an application is a daunting task even for very small applications. The number of candidate custom instructions for a given application grows exponentially with the application size. It is not uncommon for functions with a few tens of operations to contain several hundred custom instruction candidates. The only commercial tool that is capable of instruction set customization is the Mimosys Clarity tool [63]. However, the tool is not capable of exploring the design space for selecting custom instructions that meet the area-time constraints of the application for FPGA realization. The absence of efficient instruction set customization in commercial EDA tools is one of the shortcomings in the successful adoption of RISPs in commercial products.

The instruction set customization problem spans across many different fields, such as engineering and graph theory. In particular, graph theory has become a dominant approach for tackling the instruction set customization problem as it seems to provide the right analytical framework [64]. The process of instruction set customization can be viewed as fine-grained hardware-software partitioning step, where the application is represented as a directed graph or Data Flow Graph (DFG), and the custom instructions are sub-graphs with certain properties. The DFG of an application is typically obtained from the Intermediate Representation (IR) of a compiler (see Appendix B.1 and B.2 for details on the compiler and IR). Instruction set customization then translates into the problem of identifying the sub-graphs (corresponding to custom instructions) in order to achieve certain objectives (e.g. increasing performance).

Figure 2-1 shows an example of instruction set customization. A custom instruction (denoted as *CI1* and *CI2*) encapsulates a sub-graph in the application's DFG. The sub-graph comprises a group of adjoining primitive operations that can be implemented on the CFU (for configurable processors) or RFU (for RISPs). In the example, two custom instructions, *CI1* and *CI2*, have been generated from the application DFG through instruction set customization. The compiler will then replace the corresponding primitive operations in the DFG with custom instructions.

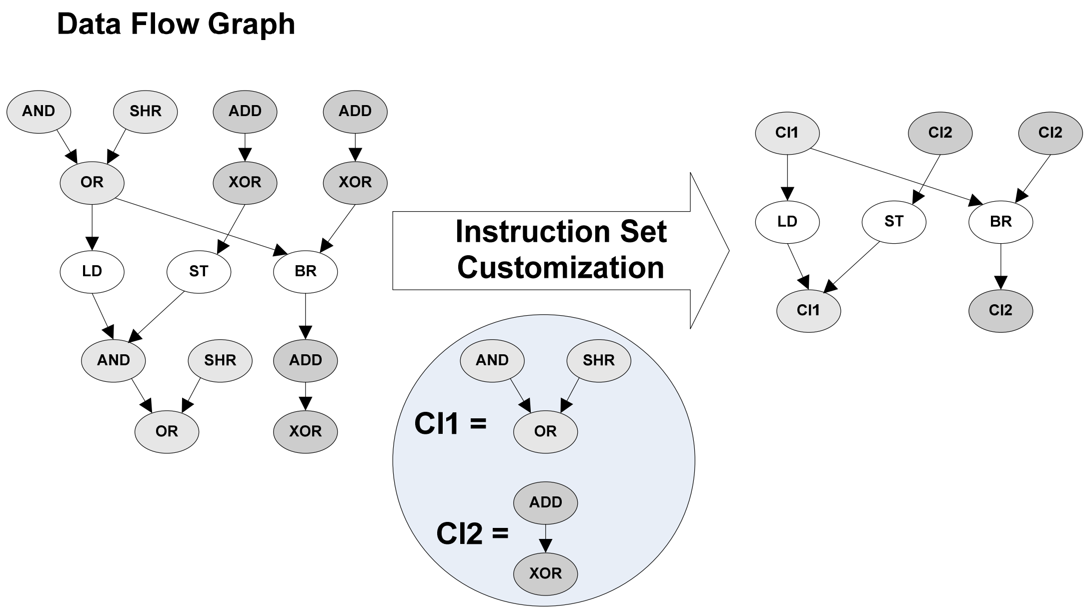


Figure 2-1: Instruction set customization

Instruction set customization consists of the following sub-problems: 1) *custom instruction identification*, 2) *custom instruction selection*, 3) *high-level estimation*, 4) *hardware optimization*, and 5) *runtime configuration generation*. In this thesis, '*custom instructions*' and '*templates'* are used interchangeably. Hence, template identification/selection also refers to custom instruction identification/selection. We will provide a detailed literature review of the existing work in the respective areas and highlight their limitations.

2.4.1 Custom Instruction Identification (Template Identification)

Custom instruction identification can be loosely described as a process of detecting a group of operations or sub-graphs from the application DFG that is to be collapsed into a single custom instruction to maximize some metric (e.g. performance). The custom instruction identification process generates a set of *custom instruction candidates*, which will be considered for custom instruction implementation. In this thesis, the terms '*template instances*' and '*custom instruction candidates*' are used interchangeably. The requirements that are imposed on the template instances during custom instruction identification are influenced by certain characteristics of the core processor, cost considerations and compiler limitations [65]. Appendix B.3 describes the typical constraints that are commonly imposed on the template instances.

There are a number of existing works in the literature on custom instruction identification. In [66], an approach that combines template matching and generation has been proposed to identify sub-graphs based on recurring templates. The approach in [67] iteratively solves a set of Integer Linear Programming (ILP) problems to maximize the code covered by custom instructions, given the available data bandwidth and transfer latencies between the base processor and custom logic.

Other approaches [62][68][69] rely on heuristics to identify good templates while discarding less promising ones. In [62], a method that identifies templates in three phases was proposed. In the first phase, initial templates, consisting of a single node in the DFG that satisfies a given selection criteria are identified. In the second phase, dependent templates are formed using an initial template as seed, through the inclusion of dependent predecessors and successors that satisfy the selection criteria. In the third phase, the initial and dependent templates are used as seeds and nodes that are independent of the seed template are added to construct independent templates. The method proposed in [68][69] attempts to grow a candidate sub-graph from a seed node. The direction of growth relies on a guide function that reflects the merit of each growth direction.

The methods discussed above have demonstrated possible gains, but they can potentially miss out on identifying some good templates. This provides the motivation for template enumeration approaches for custom instruction identification. Template enumeration guarantees the identification of all possible custom instruction candidates. The template enumeration method proposed in [70] employs a binary tree search approach to identify all possible template instances in a DFG. In order to speed up the search process, unexplored sub-graphs are pruned from the search space if they violate a certain set of constraints (i.e. number of input-output ports, convexity, operation type, etc.). In the worst case, the algorithm has to consider 2^n template instances where n is the number of nodes in the DFG. Other approaches have been presented in [71]-[73] to accelerate the template enumeration process. As the template identification problem has been widely studied with notable contributions made in previous work, we will focus on other sub-problems of instruction set customization in this thesis.

2.4.2 Custom Instruction Selection (Template Selection)

Template selection evaluates the template instances in terms of their performance, area or power, and selects a subset of them that meets the design objectives. The selected templates will be implemented as custom instructions on the CFU (for configurable processors) or RFU (for RISPs). Our work published in [J-2][C-3] has shown that exact algorithms for template selection are prohibitive for large sized problems. Hence, approximate solutions are often used for template selection.

Existing approximate custom instruction selection techniques have been formulated as knapsack or graph covering problems. Dynamic programming and greedy algorithms have been employed to obtain a subset of custom instruction candidates that meet certain design objectives. In [66], a covering algorithm was presented to select a minimal set of templates that maximizes the number of covered nodes. In [68]-[70], greedy selection policies were employed to heuristically select a subset of templates. The work in [67][71] formulated the template selection process as a knapsack problem. Each template instance is associated with a performance gain and area cost. A dynamic programming algorithm is then employed to select a subset of the instances that maximizes the performance gain subjected to an area cost bound. ILP based methods for template selection have also been discussed in [65][74][75].

The method presented in [76] employs a graph-covering algorithm, formulated as the Maximum Independent Set (MIS) problem, on a conflict graph to maximize the number of covered nodes using a minimum number of templates. However, the optimization objective to minimize the number of templates may not lead to custom instruction realizations with high performance gain. More recently, the work in [77] proposed a hybrid algorithm for recurrence-aware template selection that combines a greedy covering algorithm and an exact branch and bound algorithm that operates on a restricted problem space. Even though the technique in [77] has restricted the problem size for the exact algorithm, it still requires a runtime in the order of seconds for certain applications.

The existing approximate techniques for template selection often employ heuristics that are formulated without a detailed analysis of the template characteristics in typical applications. In order to formulate a suitable heuristic for template selection, there is a need to perform proper template classification in order to study their statistical properties in commonly used benchmark applications.

2.4.3 High-Level Estimation

In order to facilitate effective custom instruction selection, rapid design exploration must be undertaken without delaying the short TTM requirements for embedded systems. Rapid design exploration can be achieved with the presence of a fast and accurate method to estimate the performance-cost mapping of custom instructions on hardware. While previously reported design flows for instruction set customization have focused on efficient algorithms for custom instruction identification and selection, they do not incorporate an effective technique for area-time estimation that takes into account the architectural constraints of commercial FPGAs. For example, the estimation process in [65][68]-[70] is obtained by pre-computing the area-time of the custom instruction operations using standard-cell design tools. The area and delay of a custom instruction is then derived by summing up the pre-computed area–time values of the corresponding operations. In a similar manner, the delay estimation strategy in [71] predicts the relative speedup of the custom instructions on FPGA by utilizing a rough approximation of the throughput of each instruction. While these approaches may provide reasonable estimations for standard-cell implementations, they do not lend themselves well towards FPGA estimations. This is due to the fact that these methods do not take into consideration FPGA optimization strategies that maximize the resource utilization of the programmable logic structures. Other reported design flows (e.g. [62][67]) incorporate a time consuming hardware synthesis flow to facilitate the selection of custom instruction candidates that maximize performance under a given area constraint. Hence, there is a need to incorporate efficient high-level estimation approaches that take into account the architectural constraints of commercial FPGAs for instruction set customization.

High-level estimation is directly performed on the high-level algorithmic representations of an application (e.g. CDFG, C, Matlab, etc.) without the need for time consuming hardware design entry and implementation. It is worth mentioning that high level estimation techniques differ from existing technology mapping approaches for area-time FPGA optimizations (e.g. [78][79]) as the latter rely on the availability of gate-level representations of the applications.

The work in [80] estimates the FPGA data-path area by using a formula, which is a function of the operator and register properties that are derived from Register Transfer Level (RTL) code (generated from Matlab). The number of Configurable Logic Blocks (CLBs) that correspond to the operators is obtained by pre-characterizing the area consumed by each operator type and size. The area estimation error is within 16% of those reported by commercial FPGA implementation tools. The work in [81] derives area-time estimation from a DFG that is generated from an execution trace (obtained from simulating a Matlab program), which contains information on the type and frequency of the operations. A FPGA performance model is used to estimate the area-time of the operations in the DFG. The performance model incorporates information of the operations which includes the characterized FPGA area-time measures. Accuracy of the estimation is within 10% of actual implementations.

The authors in [82] presented a two-level model to estimate the area of System-C designs. The high-level model analyzes the System-C description and estimates the number of intermediate variables. The low-level model substitutes these high-level variables into a set of equations to estimate the number of LUTs and Flip Flops (FFs). The proposed models must be re-tuned for a new set of benchmarks for different target devices or when there is a change in the design tools. For the applications and models considered, the authors reported an average error of about 17% for LUT estimation.

The work presented in [83] estimates the FPGA data-path area (in terms of LUTs) from a DFG that is generated from high-level SA-C codes. The estimation method relies on a formula that is derived from characterizing the resource consumption of all DFG nodes. In order to take into account some synthesis optimizations in the estimation for improved accuracy, heuristics are employed on patterns that are frequently optimized by the synthesis tool. However the work does not consider more complex optimizations for maximizing the FPGA resource utilization. The area estimation error is within 5%.

Most recently, the work in [84] performs area-time estimations of RTL solutions using a two-step approach. In the first step, a structural exploration is performed to obtain several RTL solutions. In the second step, the area-time estimation of mapping the RTL solutions is undertaken. The physical mapping estimation relies on a FPGA characterization file for a target device. The FPGA characterization file (contains the number and type of FPGA resources, and area-delay information of the operators and memories) is obtained from the datasheet of the target device, and from synthesis of basic operators. Experimental results performed on FPGA devices from different vendors reported an average area estimation error of 18%.

The existing high-level estimation methods discussed above may not lead to reliable estimation results as they do not take into account the FPGA architectural constraints and synthesis optimizations for maximizing the utilization of the FPGA resources. The existing methods also commonly rely on a pre-characterization step, which limits the scope of representing all possible combinations of the design under examination. In addition, the methods in [80]-[83] are not performed on the IR of ANSI C applications, which are commonly employed in embedded applications. Hence, there is a need to develop more reliable high level estimation techniques for instruction set customization in order to facilitate reliable and rapid design exploration for custom instruction selection.

2.4.4 Hardware Optimization

Although there are a number of reported works in academia on the various sub-problems involved in instruction set customization [64], they do not incorporate effective techniques for area-time efficient realization of custom instructions on FPGAs. Area-time optimization can be performed during high-level [85] or gate level synthesis (i.e. technology mapping [78][79]). In contrast to gate level synthesis which operates on the available logic gates of a target architecture library, high-level synthesis operates on the primitive operations derived from the behavioral/algorithmic representations. Since designs at higher levels of abstraction are less confined to the physical architecture, optimizations at these levels usually lead to high quality results.

High-level area minimization has often relied on strategies to maximize resource sharing of the data-paths. These approaches aim at maximizing the reuse of operations and interconnections by identifying similarities between two data-paths. Commercial FPGA tools also adopt the resource sharing approach as one of their main area optimization strategies. For example, the Xilinx synthesis tool supports resource sharing for a limited number of hardware resources (e.g. adders, subtractors and multipliers), by implementing one single arithmetic operator for similar operations that are never used concurrently [86].

Resource sharing has also been employed for reducing the system reconfiguration overhead of reconfigurable architectures [87][88]. The work in [87] minimizes the run-time reconfiguration time by identifying common components in two successive configurations. A weighted bipartite graph is constructed from two successive configurations, and an algorithm that performs graph matching and combining is employed to produce a combined configuration. In [88], a data-path merging algorithm has been presented to reduce the reconfiguration overhead of an architecture template with a reconfigurable interconnection network. In order to optimize interconnect sharing, the data-paths are merged by solving the maximum bipartite matching problem.

The work in [89], which was later extended in [51], represents custom instruction data-paths as path sequences, and computes the longest common subsequence to identify possible resource sharing between the sequences. In [90], the left-edge binding algorithm was employed to minimize the number of functional units and registers through resource sharing. Other methods for improving the interconnection mapping include iterative improvement and ILP approaches [91].

In [92], a data-path merging algorithm is presented to merge several DFGs in order to produce a reconfigurable data-path with minimum hardware operations and interconnection. The algorithm first constructs a compatibility graph that represents all the possible mappings of common operations and interconnectivity between two DFGs. The maximum weight clique problem is then solved to maximize resource sharing between the two DFGs. The authors in [92] demonstrated that their technique outperforms other approaches based on bipartite matching, iterative improvement and ILP.

The resource sharing methods discussed above rely heavily on multiplexers to facilitate sharing of common resources between the data-paths. While previous results have shown that resource sharing based approaches lead to significant reductions in the total number of operations, they may not be able to maximize the FPGA resource utilization and can result in high critical path delay. This is due to the fact that the existing resource sharing approaches do not take into consideration the FPGA architecture constraints during data-path merging.

2.4.5 Runtime Configuration Generation

Runtime reconfiguration offers the potential to realize low cost systems that can still lead to high performance by changing the configuration of a small reconfigurable hardware at runtime. However, the fine-grained programmable structure of commercially available reconfigurable architectures incapacitates efficient application mapping and runtime reconfiguration, both of which are necessary to satisfy the non-functional constraints of embedded systems. The authors in [93] have highlighted two main drawbacks that discourage the use of runtime reconfiguration in embedded real-time systems. The first is the large reconfiguration overhead in commercial FPGA architectures and the second is related to the lack of support for run-time reconfiguration management.

Recently, runtime reconfiguration has been investigated for cost effective realizations on FPGA based RISPs [94]-[96] that have access to the memory hierarchy. These works have demonstrated the benefits of runtime reconfiguration on customized implementations of the JPEG and H.264 encoder/decoder. However, the feasibility of runtime reconfiguration on general RISPs depends largely on the type of application and the ability of the compiler to extract custom instructions that can mitigate the high reconfiguration overhead of existing FPGA architectures. For example, the DISC processor [46] requires a reconfiguration time that is projected to contribute up to 16% of an application's total execution time. The Stretch S6000 processor [45] requires 20 milliseconds to change an instruction on their proprietary programmable logic. Partial reconfiguration on the Xilinx Virtex FPGA is accomplished in the order of milliseconds [96]. This high reconfiguration overhead could suppress the benefits of dynamically reusing the FPGA hardware resources, if the reconfiguration overhead cannot be compensated by the speedup. Hence, while the programmability in commercial FPGA (i.e. Stretch and Xilinx devices) provides flexibility, it still comes at a significant cost in performance and power consumption that may be intolerable for the highly demanding requirements of embedded applications. Efficient architectures and techniques must be devised to maximize the performance of custom instruction realization through runtime reconfiguration, while minimizing the reconfiguration overhead.

In order to select custom instructions for different runtime configurations, temporal partitioning must be performed to divide the design into mutually exclusive configurations such that the computational resource requirement of each configuration is less than or equal to the reconfigurable resource capacity of the RFU.

The task partitioning algorithm presented in [97] for minimizing the communication cost is achieved in two steps. In the first step, an initial partition is obtained by using a network flow based algorithm to produce a set of feasible mean cuts. In the second step, a scheduling technique is employed to select an optimal global solution. The work in [98] employs ILP to achieve near-optimal latency designs for temporal partitioning of the application task graph. A loop transformation strategy was used to maximize the throughput while minimizing the runtime reconfiguration overhead.

The framework presented in [99] automatically partitions loops to a target platform consisting of a processor, RFU and memory hierarchy. The approach employed in [99] is based on partitioning a loop into smaller components in order to perform optimal hardware-software partitioning of the nested loops. The loop partitioning strategy traverses the hierarchical loop graph in a top-down fashion and recursively combines the nested loops until the sizes of all the nested loops are within a pre-defined limit.

The work in [100] describes an architecture-aware temporal partitioning strategy for mapping custom instructions on an adaptive extensible processor, which incorporates coarse-grained functional units. The strategy partitions and modifies custom instructions that violate the RFU constraints in order to map them onto the RFU.

Recently, a framework was presented in [101] to select custom instruction versions to be mapped onto appropriate configurations. A custom instruction version consists of a set of custom instructions from a particular loop that satisfy the area constraint of the RFU. The partitioning scheme consists of temporal partitioning of frequently executed application loops with custom instructions into one or more configurations, and spatial partitioning to select an appropriate custom instruction version for each loop within a configuration. The temporal partitioning problem has been modeled as a *k*-way graph partitioning problem, and spatial partitioning is resolved using dynamic programming. The framework assumes the availability of the custom instruction versions and their corresponding hardware area-time measures. This necessitates time-consuming hardware implementation of the custom instructions prior to the partitioning process if no high level estimation strategy is in place. In the worst case, the temporal partitioning algorithm in [101] iterates *l* times, where *l* is the number of hot loops.

2.5 SUMMARY

Future embedded systems must continue to meet the shrinking TTM window and lower NRE costs. Product differentiation needs and increasing complexity of applications will demand customization in the form of hardware acceleration. The literature survey in this chapter has provided strong evidence to show that RISPs offer appropriate customization and flexibility trade-offs in order to meet the NRE and TTM requirements of embedded systems.

The reported techniques and design methodologies in the literature lack strategies that can rapidly generate area-time efficient custom instructions on FPGAs. Despite the large amount of custom instruction generation related research activities that has taken place recently, there is still a need for further research in this area. In particular, the literature review has highlighted the need to devise innovative solutions in the following areas of instruction set customization: i) custom instruction selection, ii) high-level estimation, iii) hardware optimization of custom instructions, and iv) generation of runtime custom instruction configurations.

To date, almost all commercial tools cannot automatically identify custom instructions from applications, although this feature remains a key interest to companies offering RISP based product solutions. In addition, while RISP solutions are appropriate for use in embedded systems, novel runtime techniques that can also meet the NRE and TTM requirements must be developed.

In the next chapter, we will propose a novel custom instruction selection method to maximize the performance gain of RISPs.

CHAPTER 3

SELECTING CUSTOM INSTRUCTIONS FOR HIGH PERFORMANCE

In this chapter, we present a Largest-Fit-First (LFF) based technique for selecting custom instructions that can potentially lead to large number of software clock cycle savings due to the migration of the native instructions of the processor to hardware. The LFF approach relies on a graph covering heuristic to select non-overlapping superset templates, which typically incorporate frequently used basic templates. The solution of LFF is further refined by considering overlapping templates that were ignored previously to see if their introduction could lead to higher performance. This process examines those with lowest gains first to minimize disruptions to the high gain superset templates.

Performance analysis shows that the proposed techniques outperform other reported approximate strategies. Although the runtime of the proposed methods is comparable to existing approximate strategies, they have been shown to select custom instructions for highest performance, with an increase in performance by over 32% for certain benchmark applications. Also, the proposed techniques can be realized in the order of milliseconds justifying their deployment in rapid design space explorations for custom instruction selection. A preliminary version of this work has been published in [C-8].

3.1 GRAPH-COVERING FOR TEMPLATE SELECTION

In this section, we will first give an introduction to the graph covering problem and the commonly used heuristics for template selection. The effectiveness of graph-covering algorithms for template selection have been demonstrated in [76][77]. The following describes the problem formulation of template selection based on the graph-covering approach:

Given an application DFG G, a unique set of templates $T = \{T_1, T_2, \ldots, T_i\}$ and the template instances of each template T_i, $I_i = \{I_{i1}, I_{i2}, \ldots, I_{ij}\}$, find a subset of the set I that covers G. Figure 3-1(a) shows an example with three templates (i.e. T_1, T_2 and T_3) and nine instances in a DFG. An efficient cover can be achieved by selecting a set of non-overlapping instances that maximizes the number of covered nodes. Hence, a template selection approach that result in an efficient cover can lead to higher performance gain as the instances cover a larger number of operations.

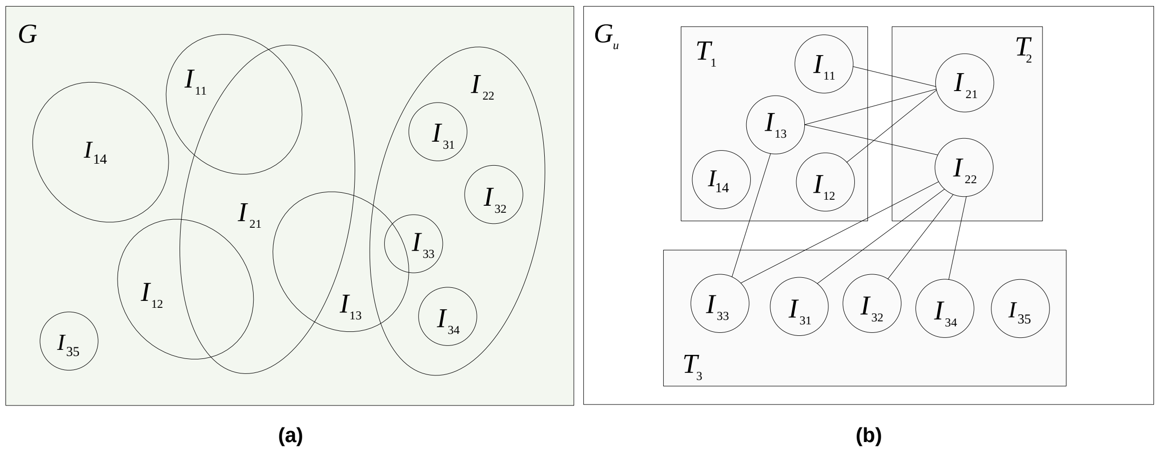


Figure 3-1: **(a)** Template instances in a DFG *G*, **(b)** Conflict graph of *G*

The covering algorithm that we have adopted in the proposed technique is based on the conflict graph approach that was presented in [76]. A conflict graph is an undirected graph $G_u(V_u,E_u)$. Each vertex represents a template instance I_{ij} that is associated with a unique template T_i. An edge $e \in E_u$ between two instances signifies that the instances have at least one overlapping node. The number of nodes in an instance I_{ij} is denoted as $size(I_{ij})$. Figure 3-1(b) shows the conflict graph for the example in Figure 3-1(a). In this example, we assume $size(I_{1j}) = 5$, $size(I_{2j}) = 9$, and $size(I_{3j}) = 1$.

The covering algorithm first constructs a conflict graph from the template instances, and then iteratively computes the Maximum Independent Set (MIS) of each unique template T_i. The MIS of template set T_i, denoted as MIS_i, is defined as the largest subset of instances in T_i that are mutually non-adjacent, which means that the instances do not share any common edges. The number of instances in MIS_i is denoted as $size(MIS_i)$. The computation of the MIS can be implemented in time linear in the number of vertices and edges of G_u by using a simple heuristic, which has been shown to provide good solutions [102].

The MIS_i with the largest objective function ($w(MIS_i)$) is then selected and the instances corresponding to the selected MIS are stored as the selected instances. These selected instances and their adjacent neighbors are then permanently deleted from the conflict graph. The algorithm is repeated until the conflict graph is empty. Details of the algorithm are described in Figure 3-2. NON_i refers to the non-overlapping nodes of the selected instances in T_i. As shown in line 7 of Algorithm 3-1, the NON_i of each selected template T_i is stored in C. In the worst case, the number of iterations required by the selection algorithm is equivalent to the number of vertices in G_u.

The result of template selection is governed by the choice of the objective function used (i.e. $w(MIS_i)$). We will discuss the objective functions that are analogous to commonly used heuristics in existing template selection methods, before describing the proposed approaches. The description of the objective functions and the proposed approaches in the subsequent sections will refer to the example in Figure 3-1.

Algorithm 3-1

```
1.  TEMPLATE-SELECTION ($C$, $G_u$) {
2.      $G_u^{'} = G_u$
3.      while $G_u^{'} \neq \phi$ {
4.          Find $MIS_i$ of each template group $T_i$ in $G_u^{'}$
5.          Compute $w(MIS_i)$ for each $MIS_i$
6.          Select $MIS_i$ with the largest objective function
            (corresponding $T_i$ is the selected custom instruction)
7.          Store nodes associated with the selected $MIS_i$ (i.e. $NON_i$) in $C$
8.          Delete $NON_i$ and the adjacent nodes from $G_u^{'}$
9.      }
10. }
```

Figure 3-2: Pseudo code of conflict graph based template selection

3.1.1 Most-Frequently-Fit-First (MFF)

This objective function for MFF is: $w(MSI_i) = size(MIS_i)$. It aims to select a set of frequently occurring templates, as the algorithm will select the MIS with the largest number of instances first. This approach has a similar objective to the work in [66]. Figure 3-3(a) shows the final covering solution with MFF for the example in Figure 3-1 after two iterations. In the first iteration, MIS_3 will be selected as it has the largest objective function (i.e. $size(MIS_3) = 5$). MIS_1 is then selected in the subsequent iteration. The result of the algorithm is the selection of instances I_{11}, I_{12}, I_{14}, I_{31}, I_{32}, I_{33}, I_{34} and I_{35} that are associated with the templates T_1 and T_3. The gain (measured in terms of the total number of nodes covered) is 20.

3.1.2 Most-Frequent-Largest-Fit-First (MLFF)

The objective function for MLFF is $w(MSI_i) = size(v_x) \times size(MIS_i)$, which takes into account both the frequency of template occurrence and the size of the templates. This objective function is analogous to the heuristic used in [76][77]. Figure 3-3(b) shows the covering solution of MLFF for the example in Figure 3-1 after two iterations. MIS_1 is selected in the first iteration and this is followed by selection of MIS_3 in the subsequent iteration. The result of the algorithm is the selection of T_1 and T_3 (corresponding to I_{11}, I_{12}, I_{13}, I_{14}, I_{31}, I_{32}, I_{34} and I_{35}) which covers 24 nodes.

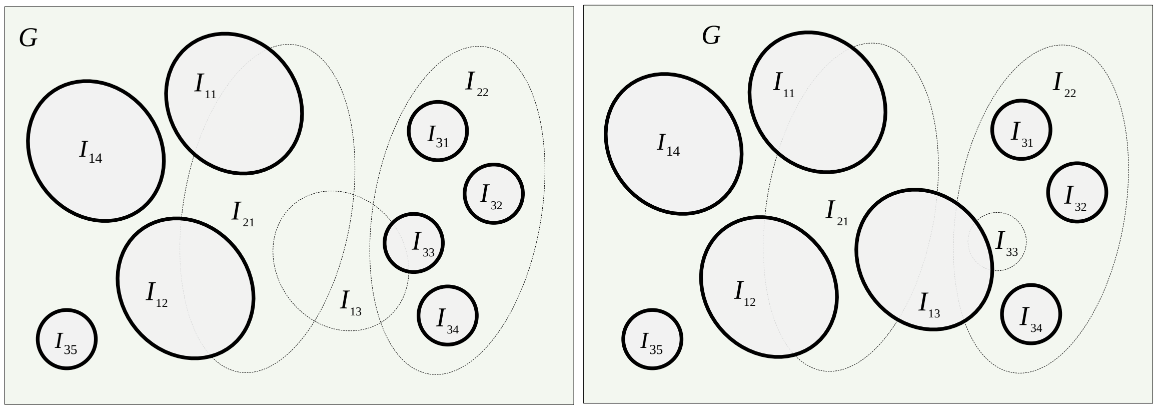


Figure 3-3: (a) MFF based selection (*Gain* = 20), **(b)** MLFF based selection (*Gain* = 24)

3.2 LFF BASED CUSTOM INSTRUCTION SELECTION

In the previous section, we introduced the graph-covering approach for custom instruction selection, which relies on the conflict graph. The commonly used heuristics (i.e. MFF and MLFF) were also discussed. In this section, we proposed the LFF based approaches for custom instruction selection.

3.2.1 Largest-Fit-First (LFF)

This objective function for the proposed LFF heuristic is $w(MSI_i) = size(v_x)$. The LFF approach attempts to select the MIS with the largest instances first. This objective function aims to overcome the limitation of MFF that leads to the selection of small templates with large numbers of instances (e.g. I_{3j}), which could lead to the omission of templates with larger numbers of nodes (e.g. I_{22}). Figure 3-4(a) shows the covering solution of LFF for the example in Figure 3-1 after three iterations. MIS_2 is selected in the first iteration as it has the largest instances (i.e. $size(I_{2j})$ = 9). This is followed by MIS_1 and MIS_3 in the subsequent iterations. The result of the algorithm is the selection of T_1, T_2 and T_3 (consisting of instances I_{14}, I_{21}, I_{22}, and I_{35}) which covers 24 nodes.

From Figure 3-4(a), it can be observed that employing LFF can lead to a larger number of covered nodes if (and only if) most of the instances belonging to frequently occurring templates are entirely consumed by large instances (e.g. instances I_{31}, I_{32}, I_{33} and I_{34} are consumed by instance I_{22}). Otherwise, the instances associated with frequently occurring templates which partially overlap with the large instances will be discarded from the conflict graph when the large instances are selected. In the experimental section, we will present statistical analysis of the spatial locality of templates in a number of applications to justify the feasibility of the LFF approach.

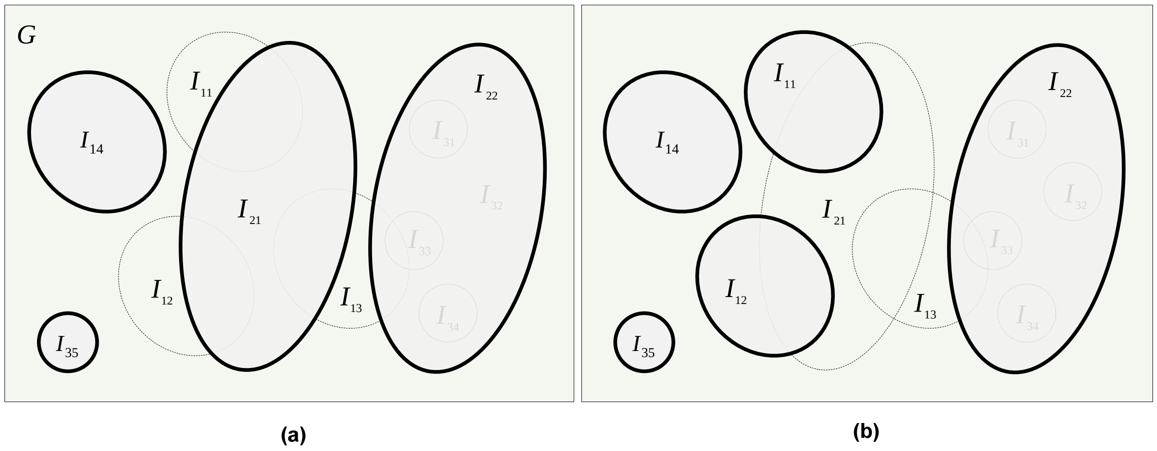


Figure 3-4: (a) LFF based selection (*Gain* = 24), **(b)** Refining LFF (*Gain* = 25)

3.2.2 Refining LFF

The LFF based covering algorithm can be further refined by evaluating the benefits of replacing the initial solution with the non-selected template instances that overlap with the selected instances. This is achieved by evaluating the instances of each selected template (in order of ascending template gain) in the initial solution to check whether they should be replaced by the overlapping instances. For example, in Figure 3-4(b), the initial selected instance I_{21} in Figure 3-4(a) is replaced with the instances I_{11} and I_{12} as they lead to higher gain (i.e. $gain(I_{11}) + gain(I_{12}) > gain(I_{21})$). We denote the LFF algorithm with refinements as RLFF.

Algorithm 3-2

```
1.  RLFF-BASED-TEMPLATE-SELECTION {
2.      $C = \phi$
3.      Build conflict graph $G_u$ from DFG $G_i$
4.      TEMPLATE-SELECTION ($C$, $G_u$)  with LFF objective function
5.      REFINE-SELECTION ($C$, $G_u$)
6.  }
```

Algorithm 3-3

```
1.  REFINE-SELECTION ($C$, $G_u$)  {
2.      $G_u^{'} = G_u$
3.      Calculate gain of each selected template $T_i$ based on $NON_i$ in $C$
4.      Sort the selected $T_i$ in ascending gain
5.      for each selected $T_i$ starting with the lowest gain {
6.          Remove corresponding $NON_i$ from $C$
7.          Identify all neighboring vertices of $MIS_i$ in $G_u^{'}$ and store in $N$, where
              the nodes associated with the vertices in $N \notin C$
8.          Find MIS of $N$ ($MIS_N$)
9.          Calculate gain of $MIS_N$
10.         if $gain(MIS_N) > gain\ (NON_i)$
11.             Store nodes associated with $MIS_N$ in $C$
12.          else restore $NON_i$ in $C$
13.     }
14. }
```

Figure 3-5: Pseudo code of RLFF algorithm

The RLFF algorithm is described in Algorithm 3-2 and 3-3 of Figure 3-5. First the gain of NON_i of each selected template T_i is calculated (line 3 of Algorithm 3-3). Starting from the selected template T_i with the lowest NON_i gain, the corresponding NON_i is temporary removed from *C*. The non-selected instances that overlap with the selected instances T_{ij} are identified and stored in *N* (Line 7). Note that the instances in *N* must not overlap with any selected instances I_{kj} where $k \neq i$. The MIS of *N* is then computed to find a maximal non-overlapping set of instances in *N* (line 9). If the gain of MIS_N is larger than the gain of NON_i, then the instances of *N* replace the initial instances of T_i (line 11). Otherwise, the original selected instances in T_i are restored (line 12). This process is repeated until all the selected templates T_i in the initial solution have been considered. As can be observed from Figure 3-4(b) (based on the example in Figure 3-1), the proposed method leads to the highest gain among the various objective functions (i.e. *gain* = 25) on the given example.

3.3 EXPERIMENTAL RESULTS

In this section, we first describe the experimental setup. We then analyze the spatial locality of template instances from sixteen benchmark applications in order to justify the feasibility of the LFF approach. Finally, we compare the template selection results obtained from the proposed methods (i.e. LFF and RLFF) with the MFF, MLFF and ILP based methods.

3.3.1 Experimental Setup

We have adopted the template enumeration algorithm in [70] to identify all the template instances from the given application set. As mentioned earlier, the method in [70] employs a binary tree search approach that prunes unexplored sub-graphs from the search space if they violate a certain set of constraints. We have used the Trimaran IR [103] for the enumeration process. In order to avoid false dependencies within the DFG, the IR is generated prior to register allocation. For the purpose of this study, we have imposed the following constraints on the templates to increase the efficiency of the identification process:

- Only integer operations are allowed in the template instance. Including memory accesses in custom instructions can lead to non-deterministic latencies and increased complexity in the RFU. In addition, custom instructions with floating-point operations often do not lead to notable speedup [65].
- Each template instance must be a connected sub-graph as we assume the parallelism in custom instructions associated with disconnected sub-graphs is not exploited in the target architecture.
- Maximum number of input ports is 5 and maximum number of output ports is 2. Previous work [65] has shown that input-output ports more than this range result in little performance gain. It is noteworthy that even if the actual number of input-output ports of the RFU is less than the imposed constraints, existing techniques that exploits pipelining and multi-cycle register file access can be employed to efficiently map the custom instructions onto the RFU [104]. For simplicity, we assume that pipelining and multi-cycle register file accesses are not supported in the target architecture.
- An operation that feeds an input to the template instance must execute before the first operation in that instance, to avoid dependency violation when the instance is realized as a custom instruction.

- Only convex sub-graphs are allowed in template instances to ensure a feasible schedule exists when the sub-graph is collapsed into a custom instruction [51].

Table 3-1: Number of template instances

Application	**Benchmark Suite**	**Domain**	**Number of Template Instances**
Adpcm Dec	MiBench	Telecom	17
Adpcm Enc	MiBench	Telecom	22
Aes	EEMBC	Consumer	588
Basicmath Large	MiBench	Automotive-Industrial	6
Bitcount	MiBench	Automotive-Industrial	345
Blowfish Dec	MiBench	Security	1207
Blowfish Enc	MiBench	Security	1207
Cjpeg	MediaBench	Consumer	505
CRC32	MiBench	Telecom	11
Dijkstra Large	MiBench	Network	21
FFT	MiBench	Telecom	15
Patricia	MiBench	Network	8
Pegwit	MediaBench	Security	536
Rijndael Dec	MiBench	Security	1608
Rijndael Enc	MiBench	Security	1633
Sha	MiBench	Security	95

Our experiments are based on sixteen benchmark applications that are obtained from the widely-used MediaBench [105], MiBench [106], and EEMBC [107] benchmark suites. Table 3-1 provides additional information of the applications (i.e. origin benchmark suite and application domain), and the number of template instances found using the template enumeration algorithm in [70].

3.3.2 Feasibility Study of the LFF Approach

In Section 3.2.1, we explained that the proposed LFF approach can lead to a larger number of covered nodes if (and only if) most of the instances belonging to frequently occurring templates are entirely consumed by large instances. In order to investigate the feasibility of the LFF approach, we analyzed the spatial locality of the template instances in the sixteen benchmark applications.

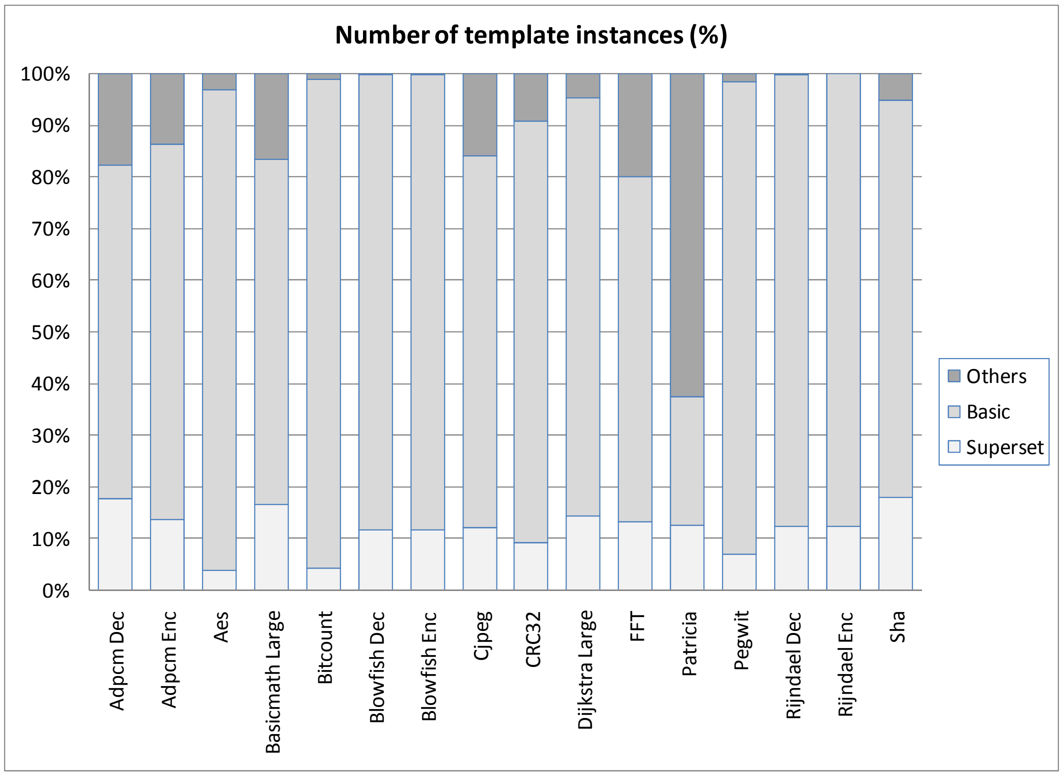


Figure 3-6: Percentage of different template types

Templates can be divided into: 1) *Superset templates*, 2) *Basic templates*, and 3) *Others*. A superset template consists of large template instances that cannot be entirely consumed by other instances or subsume one or more basic template instances. In Figure

3-1(a), T_2 is a superset template as both its instances (I_{21} and I_{22}) are not entirely consumed by other template instances. In addition, I_{22} subsumes template instances I_{31}, I_{32}, I_{33} and I_{34}. A basic template consists of template instances that do not subsume any template instances or are entirely consumed by superset templates. In Figure 3-1(a), T_3 is a basic template as its instances I_{31}, I_{32}, I_{33} and I_{34} are entirely consumed by I_{22}. Template T_1 in Figure 3-1(a) falls under the category 'Others' as none of its instances (I_{11}, I_{12}, I_{13} and I_{14}) subsumes other instances or are entirely consumed by other instances.

Figure 3-6 shows the percentage of instances of the three template types for the sixteen benchmark applications. The statistics imply that a significant number of frequently occurring templates (i.e. basic templates) are contained within the superset templates. In particular, an average of 77.4% of the template instances, are basic templates that are consumed within the superset templates in the sixteen applications. Hence, employing LFF for template selection can lead to the selection of large templates that are also likely to subsume frequently occurring templates.

3.3.3 Performance Evaluation

In this section, we will compare the performance of the selected templates that are obtained using the MFF, MLFF and proposed methods (i.e. LFF and RLFF). In addition, we have implemented an ILP approach for template selection in order to evaluate the quality of the solutions obtained using the proposed methods. Unlike the ILP models that have been used in other approaches (e.g. [65]), the ILP model that we have adopted does not incorporate area constraints as we are only interested in selecting templates that will lead to the highest number of operations. The ILP-based method, although capable of producing optimal solutions, can have extremely long runtime due to the complexity of the problem. We have obtained the results of the ILP approach after 15 minutes of runtime. The runtime of the methods discussed in this chapter (i.e. MFF, MLFF, and proposed) are less than 1 second for all the applications considered in the experiment. The performance gain is reported in terms of the number of nodes covered in the various graph covering approaches. This metric is analogous to the reduction in the number of Instruction Set Architecture (ISA) operations executed on the base processor. While this may not be a realistic measure of the performance gain due to custom instruction realizations, it is sufficient at this stage for comparing the quality of results obtained using the various graph covering approaches.

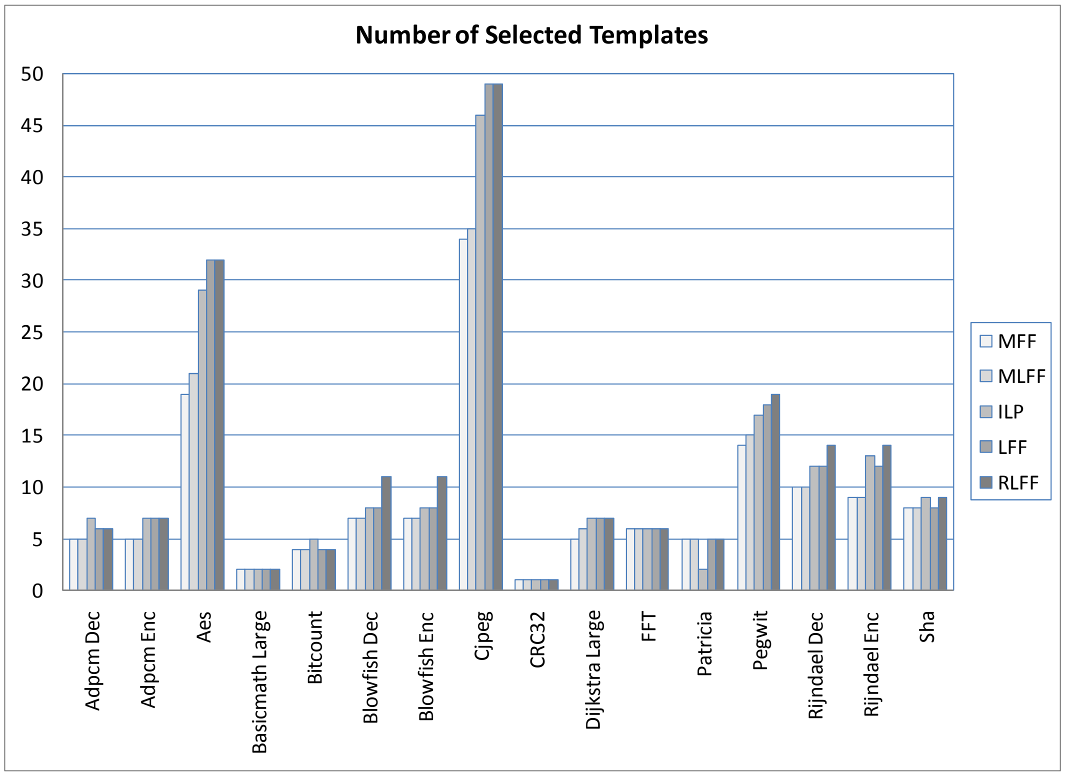
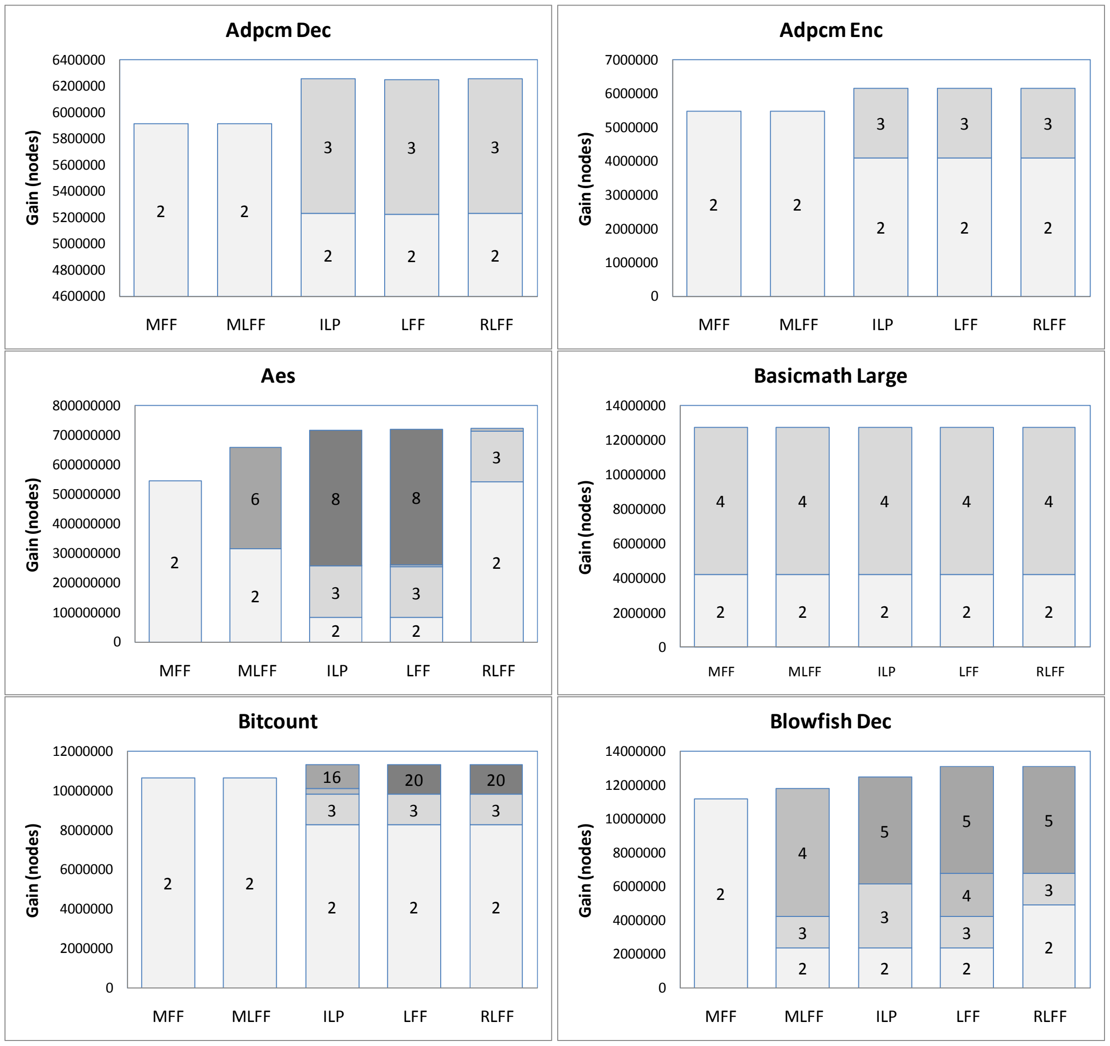
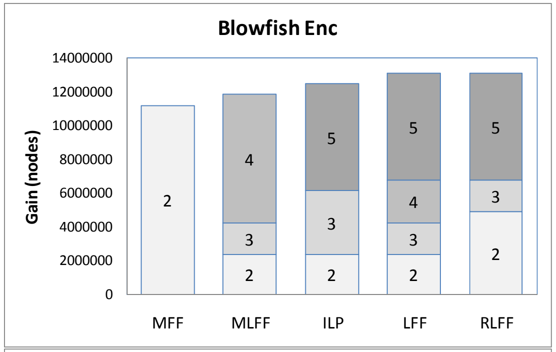
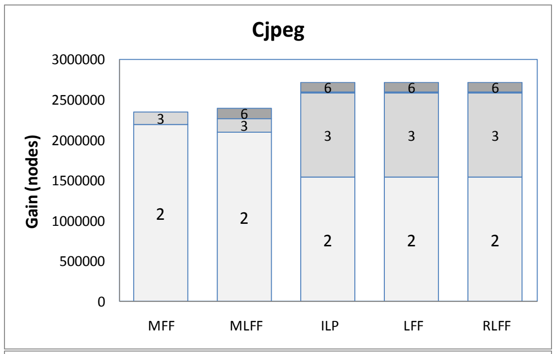


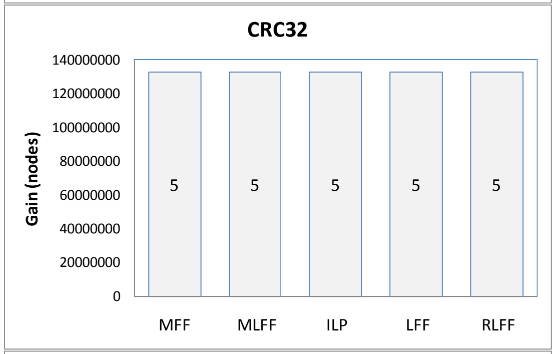
Figure 3-7: Number of selected templates

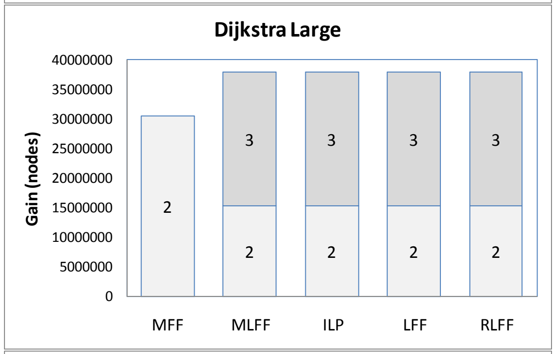
Figure 3-7 shows the number of selected templates that are obtained using the various approaches. The average number of selected templates obtained using MFF, MLFF, ILP, LFF and RLFF is 8.8, 9.1, 11.2, 11.6 and 12.3 respectively. It is evident from this set of results that the proposed approaches (i.e. LFF and RLFF) lead to the selection of a larger number of templates compared to the existing approaches. The results also show that template selection based on the frequency of occurrence of the templates (e.g. MFF and MLFF) can lead to the disposal of a notable number of overlapping templates.

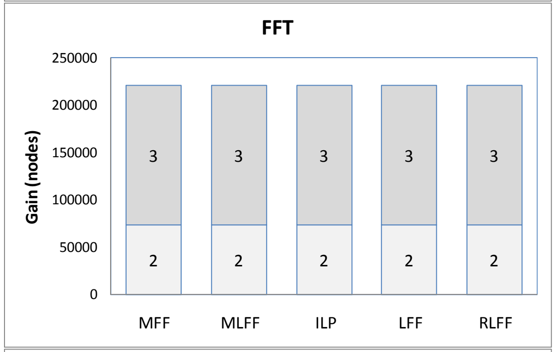


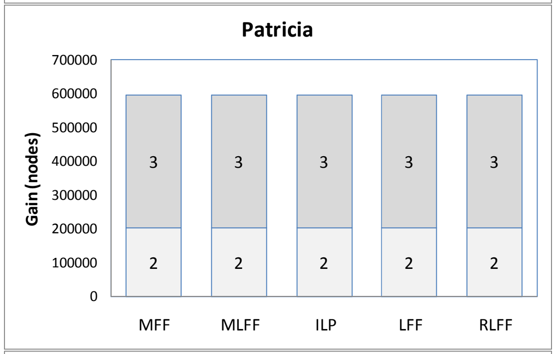

Blowfish Enc
Gain (nodes)
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
2
4
3
2
5
3
2
5
4
3
2
5
3
2
MFF
MLFF
ILP
LFF
RLFF

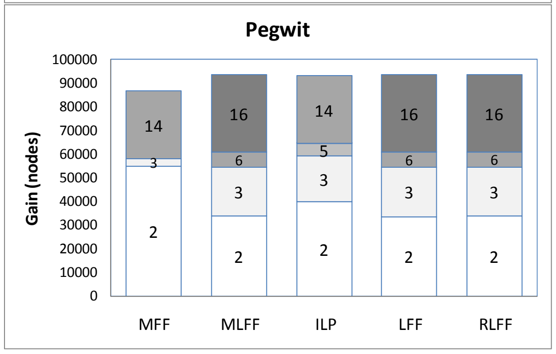

Cjpeg
Gain (nodes)
3000000
2500000
2000000
1500000
1000000
500000
0
3
2
6
3
2
6
3
2
6
3
2
6
3
2
MFF
MLFF
ILP
LFF
RLFF

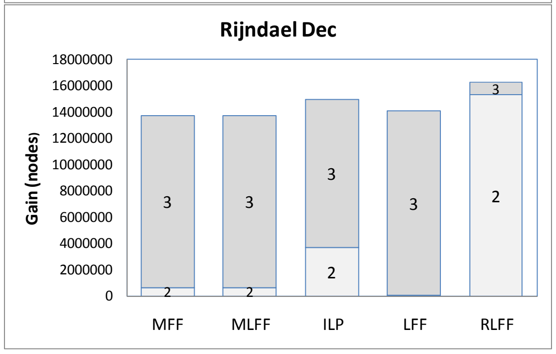

CRC32
Gain (nodes)
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
5
5
5
5
5
MFF
MLFF
ILP
LFF
RLFF


Dijkstra Large
Gain (nodes)
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
2
3
2
3
2
3
2
3
2
MFF
MLFF
ILP
LFF
RLFF


FFT
Gain (nodes)
250000
200000
150000
100000
50000
0
3
2
3
2
3
2
3
2
3
2
MFF
MLFF
ILP
LFF
RLFF


Patricia
Gain (nodes)
700000
600000
500000
400000
300000
200000
100000
0
3
2
3
2
3
2
3
2
3
2
MFF
MLFF
ILP
LFF
RLFF


Pegwit
Gain (nodes)
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
14
3
2
16
6
3
2
14
5
3
2
16
6
3
2
16
6
3
2
MFF
MLFF
ILP
LFF
RLFF


Rijndael Dec
Gain (nodes)
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
3
2
3
2
3
2
3
3
2
MFF
MLFF
ILP
LFF
RLFF

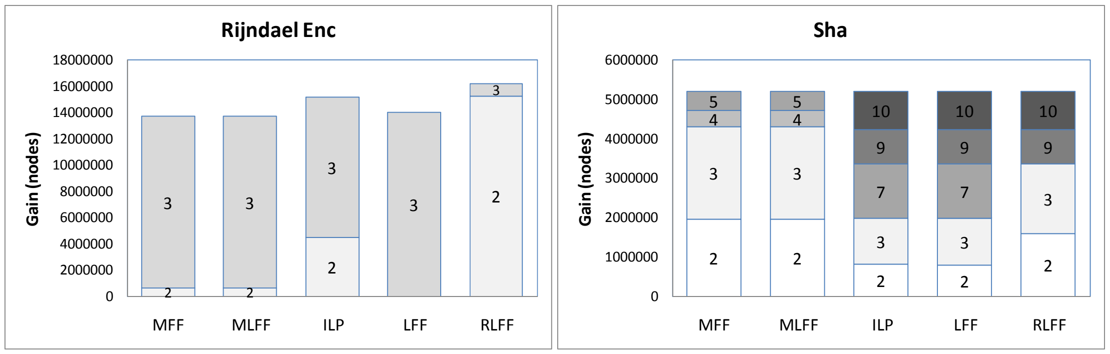


Figure 3-8: Percentage performance contribution of selected templates

Figure 3-8 shows the gain contribution of the selected templates that have been grouped according to their size, in terms of number of nodes (as labeled on the charts), for MFF, MLFF, ILP, LFF and RLFF respectively. For example in Adpcm Dec, only templates of size 2 are selected using MFF and MLFF, while templates of size 2 and 3 are selected using ILP, LFF and RLFF. When compared to the MFF and MLFF approaches, LFF and RLFF generally leads to the selection of more templates (see Figure 3-7) with larger number of nodes (see Figure 3-8), which has resulted in higher gain. For Aes, Rijndael Dec and Rijndael Enc, RLFF leads to the selection of smaller templates compared to those selected using MFF and MLFF approaches. However, as shown in Figure 3-7, the RLFF approach leads to the selection of more templates and hence, the overall gain of RLFF is still higher than MFF and MLFF for these applications. In general, the selected templates of LFF have the largest number of nodes.

Figure 3-9 compares the number of nodes covered using the various template selection strategies, i.e. MFF, MLFF, ILP and proposed methods (i.e. LFF and RLFF). It can be observed that when compared to MFF and MLFF, the ILP approach (which produces optimal or near-optimal results) leads to the highest number of nodes covered in almost all cases (and comparable in the remaining cases). It can be observed from Figure 3-9 that on average, the proposed LFF approach leads to a comparable number of nodes covered when compared to the time-consuming ILP method. In addition, when compared to the ILP method, the proposed RLFF approach has a higher number of nodes covered in several applications (e.g. Blowfish Dec, Blowfish Enc, Rijndael Dec and Rijndael Enc), and is comparable in the remaining ones. It is noteworthy that the proposed methods can execute in a fraction of the time that is required by the ILP method.

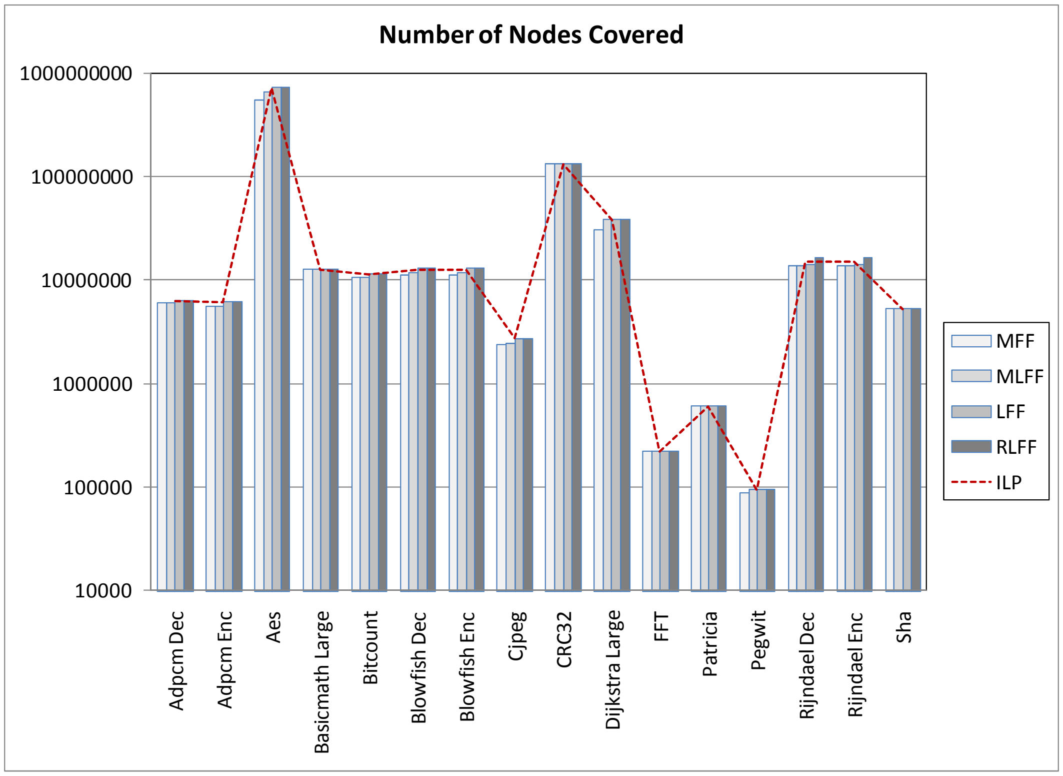


Figure 3-9: Performance comparison

Table 3-2 shows the percentage performance gain (in terms of nodes covered) of the proposed LFF method over other template selection approaches (i.e. MFF, MLFF and

ILP). On average, the proposed LFF method outperforms the MFF and MLFF method by 9.0% and 4.6% respectively. It can be observed that for Aes and Dijkstra Large, LFF outperforms MFF by over 32% and 24% respectively. The LFF approach also outperforms the MFF and MLFF methods by over 10% for a number of applications. Except for two applications (i.e. Rijndael Dec and Rijndael Enc), LFF either performs more favorably or is comparable with the ILP approach. For the Rijndael Dec and Rijndael Enc applications, the percentage performance gain difference between the LFF and ILP approach is less than 10%.

Table 3-2: Percentage performance gain of LFF over MFF, MLFF and ILP

Application	MFF	MLFF	ILP
Adpcm Dec	5.80	5.80	0.00
Adpcm Enc	12.51	12.51	0.00
Aes	32.38	9.33	0.38
Basicmath Large	0.00	0.00	0.00
Bitcount	6.33	6.33	0.00
Blowfish Dec	16.92	10.68	5.07
Blowfish Enc	16.92	10.68	5.07
Cjpeg	15.80	13.40	0.00
CRC32	0.00	0.00	0.00
Dijkstra Large	24.88	0.00	0.00
FFT	0.00	0.00	0.00
Patricia	0.00	0.00	0.00
Pegwit	8.12	0.02	0.55
Rijndael Dec	2.27	2.27	-6.11
Rijndael Enc	2.27	2.27	-7.45
Sha	0.00	0.00	0.00

Table 3-3 shows the percentage performance gain (in terms of nodes covered) of the proposed RLFF method over other template selection approaches (i.e. MFF, MLFF, ILP and LFF). It can be observed that the RLFF approach outperforms the LFF method in only two applications (i.e. Rijndael Dec and Rijndael Enc). In particular, the RLFF method outperforms the LFF method by over 15% in these two applications. On average, the proposed RLFF method outperforms the MFF and MLFF method by 11.0% and 6.6% respectively. Moreover, when compared to the ILP approach, the RLFF approach has a higher number of nodes covered in a number of applications and is comparable in the remaining ones.

Table 3-3: Percentage performance gain of RLFF over MFF, MLFF, ILP and LFF

Application	MFF	MLFF	ILP	LFF
Adpcm Dec	5.80	5.80	0.00	0.00
Adpcm Enc	12.51	12.51	0.00	0.00
Aes	32.38	9.33	0.38	0.00
Basicmath Large	0.00	0.00	0.00	0.00
Bitcount	6.33	6.33	0.00	0.00
Blowfish Dec	16.92	10.68	5.07	0.00
Blowfish Enc	16.92	10.68	5.07	0.00
Cjpeg	15.80	13.40	0.00	0.00
CRC32	0.00	0.00	0.00	0.00
Dijkstra Large	24.88	0.00	0.00	0.00
FFT	0.00	0.00	0.00	0.00
Patricia	0.00	0.00	0.00	0.00
Pegwit	8.12	0.02	0.55	0.00
Rijndael Dec	18.13	18.13	8.45	15.51
Rijndael Enc	18.18	18.18	6.94	15.55
Sha	0.00	0.00	0.00	0.00

Based on the experiments discussed in this section, it can be deduced that template selection based on frequently occurring templates (using the MFF and MLFF approach) can result in the disposal of a notable number of overlapping templates that are likely to have a larger number of operations. This concurs with our earlier analysis in Section 3.3.2 which shows that a significant number of frequently occurring templates (i.e. basic templates) are contained within the superset templates. Hence, we can conclude that template selection strategies that give preference to the selection of large templates can lead to better results in all the applications considered.

3.4 SUMMARY

In this chapter, we have presented an efficient strategy for the rapid selection of high performance custom instructions for RISPs based on the graph covering approach. The proposed LFF selection strategy is based on our findings that over 77% of the template instances are high frequency basic templates that are incorporated into larger templates. The proposed RLFF approach has been shown to benefit from cases in which overlapping templates could be re-introduced to maximize performance. Comparisons with previously reported approximate strategies such as MFF, MLFF and ILP show that the proposed techniques select custom instructions with highest performance. It is noteworthy that the custom instruction strategies presented in this chapter for selecting custom instructions that can lead to high performance are suitable for RISPs as well as configurable processors.

The experimental results in this chapter reveal that the proposed approaches (i.e. LFF and RLFF) can rapidly select custom instructions with highest performance (in terms of number of nodes covered). However, the number of nodes covered (or number of operations in the ISA that can be mapped to hardware) does not provide a realistic measure of the effective performance gain that can be achieved by the custom instructions. This is due to the fact that the hardware latencies of the custom instructions have not been taken into consideration. However, manually implementing the selected custom instructions in hardware to obtain the hardware latencies is a time consuming process and does not facilitate efficient design exploration. In the next chapter, we will present a reliable high-level estimation approach to rapidly predict the required critical path of the selected custom instructions when they are implemented on the RFU.

CHAPTER 4

RAPID AREA-TIME ESTIMATION OF CUSTOM INSTRUCTIONS FOR FPGA

In the previous chapter, we presented two approaches for custom instruction selection that were shown to outperform existing methods. However, the performance metric used in the previous chapter is in terms of the number of ISA operations that are implemented as custom instructions. This metric does not provide a realistic measure of the effective performance gain that can be achieved as it does not take into account the hardware latencies of the custom instructions. Implementing custom instructions in hardware using conventional design entry methods, e.g. writing Hardware Description Languages (HDL) or schematic-based drawing, to obtain the hardware latencies is time consuming and does not facilitate effective design exploration.

In this chapter, we propose a novel strategy to estimate critical path delays of custom instructions on LUT based FPGAs. In contrast to existing high-level estimation methods discussed in Section 2.4.3, the proposed technique takes into account the FPGA architectural constraints and synthesis optimizations for maximizing the utilization of the FPGA resources. It also does not rely on a pre-characterization step, which limits the scope of representing all possible combinations of the design under examination. In addition, the proposed estimation strategy is performed on the IR of ANSI C applications, which are commonly employed in embedded applications. It is noteworthy that the proposed strategy is targeted towards state-of-the-art FPGA architectures (e.g. the Xilinx

Virtex devices), and can be easily extended for newer and future FPGA devices that are likely to incorporate similar programmable logic elements. Since the proposed technique targets reconfigurable structures that are similar to commercially available technologies, it can be readily integrated with existing hardware synthesis tools.

Our investigations show that the average estimated critical paths of 150 custom instructions from sixteen applications using the proposed method are within 3% of those obtained using hardware synthesis (based on Xilinx ISE Version 11.2). Our experimental results also demonstrate the importance of high level estimation by showing that the proposed LFF custom instruction selection approach outperforms the RLFF approach (contrary to the results in the previous chapter that do not take into account the custom instruction latencies on FPGAs). Finally, we show that the proposed high level estimation method can also accurately estimate the area utilization of the custom instructions on FPGA. In particular, our results show that the average estimated area utilization of 150 custom instructions using the proposed method are within 1% of those obtained using hardware synthesis. Part of the work in this chapter has been published in [J-5][C-6][C-7].

In the following section, we will first present the target RISP model, which incorporates a LUT based RFU that is similar to commercial FPGA architectures. Next, the proposed high level estimation technique is presented. Finally, experimental results will be shown to demonstrate the accuracy of the proposed method for estimating the critical path of the custom instructions. In addition, using the proposed method, we will reexamine the performance of the custom instruction selection approaches discussed in the previous chapter in order to demonstrate the importance of high level estimation for instruction set customization. Finally, we evaluate the accuracy of the proposed method for estimating area utilization of custom instructions on FPGA.

4.1 TARGET RISP MODEL

The target RISP architecture model, which is shown in Figure 4-1, is a four-wide Very Long Instruction Word (VLIW) architecture that has been extended with an RFU for implementing custom instructions (i.e. CI_1, CI_2, ..., CI_n). The RFU consists of a two dimensional array of logic blocks that are similar to the commercially available FPGA architectures shown in Figure A-1. In this thesis, we will use the terms *logic block* and *FPGA logic block* interchangeably. The RFU can also incorporate Coarse Grained Arithmetic Units (CGAUs) to implement complex arithmetic functions.

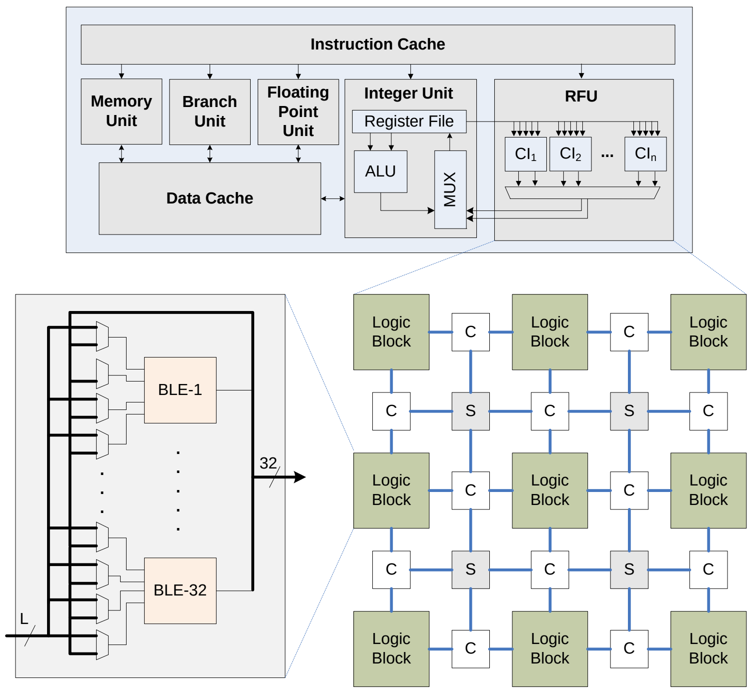


Figure 4-1: Overview of target RISP

The following lists the assumptions of the target model and the constraints it imposes on the custom instruction realizations:

- The target RISP only provides coupling logic between the integer unit and RFU. Hence, the RFU only facilitates custom instruction implementations of integer type operations.
- The RFU cannot directly access the data cache. Hence, memory operations are not permissible in custom instructions. The RFU obtains input data from the register file, and outputs the results to a multiplexer (MUX) that facilitates the sharing of register files between the Arithmetic Logic Unit (ALU) and custom logic.
- The target RISP consists of a register file with five read ports and two write ports for the RFU, and hence only custom instructions with at most five-input and two-output operands are considered for implementation. The data width of the operands is 32-bits.
- Each logic block in the RFU consists of 32 Basic Logic Elements (BLEs). Each BLE is composed of a *K*-input LUT (*K*-LUT) and fast carry-logic as shown in Figure A-1.
- For simplicity, we assume pipelining and multi-cycle register file accesses are not supported in the target architecture. However, multi-cycle custom instruction implementation is permissible.

The model shown in Figure 4-1 will be used as the target RISP in the rest of the thesis. As such, experimental results pertaining to the number of clock cycles for the baseline processor and target RISP are calculated based on profiling results of Trimaran that has been configured to target a four-wide VLIW architecture.

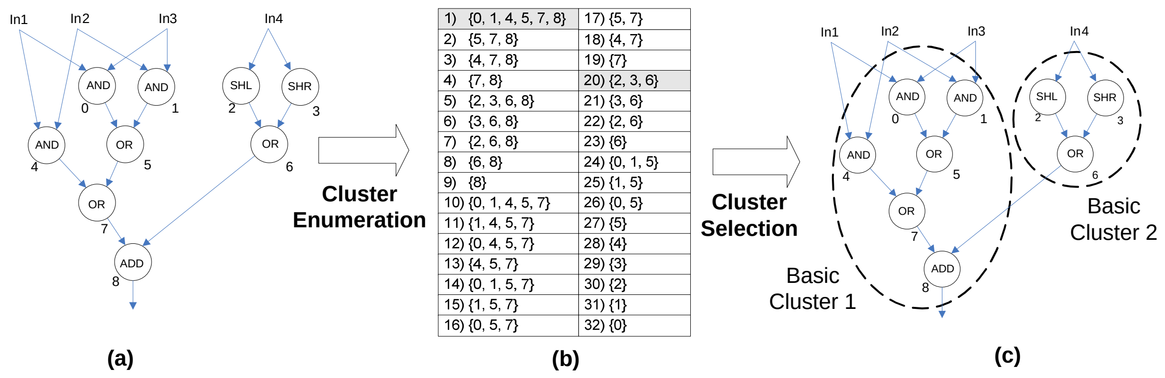
4.2 PROPOSED CLUSTER GENERATION TECHNIQUE

The proposed high-level estimation technique is also termed as *cluster generation*. The following definitions are given to aid the description of the proposed cluster generation technique.

Definition 4-1: A template (custom instruction) can be defined as a directed graph $G_i = (V_i, E_i)$ for $i = 1,2,\dots,n$ and n is the number of templates obtained from template selection (see previous chapter), where:

- A vertex $v \in V_i$ for $1 \le i \le n$ is a primitive integer operation in a compiler's IR. These operations can be categorized as 1) arithmetic i.e. addition (*ADD*), subtraction (*SUB*), multiplication (*MUL*), 2) logical (*AND*, *OR*, *XOR*), and 3) relational e.g. logical/arithmetic shift by a constant/non-constant (*SHL*, *SHR*, *SHRA*). Each vertex is also associated with at most two input ports and one output port.
- An arc $e = (u, v) \in E_i$ indicates a data transfer from vertex *u* to vertex *v*, whereby the output port of *u* is connected to one of the input ports of *v*.

Definition 4-2: A basic cluster $C_i^j = (V_i^j, E_i^j)$ is a sub-graph of a template G_i, which can be implemented either: 1) on a single FPGA logic block (we assume the size of the logic block is equivalent to the custom instruction bit-width), or 2) using CGAUs. None of the basic clusters in G_i overlap, i.e. $V_i^j \cap V_i^k = \varnothing$ and $E_i^j \cap E_i^k = \varnothing$ for $j \ne k$. In addition $\bigcup_{j=1}^{c} V_i^j = V_i$ and $\bigcup_{j=1}^{c} E_i^j = E_i$, where c is the number of basic clusters in G_i.



1) {0, 1, 4, 5, 7, 8}	17) {5, 7}
2) {5, 7, 8}	18) {4, 7}
3) {4, 7, 8}	19) {7}
4) {7, 8}	20) {2, 3, 6}
5) {2, 3, 6, 8}	21) {3, 6}
6) {3, 6, 8}	22) {2, 6}
7) {2, 6, 8}	23) {6}
8) {6, 8}	24) {0, 1, 5}
9) {8}	25) {1, 5}
10) {0, 1, 4, 5, 7}	26) {0, 5}
11) {1, 4, 5, 7}	27) {5}
12) {0, 4, 5, 7}	28) {4}
13) {4, 5, 7}	29) {3}
14) {0, 1, 5, 7}	30) {2}
15) {1, 5, 7}	31) {1}
16) {0, 5, 7}	32) {0}

Figure 4-2: (a) Cluster enumeration, **(b)** basic cluster instances**, (c)** cluster selection

Cluster generation partitions the templates into a set of basic clusters. This resembles the technology mapping process, where a set of basic clusters that effectively cover the template is identified. It is worth mentioning that unlike existing works in technology mapping, cluster generation operates on the high-level representation of the custom instructions.

The cluster generation process consists of the following steps: 1) *cluster enumeration* and 2) *cluster selection*. These steps are illustrated in Figure 4-2, where Figure 4-2(a) shows an example template with nine vertices (primitive operations). The cluster enumeration step decomposes the data-path into a set of basic cluster instances, where each instance represents a connected sub-graph that can be realized with an FPGA logic block. Since the data width of the template is 32-bits, each basic cluster can be essentially mapped to a set of 32 BLEs with the same configuration. In the example, there are 32 enumerated cluster instances for K = 4 (see Figure 4-2(b)). A set of basic clusters is then selected to effectively cover the template in order to meet a certain criteria. For example in Figure 4-2(c), basic clusters 1 and 20 are selected from the enumerated set such that the number of basic clusters required to cover the data-path is minimized. In this example, two FPGA logic blocks will be required to realize the template in Figure 4-2(a).

The cluster generation process does not require actual hardware synthesis. Instead, it relies on certain rule-sets that take into account the size of the LUT for mapping logical and relational operations, and the carry-logic architecture for mapping addition/subtraction operations. The validity of the proposed rules has been verified by implementing the basic clusters in VHDL and synthesizing them using the Xilinx FPGA design tool.

It is worth mentioning that the proposed approaches in this chapter are applicable to FPGAs that consist of BLEs with LUT of any arbitrary input *K* and a carry-logic structure as shown in Figure A-1. Hence, although the experiments in this chapter are based on FPGA devices which consist of BLEs with 4-input LUT (similar to Xilinx Virtex-2 [43] and Virtex-4 devices [108]), the proposed approaches can be easily extended for newer and future FPGA architectures. For example, the recent Virtex-5 devices incorporates 6-input LUTs with similar carry-chain primitives that can be used to implement any 6-input function or two dual-output 5-input functions [109]. The proposed method can be used to estimate the hardware area-time of these devices by specifying the appropriate value of *K* (e.g. K = 5 or 6). Further optimization techniques can be incorporated to enable the mapping of two 5-input functions onto a single BLE in the Virtex-5 devices. It has also been predicted that future FPGAs are likely to incorporate similar BLEs with *K*-LUTs and carry-logic [110].

4.2.1 Cluster Enumeration

We have adopted the enumeration algorithm in [70] to identify all the basic cluster instances from the selected templates. As discussed in the previous chapter, the method in [70] employs a binary tree search approach that prunes unexplored sub-graphs from the search space if they violate a certain set of rules. Valid basic cluster instances, comprising connected sub-graphs where each node is a primitive operation, must comply with a set of rules that enable them to be mapped onto the BLEs or CGAUs. We will describe these rules with the help of the example template in Figure 4-3(a), and Figure 4-3(b) that shows the implementation of an *ADD* operation using a BLE similar to those found in the 4-input LUT based Xilinx Virtex device [43][108].

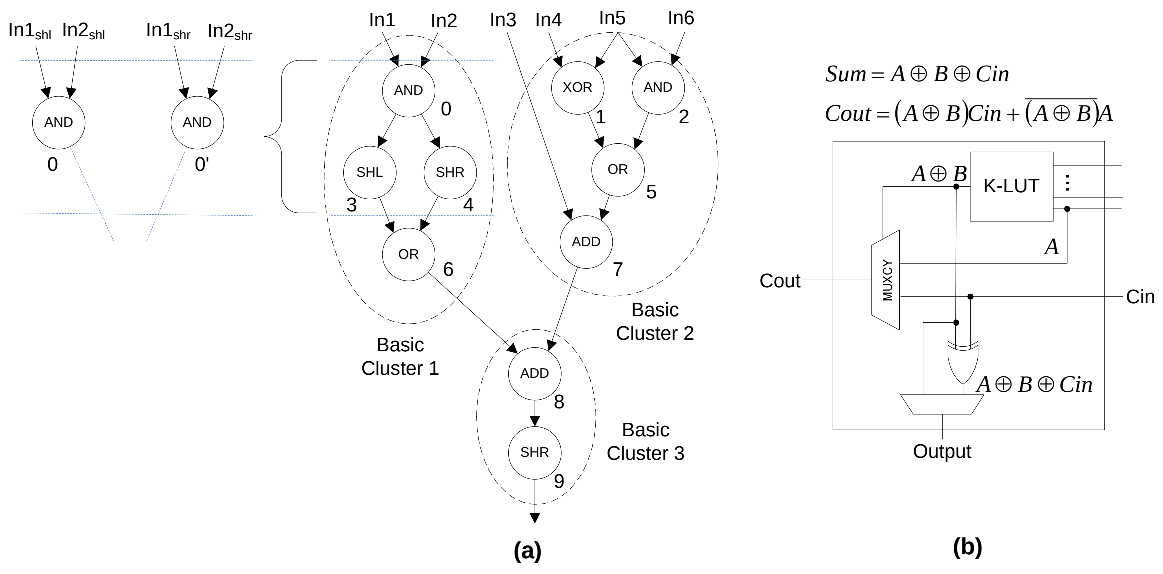


Figure 4-3: (a) Custom instruction with 3 basic clusters**, (b)** implementing an *ADD* operation in a BLE

Two sets of legality checks have been used to determine a basic cluster. These legality checks consist of a set of rules which are used to ensure that the basic clusters conform to the requirements in Definition 4-2. The legality checks have been formulated based on our understanding on how the operations are mapped onto the BLEs of the target FPGA. The first set of legality checks determines the primitive operations that can be included in the basic cluster. An operation can be included in a basic cluster if:

1. *It is a logical or shift-by-constant operation and the basic cluster does not consist of any arithmetic operations* (e.g. Basic Cluster 1 in Figure 4-3(a)). The programmable *K*-input LUTs (*K*-LUTs) can implement non-arithmetic functions of up to *K* inputs.
2. *It is a logical operation that is executed before a single ADD/SUB operation in the cluster* (e.g. Basic Cluster 2 in Figure 4-3(a)). The *ADD*/*SUB* operation can be mapped onto the LUT based BLEs to exploit the fast carry chains as shown Figure 4-3(b) [111]. Since the output of the LUT is the partial sum ($A \oplus B$), all logical operations must be executed first to generate the required operands (i.e. *A* and *B*) for the *ADD* operation.
3. *It is a shift-by-constant operation and the cluster contains a single ADD/SUB operation* (e.g. Basic Cluster 3 in Figure 4-3(a)). The shifted value of an addition/subtraction result by a constant can be realized by configuring the routing architecture to feed the suitable data range as inputs to another BLE.
4. *It is an ADD/SUB operation and the cluster does not contain any other arithmetic operations*. A LUT based BLE can only implement a single *ADD*/*SUB* operation

effectively by exploiting the fast carry chain. Other arithmetic operations (e.g. multiplication, division, etc.) are mapped to the CGAUs.

The second set of legality checks evaluates whether the operations that have been included in a basic cluster conform to the input-output constraints of the BLE:

1. *The total number of cluster inputs is at most K with only one output*. The maximum number of input ports and output ports for a LUT is *K* and 1, respectively. Note that this legality check can be easily modified to cater to newer FPGA devices with more than one output port (e.g. [109]).
2. *One of the inputs to the ADD/SUB operation must be directly connected to an external input of the basic cluster* (e.g. Basic Cluster 2 in Figure 4-3(a)). As shown in Figure 4-3(b), the carry-out (i.e. *Cout*) is selected from either the carry-in (i.e. *Cin*) or an input operand (i.e. *A*) that is also fed directly to the input pin of the LUT.

If the basic cluster contains only logical and *ADD*/*SUB* operations, the number of inputs can be easily derived by evaluating the external inputs to the basic cluster. For example in Figure 4-3(a), it can be easily observed that Basic Cluster 2 requires 4 inputs. However, basic clusters with shift-by-constant operations must be evaluated differently. For example, the hardware synthesis tool will not be able to map the two physical inputs (i.e. *In1* and *In2*) of Basic Cluster 1 in Figure 4-3(a) to only two LUT pins. *SHL* and *SHR*, which are shift-by-constant operations, will result in *In1* and *In2* being routed to four input pins in order to realize the required operands for node 6. To illustrate this, Figure 4-3(a) shows an equivalent representation of the *AND*-*SHL*-*SHR* operations of the original template. It can be observed that node 0 can be duplicated in order to produce a pair of operands for node 6. The inputs to the duplicated *AND* operations comprise the shifted values of *In1* and *In2*. It is evident from this representation that Basic Cluster 1 requires 4 inputs (i.e. $In1_{shl}$, $In2_{shl}$, $In1_{shr}$, $In2_{shr}$) instead of 2.

Algorithm 4-1

```
CALCULATE-CLUSTER-INPUTS ($C_i$)
1.  Identify roots of sub-trees and store in roots
2.  input_count = 0
3.  for each root i in roots {
4.      Perform reverse DFS to identify members j of sub-tree with root i
5.      for each node j in the sub-tree
6.          if new operand is found, increment input_count
7.  }
8.  return input_count
```

Figure 4-4: Pseudo code to compute number of inputs in a basic cluster

Given a basic cluster in the form of a directed graph $C_i^j = (V_i^j, E_i^j)$, the pseudo code in Figure 4-4 computes the number of inputs of the basic cluster and returns the result as *input_count*. The function identifies sub-trees in the basic cluster that are rooted at a shift-by-constant operation or the output node. For example in Figure 4-3(a), the root nodes of Basic Cluster 1 are nodes 3, 4, and 6. The function first identifies all the root nodes in the basic cluster (line 1 in Figure 4-4) before finding the sub-tree members of the corresponding root nodes using the reverse Depth First Search (DFS) algorithm (line 4). A sub-tree can only have one shift operation that must be a root node. Hence in Basic Cluster 1, there are three sub-trees with the following node members: {3, 0}, {4, 0}, and

{6}. Each of these sub-trees is evaluated independently and the variable *input_count* is incremented whenever a new operand is detected in one of the node members (lines 5-6 of Figure 4-4). In the example, two new operands will be detected for each of the sub-tree {3,0} and {4, 0}.

4.2.2 Cluster Selection

We have adopted the conflict graph approach described in the previous chapter to select a set of basic clusters to cover the selected templates. The LFF objective function is used to heuristically select basic clusters with large number of primitive operations, as we aim to minimize the required number of basic clusters of the RFU, which will indirectly maximize the hardware utilization.

4.3 CRITICAL PATH DELAY ESTIMATION

Figure 4-5 shows the critical path delay estimation of a template that has been partitioned to basic clusters. Note that the critical path is the path with the maximum number of basic clusters from the input buffer to output buffer. The timing characteristics for the various logic and interconnect is also shown in Figure 4-5 with their default values, which are obtained empirically or from data sheets of the target device. In the example, the target device is Xilinx Virtex-4 xc4vlx40-10ff1148. In general, the critical path delay estimation model of a template is shown in (4-1), where *m* is the number of basic clusters in the critical path of the template (*m* = 3 in the example). The parameters in equation (4-1) are shown in Figure 4-5.

$$T_{template} = t_{ibuf} + t_{buf-lut} + \sum_{i}^{m} t_{lb}^{i} + \sum_{i}^{m-1} t_{lb-lb}^{i} + t_{lb-obuf} + t_{obuf} \qquad (4\text{-}1)$$

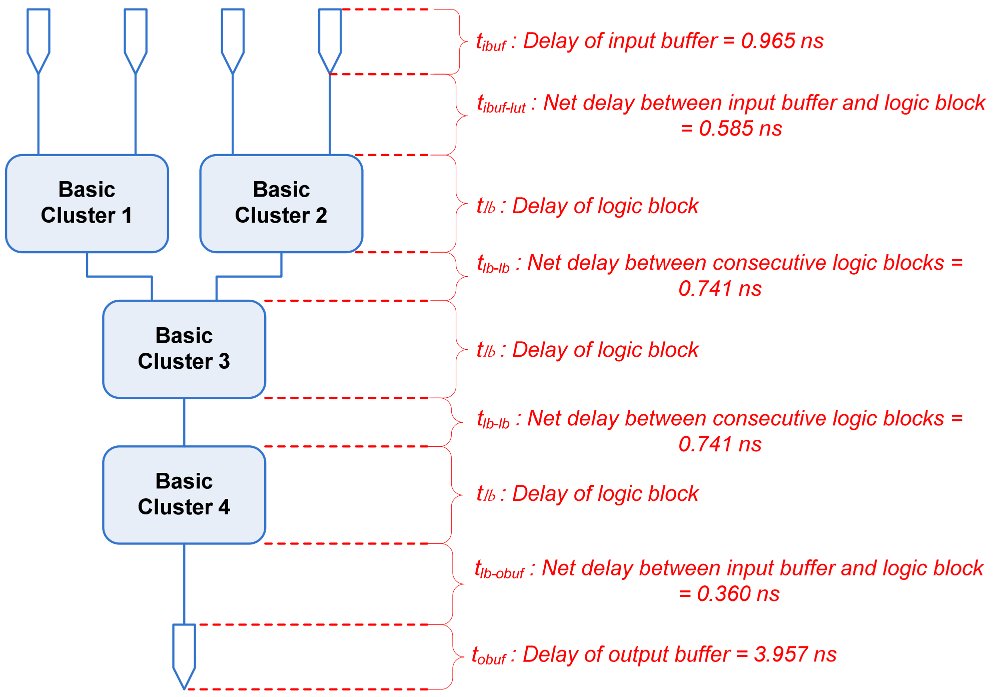

Basic Cluster 1
Basic Cluster 2
Basic Cluster 3
Basic Cluster 4
t_{ibuf} : Delay of input buffer = 0.965 ns
$t_{ibuf-lut}$: Net delay between input buffer and logic block = 0.585 ns
t_{lb} : Delay of logic block
t_{lb-lb} : Net delay between consecutive logic blocks = 0.741 ns
t_{lb} : Delay of logic block
t_{lb-lb} : Net delay between consecutive logic blocks = 0.741 ns
t_{lb} : Delay of logic block
$t_{lb-obuf}$: Net delay between input buffer and logic block = 0.360 ns
t_{obuf} : Delay of output buffer = 3.957 ns

Figure 4-5: Example of critical path delay estimation of template

The delay of a logic block *i* (i.e. t_{lb}^{i}) consists of two components, i.e. the delay of the LUT (i.e. t_{lut}^{i}) and the delay of the carry chain. The estimation model of t_{lb}^{i} is shown in (4-2), where x_i indicates the existence of an *ADD* operation in basic cluster *i*:

$$t_{lb}^{i} = t_{lut}^{i} + x_i \cdot \left(t_{muxcy\_so} + n_{cc}^{i} \cdot t_{muxcy\_cio} + t_{xorcy\_cio}\right)$$
$$where \quad x_i = \begin{cases} 1 & if \;\; ADD \; exists \; in \; basic \; cluster \; i, \\ 0 & else. \end{cases} \qquad (4\text{-}2)$$

The parameters t_{muxcy_so}, t_{muxcy_cio} and t_{xorcy_cio} correspond to the delay of the multiplexer and XOR components in the carry-chain structure with the following values (obtained from the data sheets): 0.366, 0.044 and 0.360 respectively. n_{cc}^{i} is the number of multiplexers in the carry-chain path (excluding the first one) and is assumed to be 30. The estimation model in (4-2) can be easily verified from Figure 4-6, which shows the delay path of an addition operation. Figure 4-7 shows the corresponding synthesis timing report.

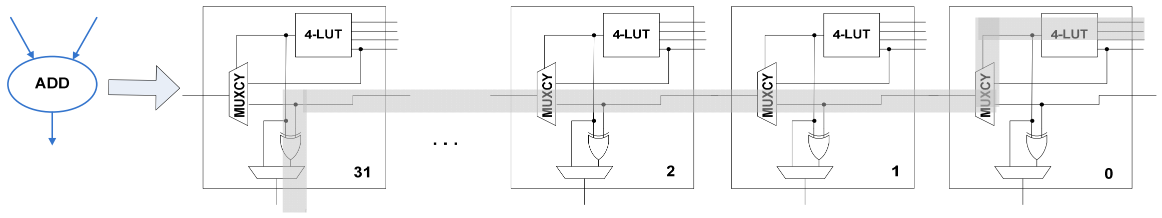
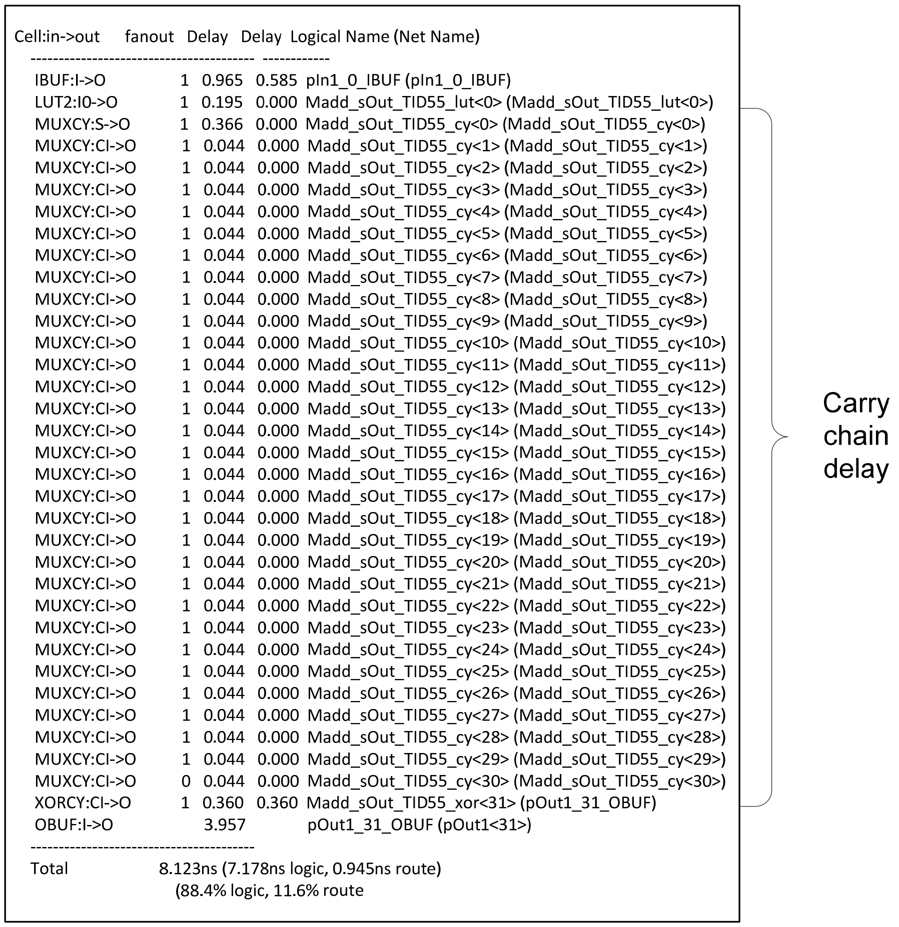


Figure 4-6: Delay path of an addition operation



```
Cell:in->out     fanout   Delay   Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   IBUF:I->O              1  0.965  0.585  pIn1_0_IBUF (pIn1_0_IBUF)
   LUT2:I0->O             1  0.195  0.000  Madd_sOut_TID55_lut<0> (Madd_sOut_TID55_lut<0>)
   MUXCY:S->O             1  0.366  0.000  Madd_sOut_TID55_cy<0> (Madd_sOut_TID55_cy<0>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<1> (Madd_sOut_TID55_cy<1>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<2> (Madd_sOut_TID55_cy<2>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<3> (Madd_sOut_TID55_cy<3>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<4> (Madd_sOut_TID55_cy<4>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<5> (Madd_sOut_TID55_cy<5>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<6> (Madd_sOut_TID55_cy<6>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<7> (Madd_sOut_TID55_cy<7>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<8> (Madd_sOut_TID55_cy<8>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<9> (Madd_sOut_TID55_cy<9>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<10> (Madd_sOut_TID55_cy<10>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<11> (Madd_sOut_TID55_cy<11>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<12> (Madd_sOut_TID55_cy<12>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<13> (Madd_sOut_TID55_cy<13>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<14> (Madd_sOut_TID55_cy<14>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<15> (Madd_sOut_TID55_cy<15>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<16> (Madd_sOut_TID55_cy<16>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<17> (Madd_sOut_TID55_cy<17>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<18> (Madd_sOut_TID55_cy<18>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<19> (Madd_sOut_TID55_cy<19>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<20> (Madd_sOut_TID55_cy<20>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<21> (Madd_sOut_TID55_cy<21>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<22> (Madd_sOut_TID55_cy<22>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<23> (Madd_sOut_TID55_cy<23>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<24> (Madd_sOut_TID55_cy<24>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<25> (Madd_sOut_TID55_cy<25>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<26> (Madd_sOut_TID55_cy<26>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<27> (Madd_sOut_TID55_cy<27>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<28> (Madd_sOut_TID55_cy<28>)
   MUXCY:CI->O            1  0.044  0.000  Madd_sOut_TID55_cy<29> (Madd_sOut_TID55_cy<29>)
   MUXCY:CI->O            0  0.044  0.000  Madd_sOut_TID55_cy<30> (Madd_sOut_TID55_cy<30>)
   XORCY:CI->O            1  0.360  0.360  Madd_sOut_TID55_xor<31> (pOut1_31_OBUF)
   OBUF:I->O                 3.957         pOut1_31_OBUF (pOut1<31>)
  ----------------------------------------
  Total                 8.123ns (7.178ns logic, 0.945ns route)
                          (88.4% logic, 11.6% route
```

Figure 4-7: Synthesis timing report for an addition operation

The estimation model in (4-1) and (4-2) must be extended to take into consideration consecutive basic clusters with addition operations in the critical path. Figure 4-8 illustrates the delay path of two consecutive addition operations (where each adder is in a separate basic cluster). It can be observed that the carry chain delay of the second addition operation partially overlaps with the first and hence, should not be included in the estimation model. This can be verified in Figure 4-9, which shows the synthesis timing report for the template that composed of two consecutive basic clusters with addition operations in the critical path.

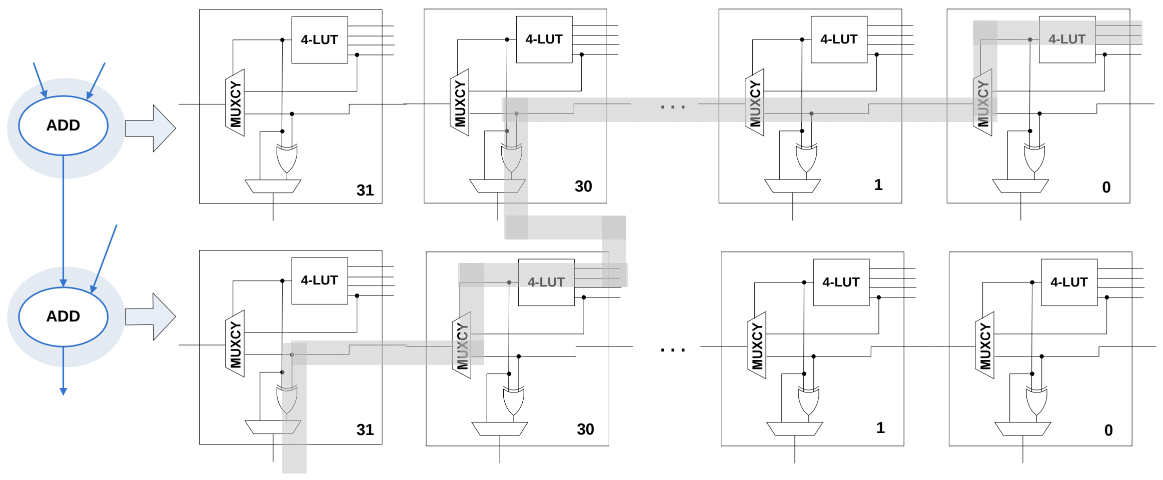
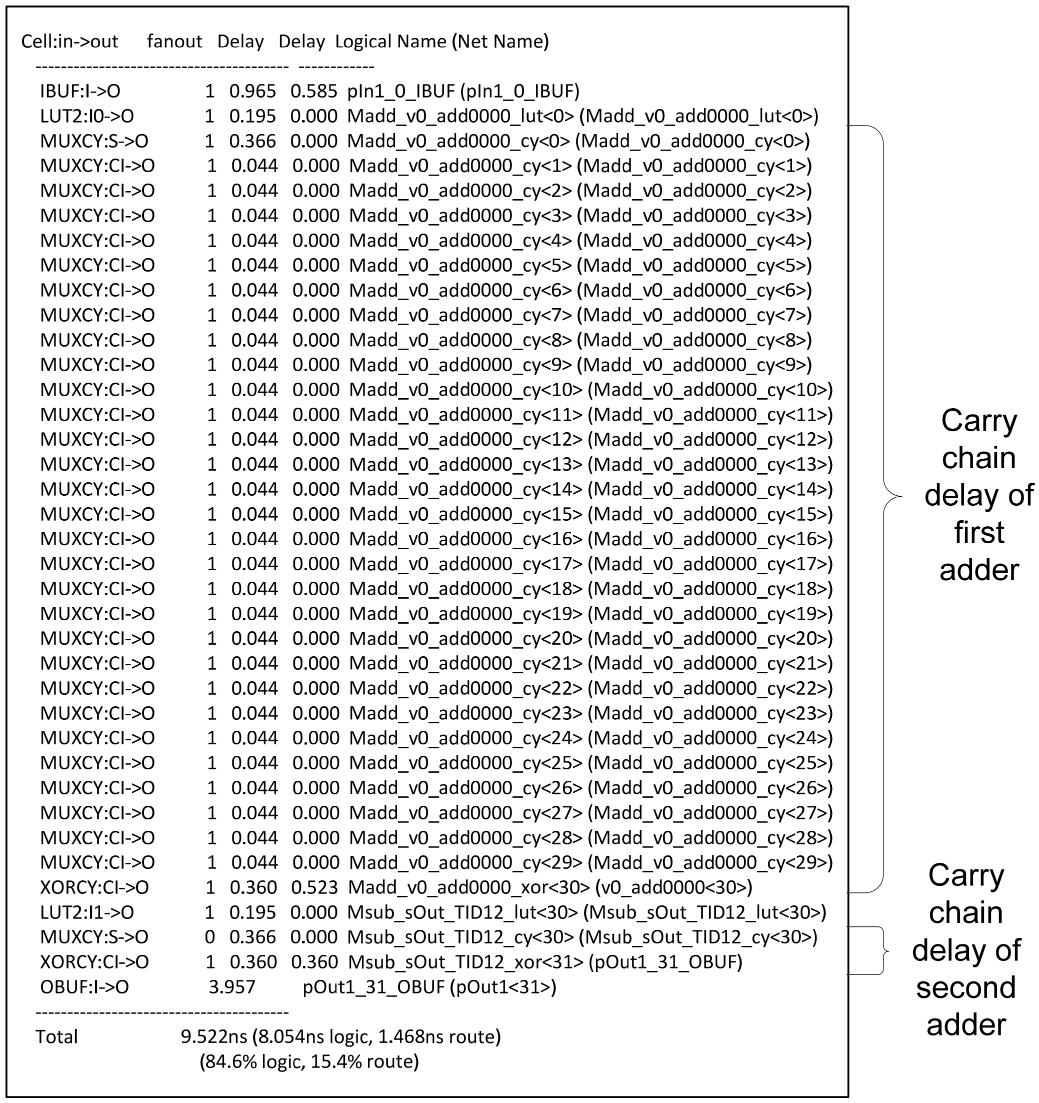


Figure 4-8: Delay path of two consecutive additions



```
Cell:in->out     fanout   Delay   Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   IBUF:I->O             1  0.965   0.585  pIn1_0_IBUF (pIn1_0_IBUF)
   LUT2:I0->O            1  0.195   0.000  Madd_v0_add0000_lut<0> (Madd_v0_add0000_lut<0>)
   MUXCY:S->O            1  0.366   0.000  Madd_v0_add0000_cy<0> (Madd_v0_add0000_cy<0>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<1> (Madd_v0_add0000_cy<1>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<2> (Madd_v0_add0000_cy<2>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<3> (Madd_v0_add0000_cy<3>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<4> (Madd_v0_add0000_cy<4>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<5> (Madd_v0_add0000_cy<5>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<6> (Madd_v0_add0000_cy<6>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<7> (Madd_v0_add0000_cy<7>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<8> (Madd_v0_add0000_cy<8>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<9> (Madd_v0_add0000_cy<9>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<10> (Madd_v0_add0000_cy<10>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<11> (Madd_v0_add0000_cy<11>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<12> (Madd_v0_add0000_cy<12>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<13> (Madd_v0_add0000_cy<13>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<14> (Madd_v0_add0000_cy<14>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<15> (Madd_v0_add0000_cy<15>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<16> (Madd_v0_add0000_cy<16>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<17> (Madd_v0_add0000_cy<17>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<18> (Madd_v0_add0000_cy<18>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<19> (Madd_v0_add0000_cy<19>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<20> (Madd_v0_add0000_cy<20>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<21> (Madd_v0_add0000_cy<21>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<22> (Madd_v0_add0000_cy<22>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<23> (Madd_v0_add0000_cy<23>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<24> (Madd_v0_add0000_cy<24>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<25> (Madd_v0_add0000_cy<25>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<26> (Madd_v0_add0000_cy<26>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<27> (Madd_v0_add0000_cy<27>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<28> (Madd_v0_add0000_cy<28>)
   MUXCY:CI->O           1  0.044   0.000  Madd_v0_add0000_cy<29> (Madd_v0_add0000_cy<29>)
   XORCY:CI->O           1  0.360   0.523  Madd_v0_add0000_xor<30> (v0_add0000<30>)
   LUT2:I1->O            1  0.195   0.000  Msub_sOut_TID12_lut<30> (Msub_sOut_TID12_lut<30>)
   MUXCY:S->O            0  0.366   0.000  Msub_sOut_TID12_cy<30> (Msub_sOut_TID12_cy<30>)
   XORCY:CI->O           1  0.360   0.360  Msub_sOut_TID12_xor<31> (pOut1_31_OBUF)
   OBUF:I->O                3.957          pOut1_31_OBUF (pOut1<31>)
  ----------------------------------------
  Total                 9.522ns (8.054ns logic, 1.468ns route)
                           (84.6% logic, 15.4% route)
```

Carry chain delay of first adder
Carry chain delay of second adder

Figure 4-9: Synthesis timing report for two consecutive additions

In order to take into account the partial overlapping carry chain delay of the second addition operation, the estimation model in (4-1) is modified as shown in (4-3), where the modified logic block delay consists of two components (i.e. t_{lb-1}^{i} and t_{lb-2}^{i}) as shown in (4-4) and (4-5). *a* denotes the number of *adder groups*, where each adder group consists of either one disjoint basic cluster with *ADD* operation or several consecutive basic clusters with *ADD* operations.

$$T_{template} = t_{ibuf} + t_{buf-lut} + \sum_{i}^{a} t_{lb-1}^{i} + \sum_{i}^{m-a} t_{lb-2}^{i} + \sum_{i}^{m-1} t_{lb-lb}^{i} + t_{lb-obuf} + t_{obuf} \qquad (4\text{-}3)$$

$$t_{lb-1}^{i} = t_{lut}^{i} + x_i \cdot \left(t_{muxcy\_so} + n_i^{cc} \cdot t_{muxcy\_cio} + t_{xorcy\_cio} \right) \qquad (4\text{-}4)$$

$$t_{lb-2}^{i} = t_{lut}^{i} + x_i \cdot \left(t_{muxcy\_so} + t_{xorcy\_cio} \right) \qquad (4\text{-}5)$$

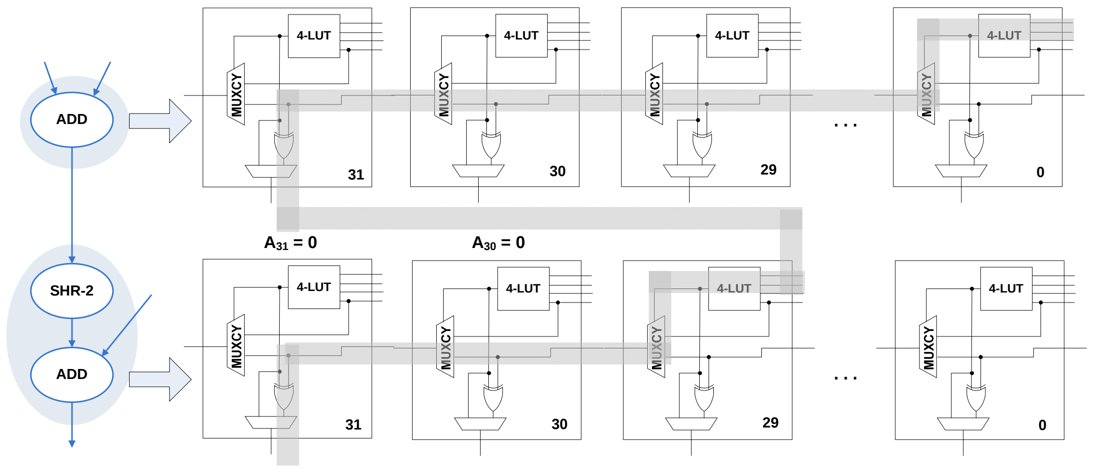


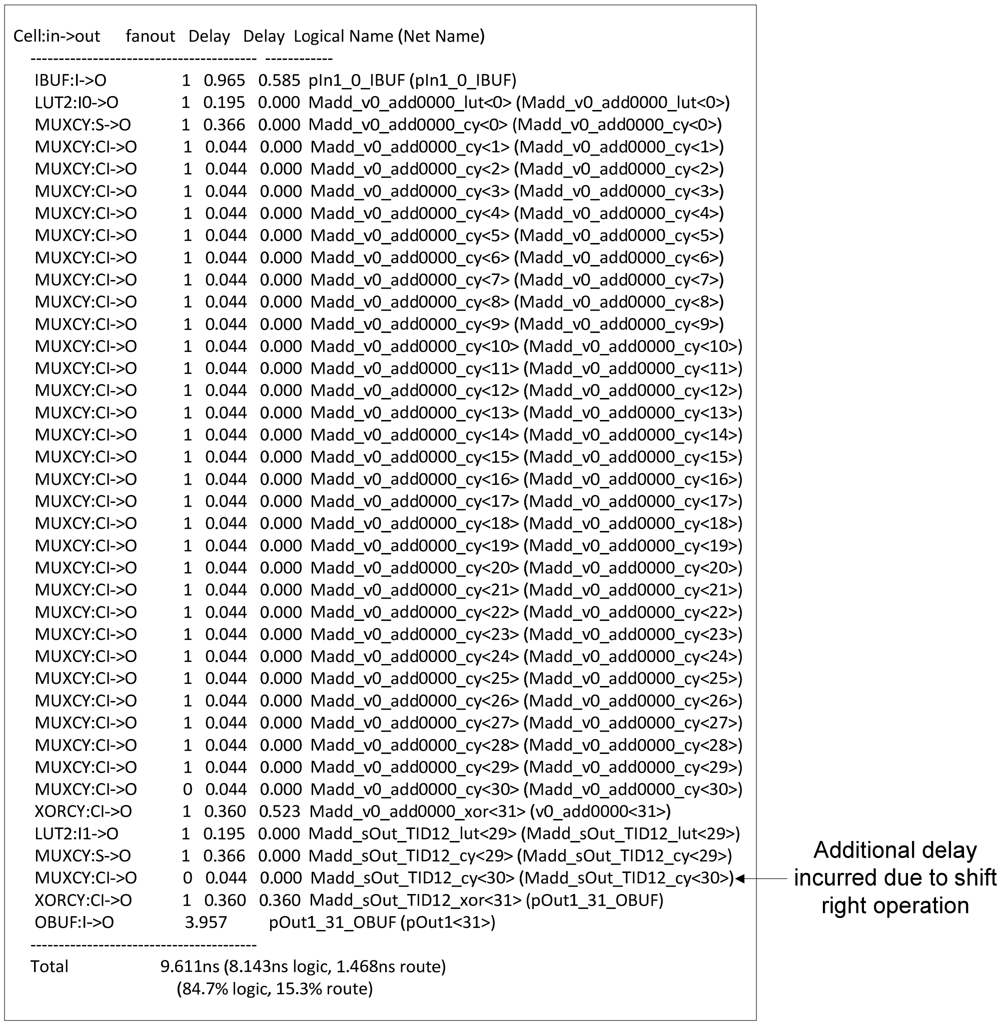
Figure 4-10: Delay path of two consecutive additions with a right shift operation in between

The estimation model for the logic block delay can be further extended to take into account the *effective shift-right constant offset*[10] that occurs between two consecutive additions. Figure 4-10 illustrates this scenario, whereby the first basic cluster consists of an *ADD* operation, while the second basic cluster consists of a shift-right-constant operation (by a factor of two) followed by an *ADD* operation. It can be observed that due to the shift operation, the full result of the first addition must be obtained and shifted to the right by two bits, before being fed to the second basic cluster. This incurs additional

[10] In cases where the basic cluster consists of more than one shift operation preceding the *ADD* operation, the effective shift right offset must be computed. This can be calculated using existing bit-width analysis approaches such as that proposed in [112].

carry chain delay in the second basic cluster as shown in Figure 4-11. In order to take into account the addition delay incurred by shift-right-by-constant operations between two consecutive *ADD* operations, the estimation model for the logic block delay (t_{lb-2}) in (4-5) is extended as shown in (4-6) to incorporate a new parameter (i.e. n^i_{shr}), which denotes the effective shift right constant offset of the basic cluster *i*.

$$t^i_{lb-2} = t^i_{lut} + x_i \cdot \left(t_{muxcy\_so} + \left(n^i_{shr} - 1\right) \cdot t_{muxcy\_cio} + t_{xorcy\_cio}\right) \qquad (4\text{-}6)$$



```
Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ---------------------------------------  ------------
    IBUF:I->O              1  0.965   0.585  pIn1_0_IBUF (pIn1_0_IBUF)
    LUT2:I0->O             1  0.195   0.000  Madd_v0_add0000_lut<0> (Madd_v0_add0000_lut<0>)
    MUXCY:S->O             1  0.366   0.000  Madd_v0_add0000_cy<0> (Madd_v0_add0000_cy<0>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<1> (Madd_v0_add0000_cy<1>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<2> (Madd_v0_add0000_cy<2>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<3> (Madd_v0_add0000_cy<3>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<4> (Madd_v0_add0000_cy<4>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<5> (Madd_v0_add0000_cy<5>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<6> (Madd_v0_add0000_cy<6>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<7> (Madd_v0_add0000_cy<7>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<8> (Madd_v0_add0000_cy<8>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<9> (Madd_v0_add0000_cy<9>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<10> (Madd_v0_add0000_cy<10>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<11> (Madd_v0_add0000_cy<11>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<12> (Madd_v0_add0000_cy<12>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<13> (Madd_v0_add0000_cy<13>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<14> (Madd_v0_add0000_cy<14>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<15> (Madd_v0_add0000_cy<15>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<16> (Madd_v0_add0000_cy<16>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<17> (Madd_v0_add0000_cy<17>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<18> (Madd_v0_add0000_cy<18>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<19> (Madd_v0_add0000_cy<19>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<20> (Madd_v0_add0000_cy<20>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<21> (Madd_v0_add0000_cy<21>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<22> (Madd_v0_add0000_cy<22>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<23> (Madd_v0_add0000_cy<23>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<24> (Madd_v0_add0000_cy<24>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<25> (Madd_v0_add0000_cy<25>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<26> (Madd_v0_add0000_cy<26>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<27> (Madd_v0_add0000_cy<27>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<28> (Madd_v0_add0000_cy<28>)
    MUXCY:CI->O            1  0.044   0.000  Madd_v0_add0000_cy<29> (Madd_v0_add0000_cy<29>)
    MUXCY:CI->O            0  0.044   0.000  Madd_v0_add0000_cy<30> (Madd_v0_add0000_cy<30>)
    XORCY:CI->O            1  0.360   0.523  Madd_v0_add0000_xor<31> (v0_add0000<31>)
    LUT2:I1->O             1  0.195   0.000  Madd_sOut_TID12_lut<29> (Madd_sOut_TID12_lut<29>)
    MUXCY:S->O             1  0.366   0.000  Madd_sOut_TID12_cy<29> (Madd_sOut_TID12_cy<29>)
    MUXCY:CI->O            0  0.044   0.000  Madd_sOut_TID12_cy<30> (Madd_sOut_TID12_cy<30>)
    XORCY:CI->O            1  0.360   0.360  Madd_sOut_TID12_xor<31> (pOut1_31_OBUF)
    OBUF:I->O                 3.957          pOut1_31_OBUF (pOut1<31>)
    ---------------------------------------
    Total                  9.611ns (8.143ns logic, 1.468ns route)
                               (84.7% logic, 15.3% route)
```

Figure 4-11: Synthesis timing report of two consecutive additions with a right shift operation in between

4.3.1 Experimental Results

Table 4-1 shows the average percentage error of the proposed critical path delay estimation technique with respect to the synthesis results of equivalent hand-crafted designs for 150 custom instructions from sixteen benchmark applications. Only custom instructions that do not contain complex operations (e.g. multiplication, division, shift-by non-constant) are considered. These complex operations can be implemented using CGAUs. The hand-crafted designs are realized with VHDL and synthesized with Xilinx ISE Version 11.2. The target FPGA device is Xilinx Virtex-4 xc4vlx40-10ff1148, which incorporates BLEs with 4-input LUTs. Additional details of the custom instructions (e.g. size, number of operands, actual delay, and estimated delay) are shown in Appendix C.

The proposed high level estimation technique performs cluster enumeration and selection for each custom instruction using the estimation models shown in (4-3), (4-4) and (4-6). It can be observed that the proposed technique has an average percentage error of only 2.85% for the 150 custom instructions. In addition, except for Basicmath Large, the average percentage error for each application is within 4%. It can be seen in Appendix C that for the Basicmath Large application, the absolute error of critical path estimation is less than 1 ns.

These results are very encouraging as the critical delay is directly estimated from the high-level primitive operations without time-consuming design entry and synthesis. In particular, the cluster generation process (cluster enumeration and selection) for each custom instruction can be achieved in milliseconds. This enables the proposed high level estimation technique to be use for rapid design exploration during instruction set customization.

Table 4-1: Average error of critical path delay estimation

Application	Number of Custom Instructions	Average Error (%)
Adpcm Dec	6	2.89
Adpcm Enc	7	1.45
Aes	27	3.30
Basicmath Large	2	9.63
Bitcount	4	3.13
Blowfish Dec	8	2.20
Blowfish Enc	8	2.20
Cjpeg	29	1.71
CRC32	1	1.00
Dijkstra Large	5	1.45
FFT	5	3.03
Patricia	4	1.27
Pegwit	14	2.29
Rijndael Dec	12	3.56
Rijndael Enc	11	3.44
Sha	17	3.04

Next, we reexamine the performance of the custom instruction selection approaches discussed in the previous chapter by using the proposed cluster generation technique to estimate the custom instruction latencies on FPGAs. The performance of an application with custom instruction extension for application *A* can be calculated in terms of *SCS* (Software Cycle Savings) as shown in (4-7). *SCS(A)* is defined as the number of software clock cycle savings due to the migration of the native instructions of the processor to hardware for application *A*. In (4-7), G_i for $i = 1,2,\ldots,n$ is a custom instruction, where *n* is the total number of custom instructions obtained from template selection for application *A*, $F(G_i)$ is the execution frequency of instruction G_i in application *A*, $TS(G_i)$ denotes the number of nodes covered in instruction G_i using the template selection methods discussed in the previous chapter, and r is the ratio of the clock frequency of the RFU and the base processor. In this chapter, we assume $r = 1$. For simplicity, $T_{Template}(G_i)$, which is the estimated critical path delay of G_i is assumed to be m, which is the number of basic blocks[11] in the critical path of G_i.

$$SCS(A) = \sum_{i}^{n} F(G_i) \cdot \left(TS(G_i) - r \cdot T_{Template}(G_i)\right) \qquad (4\text{-}7)$$

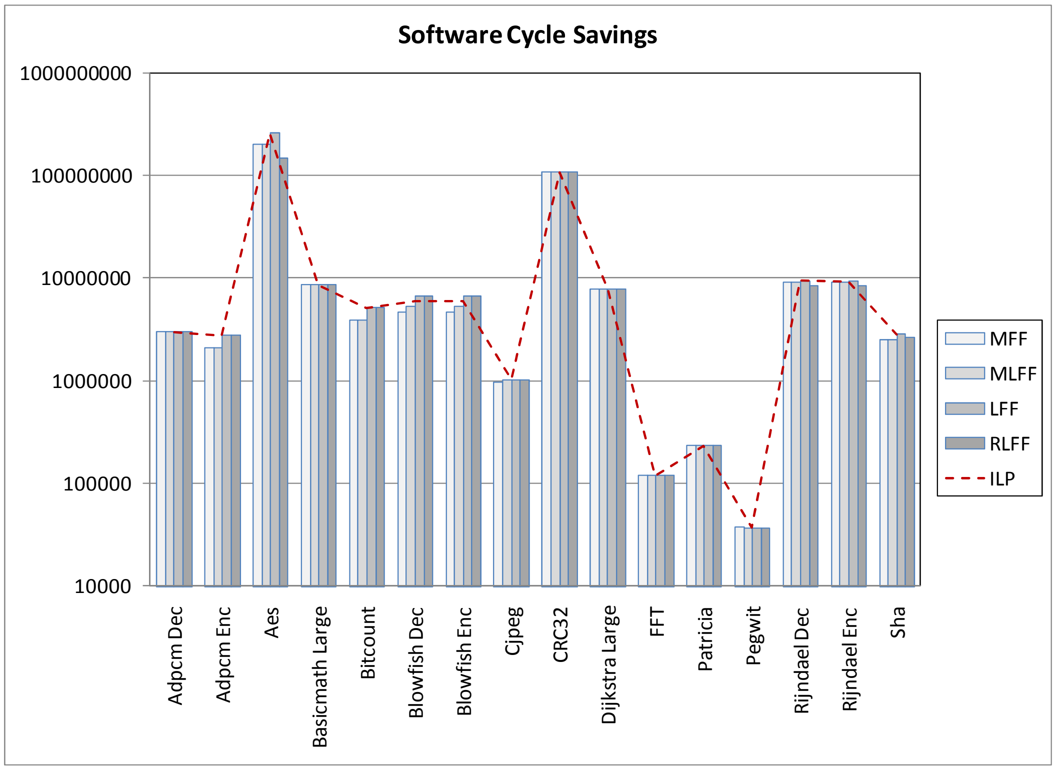


Figure 4-12: Performance comparison after high level estimation

Figure 4-12 compares the software cycle savings of the various template selection strategies, i.e. MFF, MLFF, ILP and proposed methods (i.e. LFF and RLFF). Similar to the previous results (see Figure 3-9), the ILP approach leads to higher performance in

[11] A basic block is a sequence of consecutive statements in a program, in which the flow of control has only one entry point and one exit point. See Appendix B for a graphical example of a basic block in a CFG.

almost all cases (and comparable in the remaining cases) when compared to MFF and MLFF. However, unlike the previous results, the proposed RLFF method performs less favorably than the ILP method in several applications (e.g. AES, Rijndael Dec, Rijndael Enc, Sha). In addition, MFF and MLFF outperform the proposed RLFF method in some of these applications. It can be observed from Figure 4-12 that when compared to the ILP method, the proposed LFF approach performs more favorably in several applications (e.g. Blowfish Dec, Blowfish Enc), and comparable to the remaining ones. In addition, the LFF method outperforms the RLFF method in more than one application (e.g. AES, Rijndael Dec, Rijndael Enc, Sha), with comparable performance in the remaining ones.

Table 4-2: Percentage performance gain of LFF over MFF, MLFF, ILP and RLFF

Application	MFF	MLFF	ILP	RLFF
Adpcm Dec	0.02	0.02	0.02	0.00
Adpcm Enc	33.16	33.16	0.00	0.00
Aes	30.27	30.27	0.54	79.39
Basicmath Large	0.00	0.00	0.00	0.00
Bitcount	33.33	33.33	0.00	0.00
Blowfish Dec	40.70	23.89	10.67	0.00
Blowfish Enc	40.70	23.89	10.67	0.00
Cjpeg	6.91	1.64	0.00	0.00
CRC32	0.00	0.00	0.00	0.00
Dijkstra Large	0.00	0.00	0.00	0.00
FFT	0.00	0.00	0.00	0.00
Patricia	0.00	0.00	0.00	0.00
Pegwit	-2.58	0.00	-2.75	0.00
Rijndael Dec	3.44	3.44	0.00	13.18
Rijndael Enc	3.45	3.45	0.00	13.21
Sha	15.84	15.84	0.17	7.54

Table 4-2 shows the percentage performance gain (in terms of software cycle savings) of the proposed LFF method over other template selection approaches (i.e. MFF, MLFF, ILP and RLFF). It can be observed that the proposed LFF method outperforms the RLFF method by close to 80% for the AES application, and over 10% for Rijndael Dec and Rijndael Enc. On average, the proposed LFF method outperforms the MFF, MLFF, ILP and RLFF method by 12.8%, 10.6%, 1.2% and 7.1% respectively. These results demonstrate the importance of an accurate high-level estimation technique for custom instruction selection.

The results also demonstrate the attractiveness of the proposed LFF method for selecting custom instructions with high performance from applications in the security domain (i.e. Aes and Rijndael). As explained in [113], certain complex operations in modern block ciphers are usually implemented as large groups of logical and logical shift-by-constant operations. As the proposed LFF has a preference for selecting large templates, it is capable of selecting these large groups of operations which can be mapped efficiently onto the FPGA logic blocks.

4.4 AREA ESTIMATION

Estimating the area utilization of custom instructions on FPGAs in terms of logic blocks using the cluster generation process is a straightforward process. The estimated number of logic blocks is equivalent to the number of selected basic clusters that are obtained after the cluster selection step. This estimation is undertaken with the assumption that the eventual hardware operators must cater to operands with the maximum bit-width. However, this may lead to high inaccuracies as commercial FPGA tools are capable of inferring the appropriate data-path widths of hardware operators, which processes operands that occupy a limited segment of the maximum bit-width.

In this section, we describe a simple method for estimating the area utilization of the custom instructions on FPGAs at a lower granularity (i.e. number of BLEs or LUTs) based on the cluster generation process. In particular, the proposed method is capable of inferring the appropriate data-path widths of basic clusters, by analyzing the logic shift offsets in the basic clusters. We consider two cases based on the shift-left-by-constant and shift-right-by-constant operations. Note that the effective shift offsets must be calculated when the basic cluster has multiple shift-by-constant operations. This can be computed using existing bit-width analysis methods (e.g. [112]).

In the first case, we consider shift-left-by-constant operations that occur before or after a logical or arithmetic operation in a basic cluster. Figure 4-13 shows two examples of this situation, where a shift left operation (by a constant factor of two) occurs after and before an *ADD* operation. It can be observed from Figure 4-13 that the delay path consist of only 30 BLEs (we assume the maximum bit-width is 32) in both examples. In the first example, since the two Most Significant Bits (MSBs) of the addition result will be shifted out, the FPGA synthesis tool will recognize the redundancy in computing the addition for the two operand MSBs. Hence, only the last 30 bits of the operands will be computed. In the second example, since the operands are shifted left by two, only 30 BLES will be required to compute the effective bit-width of the operands. Therefore, the number of required BLEs of a basic cluster is equivalent to the maximum bit-width (i.e. 32) minus the effective left shift offset of the basic cluster.

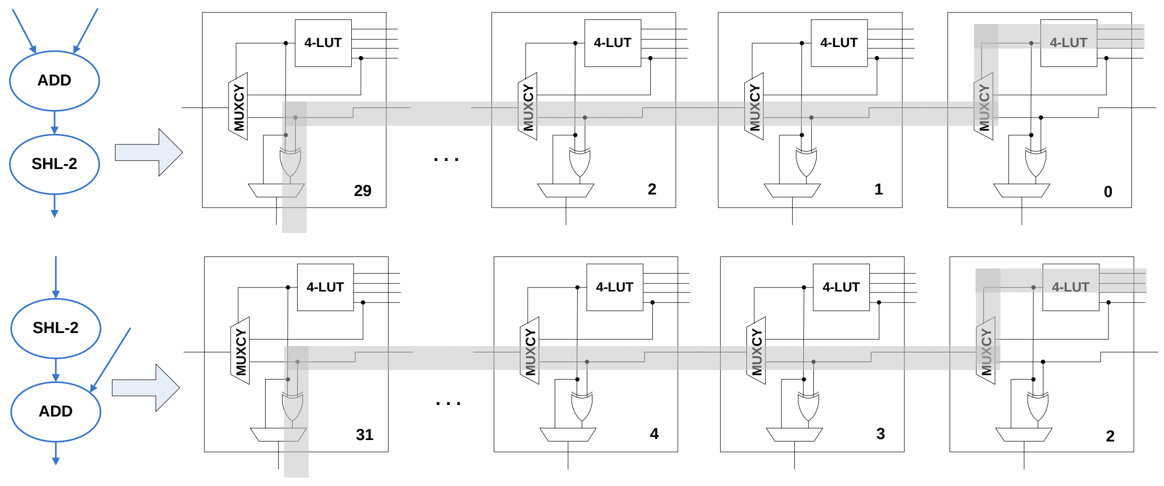


Figure 4-13: Delay path of basic cluster with shift-left-by-constant and addition operations

In the second case, we consider shift-right-by-constant operations that occur before a logical or arithmetic operation in a basic cluster. Figure 4-14 shows an example of this case where a shift right operation (by two) occurs before an *ADD* operation in a cluster. Similar to the discussion for the first case, since the operands are shifted right by two, only 30 BLES will be required to compute the effective bit-width of the operands. Therefore, the number of required BLEs of a basic cluster is equivalent to the maximum bit-width (i.e. 32) minus the effective right shift offset of the basic cluster, when the right shift operation occurs before the logical or arithmetic operations in the basic cluster.

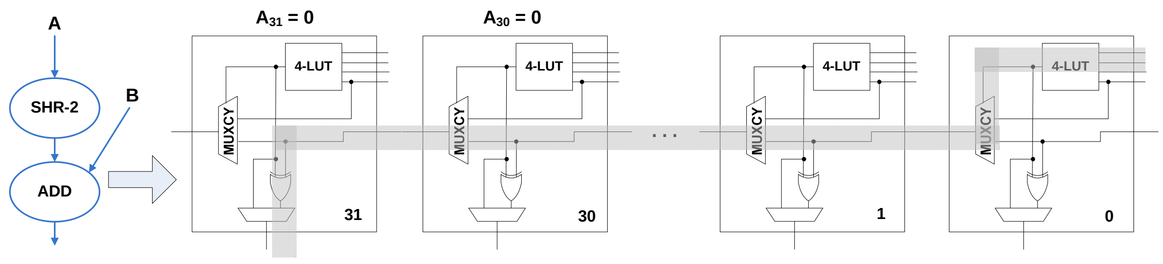


Figure 4-14: Delay path of basic cluster with shift-right-by-constant operation preceding an addition operation

4.4.1 Experimental Results

Table 4-3 shows the average percentage error of the proposed area estimation technique with respect to the implementation results of equivalent hand-crafted designs for the 150 custom instructions from sixteen benchmark applications. The actual area and estimated area of each custom instruction can be found in Appendix C. For each custom instruction, we compare the estimated number of BLEs with the number of 4-input LUTs reported in the implementation results of the FPGA tool.

It can be observed that the proposed technique has an average percentage error of only 0.40% for the 150 custom instructions. The maximum percentage error of each application is within 2%. In addition, it can be seen that the proposed method can estimate the number of BLEs without any error for several applications (e.g. Adpcm Dec, Adpcm Enc, Basicmath Large, Bitcount, CRC2 and FFT), and within 1% error in most of the other applications. These results demonstrate the reliability of the cluster generation process for high-level area estimation of custom instructions on FPGA.

Table 4-3: Average error of area estimation

Application	Number of Custom Instructions	Average Error (%)
Adpcm Dec	6	0.00
Adpcm Enc	7	0.00
Aes	27	0.54
Basicmath Large	2	0.00
Bitcount	4	0.00
Blowfish Dec	8	0.27
Blowfish Enc	8	0.27
Cjpeg	29	0.44
CRC32	1	0.00
Dijkstra Large	5	1.65
FFT	5	0.00
Patricia	4	0.81
Pegwit	14	0.79
Rijndael Dec	12	0.18
Rijndael Enc	11	0.19
Sha	17	1.23

4.5 SUMMARY

In the chapter, we have proposed a novel cluster generation strategy that partitions the custom instructions into a set of basic clusters such that the basic clusters can be efficiently mapped onto the LUT and carry-look-ahead structure of the FPGA logic blocks. We presented delay estimation models, which take into account the anomalies incurred by consecutive addition operations and shift right operations to accurately estimate the critical path of the basic clusters. Experimental results show that the average estimated critical paths of 150 custom instructions from sixteen applications using the proposed method are within 3% of those obtained using hardware synthesis. It is noteworthy that the runtime of the proposed estimation process is negligible when compared to the time taken for hardware synthesis.

We then reexamine the performance of the custom instruction selection approaches discussed in the previous chapter by using the proposed cluster generation technique to estimate the custom instruction latencies on FPGAs. Contrary to the results in the previous chapter, the new experimental results reveal that the proposed LFF based template selection approach performs more favorably than other template selection methods. This is due to the fact that the templates selected using the LFF approach incur low critical path delays when mapped onto the FPGA logic blocks. This clearly demonstrates the necessity and effectiveness of the proposed delay estimation technique for rapid design exploration of custom instructions.

Finally, we proposed strategies that take into consideration the logic shift offsets in basic clusters to accurately estimate the area utilization (in terms of number of BLEs) of the custom instructions on FPGA. Our results show that the average estimated area utilization of the 150 custom instructions using the proposed method are within 1% of those obtained from FPGA implementation results.

So far, we have presented template selection strategies for selecting high performance custom instructions. In the next chapter, we will present a strategy for selecting area-time efficient custom instructions. The proposed strategy relies on the cluster generation technique for rapid design exploration to select a reduced set of custom instructions without compromising the performance gain.

CHAPTER 5

SELECTING MOST PROFITABLE CUSTOM INSTRUCTIONS

Although there are a number of reported works in the literature on instruction set customization, they do not incorporate efficient techniques for realizing area-time efficient custom instructions on FPGAs. In the previous chapter, we have shown that the proposed LFF based template selection approach outperforms other approaches as it is able to select the custom instructions that lead to the highest performance (in terms of software cycle savings). However, as discussed in Chapter 3, the LFF based approach also results in the selection of large custom instructions, which may not lead to area-efficient implementation.

In [66], a covering algorithm was presented to select a minimal set of templates that maximizes the number of covered nodes. The authors analyzed the tradeoff between the number of templates and the percentage of node coverage. It was observed that increasing the number of templates in the covering algorithm will lead to a notable increase in the number of covered nodes only up to a certain point. This implies that selecting larger number of templates may not necessarily lead to better performance gain. Although this is an interesting observation, the work in [66], however, has not studied the effect on the hardware area-time when varying number of templates is selected.

In this chapter, we focus on the selection of the most profitable custom instructions, which provide higher performance for a given reconfigurable area. Choosing the most profitable custom instructions is essential if FPGA area constraints must be met. We propose a design exploration framework to rapidly identify a reduced set of profitable custom instructions without the need for actual hardware synthesis. Unlike the work in [66], the proposed framework leverages a high-level estimation technique (based on the cluster generation technique discussed in the previous chapter) for rapid selection of a reduced set of custom instructions that leads to high area efficiency without compromising heavily the performance gain.

Simulations based on sixteen applications from benchmark suites show that the proposed framework provides, on average, an area reduction of over 25% with negligible loss in compute performance. In addition, an average area-delay product gain of over 86% was achieved by deploying a reduced set of custom instructions obtained using the proposed framework. Our evaluations also confirm that the proposed strategy is superior to an existing area-optimization approach that relies on exploiting the regularity of custom instruction data paths.

In the following section, we first present an overview of the proposed design exploration framework. The essential steps in the proposed framework are then discussed. We then present experimental results to demonstrate the benefits of the proposed strategy for selecting area-time efficient custom instructions. The work discussed in this chapter has been published in [J-3], [J-5] and [C-5].

5.1 OVERVIEW OF PROPOSED DESIGN EXPLORATION FRAMEWORK

Figure 5-1 shows an overview of the proposed design exploration framework. We have relied upon the Trimaran compiler infrastructure [103] to generate the IR of C applications in the form of DFGs. The Trimaran compiler also performs application profiling to compute the execution frequency of the application basic blocks.

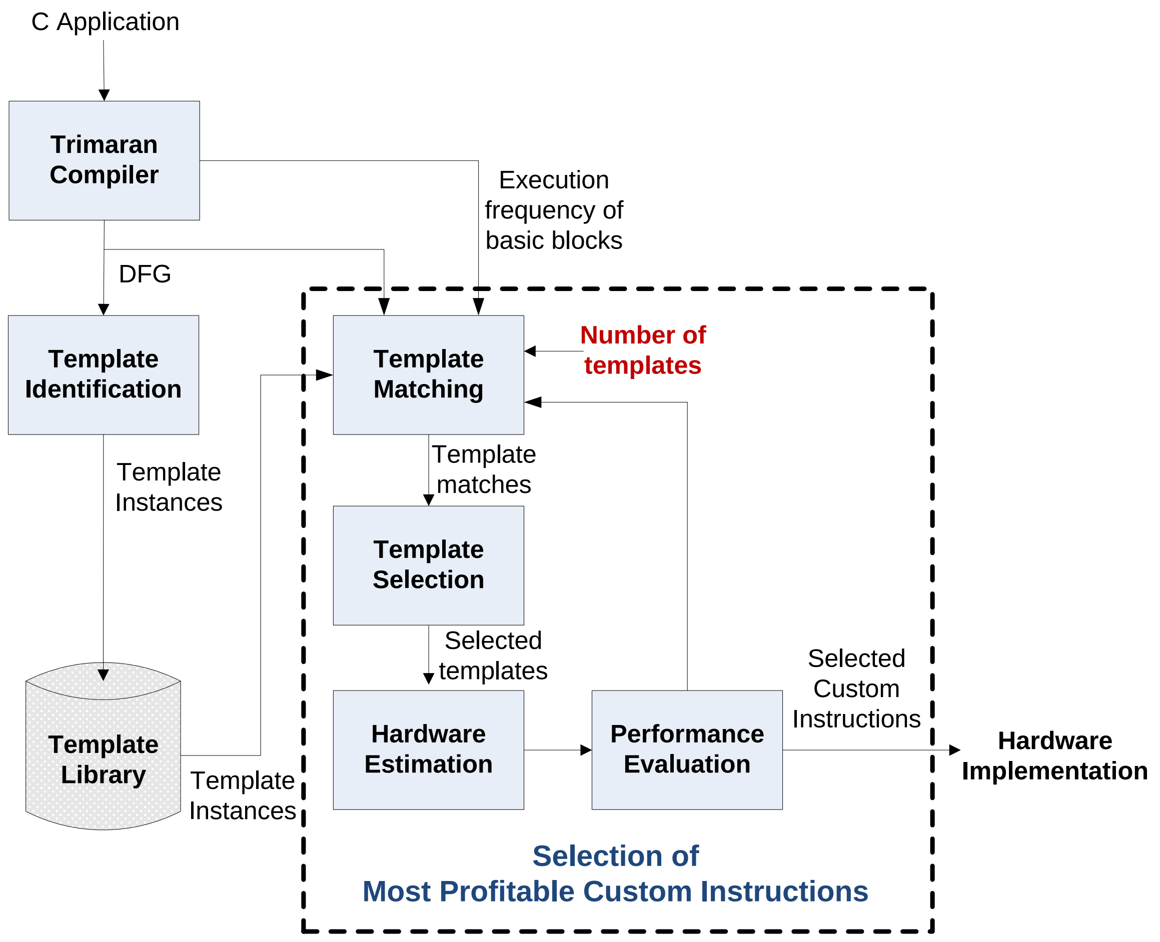


Figure 5-1: Proposed design exploration framework

In the template identification stage, a template enumeration method is combined with graph isomorphism to identify template instances from the Trimaran IR. These template instances form a set of potential custom instruction candidates to be mapped on the RFU and must satisfy a set of architectural constraints. Details of the template identification stage have been discussed in Section 3.3.1 and will not be elaborated here. Note that template identification and application profiling are performed only once for each application. The templates corresponding to the unique template instances are stored in the template library. The selection of the most profitable custom instructions stage iteratively selects custom instructions from the templates based on a strategy that aims to maximize the area-time efficiency of the custom instructions.

5.2 SELECTION OF MOST PROFITABLE CUSTOM INSTRUCTIONS

The three main steps in the selection of the most profitable custom instruction stage are: 1) template matching, 2) template selection, and 3) hardware estimation.

5.2.1 Template Matching

A heuristic is used to first identify a set of templates for template matching, which account for the performance gain and area utilization of the custom instruction in hardware. Each template T_i is assigned a gain as shown in (5-1), where the estimated speedup obtained by mapping T_i on hardware is calculated as shown in (5-2). $T_{SW}(T_i)$ denotes the estimated number of clock cycles taken for T_i to run on a processor. $T_{HW}(T_i)$ denotes the number of clock cycles of T_i in hardware, and we estimate this by the length of the critical path in the custom instruction sub-graph. *Size(T_i)* denotes the size of T_i and is estimated by the number of primitive operations of T_i.

$$Gain(T_i) = \frac{Speedup(T_i)}{Size(T_i)} \qquad (5\text{-}1)$$

$$Speedup(T_i) = \frac{T_{SW}(T_i)}{T_{HW}(T_i)} \qquad (5\text{-}2)$$

In our experiments, we assume that $T_{SW}(T_i) = Size(T_i)$, and hence $Gain(T_i) = \frac{1}{T_{HW}(T_i)}$. Figure 5-2 shows an example of a template instance and the corresponding T_{SW} and T_{HW} values.

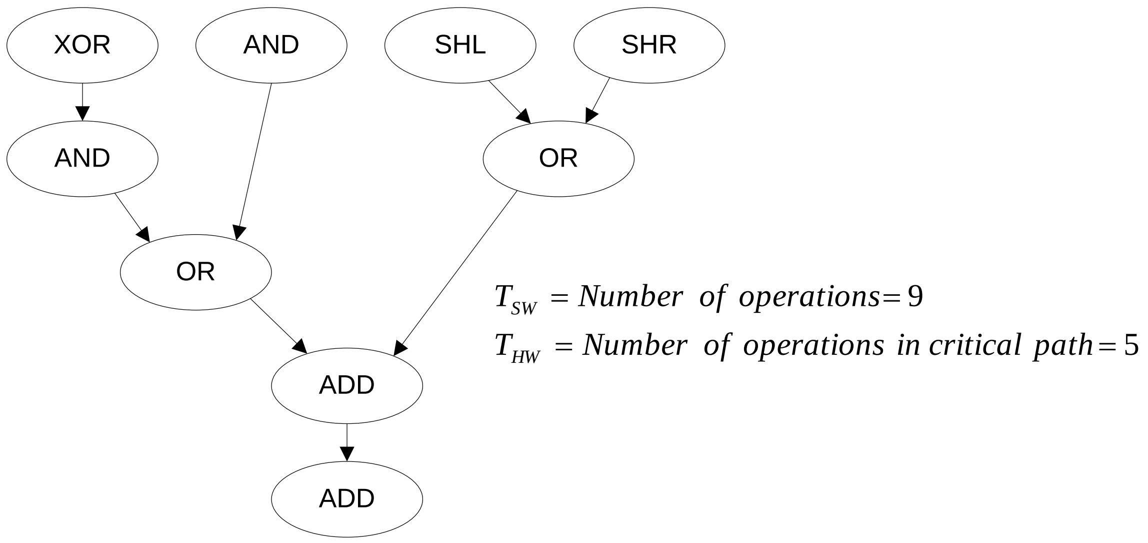

XOR
AND
SHL
SHR
AND
OR
OR
ADD
ADD
T_{SW} = Number of operations = 9
T_{HW} = Number of operations in critical path = 5

Figure 5-2: Example of template instance

The templates are then sorted in decreasing gain, and a varying range of templates (each range includes templates with highest gain) is iteratively selected for template matching. The template matching problem can be described as follows: Given an application DFG that is represented as a directed labeled graph *G(V, E)* and the templates, where each template is a directed graph $T_i(V_i, E_i)$, find every sub-graph of G that is isomorphic to T_i. This problem is essentially equivalent to the sub-graph isomorphism problem. We have used the vflib graph-matching library [114] to identify all the matches in the DFG for the templates.

5.2.2 Template Selection

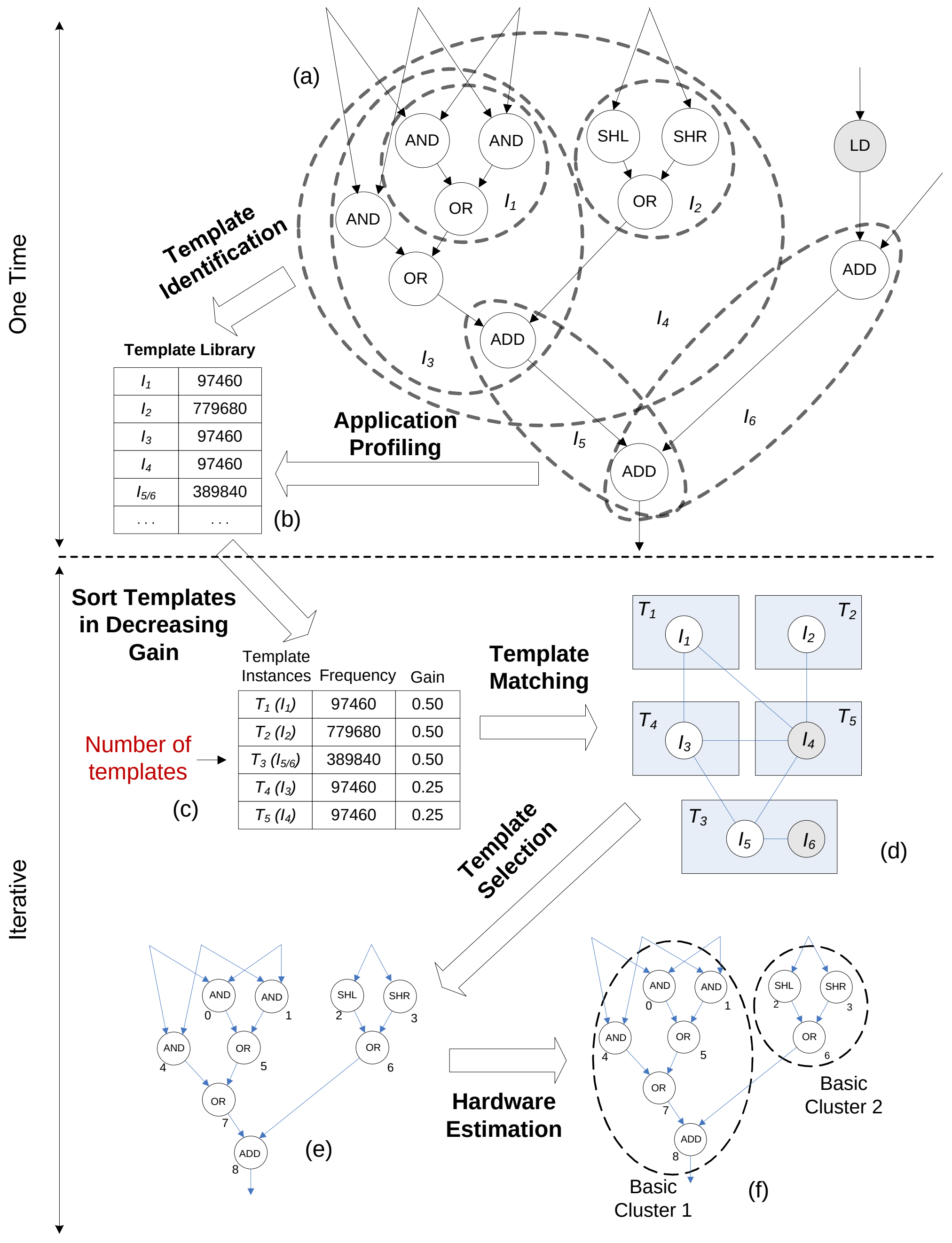
We have adopted the LFF approach for template selection as it has been shown to lead to the selection of custom instructions with the highest performance. Details of the LFF template selection approach have already been discussed in Chapter 3.

5.2.3 Hardware Estimation

This step estimates the critical path delays of the selected custom instructions when they are realized with the BLEs of the RFU. The estimation relies on the cluster generation technique discussed in the previous chapter.

5.2.4 Reduced Template Selection Process

In this section, we describe how the proposed design exploration framework in Figure 5-3 can be used for selecting a reduced set of custom instructions with the help of an example. Figure 5-3(a) shows a DFG and the corresponding template instances (I_1, I_2, . . . , I_6) that have been identified in the template identification stage. Note that only integer operations are considered in template instances. Other architectural constraints that are imposed on the template instances are discussed in Section 3.3.1. For simplicity, we have only shown six enumerated template instances in this example.



I_1	97460
I_2	779680
I_3	97460
I_4	97460
$I_{5/6}$	389840
. . .	. . .

Template Instances	Frequency	Gain
T_1 (I_1)	97460	0.50
T_2 (I_2)	779680	0.50
T_3 ($I_{5/6}$)	389840	0.50
T_4 (I_3)	97460	0.25
T_5 (I_4)	97460	0.25

Figure 5-3: Example of selecting a reduced set of custom instructions

The templates corresponding to the unique template instances are stored in the template library (Figure 5-3(b)), and their frequency of occurrence based on an input data set is obtained from application profiling. As shown in Figure 5-3, template identification and application profiling are computed only once for each application. The templates in the template library are then sorted in decreasing gain as described in Section 5.2.1, and a varying range of templates (each range includes the templates with highest gain) is iteratively selected for template matching and template selection. In the example in Figure 5-3(c), five templates (T_1, T_2, . . . , T_5) with instances (I_1, I_2, . . . , I_6) are selected for template matching. Figure 5-3(d) shows an example of a conflict graph, where the template instances I_4 and I_6 are selected as custom instructions. Figure 5-3(e) shows the selected custom instruction T_5 that corresponds to the template instance I_4. The selected custom instructions then undergo hardware estimation using the clustering technique as shown in Figure 5-3(f).

In the first iteration, the full range of templates is used in template matching and selection, and the performance of the selection is evaluated using the estimated hardware measures (based on the cluster generation process discussed in the previous chapter). The range of templates used in template matching and selection is progressively reduced in subsequent iterations. This is repeated until the estimated performance of the selected custom instructions is notably less than the previous iteration. When this occurs, the selected custom instructions in the previous iteration will be implemented onto the RFU using commercially available FPGA implementation tools (e.g. Xilinx FPGA tool).

5.3 EXPERIMENTAL RESULTS

In this section, we provide experimental results for sixteen applications from MiBench, MediaBench and EEMBC benchmark suites to demonstrate the area-time benefits of the proposed approach.

We compare the proposed technique with the area optimization approach, presented in [J-6], which is based on the concept of data-path merging. The work in [J-6] finds a maximal unique set of data-path structures that can cover all the selected custom instructions. This is achieved by merging the larger custom instruction sub-graphs with smaller sub-graphs that are subsumed by it. Figure 5-4 shows an example of using a maximal data-path structure (Figure 5-4(a)) to implement several custom instructions (Figure 5-4(b)) in a cost-efficient manner. It can be observed that each of the custom instruction in Figure 5-4(b) is sub-graph isomorphic to the maximal data-path structure in Figure 5-4(a). These custom instructions can be efficiently mapped onto the maximal data-path structure by taking advantage of the fact that most operations (i.e., *ADD, SUB*, *AND*) have an associated input that allows values to pass through the operators without changing. This is achieved by setting one of the inputs to 0 or 1. For operators that do not possess this property (i.e. *SHL* in the example which is a shift-by-constant operation), a multiplexer is employed to allow the intermediate values to bypass the operator when the corresponding operation is not required. Similar concepts have been employed in [68] to maximize the area utilization of custom instructions.

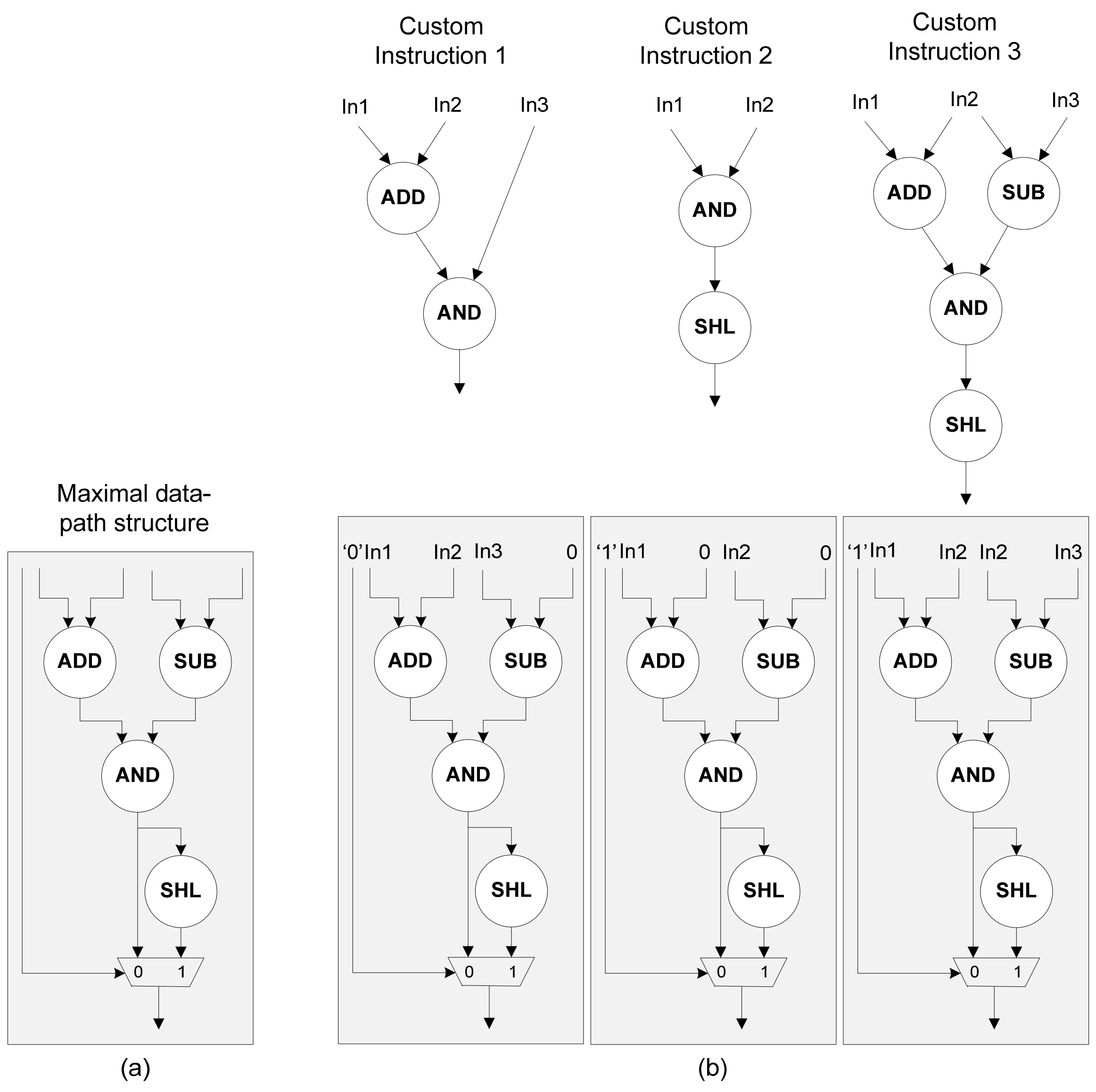


Figure 5-4: Implementing custom instructions using maximal data-path structure

In our experiments, we compare the area-time results of the proposed technique, which selects a reduced set of custom instructions (denoted as *Reduced Templates without Data-path Merging* (*RT*)), with the following approaches:

1. *All Templates without Data-Path Merging* (*AT*): Custom instruction selection based on the full range of templates without data-path merging.

2. *All Templates with Data-Path Merging* (*AMT*): Custom instructions selected from the full range of templates are subjected to the area optimization strategy in [J-6], which merges custom instructions in order to maximize resource sharing.
3. *Reduced Templates with Data-Path Merging* (*RMT*): Selected custom instructions using the proposed RT approach are further subjected to the area optimization method in [J-6].

The performance of the selected custom instructions is reported in terms of software cycle savings as discussed in Section 4.3.1, and the target RISP is based on the model described in Section 4.1. The custom instructions obtained with the approaches RT, AT, AMT, and RMT have been designed in VHDL and implemented using Xilinx ISE Version 11.2. The target FPGA device is Xilinx Virtex-4 xc4vlx40-10ff1148, which incorporates BLEs with 4-input LUTs. The area-time product is obtained by multiplying the software cycle savings by the inverse of the area (in terms of number of slices).

5.3.1 Area-Time Comparison of AMT With AT

Tables 5-1 and 5-2 show the performance and area of custom instructions obtained using the AMT and AT approaches. For the AMT approach, area optimization is performed on the selected custom instructions after template matching and selection. It can be observed in column 6 of Table 5-1 that custom instructions that have undergone area optimization (i.e., AMT) can suffer from notable performance degradation in a majority of the applications. In particular, the AT approach has an average of 8.1 times speedup over the AMT approach. The reason for this is that area optimization introduces multiplexers in the merged custom instruction data paths (see Figure 5-4), which leads to an increase in the critical path delay of the custom instructions (i.e. $T_{Template}(G_i)$ in equation (4-7)).

Table 5-1: Performance comparison between AMT, AT, RMT and RT

Application	Performance (Software Cycle Savings)				Speedup of AT over AMT (x)	Gain of RT over RMT (%)	Gain of RT over AT (%)
	AMT	AT	RMT	RT			
Adpcm Dec	1587982	2956846	2956161	2956161	2	0.00	-0.02
Adpcm Enc	1379816	2748680	1374340	2745942	2	99.80	-0.10
Aes	242699155	259784912	127601391	259722032	1	103.54	-0.02
Basicmath Large	8497152	8497152	8497152	8497152	1	0.00	0.00
Bitcount	5100000	5100000	5100000	5100000	1	0.00	0.00
Blowfish Dec	4190206	6553954	4190206	6553954	2	56.41	0.00
Blowfish Enc	4190206	6553954	4190206	6553954	2	56.41	0.00
Cjpeg	571090	1022681	918339	951029	2	3.56	-7.01
CRC32	106444800	106444800	106444800	106444800	1	0.00	0.00
Dijkstra Large	75721	7649098	7572200	7572200	101	0.00	-1.01
FFT	118797	118797	118797	118797	1	0.00	0.00
Patricia	232869	232869	232869	232869	1	0.00	0.00
Pegwit	0	36185	11619	38233	-	229.06	5.66
Rijndael Dec	5009554	9375580	9375349	9375349	2	0.00	0.00
Rijndael Enc	4989733	9355551	9355455	9355455	2	0.00	0.00
Sha	2460865	2850705	2845832	2845832	1	0.00	-0.17

As shown in column 6 of Table 5-2, the area optimization approach AMT exhibits higher area efficiency than AT for a number of applications (maximum area reduction of over 30% can be achieved). However, the average area reduction achieved by AMT is only about 7.1%. Interestingly, in certain applications (e.g., Sha), the AT approach has slightly better area efficiency than the AMT approach. This can be explained by the fact that the area optimization strategy in [J-6] inherently introduces multiplexers in the data paths to facilitate data-path merging. These multiplexers will contribute to additional area requirements for realizing the custom instructions on FPGA. Hence, an effective area reduction can only be achieved if area savings due to resource sharing outweigh the cost that is introduced to facilitate data-path merging.

Table 5-2: Area comparison between AMT, AT, RMT and RT

Application	Area (Number of Slices)				Area Increase of AT over AMT (%)	Area Reduction of RT over RMT (%)	Area Reduction of RT over AT (%)
	AMT	AT	RMT	RT			
Adpcm Dec	117	156	136	136	33.33	0.00	12.82
Adpcm Enc	158	169	145	154	6.96	-6.21	8.88
Aes	4643	5939	4657	5965	27.91	-28.09	-0.44
Basicmath Large	16	16	16	16	0.00	0.00	0.00
Bitcount	160	160	96	96	0.00	0.00	40.00
Blowfish Dec	221	258	221	258	16.74	-16.74	0.00
Blowfish Enc	221	258	221	258	16.74	-16.74	0.00
Cjpeg	9394	9520	1396	1409	1.34	-0.93	85.20
CRC32	15	15	15	15	0.00	0.00	0.00
Dijkstra Large	1604	1652	351	351	2.99	0.00	78.75
FFT	180	180	148	148	0.00	0.00	17.78
Patricia	149	149	48	48	0.00	0.00	67.79
Pegwit	2905	3103	2967	3113	6.82	-4.92	-0.32
Rijndael Dec	303	306	233	246	0.99	-5.58	19.61
Rijndael Enc	1076	1116	507	519	3.72	-2.37	53.49
Sha	299	285	224	224	-4.68	0.00	21.40

Figure 5-5 compares the area-delay product between the various techniques. It can be observed that AT outperforms AMT in terms of area-delay product for most of the cases considered. In particular, AT achieves an average area-delay product gain of about 7.3 times that of AMT.

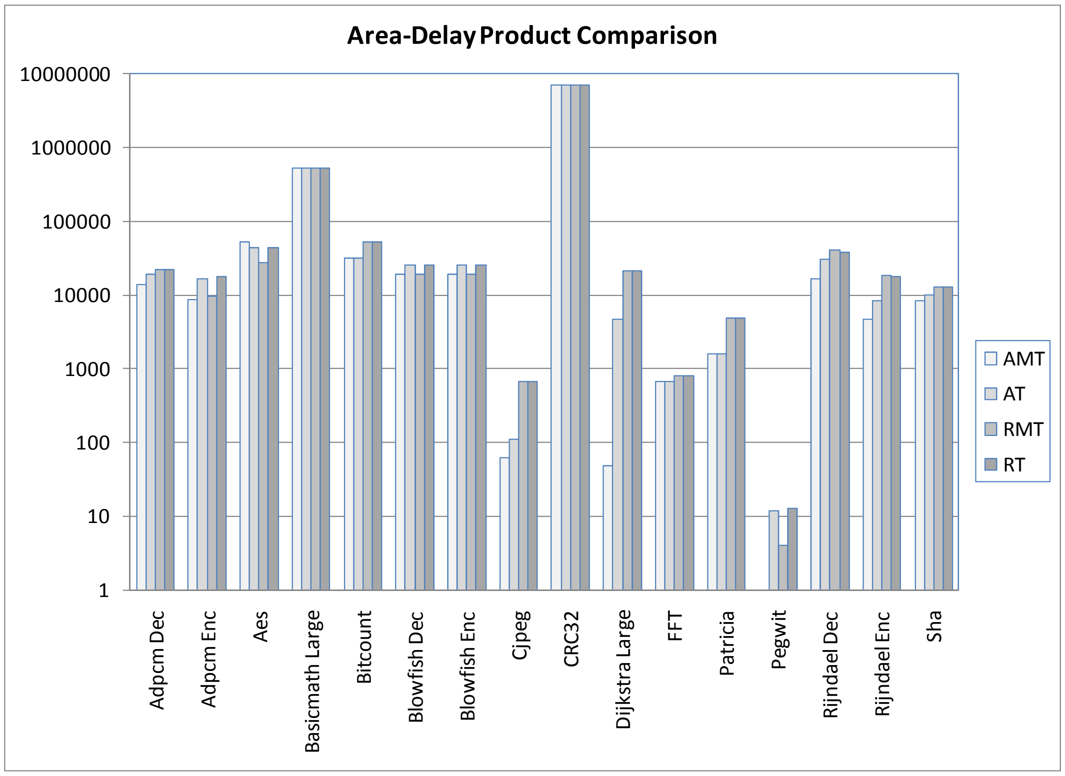


Figure 5-5: Area-delay product comparison between AMT, AT, RMT, and RT

The AT and AMT approach is based on the full range of templates (100%) used for template matching and selection. Figure 5-6 compares the average performance of custom instructions that have not undergone area optimization (indicated as Non-Merged templates or Non-MT) with the average performance of custom instructions that have undergone area optimization (indicated as Merged Templates or MT) for varying number of templates used (from 10% to 90%) during custom instruction selection.

It can be observed that the average performance of non-MT tends to increase (in a non-monotonic fashion) with increasing number of templates used for custom instruction selection. It is interesting to note that the opposite applies for MT. In particular, the average performance of MT decreases (in a non-monotonic fashion) when more templates are used for custom instruction selection. The increase in the critical path delay due to the introduction of multiplexers in MT becomes more prominent when more templates are used for custom instruction selection as there are more opportunities to merge the custom instruction data paths for area minimization. Hence, the clock cycle savings of MT decrease with the increase in the number of templates used for custom instruction selection.

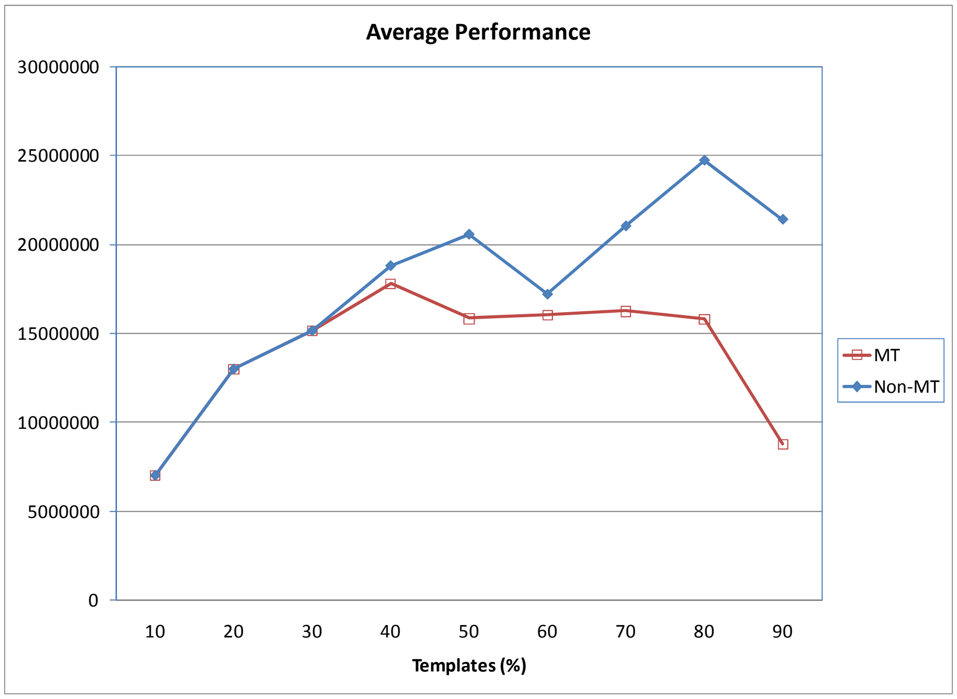
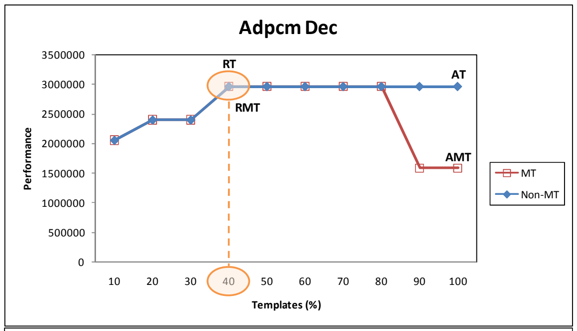
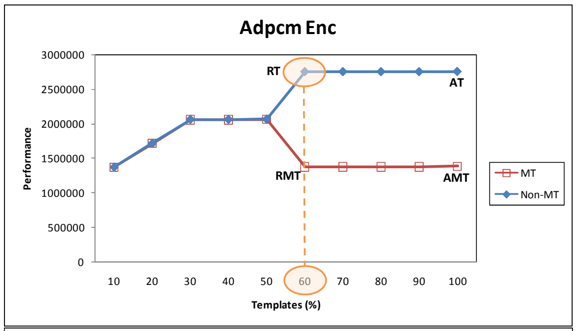


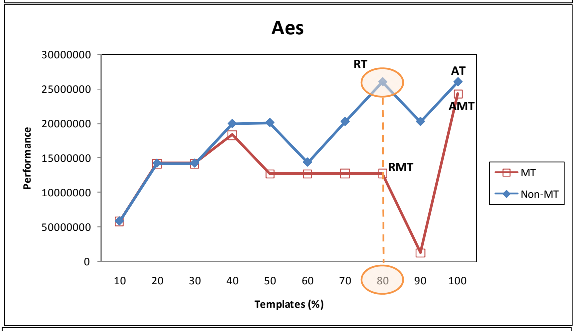
Figure 5-6: Average performance comparison with varying number of templates

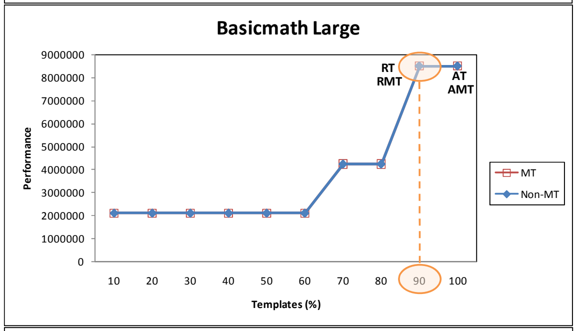
5.3.2 Selecting Reduced Set of Area-time efficient Custom Instructions

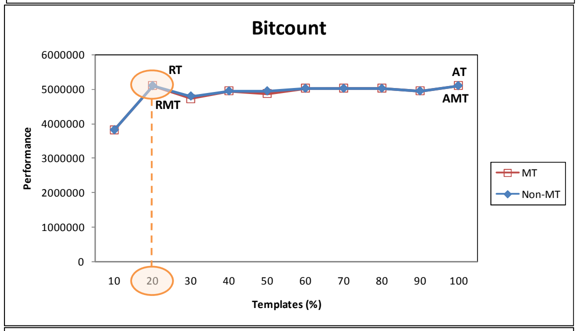
Figure 5-7 shows the estimated performance for the sixteen applications using the proposed method (RT).

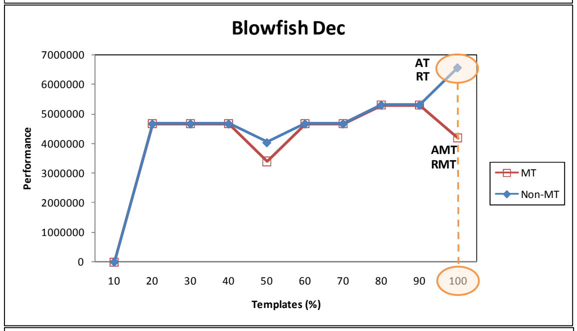

Adpcm Dec
Performance
Templates (%)
RT
RMT
AT
AMT
MT
Non-MT

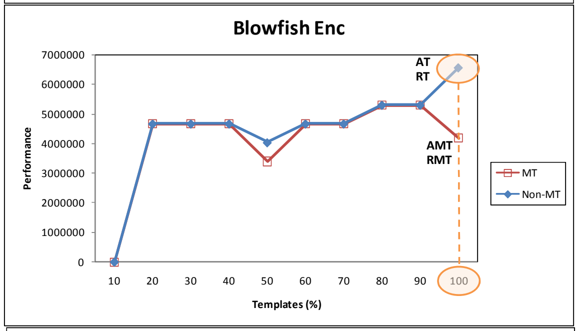

Adpcm Enc
Performance
Templates (%)
RT
RMT
AT
AMT
MT
Non-MT

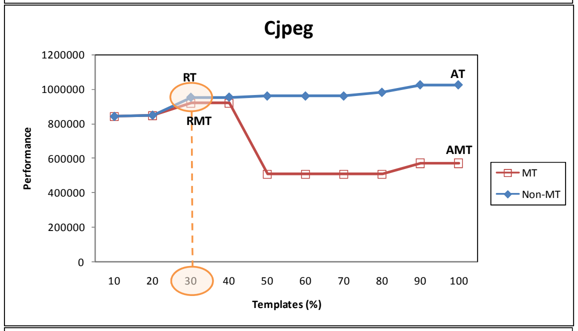

Aes
Performance
Templates (%)
RT
RMT
AT
AMT
MT
Non-MT

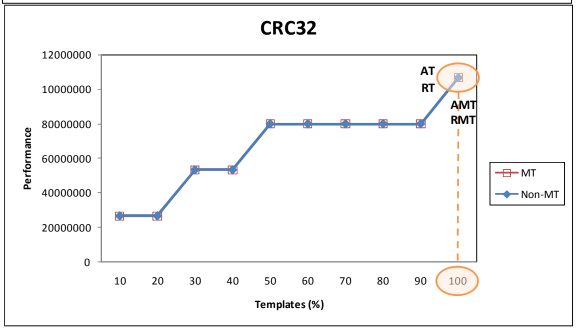

Basicmath Large
Performance
Templates (%)
RT
RMT
AT
AMT
MT
Non-MT

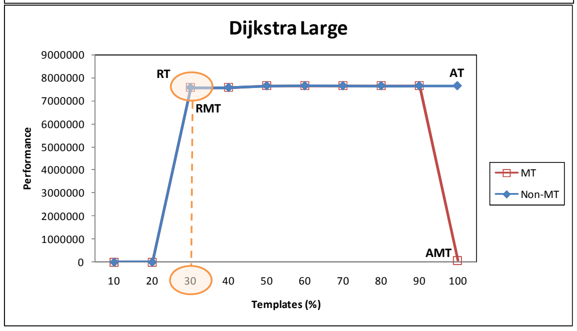

Bitcount
Performance
Templates (%)
RT
RMT
AT
AMT
MT
Non-MT


Blowfish Dec
Performance
Templates (%)
AT
RT
AMT
RMT
MT
Non-MT


Blowfish Enc
Performance
Templates (%)
AT
RT
AMT
RMT
MT
Non-MT


Cjpeg
Performance
Templates (%)
RT
RMT
AT
AMT
MT
Non-MT


CRC32
Performance
Templates (%)
AT
RT
AMT
RMT
MT
Non-MT


Dijkstra Large
Performance
Templates (%)
RT
RMT
AT
AMT
MT
Non-MT

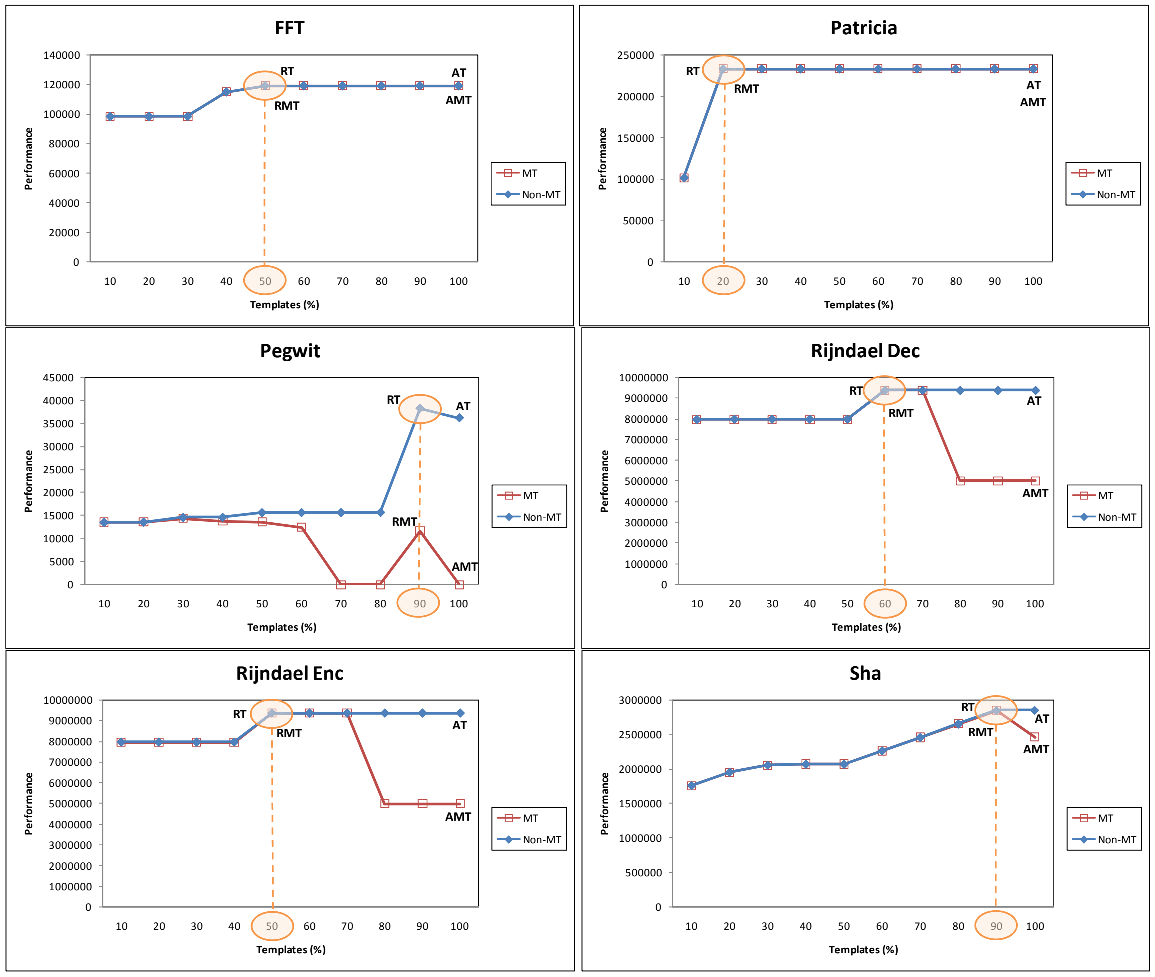


Figure 5-7: Performance comparison with varying number of templates

It is evident that increasing the number of templates for custom instruction selection will not lead to any notable gain after a certain point (marked as RT in the plots) for almost all the applications (except for Blowfish Dec, Blowfish Enc, CRC32 and Pegwit). These observations imply that it is possible to reduce the number of custom instructions for mapping onto the RFU without compromising heavily the performance gain.

5.3.3 Area-Time Comparison of RMT with RT

In the following experiments, we will investigate whether the proposed method can still lead to higher gains when compared to the case where the same reduced set of templates is subjected to area optimization. We denote the latter approach as RMT, where the selected custom instructions of RT are further subjected to area optimization using the method discussed in [J-6]. In both cases, the number of template instances used for custom instruction selection is shown in Figure 5-7 (i.e. 40% for Adpcm Dec, 60% for Adpcm Enc, 80% for AES, etc.). Table 5-1 shows the performance of the custom instructions obtained using RMT and the proposed RT approach. It can be observed in column 7 of Table 5-1 that even though both methods are based on the same set of custom instructions, the proposed method RT can still achieve significant performance gain over RMT for certain applications. In particular, the speedups that are achieved by RT over RMT are about 2, 2, 1.6 and 3.3 times for Adpcm Enc, Aes, Blowfish and Pegwit, respectively. The average performance gain of RT over RMT is over 34%. The reason for the performance degradation in RMT is due to the increase in critical path delay as a result of the area optimization technique which introduces multiplexers in the custom instruction data paths.

While it is expected that the RMT approach will lead to more area efficiency than RT, it is interesting to note from column 7 of Table 5-2 that the average area reduction of RMT is only about 5.1%. The reason for this is twofold. First, since only a reduced set of custom instructions is subjected to area optimization, there are fewer opportunities to merge custom instructions in order to reduce the area utilization. Hence, it can be observed that in most of the applications considered, the areas of RMT and RT are comparable. Second, the area-optimization strategy in [J-6] inherently introduces multiplexers in the data paths to facilitate data-path merging. These multiplexers will contribute to additional area requirements for realizing the custom instructions on FPGA.

Next, we compare the area-delay product between RMT and RT. It can be observed from Figure 5-5 that RMT has a higher area–time product than RT in only two applications (i.e. Rijndael Dec and Rijndael Enc). On the other hand, RT exhibits significant area-time gains over RMT in a number of applications (e.g., Adpcm Enc, Aes, Blowfish Dec, Blowfish Enc and Pegwit). In particular, the average area-delay product gain of RT over RMT is over 26.5%. This set of results demonstrates that the proposed method can still achieve significant area-time gains over the existing area-optimization approach for the same reduced set of custom instructions.

5.3.4 Area-Time Comparison of AT with RT

In the previous experiments, we have established that existing area optimization approaches that commonly rely on data-path merging methods may not lead to area-time-efficient realizations. In particular, we have shown that AT outperforms AMT in terms of area-time product by as much as 7.3 times. In addition, we have shown that the proposed method (RT) has an area-delay product gain over RMT, which employs the same reduced set of custom instructions, of over 26%. In this section, we will compare the area-time results of the proposed technique (RT) with AT.

Tables 5.1 and 5.2 show the performance and area of the custom instructions obtained using the AT and the proposed RT approach. The number of templates used in each application for RT is shown in Figure 5-7. It can be observed from column 8 of Table 5.1

that the average performance loss of RT is insignificant compared to AT. When compared to the case when all templates are used for custom instruction selection (i.e., AT), it is evident from column 8 of Table 5.2 that the proposed method (i.e. RT) can lead to significant area reduction (i.e. average of 25.3% area reduction). This is due to the fact that RT leads to the selection of significantly fewer custom instructions in most of the applications when compared to AT.

Finally, it can be observed from Figure 5-5 that when compared to the case where all template instances are used for custom instruction selection (AT), the proposed method (RT) has higher area-delay product gain for most of the applications (and comparable with the remaining ones). In particular, the average area-delay product gain of the proposed approach is about 1.9 times that of AT.

5.3.5 Performance Gain over Base Processor

Figure 5-8 shows the percentage performance gain of a custom processor with custom instructions obtained using the RT approach over the base processor implementation. We assumed that the base processor is a soft-core processor with the area-optimized configuration in [115]. The performance gain is obtained by computing the percentage execution time savings of the custom processor over the total execution time of the base processor (calculated based on the profiling results of Trimaran). We assumed that each operation in the base processor utilizes one clock cycle. The execution time savings of the custom processor are computed using equation (4-7), where $TS(G_i)$ and $T_{Template}(G_i)$ are substituted with the software latency of the selected custom instructions G_i and the hardware delay of G_i (obtained from FPGA tool), respectively. Figure 5-8 also shows the execution time of the custom processor with custom instructions obtained using the RT approach (secondary axis). The execution time is computed by subtracting the execution time savings of the custom processor from the total execution time of the base processor (calculated based on the profiling results of Trimaran).

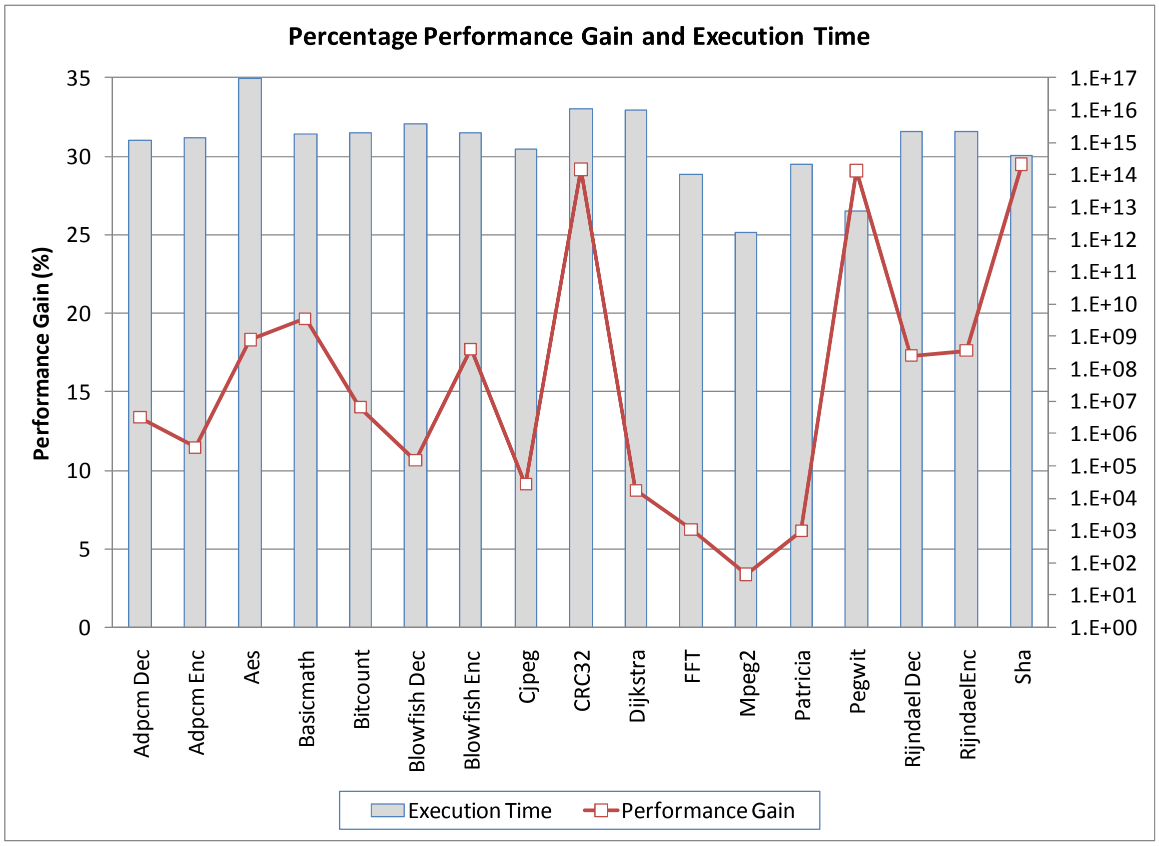


Figure 5-8: Performance gain over base processor

It can be observed from Figure 5-8 that the custom processor can achieve a notable average gain of over 15% and a maximum gain of about 30%. It is noteworthy that this performance gain is achieved in an area-efficient manner using the proposed strategy.

5.4 SUMMARY

A design exploration framework for RISPs has been proposed for the rapid selection of a reduced set of profitable custom instructions. The framework incorporates the cluster generation technique discussed in the previous chapter to facilitate rapid area-time estimation of custom instruction implementations on FPGA. Experimental results show that the number of candidates for custom instruction selection can be significantly reduced with insignificant degradation in resulting performance gain. Our investigation also reveals that the proposed method leads to higher area-time gains than is possible with existing area-optimization approaches that suffer from undesirable critical path delay in the resulting data paths due to resource sharing. Finally, the notable savings in area can be readily traded to increase performance or to reduce power consumption.

The proposed strategy in this chapter can effectively select custom instructions with high area-time efficiency by choosing a minimal set of custom instructions that does not lead to notable loss in performance when compared to the case when the maximal set of custom instructions are selected. This strategy does not take into consideration the FPGA architecture during the selection process. In the next chapter, we will present an architecture-aware strategy for mapping the custom instructions onto the FPGA architecture, which results in further area-time benefits by maximizing the utilization of the FPGA logic blocks.

CHAPTER 6

FPGA-AWARE MERGING OF CUSTOM INSTRUCTIONS FOR AREA-TIME EFFICIENCY

Area-time efficient custom instructions are desirable for maximizing the performance of RISPs. In the previous chapter, we have presented a strategy for selecting a reduced set of custom instructions that leads to high area efficiency without compromising heavily the performance gain. Existing data-path merging techniques based on resource sharing can be deployed to further improve area efficiency of custom instructions. We have briefly discussed the concept of data-path merging in the previous chapter. As shown in the experimental results in the previous chapter, techniques based on data-path merging can lead to a large increase in the critical path delay.

In this chapter, we propose a novel strategy that takes into account the architectural constraints of the FPGA device in order to realize custom instructions with low area-delay product. The proposed strategy leverages the cluster generation technique (discussed in Chapter 4) to partition the custom instruction data-paths into a set of basic clusters such that they can be combined using a heuristic based cluster merging process to maximize the utilization of FPGA logic blocks. Unlike the resource sharing method, the proposed cluster merging process does not maximize sharing of common resources and this leads to less reliance on multiplexers for implementing custom instructions. Resource sharing is only applied sparingly at the final stage to increase utilization of logic blocks.

We show that the proposed technique leads to more than 34%, 34% and 42% average reduction in area costs for Spartan-3, Virtex-4 and Virtex-5 architectures respectively when compared to optimizations achieved through commercial synthesis tools. We have also shown that the proposed technique leads to more than 18%, 17% and 13% average reduction in area costs for Spartan-3, Virtex-4 and Virtex-5 respectively when compared to results obtained using one of the most efficient resource sharing based method reported in the literature. In addition, the proposed technique outperforms the resource sharing based method in terms of area-delay product, with average reductions of more than 27%, 34% and 19% for Spartan-3, Virtex-4 and Virtex-5 respectively.

In the following section, we provide an example to highlight the limitations of existing high-level area optimization approaches based on resource sharing. Next, we give an overview of the proposed strategy and highlight the differences between our proposed method and existing approaches. This is followed by a detail description of the proposed technique. Finally, we present experimental results to demonstrate the superiority of the proposed technique over one of the most efficient resource sharing based method reported in the literature. The work in this chapter has been published in [J-1] and [C-6].

6.1 RESOURCE SHARING BASED AREA OPTIMIZATION

Figure 6-1 illustrates an example of resource sharing of two custom instruction data-paths (i.e. G_1 and G_2 in Figure 6-1(a)). In this example, we assume that there is only one available output port on the RFU and, hence, the outputs of G_1 and G_2 have been multiplexed. The two custom instruction data-paths in Figure 6-1(a) are combined into a single data-path in Figure 6-1(b) by merging similar operations and interconnections between the two data-paths. $x \oplus y$ denotes that operations/inputs x and y have been merged. The resulting data-path in Figure 6-1(b) is capable of performing the functionality of the original data-paths. It can be observed that the resulting number of operations and interconnections has been reduced. Figure 6-1 also reports the FPGA implementation results which show that the area (in terms of number of slices) has been reduced due to resource sharing.

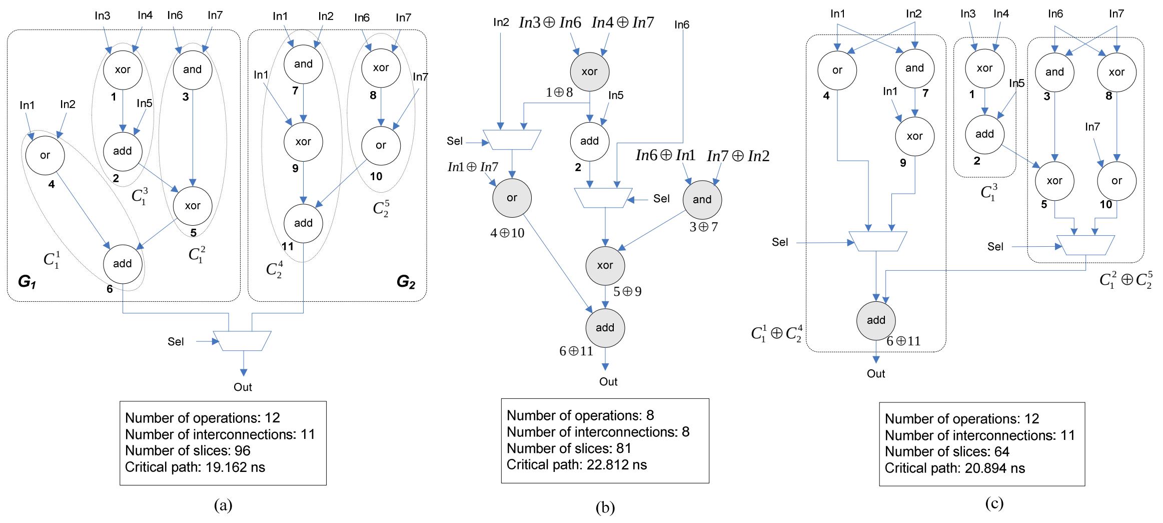


Figure 6-1: (a) Original, **(b)** resource sharing, and **(c)** cluster merging

Conventional area optimization algorithms based on resource sharing typically merge graph representations of two or more custom instructions that contain similar sub-graphs. Our results reveal that resource sharing based approaches (e.g. [51][87]-[92]) often do not lead to the most efficient FPGA resource utilization and can result in high critical path delay. This chapter presents a novel strategy for generating area-time efficient FPGA realization of custom instructions on commercial architectures. The proposed strategy first maps the custom instructions onto the logic blocks to maximize the area utilization of FPGA resources. This process resembles technology mapping, and it is treated as a covering problem rather than a merging problem. The mapped custom instructions are then merged to maximize the utilization of the logic blocks prior to judiciously considering resource sharing to avoid increasing the area-delay product. The proposed strategy can be completed in the order of milliseconds and hence does not incur an overhead in existing design flows.

6.2 OVERVIEW OF PROPOSED METHOD

The proposed method consists of the following three main steps: 1) cluster generation, 2) cluster merging, and 3) data-path combination and resource sharing. We describe the proposed method by using the example in Figure 6-1.

As discussed in Chapter 4, cluster generation partitions the custom instruction data-paths into a set of basic clusters. For example in Figure 6-1(a), *G₁* consists of basic clusters C_1^1, C_1^2 and C_1^3, and *G₂* consists of basic clusters C_2^4 and C_2^5. Cluster identification resembles the technology mapping process, where a set of basic clusters that effectively covers the custom instruction data-path is identified. It is worth mentioning that unlike existing works in technology mapping, cluster generation operates on the high-level representation of the custom instructions.

Definition 6-1: Let $C_x^j = (V_x^j, E_x^j)$ and $C_y^k = (V_y^k, E_y^k)$ be basic clusters (see Definition 4-2 in Chapter 4). $\overline{G} = (\overline{V}, \overline{E})$ is known as the *merged cluster* of C_x^j and C_y^k, denoted as $\overline{G} = C_x^j \oplus C_y^k$, if and only if:

- $x \neq y$, i.e. only the basic clusters in different data-paths can be merged.

- The merged cluster $\overline{G}$ can be implemented on a single FPGA logic block.
- $\overline{V} = V_x^j \cup V_y^k \cup V^o$, where V^o is a set of extra vertices and $0 \le \left|V^o\right| \le 1$. The extra vertex consists of a *m*-1 multiplexer (*MUX*) to facilitate time-multiplexed computations of the basic clusters C_x^j and C_y^k. For example in Figure 6-1(c), the basic clusters C_1^1 and C_2^4, and the basic clusters C_1^2 and C_2^5 are merged to produce the merged clusters $C_1^1 \oplus C_2^4$ and $C_1^2 \oplus C_2^5$ respectively. The resulting merged clusters require an additional component, which is a 2-1 multiplexer (*m* = 2).
- $\overline{E} = E_x^j \cup E_y^k \cup E^o$, where E^o is a set of extra arcs and $0 \le \left|E^o\right| \le 1$. The extra arcs are introduced along with the *MUX*. For example in Figure 6-1(c), the introduction of a 2-1 *MUX* in the merged cluster $C_1^1 \oplus C_2^4$ has resulted in an additional arc from the output port of the *MUX* to the input port of *ADD*. Note that the output ports of vertex $u \in V_x^j$ and $v \in V_y^k$ are assigned to the input ports of the *MUX*. In addition, a select signal *Sel* is required for the *MUX*. The same can be observed for $C_1^2 \oplus C_2^5$.

Cluster merging first identifies all combinations for merging the basic clusters in order to further increase the utilization of the FPGA resources by maximizing the functionality of the logic blocks. Basic clusters can be merged only if they belong to different data-paths and the resulting merged cluster can be implemented onto a single FPGA logic block. Note that the generation of a merged cluster may introduce an additional input for the multiplexer select pin as shown in Figure 6-1(c). This poses a limitation on the combination of basic clusters that can be merged, as the number of inputs of the resulting merged cluster cannot violate the input constraint of the logic block. In cases where a multiplexer is introduced in a merged cluster, the multiplexer select signal is used to select the desired functionality between the corresponding basic clusters in a time-multiplexed manner. This is possible as the basic clusters that are associated with the merged clusters do not execute concurrently as they belong to different data-paths. In order to ensure that each basic cluster can only be merged in a unique fashion, a heuristic is used to select a unique set of merged clusters with the aim to maximize the area utilization of the FPGA resources.

In the final stage (*data-path combination and resource sharing*), the basic clusters in the custom instruction data-paths are replaced with the selected merged clusters. Custom instruction data-paths that incorporate common merged clusters are then combined to construct a final data-path such as that shown in Figure 6-1(c). Note that multiplexers may be inserted to facilitate interconnect sharing between the clusters. In the example, the resulting data-path consists of a set of merged clusters (i.e. $C_1^1 \oplus C_2^4$ and $C_1^2 \oplus C_2^5$) and one basic cluster (i.e. C_1^3). In order to further minimize the area costs, we can perform resource sharing on the clusters of separate data-paths (data-paths that have not been combined) only if the resulting data-path does not lead to higher area-delay product. Since the example in Figure 6-1(c) has only one single resulting data-path, resource sharing is not performed.

It can be observed that the resulting data-path in Figure 6-1(c) contains an equivalent number of operations and interconnections as the original data-paths in Figure 6-1(a). Hence, in contrast to the resource sharing based method, our approach may not lead to fewer operations and interconnections. However, it can be seen in Figure 6-1(c) that the actual FPGA implementation results of our approach have higher area reduction when compared to the resource sharing based approach (i.e. Figure 6-1(b)). In addition, our approach leads to a lower critical path delay.

The proposed method only attempts to merge basic clusters that can be mapped uniformly onto BLEs to form logic blocks that share the same hardware configuration. This is applicable for most of the operations in our experiments. Operations, which cannot be mapped uniformly onto the logic blocks, form basic clusters that will not be considered for merging. Note that these operations are usually realized using embedded FPGA IP cores. The current method also does not consider optimized circuits (e.g. carry-select adder, compressor tree, etc.), or specialized operations (e.g. counters) that exploits the carry-chain and multiplexers within the BLEs for efficient realization. These circuits/operations cannot be directly identified from the high-level representation of the applications [116] and, therefore, are not considered in the proposed method.

6.3 CLUSTER MERGING FOR AREA OPTIMIZATION

The proposed cluster merging technique relies on the basic clusters obtained from the cluster generation process. In this chapter, we focus on cluster merging. The resource sharing algorithm employed in the last stage of the proposed method is adopted from [92].

6.3.1 Categorizing the Basic Clusters into Cluster Groups

The basic clusters from the cluster generation process can be categorized into three cluster groups as shown in Figure 6-2.

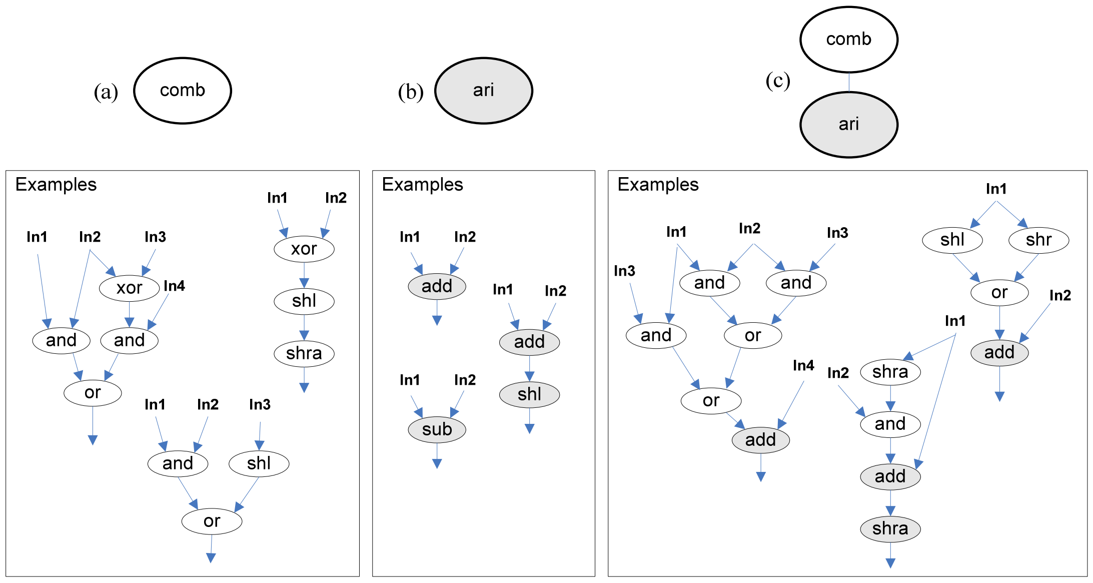


Figure 6-2: Cluster groups and the corresponding examples

The *comb* group (Figure 6-2(a)) consists of only logical and/or relational operations (i.e. *SHL*, *SHRA*). The *ari* group (Figure 6-2(b)) consists of an *ADD*/*SUB* operation that may co-exist with a relational operation. Relational operations in the *ari* group (i.e. *SHL*, *SHRA*) must execute after the *ADD*/*SUB* operation (refer to the legality check for including an operation in the basic cluster in Section 4.2.1). Finally, the third group consists of a combination of the *comb* and an *ari* group (Figure 6-2(c)). Operations in the *comb* group must execute before the operations in *ari*.

6.3.2 Cluster Merging

Figure 6-3 shows the algorithm for cluster merging. Initially, the basic clusters that are obtained from the cluster identification stage are stored in both Cs_i and Cs_0 (line 1). In the first iteration (lines 4-19), each pair of basic clusters are evaluated for cluster merging. The resulting merged clusters are stored in Cs_{i+1} (line 14). In subsequent iterations, the newly computed merged clusters from the previous iteration (i.e. Cs_i) are evaluated for cluster merging with the original set of basic clusters (i.e. Cs_0). This iterative process is repeated until no merged clusters are found in a particular iteration (i.e. condition expression equates to *false* in line 4). In the second part of the algorithm, a unique set of merged clusters is selected from the merged cluster sets (i.e. from the sets $Cs_i ,\ldots, Cs_1$) and the original set of basic clusters (i.e. Cs_0) (line 20).

There are two ways for merging a pair of clusters: with or without introducing a multiplexer. The pair of clusters is first evaluated to determine if they can be merged without incurring a multiplexer (line 8). If this is not possible, then the cluster pair is evaluated to determine if they can be merged by introducing a multiplexer (line 10). Preference is given to the former as introducing a multiplexer requires additional input pins (i.e. the multiplexer select pin) and this may violate the input constraints of the cluster. In the following sections, we will discuss these two cluster merging methods.

Algorithm 6-1

```
CLUSTER-MERGING ($Cs_0$)
 1. $Cs_1 = Cs_0$
 2. $i = 1$
 3. merge_cluster_pair_found := true
 4. while merge_cluster_pair_found = true
 5.     merge_cluster_pair_found := false
 6.     for all clusters $C_x^i$, where $C_x^i \in Cs_i$
 7.         for all clusters $C_y^0$, where $C_y^0 \in Cs_0$
 8.             success := MERGE-CLUSTERS-WITHOUT-MUX $\left(C_x^i, C_y^0\right)$
 9.             if success = false then
10.                 success := MERGE-CLUSTERS-WITH-MUX $\left(C_x^i, C_y^0\right)$
11.             end
12.             if success = true then
13.                 merge_cluster_pair_found := true
14.                 $Cs_{i+1}$ = STORE-MERGE-CLUSTER $\left(C_x^i, C_y^0\right)$
15.                 $i{+}{+}$
16.             end
17.         end
18.     end
19. end
20. solution := STORE-MERGE-CLUSTER ($Cs_i$, $Cs_{i-1}$, ..., $Cs_0$)
21. return solution
```

Figure 6-3: Cluster merging algorithm

1. Merging Clusters Without Multiplexer

Figure 6-4 describes the algorithm for determining if a cluster pair can be merged without introducing a multiplexer. As mentioned earlier, a pair of clusters can be merged if the resulting merged cluster can still be implemented using a single logic block. The first step of the algorithm partitions the cluster pair into their corresponding *comb* and *ari* components (lines 2-3).

Figure 6-5 shows two approaches to merge clusters without introducing a multiplexer. Let C_x be a cluster, and $comb_x / ari_x$ the corresponding *comb*/*ari* components. In Figure 6-5(a), $comb_x$ consists of a multiplexer that has been introduced in previous iterations. Note that there is an external input to the multiplexer (i.e. In_1) that is not connected to any internal logic within $comb_x$. We denote such an input as an *unused-mux-input*. For simplicity, we have omitted all inputs/outputs of the clusters that are irrelevant to the current discussion. Cluster merging of C_x and C_y is achieved by connecting the output port of $comb_y$ to *unused-mux-input*.

The second approach to merge C_x and C_y is shown in Figure 6-5(b). In this scenario, C_x does not have a *comb* component and C_y does not have an *ari* component. Cluster merging is achieved by simply connecting the output port of $comb_y$ to the input port of ari_x.

Algorithm 6-2

```
MERGE-CLUSTERS-WITHOUT-MUX ($C_x$, $C_y$, $K$)
1.  $success$ := true
2.  ($comb_x$, $ari_x$) = PARTITION-CLUSTER ($C_x$)
3.  ($comb_y$, $ari_y$) = PARTITION-CLUSTER ($C_y$)
4.  if $(ari_x \neq \phi)$ and $(ari_y \neq \phi)$
5.      if $(ari_x \neq ari_y)$
6.          $success$ := false
7.      end
8.  end
9.  if $(ari_x \neq \phi)$ and $(ari_y = \phi)$
10.     if SHIFT-BY-CONSTANT-EXIST ($ari_x$)
11.         $success$ := false
12.     end
13. end
14. if $success$ = true and $(C_x \neq C_y)$
15.     if $(comb_x \neq \phi)$ and $(comb_y \neq \phi)$
16.         $success$ := UNUSED-MUX-PIN-EXIST ($comb_x$)
17.     end
18.     if $(ari_x \neq \phi)$ and $(comb_y \neq \phi)$
19.         $success$ := IS-IDENTITY ($comb_y$)
20.     end
21. end
22. if INPUT-CONSTRAINT-VIOLATED ($K$, $comb_x$, $ari_x$, $comb_y$, $ari_y$)
23.     $success$ := false
24. end
25. return $success$
```

Figure 6-4: Algorithm to evaluate if a cluster pair can be merged without a multiplexer

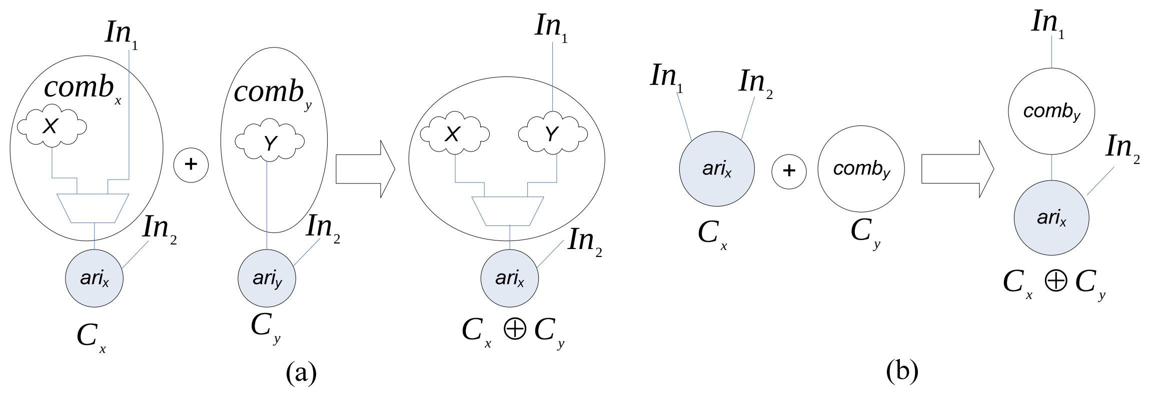


Figure 6-5: Two approaches for merging clusters without introducing an additional multiplexer

The cluster merging approaches discussed above can only be achieved when certain conditions are satisfied. These four conditions, which are evaluated in lines 4-21 of Algorithm 6-2, are described below. When the necessary Conditions 1 and 2 are met, Conditions 3 and 4 form the sufficient conditions to merge a pair of clusters. Note that the conditions for merging clusters C_x and C_y are commutative.

- **Necessary Condition 1**: *If C_x and C_y consist of ari components (e.g. Figure 6-5 (a)) and they can be merged, then $ari_x = ari_y$.* **Proof**: We prove this by contradiction. Suppose that C_x and C_y can be merged and they both consist of different *ari* components ($ari_x \neq ari_y$). The merged cluster $C_x \oplus C_y$ is implemented in a single logic block (according to Definition 6-1), which can perform the functionalities of C_x and C_y including the arithmetic functions ari_x and ari_y. However, this contradicts the fact that each logic block can only implement one *ari* component (see Section 4.2.1). Hence, for C_x and C_y to merge, ari_x and ari_y must be the same. This condition is evaluated in lines 4-8 of Algorithm 6-2.

- **Necessary Condition 2**: *If only one of the clusters C_x or C_y has an ari component (e.g. Figure 6-5(b)) and they can be merged, then the ari component cannot have a shift-by-constant operation*. **Proof**: We prove this by contradiction. Suppose that C_x and C_y can be merged and only C_x has an *ari* component (i.e. ari_x), which includes a shift-by-constant operation. The merged cluster $C_x \oplus C_y$ is implemented in a single logic block, which can perform the functionalities of C_x and C_y. In order for the merged cluster to perform the functionalities of C_y, the *ari* component must be convertible to an identity function (for example in Figure 6-5(b), by assigning an identity operand to In_2 (e.g. $In_2 = 0$) in $C_x \oplus C_y$, we can obtain the result of $comb_y$ if ari_x can be converted to an identity function). However it is not possible to convert ari_x with a shift-by-constant operation (assuming the constant is non-zero) to an identity function as the shift operation will always alter the input value (except when the input value is zero). Hence, this contradicts the assumption that C_x and C_y can be merged. This condition is evaluated in lines 9-13 of Algorithm 6-2, where the SHIFT-BY-CONSTANT-EXIST function checks if the *ari* component has a shift-by-constant operation.
- **Sufficient Condition 3**: *If Conditions 1 and 2 are met, and if both C_x and C_y have a comb component, then cluster merging is possible if one of the clusters has*

an unused-mux-input (e.g. Figure 6-5(a)). This condition is evaluated in lines 15-17 in Algorithm 6-2 using function UNUSED-MUX-PIN-EXIST.

- **Sufficient Condition 4**: *If Conditions 1 and 2 are met, and if at least one of the clusters has a comb component (e.g.* $comb_y$ *in Figure 6-5(a) and Figure 6-5(b)) while the other clusters has an ari component, then cluster merging can be achieved if the comb component can be converted to an identity function.* This enables the merged cluster to perform the functionality of the *ari* component. This condition is evaluated in lines 18-20 of Algorithm 6-2 using the IS-IDENTITY function.

Figure 6-6 provides examples to show how two clusters can be merged without introducing a multiplexer. In Figure 6-6(a), conditions 1, 2 and 4 are not applicable as none of the clusters have an *ari* component. The clusters C_x and C_y can be merged as they satisfy condition 3, i.e. C_x has an *unused-mux-input*.

In Figure 6-6(b), conditions 1 and 3 are not applicable. Clusters C_x and C_y can be merged as they satisfy condition 2 and 4, i.e. the *ari* component in C_x does not have a shift-by-constant operation and the *comb* component in C_y can be converted to an identity function. Note that the latter condition allows the merged cluster to perform the arithmetic operation in C_x by assigning In_4 to ‘1’, and In_5 to ‘0’.

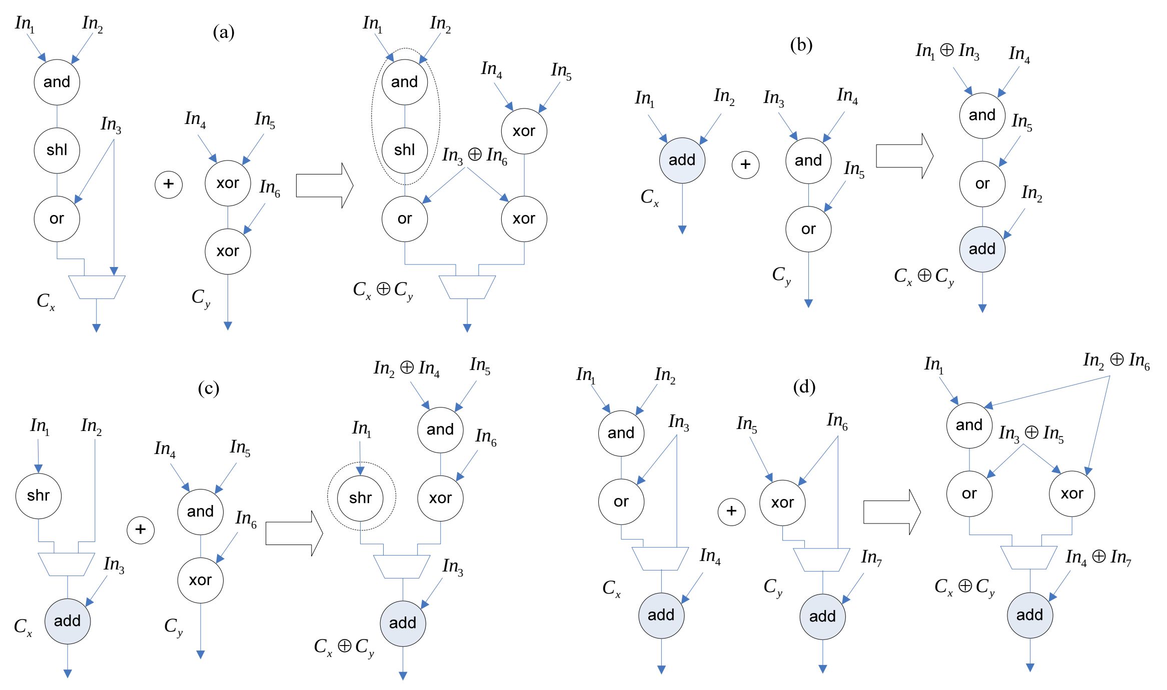


Figure 6-6: Examples of cluster pairs that can be merged without a multiplexer

In Figure 6-6(c), condition 1 is not applicable. Clusters C_x and C_y satisfy conditions 2, 3 and 4. In this case, the clusters are merged by connecting the output of C_y to the *unused-mux-input* of C_x. In addition, the merged cluster $C_x \oplus C_y$ can perform the arithmetic operation in C_x by assigning In_5 to '1', and In_6 to '0'.

Finally in Figure 6-6(d), condition 2 is not applicable. Both clusters have an *ari* component and hence they must first satisfy condition 1. In addition, they also satisfy condition 3 so that the *comb* component of C_y can be connected to the *unused-mux-input* of C_x. Finally, condition 4 is also satisfied and the merged cluster $C_x \oplus C_y$ can perform the arithmetic operation in C_x by assigning $In_2 \oplus In_6$ to '0'.

The evaluation of the conditions for cluster merging relies on three functions i.e. SHIFT-BY-CONSTANT-EXIST, UNUSED-MUX-PIN-EXIST and IS-IDENTITY. The first two functions are trivial as long as the program maintains a record of all the operations and the connectivity information between the operations in each cluster. Hence, we will only describe the implementation of the IS-IDENTITY function.

Implementation of IS-IDENTITY Function

The IS-IDENTITY function aims at evaluating whether the *comb* component of a cluster can produce an output that is identical to one of the inputs. In order to evaluate whether a particular input can be produced at the output, the remaining inputs are first assigned to ‘0’s or ‘1’s. Then the DFG of the *comb* component is progressively simplified as the inputs are propagated to the output by means of dataflow identity and dominance laws [117]. If the input is directly connected to the output when the process completes, then the *comb* component can be converted to an identity function.

The proposed approach to determine if a DFG is an identity function draws certain similarities with the method presented in [118] to transform a complex DFG pattern to a simpler pattern by applying identity operands to the nodes in order to eliminate them from the pattern. However, the method presented in [118] assumes that each node in the DFG has an external input that can be assigned to an identity operand. In addition, the method in [118] does not take into account shift-by-constant operators, which cannot be converted into an identity operation. The proposed method overcomes these limitations.

Algorithm 6-3

```
IS-IDENTITY (Comb_x)
1.  Identify shift-by-constant operations and store them in roots
2.  for all shift operations i in roots
3.      perform depth-first-search to obtain reverse sub-tree S rooted at i
4.      remove all vertices/edges $(v,e) \in S$, where $(v,e) \notin S \cap C_x$
5.      assign '0' to edges $e' = (i, v')$, where $(v', e') \in C_x - S$
6.  end
7.  success := false
8.  for all inputs u of $comb_x$
9.      find next input v, where $u \neq v$
10.     success := ASSIGN-OPERAND $(u, v, 0, comb_x)$
11.     if success = true then break; end
12.     success := ASSIGN-OPERAND $(u, v, 1, comb_x)$
13.     if success = true then break; end
14. end
15. return success
```

Figure 6-7: Algorithm to identify if a *comb* component can be converted to an identity function

Figure 6-7 and Figure 6-8 show the proposed algorithms for determining if $comb_x$ can be converted to an identity function. We will describe the algorithms based on the example in Figure 6-9. The first part of the algorithm (lines 1-6 of Algorithm 6-3) removes all the nodes in the reverse sub-trees rooted at the shift-by-constant operations (e.g. node 4 in Figure 6-9(a)). The function first identifies all the root nodes in $comb_x$ (line 1 of Algorithm 6-3) before finding the reverse sub-tree members of the corresponding root nodes using the depth-first-search algorithm (line 3 of Algorithm 6-3). Each sub-tree can only have one shift operation that must be a root node. Note that only the nodes that are exclusive to the sub-tree are removed (line 4 of Algorithm 6-3).

Algorithm 6-4

```
ASSIGN-OPERAND ($u, v, op\_value, comb_x$)
 1. success := false
 2. assign op_value to edges that are incident to v
 3. find next input w, where $w \neq u \neq v$
 4. if $w = \phi$ then
 5.     ELIMINATE-OP ($comb_x$)
 6.     if input u is connected directly to output of $comb_x$ then
 7.         success := true
 8.     end
 9. else
10.     success := ASSIGN-OPERAND ($u, w, 0, comb_x$)
11.     if success = true then return success; end
12.     success := ASSIGN-OPERAND ($u, w, 1, comb_x$)
13. end
14. return success
```

Figure 6-8: ASSIGN-OPERAND function

The rationale for removing the nodes in the reverse sub-trees rooted at the shift-by-constant operations is as follows. As discussed in Section 4.2.1, the inputs of these reverse sub-trees are assigned dedicated input pins (e.g. In_1^{shr} in Figure 6-9(b)), which are hardwired to the required shifted operands. Each sub-tree can then be treated as a hypothetical node with dedicated input pins. As the hypothetical nodes cannot be converted to an identity function (due to the fact that the shift operations will alter any non-zero input values), they are removed from $comb_x$ by assigning the dedicated input pins to '0's. This produces '0's at the output of the hypothetical nodes as shown in Figure 6-9(c) (line 5 of Algorithm 6-3).

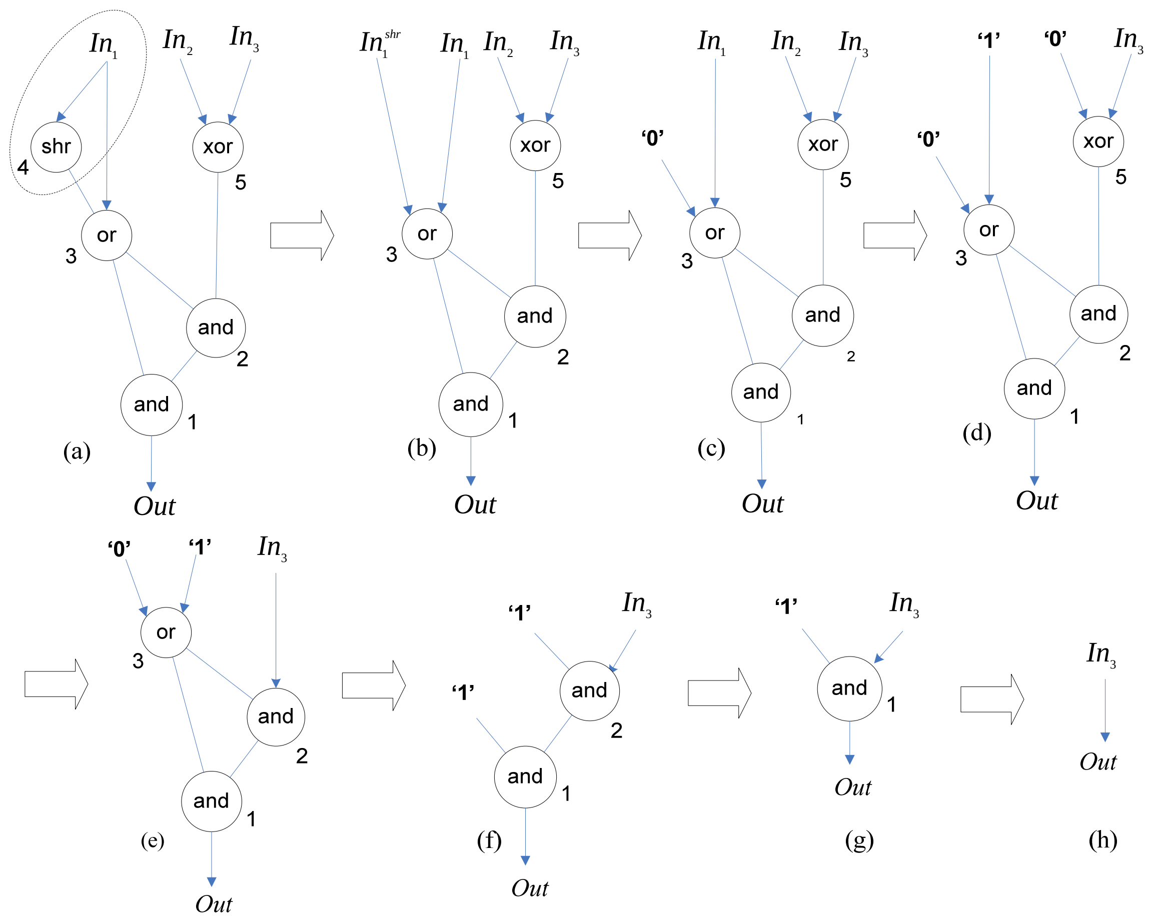


Figure 6-9: Identifying if a *comb* component can be converted to an identity function

Next, the inputs of $comb_x$ are evaluated one at a time to determine if they can be produced at the output after eliminating all the operations within $comb_x$ by means of dataflow identity and dominance laws. This is achieved by recursively assigning ‘0’s and ‘1’s to the remaining inputs (using the ASSIGN-OPERAND function in lines 10 and 12 of Algorithm 6-3 and Algorithm 6-4) and propagating these values through the operators in $comb_x$ (using the function ELIMINATE-OP in line 5 of Algorithm 6-4). The ASSIGN-OPERAND function considers all the 2^n Boolean assignments of the remaining *n* number of external inputs to the *comb* component. For example in Figure 6-9(d), to evaluate whether In_3 can be produced at the output, In_1 and In_2 are assigned to ‘1’ and ‘0’ respectively (using the ASSIGN-OPERAND function). In the subsequent steps (Figure 6-9(e)-(h)), these values are propagated to the operations one at a time beginning from the predecessor nodes (using the ELIMINATE-OP function).

As the operations in the *comb* component have been sorted according to their order of dependency, the operations are progressively removed in the order of the dependency graph by means of dataflow identity and dominance laws. At each step, the output of an operation is produced based on whether the operands are identity and dominating operands. The operation is then removed. In this example, In_3 is directly connected to the output after all the operations have been removed and, hence, $comb_x$ can be converted to an identity function.

In the worst case, the time complexity of ASSIGN-OPERAND for a single iteration of Algorithm 6-3 (lines 8-14) is $O(2^{n_{in}-1})$, where n_{in} is the number of external inputs to the *comb* component. In practice, n_{in} is usually very small (e.g. 4-6 depending on the target FPGA). The execution time complexity of ELIMINATE-OP is $O(n_{op})$, where n_{op} is the number of operations in the *comb* component. Note that the linear time complexity can be achieved as the operations in the *comb* component have been sorted according to their order of dependency.

Checking for Input Constraints Violation of Merged Cluster

As no additional multiplexer has been introduced in the merging process, the input constraints of the merged cluster will not be violated if all the input pins of the original cluster with fewer inputs can be merged. For example, in Figure 6-6(d), all the input pins of C_y have been merged with those of C_x. Since both the original clusters already conform to the input constraints, the number of input pins of the resulting merged cluster will also not exceed *K*. However, there is a possibility that the input pins of the original clusters cannot be merged. This is shown in Figure 6-6(a) and (c), whereby only a single input pin of the original clusters is merged (i.e. $In_3 \oplus In_6$ in Figure 6-6(a) and $In_2 \oplus In_4$ in Figure 6-6(c)). The reason for this is that shifted values of the inputs affected by the shift-by-constant operation (e.g. *AND-SHL* in Figure 6-6(a) and *SHR* in Figure 6-6(c)) will be hardwired to the BLEs and, hence, cannot serve as valid inputs to other logic functions (that do not require the same shifted values). Hence, there is a need to check if the required inputs of the merged cluster exceed *K* (line 22 of Figure 6-4). We have employed the Algorithm 4-1 for checking the input constraint violation of the merged clusters.

2. Merging Clusters With Multiplexer

If the cluster pair cannot be merged with the method described in the previous section, a multiplexer will be introduced in an attempt to merge the cluster pair. Figure 6-10 shows the various scenarios for cluster pair C_x and C_y to be merged by introducing a multiplexer. When neither of the clustered pair consists of an *ari* component (e.g. Figure 6-10(a)), the multiplexer is inserted at the output of C_x and C_y. However, if at least one of the cluster pair has an *ari* component (e.g. Figure 6-10(b) and (c)), the multiplexer must be inserted at the input of the arithmetic operation. This is due to the fact that all logical operations (we assume that the multiplexer is a logic operator) must execute before the arithmetic operation in a cluster (see Section 4.2.1).

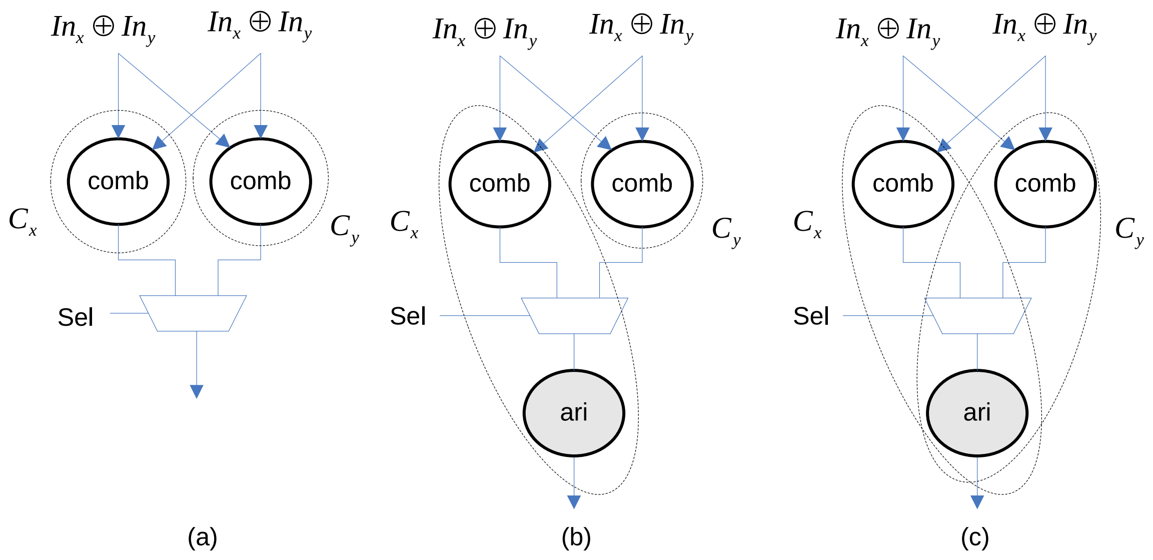


Figure 6-10: Incorporating multiplexer into a merged cluster

The input pins of C_x and C_y are also merged whenever possible and an additional input pin is required for the multiplexer select. The actual number of required input pins of the merged cluster candidate must be recalculated using the algorithm in Algorithm 4-1 to verify if the input pin constraint is still met.

3. Choosing Unique Sets of Merged Clusters

The cluster merging method will result in a basic cluster appearing in a number of merged clusters. In order to ensure that each basic cluster can only be merged once, there is a need to select a unique set of merged clusters from the merged cluster set.

A compatibility graph is constructed to select a unique set of merged clusters from $Cs_i, \ldots, Cs_0$ (line 20 in Figure 6-3). The set Cs_i for $i \neq 0$ is a set of merged clusters where each merged cluster can be formed by merging *i* basic clusters. The compatibility graph approach is similar to that proposed in [92] for selecting a set of resources in two data-paths for merging. However, our approach differs from that in [92] as we consider the selection of basic clusters for merging in all the custom instruction data-paths (instead of two data-paths at a time as in [92]). This global selection strategy can lead to better quality results.

Definition 6-2: A compatibility graph is an undirected graph $G_u(V_u, E_u)$ where:

- A vertex $v \in V_u$ is a merged cluster which consists of *i* basic clusters that can be merged to form *v*. A vertex *v* is associated with a weight $w(v) = i^2 \times freq$, where *freq* is the number of occurrences of the basic clusters in *v* that is found across all the custom instructions. The weight has been chosen in order to maximize cluster sharing among the most frequently occurring clusters in the custom instruction data-paths. The factor i^2 is used as the value of *i* is typically much lower than *freq*. Note that the value of *i* varies according to the number of basic clusters that are used to form the merged cluster (see line 14 of Algorithm 6-1).

- There is an arc $e = (u, v) \in E_u$ if the merged clusters represented by *u* and *v* are compatible.

Definition 6-3: Vertex *u* and vertex *v* are not compatible if a basic cluster associated with *u* also exists in *v*.

In the worst case, the number of vertices $|V_u|$ in the compatibility graph, is $\sum_{r=1}^{i} \frac{n_{bc}!}{r!(n_{bc} - r)!}$, where n_{bc} is the number of basic clusters in Cs_0. *i* is the number of basic clusters that are used to form a merged cluster in the corresponding iteration in the *while* loop of Algorithm 6-1 (lines 4-19). This equation assumes that in iteration *i*, all combination of *i* basic clusters can be merged. Hence, in each iteration of the *while* loop, the number of possible merged clusters can be calculated using the combination function $\frac{n_{bc}!}{i!(n_{bc} - i)!}$. Note that this assumption will not be the case in practice. The maximum number of compatibility graph vertices in our experiments is less than 200. In addition, the maximum value of *i* in our experiments is five, as it is unlikely to find more than five clusters that can be merged into a logic block. The time complexity to construct the compatibility graph is $O(|V_u|^2)$ as there is a need to check for the compatibility of each vertex with all other vertices.

In order to identify a unique set of merged clusters that would maximize the FPGA resource utilization, the maximum weight clique is heuristically computed from the compatibility graph. The following definition of maximum weight clique is obtained from [92].

Definition 6-3: The maximum weight clique of a graph $G_c(V_c, E_c)$ is a set of vertices $C \subseteq V_c$ where for all vertices $u\ ,v\ \in C$, the arc $(u,v) \in E_c$ and $\sum_{\forall v \in C} w(v)$ is maximum.

As mentioned in [92], the execution time to compute the maximum weight clique can be polynomially bounded by $|V_u|$.

6.4 RESOURCE SHARING OF CLUSTERS IN THE COMBINED DATA-PATH

The basic clusters in the custom instruction data-paths are replaced with the selected merged clusters so as to determine the resulting data paths. This involves combining custom instruction data-paths based on the binding of basic clusters that are associated with the unique merged cluster set. This may necessitate introducing multiplexers in order to maintain the correctness of data-path of the associated custom instruction.

In order to further minimize the area costs without incurring additional area-time overhead, resource sharing is performed on the basic clusters and merged clusters that do not reside on the same data-path. A cluster can be considered for resource sharing only once to avoid increasing the critical path delay. In addition, if resource sharing between the clusters leads to inferior area-time results, the solution prior to resource sharing is adopted. In this work, we have employed the resource sharing method that was presented in [92] for merging a pair of clusters. It is noteworthy that in most of the experiments considered, the proposed approach leads to very little opportunity for resource sharing.

6.5 EXPERIMENTAL RESULTS

In this section, we compare the results of the proposed approach with results from: 1) a commercial FPGA implementation tool [119] that is targeted for area optimization with resource sharing option selected, and 2) one of the best known methods for resource sharing [92].

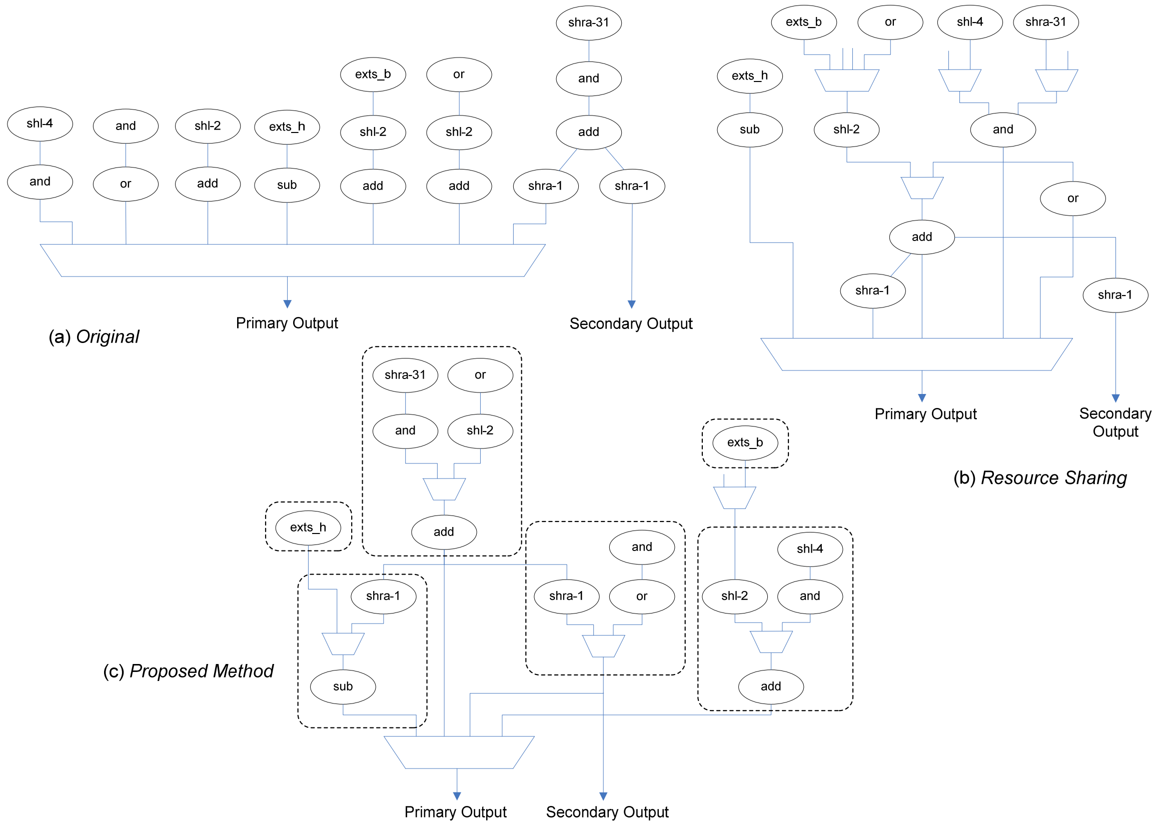


Figure 6-11: Implementation of custom instructions for Adpcm Enc on Virtex-5

We have used eight applications from the MiBench embedded benchmark and MediaBench benchmark suites. The custom instructions have been obtained using the LFF based template selection approach that is described in Chapter 3. The outputs of custom instruction data-paths of all three approaches are multiplexed to meet the two-output port constraint of the RFU. In particular, single output custom instructions are multiplexed to the primary output port of the RFU, and dual-output custom instructions are multiplexed to the primary and secondary output ports of the RFU as shown in Figure 6-11.

Figure 6-11 shows various implementation approaches for seven custom instructions from the Adpcm Enc application, where the target architecture is the Virtex-5 device. The *Original* approach (Figure 6-11(a)) is based on custom instruction data-paths that have been obtained using the LFF approach without further optimization. The *Resource Sharing* approach (Figure 6-11(b)) is based on the method presented in [92] to obtain an optimized data-path that maximizes the resource sharing of the original data-paths. The approach in [92] performs data-path merging on two data-paths at a time until all the data-paths have been considered. It can be observed that the number of operations has been significantly reduced compared to the original data-paths. For the *Proposed Method* (Figure 6-11(c)), the original custom instruction data-paths are subjected to the methods presented in this paper. The rectangular enclosures in Figure 6-11(c) indicate the basic/merged clusters. The resulting data-paths have larger number of operations compared to the *Resource Sharing* approach. However, we will show that the actual area utilization of *Proposed Method* is significantly lower than *Resource Sharing*. Note that the data-paths in Figure 6-11 will be subjected to further optimization during implementation with the FPGA tool.

Table 6-1 reports the total number of operations in the resulting data-paths that are generated using the various methods. The optimized data-paths that are generated using

Proposed Method are based on *K* = 4. When compared to results of *Original*, the *Resource Sharing* approach leads to an average reduction of over 29% in the number of operations. In contrast, *Proposed Method* has an average percentage reduction in the number of operations of less than 7%. These results confirm that unlike the resource sharing approach, the proposed method does not aim at maximizing resource sharing between the data-paths.

Table 6-1: Number of operations

Application	Original	Resource Sharing	Proposed Method
Adpcm Dec	16	12	14
Adpcm Enc	19	16	22
Bitcount	27	25	25
Blowfish Dec	25	16	23
Pegwit	65	40	60
Rijndael Dec	35	21	29
Rijndael Enc	38	29	44
Sha	51	26	36

The optimized data-paths for each of these methods have been designed in VHDL, implemented using Xilinx FPGA tool [119] and targeted on three state-of-the-art FPGA architectures, i.e. Spartan-3 (xc3s5000fg1156-4) [120], Virtex-4 (xc4vlx200ff1513-10) [108] and Virtex-5 (xc5vlx50ff1153-1) devices [109].

The Spartan-3 and Virtex-4 devices incorporate BLEs with 4-input LUTs. The Virtex-5 devices incorporate BLEs with 6-input LUTs that can be used to implement any 6-input function or two dual-output 5-input functions [121]. For the Spartan-3 and Virtex-4 solutions, the proposed method generates optimized data-paths for *K* = 4 such that all the clusters produced cannot have more than 4 inputs. For the Virtex-5 solution, two sets of results for $K = 5$ and $K = 6$ are first generated. The set which leads to the best area-time results is chosen. The data-paths produced using the various methods are implemented with the FPGA tool under the same design constraints and optimization options. In particular, we have enabled the implementation options for area optimization and resource sharing.

6.5.1 Comparing Area Measures

Table 6-2 shows the optimized area for the various approaches. In order to compare the area utilization using a common measure (i.e. number of slices or slice LUTs), we have disabled the option to map multiplication operations onto embedded multipliers and DSP blocks in the FPGA. The area utilization for the Spartan-3 and Virtex-4 are reported in slices, while the area utilization for Virtex-5 is reported in terms of slice LUTs. This enables us to gauge the relative complexity of the designs on various devices as the slices in the Virtex-5 devices are much larger than those in Spartan-3 and Virtex-4. The percentage values are the percentage area reduction of *Proposed Method* over *Resource Sharing*. It is worth mentioning that the original data-paths have also undergone area optimizations (with resource sharing as one of the optimization strategy) that are provided by the commercial tool.

Table 6-2: Area comparison

Application	Spartan-3				Virtex-4				Virtex-5			
	Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method	
Adpcm Dec	157	113	96	15.0%	156	113	96	15.0%	219	151	129	14.6%
Adpcm Enc	171	212	137	35.4%	171	211	139	34.1%	268	253	180	28.9%
Bitcount	160	176	117	33.5%	160	176	116	34.1%	324	254	252	0.8%
Blowfish Dec	261	197	158	19.8%	261	194	158	18.6%	419	247	225	8.9%
Pegwit	591	466	391	16.1%	590	466	391	16.1%	904	540	493	8.7%
Rijndael Dec	356	226	247	-9.3%	355	226	247	-9.3%	483	290	294	-1.4%
Rijndael Enc	1476	847	711	16.1%	1476	844	709	16.0%	3492	1745	1110	36.4%
Sha	430	348	288	17.2%	428	348	287	17.5%	628	384	350	8.9%

It can be observed that *Resource Sharing* and *Proposed Method* both lead to notable area reduction when compared to *Original*. The average area reduction of *Resource Sharing* over *Original* is 17.2%, 17.3% and 33.6% on Spartan-3, Virtex-4 and Virtex-5 respectively. For certain applications (e.g. Adpcm Enc and Bitcount) on Spartan-3 and Virtex-4, *Resource Sharing* results in lower area efficiency than *Original*. This is due to the fact that custom instructions in these two applications have little opportunity for resource sharing and hence, the area of the multiplexers introduced through the limited sharing of resources outweighs the area savings. In comparison, *Proposed Method* is capable of achieving higher area efficiency over *Original* in all cases. The average area reduction of *Proposed Method* over *Original* is 34.3%, 34.2% and 42.4% on the Spartan-3, Virtex-4 and Virtex-5 respectively. These results demonstrate that unlike resource sharing methods, the proposed method is still favorable for merging data-paths which do not have a high degree of similarity in the operations. It can be observed that the area reduction in both methods for Spartan-3 and Virtex-4 is comparable due to the similar characteristics of the logic elements in both architectures. The higher area reduction for Virtex-5 is due to the larger LUTs in the architecture that enable more operations to be mapped onto a BLE. In the proposed method, the higher number of input pins allowable in a logic group for the Virtex-5 architecture also enables more clusters to be merged.

Proposed Method results in higher area efficiency than *Resource Sharing* in almost all cases. The only case where *Resource Sharing* leads to notably higher area efficiency than *Proposed Method* is for application Rijndael Dec on Spartan-3 and Virtex-4. Note that the percentage area difference for this case is less than the percentage area reduction of *Proposed Method* in all the other applications on the Spartan-3 and Virtex-4 device. The custom instructions in Rijndael Dec have many similar operations and hence the resource sharing method is more favorable in terms of area minimization. However as shown in the next sub-section, the area optimization of *Resource Sharing* is achieved at the cost of incurring large critical path delays. When compared to one of the best known resource sharing based method, the proposed method can achieve an average area reduction of 18%, 17.8% and 13.2% on the Spartan-3, Virtex-4 and Virtex-5 respectively. These results demonstrate that an approach that maximizes resource sharing may not be the best method for FPGA area optimization.

6.5.2 Comparing Critical Path Delays

While resource sharing based approaches can lead to area savings, they may incur undesirable delay in the data-paths due to the extensive introduction of multiplexers whenever operations are shared across data-paths. In contrast, the proposed method judiciously introduces multiplexers at a coarser grain, i.e. for interconnect sharing when the clusters in different data-paths are merged. In addition, the proposed method attempts to maximize the logic utilization of the FPGA logic elements, which can lead to less critical path delay in the data-paths. Table 6-3 shows the critical path delay of the optimized data-paths. The percentage values are the percentage delay reduction of *Proposed Method* over *Resource Sharing*. The bracket below each critical path value shows the logic and route delay that constitute the critical path.

It can be observed that *Resource Sharing* leads to highest critical path delay in almost all cases. In Virtex-5, *Proposed Method* has a critical path delay that is marginally higher than *Resource Sharing* for only one application, i.e. Rijndael Enc (difference of less than 1ns). When compared to the implementation results of *Original*, *Resource Sharing* has an average increase in critical path delay of 20.0%, 28.8% and 18.9% on the Spartan-3, Virtex-4 and Virtex-5 respectively. In comparison, *Proposed Method* has an average increase in critical path delay over *Original* of only 4.3%, 2.2% and 11.3% on the Spartan-3, Virtex-4 and Virtex-5 respectively. In certain applications on the three FPGA devices, it can be observed that *Proposed Method* can lead to lower critical path delay than the original data-paths. This is contributed by two reasons. Firstly, in certain applications, the combination of data-paths due to cluster merging in the proposed method has resulted in less complex output multiplexer. Secondly, in cases where the *Proposed Method* has lower critical path delay than the original data-paths, it can be observed that the proposed method has led to a notable reduction in the routing delay. This is due to the efficient packing of operations within the logic blocks in the proposed method that has enabled the implementation tool to perform tighter placement which, in turn, leads to more effective routing.

Table 6-3: Critical path (ns)

Application	Spartan-3				Virtex-4				Virtex-5			
	Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method	
Adpcm Dec	28.1 (10.2, 17.9)	34.9 (12.0, 23.0)	26.9 (10.3, 16.7)	22.9%	21.3 (5.2, 16.1)	20.1 (7.9, 12.2)	15.5 (5.9, 9.7)	22.7%	12.2 (5.3, 6.9)	14.1 (5.0, 9.2)	12.2 (4.7, 7.5)	13.5%
Adpcm Enc	28.1 (9.5, 18.6)	38.2 (11.7, 26.6)	26.7 (11.6, 15.2)	30.1%	18.9 (5.3, 13.6)	26.3 (8.1, 18.2)	18.1 (7.3, 10.8)	31.1%	13.7 (5.0, 8.8)	14.7 (5.4, 9.3)	14.3 (5.4, 8.9)	2.8%
Bitcount	44.9 (18.9, 26.0)	44.8 (19.3, 25.5)	49.5 (19.1, 30.5)	-10.6%	29.5 (11.8, 17.6)	35.0 (11.9, 23.1)	27.7 (11.3, 16.4)	20.6%	21.8 (9.0, 12.8)	25.3 (8.8, 16.5)	22.5 (8.6, 13.9)	11.3%
Blowfish Dec	30.9 (12.7, 18.2)	40.7 (15.2, 25.5)	33.9 (14.7, 19.1)	16.7%	20.4 (7.9, 12.5)	26.9 (9.6, 17.3)	21.6 (9.2, 12.5)	19.5%	14.5 (5.8, 8.7)	18.3 (6.5, 11.8)	18.1 (7.1, 11.0)	1.4%
Pegwit	43.8 (18.7, 25.0)	50.0 (23.6, 26.4)	49.5 (21.6, 27.9)	1.0%	27.4 (12.8, 14.6)	36.3 (13.8, 22.5)	34.3 (12.7, 21.7)	5.6%	23.8 (8.8, 14.9)	29.4 (9.1, 20.3)	29.1 (9.1, 19.9)	1.2%
Rijndael Dec	35.1 (14.0, 21.0)	42.8 (17.3, 25.6)	37.0 (17.1, 19.9)	13.7%	20.8 (8.8, 12.0)	28.9 (11.2, 17.7)	23.9 (10.6, 13.3)	17.4%	15.9 (5.5, 10.3)	19.3 (7.7, 11.6)	16.2 (6.3, 9.9)	15.7%
Rijndael Enc	50.0 (23.1, 26.9)	49.8 (22.5, 27.2)	49.1 (22.5, 26.5)	1.3%	33.8 (12.6, 21.2)	42.9 (13.0, 29.8)	32.1 (13.8, 18.2)	25.2%	28.0 (9.4, 18.6)	29.3 (7.9, 21.4)	30.1 (9.8, 20.4)	-2.8%
Sha	33.8 (13.2, 20.6)	44.5 (17.4, 27.1)	35.9 (16.0, 19.9)	19.4%	21.3 (8.8, 12.5)	31.6 (10.3, 21.3)	24.2 (10.1, 14.0)	23.5%	16.8 (6.0, 10.8)	22.8 (6.7, 16.1)	21.1 (6.9, 14.3)	7.3%

When compared to *Resource Sharing*, *Proposed Method* has an average critical path delay reduction of 11.8%, 20.7% and 6.3% on the Spartan-3, Virtex-4 and Virtex-5 respectively. It can be observed that when compared to the original data-paths, *Proposed Method* generally leads to a lower increment in the routing delay than *Resource Sharing*. This is due to the more compact designs generated by the proposed method. In particular, when compared to *Original*, *Resource Sharing* has an average increase in routing delay of 21.1%, 36.7% and 26.9% on the Spartan-3, Virtex-4 and Virtex-5 respectively. In contrast, *Proposed Method* leads to an average increase in routing delay of less than 1% on the Spartan-3 and Virtex-4, and only 14.5% on Virtex-5. This confirms that the proposed method is not only capable of achieving a higher degree of area optimization when compared to resource sharing based approaches, but it can also lead to less critical path delay.

6.5.3 Comparing Area-Time Measures

In order to compare the overall benefits of the proposed method, we use the metric area-delay product which is obtained by multiplying the area (in terms of number of slice/ slice LUTs) with the critical path delay (in terms of nanoseconds). Table 6-4 shows the area-delay product of the optimized data-paths. The percentage values (in brackets) are the percentage area-delay product reduction of *Proposed Method* over *Resource Sharing*.

Table 6-4: Area-delay product

Application	Spartan-3				Virtex-4				Virtex-5			
	Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method		Original	Resource Sharing	Proposed Method	
Adpcm Dec	4406.8	3945.4	2582.9	34.5%	3323.6	2271.8	1491.4	34.4	2667.9	2134.1	1577.0	26.1%
Adpcm Enc	4800.1	8107.7	3663.4	54.8%	3229.3	5544.2	2515.3	54.6%	3678.3	3719.9	2573.1	30.8%
Bitcount	7187.4	7887.6	5797.1	26.5%	4712.2	6152.6	3218.4	47.7%	7071.6	6427.0	5657.9	12.0%
Blowfish Dec	8055.0	8010.2	5350.4	33.2%	5327.5	5209.9	3417.4	34.4%	6062.5	4524.8	4063.3	10.2%
Pegwit	25863.3	23296.3	19357.6	16.9%	16151.8	16933.0	13416.4	20.8%	21480.8	15879.8	14325.1	9.8%
Rijndael Dec	12489.5	9679.8	9130.1	5.7%	7382.6	6532.3	5895.4	9.8%	7660.9	5584.5	4771.0	14.6%
Rijndael Enc	73726.2	42139.9	34897.3	17.2%	49879.9	36166.2	22727.7	37.2%	97793.5	51116.3	33431.0	34.6%
Sha	14526.3	15499.2	10336.0	33.3%	9097.6	10984.6	6934.2	36.9%	10559.8	8750.2	7396.2	15.5%

Table 6-4 confirms that *Proposed Method* has lower area-delay product than *Original* and *Resource Sharing* in all cases. It can also be observed that in certain applications, the implementation of original data-paths using the optimization provided by the commercial tool can lead to lower area-delay product than the resource sharing approach. In particular when compared to *Original*, *Proposed Method* has an average area-delay product reduction of 31.4%, 32.5% and 36.3% on the Spartan-3, Virtex-4 and Virtex-5 respectively. When compared to *Resource Sharing*, *Proposed Method* has an average area-delay product reduction of 27.8%, 34.5% and 19.2% on the Spartan-3, Virtex-4 and Virtex-5 respectively. These results reinforce the benefits of the proposed method for area-time optimization on FPGA.

6.5.4 Execution Time

The execution time of the proposed method is longer than that required by the resource sharing approach to generate the optimized data-paths. However, both methods (resource sharing and proposed) can be executed in the order of milliseconds on a HP Workstation with two 2.66GHz processors and 2GB RAM. This is an insignificant overhead when compared to the time taken for the commercial FPGA tool to implement the optimized data-paths, which is typically in the order of seconds/minutes.

6.6 SUMMARY

In this chapter, we proposed a novel high-level optimization strategy for realizing area-time efficient custom instructions on FPGA devices. It leverages our existing technique to partition the custom instructions into a set of basic clusters such that the basic clusters can be efficiently mapped onto the LUT and carry-look-ahead structure of the FPGA logic blocks. We have proposed conditions to aid the merging of basic clusters without introducing multiplexers. We have employed a heuristic based on the degree of cluster merging and the frequency of occurrence of the basic clusters to accelerate the selection of a unique set of merged clusters. Resource sharing is then performed on clusters only if the resulting data-paths do not lead to an increase in the area-delay product. When compared to the area optimization capabilities of the commercial tool and to one of the most efficient methods reported in the literature for resource sharing, the proposed method can achieve significantly lower area-delay product for all cases considered in this study due to its architecture-aware cluster merging strategy to maximize utilization of FPGA logic blocks.

In the next chapter, we will show that the cluster merging technique can also lead to performance benefits for area-constrained RISPs with runtime reconfiguration support. In particular, cluster merging can maximize the utilization of the restricted FPGA space to implement the most profitable custom instructions in each runtime configuration. In addition, we will also show that cluster merging can result in significant reduction in reconfiguration overhead for partial reconfigurable models.

CHAPTER 7

LOOP-AWARE GROUPING OF MERGED CUSTOM INSTRUCTIONS

In the previous chapters, we presented strategies for realizing area-time efficient custom instruction on RISPs. These methods aim at maximizing the utilization of the FPGA space for implementing custom instructions. So far, we have assumed that the available FPGA space can sufficiently cater for the implementation of all the selected custom instructions. However, as explained in Chapter 2, with the advent of soft customizable embedded FPGA cores that can be integrated into microprocessors, RISPs in the near future are likely to impose tight area constraint on the FPGA for implementing custom instructions. This must be taken into consideration during instruction set customization for RISPs.

Runtime reconfiguration offers the potential to realize low cost systems that can still lead to high performance by changing the configuration of a small reconfigurable hardware at runtime. This provides an efficient means to reduce the hardware cost, while satisfying the performance, flexibility and power requirements of embedded systems. The two main drawbacks that discourage the use of runtime reconfiguration in embedded real-time systems is the large reconfiguration overhead in commercial FPGA architectures, and the lack of supporting tools and methodologies.

The growing complexity of the applications necessitates methods that can rapidly identify a suitable set of configurations by splitting the computational structures into temporal partitions in order to evaluate the benefits of runtime reconfiguration early in the design cycle. In this chapter, we present a hierarchical loop partitioning strategy that reduces the complexity of the search space for determining the runtime custom instruction configurations for RISPs. In particular, the proposed method leverages the cluster merging technique discussed in the previous chapter to: 1) maximize the utilization of the reconfigurable logic blocks in each configuration, and 2) reduce the runtime reconfiguration overhead. Experimental results show that runtime reconfiguration can lead to an average performance gain of over 45% and 52% for the full reconfiguration and partial reconfiguration model respectively. In addition, cluster merging can lead to over 39% average reduction in the runtime overhead for the partial reconfiguration model. When compared to the case where cluster merging is not considered during hierarchical loop partitioning, the proposed method (which incorporates cluster merging) can lead to an average performance gain of over 32% and 34% for the full and partial reconfiguration model respectively.

This chapter is organized as follows: In the next section, we present the target runtime reconfigurable RFU that incorporates multi-bit logic blocks with low reconfiguration latency. We will then introduce the proposed method for generating runtime custom instruction configurations for area-constrained RISPs that is based on a hierarchical loop partitioning strategy. Experimental results will be shown to demonstrate the viability of the proposed method for generating runtime configurations based on full and partial reconfiguration models. We will also show the significance of cluster merging in maximizing the performance benefits of runtime reconfiguration in RISPs. A preliminary version of this work has been published in [C-1].

7.1 TARGET ARCHITECTURE

In Appendix A, we discuss the multi-bit logic block architecture proposed in [122], where a group of BLEs share the same configuration memory to implement a single bit-slice of the data-path. We modified the multi-bit logic block architecture as shown in Figure 7-1 for the target architecture model used in our experiments in this chapter.

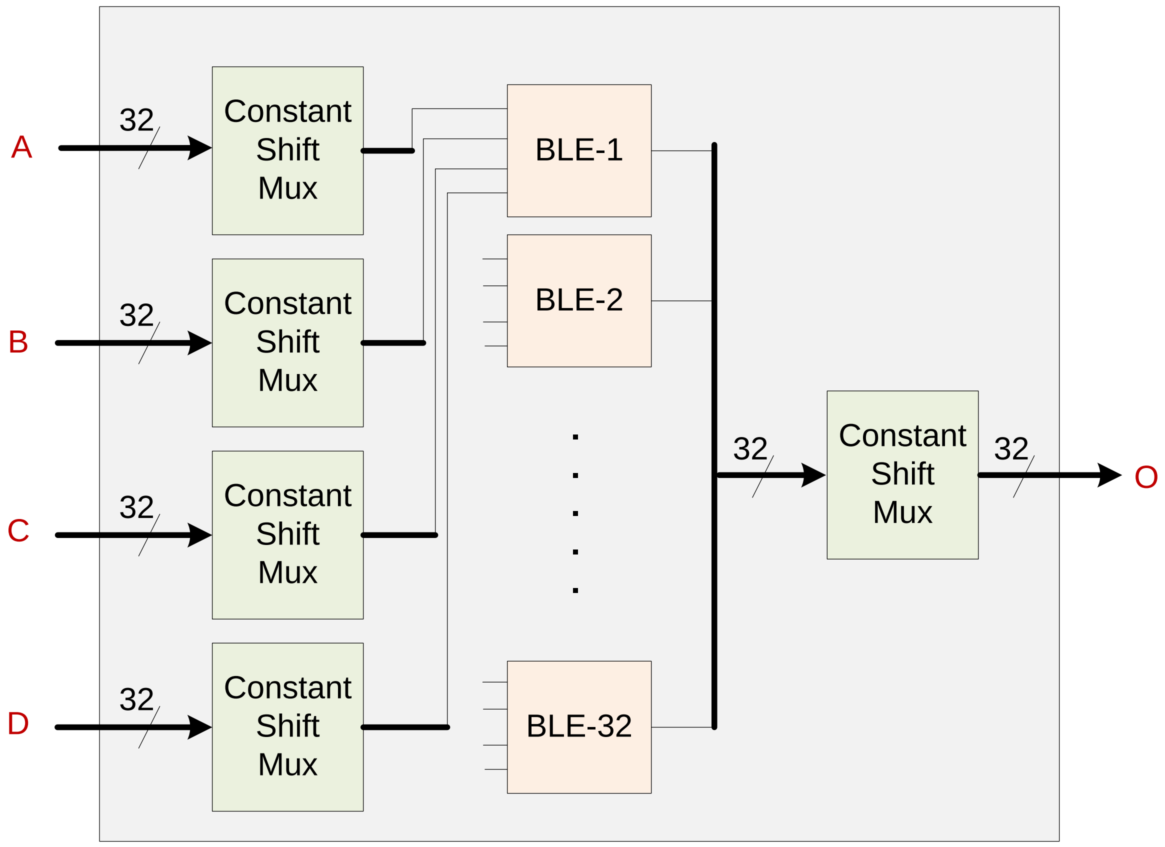


Figure 7-1: Multi-bit logic block architecture

Each logic block consists of 32 BLEs, where each BLE is composed of a 4-input LUT and fast carry-logic (see Figure A-1). The BLEs in a logic block share the same configuration memory and a multi-bit logic block can be used to implement a basic or merged cluster. As discussed in [122], each BLE has a total of 19 configuration bits.

The multi-bit logic block has four 32-bit inputs (i.e. *A*, *B*, *C*, *D*) and a 32-bit output (i.e. *O*). Each of these inputs and output is connected to a constant shift multiplexer unit that can efficiently perform the required logical/arithmetic shift-by-constant operations (e.g. *SHR*, *SHL*, *SHRA*) in the custom instructions. The constant shift multiplexer units located at the input of the logic block perform shift operations on the input data and outputs the shifted values to the corresponding BLE. The constant shift multiplexer unit located at the output of the logic block performs shift operations on the BLE outputs.

Figure 7-2 shows the architecture of the constant shift multiplexer unit. For simplicity, we have only shown the constant shift multiplexer unit with 4-bit input/output. The constant shift multiplexer unit consists of three sets of multiplexers that share the same configuration memory. It can be observed that the first set of multiplexers performs the shift-left-by-constant operation while the second set of multiplexers performs the shift-right-by-constant operation. Note that when the multiplexer select is '0', the input data are not shifted. The shift constants are used as the multiplexer select to obtain the corresponding shifted values of the input data. The third set of multiplexers chooses the required output (i.e. shift-left or shift-right). The number of configuration bits required by the first and second set of multiplexers is log_2N (number of multiplexer select signals) each, where N is the data width of the input/output. The number of configuration bits for the third set of multiplexer is only 1 as only a single bit is required for the multiplexer select. Hence, the total number of configuration bits for a constant shift multiplexer unit in our multi-bit logic block is $2log_2N + 1 = 11$ (where $N = 32$).

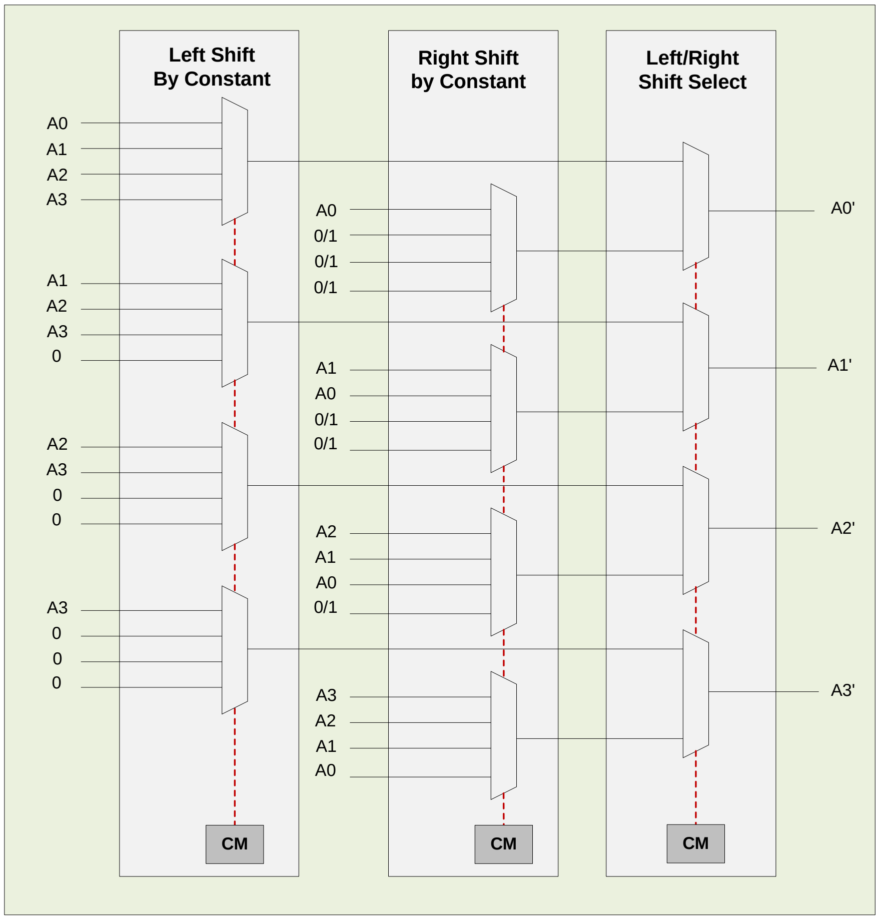


Figure 7-2: Architecture of constant shift multiplexer unit

As discussed in [123], the multi-bit logic block architecture also facilitates configuration memory sharing in the FPGA routing architecture. Figure 7-3 shows a possible multi-bit (or bus-based) routing architecture for our target multi-bit logic block. For simplicity, we have only shown an example of a bus-based routing architecture with a bus width of four. It can be observed that the input, output and switch block connections (we assume that only full connections are used) of the bus based routing architecture facilitate configuration memory sharing. Assuming that a single bit is used for each connection, the number of configuration bits for the input, output and switch block connections are 4, 4 and 16 respectively. Hence, the total number of configuration bits required for the routing architecture that are adjacent to a multi-bit logic block is 24.

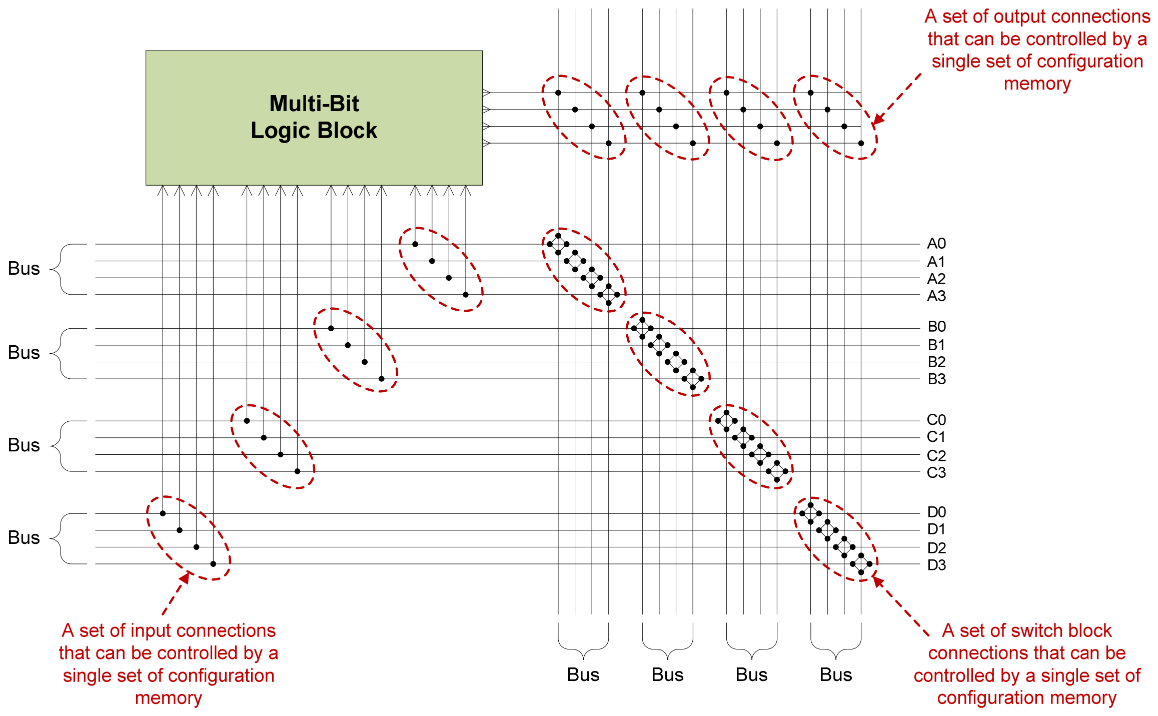

Multi-Bit Logic Block
A set of output connections that can be controlled by a single set of configuration memory
Bus
Bus
Bus
Bus
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
D0
D1
D2
D3
A set of input connections that can be controlled by a single set of configuration memory
Bus
Bus
Bus
Bus
A set of switch block connections that can be controlled by a single set of configuration memory

Figure 7-3: Configuration memory sharing in bus-based routing architecture

7.2 PROPOSED METHOD

In this chapter, we present a framework that rapidly partitions loops, which constitute the most frequently executed segments of embedded applications, into configurations and selects profitable custom instructions in the respective configurations. This enables the benefits of runtime reconfiguration to be evaluated early in the design cycle without undergoing time consuming hardware implementation. Unlike the framework in [101], our work does not generate custom instruction versions and their corresponding hardware area-time measures prior to the partitioning process. Instead, we employ the cluster generation technique that rapidly estimates the hardware area-time information of the custom instructions (as discussed in Chapter 4). In addition, the proposed framework relies on a hierarchical loop partitioning strategy that is similar to [99] for rapid partitioning of the application loops with custom instructions into one or more configurations. Unlike the work in [99], the proposed hierarchical loop partitioning strategy aims to maximize the performance gain of each configuration by increasing the utilization of each configuration and reducing the reconfiguration cost. In particular, the proposed strategy employs cluster merging (discuss in the previous chapter) to maximize the performance gain of the configurations.

Figure 7-4 shows an overview of the proposed framework. The framework relies on the Trimaran compiler infrastructure to generate the IR of the C-application in the form of a DFG. The IR serves as input to the Template Identification and Selection stage to select a set of custom instructions. The Template Identification and Selection stage have been described in previous chapters.

The hardware area-time information of the selected custom instructions is then rapidly estimated using the cluster generation technique discussed in Chapter 4. This step estimates the area costs and critical path delays of the custom instructions when they are implemented on the multi-bit logic blocks of the RFU. Cluster merging (discussed in the previous chapter) is then performed on the selected custom instructions to determine the merged clusters.

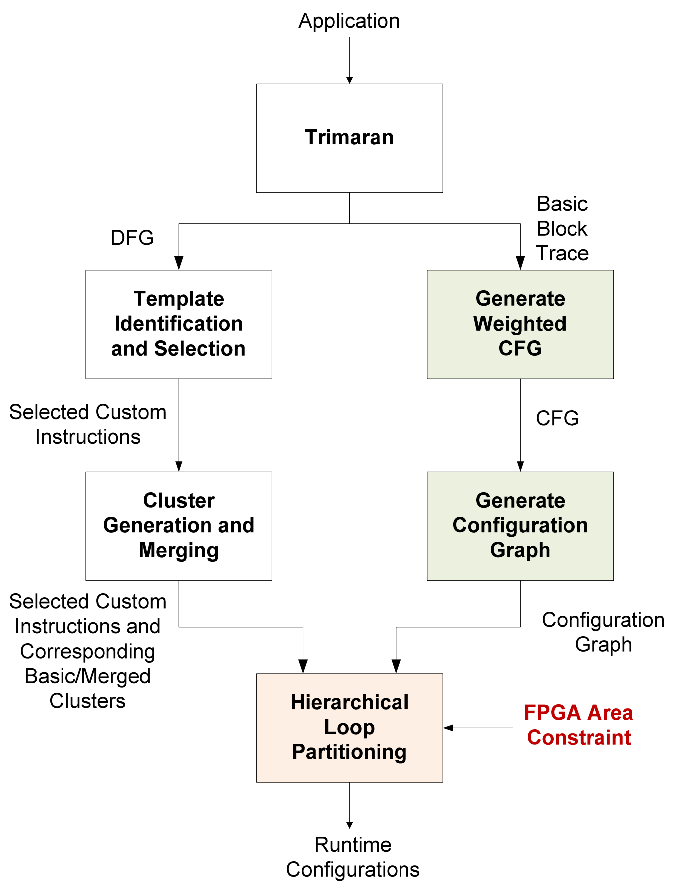

Application
Trimaran
DFG
Basic Block Trace
Template Identification and Selection
Generate Weighted CFG
Selected Custom Instructions
CFG
Cluster Generation and Merging
Generate Configuration Graph
Selected Custom Instructions and Corresponding Basic/Merged Clusters
Configuration Graph
Hierarchical Loop Partitioning
FPGA Area Constraint
Runtime Configurations

Figure 7-4: Framework for generating runtime configurations

A configuration graph is then generated to enable temporal partitioning of loops using the proposed hierarchical loop partitioning strategy. We will discuss the generation of the configuration graph in the following sub-section. Note that the partitioning strategy relies on the hardware estimation results from the cluster generation process in order to obtain a set of custom instruction configurations. In addition, the partitioning strategy also utilizes the results from cluster merging to increase the performance gain of the configurations and to reduce the reconfiguration cost.

7.2.1 Generating the Configuration Graph

Figure 7-5 shows an example of configuration graph generation from the basic block trace obtained from Trimaran's simulator. The basic block trace lists the actual execution sequence of the basic blocks for a given input dataset.

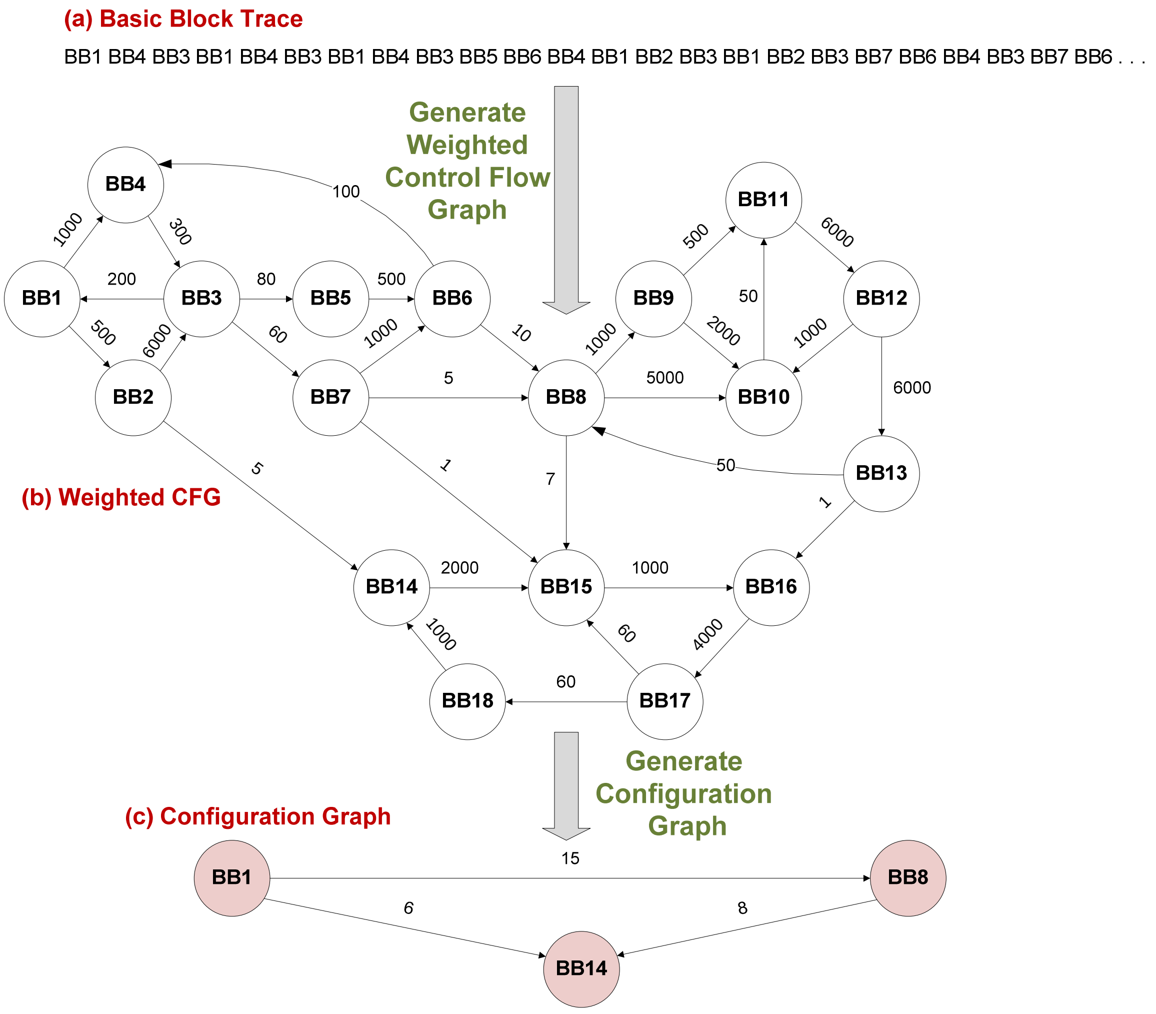


Figure 7-5: Example of configuration graph generation

We first convert the basic block trace into a weighted CFG, which encapsulates the control flow between unique basic blocks and the corresponding frequency. In particular, the weighted CFG is a directed graph *G(V,E,w)*, where *V* is a set of vertices that represent the unique basic blocks in the basic block trace. An edge $e \in E$ is an ordered pair *(u,v)*, where $u, v \in V$, that represents the control flow between basic blocks *u* and *v*. Each edge *(u,v)* is associated with a weight *w* that represents the frequency of the control flow between basic blocks *u* and *v*.

The configuration graph is a directed graph $G_c(V_c,E_c,w_c)$ that is generated from the weighted CFG. Each vertex $u_c \in V_c$ in the configuration graph, denoted as a configuration, is a set of basic blocks (i.e. $u_c = \{u_1, u_2, ..., u_k\} \in V$) that are reachable from one another. In other words, a cycle can be found between any pair of basic blocks in a configuration. In addition, there are no duplicated basic blocks in different configurations (i.e. $u_c \cap v_c = \phi$, where $u_c, v_c \in V_c$ and $u_c \neq v_c$). For example in Figure 7-5, configuration *BB1* in the configuration graph consists of basic blocks *BB1*, *BB2*, ... *BB7*, configuration *BB8* in the configuration graph consists of basic blocks *BB8*, *BB9*, ... *BB13*, and configuration *BB14* in the configuration graph consists of basic blocks *BB14*, *BB15*, ... *BB17*. It is noteworthy that the basic blocks in each configuration belong to application loops, which are the most frequently executed segments of embedded applications.

We have used the transitive closure [124] to identify the existence of cycles between each pair of basic block in the weighted CFG in order to determine their associated configurations in the configuration graph. The acyclic graph is then generated by collapsing the basic blocks into the corresponding configurations using the method described in [125]. It can also be observed that the edges of the configuration graph are associated with a weight, which is the sum of edge weights between basic blocks in different configurations. Note that weights of the edges in the configuration graph are typically very small, as these edges represent the less occurring control flow between disjoint loops in the application.

Each configuration in the initial configuration graph is a potential runtime configuration candidate. Hence, the weight of an edge in the configuration graph $w_c(u_c, v_c)$, where $u_c, v_c \in V_c$ represent the number of times that configuration u_c is reconfigured to v_c.

7.2.2 Hierarchical Loop Partitioning

The proposed hierarchical loop partitioning temporally partitions the application loops, in a top-down fashion starting from the initial acyclic configuration graph, into one or more configurations such that the overall performance gain of runtime reconfiguration is maximized. The final output of the partitioning process is a set of configurations and the selected custom instructions in each configuration.

Figure 7-6 shows an example of the proposed hierarchical loop partitioning strategy. In the initial step, the performance gain of the custom instructions in each configuration is calculated. The performance gain is computed by selecting the set of custom instructions in each configuration that leads to the highest software cycle savings while meeting the FPGA area constraint.

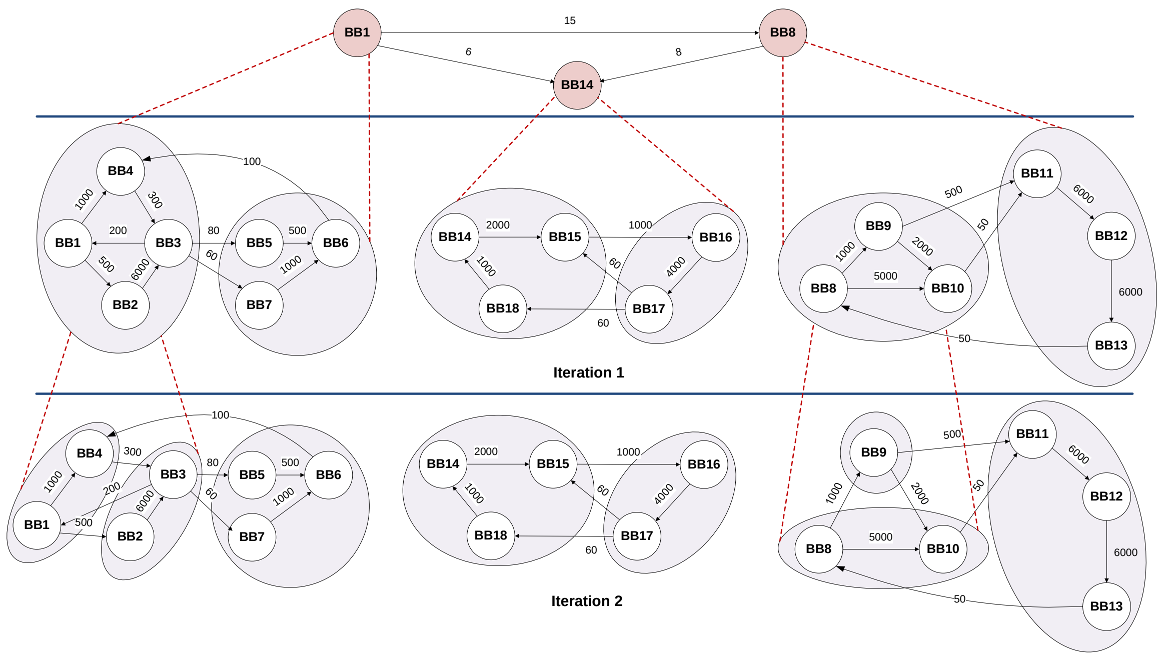


Figure 7-6: Example of hierarchical loop partitioning

In the subsequent iterations of the partitioning process, each configuration is partitioned into two new configurations. We have used the multilevel *k*-way partitioning algorithm in [126] to partition each configuration into two equal-size parts ($k = 2$) with the objective to minimize the edge-cut. The edge-cut is defined as the sum of the weight of the straddling edges between the partitions. Each new partition can be represented by a new vertex in G_c, which represents a possible runtime configuration candidate. Note that the partitioning also introduces additional edges in the configuration graph which represents the straddling edges between the basic blocks in the various partitions.

The effective performance gain for each new configuration $x \in V_c$ (in terms of software cycle savings) is computed as shown in (7-1), where G_i^x is a custom instruction in configuration x, $F(G_i^x)$ is the execution frequency of instruction G_i^x in x, $TS(G_i^x)$

denotes the number of nodes covered in instruction G_i^x using the LFF template selection method discussed in Chapter 3, and *r* is the ratio between the clock frequency of the RFU and base processor (*r* is chosen based on the area-optimized configuration of the soft-core processor in [115]). $T_{Template}(G_i^x)$, which is the estimated critical path delay of G_i^x is assumed to be the number of basic/merged clusters in the critical path of G_i^x. T_{RTR}^x is the reconfiguration cost of *x*. The area utilization of the custom instructions G_i^x in *x* cannot exceed the FPGA area constraint A_{FPGA} (in terms of number of logic blocks) as shown in (7-2). In our work, G_i^x is selected from the set of custom instructions in configuration *x* that leads to the highest software cycle savings while meeting the FPGA area constraint.

$$SCS(x) = \sum_i^c F\left(G_i^x\right) \cdot \left(TS\left(G_i^x\right) - r \cdot T_{Template}\left(G_i^x\right)\right) - T_{RTR}^x \qquad (7\text{-}1)$$

$$A(x) = \sum_i^c A\left(G_i^x\right) \le A_{FPGA} \qquad (7\text{-}2)$$

The reconfiguration cost of configuration *x* is computed differently for full reconfiguration and partial reconfiguration as shown in (7-3). $\sum w_c(u_c, x)$ is the sum of weights of the incoming edges of *x* in the configuration graph. In other words, $\sum w_c(u_c, x)$ represents the number of times configuration *x* will be reconfigured on the FPGA at runtime. T_{RTR}^{lb} is the reconfiguration cost of a single multi-bit logic block as discussed in Section 7.1, and is measured in terms of software clock cycles. Finally, n_{common} is the number of common basic/merged clusters in configuration *x* and the previous configuration u_c, i.e. $(u_c, x) \in E_c$. For partial reconfiguration, we can avoid reconfiguring logic blocks with common basic/merged clusters in two consecutive configurations.

$$T_{RTR}^{x} = \begin{cases} \sum w_c(u_c, x) \times T_{RTR}^{lb} \times A_{FPGA} & \text{if full RTR,} \\ \sum w_c(u_c, x) \times T_{RTR}^{lb} \times \left(A_{FPGA} - n_{common}\right) & \text{if partial RTR.} \end{cases} \quad (7\text{-}3)$$

For each partition solution, the total performance gain of the resulting partitions is compared to the performance gain of the initial configuration. If the post-partition performance is less than the initial performance, then the new partitions are discarded and the initial configuration is restored. This can be observed in Iteration 2 of Figure 7-6, where some of the configurations in Iteration 1 do not lead to any further partitions. The partition process is repeated until no new partitions are formed in a particular iteration. The final set of partitions is the runtime configurations. Note that this hierarchical partitioning strategy reduces the search space by avoiding further partitioning of a particular configuration if the resulting partitions do not lead to higher performance.

Figure 7-7 shows the pseudo code for the proposed hierarchical loop partitioning strategy. It can be observed that in each iteration (lines 4-17), the performance gain of each existing configuration in the configuration graph G_c is first evaluated (line 5) using the function CALCULATE-PERFORMANCE-GAIN and temporarily removed from G_c (line 6). The existing configuration is then partitioned into two smaller configurations using the K-WAY-PARTITION function (line 7) and the new configurations are inserted into G_c along with the corresponding edges (line 8). The performance gain of the two new configurations is evaluated (lines 9-10) and compared to the performance gain of the initial configuration (line 11). In the event that the partitioning has led to less favorable performance gain, the initial partition is restored (line 12) in the configuration graph and the new configurations are removed from the configuration graph (line 13). When no new partitions are generated in an iteration (evaluated in line 2), the algorithm returns the configuration graph consisting of the final set of configurations (line 18).

Algorithm 7-1

```
PARTITION-LOOP ($G_c$, $A_{FPGA}$)
 1. partition_exist := true
 2. while (partition_exist = true)
 3.     partition_exist := false
 4.     for each node $u_c \in G_c$
 5.         $SCS(u_c)$ = CALCULATE-PERFORMANCE-GAIN ($u_c$, $A_{FPGA}$)
 6.         remove $u_c$ from $G_c$
 7.         $u_c^1, u_c^2$ = K-WAY-PARTITION ($u_c$)
 8.         insert $u_c^1$ and $u_c^2$ in $G_c$
 9.         $SCS(u_c^1)$ = CALCULATE-PERFORMANCE-GAIN ($u_c^1$, $A_{FPGA}$)
10.         $SCS(u_c^2)$ = CALCULATE-PERFORMANCE-GAIN ($u_c^2$, $A_{FPGA}$)
11.         if $SCS(u_c^1) + SCS(u_c^2) < SCS(u_c)$
12.             restore $u_c$ in $G_c$
13.             remove $u_c^1$ and $u_c^2$ from $G_c$
14.         else partition_exist := true
15.         end
16.     end
17. end
18. return $G_c$
```

Figure 7-7: Proposed algorithm for hierarchical loop partitioning

7.3 EXPERIMENTAL RESULTS

Runtime reconfiguration on RISPs is only feasible for applications where the performance of the custom instructions can mitigate the high reconfiguration overhead of the FPGA architecture. Our investigations reveal that the Cjpeg application offers good potential for exploiting runtime reconfiguration for realizing high performance custom instructions in RISPs with stringent area constraints. We have evaluated the proposed hierarchical loop partitioning strategy for both full reconfiguration and partial reconfiguration models (see Appendix A). In addition, we investigated the impact of cluster merging on increasing the performance gain of runtime reconfigurable RISPs. The experiments in this section are based on the target RFU model described in Section 7.1.

7.3.1 Full Reconfiguration Model

The full reconfiguration model requires the complete reprogramming of the entire configuration memory during runtime reconfiguration. Figure 7-8 shows the total runtime reconfiguration cost (primary axis), which is calculated based on (7-3), and the number of configurations (secondary axis) for the full reconfiguration model. These values are obtained for varying FPGA area constraints (in terms of number of logic blocks), i.e. 2% to 20% of the maximum FPGA area that is required to implement all the selected custom instructions.

It can be observed that in general (except for cases where the area constraint is less than 6%), the runtime reconfiguration cost where cluster merging is not considered during hierarchical loop partitioning (denoted as *No Cluster Merging*), is lower than the case where cluster merging is considered during hierarchical loop partitioning (denoted as *Cluster Merging)*. This is due to the fact that when cluster merging is taken into account during hierarchical loop partitioning, more configurations are generated as shown in Figure 7-8. Hence, the number of times the FPGA undergoes reconfiguration at runtime also increases for *Cluster Merging*. The gradual increase in reconfiguration cost for area constraint larger than 5% is due to the increase in the number of logic blocks that undergo runtime reconfiguration for the full reconfiguration model.

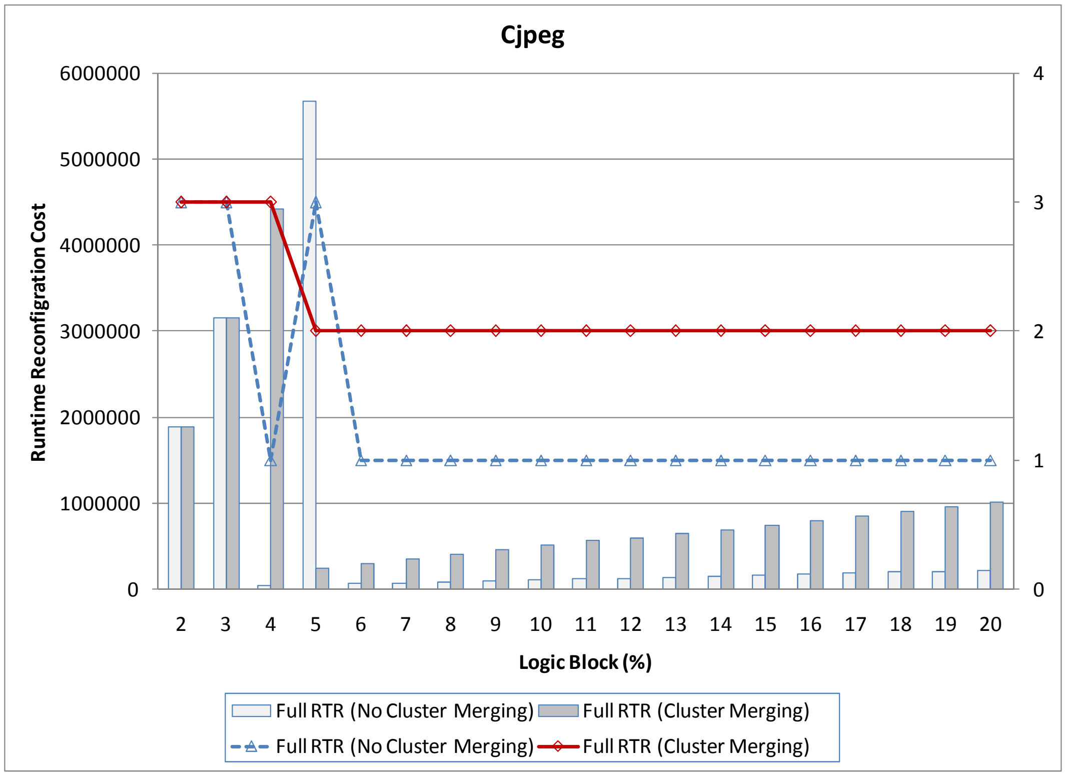


Figure 7-8: Runtime reconfiguration cost and number of partitions for full reconfiguration model

Figure 7-9 compares the performance between *No Cluster Merging* and *Cluster Merging* for the full reconfiguration model. The performance is calculated by summing up the software cycle savings of all the configurations (calculated based on (7-1)). In addition, the performance without runtime reconfiguration (*No RTR*) is also shown in Figure 7-9. In order to obtain the performance of *No RTR*, a greedy algorithm is used to select a set of custom instructions that lead to the highest performance while meeting the area constraint. Hierarchical loop partitioning is not employed for *No RTR*.

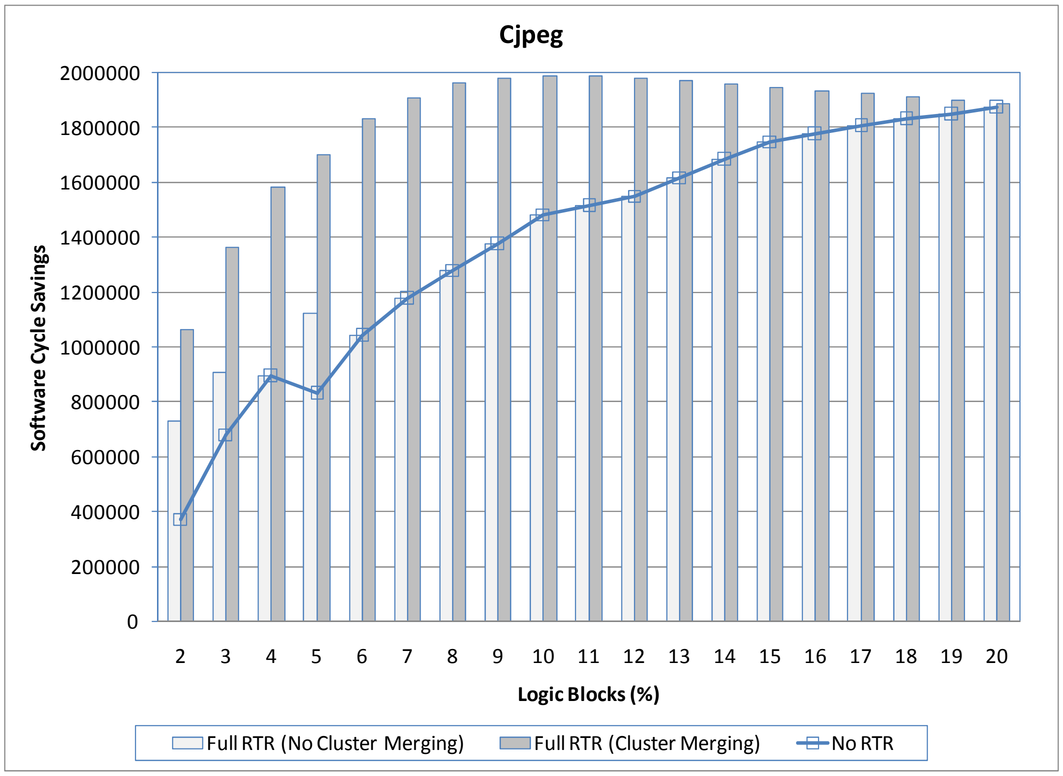


Figure 7-9: Performance gain for full reconfiguration model

It can be observed that *No Cluster Merging* outperforms *No RTR* only for a few cases when the area constraint is less than 6%. Thereafter, there is no significant difference between the performance of *No Cluster Merging* and *No RTR*. On the other hand, *Cluster Merging* outperforms both *No RTR* and *No Cluster Merging* for almost all the cases (except for the case where the area constraint is 20%). In particular, on average, *Cluster Merging* outperforms *No RTR* and *No Cluster Merging* by 45.6% and 32.9% respectively. It is noteworthy that *Cluster Merging* can outperform *No Cluster Merging* by over 77% (i.e. for an area constraint of 4%). In addition, *Cluster Merging* outperforms *No RTR* by two times or more for area constraints 2%, 3% and 5%. The performance gain of *Cluster Merging* over the *No RTR* and *No Cluster Merging* gradually decreases when the area constraint is more relaxed due to the increase in reconfiguration cost as shown in Figure 7-8**.**

These results show that cluster merging can effectively increase the utilization of the configurations, which has led to the generation of a larger number of configurations. This in turn has resulted in higher performance benefits for the full reconfiguration model.

7.3.2 Partial Reconfiguration Model

Partial reconfiguration enables a portion of the configuration memory to be programmed during runtime reconfiguration and hence this can lead to higher savings in the runtime reconfiguration cost. Figure 7-10 shows the total runtime reconfiguration cost (primary axis), which is calculated based on (7-3), and the number of configurations (secondary axis) for the partial reconfiguration model. The range of area constraint is the same as the previous sub-section.

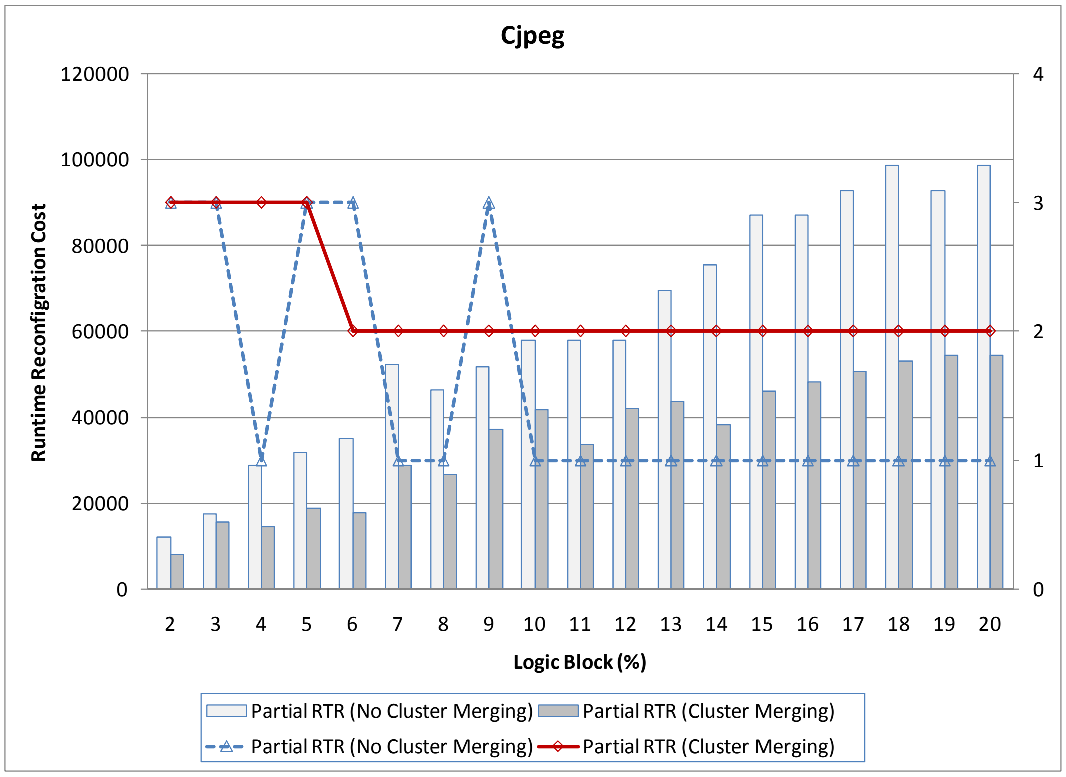


Figure 7-10: Runtime reconfiguration cost and number of partitions for partial reconfiguration model

It can be observed that similar to the full reconfiguration method, the number of configurations obtained using the proposed method without considering cluster merging (denoted as *No Cluster Merging)* is generally lower than the number of configurations obtained using the proposed method that takes into account cluster merging (denoted as *Cluster Merging)*. However, unlike the full reconfiguration model, the runtime reconfiguration cost of *Cluster Merging* is lower than *No Cluster Merging* for all the area constraints considered. On average, *Cluster Merging* has 39.6% lesser reconfiguration cost compared to *No Cluster Merging*. The maximum percentage of reduction in configuration cost is 49.4% when the area constraint is 6% of the maximum FPGA area. These results imply that cluster merging is capable of reducing the runtime reconfiguration cost due to the increase in common basic/merged clusters in consecutive configurations.

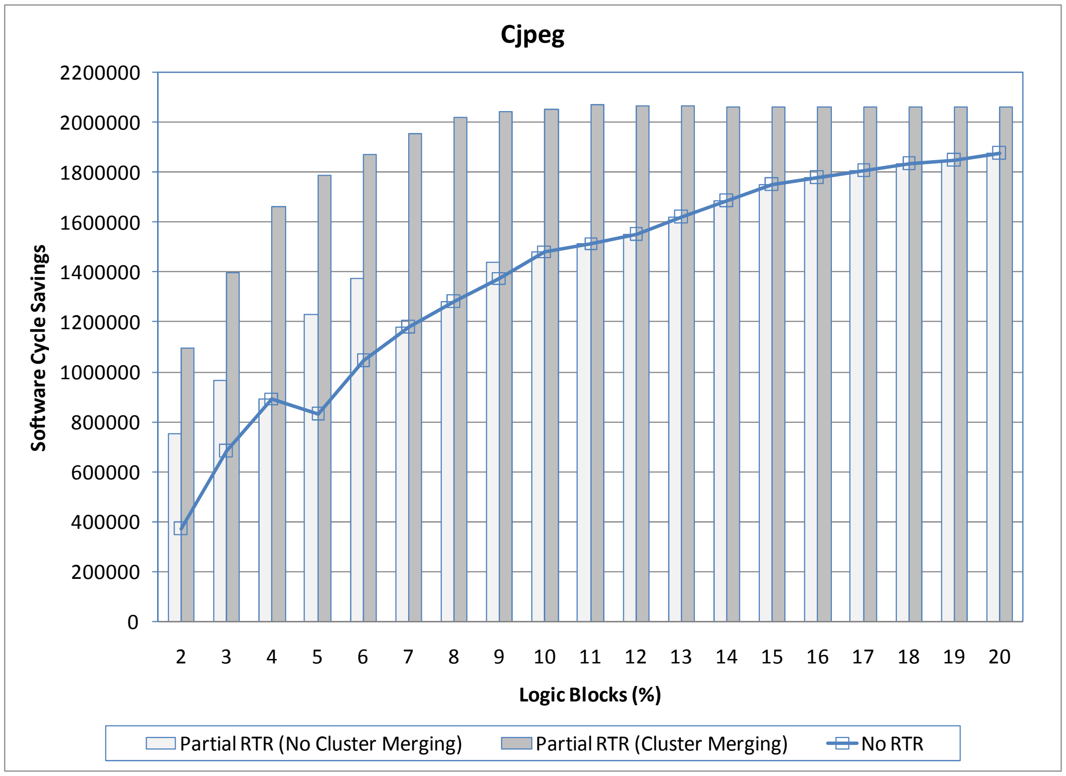


Figure 7-11: Performance gain for full reconfiguration model

Figure 7-11 compares the performance between *No Cluster Merging* and *Cluster Merging* for the partial reconfiguration model. The performance of *No RTR* is also shown. Firstly, it can be observed that the partial reconfiguration model leads to higher performance than the full reconfiguration model. For example, unlike the full reconfiguration model, *Cluster Merging* in the partial reconfiguration model still outperforms *No RTR* when the area constraint is 20% of the maximum FPGA area. However *No Cluster Merging* still outperforms *No RTR* for only a few cases when the area constraint is less than 6%. This shows the significance of cluster merging for increasing the performance of runtime reconfiguration for the partial reconfiguration model. It is also evident that *Cluster Merging* outperforms both *No RTR* and *No Cluster Merging* for all cases. In particular, on average, *Cluster Merging* outperforms *No RTR* and *No Cluster Merging* by 52.2% and 34.9% respectively. In addition, *Cluster Merging* can outperform *No Cluster Merging* by over 86% (i.e. for area constraint of 4%). Similar to the full reconfiguration model, *Cluster Merging* outperforms *No RTR* by two times or more for area constraints 2%, 3% and 5%. Specifically, a maximum performance gain of up to 2.94 times can be observed when the area constraint is 2% of the maximum FPGA area.

These results show that cluster merging can lead to higher performance benefits for the partial reconfiguration model in two ways: 1) increasing the utilization of the configurations, and 2) reducing the runtime reconfiguration cost.

7.4 SUMMARY

We have presented a framework for RISPs that employs a hierarchical partitioning strategy which aims to maximize the performance of custom instruction realization through runtime reconfiguration, while minimizing the reconfiguration overhead. The proposed framework targets area-constrained RISPs that incorporates RFU with multi-bit logic blocks and bus-based architecture. We have introduced a RFU model that supports a high degree of configuration memory sharing in order to reduce the runtime reconfiguration overhead. It is noteworthy that the proposed RFU incorporates BLEs that are similar to those found in commercial FPGAs. The proposed partitioning strategy relies on the cluster generation technique to rapidly estimate the hardware area-time information of the custom instructions. It also employs cluster merging to increase the utilization of each configuration by enabling a larger number of profitable custom instructions to be implemented in each configuration. Experimental results show that this also leads to the generation of more configurations containing profitable custom instructions. In addition, hierarchical loop partitioning with cluster merging can achieve significant reduction in the reconfiguration cost for the partial reconfiguration model. This is due to the fact that cluster merging results in a larger number of common basic/merged cluster realizations in consecutive runtime configurations. Experiment results show that both the full and partial reconfiguration models can benefit notably from the proposed cluster merging based hierarchical loop partitioning strategy.

In the next chapter, we will present an integrated framework that generates profitable custom instructions for RISPs with runtime reconfiguration support. The proposed framework integrates all the optimization strategies that have been presented in Chapters 3 to 7.

CHAPTER 8

FRAMEWORK FOR GENERATING PROFITABLE CUSTOM INSTRUCTIONS FOR RUNTIME RISP

In the previous chapters, we have presented strategies to select area-time efficient custom instructions by employing: 1) suitable heuristics for selecting custom instructions that can lead to high performance (Chapter 3), and 2) rapid design exploration, which relies on the proposed cluster generation based high-level estimation, to select a reduced set of profitable custom instructions with negligible loss in performance gain (Chapter 4 and 5). We have also presented the cluster merging technique for mapping the selected custom instructions efficiently onto the FPGA architecture (Chapter 6). Finally, we demonstrated that the proposed cluster merging technique can significantly increase the performance of RISPs with runtime reconfiguration support, by maximizing utilization of the runtime configurations and reducing the reconfiguration overhead (Chapter 7).

In this chapter, we will integrate all the components discussed in the previous chapters into a complete framework that generates FPGA-aware custom instructions for runtime reconfiguration of RISPs. We will also propose a scheme for managing the runtime reconfiguration of custom instructions on the target architecture. The proposed scheme aims to minimize the runtime reconfiguration overhead by judiciously replacing the configurations of the multi-bit logic blocks based on the dynamic execution profile of the application. Our preliminary results, which were published in [J-4] and [C-4], show that the proposed replacement technique outperforms well-known heuristics for memory-page replacement (which have also been used for runtime reconfiguration management e.g. [127]) in RFUs with small hardware area.

8.1 PROPOSED FRAMEWORK

The proposed framework is shown in Figure 8-1, which consists of the following main components: 1) compiler infrastructure, 2) template identification, 3) selection of most profitable custom instructions, 4) cluster generation and merging, and 5) loop-aware grouping of merged custom instructions.

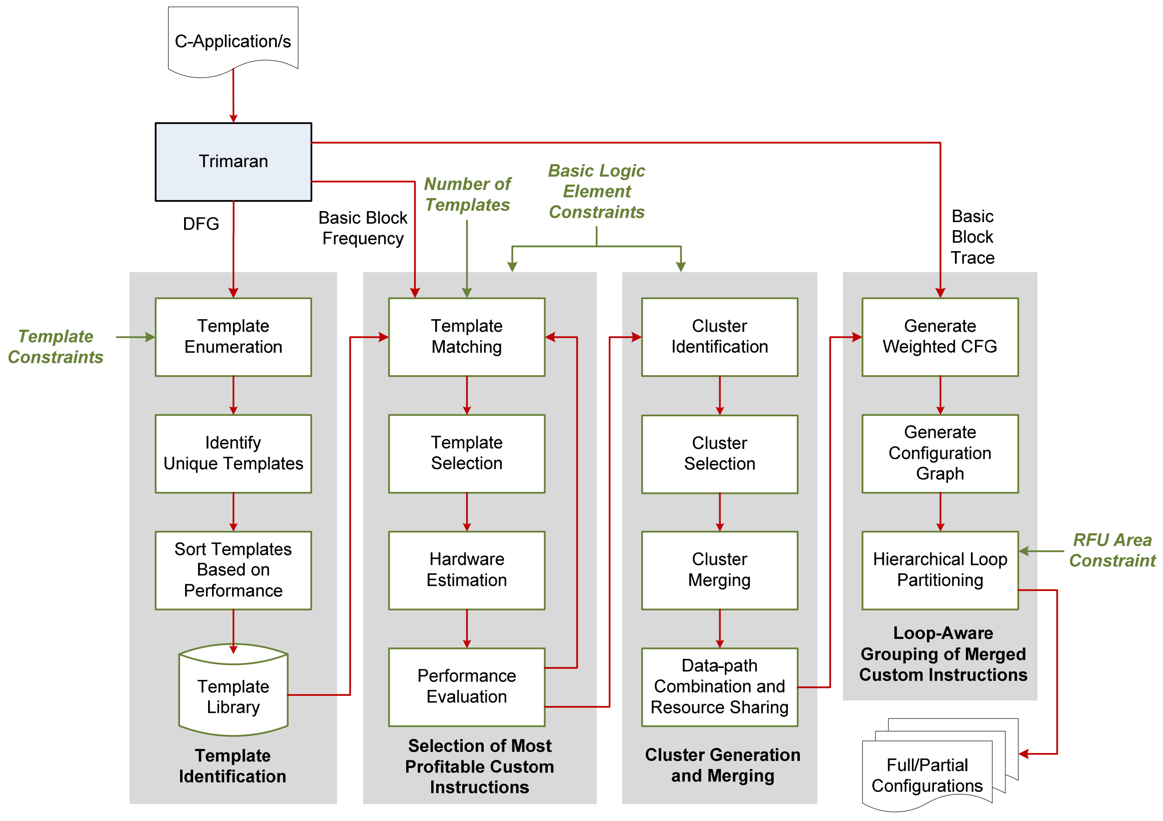

C-Application/s
Trimaran
DFG
Basic Block Frequency
Number of Templates
Basic Logic Element Constraints
Basic Block Trace
Template Constraints
Template Enumeration
Identify Unique Templates
Sort Templates Based on Performance
Template Library
Template Identification
Template Matching
Template Selection
Hardware Estimation
Performance Evaluation
Selection of Most Profitable Custom Instructions
Cluster Identification
Cluster Selection
Cluster Merging
Data-path Combination and Resource Sharing
Cluster Generation and Merging
Generate Weighted CFG
Generate Configuration Graph
Hierarchical Loop Partitioning
RFU Area Constraint
Loop-Aware Grouping of Merged Custom Instructions
Full/Partial Configurations

Figure 8-1: Proposed framework

The input to the framework is a C-application. It is noteworthy that the framework can also process multiple applications at a time, e.g. applications in the same domain (see [J-5]). The output of the framework is a set of full/partial configurations. We will briefly discuss each of these components in the following subsections. Detailed descriptions of the components are found in Chapters 3 to 7.

8.1.1 Compiler Infrastructure

In the current framework, we have employed the Trimaran compiler infrastructure [103] mainly due to the following features:

1. Supports a wide range of commonly used benchmark suites.
2. Enables the user to choose the appropriate front-end and back-end optimizations for instruction set customization. For example, register allocation is disabled in order to generate an IR without false dependencies for custom instruction identification and selection.
3. Generates DFGs of the basic blocks in both graphical and textual format that can be conveniently used to identify and select custom instructions.
4. Performs basic block profiling, which facilitates template selection and performance evaluation.
5. Generates a basic block trace that can be used to construct the configuration graph for temporal partitioning in order to identify runtime configurations.

It is noteworthy that the remaining stages of the proposed framework are not restricted to the Trimaran compiler infrastructure. Only portions pertaining to the translation of the input data format to the internal data structures used in the framework need to be modified if a new compiler infrastructure (which possesses the above features) is used.

8.1.2 Template Identification

Although this stage is not the main contribution of our work, it is a necessary and essential step in the framework. As discussed in Section 3.3.1, we have adopted the template enumeration algorithm in [70] to identify all the template instances from the DFG generated by Trimaran. A set of template constraints based on the characteristics of the core processor, cost considerations and compiler limitations (see Appendix B.3) must be provided to the template identification algorithm. This stage can also adopt more recently reported template enumeration algorithms that can achieve equivalent results in a fraction of the time that is required by the method in [70].

The template instances are subjected to template grouping, whereby identical template instances that occur in different basic blocks are grouped to create a unique set of unique templates. Template instances are considered identical if they are isomorphic when the input and output operands are not considered. We have used the graph isomorphism method in the vflib graph-matching library [114] to identify the unique templates. The templates are then sorted in decreasing gain, which is calculated based on equation (5-1), and stored in the template library.

8.1.3 Selection of Most Profitable Custom Instructions

The aim of this stage is to select a reduced set of custom instructions that leads to high area efficiency without compromising heavily the performance gain (see Chapter 5). This is achieved by iteratively choosing a varying range of templates (each range includes templates with highest gain) for template matching and selection. In the first iteration, the full range of templates is used for template matching and selection. As described in Section 5.2.1, we have employed the vflib graph-matching library [114] to identify all the matches in the DFG for the templates. In each iteration, the conflict graph (see Section 3.1) is constructed based on the template matches. The LFF based template selection method is then employed to select a set of non-overlapping template matches that leads to the highest performance (see Chapter 4).

In order to evaluate the performance of the selected custom instructions, we employed the cluster generation technique (see Chapter 5) to estimate the critical path delays of the custom instructions when they are realized with the BLEs of the target RFU. We have used equation (4-7), which takes into account the software latency, estimated critical path delay and frequency of the custom instructions, to calculate the performance gain of a custom processor (that incorporates the selected custom instructions) over the baseline processor. The range of templates used in template matching and selection is progressively reduced in subsequent iterations until the estimated performance gain due to the selected custom instructions is notably less than the previous iteration. The selected custom instructions in the previous iteration are finally chosen as the set of profitable custom instructions that can lead to area-time efficient realizations.

8.1.4 Cluster Generation and Merging

In this stage, the reduced set of custom instructions selected in the previous stage undergoes further optimization so that they can maximize the utilization of the FPGA logic blocks. The cluster generation process (presented in Chapter 4), which consists of the cluster identification and selection stage, is first employed to partition the reduced set of custom instructions into basic clusters, where each basic cluster can be efficiently mapped onto the LUT and carry-look-ahead structure of the target FPGA logic block. Note that cluster generation is also used in the previous stage to estimate the critical path delays of the custom instruction for performance evaluation. Cluster merging (discussed in Chapter 6) then merges the basic clusters in order to maximize the utilization of the FPGA blocks. Resource sharing is performed on the clusters only if the resulting data-paths do not lead to an increase in the area-delay product. We have shown in Chapter 6 that the proposed cluster merging method can achieve significantly lower area-delay product for all cases considered when compared to the area optimization capabilities of the commercial tool and to one of the most efficient methods reported in the literature for resource sharing.

8.1.5 Loop-Aware Grouping of Merged Custom Instructions

This final stage of the framework rapidly identifies a suitable set of configurations in order to evaluate the benefits of runtime reconfiguration early in the design cycle. A configuration graph is first generated to enable temporal partitioning of loops in the application. As described in Section 7.2.1, this is achieved by constructing a weighted

CFG from the basic block trace that is produced by the compiler infrastructure. The weighted CFG encapsulates the control flow and execution frequency between unique basic blocks. The configuration graph is then generated from the weighted CFG, where each node in the configuration graph consists of frequently executed loops in the application. The proposed hierarchical loop partitioning strategy is then employed to partition the application loops in the configuration graph into one or more configurations such that the overall performance gain of runtime reconfiguration is maximized. We have shown in Chapter 7 that hierarchical loop partitioning, which relies on the cluster merging results, can maximize the utilization of the area-constrained reconfigurable logic blocks in each configuration and reduce the runtime reconfiguration overhead.

8.2 HIGH LEVEL FLOW OF RUNTIME RISP

Figure 8-2(a) shows an example of a RISP with partial reconfiguration support. It can be observed that the RISP consist of a baseline VLIW processor that is augmented with the following components: 1) bit-stream memory, 2) reconfiguration manager, 3) configuration interface, and 4) RFU (with multi-bit logic blocks). Apart from the components for partial reconfiguration support, the RISP in Figure 8-2 is similar to the target RISP described in Figure 4-1.

The bit-stream memory stores all the full/partial bit-streams that are required to reconfigure the RFU at runtime. These bit-streams contain the required information to configure the FPGA configurable resources in order to implement the required custom instructions. Existing tools and techniques e.g. [127]-[131] can be employed by the RFU-

Aware Bit-Stream Generation step (Figure 8-2(b)) to automate the generation of bit-streams based on the runtime configurations obtained from the proposed framework.

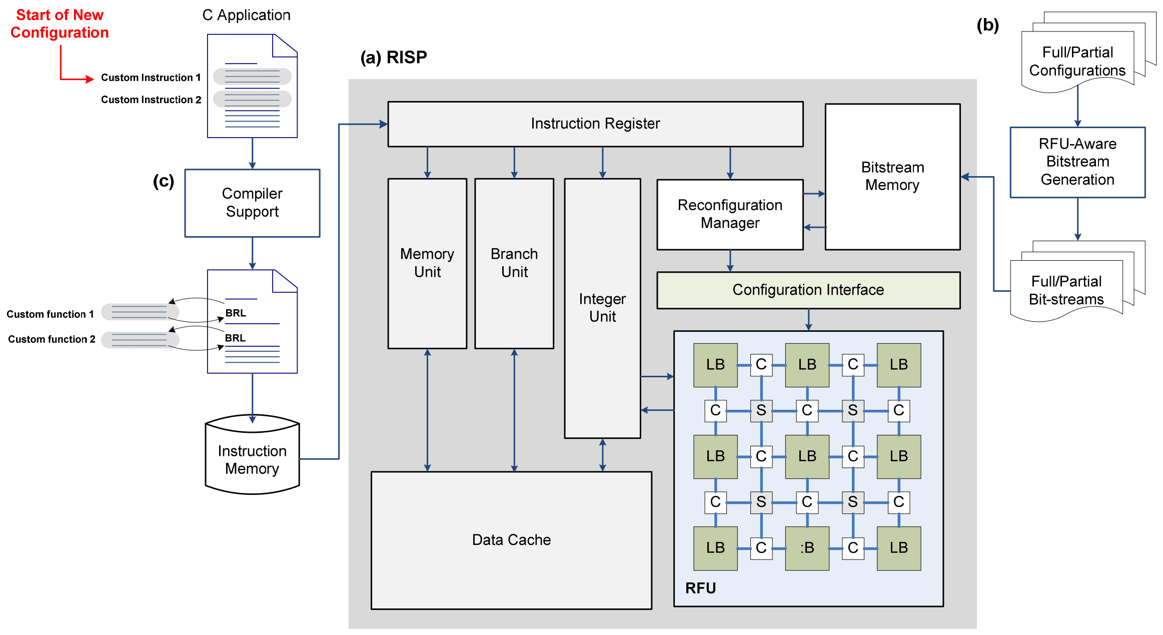


Figure 8-2: High level flow of a RISP with runtime reconfiguration support

The reconfiguration manager is responsible for loading configurations from the configuration memory to execute on the RFU through the configuration interface. It detects reconfiguration events from the Instruction Register and fetches the required bit-stream from the configuration memory to be loaded into the appropriate locations of the RFU. A possible implementation of the reconfiguration manager can be found in [132]. As discussed in [133], the reconfiguration manager, bit-stream memory and configuration interface can reside on the FPGA fabric to facilitate rapid runtime reconfiguration through self-reconfiguration. The reconfiguration manager also incorporates a scheme that relies on the dynamic execution profile to reconfigure the multi-bit logic blocks with the goal of minimizing the overall reconfiguration overhead. We will introduce this scheme in the next section.

The high-level process of the runtime RISP is similar to that presented in [134]. Initially, the compiler support for custom instruction extension extracts the operations in the input application that correspond to the selected custom instructions and places them in a separate function body as shown in Figure 8-2(c). These portions in the application code are replaced with an instruction (e.g. BRL (Branch and Link)) that serves as a function call to these new custom functions.

At runtime, the custom instructions in a loop that correspond to a new configuration are initially executed on the base processor in the form of custom functions. During the execution of the custom functions on the base processor, the reconfiguration manager detects a reconfiguration event based on the address of the first custom function in the new configuration. Based on the scheme discussed in the following section, the reconfiguration manager then determines the suitable configuration for the current set of custom instructions and reconfigures the RFU accordingly. This approach of overlapping the initial execution of the operations corresponding to the custom instructions on the baseline processor with the reconfiguration of the RFU prevents pipeline stalls due to runtime reconfiguration. On the subsequent executions of the same loop, the custom instructions are executed on the RFU whenever a function call to the custom functions is detected. This continues until the application exits the loop in the current configuration. The entire process described above is repeated for other parts of the application, and a new configuration will be loaded onto the RFU when the reconfiguration manager encounters a new reconfiguration event.

8.3 RUNTIME MANAGEMENT OF CUSTOM INSTRUCTIONS

Figure 8-3 describes the proposed scheme for runtime reconfiguration management. The proposed scheme relies on two tables that store the dynamic activity information: 1) Cluster Frequency Table (CFT) and 2) Logic Block Activity Table (LBAT).

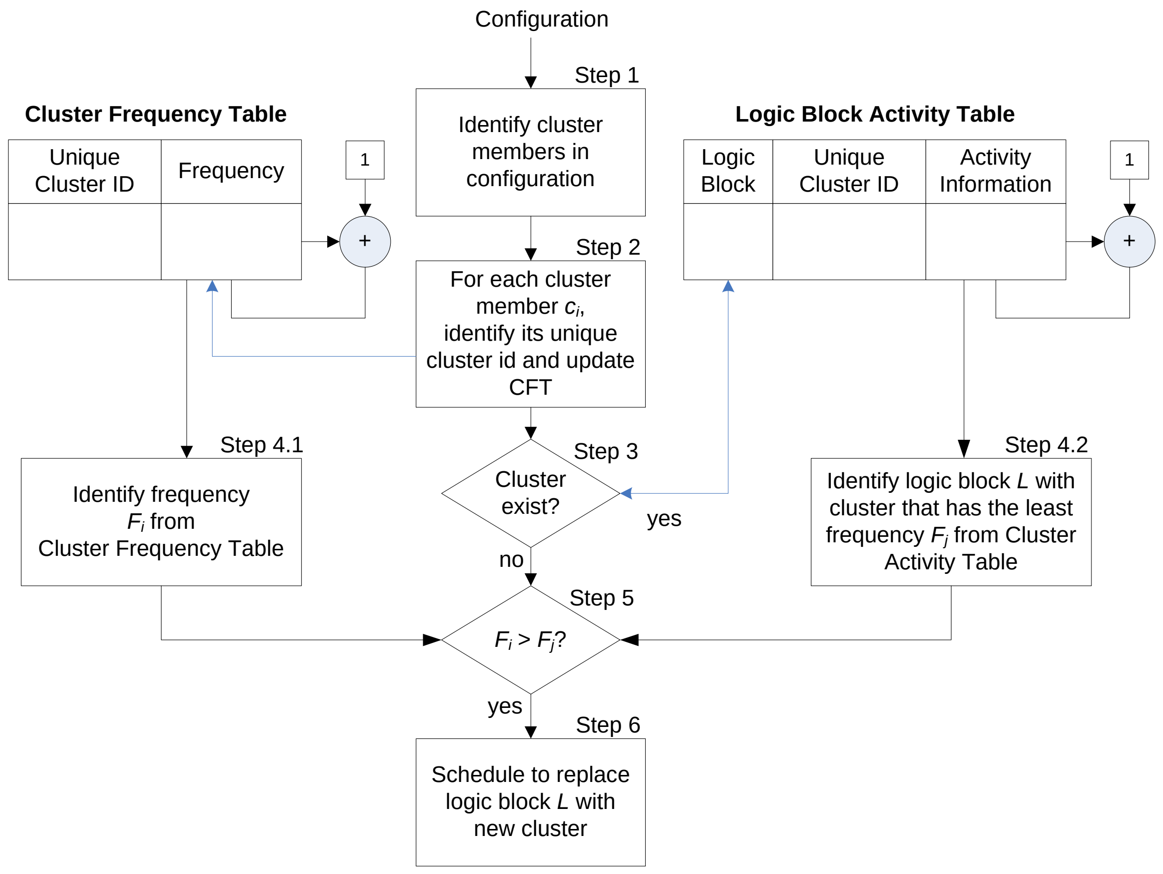

Configuration
Step 1
Identify cluster members in configuration
Cluster Frequency Table
Unique Cluster ID
Frequency
1
+
Logic Block Activity Table
Logic Block
Unique Cluster ID
Activity Information
1
+
Step 2
For each cluster member c_i, identify its unique cluster id and update CFT
Step 4.1
Identify frequency F_i from Cluster Frequency Table
Step 3
Cluster exist?
yes
no
Step 4.2
Identify logic block L with cluster that has the least frequency F_j from Cluster Activity Table
Step 5
$F_i > F_j$?
yes
Step 6
Schedule to replace logic block L with new cluster

Figure 8-3: Proposed scheme for runtime reconfiguration management

The CFT records are the dynamic frequency of the unique clusters (merged/basic clusters) that have been identified so far, while the LBAT records are the information of the current configurations (i.e. the unique cluster and its corresponding dynamic frequency) in the respective logic blocks. The depth of CFT is governed by the number of unique clusters that are found in the application, while the depth of LBAT is dependent on the number of multi-bit logic blocks in the target architecture.

When a new configuration is considered for runtime reconfiguration on the target architecture, the cluster members of the configuration are first identified (Step 1). For each cluster member c_i, the corresponding unique cluster pattern entry in CFT is identified and the dynamic execution frequency of the unique cluster is incremented in CFT (Step 2). The LBAT is then checked to determine if any of the logic blocks have already been configured to implement c_i (or the unique cluster associated with c_i) (Step 3). If the cluster configuration of c_i already exists in one the logic blocks, the activity information (which stores the frequency of the cluster) in the corresponding entry in LBAT is also incremented. However, if none of the logic blocks is configured with c_i, the frequency F_i of the unique cluster associated with c_i is accessed from CFT (Step 4.1), and the logic block L with the lowest cluster frequency F_j is identified from LBAT (Step 4.2). If $F_i > F_j$ (i.e. the dynamic execution frequency of the unique cluster associated with c_i is larger than the least frequently occurring cluster c_j in the logic blocks), logic block L is scheduled to replace the configuration of c_j with c_i (Step 5 and 6). This process is repeated for all the cluster members that are associated with the configuration.

Runtime reconfiguration is performed by writing the configurations of the clusters (that are scheduled to be reconfigured onto the logic blocks) from the bit-stream memory to the configuration memories in the logic blocks. Note that runtime reconfiguration is undertaken only if the resulting configuration of the logic blocks incorporates all the cluster members of the custom instruction. When a logic block is reconfigured, the corresponding entries in LBAT must be updated.

Table 8-1: Cluster statistics

Application	Selected Templates	Basic Clusters	Unique Clusters (Before Cluster Merging	Unique Clusters (After Cluster Merging)
Adpcm Dec	6	8	6	5
Adpcm Enc	7	11	9	7
Aes	32	65	31	19
Basicmath Large	2	2	2	2
Bitcount	4	9	9	7
Blowfish Dec	8	15	9	6
Blowfish Enc	8	15	9	6
Cjpeg	49	99	32	21
CRC32	1	1	1	1
Dijkstra Large	7	16	9	4
FFT	6	8	7	4
Patricia	5	8	7	3
Pegwit	18	43	18	12
Rijndael Dec	12	17	13	8
Rijndael Enc	12	24	16	11
Sha	8	17	11	7

The replacement strategy that is used to determine which logic block to be reconfigured tries to maximize the probability of reusing the clusters in the future by maintaining clusters with the highest dynamic execution frequency in the logic blocks. Hence, this strategy can lead to reconfiguration overhead reduction if the most profitable custom instructions in the application have common cluster patterns. Table 8-1 reports the cluster statistics from sixteen applications. The second column lists the number of selected custom instructions obtained using the LFF approach, the third column lists the number of basic clusters that are obtained using the cluster generation technique, the forth column lists the number of unique basic clusters, and the final column reports the number of unique basic/merged clusters after the cluster merging step. The unique clusters in the fourth and fifth column of Table 8-1 is the set of non-isomorphic clusters before and after cluster merging respectively. It can be observed that on average over 28% of the basic clusters in each application are isomorphic. The number of isomorphic clusters can be further increased to over 49% on average through cluster merging. The notable number of isomorphic clusters found in different custom instructions provides a strong justification for adopting the cluster-based runtime reconfiguration approach to reduce the reconfiguration overhead.

8.4 SUMMARY

We have presented an integrated framework to generate runtime configurations of FPGA-aware custom instruction for RISPs. The proposed framework incorporates a systematic flow consisting of multiple optimization strategies to produce a set of runtime configurations that can be implemented on the RFU in an area-time efficient manner. These optimizations include a heuristic based strategy to select custom instructions with high performance, a rapid design exploration approach for selecting most profitable custom instructions, architecture-aware methods for maximizing the utilization of the FPGA logic blocks for implementing the custom instructions, and a temporal partitioning strategy to generate runtime configurations with high performance and low reconfiguration overhead.

A scheme for managing the runtime reconfiguration of custom instructions on a partially reconfigurable architecture that incorporates multi-bit logic blocks has also been presented. The proposed scheme incorporates a replacement strategy that aims to reduce the runtime reconfiguration overhead by preserving custom instruction clusters with the highest dynamic execution frequency in the multi-bit logic blocks. The feasibility of this scheme is supported by our analysis that shows that a notable number of isomorphic clusters can be found in the applications considered.

CHAPTER 9

CONCLUSION AND RECOMMENDATIONS

9.1 CONCLUSION

In this thesis, novel techniques and design methodologies have been proposed for the automatic generation of custom instructions for FPGA based RISPs. In particular, the emphasis has been to maximize the utilization of available reconfigurable resource, thereby strengthening the chances of incorporating RISPs in commercially viable embedded products that must also cater for TTM and NRE pressures.

The method proposed for custom instruction selection relies on a graph based heuristic to locate high performance templates consisting of primitive operations. The Largest-Fit-First (LFF) approach, which selects non-overlapping superset templates, is motivated by our findings that over 77% of high frequency basic templates are incorporated in larger templates. Noting that LFF may omit some of the profitable templates that overlap with the superset templates identified by LFF, we have further refined LFF by selectively reintroducing high performance overlapping templates that were previously ignored. Comparisons with previously reported approximate strategies such as MFF, MLFF and ILP show that the custom instructions selected using the proposed techniques can provide for performance increase of up to 32% for certain benchmark applications. Moreover, the proposed techniques can be executed in the order of milliseconds justifying their deployment in rapid design space explorations for selecting custom instructions.

A rapid method for the accurate estimation of the critical path delays and area measures of custom instructions was devised using a cluster generation based technique. It relies on a set of well defined rules for mapping the basic operations onto the BLEs of the target FPGA. This involved the partitioning the custom instructions into a set of basic clusters that can be efficiently mapped onto the FPGA logic blocks. The impact of consecutive addition operations and shift right operations to the resulting bit-widths of the basic clusters were considered to devise accurate models for the estimation of the critical path delays of basic clusters. Experimental results show that the average estimated critical paths of 150 custom instructions using the proposed method are within 3% of those obtained using hardware synthesis. The strategies for estimating the area utilization of custom instructions on FPGA take into account the influence on area measures due to logic shift offsets. Experimental results show that the average estimated area utilization of the 150 custom instructions using the proposed method is within 1% of the outputs generated by the Xilinx tool. Moreover, the runtime of the proposed estimation process is negligible when compared to the time taken for hardware synthesis.

The proposed high-level estimation technique (based on cluster generation) to select most profitable custom instructions has been shown to result in area-efficient custom instructions without compromising performance. This confirms that increasing the number of templates for custom instruction selection beyond a certain point does not contribute to notable performance gain. Experimental results based on sixteen widely used benchmark applications show that on average, an area reduction of over 25% with negligible loss in compute performance can be achieved using the proposed strategy. In addition, an average area-delay product gain of over 86% was achieved by employing only the most profitable set of custom instructions. It was evident that the proposed method leads to higher area-time gains than an existing alternative that incorporates multiplexers for resource sharing.

The proposed architecture-aware strategy for mapping the custom instructions onto the FPGA architecture leads to further area-time savings by maximizing the utilization of the FPGA logic blocks. It relies on a set of architecture-specific conditions to merge basic clusters together with a heuristic, based on the degree of cluster merging and the frequency of occurrences of the basic clusters, to accelerate the selection of a unique set of merged clusters. The proposed method is capable of achieving significantly lower area-delay products when compared to commercial tools and one of the most efficient methods reported in the literature for resource sharing. In particular, we show that the proposed technique outperforms the resource sharing based method in terms of area-delay product, with average reductions of more than 27%, 34% and 19% for Spartan-3, Virtex-4 and Virtex-5 respectively.

A hierarchical loop-aware partitioning strategy has been proposed to reduce the complexity of the search space for determining the runtime configuration of custom instructions for RISPs. A RFU model, based on BLEs of commercial FPGAs with high degree of configuration memory sharing, was employed to reduce the runtime reconfiguration overhead. Cluster merging was also relied upon to implement a larger number of profitable custom instructions for each configuration. Experimental results based on the Cjpeg application show that cluster merging also leads to the generation of more configurations containing profitable custom instructions. In addition, we have demonstrated that hierarchical loop-aware partitioning with cluster merging can achieve significant reduction in the reconfiguration cost for the partial reconfiguration model due to the presence of large number of common basic/merged clusters in consecutive runtime configurations. The proposed approach leads to over 39% average reduction in the runtime overhead for the partial reconfiguration model. Experimental results show that runtime reconfiguration can lead to an average performance gain of over 45% and 52% for the full reconfiguration and partial reconfiguration model respectively.

We have proposed a framework to generate profitable custom instructions for RISPs with runtime reconfiguration support. The framework systematically integrates the proposed strategies for the automatic generation of profitable custom instructions that can be efficiently ported to the FPGA based RFU model discussed in this thesis. We have also introduced a runtime reconfiguration management scheme that judiciously replaces the configurations of the multi-bit logic blocks in the RFU based on the dynamic execution profile of the application to reduce the reconfiguration overhead. This was motivated by our findings that on average, 50% of the clusters in the applications considered are isomorphic.

It is envisioned that RISPs will play an increasingly important role in future embedded computing platforms due to TTM and NRE pressures. The techniques proposed in this thesis pave the way for a commercial tool that can automatically generate profitable custom instructions for a runtime RISP.

9.2 RECOMMENDATIONS FOR FUTURE RESEARCH

The following identifies future research areas based on the work in this thesis:

- **Architecture-Aware Custom Instruction Identification**: Existing template enumeration algorithms commonly rely on the template constraints to prune off invalid solutions in the search space in order to reduce the computation complexity. These template constraints do not incorporate FPGA architecture specific information, which could lead to the generation of template instances that cannot be mapped efficiently onto the FPGA logic blocks. It will be of immense interest to develop a template identification strategy that takes into consideration the FPGA architecture specific information in order to generate a reduced set of template instances that provide for area-time efficient mapping onto FPGA logic blocks. For example, the legality checks used in the proposed cluster generation technique can be incorporated as a pruning constraint during the enumeration process. Our preliminary results that have been published in [C-2] show that these additional pruning constraints lead to significant reduction in the enumeration runtime. In addition, this strategy has been shown to reduce the problem size of the selection process without sacrificing the quality of the final solution.
- **Architecture-Aware Custom Instruction Selection**: The current framework employs a custom instruction selection algorithm (i.e. LFF) that gives preference to the selection of large templates. While it has been shown that the proposed algorithm can lead to the selection of custom instructions with high performance, they may not be the best candidates for cluster merging, which is performed only

after the custom instructions are selected. It will be of interest to investigate the effects of incorporating the proposed cluster merging technique during template selection process in order to select custom instructions that have a higher potential for area-time efficient mapping on FPGAs.

- **Performance Evaluation of Hierarchical Loop-Aware partitioning Strategy with Additional Benchmarks**: Noting that runtime reconfiguration on RISPs is attractive only when the performance of the custom instructions can outweigh the high reconfiguration overheads, we have demonstrated the effectiveness of the proposed hierarchical loop-aware partitioning strategy with the help of Cjpeg application. The hierarchical loop-aware partitioning strategy could be further refined by examining more applications such as the Mpeg2 Encoder and H264 Encoder.
- **Performance-Energy Evaluation of Runtime RISP**: Although we have compared the performance of the custom processor with a baseline processor in Section 5.3.5, it is envisaged that an instruction set simulator could be deployed to evaluate the performance gains of the runtime reconfigurable custom instructions. It should be possible to realize this with an existing open source computer architecture simulator such as the SimpleScalar toolset [135] that supports a wide range of processor architecture models. The SimpleScalar toolset can be used to develop the components for runtime reconfiguration support (discussed in Chapter 8) to facilitate realistic performance evaluations. In addition, custom instruction extensions can result in significant reduction in the application code density, which could lead to reduction in cache accesses and

circuit activity in the processor pipeline [60]. Hence, it will be of interest to extend such a tool to evaluate the energy and power savings of a RISP in the presence of custom instructions.

- **Extending Proposed Framework for Coprocessor Design**: Existing works have shown that significant speed-up can be achieved in applications with continuous stream of data to be processed when they are implemented on reconfigurable hardware with coprocessor support. The proposed framework in this thesis can be extended for coprocessor designs that incorporate customized data-paths with a mixture of generic computational elements and area-time efficient custom function units. These custom function units are synonymous to custom instructions, which can be generated using the proposed framework. In addition, the coprocessor system can also support runtime reconfiguration to maximize the utilization of the reconfigurable space. The proposed framework can be further extended to generate runtime configurations for the coprocessor system.

APPENDIX A

PRELIMINARIES OF RECONFIGURABLE COMPUTING ARCHITECTURES

A.1 GRANULARITY AND RECONFIGURATION MODEL

In general, reconfigurable computing architectures can be classified by two main parameters, namely the *granularity* of computations, and *reconfiguration model* [136].

A.1.1 Fine-Grained and Course-Grained Granularity

The granularity of reconfigurable devices is analogous to the data-path width in processor technology. More specifically, it defines the level of complexity of functional primitives that can be executed in the reconfigurable architecture. Typically, the granularity is expressed by the argument size of the functional primitives. For example, bit-level granularity characterizes reconfigurable devices that can control and process single-bit-wide data using functional primitives such as Look-Up Tables (LUT) implementations. Reconfiguration is limited to the granularity of the devices, and hence bit-level re-configurability is also referred to fine-grained re-configurability.

Commercially available FPGA architectures typically adopt SRAM based programmable logic with fine-grained re-configurability. In these architectures the configuration of the FPGA is controlled by the configuration memory, which consists of

SRAM cells. As the SRAM bits are connected to the configuration points in the FPGA, configuration of the FPGA can be achieved by programming the SRAM bits [137].

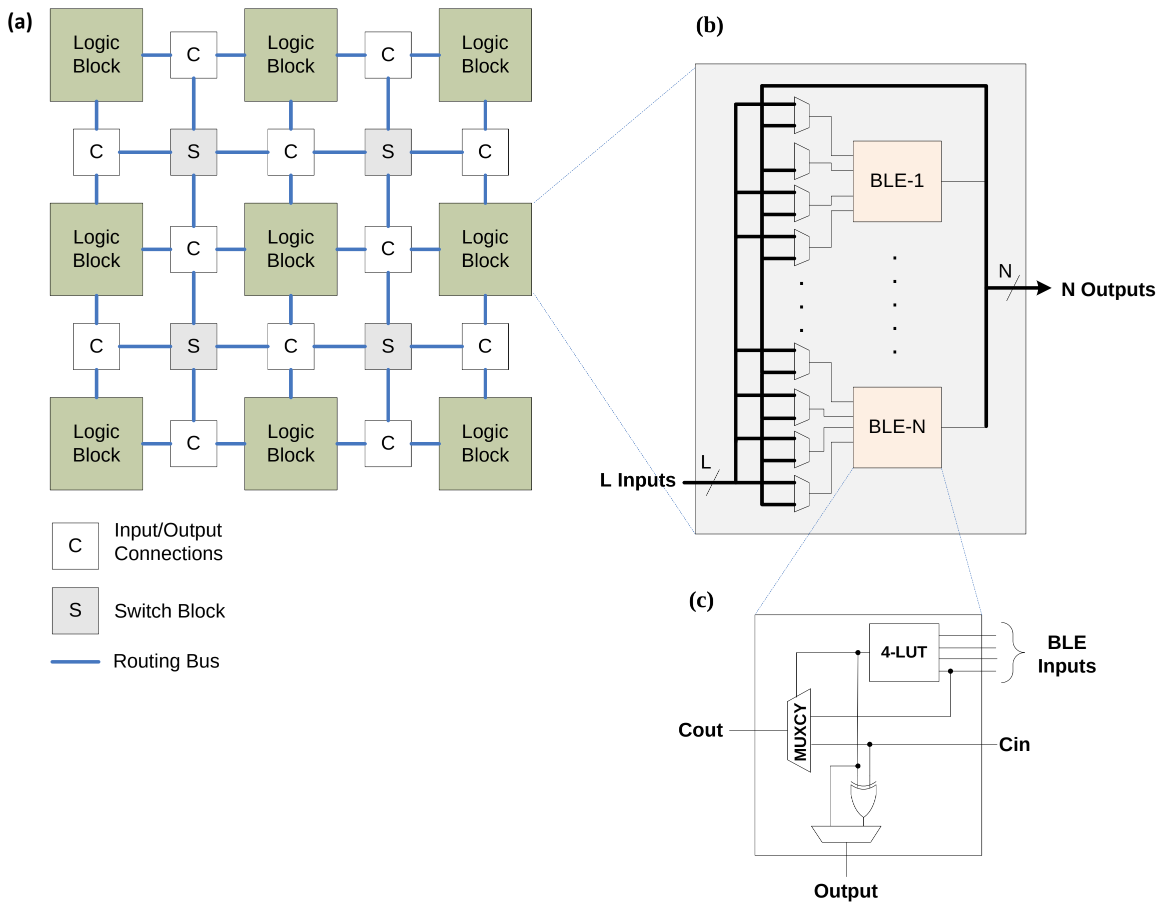


Figure A-1: (a) Island-style FPGA, **(b)** logic block, **(c)** basic logic element

Figure A-1(a) shows a generic island style FPGA architecture [138][139], which consists of a two dimensional array of logic blocks that are interconnected by switches and routing bus. A logic block consists of *N* Basic Logic Elements (BLEs) as shown in Figure A-1(b). Each BLE input is connected to the *L* logic block inputs and the *N* BLE outputs through a *L*+*N*-to-1 multiplexer. Figure A-1(c) shows the structure of a BLE, which is a simplified version of that found in commercially available Xilinx FPGA devices [43]. It can be observed that each BLE is composed of a *K*-input LUT (*K*-LUT) and fast carry-logic (*K* is 4 in the example). The fine-grained programmable LUTs allow any function of up to *K* inputs to be implemented, providing for generic logic realization. The fast carry-logic aims to speed up carry-based computations, such as addition, parity, etc. [137]. Increasingly, commercial FPGAs are incorporating coarse grained functional blocks to significantly improve the density, speed and power of the device.

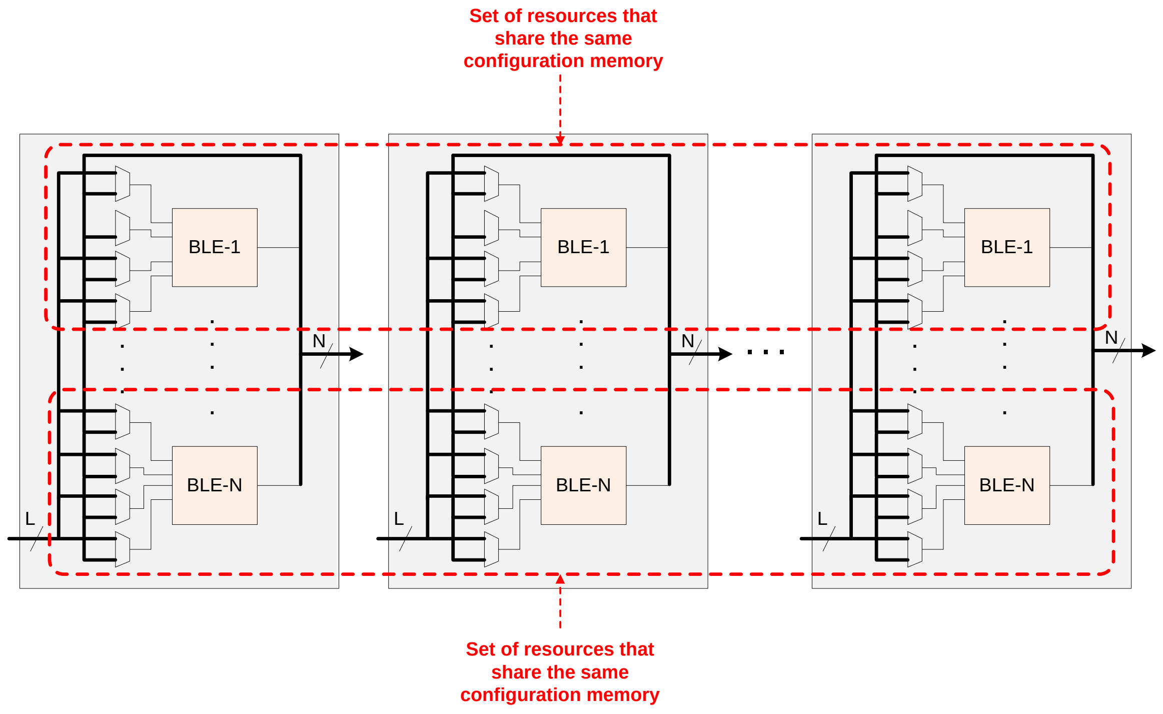

Set of resources that share the same configuration memory
BLE-1
BLE-1
BLE-1
N
N
N
L
L
L
BLE-N
BLE-N
BLE-N
Set of resources that share the same configuration memory

Figure A-2: Multi-bit logic block

FPGAs are commonly used to implement large arithmetic-intensive applications, which often requires regularly structured data-paths that process signals which can be connected in the form of a bus. Coarse-grained granularity based FPGA architectures can be employed to exploit the regularity of these data-path implementations. In [122][123], multi-bit logic blocks have been proposed to exploit configuration memory sharing for implementing of multiple-bit-wide data operations. Figure A-2 shows the multi-bit logic block architecture proposed in [122], where a group of BLEs share the same configuration memory to implement a single bit-slice of the data-path. It has been shown that significant FPGA area savings can be achieved with the multi-bit logic block FPGA architecture through configuration memory sharing in the logic blocks and routing resources.

A.1.2 Reconfiguration Model

The reconfiguration model of the FPGA defines the frequency of the reconfiguration process and the smallest amount of resources that can be reconfigured at any one time [136][137]. The reconfiguration model can be characterized by the following:

- *Static reconfiguration* refers to the case where the configuration memory can only be programmed before the operation of the FPGA.
- *Dynamic reconfiguration* or *runtime reconfiguration* refers to the process of reconfiguring the hardware at runtime to accelerate different applications or different parts of a particular application.
- *Full reconfiguration* characterizes the case where a minor change in the configuration requires the complete reprogramming of the entire configuration memory.
- *Partial reconfiguration* enables a portion of the configuration memory to be programmed without affecting the rest of the configuration.

While runtime reconfiguration can increase hardware utilization, it can incur significant reconfiguration overhead depending on the capacity and granularity of the FPGA device. Modern high-end FPGAs can have tens of millions of configuration points, and the time spent on programming the SRAM bits can reach the order of hundreds of milliseconds [49]. Hence, in order to maximize the performance of runtime reconfigurable computing systems, it is essential to manage and minimize the reconfiguration overhead.

A.2 COUPLING SCHEMES IN PLATFORM FPGAS

The coupling scheme, which defines the position of the reconfigurable fabric with respect to the microprocessor, directly affects the performance of the system and the type of applications that will benefit from the platform FPGA. Figure A-3 shows the different positions that the reconfigurable logic can be placed in relative to the processor [3]. The three main processor configurations are:

- *Attached Processor*: The reconfigurable logic is loosely coupled with the processor. Data transfer and communication control with the memory and processor are performed via a bridge between the main bus and I/O bus (e.g. PCI bus). The communication time is the highest among the three systems.
- *Coprocessor*: The reconfigurable logic shares the same bus as the processor, and hence the communication time is reduced. Interrupt schemes are typically used for the communication between the processor and coprocessor. The attached processor and coprocessor system may benefit from the parallel execution of the processor and dedicated computations in the reconfigurable fabric.

- *Reconfigurable Instruction Set Processor (RISP)*: The reconfigurable fabric or RFU (Reconfigurable Functional Unit) is placed within the processor. The instruction decoder issues instructions in the form of custom instructions to the RFU as if it were one of the standard functional units in the processor.

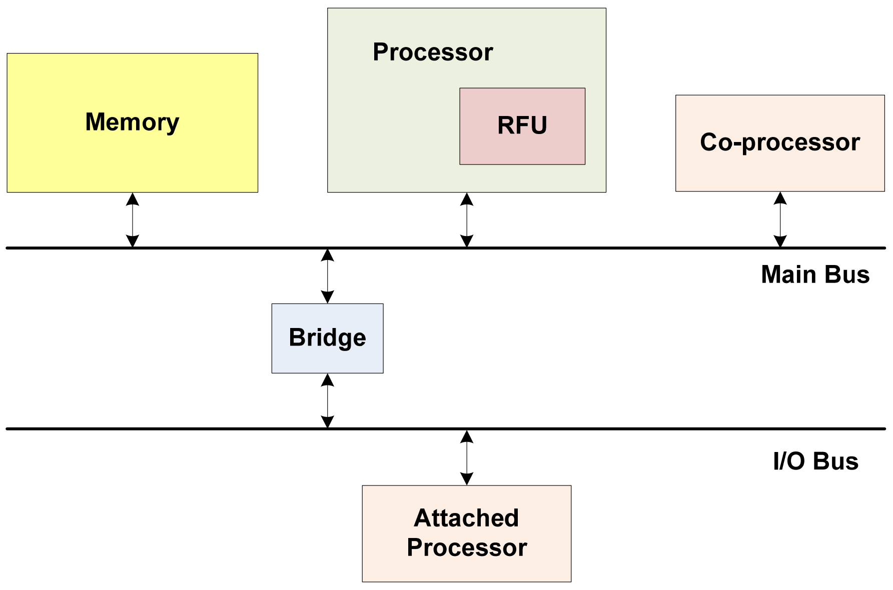


Figure A-3: Coupling schemes between the processor and reconfigurable logic

Early reconfigurable systems have mostly adopted the attached processor and coprocessor systems due to the ease of constructing the system with existing FPGA devices and the ability to employ large amount of reconfigurable resources. These systems can perform generic memory accesses to the shared memory via the main bus or I/O bus. However, the overhead in data and control transfer contribute to significant communication time, and hence the attached processor and coprocessor system is only suitable for streaming applications with continuous stream of data to be processed (e.g. finite impulse response filtering and discrete cosine transform).

APPENDIX B

OVERVIEW OF CUSTOM INSTRUCTION IDENTIFICATION

B.1 SUPPORT FOR CFG AND DFG GENERATION

Instruction set customization is usually performed on the Intermediate Representation (IR) of a compiler. To date, several compiler infrastructures have been developed for research purposes and are made readily available as open source programs. These include the Trimaran compiler infrastructure, which we have used in our experiments. Trimaran is a compiler infrastructure that supports state of the art compiler research in Instruction-Level-Parallelism based architectures. Figure B-1 shows the main components of the Trimaran compiler infrastructure, which comprises the following:

- Machine Description Language (MDES), for describing Instruction-Level-Parallelism based architectures.
- Parameterized Instruction-Level-Parallelism based architecture (HPL-PD).
- Front-end C compiler, called IMPACT that performs parsing, type checking, and a large suite of high-level (i.e. machine independent) optimizations.
- Back-end compiler called ELCOR that performs instruction scheduling, register allocation, and machine-dependent optimizations.
- Extensible IR that represents control flow, data and control dependence, and many other attributes.

- Cycle-level simulator which provides runtime information on execution time, branch frequencies, and resource utilization.
- Integrated Graphical User Interface (GUI) for configuring and running the Trimaran system.

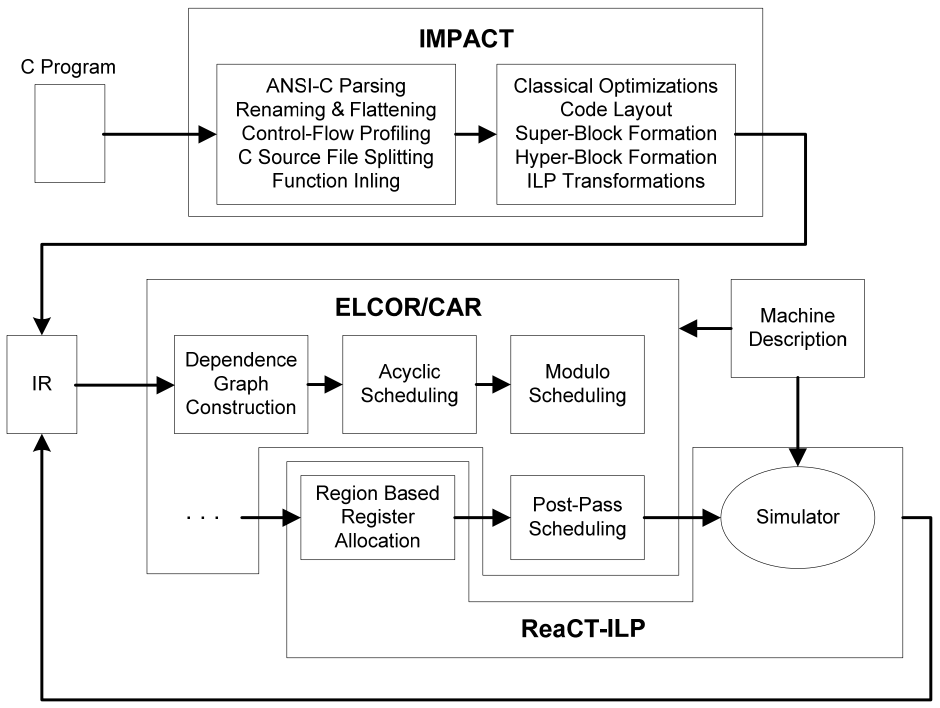


Figure B-1: Overview of the Trimaran compiler infrastructure

The most recent version of Trimaran enables the user to specify custom instructions in the form of acyclic computation graphs in the MDES. The Elcor pattern matcher will find all instances of the computation graphs that occur in the IR and replace them with the corresponding custom instructions. This feature in Trimaran, which facilitates the replacement of custom instructions in the application IR, is an important step towards increasing opportunities for research in instruction set customization.

B.2 CONTROL AND DATA FLOW GRAPHS

Control Flow Graph (CFG) and Data Flow Graph (DFG) are commonly accepted forms of IR upon which a compiler can perform some optimization. In a CFG, each node represents a basic block with one incoming and at most two outgoing edges. The edges of CFG dictate the control flow of the program. Hence, if a basic block has two outgoing edges, denoting an if-else condition, only one outgoing edge will be taken at any instance of time during program execution.

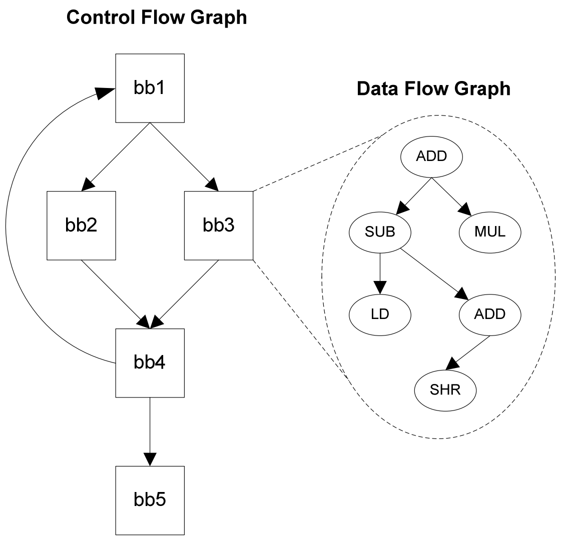


Figure B-2: Control-Data Flow Graph

Each node or basic block of the CFG can be represented as a DFG. The nodes of a DFG consist of primitive operations, while the edges represent the data dependencies of the operations. In essence, the CDFG is the combination of CFG and DFG. An example of a CDFG is shown in Figure B-2, where *bbn* denotes basic block *n*.

The Trimaran compiler infrastructure automatically generates the IR of an application in the form of a CDFG. The CDFG can be captured in graphical or textual representation. As shown in Figure B-1, the IR can be obtained after the front-end and back-end compiler optimizations. The automatic generation of the IR under a user-control optimization environment with Trimaran provides for a convenient and effective means for developing instruction set customization techniques. In addition, although not the focus of this project, the extensible nature of the Trimaran infrastructure also facilitates the development of a compiler toolset for RISPs.

B.3 CONSTRAINTS FOR TEMPLATE IDENTIFICATION

Custom instruction identification is typically performed on a DFG by imposing several topology constraints on the templates. The following describes the restrictions that are commonly imposed on the template instances.

- *Operation type*: The permissible operation types for custom instruction implementation depends on whether the processor core can facilitate them through supporting micro-architectures such as the arbitrator logic (see [C-9]), register file, memory accesses, etc. For example, an RISP can only facilitate custom instruction implementations of integer types if it only provides the coupling logic between the integer unit and RFU. Memory operation types are permissible only if the RFU can directly access the data cache.
- *Number of operands*: The architecture of the core processor may impose constraints on the maximum number of source and destination operands that can be used by the custom instructions. For example, the NIOS II processor consist of

a register file with two read ports and one write port, and hence only custom instructions with two inputs and one output operands are considered for implementation on this platform.

- *Control Flow*: Custom instruction identification is typically performed within basic block boundaries. The assumption is that the compiler cannot exploit instructions across basic block boundaries.

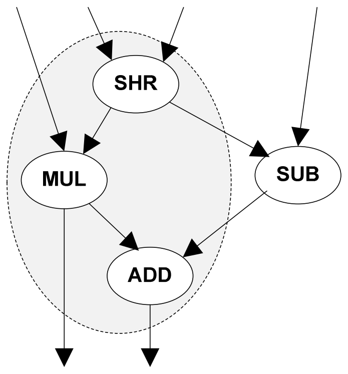


Figure B-3: A non-convex sub-graph

- *Convexity*: The convexity constraint for sub-graphs must be met before the sub-graph can be considered a custom instruction candidate. Figure B-3 shows an example of a non-convex sub-graph (consisting of operations within the shaded region), whereby there exists a path from an operation within the sub-graph (i.e. *SHR*) to another operation within the sub-graph (i.e. *ADD*), and there exist an operation in the path that is not part of the sub-graph (i.e. *SUB*). The convexity constraint is imposed as a legality check to ensure that a feasible scheduling exists. For example in Figure B-3, if we assume that all inputs to the instructions are available at issue time, no feasible scheduling exists for the sub-graph that will respect the dependences on the graph when the sub-graph is collapsed into a single custom instruction.

APPENDIX C

HIGH-LEVEL ESTIMATION RESULTS

The following table shows the results of high-level estimation for 150 custom instructions from sixteen applications.

Application	Template ID	Number of Vertices (Operations)	Number of Edges	Number of Inputs	Number of Outputs	Actual Delay (ns)	Estimated Delay (ns)	Actual Area (LUT)	Estimated Area (LUT)
Adpcm Dec	1	2	1	2	1	8.034	8.108	30	30
	2	2	1	3	1	6.165	6.062	32	32
	3	2	1	2	1	8.123	8.108	32	32
	4	3	2	2	1	6.556	6.062	32	32
	5	3	2	2	1	8.041	8.108	30	30
	6	4	3	2	1	8.644	8.108	32	32
Adpcm Enc	7	2	1	2	1	8.034	8.108	30	30
	8	2	1	2	1	8.123	8.108	32	32
	9	2	1	3	1	6.165	6.062	32	32
	10	2	1	2	1	6.062	6.062	28	28
	11	3	2	3	1	8.137	8.108	30	30
	12	3	2	2	1	8.041	8.108	30	30
	13	5	4	2	2	8.643	8.108	32	32
AES	14	2	1	3	1	9.090	9.044	62	64
	15	2	1	2	1	7.989	8.108	29	29
	16	2	1	2	1	6.568	6.062	32	32
	17	2	1	2	1	6.165	6.062	32	32
	18	2	1	2	1	8.140	8.108	32	32
	19	2	1	1	1	5.822	5.867	0	0
	20	2	1	3	1	6.165	6.062	32	32
	21	3	2	2	2	8.140	8.108	32	32
	22	2	1	3	1	9.522	9.770	64	64
	23	2	1	2	1	6.062	6.062	31	31
	24	2	1	3	1	6.165	6.062	32	32
	25	2	1	3	1	8.841	9.044	64	64
	26	3	2	4	1	9.615	9.770	96	96
	27	4	3	5	2	9.602	9.770	128	128

	28	2	1	2	1	8.123	8.108	32	32
	29	2	1	1	1	5.282	5.867	0	0
	30	2	1	1	1	5.737	5.867	0	0
	31	3	2	3	1	8.241	8.108	32	32
	32	4	3	4	1	6.218	6.062	32	32
	33	5	4	4	1	10.404	9.770	91	93
	34	4	3	2	1	8.644	8.108	32	32
	35	6	5	6	2	9.729	9.770	128	128
	36	5	4	3	1	10.090	9.770	64	64
	37	5	5	2	1	8.569	8.108	30	30
	38	6	5	3	2	10.149	9.770	64	64
	39	8	7	5	1	12.253	10.916	118	125
	40	9	9	4	1	12.282	10.916	121	125
Basicmath Large	41	2	1	1	1	5.282	5.867	0	0
	42	4	3	3	1	6.602	6.062	2	2
Bitcount	43	2	1	3	1	8.226	8.108	32	32
	44	2	1	3	1	8.841	9.044	64	64
	45	3	2	3	1	8.226	8.108	4	4
	46	20	23	10	1	15.928	14.756	160	160
Blowfish Dec	47	2	1	3	1	8.841	9.044	64	64
	48	2	1	2	1	6.062	6.062	24	24
	49	2	1	3	1	8.841	9.044	64	64
	50	3	2	3	1	8.957	9.044	60	60
	51	4	3	2	1	8.034	8.108	8	8
	52	4	3	4	1	9.536	9.044	60	60
	53	5	4	4	1	9.536	9.770	54	54
	54	6	5	4	1	11.043	11.432	94	96
Blowfish Enc	55	2	1	3	1	8.841	9.044	64	64
	56	2	1	2	1	6.062	6.062	24	24
	57	2	1	3	1	8.841	9.044	64	64
	58	3	2	3	1	8.957	9.044	60	60
	59	4	3	2	1	8.034	8.108	8	8
	60	4	3	4	1	9.536	9.044	60	60
	61	5	4	4	1	9.536	9.770	54	54
	62	6	5	4	1	11.043	11.432	94	96
Cjpeg	63	2	1	2	1	8.078	8.108	31	31
	64	2	1	2	1	8.343	8.108	32	32
	65	2	1	3	1	9.090	9.044	62	64
	66	2	1	2	1	8.034	8.108	30	30
	67	2	1	3	1	9.522	9.770	64	64
	68	2	1	2	1	8.034	8.108	30	30
	69	2	1	2	1	8.123	8.108	32	32
	70	3	2	3	1	9.584	9.770	64	64
	71	2	1	3	1	9.090	9.044	62	64
	72	3	2	3	1	9.046	9.044	62	62

	73	3	2	2	2	8.051	8.108	30	30
	74	2	1	2	1	6.220	6.062	32	32
	75	2	1	2	1	8.123	8.108	32	32
	76	3	2	4	1	9.615	9.770	96	96
	77	4	3	4	2	9.602	9.77	94	94
	78	2	1	2	1	8.034	8.108	30	30
	79	2	1	3	1	8.841	9.044	64	64
	80	2	1	2	1	8.123	8.108	32	32
	81	4	4	3	2	9.063	9.044	62	62
	82	3	2	3	2	9.540	9.770	64	64
	83	2	1	3	1	8.841	9.044	64	64
	84	2	1	1	1	5.282	5.867	0	0
	85	3	2	2	2	8.051	8.108	30	30
	86	2	1	3	1	9.522	9.770	64	64
	87	3	2	3	1	9.046	9.044	62	62
	88	3	2	3	2	9.107	9.044	62	64
	89	2	1	3	1	8.841	9.044	64	64
	90	3	2	3	1	9.310	9.044	62	64
	91	3	2	3	1	9.433	9.770	64	64
CRC32	92	5	4	4	1	8.190	8.108	30	30
Dijkstra Large	93	3	2	2	2	8.007	8.108	29	29
	94	2	1	2	1	7.989	8.108	29	29
	95	2	1	2	1	8.123	8.108	32	32
	96	3	2	3	1	9.001	9.044	60	58
	97	4	3	3	2	9.406	9.044	61	58
FFT	98	3	2	3	1	6.182	6.062	32	32
	99	2	1	3	1	8.841	9.044	64	64
	100	2	1	2	1	6.568	6.062	32	32
	101	2	1	2	1	8.034	8.108	30	30
	102	2	1	3	1	8.841	9.044	64	64
Patricia	103	2	1	1	1	5.822	5.867	0	0
	104	2	1	3	1	9.090	9.044	62	64
	105	2	1	2	1	7.989	8.108	29	29
	106	2	1	3	1	8.841	9.044	64	64
Pegwit	107	2	1	3	1	6.165	6.062	32	32
	108	2	1	3	1	8.841	9.044	64	64
	109	2	1	2	1	8.034	8.108	30	30
	110	2	1	1	1	5.282	5.867	0	0
	111	2	1	2	1	6.062	6.062	24	24
	112	2	1	3	1	9.090	9.044	62	64
	113	2	1	3	1	9.522	9.77	64	64
	114	2	1	2	1	7.989	8.108	29	29
	115	3	2	4	1	9.433	9.77	60	62
	116	3	2	3	1	9.001	9.044	60	58
	117	6	6	4	1	6.945	6.998	48	48

	118	4	3	4	1	10.377	10.706	89	90
	119	6	7	4	1	12.873	12.752	96	96
	120	8	9	6	1	15.672	16.076	160	160
Rijndael Dec	121	5	2	1	1	5.822	5.867	0	0
	122	9	2	1	1	5.282	5.867	0	0
	123	15	3	2	3	8.957	9.044	60	60
	124	18	2	1	3	6.165	6.062	32	32
	125	19	2	1	2	6.540	6.062	32	32
	126	20	2	1	2	8.123	8.108	32	32
	127	21	2	1	3	9.522	9.770	64	64
	128	22	3	2	3	8.137	8.108	30	30
	129	25	3	2	2	6.602	6.062	8	8
	130	33	4	3	3	8.137	8.108	8	8
	131	39	4	3	2	8.646	8.108	32	32
	132	42	6	5	4	11.043	10.706	94	96
Rijndael Enc	133	2	1	1	1	5.822	5.867	0	0
	134	3	2	3	1	8.957	9.044	60	60
	135	2	1	2	1	6.540	6.062	32	32
	136	2	1	3	1	6.165	6.062	32	32
	137	3	2	4	1	9.615	9.770	96	96
	138	3	2	3	1	8.137	8.108	30	30
	139	3	2	2	1	6.602	6.062	8	8
	140	3	2	2	1	6.540	6.062	32	32
	141	4	3	3	1	8.137	8.108	8	8
	142	4	3	2	1	8.646	8.108	32	32
	143	6	5	4	1	11.043	10.706	94	96
Sha	144	3	2	2	1	5.282	5.867	0	0
	145	2	1	3	1	9.090	9.044	62	64
	146	3	2	4	1	9.615	9.770	96	96
	147	2	1	2	1	8.123	8.108	32	32
	148	7	6	5	1	9.870	9.98	93	96
	149	4	3	2	1	8.644	8.108	32	32
	150	9	8	6	1	9.923	9.98	94	96

BIBLIOGRAPHY

[1] Florian Lechner and Daniel Walter, "An Introduction to Embedded Systems", November 2006, Online: http://163.15.202.98/localuser/wang1/doc/introduction.pdf (Last visited: 26th January, 2011)

[2] International Technology Roadmap for Semiconductors, "ITRS 2009 Edition - Design", 2009, Online: http://www.itrs.net/Links/2009ITRS/Home2009.htm (Last visited: 25th January, 2011)

[3] Francisco Barat, Rudy Lauwereins and Geert Deconinck, "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective", IEEE Transactions on Software Engineering, Vol. 28, No. 9, September 2002, pp. 847-862

[4] Nikil Dutt and Kiyoung Choi, "Configurable Processors for Embedded Computing", Computer, Vol. 36, No. 1, January 2003, pp.120-123

[5] BBC Research, "Embedded Systems: Technologies and Markets (IFT016)", April 2009, Online: http://www.bccresearch.com/report/IFT016C.html (Last visited: 25th January, 2011)

[6] Philip Koopman, "Embedded System Design Issues (The Rest of The Story)", Proceedings of the IEEE International Conference on Computer Design (ICCD), 1996, pp. 310-317

[7] International Technology Roadmap for Semiconductors, "ITRS 2009 Edition - System Drivers", 2009, Online:: http://www.itrs.net/Links/2009ITRS/Home2009.htm (Last visited: 25th January, 2011)

[8] Jan M. Rabaey and Sharad Malik, "Challenges and Solutions for Late- and Post-Silicon Design", IEEE Design & Test of Computers, Vol. 25, No. 4, July-August 2008, pp. 296-302

[9] Engel Roza, “Systems-On-Chip: What Are The Limits?”, IEE Electronics and Communication Journal, December 2001, pp. 249–255

[10] Jan M. Rabaey, “Silicon Platforms for the Next Generation Wireless Systems – What Role does Reconfigurable Hardware Play?”, International Conference on Field-Programmable Logic and Applications, 2000, pp. 277-85

[11] Chris Edwards, "Speeding Up Is Hard To Do", IEE Review, Vol. 50, No. 9, September 2004, pp. 44-46

[12] Ian Kuon and Jonathan Rose, "Measuring the Gap Between FPGAs and ASICs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 2, February 2007, pp 203-215

[13] Theo A.C.M. Claasen, “System on a Chip: Changing IC Design Today and in the Future”, IEEE Micro, Vol. 23, No. 3, May-June 2003, pp. 20-26

[14] Semico Research Corporation, Online: http://www.semico.com/ (Last visited: 25th January, 2011)

[15] Kim Rowe, "Time-To-Market is a Critical Consideration", Embedded.com, February 2010, Online: http://www.embedded.com/columns/guest/223100896 (Last visited: 25th January, 2011)

[16] Frank Vahid, Roman Lysecky, Chuanjun Zhang and Greg Stitt, "Highly Configurable Platforms for Embedded Computing Systems", Microelectronics Journal, Vol. 34, No. 11, November 2003, pp. 1025-1029

[17] Jordan Plofsky, "The Changing Economics of FPGAs, ASICs and ASSPs", RTC Magazine , April 2003, Online: http://rtcmagazine.com/articles/view/100227# (Last visited: 25th January, 2011)

[18] Bart Mesman, Ben Spaanenburg, Ed Brinksma, Ed Deprettere, Eric Verhulst, Floris Timmer, Hans van Gageldonk, Ludwig D.J. Eggermont, Rene van Leuken, Thijs Krol and Wim Hendriksen, "Embedded Systems Roadmap: Vision on Technology for the Future of PROGRESS", Technology Foundation (STW), The Netherlands, March 2002

[19] David Blaker, "Avoiding ASIC Expense and Risk with SiCB Technology", Electro IQ, October 2009, Online: http://www.electroiq.com/index/display/packaging-article-display/5706988372/articles/advanced-packaging/packaging0/integration/die-stacking/2009/10/avoiding-asic_expense.html (Last visited: 25th January, 2011)

[20] Dylan McGrath, "Gartner: ASIC design starts to fall by 22% in '09", EE Times, March 2009, Online: http://www.eetimes.com/showArticle.jhtml?articleID=216401584 (Last visited: 25th January, 2011)

[21] Péter Arató, Zoltán Ádám Mann and András Orbán, "Algorithmic Aspects of Hardware-Software Partitioning", ACM Transactions on Design Automation of Electronic Systems, Vol. 10, No. 1, January 2005, pp. 136-156

[22] M. D. Galanis, G. Dimitroulakos and C. E. Goutis, "Speedups from Partitioning Critical Software Parts to Coarse-Grain Reconfigurable Hardware", 16th IEEE International Conference on Application-Specific Systems, Architecture Processors, July 2005, pp. 50-55

[23] Greg Stitt, "Hardware-Software Partitioning with Multi-Version Implementation Exploration", Proceedings of the 18th ACM Great Lakes symposium on VLSI, 2008, pp. 143-146

[24] Yuanrui Zhang and Mahmut Kandemir, "A Hardware-Software Co-design Strategy for Loop Intensive Applications", IEEE 7th Symposium on Application Specific Processors, July 2009, pp. 107-113

[25] Yu Chen, Ren-Fa Li and Qiang Wu, "Automatic Reconfigurable System-on-Chip Design with Run-Time Hardware-Software Partitioning", 11th IEEE International Conference on Computer-Aided Design and Computer Graphics, August 2009, pp. 484-491

[26] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu and Steve Y-L Lin, "High-Level Synthesis: Introduction to Chip and System Design", Kluwer Academic Publishers, 1992

[27] Sumit Gupta, Rajesh Gupta, Nikil Dutt, and Alexandru Nicolau, "SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits", Kluwer Academic Publishers, 2004

[28] Michael J. Bass and Clayton M. Christensen, "The Future of the Microprocessor Business", IEEE Spectrum, Vol. 39, No. 4, February 2002, pp. 34-39

[29] Ricardo E. Gonzalez, "Xtensa: A Configurable and Extensible Processor", IEEE Micro, Vol. 20, No. 2, March-April 2000, pp. 60-70

[30] Jorg Henkel, "Closing the SoC Design Gap", Computer, Vol. 36, No. 9, September 2003, pp. 119-121

[31] Virage Logic, "ARC Processor Solutions", Online: http://www.viragelogic.com/render/content.asp?pageid=838 (Last visited: 25th January, 2011)

[32] Nick Flaherty, "On the Chip or On the Fly", IEE Review, Vol. 50, No. 9, September 2004, pp. 48-51

[33] Yankin Tanurhan, "Logic suppliers seek ways to embed FPGAs" EE Times India, March 2001, Online: http://www.eetindia.co.in/ART_8800387951_1800009_TA_7f011aab.HTM (Last visited: 25th January, 2011)

[34] Philip Garcia, Katherine Compton,Michael Schulte, Emily Blem, and Wenyin Fu, “An Overview of Reconfigurable Hardware in Embedded Systems,” EURASIP Journal on Embedded Systems, Vol. 2006, No. 1, January 2006, pp. 1–19

[35] Patrick Lysaght and P. A. Subrahmanyam, "Guest Editors' Introduction: Advances in Configurable Computing", IEEE Design & Test of Computers, Vol. 22 , No. 2, March 2005, pp. 85-89

[36] Dylan McGrath, " Study: FPGAs to Grow Faster Than Broader IC Market", DSP DesignLine, July 2009, Online: http://www.dspdesignline.com/news/218501184 (Last visited: 25th January, 2011)

[37] Juan J. Rodriguez-Andina, Maria J. Moure and Maria D. Valdes, "Features, Design Tools, and Application Domains of FPGAs," IEEE Transactions on Industrial Electronics, Vol. 54, No. 4, August 2007, pp. 1810-1823

[38] Wayne Marx and Vineet Aggarwal, "FPGAs Are Everywhere – In Design, Test & Control", RTC Magazine, April 2008, Available: http://rtcmagazine.com/articles/view/100953#b (Last visited: 25th January, 2011)

[39] David Manners, "FPGA Market Soaring To $4bn In 2010", ElectronicsWeekly.com, May 2010, Online: http://www.electronicsweekly.com/Articles/2010/05/19/48677/fpga-market-soaring-to-4bn-in-2010-says-gavrielov.htm (Last visited: 25th January, 2011)

[40] Yankin Tanurhan, "Processors and FPGAs Quo Vadis?", Computer, Vol. 39, No. 11, November 2006, pp. 108-110

[41] Core Technologies, "Soft CPU Cores for FPGA", Available: http://www.1-core.com/library/digital/soft-cpu-cores/ (Last visited: 25th January, 2011)

[42] Partha Biswas, Vinay Choudhary, Kubilay Atasu, Laura Pozzi, Paolo Ienne and Nikil Dutt, "Introduction of Local Memory Elements in Instruction Set Extensions", Proceedings of the 41st Annual IEEE/ACM Design Automation Conference, June 2004, pp. 729-734

[43] Xilinx Data Sheet, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet", DS083 (Version 4.7) November 2007

[44] Altera: NIOS II Processors, Online: http://www.altera.com/products/ip/processors/nios2/ni2-index.html (Last visited: 25th January, 2011)

[45] Stretch Inc., "S6000 Family Software Configurable Processors", Online: http://www.stretchinc.com/products/s6000.php (Last visited: 25th January, 2011)

[46] Michael J. Wirthlin and Brad L. Hutchings, "A Dynamic Instruction Set Computer", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1995, pp. 99-107

[47] Bernardo Kastrup, Arjan Bink and Jan Hoogerbrugge, "ConCISe: A Compiler-Driven CPLD-based Instruction Set Accelerator", Proceedings.of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999, pp. 92-101

[48] Francisco Barat, Murali Jayapala, Tom Vander Aa, Geert Deconinck, Rudy Lauwereins and Henk Corporaal, "Low Power Coarse-Grained Reconfigurable Instruction Set Processor", 13th International Conference on Field Programmable Logic and Applications, September 2003

[49] Scott Hauck and Andre Dehon, "Reconfigurable Computing: The Theory and Practice of FPGA-based Computing", Morgan Kauffman Publishers, 2008

[50] João M. P. Cardoso, Pedro C. Diniz and Markus Weinhardt, "Compiling for Reconfigurable Computing: A Survey", ACM Computing Surveys, Vol. 42, No. 4, June 2010

[51] Paolo Ienne and Rainer Leupers, "Customizable Embedded Processors: Design Technologies and Applications", Morgan Kaufmann Publishers, July 2006

[52] Manfred Glesner, Thomas Hollstein, Leandro Soares Indrusiak, Peter Zipf, Thilo Pionteck, Mihail Petrov, Heiko Zimmer and Tudor Murgan, "Reconfigurable Platforms for Ubiquitous Computing", Proceedings of the 1st Conference on Computing Frontiers, April 2004, pp. 377–389

[53] Greg Stitt and Frank Vahid, "Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic", IEEE Design & Test of Computers, Vol. 19 , No. 6, November 2002, pp. 36-43

[54] Shaoshan Liu, Richard Neil Pittman and Alessandro Forin, "Energy Reduction with Run-Time Partial Reconfiguration", Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2010, pp. 292-292

[55] Menta, Online: http://www.menta.fr/home.html (Last visited: 25th January, 2011)

[56] Syed Zahid Ahmed, Julien Eydoux, Laurent Rougé, Jean-Baptiste Cuelle, Gilles Sassatelli and Lionel Torres, "Exploration of Power Reduction and Performance Enhancement in LEON3 Processor with ESL Reprogrammable eFPGA in Processor Pipeline and as a Co-processor", Design, Automation & Test in Europe Conference & Exhibition, April 2009, pp. 184-189

[57] Mark LaPedus, "Challenges Ahead for FPGA Market", EE Times India, December 2008, Online: http://www.eetindia.co.in/ART_8800556293_1800000_NT_c463b6f4.HTM (Last visited: 25th January, 2011)

[58] Mike Santarini, "EDA: Get serious about FPGA...", EE Times Europe, March 2009, Online: http://www.electronics-eetimes.com/en/eda-get-serious-about-fpga...?cmp_id=7&news_id=21640 1905

[59] Peter Wells, “The Fixed Processor is Dead, Long Live the Battery”, EE Times, October 2004 Online: http://www.eetimes.com/design/power-management-design/4003555/The-fixed-processor -is-dead-long-live-the-battery (Last visited: 28th April, 2011)

[60] Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Paolo Ienne and Laura Pozzi, "Performance and Energy Benefits of Instruction Set Extensions in an FPGA Soft Core", Proceedings of the 19th International Conference on VLSI Design, 2006, pp. 651-656

[61] P.C. Kwan and Christopher T. Clarke, “FPGAs for Improved Energy Efficiency in Processor Based Systems”, Lecture Notes in Computer Science (Proceedings of the 10th Asia-Pacific Conference on Advances in Computer Systems Architecture), Springer-Verlag, Vol. 3740, October 2005, pp. 440 - 449

[62] Fei Sun, Srivaths Ravi, Anand Raghunathan and Niraj K. Jha, “Custom-Instruction Synthesis for Extensible-Processor Platforms”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 2, February 2004, pp. 216-228

[63] Jason Brown and Marc Epalza, “Automatically Identifying and Creating Accelerators Directly from C Code”, Xcell Journal, 58, 2006

[64] Carlo Galuzzi and Koen Bertels, "The Instruction-Set Extension Problem: A Survey", International Workshop on Applied Reconfigurable Computing (ARC), March 2008, pp. 209-220

[65] Pan Yu and Tulika Mitra, "Characterizing Embedded Applications for Instruction-Set Extensible Processors", Proceedings of the 41st IEEE/ACM on Design Automation Conference, June 2004, pp. 723-728

[66] Ryan Kastner, Adam Kaplan, Seda Ogrenci Memik and Elaheh Bozorgzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", ACM Transactions on Design Automation of Embedded Systems, Vol. 7, No. 4, October 2002, pp. 605-627

[67] Kubilay Atasu, Can Özturan, Günhan Dündar, Oskar Mencer and Wayne Luk, "CHIPS: Custom Hardware Instruction Processor Synthesis", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 3, March 2008, pp. 528-541

[68] Nathan T. Clark, Hongtao Zhong and Scott A. Mahlke, "Processor Acceleration Through Automated Instruction Set Customization", Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (MICRO-36), December 2003

[69] Nathan T. Clark, Hongtao Zhong and Scott A. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration", IEEE Transactions on Computers, Vol. 54, No. 10, October 2005, pp. 1258-1270

[70] Laura Pozzi, Kubilay Atasu and Paolo Ienne, "Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 25, No. 7, July 2006, pp. 1209-1229

[71] Jason Cong, Yiping Fan, Guoling Han and Zhiru Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures", Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, February 2004, pp. 183-189

[72] Pan Yu and Tulika Mitra, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors", Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, September 2004, pp. 69-78

[73] Xiaoyong Chen, Douglas L. Maskell and Yang Sun, "Fast Identification of Custom Instructions for Extensible Processors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 2, February 2007, pp. 359-368

[74] Jong-eun Lee, Kiyoung Choi and Nikil Dutt, “Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPs”, IEEE/ACM International Conference on Computer-Aided Design, 2002, pp.649-654

[75] Carlo Galuzzi, Elena Moscu Panainte, Yana Yankova, Koen Bertels and Stamatis Vassiliadis, "Automatic Selection of Application-Specific Instruction-Set Extensions", Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, October 2006, pp. 160-165

[76] Yuanqing Guo, Gerard J.M. Smit, Hajo Broersma and Paul M. Heysters, “A Graph Covering Algorithm for a Coarse Grain Reconfigurable System”, Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, June 2003, pp. 199-208

[77] Paolo Bonzini and Laura Pozzi, “Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors”, IEEE Transactions on Very Large Scale Integration Systems, Vol. 16, No. 10, October 2008, pp. 1259-1267

[78] Deming Chen and Jason Cong, "DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs", IEEE International Conference on Computer-Aided Design, November 2004, pp. 752–759

[79] Joey Y. Lin, Deming Chen and Jason Cong, "Optimal Simultaneous Mapping and Clustering for FPGA Delay Optimization", Proceedings of Design Automation Conference, July 2006, pp. 472-477

[80] Anshuman Nayak, Malay Haldar, Alok Choudhary and Prith Banerjee, "Accurate Area and Delay Estimators for FPGAs", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, March 2002, pp. 862–869

[81] Per Bjuréus, Mikael Millberg and Axel Jantsch, "FPGA Resource and Timing Estimation from Matlab Execution Traces", Proceedings of the International Symposium on Hardware Software Codesign, May 2002, pp. 31–36

[82] Carlo Brandolese, William Fornaciari and Fabio Salice, "An Area Estimation Methodology for FPGA based Designs at SystemC Level", Proceedings of Design Automation Conference, 2004, pp. 129–132

[83] 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, Vol. 11, No. 1, January 2006, pp. 104–122

[84] Sebastien Bilavarn, Guy Gogniat, Jean-Luc Philippe and Lilian Bossuet, "Design Space Pruning Through Early Estimations of Area-Delay Tradeoffs for FPAG Implementations", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 25, No. 10, October 2006, pp. 1950–1968

[85] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu and Steve Y-L Lin, “High-Level Synthesis: Introduction to Chip and System Design”, Kluwer Academic Publishers, 1992

[86] Xilinx User Guide, “XST User Guide”, UG627, Version 11.2, June 2009

[87] Nabeel Shirazi, Wayne Luk and Peter Y.K. Cheung, “Automating Production of Run-Time Reconfigurable Designs”, IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998, pp. 147-156

[88] Zhining Huang and Sharad Malik, “Managing Dynamic Reconfiguration Overhead in Systems-on-a-Chip Design using Reconfigurable Data-paths and Optimized Interconnection Networks”, Proceedings of Design Automation and Test in Europe, 2001, pp. 735-740

[89] Philip Brisk, Adam Kaplan and Majid Sarrafzadeh, “Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs”, Proceedings of Design Automation Conference, June 2004, pp. 395-400

[90] Kenshu Seto and Masahiro Fujita, “Custom Instruction Generation with High-Level Synthesis”, Symposium on Application Specific Processors, 2008, pp.14-19

[91] Werner Geurts, Francky Catthoor, Serge Vernalde and Hugo De Man, “Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications”, Kluwer Academic Publishers, 1997

[92] Nahri Moreano, Edson Borin, Cid de Souza and Guido Araujo, “Efficient Datapath Merging for Partially Reconfigurable Architectures,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 24, No. 7, pp. 969-980, July 2005

[93] Javier Resano, Daniel Mozos, Diederik Verkest and Francky Catthoor, "A Reconfiguration Manager for Dynamically Reconfigurable Hardware", IEEE Design & Test of Computers, Vol. 22, No. 5, September 2005, pp. 452-460

[94] Huynh Phung Huynh, Joon Edward Sim, Tulika Mitra, "An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization", Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, September 2007, pp. 135–144

[95] Lars Bauer, Muhammad Shafique, Simon Kramer and Jörg Henkel, "RISPP: Rotating Instruction Set Processing Platform", ACM/IEEE/EDA 44th Design Automation Conference, June 2007, pp. 791-796

[96] Lars Bauer, Muhammad Shafique, Jörg Henkel, "Efficient Resource Utilization for an Extensible Processor through Dynamic Instruction Set Adaptation", 5th Workshop on Application Specific Processors, October 2007, pp. 39-46

[97] Yung-Chuan Jiang and Jhing-Fa Wang, "Temporal Partitioning Data Flow Graphs for Dynamically Reconfigurable Computing", IEEE Transactions on Very Large Scale Systems, Vol. 15, No. 12, December 2007, pp. 1351-1361

[98] Meenakshi Kaul, Ranga Vemuri, Sriram Govindarajan and Iyad Ouaiss, "An Automated Temporal Partitioning and Loop Fission Approach for FPGA based Reconfigurable Synthesis of DSP Applications, Design Automation Conference, 1999, pp. 616-622

[99] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure and Jon Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", Design Automation Conference, 2000, pp. 507-512

[100] Farhad Mehdipour, Hamid Noori, Morteza Saheb Zamani, Kazuaki Murakami, Mehdi Sedighi and Koji Inoue, "An Integrated Temporal Partitioning and Mapping Framework for Handling Custom Instructions on a Reconfigurable Functional Unit", Asia-Pacific Computer Systems Architecture Conference, August 2006, pp. 219-230

[101] Huynh Phung Huynh, Joon Edward Sim and Tulika Mitra, "An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization", Design Automation for Embedded Systems, Vol. 13, No. 1-2, June 2009, pp. 91-113

[102] Magnús Halldórsson and Jaikumar Radhakrishna, "Greed is Good: Approximating Independent Sets in Sparse and Bounded-Degree Graphs", Proceedings of the Annual ACM Symposium on Theory of Computing, May 1994, pp. 439-448

[103] Trimaran: An Infrastructure for Research in Instruction-Level Parallelism, Online: http://www. trimaran.org (Last visited: 28th April, 2011)

[104] Laura Pozzi and Paolo Ienne, "Exploiting Pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions", International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2005, pp 2-10

[105] Chunho Lee, Miodrag Potkonjak and William H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture, December 1997, pp. 330-335

[106] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge and Richard B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", IEEE International Workshop on Workload Characterization, December 2001, pp. 3-14

[107] The Embedded Microprocessor Benchmark Consortium, Online: http://www.eembc.org/home.php (Last visited: 25th February 2011)

[108] Xilinx User Guide, "Virtex-4 FPGA User Guide", UG070 (Version 2.6), December 2008

[109] Xilinx User Guide, "Virtex-5 FPGA User Guide", UG190 (Version 5.3), May 2010

[110] Bob Zeidman, "The Future of Programmable Logic", Embedded.com, February 2003, Online: http://www.embedded.com/columns/technicalinsights/15201141?_requestid=49658 (Last visited: 25th January, 2011)

[111] Xilinx Application Note, "Design Tips for HDL Implementation of Arithmetic Functions", XAPP215 (Version 1.0), June 2000.

[112] Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber and Timothy Sherwood, “Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 11, November 2001, pp. 1355-1371

[113] Stefan Tillich and Johann Großschadl, "Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography", Computational Science and Its Applications, Lecture Notes in Computer Science, Vol. 3481, 2005, pp. 665-675

[114] L.P. Cordella, P. Foggia, C. Sansone and M. Vento, "Performance Evaluation of the VF Graph Matching Algorithm", Proceedings of the International Conference on Image Analysis and Processing, September 1999, pp. 1172-1177

[115] Daniel Mattson and Marcus Christensson, “Evaluation of Synthesizable CPU Cores”, M.S. thesis, Chalmers University of Technology, Gothenburg, Sweden, 2004

[116] Ajay K. Verma, Philip Brisk and Paolo Ienne, “Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 10, October 2008, pp. 1761-1774

[117] Matt Ramsay, “Dataflow Dominance: A Definition and Characterization”, University of Wisconsin-Madison, December 2003

[118] Gero Dittmann, “Organizing Libraries of DFG Patterns”, IEEE Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004

[119] Xilinx ISE Foundation, Online: http://www.xilinx.com/tools/designtools.htm (Last visited: 25th January, 2011)

[120] Xilinx Data Sheet, "Spartan-3 FPGA Family Data Sheet", DS099, December 2009

[121] Taneem Ahmed, Paul D. Kundarewich, Jason H. Anderson, Brad L. Taylor and Rajat Aggarwal, “Architecture-Specific Packing for Virtex-5 FPGAs”, Proceedings of the ACM/SIGDA Symposium on Field programmable Gate Arrays, February 2008, pp. 5-13

[122] Andy G. Ye and Jonathan Rose, "Using Multi-Bit Logic Blocks and Automated Packing to Improve Field-Programmable Gate Array Density for Implementing Datapath Circuits",

Proceedings of the IEEE International Conference on Field-Programmable Technology, December 2004, pp. 129 - 136

[123] Andy G. Ye and Jonathan Rose, "Using Bus-Based Connections to Improve Field-Programmable Gate-Array Density for Implementing Datapath Circuits", IEEE Transactions on Very Large Scale Integration Systems, Vol. 14, No. 5, May 2006, pp. 462-473

[124] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, “Introduction to Algorithms”, McGraw-Hill Book Company, 1990

[125] Siew Kei Lam and Thambipillai Srikanthan, "Dynamic Multicast Routing in VLSI", Journal of Computer Communications, Vol. 23, No. 11, June 2000, pp. 1055-1063

[126] George Karypis and Vipin Kumar, "A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices", University of Minnesota, September 1998

[127] Philip James-Roxby and Steven A. Guccione, "Automated Extraction of Runtime Parameterisable Cores from Programmable Device Configurations", IEEE Symposium on Field-Programmable Custom Computing Machines, 2000, pp. 153-161

[128] Xilinx, "JBits 3.0 SDK for Virtex-II", Online: http://www.xilinx.com/labs/projects/jbits/ (Last visited: 25th January, 2011)

[129] Edson L. Horta, John W. Lockwood and Sérgio T. Kofuji, "Using PARBIT to Implement Partial Run-time Reconfigurable Systems", Field-Programmable Logic and Applications, January 2002, pp. 93-202

[130] Anup Kumar Raghavan and Peter Sutton, "JPG - A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs", Proceedings of the International Proceedings on Parallel and Distributed Processing2002, pp. 155-160

[131] Xilinx Application Note, "Difference-Based Partial Reconfiguration", XAPP290, Version 2.0, December 2007

[132] Ewerson Carvalho, Ney Calazans, Eduardo Brião and Fernando Moraes, "PaDReH - A Framework for the Design and Implementation of Dynamically and Partially Reconfigurable Systems", Proceedings of the 17th Symposium on Integrated Circuits and System Design, September 2004, pp. 10-15

[133] Ewerson Carvalho, Frederico Moller, Fernando Moraes and Ney Calazans, "Design Frameworks and Configuration Controllers for Dynamic and Partial Reconfiguration", PPGCC-PUCRS Technical Report Series, Porto Alegre, Brazil, 2004

[134] Nathan T. Clark, Jason Blome, Michael Chu, Scott A. Mahlke, Stuart Biles and Krisztian Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors", Proceedings of the 32nd Annual International Symposium on Computer Architecture, June 2005

[135] SimpleScalar Tool Set, Online: http://www.simplescalar.com/v4test.html (Last visited: 25th January, 2011)

[136] Katarzyna Leijten-Nowak, “Template-Based Embedded Reconfigurable Computing”, Phd Thesis, Eindhoven University of Technology, The Netherlands, 2004.

[137] Katherine Compton and Scott Hauck, "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp. 171- 210

[138] Vaughn Betz, Jonathan Rose, "How Much Logic Should Go in an FPGA Logic Block?," IEEE Design and Test of Computers, Vol. 15, No. 1, January-March, 1998, pp. 10-15

[139] Vaughn Betz, Jonathan Rose, Alexander Marquardt, "Architecture and CAD for Deep Sub-Micron FPGAs", Kluwer Academic Publishers, 1999