# Compiler and Architecture Design for Coarse-Grained Programmable Accelerators

by

Mahdi Hamzeh

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved June 2015 by the
Graduate Supervisory Committee:

Sarma Vrudhula, Chair
Kailash Gopalakrishnan
Aviral Shrivastava
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

August 2015

## ABSTRACT

The holy grail of computer hardware across all market segments has been to sustain performance improvement at the same pace as silicon technology scales. As the technology scales and the size of transistors shrinks, the power consumption and energy usage per transistor decrease. On the other hand, the transistor density increases significantly by technology scaling. Due to technology factors, the reduction in power consumption per transistor is not sufficient to offset the increase in power consumption per unit area. Therefore, to improve performance, increasing energy-efficiency must be addressed at all design levels from circuit level to application and algorithm levels.

At architectural level, one promising approach is to populate the system with hardware accelerators each optimized for a specific task. One drawback of hardware accelerators is that they are not programmable. Therefore, their utilization can be low as they perform one specific function. Using software programmable accelerators is an alternative approach to achieve high energy-efficiency and programmability. Due to intrinsic characteristics of software accelerators, they can exploit both instruction level parallelism and data level parallelism.

Coarse-Grained Reconfigurable Architecture (CGRA) is a software programmable accelerator consists of a number of word-level functional units. Motivated by promising characteristics of software programmable accelerators, the potentials of CGRAs in future computing platforms is studied and an end-to-end CGRA research framework is developed. This framework consists of three different aspects: CGRA architectural design, integration in a computing system, and CGRA compiler. First, the design and implementation of a CGRA and its instruction set is presented. This design is then modeled in a cycle accurate system simulator. The simulation platform enables us to investigate several problems associated with a CGRA when it is deployed as an accelerator in a computing system. Next, the problem of mapping a compute intensive region of a program to CGRAs is formulated. From this formulation, several efficient algorithms are developed which effectively utilize CGRA scarce resources very well to minimize the running time of input applications. Finally, these mapping algorithms are integrated in a compiler framework to construct a compiler for CGRA.

## DEDICATION

To Mitra

*whose unconditional love and support made this dissertation possible*

and

My Parents

*who always encouraged me throughout my education*

ACKNOWLEDGEMENTS

I would like to express my gratitude to my family, friends and professors whose help and support made it possible for me to complete this thesis.

I would also like to extend my deepest gratitude to Dr. Sarma Vrudhula for his encouragement, mentorship and guidance throughout my Ph.D. studies. Your first and most important question for me, "what are you passionate about?", was a great inspiration in my graduate career. You have always been enthusiastic, supportive, and energetic with a great passion to explore new ideas and I am grateful for that. I am thankful for accepting me to be a member of your research team.

I would like to thank Dr. Aviral Shrivastava for serving in my committee. He introduced me to coarse-grain reconfigurable architectures in computer architecture course. Since then, I have enjoyed several discussions with him and he has been enthusiastic about new research ideas. I would like to thank Dr. Carole Wu for joining the committee. Her insight and suggestions have been a great resource for me. I am grateful to Dr. Kailash Gopalakrishnan, not only for serving in my committee but also for his mentorship during my internship at IBM research. Several resources and new directions became reachable for me during the time I worked in his team that had an immense influence in my research.

I have been fortunate to work and share space with my classmates and peers at ASU. I would like to thank Digant Desai, for being a great friend and his encouragement to explore new ideas. I would like to thank my friends, Vinay Hanumaiah, Niranjan Kulkarni, Moeed Haghnevis, Mohammad Ali Abbasi, Zahra Abbasi, Shahrzad Shirazipour, Amrit Panda, Yooseong Kim, Dipal Singh, Bryce Holton, Reiley Jeyapaul, Jinghua Yang, and Joseph Davis at ASU.

I would like to express my gratitude and appreciation to CIDSE graduate advisors and to individuals at ASU specially Lisa Christian, Monica Dugan and Pamela Dunn.

I am deeply thankful to Intel Labs in Santa Clara, CA and IBM research at Thomas J. Watson Research Center. I believe my internships provided me a great opportunity to be exposed to new problems and granted me access to industrial resources that were not easily accessible at school.

I gratefully acknowledge the support I received from the following sources: the National Science Foundation - grants CSR-EHS 0509540, NeTS 0905035; Science Foundation Arizona – grant SRG 0211-07; the Stardust Foundation; NSF I/UCRC Center for Embedded Systems – grant DWS-0086; NSF I/UCRC Center for Embedded Systems and from NSF grant 1361926. I would also like to thank the Department of Computer Science for granting me a research/teaching assistantship.

My family have always supported me from early education to this point. I would like to thank my parents, Bahram Hamzeh and Haydeh Moattar for their support. My sister, Anahita, who has always been a true friend for me and my brother, Milad.

Finally and most importantly, I am specially grateful to my wife and love Mitra. Throughout my Ph.D. studies, you have always been by my side without any complaint, every single moment of it. It is because of you that I could complete this work, thanks for your support and encouragement.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . . . x

LIST OF FIGURES . . . . . xi

1 INTRODUCTION . . . . . 1

1.1 Programmable Accelerators: Challenges and Opportunities . . . . . 3

1.2 Problem Statement . . . . . 6

CHAPTER

2 RECONFIGURABLE COMPUTING FROM HARDWARE PERSPECTIVE . . . . . 9

2.1 Background . . . . . 9

2.2 Reconfigurable Computing Alternatives . . . . . 11

2.3 Trends in Reconfigurable Computing . . . . . 12

2.3.1 RU bit width . . . . . 13

2.3.2 Limiting the set of operations at each RU . . . . . 13

2.3.3 Reconfiguration model . . . . . 14

2.3.4 Interconnection . . . . . 15

2.4 Representative CGRAs Designs . . . . . 16

2.4.1 KressArray . . . . . 17

2.4.2 RAW . . . . . 18

2.4.3 MorphoSys . . . . . 19

2.4.4 ADRES . . . . . 21

2.5 CGRA Components . . . . . 22

2.5.1 PE Structure . . . . . 22

2.5.2 Register File . . . . . 23

CHAPTER Page

2.5.3 Local Memory Interface . . . . . 25

2.5.4 Instruction Set . . . . . 26

2.5.5 Interconnection Network . . . . . 32

2.5.6 Control Unit . . . . . 33

3 MAPPING PROBLEM . . . . . 34

3.1 Background . . . . . 35

3.1.1 Kernels . . . . . 35

3.1.2 Input Representation . . . . . 35

3.1.3 Modulo Scheduling . . . . . 39

3.2 Related Works . . . . . 44

3.3 Problem Definition . . . . . 57

3.4 Complexity Analysis . . . . . 59

3.4.1 Mapping Problem in NP-Complete . . . . . 59

3.4.2 Notes on Routing and Re-computation Complexity . . . . . 62

4 EPIMAP . . . . . 67

4.1 Overview . . . . . 68

4.2 Scheduling . . . . . 68

4.3 Placement . . . . . 74

4.4 Re-Scheduling . . . . . 78

5 GENERALIZED RESOURCE ALLOCATION . . . . . 80

5.1 Multi-cycle Extension . . . . . 80

5.2 Supporting Pipeline Operations . . . . . 82

5.3 Placement and Register Allocation . . . . . 84

6 SUPPORTING CONDITIONALS . . . . . 90

6.1 Background and Related Work ..... 91
6.1.1 Partial Predication ..... 91
6.1.2 Full Predication ..... 93
6.1.3 Dual-Issue ..... 94
6.2 REGIMap extension to support conditionals ..... 96
7 A FRAMEWORK TO STUDY CGRAS ..... 103
7.1 Introduction ..... 103
7.2 System Simulation ..... 105
7.3 Compiler ..... 106
7.4 REGIMap Evaluation ..... 108
7.4.1 Optimizing scheduling window factor ..... 109
7.4.2 Optimizing the number of clique search attempts ..... 109
7.4.3 EPIMap performance ..... 110
7.4.4 EPIMap success rate at first mapping attempt ..... 113
7.4.5 Re-scheduling is effective ..... 114
7.4.6 Constraining resources during scheduling is effective ..... 115
7.4.7 Placement and register allocation search space ..... 115
7.4.8 Register allocation performance ..... 116
7.4.9 Multi-cycle implementation severely damages performance ..... 120
7.4.10 Compilation time of loops with EPIMap ..... 121
7.4.11 REGIMap Compilation time scales well with register file size ..... 121
7.4.12 Pipelining is effective ..... 122
7.4.13 Pipelining compilation time overhead ..... 123
7.5 Supporting Loops with Conditionals ..... 123

CHAPTER Page

7.5.1 Need for supporting Conditionals in Loops . . . . . 124
7.5.2 Performance of dual-issue scheme . . . . . 124
7.5.3 Dual issue scheme and CGRA interconnect . . . . . 125
7.6 Performance projection . . . . . 126
8 SUMMARY AND FUTURE WORKS . . . . . 136
REFERENCES . . . . . 138

LIST OF TABLES

Table Page

2.1 Features of Several CGRAs. 14

2.2 Opcode Summary 27

2.3 Details of Predicate Operand Multiplexer. 30

2.4 Details of Operand Multiplexer. 31

2.5 Memory Commands. 31

3.1 The Mapping of Operations in Figure 3.4(e) 58

# LIST OF FIGURES

Figure Page

1.1 A $4 \times 4$ CGRA. . . . . 3

2.1 Components of a Reconfigurable Computing Platform . . . . . 12

2.2 KressArray Architecture . . . . . 16

2.3 RAW Architecture . . . . . 18

2.4 MorphoSys Architecture . . . . . 20

2.5 ADRES Architecture . . . . . 21

2.6 Internal Structure of a PE . . . . . 23

2.7 Internal Structure of a Rotating Register File . . . . . 24

2.8 Flexible Register File Internal Structure . . . . . 25

2.9 CGRA Instruction Format . . . . . 26

2.10 Memory Instruction Format . . . . . 31

2.11 CGRA Interconnection Network . . . . . 32

3.1 Basic Block Highlevel View . . . . . 36

3.2 A CFG Representation of Loop in Algorithm 3.1 . . . . . 37

3.3 DFG Representation of CFG Shown in Figure 3.1 . . . . . 38

3.4 An Example of Accelerating a Loop Using Modulo Scheduling . . . . . 40

3.5 An Example of a Mapping with Re-computation . . . . . 43

3.6 An Example of a Mapping with Routing and Re-computation . . . . . 43

3.7 Problems Associated with Loop Acceleration . . . . . 45

3.8 Policies Taken by Existing Modulo Scheduling Algorithms: Integrated and Decomposed . . . . . 49

3.9 Reduction from 3-partition Problem to CGRA Mapping Problem. . . . . . 61

4.1 Examples of ASAP, ALAP and Modulo Scheduling. . . . . . 70

Figure Page

4.2 An Example of a Complete Mapping Steps. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

5.1 An Example of Mapping a DFG with Multi-cycle Operations. . . . . . . . . . . . . 81

5.2 An Example of a Mapping with Pipelined Operations. . . . . . . . . . . . . . . . . . . . 84

5.3 An Example of a Mapping with Integrated Placement and Register Allocation. 89

6.1 An Example of Partial Predication Transformation with Mapping. . . . . . . . . 92

6.2 An Example of Full Predication Transformation with Mapping. . . . . . . . . . . . 94

6.3 An Example of DFG Transformation for Dual-issue Execution. . . . . . . . . . . . 95

6.4 An Example of DFG Transformation to Torm Packed Nodes in Dual-issue Scheme. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

7.1 An Overview of CGRA Study Framework. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

7.2 Compilation Flow for CGRA Mapping. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

7.3 The Success Rate of Fining Maximum Clique. . . . . . . . . . . . . . . . . . . . . . . . . . . 110

7.4 Performance of Loops Mapped Using EPIMap vs DRESC. . . . . . . . . . . . . . . . 112

7.5 The Success Rate of Placement. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

7.6 Performance Comparison of Loops Mapped Using REGIMap and Register-Aware DRESC When Size of Register File is 8. . . . . . . . . . . . . . . . . . . . . . . . . . 117

7.7 Performance Comparison of Loops Mapped Using REGIMap and Register-Aware DRESC When Size of Register File is 4. . . . . . . . . . . . . . . . . . . . . . . . . . 119

7.8 Performance Comparison of Loops Mapped Using REGIMap and Register-Aware DRESC When Size of Register File is 2. . . . . . . . . . . . . . . . . . . . . . . . . . 127

7.9 The Compilation Time of Loops Using EPIMap and DRESC Algorithms. . 128

7.10 The Compilation Time Using REGIMap and DRESC in Single Cycle CGRA Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129

Figure Page

7.11 The Compilation Time Using REGIMap and RA-DRESC in a Pipelined CGRA. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130

7.12 The Performance Overhead of Supporting Pipelined CGRAs. . . . . . . . . . . . . . . 132

7.13 The Compilation Time Overhead of a Pipelined CGRA. . . . . . . . . . . . . . . . . . 133

7.14 The Number of Loop with Conditional Constructs in SPEC2k6 Benchmark Suit. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134

7.15 The Achieved *II* in Mapping Loops When Different Predication Scheme is Used in a Mesh Interconnected CGRA. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134

7.16 The Achieved *II* in Mapping Loops When Different Predication Scheme is Used in a CGRA with Diagonal Connections. . . . . . . . . . . . . . . . . . . . . . . . . . . . 135

Chapter 1

# INTRODUCTION

The *holy grail* of computer hardware and software design across all market segments is to achieve and sustain levels of improvement in performance on the same scale as increases in transistor density. Over past decades, increasing the number of transistors and clock frequency have driven the performance forward aggressively. On one hand, transistor density has been increasing exponentially, although it slowed down recently. On the other hand, total power consumption on chip has been kept relatively constant. Therefore, designers have been able to maintain the temperature of the chip within silicon working range. Supply voltage scaling was the major driver in power consumption reduction per transistor which has kept the increase in total power consumption modest. However, this golden era has ended recently because supply voltage scaling could not be achieved at the same pace as technology scales.

Recent empirical studies [28] have shown that current strategy to increase performance by increasing the number of cores will probably fail. This is due to the fact that voltage scaling has slowed or almost stopped, and the power consumption of individual cores are not reducing enough to allow the increase in the number of active computing units. Hence, as technology scales, an increasing fraction of the silicon will have to be *dark*, i.e., be under-clocked or powered off. In fact, it is the heat dissipation capability of the silicon package that will become the limiting factor. This study estimated that at 8nm, more than 50% of the chip will have to be dark. The dilemma is to keep the total power consumption on chip constant while transistor density increases without compromising the performance.

One promising and increasingly popular approach to improve energy efficiency is a heterogeneous multi-core which is populated with a collection of specialized or custom hardware accelerators (CHA) each optimized for a specific task such as graphics, image processing, etc. Computation is handed over to the power/energy optimized CHAs and each can be power gated when not needed. Several commercial examples of such systems include *IBM PowerEN processor* [63] and *Qualcomm Snapdragon S4 Pro* [1]. The trend is to having many smaller, highly optimized CHAs integrated with a few general purpose processors.

One of the drawbacks of the CHAs is the lack of flexibility in terms of programmability. While CHAs permit energy efficient use of the silicon area, their *utilization* can be low as they perform one specific function, their design cost is high and it is difficult and costly to *upgrade* them as the underlying algorithms they implement change. At the other end of the spectrum are special purpose instruction set processors such as Graphics Processing Units (GPUs), which have become very popular. Although they are programmable, their energy efficiency and performance advantages are limited to *parallel loops* [97]. Moreover, such specialized processors require significant effort to program them using specialized languages (e.g. CUDA). In between of these two extremes, are Field Programmable Gate Arrays (FPGAs). Their low cost and high degree of reconfigurability, however, is offset by high energy overhead due to *fine-grain* reconfigurability and long interconnects.

A promising alternative to a CHA, GPU and FPGA is the Coarse-Grained Reconfigurable Architecture (CGRA) [102]. A CGRA consists of a number of word-level functional units called programming elements or PEs, that are interconnected through a rich interconnection network. The PEs are connected to a local memory through a shared bandwidth bus. When deployed as an accelerator, CGRAs have been shown to achieve high energy efficiency [13] while demonstrating all advantages of a programmable accelerator.

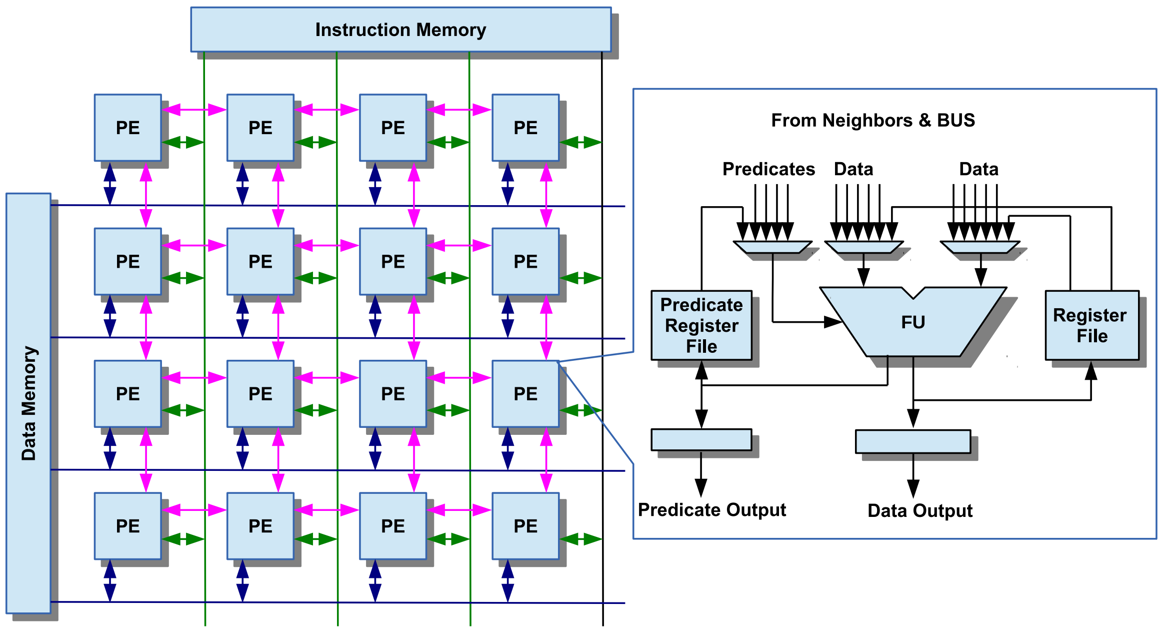


Figure 1.1. A 4 × 4 CGRA.

The promising characteristics of CGRAs have motivated us to study problems associated with using CGRAs in a general-purpose computing platform. This dissertation is categorized into three parts, each aim to address one aspect of the problem. The first part studies CGRA design and implementation. The second part is dedicated to addressing compilation problems associated with acceleration of applications in CGRAs. Finally, the problem of integrating CGRA in a computing platform as an accelerator is studied in the third part.

## 1.1 Programmable Accelerators: Challenges and Opportunities

A CGRA is an array of processing elements (PEs) connected through a rich network. PEs are generally equipped with an ALU and few registers. A PE issues an instruction every cycle, where that instruction dictates which operation to be performed on the set of inputs specified by the instruction. CGRAs vary in PEs functionality, interconnection between PEs, memory and register file organization. An example of a CGRA with PEs connected through a mesh interconnect is shown in Figure 1.1. PEs are connected to neighbouring PEs, and the output of a PE is accessible to its neighbours in the next cycle as an input. In addition, a common data bus from the data memory provides data to all the PEs in a row.

In a general purpose processor, the coordination between components is controlled at execution time while such mechanism is not present in CGRAs. A control unit (CU), either a centralized control unit of in-order processors or a distributed implementation of out-of-order processors, controls and synchronizes events in a general purpose processor. Instruction fetch, issue and dispatch, assigning operations to functional units and controlling the execution flow, all are handled by a CU. As the number of components is increased in a processor, the complexity of such control unit increases significantly. Thus, it is extremely expensive in terms of area and power to increase instruction level parallelism (ILP) beyond certain levels with such sophisticated CUs.

CU must also cross check every pair of instructions in flight to discover data and control dependencies. Increasing ILP requires increasing the number of instructions in-flight. This leads to a significant increase in CU complexity to cross check all instructions and satisfy their data dependencies. Yet another problem arises with interconnection between functional units. As the number of functional unit increases, the complexity of interconnection between functional units increases significantly to provide short paths between functional units executing dependent instructions. This leads to an increase in area, power, and overhead in performance.

Due to these facts, to increase performance at lower hardware complexity, multi-threading programming paradigm has been proposed which shifts these problem to the programmers to specify threads with minimal data dependency. To ensure correctness of the program, programmer has to explicitly handle data dependencies between threads. This is referred to as critical regions [99] and is one of the most challenging problems for programmers.

Designing an effective CU which can deliver a high ILP in a wide range of applications is very challenging nowadays. This is because an abstract view of underline micro-architecture is present to the compiler. The idea behind abstracting the micro-architecture through instruction set architecture (ISA) is to guarantee binary compatibility from implementation to implementation of a processor. In this model, a limited space, such as vector operations [75], is available to explicitly express parallelism in an application .

It is the role of a CU to extract parallelism from application and efficiently utilize computing resources in such paradigm. If binary compatibility was not a major concern and detail of micro-architecture is exposed to a compiler, CU could have been simplified significantly. This is the main idea behind CGRAs.

With a simple and regular structure, a rich set of functional units, distributed register files, and high memory bandwidth, one promising technique to reach higher performance at low power consumption is the use of CGRAs. As opposed to general purpose processors, micro-architecture details are exposed to the compiler which is responsible for assigning operations to PEs rather than a CU. In CGRAs, the mapping of operation in an application to resources in CGRA is static in the sense that operations are assigned to PEs at compile time. Developing efficient application mapping techniques is the most important problems in CGRAs.

Over the last twenty years, CGRAs have been an active field of research. During the 1990s, system design was the primary focus of CGRA research. A catalog of these designs is given in [45]. Around the time that [45] was written, a shift in the focus of CGRA research began. Researchers realized that without automated and efficient application mapping techniques, widespread use of CGRAs is not feasible. Rather than focusing on CGRA design, the majority of CGRA research has directed towards developing effective CGRA compilers.

Compiling applications for CGRAs is an important field of research for several reasons. First and foremost, the CGRA is a promising tool that can be used to increase performance and power efficiency in computing. By allowing reconfigurability at a coarse level, CGRAs provide the flexibility needed to execute a variety of applications with minimal overhead. However, hand mapping of applications to CGRAs is not a viable solution. Automated compilers are needed which can map a broad range of applications to a broad range of CGRAs to unlock the potential advantages of CGRAs.

Several characteristics inherent to CGRAs contribute to the difficulty of programming them. Operations must be scheduled for several (sometimes dozens) of processing elements, and naturally the operations must meet the data dependency requirements of the computation. The connections among the processing elements are often sparse, leading to difficulty in routing operands between two PEs. Often, CGRAs consist of resources (e.g. multipliers) that are shared by the PEs, so conflicts must be resolved. In addition, all of these design elements vary among different CGRA designs, so creating a single compiler that can successfully map applications to multiple CGRAs is further complicated.

## 1.2 Problem Statement

This dissertation aims to study the following problems associated with using CGRAs in a computing platform:

1. Mapping inner most loops onto CGRAs. Many applications execute in phases

and a few set of those phases or regions contribute to most of the execution time. Those regions are usually composed of loop nests. Acceleration of those regions can significantly reduce the application execution time. Therefore, studying the problem of mapping loop nests onto CGRA is a fundamental problem we aim to address in this research. Even though there are many heuristic techniques for this problem, this problem is not studied well. In fact, because there is no precise formulation for this problem, important characteristics of it are not extracted yet. This problem is addressed in chapter 4.

2. Efficient resource allocation in mapping of loop nests onto CGRA. An efficient utilization of available resources on CGRA plays an important role in performance and is studied in chapter 5. Registers are one of those resources which can be utilized to satisfy data dependencies between operations. An effective register utilization reduces the unwanted traffic between accelerator and memory subsystem too. In this dissertation, we present a precise formulation for the CGRA mapping problem while using register files. In contrast to the previous ad-hoc problem definitions, our problem formulation is quite general and supports re-computation, and sharing of routing paths with dependencies.
3. Many important loops embody conditional statements. Therefore, it is important to develop compiler and architectural schemes to accelerate and map such loops. In chapter 6, several problems associated with accelerating loops with conditional statements are studied and effective technique are presented to support acceleration of those loops.
4. CGRA design, implementation, and integration in a system as an accelerator. CGRA in envisioned as an accelerators in computation systems. Two major requirements of this study include an architectural design and implementation of CGRA, and a system level and cycle accurate simulator to run applications

on CGRA which is presented in chapter 7. We aim to address CGRA design problems as well as system level integration of CGRA into a computing platform.

5. Compiler construction for CGRAs. Compiler construction is an error-prone, difficult, and time consuming task. An effective accelerator without a compiler to automatically and efficiently compiles application for it is useless. Thanks to the advances in compiler design, a compiler for a new architecture can be built quickly by reusing existing libraries. We integrate the mapping techniques presented in this dissertation into llvm and study different problems associated with constructing an end-to-end compiler for CGRAs which is presented in chapter 7.

# Chapter 2

# RECONFIGURABLE COMPUTING FROM HARDWARE PERSPECTIVE

## 2.1 Background

Semiconductor device fabrication has always been an expensive and time consuming process [100]. Thus providing reconfigurability [1] feature on silicon significantly reduces implementation time and cost. The earliest attempt to present such feature on silicon dates back to early 70's when Programmable Logic Arrays (PLAs) were introduced [58]. Since then, the reconfigurability feature on silicon has improved significantly from reconfigurable interconnects to programmable logic blocks and interconnects between those blocks. The field of reconfigurable computing received enormous attention in early 1990s when Field programmable gate arrays (FPGA) became commercially available at relatively acceptable price [2]. By the end of that decade, FPGAs were widely deployed [111] in various systems due to two major reasons:

- An exponential increase in the number of logic gate count allows engineers to implement and accelerate a large set applications.
- Financial benefits of using FPGAs in low volume products.

Due to fine grain programmability, FPGAs can be programmed to implement from bit level operations up to super word level functions. This flexibility, however, comes at the cost of programming difficulty and high static power consumption [38].

[1]In past, the term programmability was more commonly used for this feature.

[2]Acceptable price is a vague term but FPGAs are usually used in product prototyping or in application with relatively small market (low volume production). For such applications, non recurring costs are usually dominant and price per device does not play a major role in total expenses.

Limitations and overhead of reconfiguring FPGAs at run-time (in flight), in addition to above-mentioned issues, impose a significant restriction on using FPGAs extensively in wider set of energy-constrained applications.

FPGA vendors acknowledged these problems and started addressing these issues around 2000s. By then, the functionality of CAD tools [3] improved as well as a rich set of libraries for commonly used functions have been developed [11]. In addition, FPGA devices were equipped with rich set of standard hard core and soft core IPs [4]. Yet, engineers need an extensive training to use FPGAs and CAD tools effectively. Better CAD tools made FPGAs easier to program but there are many obstacles yet to be addressed to make FPGAs usable for engineers without extensive hardware training. For instance, programming FPGAs using high level programming labguages has been an active research problem [54, 55, 56, 86] for more than a decade. Static power consumption, however, is unlikely to be addressed well in FPGAs because fine-grained programmability requires high static power consumption.

Coarse-grained reconfigurable architectures or CGRAs aim to extend reconfigurable computing application by presenting two important features. As opposed to FPGAs, CGRAs are programmable at instruction level granularity. Due to this feature, compared to FPGAs, a significantly less silicon area is required to implement CGRAs. Besides, static power consumption is much lower in CGRAs compared to FPGAs. More importantly, given major improvements in compiler technologies, automation in mapping applications specified at high level programming languages onto CGRA can be achieved in near future where a programmer and a compiler interactively generate and optimize a target application for CGRA acceleration. Therefore, there is no need

[3]Tools developed for synthesis and simulation of hardware designs in FPGA.

[4]Hard cores such as embedded powerPC processor, memory controllers, etc., and soft cores such as FFT units, microBlaze processor provided as library in CAD tools, etc.

to extensively train engineers to use CGRAs as compared to FPGAs. This removes a major burden in widespread use of CGRAs.

## 2.2 Reconfigurable Computing Alternatives

Several computing platforms have also received a significant attraction recently due to their performance and financial benefits including Graphics Processing Units (GPUs) and vector accelerators. Each of these acceleration platforms, GPUs, vector machines, and reconfigurable computing platforms, are well suited for one class of application with some overlap.

GPUs have rapidly evolved in last decade with extended programming capability. These changes allow GPUs to be programmed to perform general-purpose computation as opposed to be limited to perform graphic computation only. This topic is generally referred to as GPGPU [90].

GPGPUs are generally equipped with thousands cores and this number has been increasing rapidly over the past decade. GPGPUs are an excellent computing platforms for the workloads that can be partitioned into a large number of threads with minimal interaction between those threads. The effectivity of GPGPUs decreases significantly as the number of workload partitions decreased or the interaction between them increases [52, 107]. In addition, the collision between memory access across threads should be minimized to minimize the performance penalty [107]. In such acceleration model, programmers are responsible to find an effective way to partition the workload.

Single instruction multiple data computing model or SIMD is another alternative acceleration platform. They can effectively accelerate a number of applications at low energy cost. Such platform work well for applications with regular access pattern [75].

This is important because operations are executed on a wide bucket of data. Any irregularity in data bucket should be handled sequentially with extensive penalty [88].

## 2.3 Trends in Reconfigurable Computing

The hardware design aspects of CGRAs have been studied extensively in past three decades. A majority of these designs have target a specific application for acceleration such as PADDI [17] and ULIW[64] that target real-time signal processing applications or MorphoSys [67] that accelerates video compression. A small set of these designs, instead, aim at providing a general computing platform such as ADRES [13].

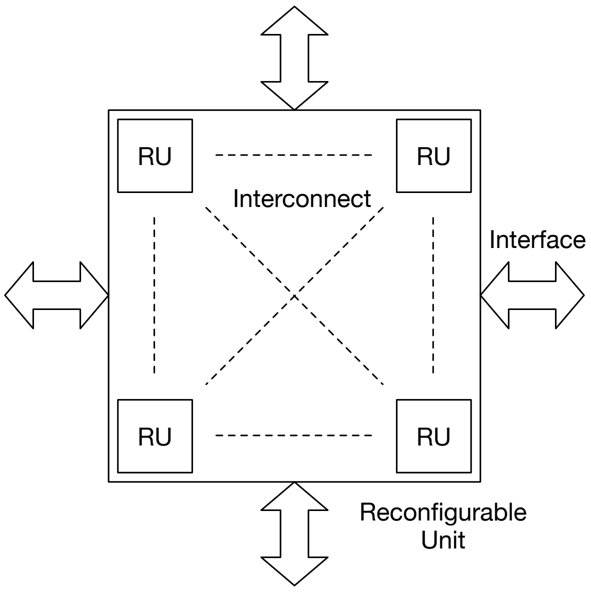


Figure 2.1. Components of a Reconfigurable Computing Platform.

Because several CGRAs are optimized for different goals, there are a number of variations in their designs such as unit operation bit-width, reconfiguration model, programming model, interconnection. In this section, several CGRA design are categorized based on those such variations.

In Figure 2.1, several components on a reconfigurable platform are shown. These components include reconfigurable computing units (RU), interconnection between RUs, and interface between recomputing platform and the rest of a computing system.

### 2.3.1 RU bit width

A gradual trend in reconfigurable computing, from FPGAs to CGRAs, is that the bit width of the functional units has been increasing. Major FPGA vendors such as Xilinx and Altera have been increasing the size of configurable logic blocks (CLBs) as well as integrating an increasing number of digital signal processing units (DSPs). Those DSP units perform standard signal processing operations such as different filters on inputs.

RUs in Figure 2.1 represent CLBs in an FPGA. In an FPGA with DSP units, RUs are heterogeneous representing both CLBs and DSPs. The input bit width of those RUs representing DSPs are generally 16-bits or higher and has been increasing in past decade.

The trend is to integrate an increasing number of wider RUs such as DSPs on FPGAs. A similar trend can be seen in published CGRA design reports. In Table 2.1, the operation width of RUs in several CGRAs are shown.

### 2.3.2 Limiting the set of operations at each RU

A consequence of increasing the bit width of inputs is that RUs would only be able to implement a limited set of boolean functions on those inputs. Supporting all Boolean functions on inputs is not practical. However, the supported operations can

Table 2.1. Features of several CGRAs.

| Name | Year | Operation width | Reconfig. model | Interconnect |
|---|---|---|---|---|
| PADDI [17] | 1990 | 16 bits | Static | Crossbar |
| PADDI-2 [110] | 1993 | 16 bits | Static | Crossbar |
| KressArray [46] | 1995 | 32 bits | Static | Mesh |
| RaPiD [26] | 1996 | 16 bits | Static | Linear |
| MATRIX [83] | 1996 | 8 bits | Dynamic | Mesh |
| RAW [108] | 1997 | 32 bits | Static | Mesh |
| PipeRench [34] | 1998 | 4 bits | Dynamic | Linear |
| REMARC [84] | 1998 | 16 bits | Static | Mesh |
| MorphoSys [67] | 1998 | 16 bits | Dynamic | Mesh |
| DPR [103] | 2002 | 8 bits | Dynamic | Segmented |
| ULIW [64] | 2002 | 16 bits | Dynamic | Mesh |
| ADRES [13] | 2005 | 32 bits | Dynamic | Mesh |
| PPA [93] | 2009 | 32 bits | Dynamic | Mesh |
| DySer [37] | 2011 | 32 bits | Dynamic | Mesh |

be implemented very efficiently at far less area compared to RUs supporting bit level operations.

### 2.3.3 Reconfiguration model

In early reconfigurable platforms, an application could only be supported if the entire application would be mapped onto the computing platfrom. Later, computing platforms were attached to general computing platforms as an accelerator. In this model, the main processor is responsible for programming and controlling the reconfigurable platform. The latency of attaching such accelerator has dramatically decreased as new bus interfaces has been introduced.

Current computing platforms enable us to reprogram the reconfigurable computing platfrom in-flight, frequently. Consequently, it opens the door to accelerate a wide set of applications as the silicon area limitation, imposed on earlier reconfigurable computing platforms, is significantly relaxed. An application can be partitioned into multiple contexts which can be programmed onto the reconfigurable computing platform in several steps.

An important feature that recent reconfigurable platforms present is that they can hold an even increasing context size to minimize the overhead of in-flight reconfiguration. Internally, the context for an RU [5] can change at cycle granularity. By increasing the bit width of interface between general computing platform and reconfigurable computing platform, large contexts can be sent faster.

### 2.3.4 Interconnection

There are several interconnects between RUs that can connect RUs to each other. In FPGAs, these interconnects are partitioned into several segmented. There are few switches that enable or disable connection between different segments. By enabling few switches, a path can be established between an output of an RU to an input of another RU. This is referred to as routing in FPGA CAD tools.

As the length of a path between two RUs increases, the delay to send a data between those units increases too. Due to this negative effects, FPGA CAD tools try to minimize the physical distance between connected components. Designers on the other hand, optimize their performance by pipelining the communication between connected units placed inevitably at far distances.

The number of those segments have been increasing to minimize the delay between neighboring RUs. In addition, interconnects have changed to be increasingly local, short, and regular. Mesh interconnect, for example, is widely used in several CGRA designs. Instead of providing a rich set of switches, in recent CGRAs, RUs would act

[5]we will refer to this as instruction.

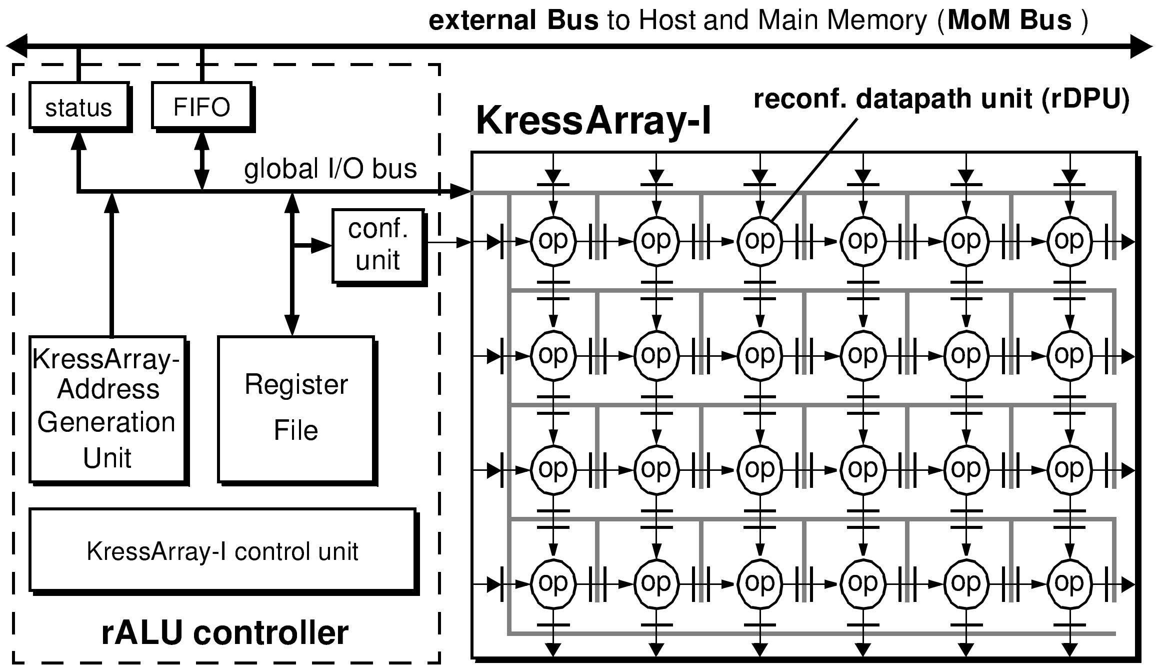

external Bus to Host and Main Memory (MoM Bus )
status
FIFO
global I/O bus
conf. unit
KressArray-Address Generation Unit
Register File
KressArray-I control unit
rALU controller
KressArray-I
reconf. datapath unit (rDPU)
op

Figure 2.2. KressArray Architecture [87].

as switches too. The mapping software is responsible to map dependent computation blocks onto close RUs or pay the penalty of routing in term of RUs and delay.

## 2.4 Representative CGRAs Designs

In this section, several CGRA designs that represent a wide range of CGRAs based on above-mentioned variations are reviewed.

### 2.4.1 KressArray

KressArray [46, 47] (depicted in Figure 2.2) consists of PEs [6] connected through a mesh interconnection network. The interface between reconfigurable array and general computing platform is provided through a hierarchal global routing network to route data from outside to PEs and vice versa. A controller is responsible for managing input and output of data streams, and configuration process. KressArray is among pioneer CGRAs designs with emphasis on scalability. Several switches are provided at the boundaries of interconnection network which can be utilized to connect multiple CGRA devices in a mesh style network.

The datapath of PEs is 32-bit wide. In addition to arithmetic and logical operations, PEs can route data to the neighboring PEs. PEs are equipped with a small register file, a routing switch, and an ALU. The main application of this architecture is multimedia application with high computation and high data throughput demands.

KressArray configuration memory is programmed by the host computer via a configuration bus. PEs are programmed selectively in the configuration process. During configuration process, the address of a PE in the mesh is first asserted. In the next step, the configuration is directly forward to the selected PE. Through configuration, registers can be set in PEs to hold constant values or memory pointers. KressArray is a data driven architecture. Execution of instructions is triggered when all inputs are available. This scheme simplifies the mapping problem significantly.

This architecture is envisioned as an off-chip accelerator. It is a passive device on bus so that the programming or configuration should be initiated by a host machine.

[6] rDPU: reconfigurable datapath unit

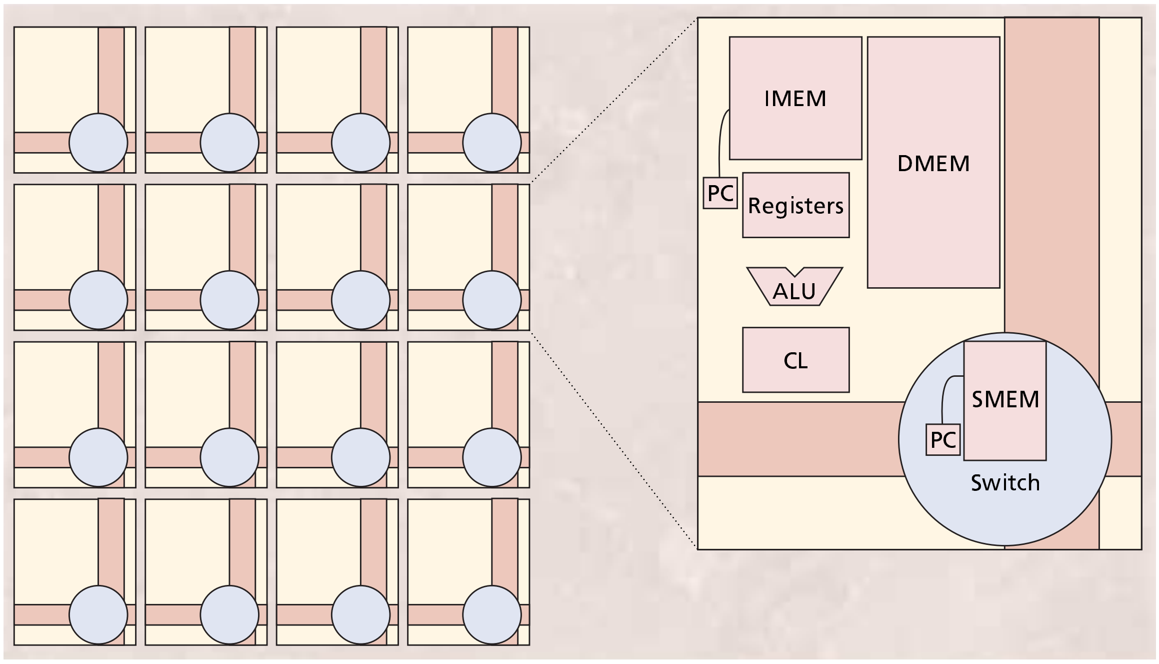


Figure 2.3. RAW Architecture [108].

Besides, input data for computation should be delivered through a data sequencer unit. Several designs such as XPP [9] are inspired from KressArray architecture.

### 2.4.2 RAW

Reconfigurable architecture workstation (RAW) [108] is made up of a set of interconnected PEs [7]. Each PE contains a RISC-like [8] pipeline, data memory, instruction memory, a reconfigurable logic, and a switch. As opposed to most CGRAs, data and instruction memory is distributed in PEs in this architecture. The pipeline is kept minimal without any support of register renaming nor dynamic instruction issuing. In

[7]In their original article, the term tile is used instead of PE.

[8]Reduced instruction set computing.

an inspiring approach, this architecture exposes all low-level details of hardware to the compiler. This enables compiler to allocate resources more efficiently as opposed to pure hardware-based resource allocation.

Each PE is connected to a local switch. A switch can be configured at cycle granularity thanks to a configuration memory in switch. The separation of instruction memory in PE and network switch allows the PE to take arbitrary branches without disturbing the routing of independent messages. Compiler is responsible for orchestrating data with minimal stall occurrence. Should compiler fails to find a static schedule, RAW provides dynamic support of flow control. Figure 2.3 depicts an overview of a RAW Microprocessor along with external RAM interface.

The interconnect between PEs is optimized for single data word transfer. The interconnection is divided into two logical networks, static and dynamic. The static one is utilized for static schedule with data transfer determined by compiler. When no data transfer is scheduled on the network, the instruction scheduled dynamically by RAW control unit can utilize the network as a dynamic one.

### 2.4.3 MorphoSys

The MorphoSys architecture [67] comprises five major components, Reconfigurable Cell Array or RC Array (shown in Figure 2.4(b), a control processor, context memory, frame buffer, and a DMA controller. The RC Array is composed of 64 PEs [9] arranged in a 2-dimensional mesh. The computation model of PEs in a row is SIMD [10] where all of these PEs share the context (instruction). PEs are connected through a three-layer

[9]The term RC is used in their article

[10]Single instruction multiple data

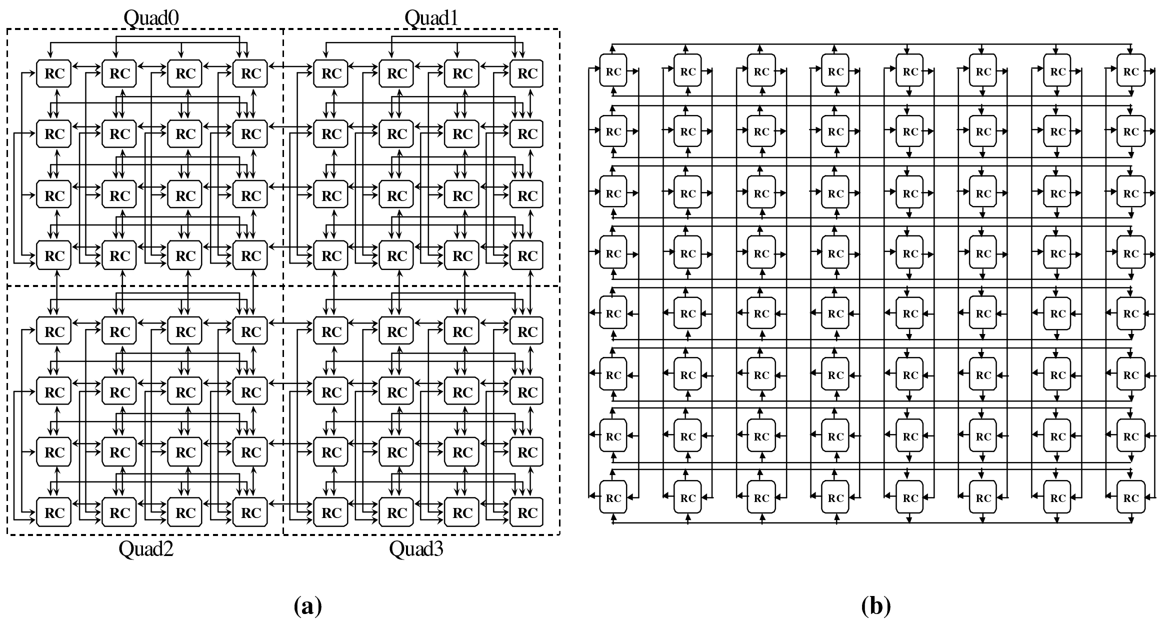


Figure 2.4. MorphoSys Architecture [67].

network which enables fast data exchange between PEs. Each PE incorporates an ALU and a register file. The idea behind RC Array is to implement the datapath for custom instructions for the processor.

Instructions are hold in context memory. This memory is to be programmed through DMA transactions. In fact, RC Array does not have direct access to memory subsystem, rather, all memory operations have to be performed through DMA operations. Frame buffer is used as local memory during computation. Inputs, through DMA operations are copied to this memory and the result of computation would be collected back by DMA.

RC Array can be logically divided into tiles of 4 by 4 PEs as shown in Figure 2.4(a). MorphoSys framework includes a compiler (mCom) to map hybrid code to this architecture. The partitioning of the code between host processor and RC Array, however, should be done manually by user.

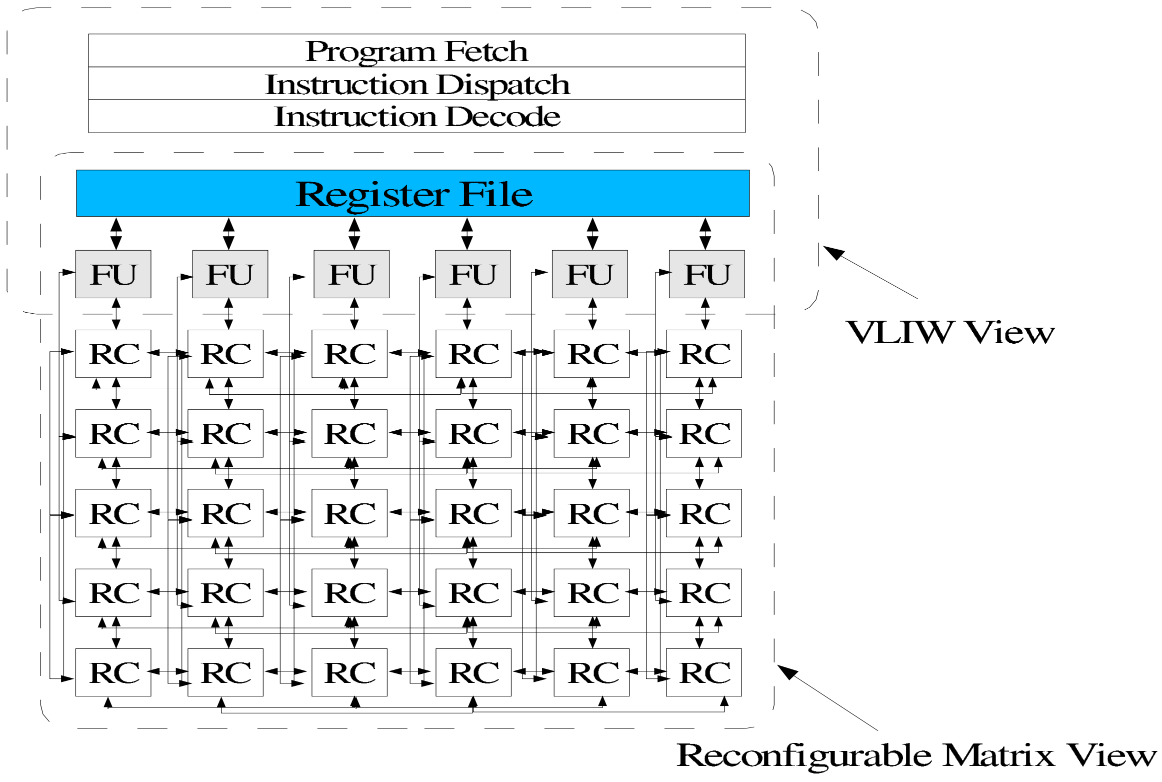


Figure 2.5. ADRES Architecture [80].

### 2.4.4 ADRES

Shown in Figure 2.5, Architecture for Dynamically Reconfigurable Embedded System [13, 80] or ADRES is a VLIW processor tightly coupled with a array of PEs [11]. Each PE is composed of a functional unit (an ALU) and a register file. PEs are logically partitioned into a VLIW processors and a reconfigurable matrix. The VLIW processor executes unaccelerated parts of the application while reconfigurable matrix would accelerate repetitive sections of the code. Due to tight integration of

[11]RC is used instead of PE in their article.

reconfigurable matrix and VLIW processor, the programming model of ADRES is significantly simplified.

Each PE is composed of a functional unit, a register file and small instruction memory. PEs are connected through a mesh interconnection network. There is a predication network which enables ADRES to execute regions with conditional clauses. The register file in VLIW processor is shared with reconfigurable matrix. This reduces the communication between reconfigurable matrix and memory subsystem. A compiler framework is developed for this architecture. It targets innermost loops in application to map on reconfigurable matrix.

## 2.5 CGRA Components

In this section, several components of our CGRA design are presented.

### 2.5.1 PE Structure

The internal structure of a PE is shown in Figure 2.6. An instruction is fed to a PE at cycle granularity which controls all Components of a PE. An instruction controls the functional unit to perform an operation on input(s). Each input is selected using a multiplexer. The inputs to this multiplexer are register file output, output register of the PE, output registers of neighboring PEs [12], immediate value from instruction, or data bus. The predicate inputs are also selected by a multiplexer. Inputs of this multiplexer are pairs of predicate bits. Its inputs come from predicate register file, predicate output of the PE, or predicate outputs of neighboring PEs.

[12] Up, down, left, and right

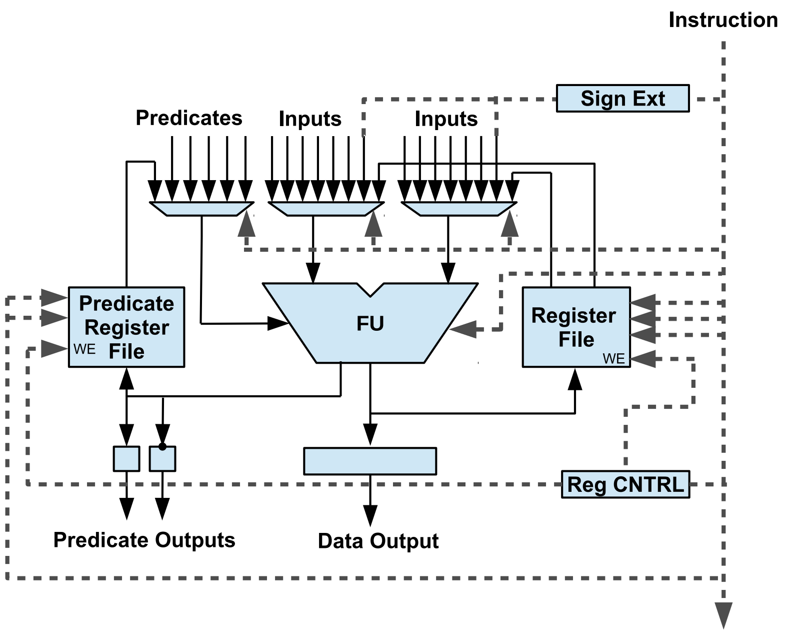


Figure 2.6. Internal Structure of a PE.

An important component of a PE is how writing to register files are enabled. Note that there are two register files, one for data and one for predication. Given an instruction opcode and instruction destination, write enable is asserted for selected register file by register control unit.

### 2.5.2 Register File

The datapath register file, has 4 inputs, 2 for selecting read registers indexes, 1 to select write register index, and 1 bit to enable or disable writing to the register file. As we will discuss later, a rotating register file is an ideal structure for CGRA acceleration model. It is because a rotating register file enables compiler to generate a compact code which virtually displaces the register indexes at run-time. Since CGRAs are well suited to accelerate loop, this structure enables CGRAs to avoid writing to

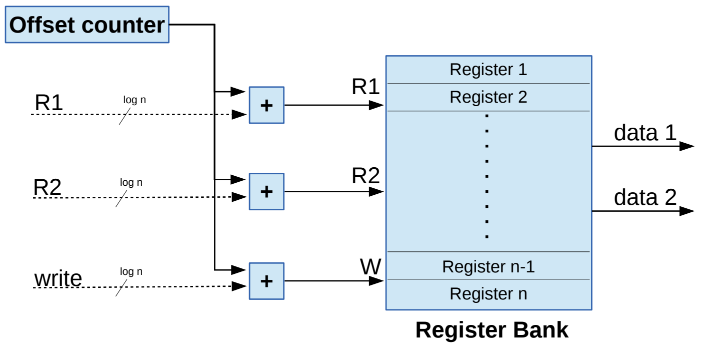


Figure 2.7. Internal Structure of a Rotating Register Rile.

the same register index over and over in consecutive loop iterations. A structural view of a rotating register file is presented in Figure 2.7.

In rotating register file, there is a counter which is incremented at the end of every iteration of the loop. By increasing this counter, a selected index in register file would be increased by one every iteration. For instance, when register 1 at iteration 2 of the loop is written, the value is in fact would be stored in register 3. It is because counter at iteration 2 is 2 which would be accumulated with register index.

In addition to rotating register file, a non-rotating register file is essential at each PE. We logically split the register file into rotating and non-rotating register files as shown in Figure 2.8. Non-rotating register files are necessary to hold constant values and address pointers are usually used to load data from memory or store data back. Such variables do not often change during an execution, an even if they do, it is very infrequent. Thus, it is better to store them in a register file where their indexed fo not change from iteration to iteration. When register indexes are changing, a value stored in a register can be read with different indexes at every iterations. This makes generating instruction for load and store very difficult. In fact, indexing such variables would require extra operations at run-time.

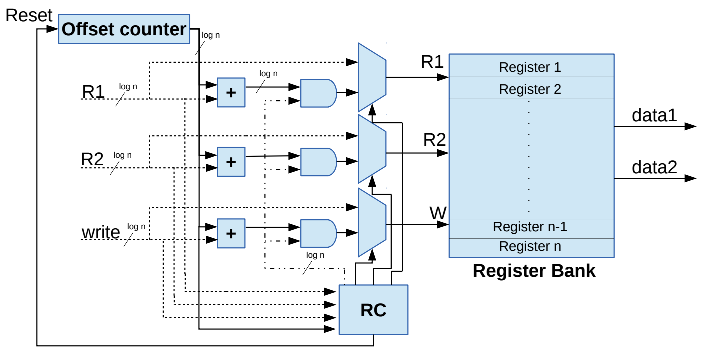
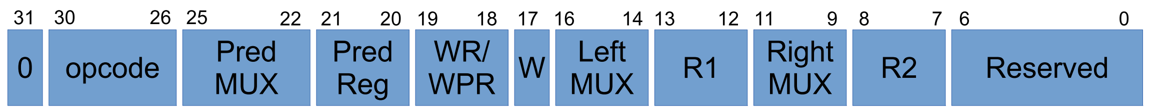


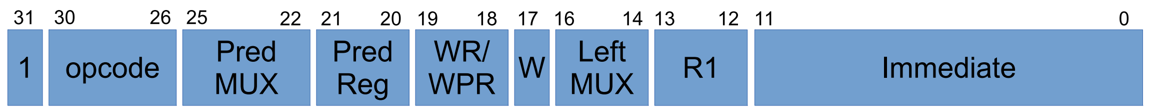
Figure 2.8. Flexible Register File Internal Structure.

### 2.5.3 Local Memory Interface

CGRA is connected to memory subsystem through CGRA memory. This memory acts as a private cache for CGRA. There is a shared bus at each row which provides communication to the memory for PEs at that row. This bus is shared among all PEs at that row, so at any given cycle, only one PE can communicate with memory. This bus consists of address bus and data bus. To load a data from memory, the address bus is asserted first. In the following cycle, data will be available on data bus or there is a cache miss. For a store transaction, address bus and data bus are both asserted at a cycle. If that address is not present in CGRA memory, it would be handled by CGRA memory controller. Memory transactions are controlled by memory instructions. A memory write transaction consists a memory command and two PE indexes. The first index specifies the PE asserting address bus. The second index selects PE asserting data bus if the transaction is a store operation.



(a) Format 1



(b) Format 2

Figure 2.9. CGRA Instruction Format.

### 2.5.4 Instruction Set

There are two types of instructions: instructions issued to PEs and memory instructions. PE instructions consist of arithmetic, logic, and data manipulation operations. Memory instructions deal with load and store operations between PEs and memory subsystem.

As shown in Figure 2.9, PE instructions are categorized into two formats. In the first type, input operands of an instruction are registers, either read from local register file or read from output register of neighboring PEs. In the second type of PE instruction, there is an immediate [13] input operand.

There is no branch instruction in CGRA ISA. However, there is a predication network which can be utilized to conditionally execute instructions. A predication input in an instructions determines whether an instruction should be executed or squashed at a given cycle. When the predicate input is 0, the instruction would not change the state of a PE at that cycle. There is one exception that if an instruction

[13]The value of the operand is static and can be known at compile time.

updates the predicate output, such as conditional instructions [14], it is executed regardless of predicate input. However, if the predicate input is 0 for such instruction, the predicate output is always 0. This is essential to correctly execute predicate instruction in the presence of nested if-clauses.

The first type of PE instructions is depicted in Figure 2.9(a). The first field in the instruction is used to identify instruction type. When it is 0, the instruction has two inputs and one output operands. *Opcode* is 5 bits supporting up to 32 different operations. Details of supported operations in summarized in Table 2.2.

Table 2.2. Opcode Summary.

| Opcode | Mnemonic | Instruction | Action (C style) |
|---|---|---|---|
| 0 | AND | Bitwise AND | $R = Op1 \& Op2$ |
| 1 | ORR | Bitwise OR | $R = Op1 \| Op2$ |
| 2 | EOR | Bitwise XOR | $R = Op1 \oplus Op2$ |
| 3 | ORN | Bitwise OR complement | $R = Op1 \| \overline{Op2}$ |
| 4 | BIC | Bit clear | $R = Op1 \& \overline{Op2}$ |
| 5 | ASR | Arithmetic shift right | $R = Op1 >> Op2$ |
| 6 | LSR | Logical Shift Right | $R = (U)(Op1 >> Op2)$ |
| 7 | LSL | Logical Shift Left | $R = Op1 << Op2$ |
| 8 | ROR | Logical rotate right | $R = ror(Op1, Op2)$ |

[14] Such as IF statement.

|  |  |  |  |
|---|---|---|---|
| 9 | RRX | Sign extended rotate right | $R = rrx(Op1, Op2)$ |
| 10 | CLZ | Count the number of leading zeros | $R = \#$of zeros in $Op1$ |
| 11 | MOV | Copy input to output | $R = Op1$, $PR = POp1$ |
| 12 | MVH | Copy 16 MSB bits from input to output | $R = Op2 << 16\&FFFF0000$ [15] |
| 13 | MVL | Copy 16 LSB bits from input to output | $R = 0000FFFF\&(U)(Op2 >> 16)$ [16] |
| 14 | CMP | Compare | $PR$ [17] $= (Op1 == Op2)$ |
| 15 | SLT | Set less than | $PR = (Op1 < Op2)$ |
| 16 | SLE | Set less than or equal | $PR = (Op1 <= Op2)$ |
| 17 | SLTU | Unsigned set less than | $PR = (U)(Op1 < Op2)$ |
| 18 | SLEU | Unsigned set less than or equal | $PR = (U)(Op1 <= Op2)$ |
| 19 | SOC | Set overflow | $PR = C$ |

[15] Note that if instruction is immediate format, the immediate operand is to be sign extended.

[16] Note that if instruction is immediate format, the immediate operand is to be sign extended.

[17] Predicate output

|  |  |  |  |
|---|---|---|---|
| 20 | ADC | Add with carry | $R = Op1 + Op2$ [18] $+C$ |
| 21 | ADD | Add | $R = Op1 + Op2$ |
| 22 | SUB | Subtract | $R = Op1 - Op2$ |
| 23 | SBC | Subtract with carry | $R = Op1 - Op2 - \overline{C}$ |
| 24 | SEL | Select | $R = Select(Op1, Op2)(predicate)$ |
| 25 | RSC | Reverse subtract with carry | $R = Op2 - Op1 - \overline{C}$ |
| 26 | MUL | Multiply | $R = Op1 \times Op2$ |
| 27 | UML | Unsigned multiply | $R = (U)(Op1 \times Op2)$ |
| 28 | ADD16 | Parallel add | $R\,[31:15] = Op1\,[31:15] + Op2\,[31:15]$, $R\,[31:15] = Op1\,[15:0] + Op2\,[15:0]$ |
| 29 | SUB16 | Parallel subtract | $R\,[31:15] = Op1\,[31:15] - Op2\,[31:15]$, $R\,[31:15] = Op1\,[15:0] - Op2\,[15:0]$ |
| 30 | REM [19] | Remainder | $R = Op1\%Op2$ |
| 31 | DIV | Divide | $R = Op1/Op2$ |

Predicate MUX (*Pred MUX*) field selects predicate input among 4 neighboring PEs predicate output, one from the PE predicate output executing the instruction, and one from its predicate register file. Details are given in Table 2.3. Note that predicate

[18]If Op2 is immediate, it is sign extended.

[19]Division is not implemented yet but reserved for future developments.

Table 2.3. Details of Predicate Operand Multiplexer.

| Predicate MUX | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Predicate Operand | Predicate Register File | Complement Predicate Register File | Left PE Predicate | Left PE Complement Predicate |
| Predicate MUX | 4 | 5 | 6 | 7 |
| Predicate Operand | Right PE Predicate | Right PE Complement Predicate | Up PE Predicate | Up PE Complement Predicate |
| Predicate MUX | 8 | 9 | 10 | 11 |
| Predicate Operand | Down PE Predicate | Down PE Complement Predicate | Output Predicate | Output Complement Predicate |
| Predicate MUX | 12 | 13 | 14 | 15 |
| Predicate Operand | Unused | Unused | Unused | Unused |

output is formed from a predicate result and its complement. This implementation is chosen to simplify the execution both the outcome paths of a conditional clause.

Predicate register (*Pred Reg*) field is used to select the predicate register. *WR* represents the destination register that is to be updated by an instruction. It is also used to update predicate register file (*WPR*). When *W* is 1, the register file is updated otherwise, the output register of a PE is updated only after execution of an instruction. *Left MUX* and *Right MUX* specify the first and second input operands of the instruction, which can be among neighboring PEs output, selected PE output, register file of PE, or Data Bus. Details are given in Table 2.4. *R1* and *R2* are used to specify the first and second register operand, should local registers be used in an instruction. 7 bits are reserved for future increase in size of register files.

The second PE instruction type is shown in Figure 2.9(b). The first field is 1

Table 2.4. Details of Operand Multiplexer.

| Left/Right MUX | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Input Operand | Register File | Immediate | Left PE | Right PE | Up PE | Down PE | Data Bus | Output Register |

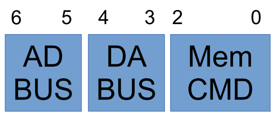


Figure 2.10. Memory Instruction Format.

Table 2.5. Memory Commands.

| Mem CMD | Action |
|---|---|
| 0 | NOP |
| 1 | Load word |
| 2 | Load sub-word high |
| 3 | Load sub-word low |
| 4 | Store word |
| 5 | Store sub-word high |
| 6 | Store sub-word low |
| 7 | Exchange word [20] |

representing immediate format instruction. All fields up to immediate are identical to the first format. When one operand is immediate, the *Right MUX* and *R2* fields are not needed, thus immediate field can be extended to 12 bits.

Memory operations are handled through memory instructions. There is a memory bus arbiter at each row in CGRA. This element executes a memory instruction at every cycle. A memory instruction is formed from 3 fields as shown in Figure 2.10. *AD BUS* specifies which PE in the row asserts address bus. *DA BUS* specifies the PE number in the row asserting data bus. The memory operation, whether it is a load or store, is specified by *Mem CMD*. Memory commands are summarized in Table 2.5.

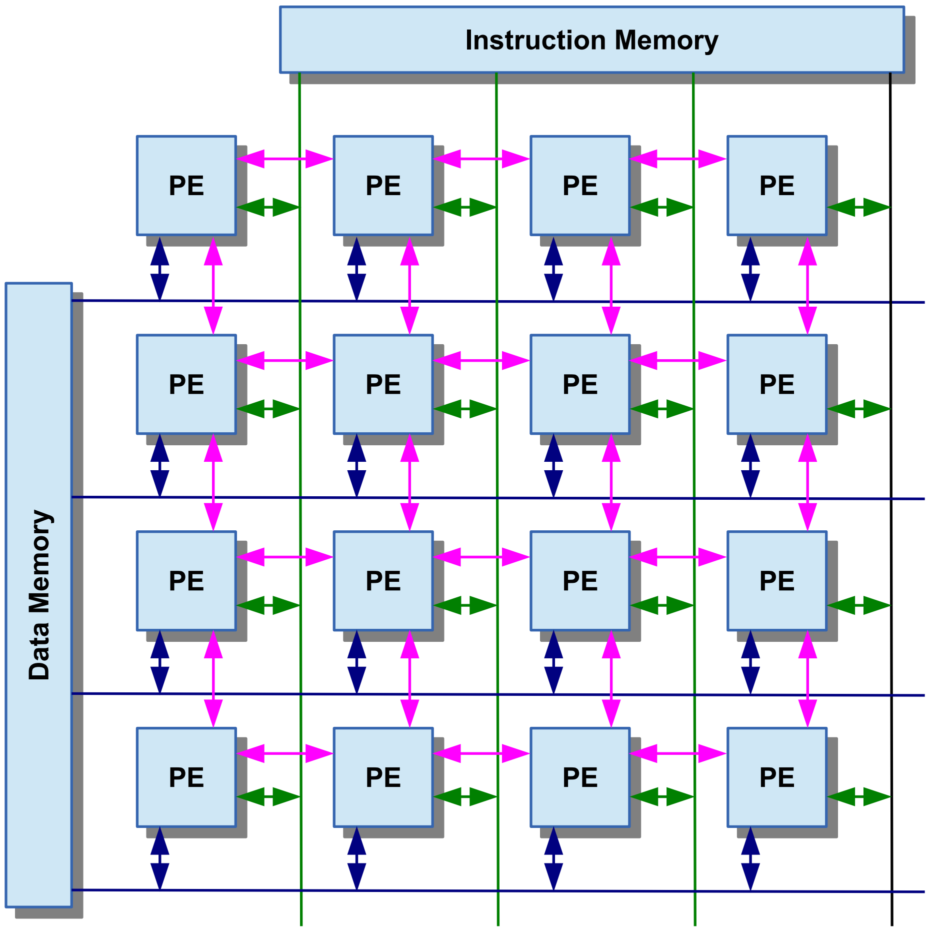


Figure 2.11. CGRA Interconnection Network.

### 2.5.5 Interconnection Network

Interconnection between PEs in CGRA is essential to leverage spatial and temporal locality. As shown in Figure 2.11 PEs are connected through a mesh network. Each PE can read from output of its 4 neighboring PEs at every cycle.

All PEs in CGRA receive an instruction from instruction memory at every cycle. PEs in a row share data bus. At a given cycle, only one PE can write to data bus. For simplicity, we assume that a PE at row 0 has an input from PE at row 3. This is the case for PEs at column 0 too. In fact, we assume a torus connection between PEs to simplify compiler design.

### 2.5.6 Control Unit

Applications execute in phases and often just a few phases or regions contribute most to the execution time. Those regions are usually composed of loops and it is the acceleration of those loops that significantly reduces the execution time of an application. The execution of a loop nest consists of three phase in CGRA: Prolog, Kernel, and Epilog. The length of those regions are sent to CGRA controller. It starts with executing prolog. Once the prolog region is complete, the kernel execution is initiated which continues in a repetitive manner until it is completed.

A PE is connected to the controller which checks the condition of the loop. For instance, many loops are expressed with a number of iteration that is checked at the beginning or end of each iteration. The PE that checks the end loop condition, simply compares the number of executed iterations with the number of iteration loop has to be executed in the program.

Once execution of the kernel is over, control unit fetches the instruction from epilog region unit execution is over. At this point, controller is responsible for interrupting the main processor. The corresponding interrupt service routine would hand the execution to the program on the main processor.

In addition to those roles, control unit is responsible to stop and resume the execution in CGRA when a cache miss occurs due to a load operation. In addition, the counter registers in rotating register files are controlled by this unit.

Chapter 3

# MAPPING PROBLEM

As opposed to general-purpose processors that only instruction set is visible to the compiler, the micro-architecture details of a CGRA is exposed to a CGRA compiler. This enables compilers to optimize applications for underline CGRA and take advantage of interconnection between PE to maximize performance. In CGRAs, the mapping of operation of the input application to resources on CGRA is static because operations are assigned to PEs at compile time. Developing efficient application mapping techniques is one of the most important problems in CGRAs.

Over the last twenty years, CGRA design have been an active field of research. However, without automated and efficient application mapping techniques, widespread use of CGRAs is not feasible. Recently, CGRA research community has focused on developing effective CGRA compilers.

Several CGRA characteristics contribute to the difficulty of programming CGRAs. Operations must be scheduled for several (sometimes dozens) of PEs, and naturally the operations must meet the data dependency requirements of the computation. The interconnection among the PEs are often sparse, leading to difficulty in routing operands between PEs. Often, CGRAs consist of resources (e.g. multipliers) that are shared by the PEs, so resource conflicts must be resolved. In addition, all of these design elements vary among different CGRA designs, so creating a universal compiler that can successfully map applications to multiple CGRAs is further complicated.

The most important problem, by far, is to effectively accelerate compute intensive kernels of an application on CGRAs. In this chapter, the problem of mapping loop kernels onto CGRA is studied. First, a background is presented along with a review of state-of-the-art. Then, the mapping problem is formulated and its complexity is studied.

## 3.1 Background

### 3.1.1 Kernels

Applications execute in phases and often just a few phases or regions contribute most to the execution time. Those regions are usually composed of *loops* and it is the acceleration of those loops that significantly reduce the execution time of an application. *Software pipelining* is a classical technique to accelerate loops by reordering the instructions [65]. *Modulo scheduling* [94] is a form of *software pipelining.* The goal in modulo scheduling is to overlap the execution of successive iterations of a loop to minimize the execution time. The performance metric in modulo scheduling in the time interval between initiating two successive iterations of the loop, referred to as *Initiation Interval* or $II$.

### 3.1.2 Input Representation

Control flow graph or CFG is an excellent intermediate representation of different code segments, used extensively in compilers for optimization purposes [85]. Compilers, generally break an input program into a set of basic blocks with few arcs connecting those blocks to each other. From compiler perspective, a basic block is a sequence of instructions with one entry point at the beginning of the basic block and one exit point at the end of it. In other word, in this representation, no control flow instruction can jump into a middle of a basic block and no control flow instruction in middle of

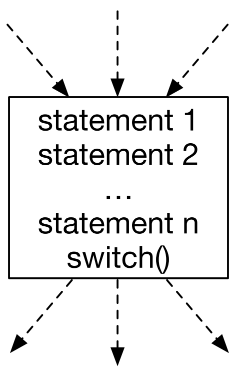


Figure 3.1. Basic Block Highlevel View.

a basic block can jump outside of the block other than at exit point of the block as shown in Figure 3.1.

Note that an entry point of a basic block can be connected to exit point of a set of basic blocks. Additionally, the exit point of a basic block can be a branch. An exit point of a basic block that represents a conditional clause, for example, can be connected to multiple basic blocks representing various possible outcomes of a conditional clause in a high-level programming language construct.

To schedule instructions within a basic block, compilers generally construct an acyclic data dependency graph called data acyclic graph (DAG) [85]. Consider the snippet of code shown in Listing 3.1. A CFG is constructed for this code in Figure 3.2. Without any optimization, three basic blocks are required to represent this loop. Loop counter, $i$, is initialized in $BB1$, loop condition is checked in $BB2$, loop body is executed $BB3$ where loop counter is incremented as well.

Listing 3.1. An example of CFG representation.

```
for(i=0; i < 100; i++) {
  C[i]=A[i]+B[i];
}
```

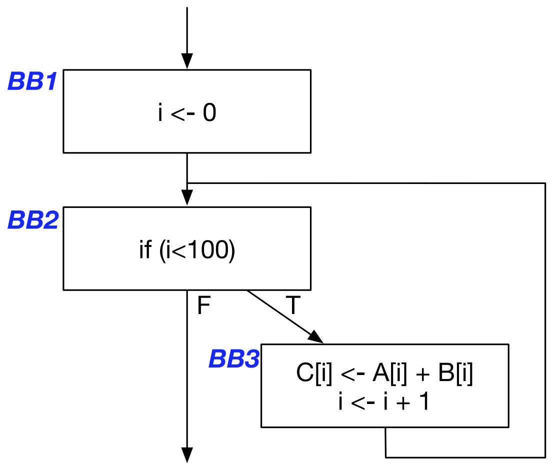


Figure 3.2. A CFG Representation of Loop in Algorithm 3.1.

A loop with sufficiently large number of iterations with high computation demand is generally selected for CGRA acceleration. A selected loop for CGRA acceleration is represented as a CFG. As opposed to traditional scheduling in compilers where each basic block is initially scheduled independently, a CGRA compiler schedules all instructions within a selected loop as a whole. This is accomplished by constructing a data dependency graph called data flow graph (DFG) from CFG representation of the loop. As opposed to a DAG, cyclic dependencies may be present in a DFG. This makes the scheduling and mapping of a loop onto a CGRA challenging. In Figure 3.3, a DFG constructed from CFG shown in Figure 3.1 is presented. In this figure, the node labeled $A$ represents loading from $i^{th}$ index of array $A$, $B$ represents loading from $i^{th}$ index of array $B$, $C$ represents storing to $i^{th}$ index of array $C$. Node $i$ represents $i \leftarrow i + 1$ operation and the node labeled $IF$ represents instruction that checks exit condition of the for loop.

A DFG $I = (V_I, E_I)$ is formed from a set of nodes and a set of arcs. In this structure, a node represents an operation or an instruction in single assignment (SSA) [85] form

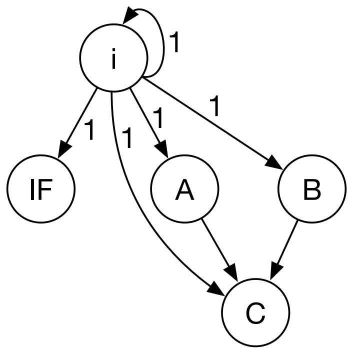


Figure 3.3. DFG Representation of CFG Shown in Figure 3.1.

of the loop. An arc from node $u$ to $v$ represented by $(u, v) \in E_I$ implies that operation $v \in V_I$ requires the output of operation $u \in V_I$ as an input.

There is a weight associated with each arc, represented by $e_{(u,v)}$. This weight represents iteration distance between dependent instructions in a loop. For instance, when $e_{(u,v)}$ is 0, there is a data dependency between $u$ and $v$ at the same loop iteration. The weight of arc from node $A$ to node $B$ in Figure 3.3 is 0. On the other hand, when the weight of an arc is a positive number, it implies a data dependency between instructions across loop iterations. For example, the weight of arc from node $i$ to node $IF$ is 1 indicating that the $IF$ instruction at iteration $j$ reads the output of instruction $i$ updated at iteration $j - 1$. Note that in a for loop, counter is incremented at the end of an iteration, thus, any instance of using that counter within loop body reads the value of $i$ updated at previous iteration [21].

Let $u_i$ represents operation $u \in V_I$ at $i^{th}$ iteration of the loop. When $e_{(u,v)} = k$ where $k \geq 0$, it implies that $v_i$ has an input from $u_{i-k}$ [22]. Note that $k$ cannot be negative and this relation is only defined when $i \geq k$.

[21]In case of $1^{st}$ iteration, it reads the value of loop initialization.

[22]operation $u$ at $(i-k)^{th}$ iteration.

Finally, control dependencies, the ones that determine the flow of the execution, are converted into data dependency using predication transformation. (details will be discussed in Chapter 6).

Once nodes are created, data dependency between pair of nodes has to be discovered. An arc $(u, v)$ is added to $E_I$ if the output register [23] of node $u$ is an input operand of node $v$. By traversing through the basic blocks once, the iteration distance between dependent operations are discovered. Consider $(u, v)$ in a basic block. If in that basic block $u$ appears first, the weight is 0; otherwise, it is 1 because that dependency is extended across two successive iterations of the loop. Since every register can only be assigned once, when $v$ appears first in a basic block before $u$, it implies that $v$ is reading output of operation $u$ produced in the previous iterations.

For an arc $(u, v)$ when $u$ and $v$ are in different basic blocks of a loop, the order in which their basic block appears in the loop determines the weight of this arc. If there is a path from basic block of $u$ to $v$ without passing the last basic block of the loop, referred to as loop latch, the weight is 0. Otherwise it is 1. A simple DFG constructed from CFG shown in Figure 3.1 is presented in Figure 3.4(a).

### 3.1.3 Modulo Scheduling

In this section, we present several problems associated with modulo scheduling and mapping a loop onto a CGRA. Consider the DFG shown in Figure 3.4(a). We would like to map this DFG onto a $2 \times 2$ CGRA shown in Figure 3.4(b). To better visualize this example, PEs on the CGRA are shown in linear form in Figure 3.4(c).

The first mapping is presented in Figure 3.4(d). In this mapping, it takes 4 cycles

[23]Note that all instructions are in SSA form.

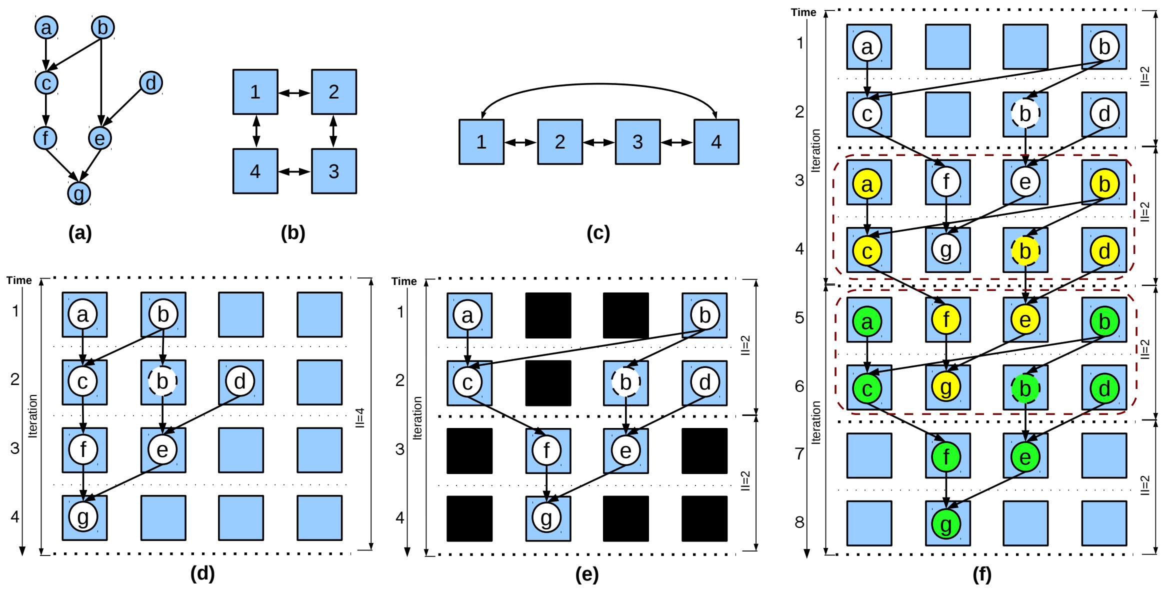


Figure 3.4. (a) An input DFG, (b) A $2 \times 2$ CGRA, (c) The same CGRA shown in linear form, (d) A valid mapping of the given DFG on CGRA with iteration latency $= II = 4$, (e) Another mapping for the given DFG with iteration latency$= 4$ and $II = 2$, lower $II$ is achieved because two iterations of the loops are executed simultaneously. Dark PEs are used to execute operation from other iterations of the loop. (f) Detail of execution overlap of three successive iterations of the loop. Nodes from first iteration are white, from second iteration are yellow, and from third iteration are green.

to execute one iteration of the loop. Mapping starts at cycle 1 when nodes $a$ and $b$ are mapped to $PE_1$ and $PE_2$. At the next cycle, operations $c$ and $d$ are executed at $PE_1$ and $PE_3$. Because the output of operation $b$ is to be used in cycle 3, $PE_2$ retains its output at cycle 2 to hold the output of operation $b$ executed at cycle 1. We refer to such operation as *routing*.

At cycle 3, operation $f$ and $e$ are executed on $PE_1$ and $PE_2$. Finally, $PE_1$ executes operation $g$ and the execution of one iteration of this loop is completed. The next iteration of the loop can be initiated at cycle 5. This implies that $II$ in this mapping is 4.

$II$ is proportional to execution time and is inversely proportional to performance.

We wish to minimize the execution time, hence, the goal in modulo scheduling is to minimize $II$. It is possible to increase the performance by $2X$ only by allocating different resources to execute operations. This resource allocation is referred to as *placement* in CGRA mapping literature. In Figure 3.4(e), another mapping of input DFG to input CGRA is presented. At cycle 1, operations $a$ and $b$ are executed on $PE_1$ and $PE_4$. In the next cycle, $c$ and $d$ are mapped to $PE_1$ and $PE_4$. $PE_3$ routes operation $b$. At cycle 3, operations $f$ and $e$ are executed at $PE_2$ and $PE_3$. Finally, $PE_2$ executes operation $g$ to complete one iteration of the loop. PEs shown in black color are used to execute operation from other iterations of the loop.

Similar to the previous mapping, it takes 4 cycles to execute one iteration of the loop. $II$, however, is reduces to 2. To better illustrated this $II$, the execution of three consecutive iterations of this loop is presented in Figure 3.4(f). Note that it is a snapshot of execution when mapping shown in Figure 3.4(e) is used. Operations in the first iteration of the loop are colored white, second are yellow, and the third iteration are green. As it is shown in this mapping, at cycle 3, $PE_1$ and $PE_4$ are not used to execute any instruction of the first iteration. Thus, it is possible to utilize them to execute operation $a$ and $d$ of the second iteration. Any resource that is needed to carry out the execution of $c$ and $d$ ($PE_1$ and $PE_4$), as well as the routing of $b$ ($PE_3$) in the second iteration of the loop are free at cycle 4. Therefore, the execution of the second iteration of the loop can be completed without any resource conflict with the execution of the first iteration of the loop.

Although the latency of completing one iteration of the loop is unchanged, a new iteration of the loop can be initiated every 2 cycles. Hence, the throughput and performance are improved by $2X$. In modulo scheduling, as is shown in this example, operations from successive iterations of the loop are executed together. Before formally defining the mapping problem, let's review few characteristics of the problem which make the mapping different from existing definitions [18, 78] in literature.

First, to define the mapping problem, it is important to precisely define range and domain in mapping function. Previous studies [18, 78] defined mapping a function from operations in DFG to resources in CGRA. However, this definition is to restrictive. Consider the mapping depicted in Figure 3.5. We wish to map Figure 3.5(a) onto CGRA Figure 3.5(b) at the minimum $II$. The mapping shown in Figure 3.5(c) is the best [24] possible mapping. This mapping is achieved because operation $b$ is mapped onto two PEs, $PE_1$ and $PE_1$, at the same cycle. We refer to this as *re-computation*.

Re-computation is different from routing. A routing PE only copies one of its inputs to the output register. In this mapping, however, operation $b$ is indeed executed twice. At cycle 1, $PE_2$ issues operations $a$. Both $PE_1$ and $PE_3$ read the output register of $PE_2$ and execute operation $b$. The output register of $PE_1$ is read by $PE_1$ to execute operation $c$ at cycle 3. At the same cycle, $PE_3$ reads its own output register and executes operation $d$. Finally, at cycle 4, operation $e$ and $f$ are issued by $PE_2$ and $PE_4$ to complete the execution of one iteration.

$II$ in this mapping is 2. This is because the first iteration of the loop is initiated at cycle 1 and the second iteration is initiated at cycle 3. This is accomplished only because of the re-computation through which at cycle 3 and 4, all the resources to initiate the execution of the next iteration of the loop become available. Those resources are colored black in Figure 3.5(c).

This example clearly shows that the mapping is not simply a function from operations in DFG to PEs in resource graph. This is counter-intuitive because we expect $II$ to increase as the number of operation executed by PEs increases.

[24]The one with the minimum $II$.

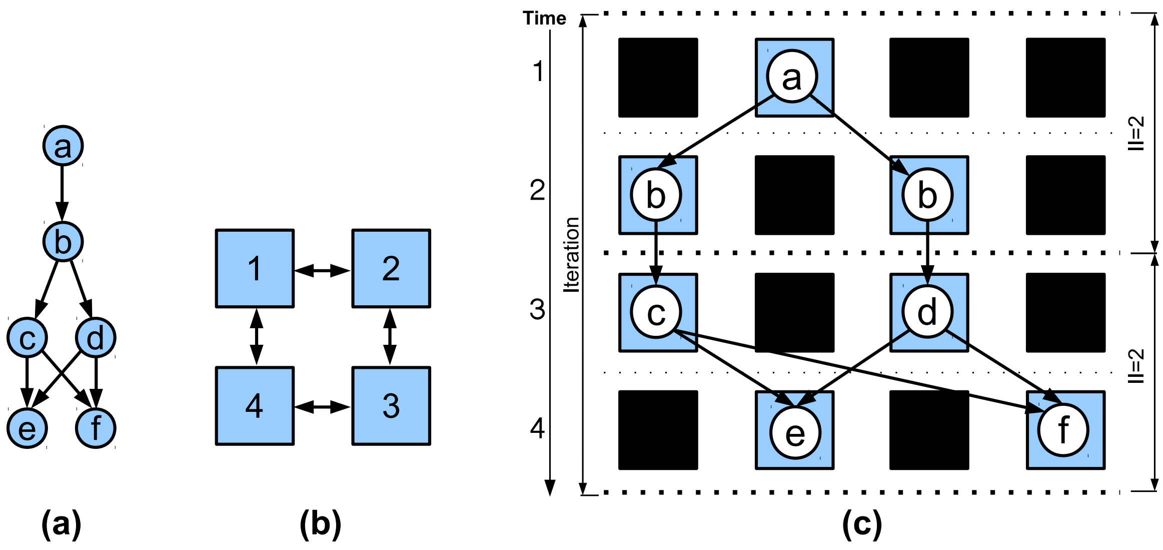


Figure 3.5. (a) an input DFG, (b) a $2 \times 2$ CGRA, (c) A mapping of the input DFG into the CGRA at the minimum $II$. One iteration of the loop requires 4 cycles to execute and $II = 2$. The minimum $II$ is achieved because operation $b$ is executed twice. Without re-computation, mapping at $II = 2$ is impossible.

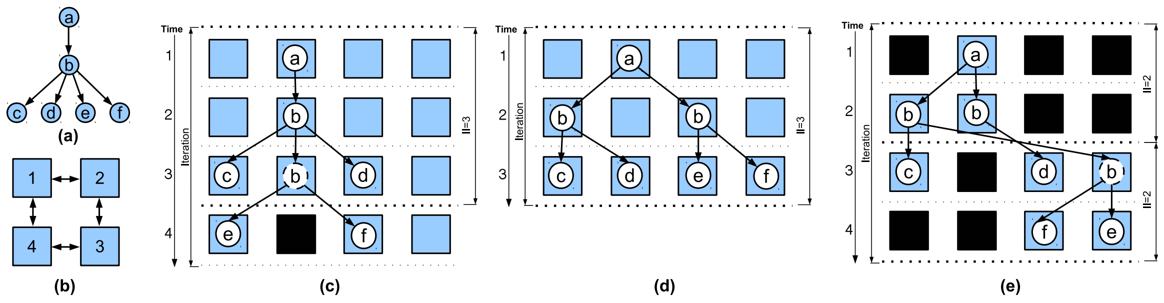


Figure 3.6. (a) an input DFG, (b) a $2 \times 2$ CGRA, (c) The same CGRA shown in linear form, (d) a valid mapping of the given DFG on CGRA with iteration latency $=II=$ 4, (e) another mapping for the given DFG with iteration latency= 4 and $II = 2$, lower $II$ is achieved because two iterations of the loops are executed simultaneously, (f) Detail of execution overlap of three successive iterations of the loop. Nodes from the first iteration are white, from second iteration are yellow, and from third iteration are green.

An important aspect of this problem is that only permitting routing or re-computation cannot guarantee the optimality of a solution. For instance, in Figure 3.6, a mapping is presented where neither routing nor re-computation alone could yield to a mapping at minimum $II$. The first mapping, depicted in Figure 3.6(c), uses routing to map operation $b$. The minimum $II$ that can be achieved by utilizing routing scheme is $II = 3$. Figure 3.6(d) shows another mapping that utilizes re-computation scheme. This mapping is no better than the former one because $II = 3$ while the latency of executing one iteration is reduced from 4 to 3. In the third mapping presented in Figure 3.6(e), routing and re-computation schemes are combined to map operation $b$. Even though the number of nodes and the latency of executing one iteration is increased, $II$ is reduce to 2 cycles.

**Definition 1** *A node $v$ in the resource graph $R_{II}$ is said to be associated with an operation $i$ in the DFG if $v$ is a PE executing* [25] *operation $i$ at cycle $t$. For instance, in Figure 3.6(d), $PE_1$ and $PE_2$, at time 2 and $PE_4$ at time 3 are associated with $b$.*

**Observations:** Using the examples presented before, we make two important observations that are essential to the general problem formulation.

1. Every node in the DFG is associated with at least one node in the $R_{II}$.
2. For every edge $(i, j)$ in the DFG, all nodes in the $R_{II}$ that are associated with $j$ have a path $P$ from a PE associated with $i$.
3. All intermediate nodes in $P$ are PEs associated with $i$ (routing $i$) or $j$ (routing $j$).

## 3.2 Related Works

Loop acceleration has consistently been an attractive research problem in compiler community. Several problems have to be addressed to achieve a reasonable acceleration factor in loops [85]. The fundamental goal in all of these problems is to

[25]Execution here includes routing too.

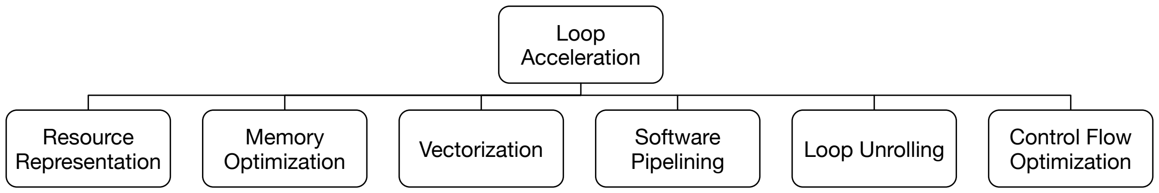


Figure 3.7. Important problems that need to be addressed for an effective loop acceleration.

minimize execution stalls and to maximize the utilization of computational resources. These problems include memory optimization, loop unrolling, vectorization, software pipelining, and an efficient resource representation as shown in Figure 3.7.

The goal of memory optimization is to layout loop variables in such a way that they can be accessed with highest bandwidth with respect to loop access pattern [77, 95, 101]. Several techniques are available such as polyhydral model [10, 77] that captures the loop access pattern and transform the loop to maximize iteration distance between memory dependencies [7, 77].

Loop unrolling is another important optimization which unrolls a loop for a number of time with aim of increasing resource utilization in underline processor [14, 15, 39, 98]. For a limited set of loops, vectorization is an orthogonal optimization to loop unrolling. Fo such loops, operations in several consecutive loop iterations are transformed into a single vector operation. Auto-vectorization in loops is an active research problem [7, 62, 104] and several such techniques are implemented in commercial compilers [35, 75].

Control dependencies, either within operation in a loop or within consecutive loop iterations, limits the performance by enforcing a serial dependencies between loop instructions, which in turn limits instruction level parallelism and iteration level parallelsim. Several techniques are available to minimize this effect including converting control flows to data flow through predication [73, 74].

When an underline architecture is exposed to a compiler, resource representation plays an important role in scheduler. Several representations has been proposed for resources such as table representation [94], and graph representation [48, 79]. All of these problems need to be addressed in a compiler to be able to schedule the instructions effectively. The goal in scheduling is to order instructions such that at run-time, the loop execution time is minimized.

Perfect pipelining [3] is among the earliest scheduling technique to accelerate loop by unrolling a loop for a number of time. With the advent of Very Long Instruction Word (VLIW) architectures, extracting instruction level parallelism beyond instructions within a basic block became necessary to utilize functional units well. Software pipelining [65] and modulo scheduling [94] have been shown to effectively accelerate loop execution in VLIW architectures by overlapping the execution of successive loop iterations. Lam [65] formally studied the problems associated with software pipelining a loop such as scheduling cyclic dependency between loop iterations, replicating variables in the loop to avoid resource congestion, and supporting loops with conditional statements. Earlier work which proposed algorithmic technique to software pipeline loops include [57] and [109].

Most notably, Rau [94] proposed an effective iterative modulo scheduling algorithm for VLIW architectures. This heuristic relies on a modulo reservation table (MRT) to keep track of hardware resources during scheduling. This technique follows a top to bottom scheduling policy. As such, operations without any predecessor are listed as ready for scheduling. As the algorithm proceeds, it adds more operations to ready list whose predecessors are already scheduled. Operations in this list are sorted based on some factors. Higher priority is given to operations within recurrence cycle paths. Later works devoted more effort in developing better heuristics to assign priority to nodes, and taking into account resource constraints [71].

An important problem that designers faced during 90's was to increase the number of functional units in VLIW processors. Scaling up such architecture requires increasing the number of read and write ports in register file because there is a central register file that is shared among all functional units in VLIW processors. An increase in the number of register file ports is extremely challenging. Silicon area and power consumption to realize such increase in the number of register file ports increase significantly. In addition, the latency of register file read and write increases substantially by increasing the number of read and write ports. To alleviate this problem, clustered VLIW processors [5] have been proposed. In such design, a set of functional units form a cluster in a VLIW processor with a shared register file. Register file, however, are not shared across clusters. Such clusters then replicated to form a large processor. Thus, the complexity of register file design is kept relatively low while the number of functional units are scaled up beyond what could have been achieved with a shared register file structure. Clusters, in this design, communicate through an inter-cluster bus. This was a move toward architectures more like CGRAs.

While such design is excellent in scalabity, it poses yet more challenge in software pipelining loops. It is because a cluster allocation decision has to be made during software pipelining a loop. To cope with this problem, DFG of the loop is partitioned with the objective of minimizing communication between partitions, each to be assigned to one cluster [4] for execution. Sánchez and González [96] presented an algorithm in which operations are assigned with a cycle and a cluster with the goal of minimizing cache misses. Such algorithms have been shown to be effective in architectures where all clusters can communicate with another.

However, as cluster architectures evolved which only implement local communication between clusters for scalability purposes, modulo scheduling turned out to be even more challenging. It is because multiple decisions has to be made at the same time when an operation is selected for scheduling. In the meantime, an assignment of an operation to a cluster has to be verified for routing between that operation and its dependent ones. When a failure occurs, backtracking decision, such as how far and when to backtrack, whether a failure occurs due to routing failure or resource congestion, substantially effects the mapping algorithm outcome and running time.

Distributed Modulo Scheduling (DMS) [30] introduced backtracking capability to IMS [94] which simultaneously assigns a cycles and a cluster to operations. In this algorithm, a decision to assign a cycle or cluster to an operation may require to unschedule previously scheduled operation. In addition, if a predecessor of an operation is assigned to a different cluster, while scheduling, *mov* instructions are inserted to transfer data from one cluster to another.

A hierarchical scheduling approach is taken by [81] and [29] where a loop DFG is partitioned and assigned to tiles (clusters) with the goal of minimizing inter-cluster communication. Each partition is then modulo scheduled with the goal of minimizing waits for dependent instructions assigned to different tiles.

As tile clusters such as CGRAs become dominant (mesh line interconnect), software pipelining algorithms have evolved to take into account challenges introduced by this structure. In general, existing modulo scheduling algorithms take one of the following approaches: integrated approach or decomposed approach as shown in Figure 3.8.

With an integrated mapping policy, the process of assigning time, resource, and satisfying data dependencies for a selected operation is taken place at once in an node-by-node fashion. In contrast, when a decomposed policy is taken, the mapping problem is decomposed into several simpler and well-understood sub-problems. Each problem, then, is solved with a well known objective. A majority of CGRA mapping techniques implement an integrated mapping policy.

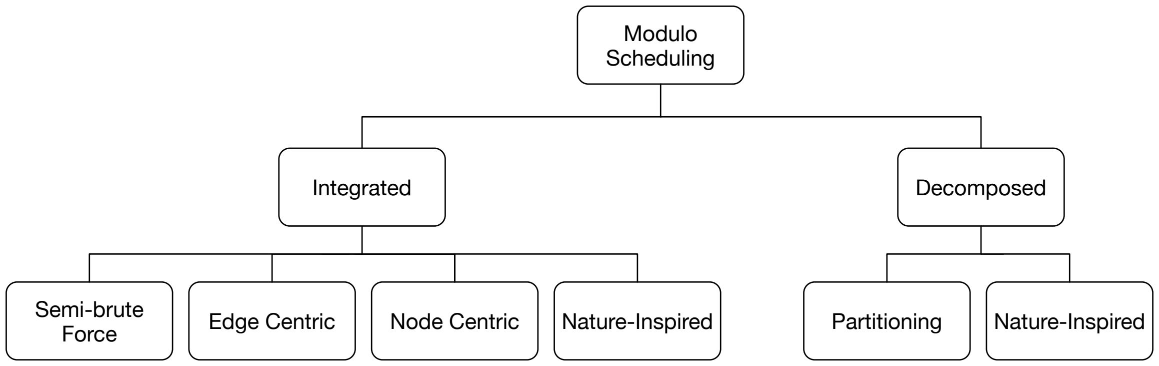


Figure 3.8. Policies taken by existing modulo scheduling algorithms: integrated and decomposed.

Mei *et al.* [78, 79] was one of the pioneers to study CGRA mapping problem. They proposed the construction of a resource graph which they named Modulo Routing Resource Graph (MRRG). They noticed that the mapping problem is indeed mapping loop DFG onto MRRG graph and a major challenge in mapping is to satisfy data dependencies between operations on a sparely connected resource graph. However, instead of using a compact representation of resources in MRRG, extensive architectural details are available in MRRG. This is because all problems including control signal generation for routing was intended to be solved at once in their technique.

This paper presents a simulated annealing [61] based mapping technique that is inspired from place and route algorithm [27] in FPGAs. In this technique, $MII$ is approximated first and used to construct a resource graph. A time and a resource is assigned to all operations respecting order of operations and their data dependency. However, it is permitted to over utilize a resource with multiple operations at a given cycle at the beginning. Obviously, the initial mapping is unacceptable and cannot be realized in practice. However, as the algorithm proceeds, it attempts to remove over-utilization of resources while preserving data dependency between operations.

This is accomplished by several steps in simulated annealing algorithm. Nodes are selected one by one and based on the current temperature, they can be assigned to new execution cycle and resources. A cost function is introduced to represent the overuse of resources. Along with this function and a gradual temperature reduction, operations are moved until an acceptable solution is found. An acceptable solution is the one where all resources execute one operation at a cycle. If an acceptable solution is not achieved within a timing budget, $II$ is increased and the algorithm starts from the beginning. Experimental results demonstrate that loops with few operations can be effectively mapped using this algorithm [79] but at a long convergence time. Details about cost function and temperature reduction of this technique can be found in [105].

Similar mapping strategy using simulated annealing search technique is taken in [48] with different cost functions. Particle Swarm Optimization (PSO) [59] is another search heuristic which mimics social behaviour of the bird flocks. This searching strategy is used [33] to iteratively assign operations to resources and associate a cost to a mapping. Mappings are evolved in time, based on permitted operations in PSO to decrease mapping cost. The major problem with such searching schemes is long convergence time.

Bansal *et al.* [8] focused on the problems posed by sparse connectivity between PEs when mapping loops on CGRAs even without software pipelining the loop. They realized that connectivity is by far the most important problem. Therefore, they applied different heuristic cost functions for associating a priority and an affinity of nodes, associating a cost to mapping a node to a PE considering the connectivity of CGRA. It uses a list scheduling based technique [81] which verifies connectivity (by routing dependencies between mapped operations) while mapping an operation.

A node centric approach is taken by Park *et al.* [91]. Mapping a DFG onto a CGRA is expressed as drawing DFG onto the CGRA space in [91]. They presented an algorithm inspired from [70] to map DFGs onto a 3D grid of PEs stacked up $II$ times. A top to bottom scheduling policy is taken where all operations are first associated with a level. Starting from operation at level 0 (operations without any predecessor), operations are assigned to a resource. An MRT holds and tracks resource status, connectivity between resources, and available register at each resource during $II$ cycles. When operations are assigned to resources, data dependencies from predecessors are also satisfied.

Various cost functions are introduced in [91] to represent the costs associated with routing data dependencies for an operation when it is assigned to a resource, including the cost of using that particular resource. Since it is possible to exhaust all resources for routing purposes while no resource is left to be assigned to subsequent operations, a heuristic technique is used to skew the schedule to reduce resource congestion. Through evaluation, this paper shows that routing can quickly exhaust all resources which are required to be used in subsequent operations. Therefore, using resources likely to be used in subsequent operations are associated with higher costs to avoid such problems. Developing an effective technique to avoid resource starvation without knowledge about subsequent instructions, however, is extremely difficult, if possible at all.

This paper also presents a simplified model for resource graph compared to MRRG presented in [79]. MRRG exposes much more information to scheduler than needed to make a mapping decision. Resource graph should only expose sufficient information to guarantee availability of resources for routing. Actual signal generation for routing can be delayed if construction of one can be guaranteed during mapping. Such resource graph simplification significantly decreases the time to find a valid mapping as demonstrated in the paper.

A similar node-by-node approach is taken in [24] while different cost functions and different scheme to prioritize nodes are introduced. In addition, this algorithm permits backtracking in the algorithm and unmapping operations that already assigned to resources. In addition, once all local registers at a PE are exhausted, memory operations are inserted in DFG to spill intermediate variables.

Chen and Mitra [18] take an exhaustive searching approach. They made an interesting observation about the resources used for routing data dependencies. When a node-by-bode mapping policy is taken, routing data dependencies for operations with multiple successors may lead to an ineffective resource utilization. This occurs because the output of a resource executing such operation may be sent to multiple resources through disjoint paths. This is an ineffective resource usage because the same data is carried out in those disjoint paths. Therefore, sharing resource to establish path between producer and multiple consumers can significantly result in better resource utilization.

This heuristic has been extensively used in [18]. Nodes are selected for mapping based on their priority in this technique. For a selected node, the set of predecessor and successor operations that are already mapped to resources in formed. Using this set, the set of potential resources that the selected operation can be mapped is discovered. Each potentially mapping in recursively expanded and the result is tested to verify if it forms a minor relation [22] between input DFG and resource graph. If at any step, it fails to find any potential mapping for an operation, the routing between its predecessors and selected operation are expanded to increase the number of potential mapping resources for the operation.

Various heuristics are used to minimize the mapping time. For instance, a number of functions look ahead and verify if a mapping will fail in subsequent steps when required resources for subsequent operations are exhausted. In addition, availability of paths between dependent operation when both successors and predecessors of a selected operation are mapped is checked at each step. Experiments results show that this technique can quickly map loops at $II$s close to $MII$. In addition, authors implemented a version of EPIMap technique, which will be presented later, and found that their technique and their EPIMap implementation map loops at close $IIs$ [18]. Their implementation only limited to resource allocation scheme presented in [40] not the whole mapping.

Park *et al.*[92] discovered that the main reason that modulo scheduling algorithms, in general, and their earlier work [91] in particular, cannot efficiently utilize resources on CGRA is that routing data dependencies between operations is a secondary problem while making a mapping decision. Therefore, they took an edge centric mapping approach. In contrast to earlier node centric approaches, EMS[92] focuses more on mapping edges in DFG onto edges in resource graph, thereby assigning operations to resources by first tackling feasibility of routing between mapped operands.

EMS exhibits an interesting characteristic by taking this mapping approach. First, when an operation is assigned to a PE, it is guaranteed that data can be routed between one of its producers to the resource the operation is to be executed at. Therefore, routing failure is less likely to occur. In addition, this guaranteed path is established in shortest distance between producer and consumer, thereby occupies minimum number of resources.

To present a global view of resources and the overhead associated with making a mapping decision, a cost function is introduced for estimating the cost of satisfying the rest of data dependencies when an operation is assigned to a resource. Note that at each step of this algorithm, an arc in DFG is selected to be mapped to a possibly minimal set of edges in resource graph. Thus, once making a decision, a data dependency between one producer and the selected operation is already routed while other data dependencies have to be estimated to make a decision about accepting or rejecting a potential mapping of a node to a resource. This cost function associates a large overhead for using sparse resources such as memory capable units or multiplier for routing purposes. Thus, proactively avoiding resource starvation for subsequent instructions.

EMS initially simplifies DFG by categorizing nodes into two sets: I) The set of important nodes such as the one within recurrence cycle of the loop and, II) The set of high fan-out nodes which potentially exhaust resources during mapping to route data dependencies. Each node in the second set along with its consumer would be represented by only one node in simplified DFG. Then, a scheduling window for each operation is established between earliest and latest cycle that operation can be scheduled.

Ansaloni *et al.*[6] also incorporated a search window for scheduling operation but they used term slack instead. Arcs in DFG are selected to be assigned to a set of arcs in resource graph in EMS, thereby assigning operations to a cycle and PE in resource graph. Experimental results show that EMS maps loops at $II$s that are around 2% higher than [79] at a fraction of the compilation time.

$MII$ in some loops is limited by recurrence dependencies between operations. To minimize $II$ in mapping, it is crucial to schedule those operation such that the latency of recurrence cycle is minimized. This importance factor is studied in [60] and [89] and mapping algorithms are adjusted to map those operations upfront. Experimental results demonstrate that for loop with recurrence cycle, when resources are assigned to operation in that cycle, it is more likely to find mapping at $MII$.

In all above-reviewed techniques, a mapping decision is made at once for each node where nodes are selected by some priority. Routing data dependencies are also taken care of while mapping decision is being made. CGRAs usually implement a sparse interconnection between functional units. The problem associated with this mapping policy is that resources necessary to map subsequent operations are quickly exhausted for routing purposes. The cost functions associated with routing in most of those algorithms is proportional to the number of resources used for a routing. However, it cannot be foreseen whether a resource used for a routing might be needed to map subsequent operations. It is very challenging to avoid this problem, thereby mapping failure is very frequent when such mapping policy is taken. In all of these techniques, $II$ is increased after a number of failures in mapping attempts at a given $II$. This failure budget varies by characteristics exhibited by DFGs and it is very challenging to adjust this budget based on input DFG.

An alternative approach is to decompose the mapping problem into several well-studied problems. Each problem can be addressed effectively independent of other problems with a well-defined objective. It has been shown that such policy is very effective in compilers and synthesis tools [82]. It is, however, still challenging to adjust objective of sub-problems that effectively lead to optimizing the main problem objective. In existing literature, the mapping problem is decomposed into scheduling, resource allocation, routing and partitioning.

Venkataramani *et al.* [106] presented a mapping technique for MorphoSys [67] architecture. A mapping is accomplished by scheduling operations first and then placement (binding). Once operations are assigned with a time, it is assumed that a valid resource allocation can be made, due to rich interconnection between PEs in MorphoSys. Afterward, scheduled DFG is partitioned into sets of maximum 16 operations. The goal of partitioning is to minimize communication between sets, and to minimize execution time.

The mapping problem is partitioned into scheduling, placement, and routing in [31]. The following steps are taken until a valid mapping is found in this algorithm. Operations are initially scheduled respecting order by taking into account data dependency between operations as well as available resources. However, the actual resource allocation is deferred after all instructions are scheduled. In the next step, a simulated annealing search technique  [61] is implemented to assign operations to resources in resource graph. During placement, operations are moved between resources as well as time (slack of schedule windows extract at scheduling step).

If placement fails due to violating data movement latency between operations, *padScheduling* heuristic is used. This heuristic permits operations to be rescheduled beyond slack windows to meet data movement latency. When all operations are associated with a resource, data dependency between operations is routed using heuristics presented in [69] and [76]. If routing fails, $II$ is increased and SPR [31] starts from the beginning.

While this technique partitions the problem into distinct problems with different objectives, it suffers from two important problems. The placement is slow and takes long time to converge not only because it is based on simulated annealing, but also because the search space for operations include resource and time dimensions. Second, a routing failure does not necessarily imply that a feasible mapping cannot be made at the same $II$. However, to keep the compilation time low and avoid exhaustive search, $II$ is increased once routing fails.

While there are many algorithms that decompose the mapping problem into several subproblems such as scheduling, resource allocation, routing, register allocation, etc., a systematic decomposition approach with well defined objective at each step that justified the optimality of the approach is still missing.

In next part, the problem of mapping and modulo scheduling loops onto CGRA is studied and its characteristics are extracted. Using insight gained from problem definition, an effective mapping algorithm will be presented.

## 3.3 Problem Definition

The general problem of mapping an input DFG to a CGRA is based on the three observations stated before. Let's describe the resource graph construction first.

Given an $II$ and a CGRA, time extended resource graph denoted by $R_{II} = (V_R, E_R)$ is constructed by replicating the nodes in the CGRA, $II$ times, representing available resources from cycles 0 through $II - 1$. For every pair $(u, v)$ of adjacent nodes in the CGRA, there is an arc from (replication of) $u$ at time $t$ to replication of $v$ at time $t + 1$. Note that every node in the CGRA is adjacent to itself. In Figure 3.4(b), $PE_2$ is connected to $PE_1$ and $PE_3$. Therefore, in Figure 3.4(f), the output $PE_2$ at cycle 2 can be read by $PE_1$, $PE_2$ and $PE_3$ at cycle 3.

**Definition 2** *Given a DFG $I = (V_I, E_I)$ and a CGRA, the mapping objective is to construct a time extended resource graph $R_{II} = (V_R, E_R)$ of minimum extension for which*

1. *there exists a mapping $M : V_I \rightarrow 2^{V_R}$, where $2^{V_R}$ is the power set of $V_R$,*
2. *for every arc in $(i, j) \in E_I$, the following property holds: for each node $r_n \in R_{II}$ associated with $j$, there is a path $P = (r_1, \ldots, r_\ell, \ldots, r_r, \ldots r_n)$ such that $r_1$ is a PE associated with $i$, $r_2$ through $r_\ell$ are the same PE as $r_1$. $\ell$ is the latency of executing operation $i$. There are $k \geq 0$ PEs between $r_\ell$ and $r_{r-1}$ that are associated with $i$ and the rest are associated with $j$.*

The above formulation can be intuitively understood as follows. For a mapping to be valid, all operations must be executed (mapped) and data dependencies between operations must be obeyed. First, $V_I$ is the domain, thus all operations in $V_I$ must be executed. To ensure that data dependencies are satisfied, when there is an arc between two operations $i$ and $j$, any PE executing operation $j$ must receive the output of operation $i$ from a PE executing $i$. This is ensured when a path exists between a PE executing operation $i$ to nodes executing or routing operation $j$, when $(i, j) \in E_i$. Last, an operation $i$ with the execution latency of $\ell$ cycles that is mapped to a PE $r_1$, the path is valid if and only if all PEs $r_1$ through $r_\ell$ are same physical PE as $r_1$. Thus, the output of an operation would take at least $\ell$ cycles to become available.

Let $PE_i^t$ represents $PE_i$ at modulo cycle $t$ where $0 \leq t < II$. In Figure 3.4(e), $V_I = \{a, b, c, d, e, f, g\}$ and $V_R = \{PE_i^t | 0 \leq t < II \wedge 1 \leq i \leq 4\}$. The mappings are shown in Table 3.1.

Table 3.1. The mapping of operations in Figure 3.4(e).

| Instruction | Resource(s) |
|---|---|
| a | $\{PE_1^0\}$ |
| b | $\{PE_4^0, PE_3^1\}$ |
| c | $\{PE_1^1\}$ |
| d | $\{PE_4^1\}$ |
| e | $\{PE_3^0\}$ [26] |
| f | $\{PE_2^0\}$ |
| g | $\{PE_2^1\}$ |

This formulation is general and allows *routing*, *re-computation*, and any combination of both. It is because any instruction can be mapped into any number of resources in resource graph. Thus, an operation can be *routed* and *computed* by any number of PEs.

[26] The modulo cycle 0 represents cycle 2 at execution ($II = 2$)

## 3.4 Complexity Analysis

In this section, we study the complexity of CGRA mapping problem in general. Besides that, we study complexities associated with routing and re-computation problems.

### 3.4.1 Mapping Problem in NP-Complete

In order to establish the complexity of the mapping problem, we introduce 3-partition problem which is known to be NP-complete in the strong sense [32] and reduce 3-partition problem into a restricted mapping problem. The reduction idea is partially borrowed from [25].

**Definition 3** *3-partition. A finite set $A$ of $3m$ elements, a bound $B \in Z^+$ and a size $s(a) \in Z^+$ for each $a \in A$ such that $s(a)$ satisfies $\frac{B}{4} < s(a) < \frac{B}{2}$, and such that $\sum_{a \in A} s(a) = mB$. Can $A$ be partitioned into $m$ disjoint sets $S_1, S_2, ..., S_m$ such that, for $\sum_{a \in S_i}^{1 \leq \forall i \leq m} s(a) = B$?*

Please not that the restriction of $s(a)$ implies that the number of elements at each set must be exactly three.

**Definition 4** *(P1). Here, we restrict the number of functional unit to 2.*

**Lemma 3.4.1** *3-partition problem polynomially transforms to (P1).*

**Proof 1** *Given an instance of 3-partition problem,*
*$A = \{a_1, a_2, ..., a_{3m}\}$, we construct an instance of problem (P1). In this reduction, the set of nodes in DFG is formed from union of three sets, i.e. $T = T_1 \bigcup T_2 \bigcup T_3$. Let $L(O_i)$ be the latency of executing operation $O_i$. $T_1 = \{O_1, O_2, ..., O_{3m}\}$ where*

*$L(O_i) = s(a_i)$, $1 \leq \forall i \leq 3m$, $T_2 = \{P_1, P_2, ..., P_m\}$ where $L(P_i) = B$, $1 \leq \forall i \leq m$ and $T_3 = \{Q_1, Q_2, ..., Q_{2(m-1)}\}$ where $L(Q_i) = 1$, $1 \leq \forall i \leq 1m$.*

*Next, we form the arc set as shown in Figure 3.9. First, $1 \leq \forall i < m$, we form two arcs from $P_i$, to $Q_{2i-1}$ and $Q_{2i}$. This ensures that a valid mapping can be made only if a PE is allocated to execute an operation from set $T_2$ that takes exactly $B$ cycles. Therefore, there is only one PE available to execute a set of operations from set $T_1$. The accumulative latency of those operations has to be exactly $B$ cycles (see Figure 3.9). It is because $Q_{2i-1}$ and $Q_{2i}$ has to be scheduled after $B$ cycles at the same cycle, thus both PE resources has to be available then.*

*Second, $1 \leq \forall i < m$, we form two arcs from $Q_{2i-1}$ and $Q_{2i}$ to $P_{i+1}$. This ensures that another PE has to be allocated for executing $P_{i+1}$ immediately after $Q_{2i-1}$ and $Q_{2i}$ are executed. Finally, $II$ is set to be $m(B+1)-1$.*

*If an instance of 3-partition problem has a solution, it is easy to see that problem (P1) has a solution. Every element in set $S_i$ is associated with an element in set $T_1$ in the constructed mapping instance. Let $S_i = \{a_i, b_i, c_i\}$. We schedule the elements in $T_1$ associated with $a_i$ at cycle $(i-1) \times (B+1)$, $b_i$ at $(i-1) \times (B+1) + s(a_i)$, and $c_i$ at $(i-1) \times (B+1) + s(a_i) + s(b_i)$. Since those associated elements execute at exactly $B$ cycles, $P_i$ can be executed in the meantime at $PE_1$. $Q_{2i-1}$ and $Q_{2i}$ are to be executed on $PE_1$ and $PE_2$ at cycle $(i+1) \times B - 1$.*

*Conversely, if the instance of problem (P1) has a solution, operations of set $T_1$ must be partitioned into $m$ sets. It is because there is a chain of operations starts from $\{P_1, Q_1, P_2, Q_3, ...., P_{m-1}, Q_{2m-1}, P_m\}$. This chain takes exactly $II = m(B+1)-1$ cycles to execute and deploys one functional unit entirely. Also, at the end of each $P_i$ operation, there are two operations that must be executed before $P_{i+1}$ can be scheduled, $Q_{2i-1}, Q_{2i}$ which need both functional unit to be used at the same cycle. So PEs cannot execute any other node at that time. Thus, there is only one functional unit available*

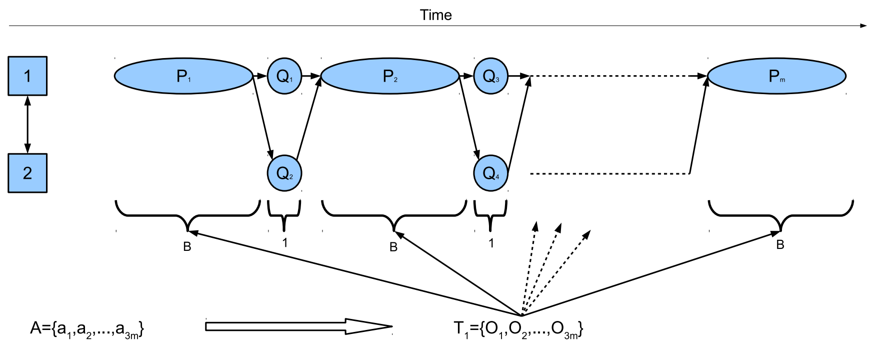


Figure 3.9. The idea behind reducing 3-partition to mapping problem. We introduce operations with latency of $B$ cycles. Those operations are connected to exactly 2 operations, whose latency is 1 cycle. Then, those nodes connected to next operation with $B$ cycles latency. This makes 1 PE available for exactly $B$ cycles.

*between $(i-1)\times(B+1)$ to $(i+1)\times B-1$ cycles for all $1 \le i \le m$ while the other functional unit is executing nodes in $T_2$. Since there is no empty time slot and execution must break every $B$ cycles, the set of tasks in $T_1$ must be partitioned into $m$ sets of $B$ cycles. It implies that an instance of 3-partition has a solution iff the constructed instance of problem (P1) has a solution.*

*Note that this proof can easily be extended to CGRA with heterogeneous PEs where only one functional unit supports operations in set $T_2$. Thus 3-partition problem can polynomially be reduced to problem (P1) either with homogeneous or heterogeneous functional units.*

**Theorem 3.4.2** *The mapping problem is NP-complete.*

**Proof 2** *We have shown that 3-partition problem can be polynomially reduced to problem (P1) which is the restricted problem of mapping problem. Thus, mapping problem is NP-complete.*

### 3.4.2 Notes on Routing and Re-computation Complexity

Throughout this section, for simplicity, we assume the input DFG does not have a self loop .

**Definition 5** *Let $G$ and $H$ be two directed graphs. A mapping $f : V(G) \rightarrow V(H)$ is a Homomorphism if $(f(u), f(v)) \in E(H) \Rightarrow (u, v) \in E(G)$. It is called an Epimorphism if it is arc surjective, i.e., every node in $H$ is an image of some node in $G$. Note that node surjective implies arc surjective [49].*

Simply stated, for every valid mapping there exists an epimorphic map $M : V_R^* \rightarrow V_I$ that satisfies certain conditions where $V_R^* \in V_R$ ($V_R^*$ is a subset of nodes in resource graph). Conversely, an epimorphic map from $M : V_R^* \rightarrow V_I$, that satisfies the same conditions corresponds to a valid mapping. Thus the optimization problem is to construct an epimorphism $M : V_R^* \rightarrow V_I$.

Now we see how the formulation works. Consider a node in resource graph (i.e. a PE) $i \in V_R^*$. Let $i' = M(i) \in V_I$. $i'$ is an operation that is mapped to node $i$ in resource graph. For example, if the node $i$ is $PE_3$ at time 2 in Figure 3.6(d), then, $i' = M(PE_3^2) = b$. Similarly, let $j \in V_R^* : \exists(j, i) \in E_R^*$, and let $j'$ be the operation that is mapped to resource $j$. For example, $j$ is the $PE_2$ at time 1, and $j' = M(PE_2^1) = a$. Then epimorphism requires that if there is an arc between $a$ and $b$, then there must be an arc between $PE_2$ at time 1, and $PE_3$ at time 2. This example illustrates how epimorphism ensures that data dependencies are preserved.

This formulation seamlessly captures routing and re-computation. Whenever we use routing/re-computation, a set of PEs in the time extended resource graph map to one operation in the DFG. For example, in Figure 3.6(c), in which the out-degree problem is resolved using routing, $PE_2$ of time 2, and $PE_2$ of time 3 are mapped to operation $b$. Since operation $a$ has an arc to operation $b$, epimorphism requires that there is at least one arc between the set of PEs that are mapped to $a$, and the set of PEs that are mapped to $b$. This is true, since there is an arc from $PE_2$ at time 1 (where operation $a$ is mapped), to $PE_2$ at time 2 (where operation $b$ is mapped). Similarly the data dependencies with operations $c$, $d$, $e$, and $f$ are satisfied.

In Figure 3.6(d), in which the out-degree problem is resolved using re-computation, a set of two PEs, $PE_1$ of time 2, and $PE_3$ of time 2 are mapped to operation $b$. In this case also, both of these nodes has an incoming arc from $PE_2$ at time 1 where $a$ is executed. Data dependencies for all other operations are satisfied as well.

Again, to use routing or re-computation, a set of PEs in the time extended resource graph map to one operation in the DFG. The only difference is the presence/absence of arc(s) between the PEs in the set. In Figure 3.6(e), $PE_1$ at time 2, and $PE_4$ of time 3 have an arc between them, so they transfer data through routing. On the other hand, in Figure 3.6(d), $PE_1$ of time 2, and $PE_2$ of time 2 do not have an arc between them, therefore the operation $b$ has to be recomputed.

We now explain the conditions. Essentially, the condition is just to ensure that the PE at which computation, or re-computation happens, receives all the input operands from its predecessor. Let $i' = M(i) \in V_I$. $i'$ is the operation that is mapped to PE $i$. Let $j \in V_R^* : \exists (j,i) \in E_R^*$. Thus, if we consider a resource node (i.e. a PE) $i \in V_R^*$. Then there are two possible cases. Either $i$ is a routing node and has an input from another node that is executing $i'$. Or $\forall k' \in V_I : \exists (k', i') \in E_I$, there must exist a $k \in V_R^* : \exists (k,i) \in E_R^*$ and $M(k) = k'$.

**Theorem 3.4.3** *Let $I = (V_I, E_I)$ be the input DFG and $R_{II} = (V_R, E_R)$ be the time extended resource graph. Every valid mapping implies an epimorphic function $M : V_R^* \longrightarrow V_I$ where $V_R^*$ is a subgraph of $V_R$.*

**Proof 3** *• $M$ is function. A mapping is valid if each PE executes the maximum of one operation per cycle. Therefore, $\forall i \in V_R^*, M(i)$ is exactly one element in $V_I$.*

- *$M$ is surjective. A valid mapping implies that every nodes in graph $I$ must be mapped onto some PEs in time extended resource graph. Therefore, it is surjective.*
- *A subset of $R_{II}$ is homomorphic to $I$. In a valid mapping, every node $i \in V_I$ is mapped to a set of nodes $s_i \subset V_R$. Therefore, $\forall a, b \in V_R^*$, if there is an arc $(a, b) \in E_R^*$, then either $a, b \in s_i$ or $a \in s_i, b \in s_j$ where $s_i \neq s_j$. The first case simply implies homomorphism according to definition. We claim that the second case also implies homomorphism. Let's assume node $j \in V_I$ is mapped to $s_j$. If there is an arc between nodes $i$ and $j$ then it implies a homomorphism. Otherwise, it is trivial to see that if there is no arc between $i$ and $j$ but between $a$ and $b$, then we can remove $(a, b)$ from $E_R^*$ and mapping will still be valid. Both graphs in Figure 3.6(c) and Figure 3.6(d) are epimorphic to Figure 3.6(a). All arcs in the mapping correspond to an arc in the input DFG.*

**Definition 6** *Input subgraph: for every node $i$ in a digraph $I = (V_I, E_I)$, there exists a subgraph $G = (V_G, E_G)$ such that $V_G$ is the set of all nodes $j : (j, i) \in E_I$ in addition to node $i$; also $E_G$ is the set of all arcs $(j, i) : (j, i) \in E_I$.*

**Definition 7** *An isomorphism from $G = (V_G, E_G)$ onto $H = (V_H, E_H)$ is defined as $f : V_G \longrightarrow V_H$ such that:*

1. *$|E_G| = |E_H|$*
2. *$|V_G| = |V_H|$*
3. *$\forall u, v \in V_G : (u, v) \in E_G$ iff $(f(u), f(v)) \in E_H$ [32].*

**Theorem 3.4.4** *Every valid mapping implies an epimorphic function $M : V_R^* \longrightarrow V_I$ such that $\forall i \in V_R^*$, **input subgraph** of $M(i)$, $K = (V_K, E_K)$, **input subgraph** of $i$, $L = (V_L, E_L)$: if $Indegree(M(i)) > 0 \Rightarrow$*

1. *$\forall j \in V_L : M(j) = M(i)$ or*
2. *$K$ and $L$ are isomorphic.*

**Proof 4**
- *Theorem 3.4.3 proves that every valid mapping implies an epimorphic function $M : V_R^* \longrightarrow D$.*
- *Let's assume $a \in V_I$ is mapped onto set $s_a \subset V_R^*$. $\forall i \in V_R^* : Indegree(i) > 0$ either:*
  - *all incoming arcs of $i$ are from nodes $j \in s_a$ which implies $\forall j \in V_L : M(j) = M(i)$.*
  - *input arcs of $i$ are from a combination of nodes $j \in s_a$ and nodes $k \notin s_a$. This case cannot happen according to mapping problem definition.*
  - *all incoming arcs are from nodes $j \notin s_a$. In this case, we show that in order to have a valid mapping, $K$ and $L$ should be isomorphic. Let's assume that $K$ and $L$ are not isomorphic. Then either $V_K \neq V_L$ or $E_K \neq E_L$. If the number of nodes or arcs are different then the mapping cannot be valid.*

**Theorem 3.4.5** *Every epimorphic function $M : V_R^* \longrightarrow V_I$ with following constraint implies a valid mapping. Constraint: $\forall i \in M : V_R^*$, **input subgraph** of $M(i)$, $K = (V_K, E_K)$, **input subgraph** of $i$, $L = (V_L, E_L)$: if $Indegree(M(i)) > 0 \Rightarrow$*

1. *$\forall j \in V_L : M(j) = M(i)$ or*
2. *$K$ and $L$ are isomorphic.*

**Proof 5**
- *Because the function is surjective, all nodes in $V_I$ must be covered by at least one node in $V_R^*$.*

- *Since epimorphism preserves node adjacency [49] and the function is arc surjective, then all arcs in $E_I$ are covered by an arc in $V_R^*$.*
- *$\forall u \in V_I$ : indegree$(u) > 0$ where $K = (V_K, E_K)$ is input subgraph of $u$ and $L = (V_L, E_L)$ is input subgraph of $M^{-1}(u)$:*
  - *if $\forall j \in V_L : M(j) = u$, there must be a node $v \in V_R^*$ such that input subgraph of $v$ is isomorphic to $K$. Because function is surjective, all arcs of $E_K$ must be mapped. Therefore, according to the constraint, since the first case cannot happen, there exists a node whose input subgraph is isomorphic with $K$ which implies $u$ is mapped properly.*
  - *if $K$ and $L$ are isomorphic, then mapping of $u$ is valid.*

**Theorem 3.4.6** *Every valid mapping from an input DFG $I = (V_I, E_I)$ onto a time extended resource graph $R_{II} = (V_R, E_R)$ is equivalent to an epimorphic function $M : V_R^* \longrightarrow V_I$ such that $\forall i \in V_I$, **input subgraph** of $M(i)$, $K = (V_K, E_K)$ and **input subgraph** of $i$, $L = (V_L, E_L)$ : if $Indegree(M(i)) > 0 \Rightarrow$*

1. *$\forall j \in V_L : M(j) = M(i)$ or*
2. *$K$ and $L$ are isomorphic.*

*where $V_R^* \subseteq V_R$.*

**Proof 6**
- *From Theorem 3.4.4, every valid mapping implies an epimorphic function $M : V_R^* \longrightarrow V_I$ with above-mentioned constraints.*
- *From Theorem 3.4.5, every Epimorphic function $M : V_R^* \longrightarrow V_I$ with above-mentioned constraints implies a valid mapping.*

Chapter 4

# EPIMAP

The mapping problem is very challenging because when an $II$ is given and fixed, finding a valid mapping is an NP-Complete problem. There are well-known techniques to approximate the lower bound $II$, referred to as minimum $II$ or $MII$. However, the existence of a valid mapping for an $II$ is not guaranteed. Therefore, existing techniques start with $MII$ and explore the search space to find a valid mapping. If a mapping at $MII$ could not be found, $II$ is increased until a valid mapping can be made. Such schemes have to carefully look for possible mappings at a given $II$ and spend adequate time to find it. However, if one spends too much time on an $II$, explore the entire search space, given that no mapping can be made at that $II$, it might be impractical to use such technique in a compiler due to huge compilation time. Therefore, a reasonable mapping technique has to heuristically decide to increase $II$ after adequate search.

In this section, we describe our heuristic algorithm called EPIMap. EPIMap divides the mapping into two well-known problems, scheduling and placement. In scheduling step, a modulo cycle is assigned to operations. When all operations are scheduled, resources will be allocated for each operation. Routing, i.e satisfying data dependency between operations, is resolved in part in scheduling and placement. Second, the priority of the nodes (selection ordering) in this algorithm is adjusted progressively based on the success of failure of mapping nodes in previous mapping attempts.

This algorithm is fundamentally different from previous techniques. First, the placement problem is formally studied in this research. We reduce the placement problem to finding the maximum clique problem. We extend the placement to support register allocation too. Second, the node selection order in our algorithm is adjusted to characteristics of input loop found during previous mapping attempts.

## 4.1 Overview

EPIMap addresses the mapping problem in a constructive manner. It first estimates $MII$ and then iteratively schedules the operations such that the scheduled graph meets certain necessary mapping conditions. In this step, $II$ estimation is adjusted too when scheduling for a give $II$ found to be not feasible. Using CGRA description and $II$, a resource graph is constructed. Resource graph and scheduled DFG are used to construct a compatibility graph. In the end, EPIMap finds the maximum clique in the constructed compatibility graph. This clique represents the mapping output.

If all DFG nodes are represented in the clique, the mapping is complete. Otherwise, EPIMap finds the set of DFG nodes that are not represented in the clique and re-schedules them with higher priority. This is what we refer to as adjusting nodes priority based on the characteristics of input DFG.

The above steps are repeated until a mapping can be made or $II$ increases. When $II$ increases, EPIMap starts over as shown in Algorithm 2. The rest of this section is devoted into 3 basic steps of EPIMap: Scheduling, Placement, and Re-Scheduling.

## 4.2 Scheduling

At this step, an execution time is assigned to each operation and $II$ is approximated. During scheduling, operations are ordered with respect to each other to form a ready list. Operations are then selected by this order to be assigned to an execution time $t \in Z^+$. This time is chosen such that there are sufficient number of resources available at the execution time of the selected operation. If a CGRA is fully connected, this step is sufficient to complete the mapping, like classical modulo scheduling in VLIW processors such as [94].

$MII$ can be expressed as

$$MII = Max(ResMII, RecMII) \tag{4.1}$$

where $ResMII$ is the resource constrained minimum $II$ and where $RecMII$ is the recurrence-constrained minimum $II$. In an $M \times N$ CGRA,

$$ResMII = \lceil \frac{n}{M \times N} \rceil \tag{4.2}$$

where $n$ is the number of operations in the DFG. $RecMII$ indicates inter-iteration dependency of operations in a loop. When such a dependency exists, the next iteration cannot start until the result(s) from the previous iteration becomes available. We use the same technique as in [94] to extract $RecMII$.

There are few necessary mapping conditions a DFG must meet to be mappable. First, the out-degree of all nodes in DFG should be less than or equal to the out-degree of PEs in the resource graph. The out-degree of a node $v$ is the number of operations using the result of $v$ as an input. On the other hand, the out-degree of a PE $r$, is the number of adjacent PEs to $r$ plus one. It is because $r$ itself and all neighboring PEs of $r$ can read the output register of $r$ and use it as an input in the next cycle. The out-degree of all nodes in DFG is checked while scheduling is conducted. If a node is found to violate this constraint, routing nodes are added at the output of the violating node.

The second necessary mapping condition is that the number of nodes scheduled at any cycle must be at most equal to the number of PEs in CGRA. When an operation $v$ is to be scheduled at a cycle $t$, the number of available resources at cycle $t$ is checked.

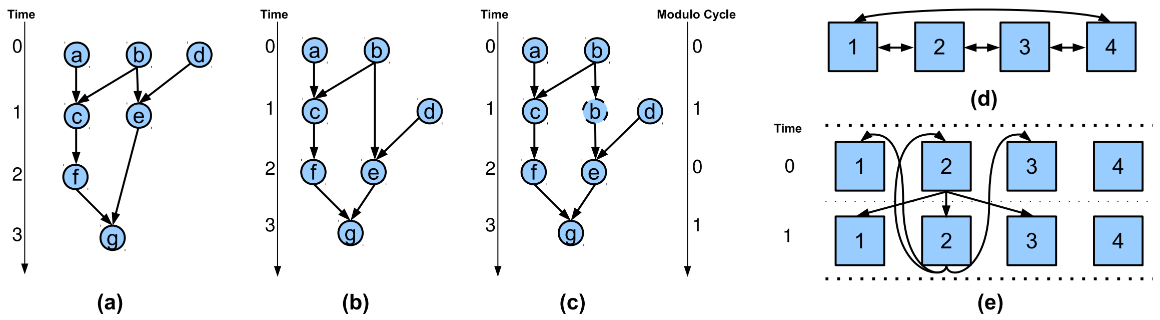


Figure 4.1. a) Nodes are scheduled as soon as all predecessors are completed (ASAP), b) Nodes are scheduled just before the earliest successor is scheduled (ALAP), c) Nodes are scheduled in a modular manner, d) A $2 \times 2$ CGRA shown in linear form, e) A resource graph constructed for the same CGRA with extension $II = 2$.

If there are sufficient number of resources available to perform operation $v$, it is scheduled at $t$ and a resource is reserved for operation $v$ in the resource table. For a multi-cycle operation with latency of $\ell$ cycles, a resource must be available from cycle $t$ to $t + \ell$.

Operations are scheduled in three steps: ASAP, ALAP, and Modulo steps. Initially operations are scheduled in an ASAP manner, that is, an operation is scheduled as soon as all predecessors are completed, that is:

$$t^j \geq t^i + d^i \qquad \forall (i,j) \in E : w^{(i,j)} = 0 \tag{4.3}$$

where $t^j$ is the cycle operation $j$ is scheduled and $d^i$ is the latency of executing operation $i$. Operation without any predecessor at the same iteration are scheduled at cycle 0.

At this step, operations are ordered with respect to other operations at the same iteration. That is, an operation is scheduled at the earliest cycle after all predecessors at the same iteration are completed. As such, arcs are limited to the one with weight of zero, e.g. $w^{(i,j)} = 0$. An arc with a weight greater than zero represents data dependency across loop iterations. A table is formed to keep track of the number of available resources at any cycle. If there are no available resources at cycle $t^i + d^i$, the time is increased until there is enough resources available. EPIMap forms a list to holds all operations that are ready to schedule, i.e. operations that all of their predecessors have already been scheduled. Operations in this list are sorted based on priority presented in [72]. Nodes in DFG shown in Figure 4.1(a) are scheduled in ASAP manner.

Critical cycle is the set of operations in a recurrence cycle that determines $RecMII$. Operations within the critical cycle have the highest priority followed by load operations. The second important factor is the earliest cycle at which an operation can be scheduled. We wish to schedule an operation as soon as all predecessor operations are completed. This is an important factor because we wish to minimize the time and resources required to hold a data on registers to satisfy data dependencies. When an operation $v$ is not scheduled immediately after all predecessors are completed, the output of all predecessors have to be hold temporarily until $v$ is executed. This imposes overhead of allocating resources such as PEs to hold (*route*) those values until all consumers are initiated. The third factor is the in-degree of an operation where operations with highest in-degree number are prioritized in this scheme. Similar to previous factor, operations with greater in-degree require more resources to temporarily hold the output of their predecessors as compared to operations with smaller in-degree number.

For an operations $i$ with execution latency of few cycles, all predecessors must be scheduled $d^i$ cycles after $u$ is scheduled. If PEs are pipelined, the number of resources at $t^i$ is reduced by 1. On the other hand, if $d^i$ is greater than 1 but $i$ cannot be pipelined, a resource from cycle $t^i$ to $t^j$ must be reserved for operation $i$. Such multi-cycle operation can impact operations that are to be scheduled in next $d^i$ cycles. For such operations, during next $d_i$ cycles, EPIMap makes sure that a resource will be available for operations within critical cycle. If this cannot be guaranteed, operation $i$ is scheduled with few cycles delay until all operations within the critical cycle are guaranteed an available resource. Note that $II$ is proportional to latency of critical cycle.

**Algorithm 1:** Schedule(Graph $I = (V_I, E_I)$, CGRA Resources, II, C)

```
1  for $\forall u \in V_I$ do
2      if $Out\_Degree(u) > k$ then
3          Insert routing node $v$ after $u$ ;
4          Move $k-1$ predecessors of $u$ to $v$ ;
5  $T_L \leftarrow$ ASAP_Schedule($I = (V_I, E_I)$, CGRA Resources) ;        /* schedule nodes in ASAP manner */
6  ALAP_Schedule($I = (V_I, E_I)$, CGRA Resources, $C \times T_L$) ;      /* schedule nodes in ALAP manner */
7  $S \leftarrow$ set of all nodes without successor in $V_I$ ;             /* initiate ready list */
8  Sort($S$) ;                                                               /* sort ready list */
   // while there is an unscheduled node
9  while $|S| \neq 0$ do
       // select a node u to schedule
10     for $\forall u \in S$ do
11         $t^u \leftarrow C \times T_L$ ;   /* scale the schedule window, tuning $C$ is discussed in Section ?? */
           // find the latest cycle u can be scheduled
12         for $\forall v \in Successors(u)$ do
13             if $t^v < t^u$ then
14                 $t^u \leftarrow t^v - d^v$;
           // select a cycle from scheduling window of u
15         for $t_C \leftarrow t^u$ downto $T^u_{ASAP} + d^u$ do
16             if a Res is available in Table for u between $t_C$ and $t_C - d^u$ then
17                 Update(Table, $u$, $t_C$, $d^u$) ;   /* allocate a resource for u during its execution
                   period */
18                 Update(S, $u$) ;   /* update S with predecessors of u that are ready (sorted
                   insertion) */
19                 Remove $u$ from S;
20         if $u$ could not be scheduled then
21             Increase II;
22             Reset(Table);
23             goto 7;
```

At the end of this step, the latency of executing one iteration of the loop, $L$ is determined. Given this latency, the operations in the loop are scheduled in an

ALAP manner. That is, an operation is scheduled at the latest cycle before all of its successors are initiated:

$$t^i \leq t^j - d^i \qquad \forall (i,j) \in E : w^{(i,j)} = 0 \tag{4.4}$$

The backward scheduling starts with the set of nodes without any successor at the same iteration. Those nodes are scheduled at time $C \times L$ where $C$ is a constant factor. In Section **??**, we run few experiment to optimize $C$ factor. Increasing the latency of an iteration by a constant factor enables EPIMap to schedule operations with more flexibility when a mapping attempt fails. Being able to increase the latency of an iteration plays an important role when EPIMap constrains the scheduling and relaxes the resource allocation to increase the chance of finding a mapping. We will discuss this later in this section. Nodes in DFG shown in Figure 4.1(b) are scheduled in ALAP manner.

When the ASAP and ALAP schedules for all operations are completed, EPIMap associates a modulo schedule to operations. Let $t^i_{ASAP}$ and $t^i_{ALAP}$ denote the cycles that operation $i$ is scheduled in ASAP and ALAP steps. Next, a table with $II$ rows is formed to keep track of the number of available resources in CGRA from cycle 0 to $II - 1$. EPIMap schedules the operations backward similar to previous step. This time, however, the number of time slots in resource table is limited to $II$, rather than $C \times L$ rows. EPIMap forms a list that holds the set of operations ready to be scheduled sorted like before. Let $v$ be the first node in this list. EPIMap finds the greatest cycle $t^i_{ASAP} \leq t^i_C \leq t^i_{ALAP}$ when there are enough resources available at row $t^i_C \% II$ to schedule $i$. Note that $t^i_C < t^j_C - d^i \;\; \forall (i,j) \in E : w^{(i,j)} = 0$. Nodes in DFG shown in Figure 4.1(c) are modulo scheduled. The modulo cycle association of nodes is shown on right and the schedule cycle is shown on left of this DFG.

If EPIMap fails to schedule operations within $II$ cycles, $II$ is increased by 1 until a feasible schedule can be made. Note that $II$ can at most increase to $L$. The details of scheduling is presented in Algorithm 1.

## 4.3 Placement

When operations are modulo scheduled within $II$ cycles, each operation has to be assigned to a PE for execution. We refer to such assignment as *placement*. For simplicity, let's assume that the latency of executing instructions is one cycle. In this case, the placement is in fact the problem of finding a subgraph in the constructed resource graph that is isomorphic to the scheduled DFG. Subgraph isomorphism is an NP-Complete problem [32]. Instead of just checking the existence of such relation between the scheduled graph and resource graph, we take a different approach to solve the placement problem.

EPIMap tries to find the maximum common subgraph (MCS) [68] between the resource graph and the scheduled DFG. If the size of nodes in MCS is equal to the size of node in DFG, the mapping is complete. Otherwise, the set of nodes not present in MCS are given higher priority in next scheduling attempts.

The placement is completed in three steps. First the resource graph is constructed from $II$ approximated in scheduling and PEs in CGRA. Second, a graph called compatibility graph is constructed from DFG and resource graph. Last, EPIMap finds the maximum clique in the compatibility graph. This clique represents the MCS between resource graph and DFG.

**Step 1 - Resource Graph Construction:**

Denoted by $R_{II} = (V_R, E_R)$, the set of nodes in resource graph are constructed by replicating the set of PEs in CGRA $II$ times. Each PE replication at a cycle

$0 \leq t < II$ represents a PE available to execute an operation at modulo cycle $t$. For every pair $(u, v)$ of adjacent PEs in the CGRA, there is an arc from (replication of) $u$ at time $t\%II$ to (replication of) $v$ at time $(t+1)\%II$. Note that every node in the CGRA is adjacent to itself. The set of nodes in the resource graph constructed for CGRA depicted in Figure 4.1(d) is shown in Figure 4.1(e) for $II = 2$. The arcs for $PE_2$ at cycle 0 and 1 are shown to visualize the set of arcs in resource graph.

**Step 2 - Compatibility Graph Construction:**

Consider the set of operations scheduled at modulo cycle $0 \leq t < II$. That is:

$$S_t = \{\forall u \in V_D : t_C^u\%II = t\} \tag{4.5}$$

Let $PE_i^t$ represents replication of $PE_i$ at cycle $t$ in resource graph. In a homogeneous, i.e. the functionality of all PEs are the same, an operation within set $S_t$ can be executed at any resource $PE_i^t$. However, in a heterogeneous CGRA, a subset of resources can execute operation $u$. Let Boolean function $S(PE_i, u)$ be true if $PE_i$ supports instruction $u$.

$$R_u = \{\forall PE_i^t \in V_R : t_C^u\%II = t \quad \wedge \quad S(PE_i, u) = true\} \tag{4.6}$$

Let pair $(PE_i^t, u)$ represents an assignment of operation $u$ to a node in resource graph that can execute this operation. The set of nodes in compatibility graph is formed from all such pairs. It is important to note that scheduling the operations significantly reduces the size of nodes (such pairs) in compatibility graph. Because in $R_u$, resources are limited to the one that have the same $t$ (time extension) that is equal to $t_C^u\%II$ (modulo cycle assigned to operation $u$). Scheduling in fact, decreases the size of $R_u$ by a factor of $II$. For instance, operations $a$ and $b$ are scheduled in modulo cycle 0 as shown in Figure 4.1(e). As such, pair $(PE_1^0, a)$ is present in node set of compatibility graph while $(PE_1^1, a)$ is not present because $a$ is scheduled at time 0 not 1.

Note that a node in compatibility graph represents a potential assignment of an operation to a resource (a potential mapping). The placement search space is composed of all of such potential assignments. In this space, we wish to find a subset of nodes in compatibility graph where no two potential assignments have a conflict with each other. For instance, consider the set of pairs formed for mapping DFG in Figure 4.1(c) to resource graph in Figure 4.1(e). $(PE_1^0, a)$ and $(PE_1^0, b)$ are two valid pairs because both operation $a$ or $b$ can be assigned to $PE_1^0$. However, those pairs cannot co-exist in any solution because $PE_1^0$ can only execute one operation per cycle. Consider $(PE_1^0, a)$ and $(PE_3^1, c)$ pairs. Both of these assignments are acceptable but they cannot co-exist in solution because if $a$ is mapped on $PE_1^0$, $PE_3^1$ that executes operation $c$ cannot read the output register of $PE_1^0$ because $PE_1$ and $PE_3$ are not directly connected in CGRA while $PE_3^1$ requires the output of $PE_1^0$ (a) to execute $c$.

Co-existence is represented by edges in compatibility graph. We wish to form the set of edges such that an edge connects two pairs in a compatibility graph when they can co-exist in a solution. Given two pairs of assignments $(PE_i^t, u)$ and $(PE_j^{t'}, v)$ where $PE_i^t \in R_u$ and $PE_j^{t'} \in R_v$, there is an edge between them unless:

1. $u = v$.
2. $(t = t') \wedge (i = j)$.
3. $((u, v) \in E_D) \wedge ((PE_i^t, PE_j^{t'}) \notin E_R)$.
4. $((v, u) \in E_D) \wedge ((PE_j^{t'}, PE_i^t) \notin E_R)$.

Using above relation, EPIMap forms the set of edges in compatibility graph using the above relation. Note that co-existence is a symmetric relation.

**Step 3 - Find the Maximum Clique:**

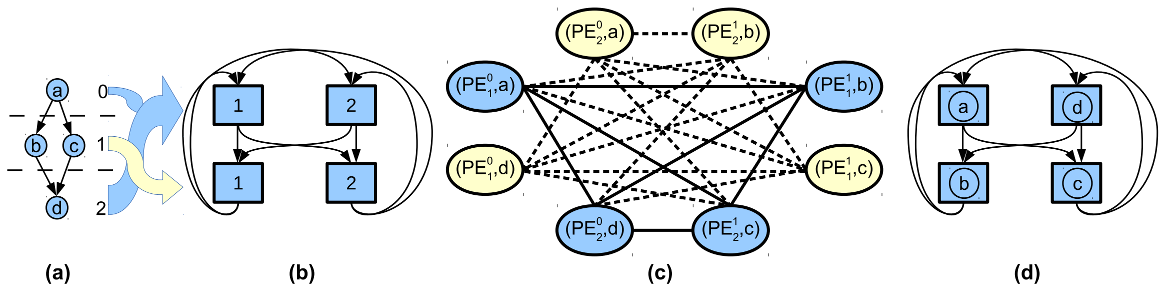


Figure 4.2. a) A scheduled DFG, b) A resource graph constructed for a $2 \times 1$ and $II = 2$, c) The compatibility graph with 8 nodes, each node represents a potential assignment of a node in DFG to a resource in resource graph. Nodes colored blue form a clique of size 4. This clique represents the final placement. d) The placement highlighted with blue nodes in (c) is shown in details.

Consider two nodes in the compatibility graph. An edge between those nodes implies that those two assignments *can* co-exist in a solution. A set of nodes in compatibility graph where every two nodes are connected with an edge implies a solution where all of those assignments can co-exits with each other. We wish to find the largest such subgraph in compatibility graph. If the size of this set is $V_D$, it implies that all operations are assigned to a resource and the mapping is complete. Finding a graph with such property is a well-known NP-Complete problem, maximum clique problem [32].

Consider a scheduled DFG shown in Figure 4.2(a) with an $II = 2$. A resource graph is constructed for a $2 \times 1$ CGRA with this $II$ as shown in Figure 4.2(b). There are 8 potential assignments of operations to resources. This is reflected as the set of nodes in compatibility graph presented in Figure 4.2(c).

An edge between two potential assignments in compatibility graph reflects whether those assignment can co-exist in a solution. For instance, there is no edge between $(PE_1^0, a)$ and $(PE_2^0, a)$. It is because both of these nodes assign the same operation, $a$, to difference resources. There is no edge between $(PE_1^0, a)$ and $(PE_1^0, d)$ either because

$PE_1^0$ can only execute either $a$ or $d$ at modulo cycle 0. A clique is highlighted with blue colored nodes in the compatibility graph. This clique represents the mapping shown in Figure 4.2(d).

## 4.4 Re-Scheduling

If a placement fails to allocate at least one PE for each operation, the set of unplaced operations are formed to be rescheduled in the next attempts. Since EPIMap schedules operations at the greatest available time slot, at this step, operations for which a resource could not be allocated in previous attempt, are scheduled one cycle earlier than their current schedule cycle. If such rescheduling is impossible for an operation, a routing node is inserted after the node (relaxing placement problem). If this rescheduling requires to schedule an operation earlier than its ASAP schedule, or adding extra nodes increases $ResMII$ and $MII$, then the DFG is rescheduled with a new heuristic. At this step, the number of available PEs is set to be $N-1$ where $N$ was the number of available PEs at the previous scheduling attempt. Thus the resource table constructed for scheduling will allow less number of operations at any given cycle in ASAP and ALAP, increasing the length of a schedule. However, it does not directly impact $II$.

This heuristic constraints the scheduling problem, however, the resource allocation problem would be relaxed. Note that decreasing the number of available resources can increase $II$ when routing nodes are required to schedule the operations. In such case, EPIMap increases $MII$ by one and resets the number of available resources to be the number of PEs. When $MII$ is increased, EPIMap proceeds with a new scheduling and placement attempt.

**Algorithm 2:** EPIMap(Graph $I = (V_I, E_I)$, CGRA Resources, C)

```
1  Schedule(I, CGRA Resources, MII, C) ;          /* modulo schedule operations and extract MII */
2  $R \leftarrow CGRAResources$ ;                 /* R holds the number of each CGRA resources */
3  $D \leftarrow I$ ;   /* D is a copy I, transformation is done on D to preserve I from any change */
4  while $true$ do
5      Schedule(D, R, II, C) ;             /* modulo schedule D with R resources and estimate II */
       // II is changed?
6      if $II > MII$ then
7          $D \leftarrow I$ ;                                          /* reset D to the original DFG */
8          $R \leftarrow CGRAResources$ ;                   /* reset R with original CGRA resources */
9          $MII \leftarrow MII + 1$ ;                                            /* increase MII by 1 */
10         continue;
       // while clique size is increasing and MII is not increased
11     while $V_{Cli}$ increasing and $II < MII$ do
12         $CMP$ ←Comp_Graph(CGRA Resources, D, II) ;   /* construct the compatibility graph */
13         $Cli$ ←Max_Clique(Comp) ;                                /* find the maximum clique */
           // if all nodes are present in the clique
14         if $|V_{Cli}| = |V_D|$ then
15             return mapping ;                                        /* mapping is complete */
           // re-schedule nodes not present in the clique upfront
16         ReSchedule(D, $V_{Cli} - V_D$, R, II, C)
17     $R \leftarrow R - 1$ ;                            /* decrease the number of resources by 1 */
```

# Chapter 5

# GENERALIZED RESOURCE ALLOCATION

Pipelining is an effective hardware technique to improve throughput at a modest hardware overhead. In addition, some operations are inherently sequential and multi-cycle implementation of such operations can significantly reduce the area without scarifying frequency. Supporting pipelined as well as multi-cycle operations are very important in any mapping algorithm.

An important performance factor in any processor is register files utilization. An effective register utilization increases the performance of an accelerator significantly and minimizes data communication with memory sub-system. In this section, we extend the mapping algorithm to allocate PEs for multi-cycle operations, pipelined PEs, as well as supporting register allocation with placement simultaneously.

## 5.1 Multi-cycle Extension

Complex arithmetic operations such as *division* are usually supported by a multi-cycle ALUs to save silicon area and maintain working frequency. Single cycle implementation of such operations severely damages performance and requires extensive silicon area. In this part, we extend the placement model to support operations with multi-cycle execution.

First, placement of multi-cycle operations requires to allocate a PE for few cycles. We need to modify compatibility graph to represent such assignment during those multi-cycle execution. We extend placement model without imposing any overhead to

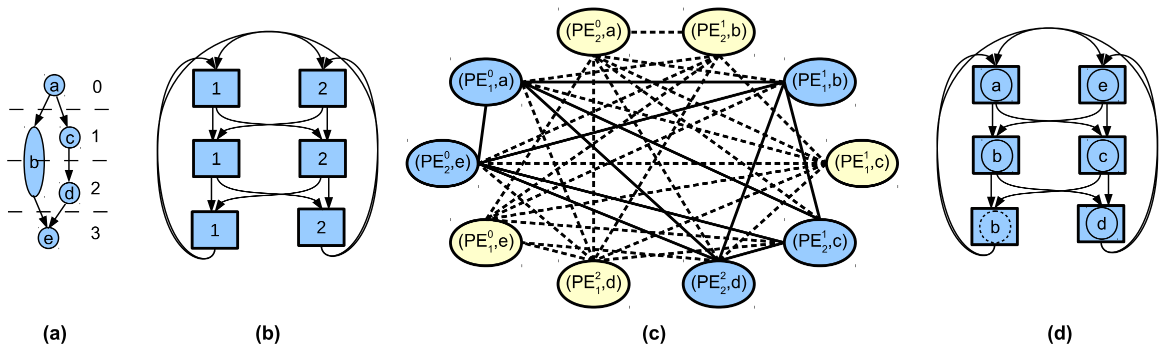


Figure 5.1. a) A DFG with 5 operations where execution of operation b takes 2 cycles. b) A resource graph constructed for a $2 \times 1$ CGRA and $II = 3$. c) The compatibility graph constructed from DFG and resource graph. Nodes with blue color forming a clique of size 5. d) The mapping of DFG onto CGRA and $II = 3$. This mapping is represented with blue colored nodes forming a clique in c.

the number of nodes in compatibility graph as well as to the size of clique representing placement.

Let $u$ be an operation which requires multiple cycles to execute on a PE. Let $(PE_i^t, u)$ represents an assignment of instruction $u$ to $PE_i$ where $u$ is scheduled to initiate at modulo cycle $t$. Although $u$ requires few cycles to complete, there is no need to add more nodes to the compatibility graph to model assignment of operation $u$ to PEs during those cycles. Rather, this can be modeled as one node in compatibility graph and an adjustment to edge formation. In fact, allocating a PE for multiple cycles is treated as resource conflict between nodes in compatibility graph. This is an important feature because the placement complexity is expected to increase when the size of compatibility graph or clique increases.

For node $(PE_i^t, u)$ in compatibility graph, when the latency of executing $u$ is $l$ cycles, all resources representing $PE_i$ for next $l$ cycles should have no edge to $(PE_i^t, u)$. This represents a resource conflict between $(PE_i^t, u)$ and all assignments to $PE_i$ in next $l$ cycles. This is illustrated in Figure 5.1. Operation $b$ in DFG shown in Figure 5.1(a) takes 2 cycles to execute. A resource graph with $II = 3$ is constructed for a $2 \times 1$

CGRA in Figure 5.1(b). Two nodes in compatibility graph represent assignment of operation $b$ to resources, $(PE_2^1, b)$ and $(PE_1^1, b)$, potential assignments of operation $b$ into two resources at cycle 1 when $b$ is scheduled to be executed.

The resource conflict of assigning operation $b$ to any resource in next cycles is considered in compatibility graph construction shown in Figure 5.1(c). For instance, no node representing resource $PE_1$ at modulo cycle 1 and 2 is connected to node $(PE_1^1, b)$ with an edge in compatibility graph. This is the case for nodes $(PE_2^1, b)$ as no assignment representing $PE_2$ at cycles 1 and 2 is connected to $(PE_2^1, b)$. A clique of size 5 is highlighted with blue colored nodes in Figure 5.1(c) and its corresponding mapping is shown in Figure 5.1(d).

For $(PE_i^t, u)$ and $(PE_j^{t'}, v)$ where $PE_i^t \in R_u$, $PE_j^{t'} \in R_v$, and $u$ is a multi-cycle operation, there is an edge between those nodes unless:

1. $u = v$.
2. $((t + d^u)\%II > t) \wedge (t' \geq t) \wedge (t' - t < d^u) \wedge (j = i)$.
3. $((t + d^u)\%II < t) \wedge (t' \geq t) \wedge (j = i)$.
4. $((t + d^u)\%II < t) \wedge (t' < (t + d^u)\%II) \wedge (j = i)$.
5. $((u, v) \in E_D) \wedge ((PE_i^{(t+d^u)\%II}, PE_j^{t'}) \notin E_R)$.
6. $((v, u) \in E_D) \wedge ((PE_j^{t'}, PE_i^t) \notin E_R)$.

## 5.2 Supporting Pipeline Operations

Pipelining is an effective technique to improve throughput and frequency. As opposed to multi-cycle operation, a pipelined PE can issue a new instruction every cycle. The result of a pipelining, similar to multi-cycle operation, becomes available after few cycles (few cycles latency).

When there are variations in latency of operations in a pipelined functional unit, structural hazards have to be carefully avoided. The structural hazard occurs when two instructions are completed at the same cycle in a PE. In such case, the output register can only be updated by one of those operations. This is classically referred to as structural hazard [50]. Structural hazards, too, are modeled as resource conflict in compatibility graph.

There is an edge between $(PE_i^t, u)$ and $(PE_j^{t'}, v)$ where $PE_i^t \in R_u$, $PE_j^{t'} \in R_v$, and $u$ and $v$ are operations to be executed on a pipelined PE unless:

1. $u = v$.
2. $((t + d^u)\%II = (t' + d^v)\%II) \wedge (j = i)$ (represents structural hazard).
3. $((u, v) \in E_D) \wedge ((PE_i^{(t+d^u)\%II}, PE_j^{t'}) \notin E_R)$.
4. $((v, u) \in E_D) \wedge ((PE_j^{(t'+d^v)\%II}, PE_i^{t}) \notin E_R)$.

This modification in constructing compatibility graph is illustrated in Figure 5.2. Operations **a** and **e** in DFG shown in Figure 5.2(a) are to be executed by pipelined functional units with latency of two cycles. A resource graph is constructed for a $3 \times 1$ CGRA and $II = 3$ as depicted in Figure 5.2(b). The compatibility graph constructed for this DFG and resource graph is partially shown in Figure 5.2(c). In this graph, $(PE_2^0, a)$ represents assignment of operation $a$ into $PE_2$ at cycle 0. Since $a$ takes two cycles to complete, any operation assigned to $PE_2$ that is completed at cycle 1 has a resource conflict with $(PE_2^0, a)$. For instance, $(PE_2^0, a)$ and $(PE_2^1, d)$ cannot co-exist in any solution, thus, there is no edge between those two nodes in compatibility graph. It is because the output register of $PE_2$ at the end of cycle 2 can only be updated by 1 operation. This is clearly a structural hazard between those assignments.

On the other hand, $(PE_2^0, a)$ and $(PE_2^1, e)$ do not cause any structural hazard. The output register of $PE_2$ at the end of cycle 1 can be updated by the result of operation $a$ when $e$ is assigned to $PE_2$ at cycle 1. It is because the result of operation $e$ is to

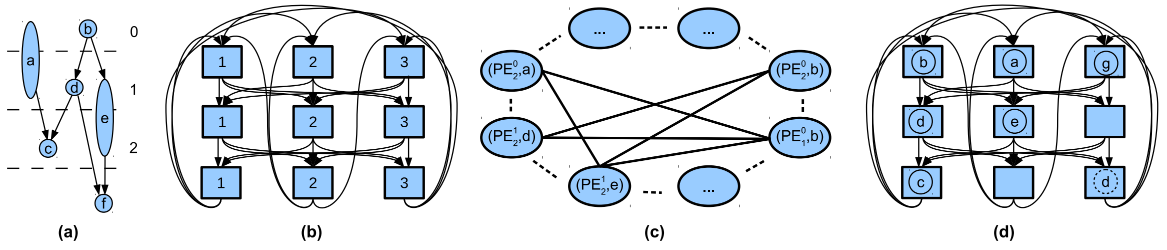


Figure 5.2. a) A DFG where operations $a$ and $e$ are to be executed on a pipelined PE. b) A resource graph constructed for a $3 \times 1$ CGRA with $II = 3$. c) A part of the compatibility graph for input DFG and resource graph is shown. d) A mapping of the input DFG to the CGRA is presented.

update output register of $PE_2$ at cycle 2. This is represented by an edge between those two assignments in this figure. A mapping of this DFG to input CGRA is shown in Figure 5.2(d).

## 5.3 Placement and Register Allocation

Register utilization is an important factor to maximize performance of an accelerator. Allocating local register files at PEs can significantly reduce data communication between PEs when data dependent instructions are scheduled few cycles apart. This data dependency can extend to instructions within successive loop iterations. In such cases, satisfying data dependencies through register files is more effective and less costly than variable spilling (sending data back and forth to the memory subsystem).

Traditionally, registers are allocated after scheduling. However, register allocation after modulo scheduling imposes a significant overhead either in performance or in compilation time. It is because in case of a failure in allocating registers, one may spill few variables and increase the $II$ to compensate for extra load/store operations. Due to repetitive nature of modulo scheduling and loop execution, the performance overhead of such policy is proportional to the number of loop iteration, which is presumably high and unacceptable.

The second plausible reaction to register allocation failure is to search for another modulo schedule and placement with the hope that a feasible register allocation is possible for the new mapping. This solution is preferable because it incurs no performance overhead, but it is not viable because it is unknown whether a new modulo schedule would permit a feasible register allocation. It is in fact very difficult to develop a reasonable heuristic to guess the non-existence of any feasible register allocation for a given $II$ when many feasible modulo schedules are available. Besides, finding a valid modulo schedule along with placement is a time consuming process. A register allocation failure in such policy imposes significant compilation time overhead.

A better policy is to modulo schedule operations, and then simultaneously place operation and allocate registers to hold variables temporarily. In this part, we extend the placement to model simultaneous PE and register allocation. An intuitive solution is to include registers in resource graph. Thus, when a compatibility graph is constructed, it inherently includes potential assignments of operations to local registers. This, however, significantly increases the size of compatibility graph by a factor of $R$ where $R$ is the size of register file. On top of this, the size of clique we search for increases substantially too.

Next, we present an integrated placement and register allocation model without any node overhead either in compatibility graph nor in clique we search for. Instead of repeating registers in resource graph, $R_{II}$, registers are encodes in the weight of arcs in the compatibility graph. As presented in previous section, the arcs in this graph were undirected and binary. Register-aware EPIMap, or REGIMap, instead, constructs a weighted directed graph $P$. In this extension, the existence of arcs is still a symmetric relation, yet the weights are not. For assignments $(PE_i^t, u)$ and

$(PE_j^{t'}, v)$ where $PE_i^t \in R_u$, $PE_j^{t'} \in R_v$, and $u$ and $v$ are operations to be executed on a pipelined PE, there is an arc between them unless:

1. $u = v$.
2. $((t + d^u)\%II = (t' + d^v)\%II) \wedge (j = i)$.
3. $((u, v) \in E_D) \wedge ((PE_i^{(t+d^u)\%II}, PE_j^{t'}) \notin E_R) \wedge (j \neq i)$.
4. $((v, u) \in E_D) \wedge ((PE_j^{(t'+d^v)\%II}, PE_i^{t}) \notin E_R) \wedge (j \neq i)$.

Now that the bidirectional arcs are formed symmetrically, we update the weight of arcs. Note that registers provide connectivity between two resources in resource graph representing the same physical PE but at different execution time.

First, the set of dependent operations (intra-iteration dependency) that are scheduled few cycles apart are detected. Let $(u, v) \in E_D$ where $w^{(u,v)} = 0$ and $t_C^v - t_C^u > 1$. The weight of arc between $(PE_i^t, u)$ and $(PE_i^{t'}, v)$ [27] is to be increased by:

$$\gamma = \left\lceil \frac{t_C^v - t_C^u}{II} \right\rceil \tag{5.1}$$

When these operations are scheduled less than $II$ cycles, one register is sufficient to carry-out data dependency between them. However, when those are scheduled more than $II$ cycles apart, this data dependency must be carried-out between successive iterations of the loop though multiple registers.

Consider $(PE_i^t, u)$ and $(PE_i^{t'}, v)$ where $(u, v) \in E_D$, $w^{(u,v)} = 0$ and $t_C^v - t_C^u > 1$. Without loss of generality, let's assume $t_C^v - t_C^u < II$. A register is used to hold the results when $u$ is executed on $PE_i^t$ and one is released after $v$ is executed on $PE_i^{t'}$. Since the interval between $t_C^u$ and $t_C^v$ can extend to multiple iterations, the number of required register is proportional to this interval.

[27]Note those operations should be mapped to the same physical PE to be able to share a register.

Let $C = (V_C, I_C)$ be the compatibility graph we are constructing. Let $S_{PE_i}$ be the set of all operation assignments to $PE_i$ in compatibility graph that is:

$$S_{PE_i} = \{\forall(\delta, PE_i^t) \in V_C\} \tag{5.2}$$

Without loss of generality, let's assume $t < t'$ [28]. All nodes in $S_{PE_i}$ representing $PE_i$ with a time extension $t \leq \theta < t'$, use those $\gamma$ registers to carryout this data dependency until $v$ is executed on $PE_i^{t'}$. Thus, the weight of arcs between those nodes to $(PE_i^{t'}, v)$ is increased by $\gamma$.

The rest of PEs $S_{PE_i}$ carryout $\gamma - 1$ outstanding data dependencies (produced but not consumed yet). The weight of arcs from those nodes to $(PE_i^t, u)$ is increased by $\gamma - 1$. When $t > t'$, the elements of those two sets should be interchanged.

Second, the set of operations with inter-iteration dependencies are detected. The weight of arc between $(PE_i^t, u)$ and $(PE_i^{t'}, v)$ where $w^{(u,v)} > 0$ is to be increased by:

$$w = w^{(u,v)} - \begin{cases} \left\lfloor \frac{t_C^v - t_C^u}{II} \right\rfloor - 1 & \text{if } t_C^v \geq t_C^u \\ \\ \left\lceil \frac{t_C^v - t_C^u}{II} \right\rceil & otherwise \end{cases} \tag{5.3}$$

Similar to the previous case, at all resources between $PE_i^t$ and $PE_i^{t'}$, $w$ registers should be available to establish a path between those resources. Therefore, the weight of arcs from all nodes in compatibility graph representing $PE_i$ to $(PE_i^t, u)$ is increased by $w$. However, the weight of arcs from nodes representing $PE_i$ between cycle $t_C^v$ and $t_C^u$ to $(PE_i^t, u)$ must be increased by $w - 1$.

[28]Because of the modulo operation $t$ can be greater than $t'$. Note that $t \neq t'$ otherwise there can be no arc between those nodes in compatibility graph (check compatibility relation).

At the end of this step, the sum of arcs, at each node $(PE_i^t, u)$ in compatibility graph, represents the number of required registers in $PE_i$ at cycle $t$. This is illustrated in Figure 5.3. Consider the DFG shown in Figure 5.3(a) to be mapped to a $2 \times 1$ CGRA with $II = 2$. As it is shown in Figure 5.3(b), registers are not present in resource graph constructed for mapping. The compatibility graph constructed from scheduled DFG and resource graph is shown in Figure 5.3(c). The weight of arcs in this graph represents the number of required registers for each assignment of operation to a resource. The sum of outgoing arcs represent the total number of required register at a PE. This number should be always less than the size of local register files.

For instance, consider operations $a$ and $d$ ins DFG shown in Figure 5.3(a). These operations are scheduled 3 cycles apart while $II = 2$. This implies that when operation $a$ from iteration $j$ is issued until the result is used by $d$ of iteration $j$, a new iteration of the loop is initiated. As shown Figure 5.3(d), operation $a$ of iteration $j$ is executed at cycle $i + 1$ and its result is stored in register 1 at $PE_2$. At cycle $i + 3$ instruction $a$ from iteration $j + 1$ is issued by $PE_2$ where the result is kept in register 2. At next cycle, instruction $d$ of iteration $j$ is issued by $PE_2$. In this example, 1 register is required in $PE_2$ at cycle $i + 2$ and $i + 4$ represented by modulo cycle 1 when $d$ is assigned to $PE_2$. However, at modulo cycle 0, two registers are required at $PE_2$ when $a$ is assigned to $PE_2$. In Figure 5.3(c), the weight of arc from node $(PE_2^0, a)$ to $(PE_2^1, d)$ is 2 ($w = \lceil \frac{3-0}{2} \rceil$) and it is 1 ($w - 1$)for the arc at the opposite direction representing exactly the number of required registers at $PE_2$ at those cycles.

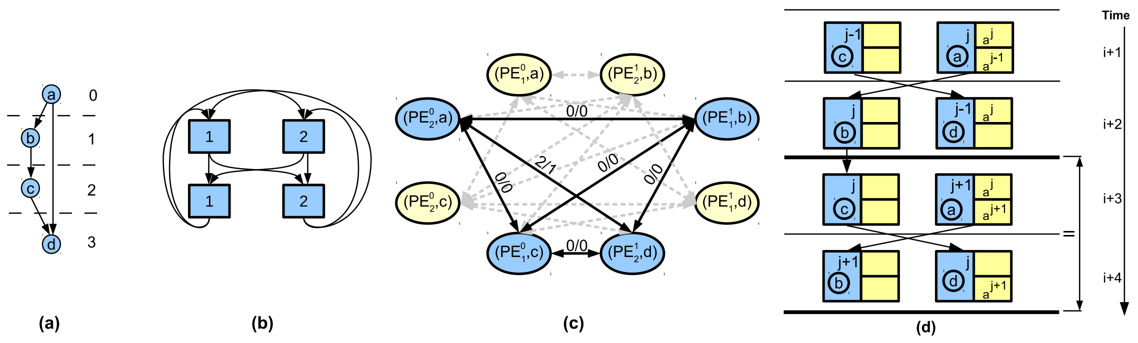


Figure 5.3. a) A DFG where dependent operations $a$ and $d$ are scheduled few cycles apart. b) A resource graph constructed for a $2 \times 1$ CGRA with $II = 2$. Registers are not present in resource graph. c) Compatibility graph is constructed of the DFG and resource graph. The weight of arcs in this graph represent the number of registers required for each operation assignment to a PE. A valid mapping is highlighted with nodes colored blue. d) A valid mapping with registers are depicted from the highlighted nodes in compatibility graph.

# Chapter 6

# SUPPORTING CONDITIONALS

One of the major challenges associated with all accelerators is to effectively execute loops that have *if-then-else* (ITE) constructs. The fundamental problem is that the outcome of a branch is unknown before runtime, thus, how to efficiently and effectively allocate computing resources to multiple execution flow branches. Accelerators use predication to execute the conditional constructs.

Hardware accelerators and FPGAs will execute both the paths of an ITE construct in parallel, and then choose the results of the taken path. This results in wasted resources and power. GP-GPUs also schedule the instructions and allocated resources for both the paths of the ITE construct, but at the runtime, do not issue the instructions for the not-taken path. This saves power, but the cycles and resources allocated for the not-taken path are still wasted. In the graphics processing community, this is referred to as the problem of "branch divergence."

This chapter deals with the problem of efficiently executing ITEs on a CGRA. Fundamentally, there are three ways to accelerate loops with an ITE construct on a CGRA. First is full predication - in which operations producing the same output are mapped to the same PE, but at different times. Second is partial predication - in which one extra *select* operation is inserted for each output which is used to merge the values produced in different branches. Third is a dual-issue architecture in which two instructions (one from each side of the ITE) are issued to the PE, and the operation to be executed is chosen at runtime by the PE.

Even though the dual-issue CGRA architecture has the potential to achieve the best performance, full predication and partial predication schemes are more common, since executing loops on dual-issue architecture requires compiler support – and none exists. Specifically, a compiler technique is needed to merge operations from either branches to be executed on a PE. How operation are paired not only affects the correctness of an execution, but also has a significant impact on the resulting performance. In this chapter, we formulate and solve the problem of merging operations from the branches to maximize performance.

## 6.1 Background and Related Work

Supporting acceleration of loops with conditionals is important, since many performance-critical loops have ITE constructs in them; ignoring them can greatly reduce the loop acceleration factor. The basic way to support conditionals in accelerators is through predication. Supporting predication on CGRAs requires a predicate network. As shown in Figure 1.1, the predicate network consists of predicate inputs from the neighboring PEs, a predicate output, and a small predicate register file. The result of the ITE expression, executed on a PE, is propagated to the PEs on which operations of the if-part and the operations of the else-part are executed through the predicate network. Most CGRAs implement the hardware of the predicate network [16, 21, 43]. There are three basic ways to support acceleration of conditionals on CGRAs.

### 6.1.1 Partial Predication

In partial predication, the operations of both the if-part and the else-part are mapped on different PEs. If the same variable is to be updated in both the if-part and the else-part, the final result is computed by selecting the output from the taken-path

```
1 $a \leftarrow a \times X_1$ ;
2 $b \leftarrow b \times X_2$ ;
3 $c \leftarrow a + b$ ;
4 if $(e<X_3)$ then
5     $e \leftarrow b \times X_4$;
6 else
7     $e \leftarrow b \times X_5$;
8 $f \leftarrow c \times e$ ;
```

(a)

```
1 $a \leftarrow a \times X_1$ ;
2 $b \leftarrow b \times X_2$ ;
3 $c \leftarrow a + b$ ;
4 $h \leftarrow (e < X_3)$ ;
5 $e_t \leftarrow b \times X_4$;
6 $e_f \leftarrow b \times X_5$;
7 $e \leftarrow Select(e_t, e_f, h)$;
8 $f \leftarrow c \times e$ ;
```

(b)

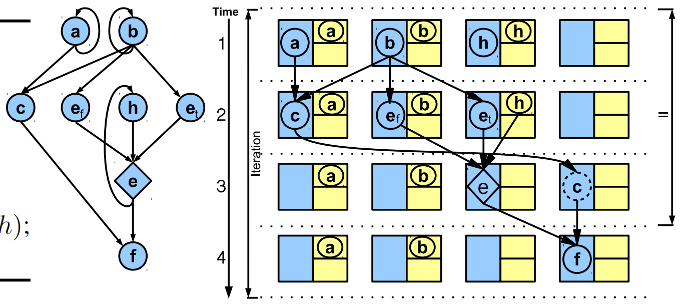


Figure 6.1. (a) shows a loop body with an ITE construct. Using partial predication, the loop is transformed to (b). The output e, that is calculated in both the paths is combined using the select operation. (c) Shows the DFG of the transformed code, and (d) shows the mapping of the DFG on a 2 × 2 CGRA. Note that the II achieved is 3 cycles.

based on the evaluation of the branch condition. This is achieved through a special operation, called *select* or a *conditional move*, which takes in the result of the branch condition (through predicate network), and two updated values of the variable to select the correct one. If a variable is to be updated in only one path, a select operation is still required to choose between the old value [29] and the new value produced at that iteration. The new value is valid only if the path of the branch in which the new value is computed should have been executed. Architectural support for partial predication is studied in [41].

Figure 6.1(b) shows the partial predication transformation of the loop body shown in Figure 6.1(a). In this scheme, new variables for operations in ITE paths are introduced. This new variable enables those paths to be executed independently, in parallel. For instance, operations at line 5 and 6 in Figure 6.1(b) can be executed independently. At the end, predicate ($h$) chooses the final value of ($e$). The select instruction is necessary to support partial predication transformation. Figure 6.1(c)

[29] The value updated in last iteration where this variable has been updated

is the DFG constructed after partial predication transformation and Figure 6.1(d) shows the mapping of this loop to a $2 \times 2$ CGRA. $II$ in this mapping is 3 and is the minimum possible for partial predication transformation.

To map an ITE that has “n” operations on each path, the number of operations for partial predication transformation is, in the worst case, $3n$. This is because all the operations from both paths must be mapped ($2n$), as well as the select operations ($n$). A select operation is needed for each output that will be needed in the rest of the program. The select operation may not be required for intermediate outputs. In the worst case, all the $2n$ outputs will be used in the rest of the program, and therefore $n$ select operations will be required. This increases $II$ substantially and results in a loss of performance.

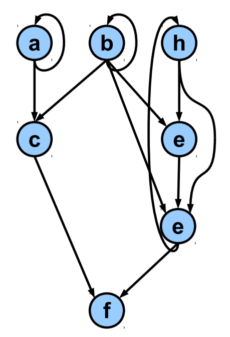
### 6.1.2 Full Predication

Full predication does not require a select operation. Instead, the operations that update the same variable are mapped to the same PE, albeit at different times. Since only one of the operations will be executed at runtime (and the other will be squashed), the correct value of the output is present in the register file of that PE by the end of that iteration. If an output is computed in only one path of the ITE construct, then the output must be computed on the PE that previously updated the same variable. This is done so that after executing an ITE, for each variable there is a unique PE, that has its value and therefore no select operation is needed.

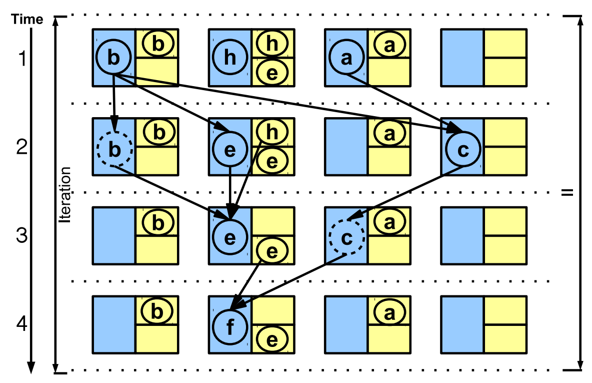
Consider the body of a loop shown in Figure 6.1(a). The result of full predication transformation is presented in Figure 6.2(a). Operations at line 5 and 7 in the original snippet of the code are guarded by ($h$) in Figure 6.2(a) at line 5 and 6. Operation at

```
1 $a \leftarrow a \times X_1$ ;
2 $b \leftarrow b \times X_2$ ;
3 $c \leftarrow a + b$ ;
4 $h \leftarrow (e < X_3)$ ;
5 $e \leftarrow b \times X_4 \;\; (h)$;
6 $e \leftarrow b \times X_5 \;\; (\overline{h})$;
7 $f \leftarrow c \times e$ ;
```

(a)



(b)



(c)

Figure 6.2. (a) shows the transformed code using full predication scheme. There is no select operation, but operations at line 5 and 6 that compute the variable "e" have to be mapped to the same PE. Note that in addition to the DFG, there are placement constraints that must be met for a legitimate mapping for full predication. (b) shows the DFG of the transformed loop. and (c) shows the mapping of the loop on a $2 \times 2$ CGRA.

line 4, uses variable ($e$) that is updated in the previous iteration. To make sure that variable ($e$) gets updated properly, operations at line 5 and 6 must be mapped on the same PE where variable ($e$) is kept. Figure 6.2(c) presents the best mapping of this loop after full predication transformation. The best $II$ achieved for full predication is 4. If we have to map only an ITE construct that has $n$ operations on either path, then the number of operation nodes for full predication DFG in the worst case is $2n$, but there are placement constraints for each of $2n$ nodes. The tight constraints on the operation placement are very restrictive, and can severely degrade the performance.

### 6.1.3 Dual-Issue

This scheme alleviates the problem of accelerating conditionals by issuing two instructions to a PE simultaneously, one from the if-path, and one from the else-path. Depending on the result of the conditional operation, only one of them is executed at

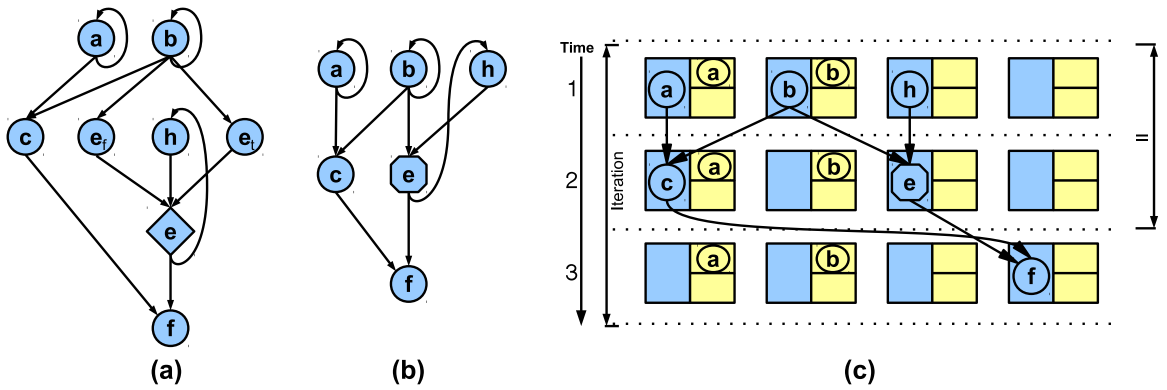


Figure 6.3. (a) shows the DFG with partial predication transformation. The operations from the two sides of the branch are merged to form packed nodes (packed nodes represent a pair of nodes executed in a dual-issue PE). (b) shows the DFG after the packing transformation. (c) shows the mapping of the transformed DFG on a $2 \times 2$ CGRA. II of 2 is achieved.

runtime. This method also does not require select operation. Hardware requirements for dual-issue execution is studied in [41].

Figure 6.3(a) shows the DFG after partial predication transformation. The nodes on either side of the DFG are merged to form a packed node. Packed node represents dual-issue operations. Operation ($e$) represented by a hexagonal shape node is a packed node. The adjacency of nodes are preserved after this transformation. Figure 6.3(b) shows the DFG after the packing transformation. Figure 6.3(c) shows the mapping after this transformation to a $2 \times 2$ CGRA. $II$ of this mapping is 2 and is the minimum possible. If we have to map only an ITE construct that has $n$ operations on either path, then the number of operation nodes for dual-issue DFG in the worst case is $n$. Plus there are no placement constraints. Therefore, dual-issue may be the best solution to accelerate conditional loops.

## 6.2 REGIMap extension to support conditionals

Although the dual-issue scheme promises the best performance, partial and full predication schemes seem to be more common in CGRAs. The reason is that the traditional schemes of full and partial predication do not require much change in the compiler. However, the new scheme of dual-issue requires extensive compiler support. This is because supporting partial predication requires generation of select operations, which is a well studied compiler topic. In fact it is a part of Single Static Assignment (SSA) transformation that is present in most compilers, and the select or conditional move operations are constructed in the *phi elimination* pass in compilers back-end. Once the DFG with select operations is generated, existing CGRA scheduling and mapping techniques (e.g., [2, 8, 18, 23, 31, 44, 48, 66, 79, 92]) designed to map non-conditional loops on CGRA, can also be used. Supporting full predication is also relatively straightforward, since the DFG remains the same. The only new aspect is placement constraints on the operations, which only means that as soon as the output of an operations is mapped, the mapping of the second operation updating that output is also fixed.

On the other hand, for a dual-issue architecture, compiler support is needed to pack operations from both paths of a branch, into a packed node. How we do operation pairing not only affects the correctness, but also has a significant impact on the resulting performance. Next, we formulate the conditions for legitimate packing, and also formulate the problem of performance optimization by packing, and then solve the problem to achieve efficient mapping for a dual-issue architecture.

Supporting execution of conditional loops in CGRAs has not received much attention in the research community. The only compiler technique we have seen is a form of full predication, presented in [42]. In this scheme, operations within body of an if-then-statement are mapped to the same PE. We believe that this is too restrictive, and it will cause significant performance loss because other PEs will not be utilized. It is important to note that the performance is proportional to PE utilization.

In this section, we extend REGIMap to support mapping loops with conditional constructs and name it BRMap. BRMap starts with a given control flow graph (CFG) of a loop. BRMap enables user to choose between full predication, partial predication, and dual issue transformations to be applied. Based on selected transformation, BRMap constructs a DFG from the input CFG. At the end, BRMap calls REGIMap to complete the mapping.

**Step 1: DFG Construction.** BRMap constructs a DFG from the CFG of a loop first. There are standard schemes to construct hyper-blocks from multiple basic blocks of a CFG using full-predication and partial-predication transformations [73]. DFG constructed from full-predication transformation can be directly fed to the underlying CGRA mapping algorithm. However, it is necessary to ensure that all instructions updating same variable are to be mapped on the same physical PE.

DFG constructed after partial-predication transformation requires minimal change in the mapping algorithm. REGIMap only allocates registers for nodes on PEs. We enhance REGIMap to allocate registers at PEs and predication network using the same technique.

Dual-issue scheme starts from a partial predicated DFG. DFG is scheduled first and $MII = Max(ResMII, RecMII)$ is extracted. Before conducting any DFG minimization (dual-issue transformation or D-transformation), we determine whether minimizing DFG by forming packed instructions would benefit the performance or not. BRMap conducts DFG minimization only if there is a performance benefit to do so. In some DFGs, reducing the number of nodes may not benefit the performance at all. The reason is that, mapping $II$ in some loops is limited by $RecMII$ rather than $ResMII$.

**Algorithm 3:** D-Transformation(Input $D$)

```
1  begin
2      while $|M|$ increasing do
3          while $|C|$ increasing do
4              for All instructions $o$ in $D$ do
5                  $S_o \leftarrow$ successors of $o$;
6                  if $S_o = S_o \cap M$ or $o$ is a select then
7                      $C \leftarrow C \cup \{o\}$;
8          ASAP_Schedule $D$;
9          ALAP_Schedule $D$;
10         for All instructions $o$ in $C$ do
11             if $o$ is a select instruction then
12                 $I_i^o \leftarrow$ if-path input of $o$ ;
13                 $I_e^o \leftarrow$ else-path input of $o$ ;
14                 if $o$ is only successor of $I_i^o$ and $I_e^o$ then
15                     Pack $o$, $I_i^o$, and $I_e^o$ into $P^o$ ;
16                     $M \leftarrow M \cup \{P^o\}$ ;
17             else
18                 $S \leftarrow$ (if-path$\times$ else-path) inputs of $o$;
19                 Sort $S$ by cost of each pair;
20                 if the best cost is positive then
21                     Replace selected pair with $P^o$ ;
22                     $M \leftarrow M \cup \{P^o\}$;
```

Although packing pairs of instructions decreases the total number of nodes in a DFG, it does not affect $RecMII$ at all. It is because reducing the number of nodes through packing does not alter the latency of any path. Therefore, such loops would not benefit from reducing the number of operations because their performance is limited by latency of critical cycles [94]. The second reason is that even if the number of nodes in a DFG is reduced, the number of reduced nodes can be insufficient to decrease $II$. Given a DFG $I = (V_I, E_I)$ and an $M \times N$ CGRA, $ResMII = \lceil \frac{|V_I|}{M \times N} \rceil$. If removing few nodes from DFG does not alter $ResMII$ (because of the non-linear relation between $|V_I|$ and $MII$), packing is unlikely to benefit performance. In this case, BRMap conducts a preliminary mapping first. If the underlying CGRA mapping algorithm finds a mapping at an $II > MII$, only then BRMap starts packing instructions to reduce the number of operations in DFG.

When CGRA mapping algorithms fail to find a mapping for a given $II$, extra nodes (in form of routing and/or recomputing nodes) are added to the DFG. Those extra nodes relax data dependencies between instructions whose data dependencies could not be satisfied in a mapping. However, increasing the number of nodes in a DFG gradually increases $ResMII$ and consequently $MII$ (note that it is non-linear relation). For such cases, reduction in the number of nodes through packing instructions provides further flexibility to the mapping algorithm to add routing and recomputing nodes when a mapping failure occurs.

**Step 2: Packing Pair of Nodes.** The minimization algorithm is presented in Algorithm 22. To conduct DFG minimization, BRMap first schedules the operations. Scheduling determines a partial order for nodes to execute. This order is essential to form dual-issue instructions. If nodes are packed without respecting the partial order of operations in a DFG, it is possible to transform the input DFG to the one for which no feasible schedule exists. Thus, it is crucial to respect partial orders of instruction and pack them carefully.

Consider the following paths in a DFG. $P_1 = (i_1, i_2, s)$ and $P_2 = (i_3, i_4, s)$ and $P_3 = (i_5, s)$. $i_5$ is the predicate Boolean input and $s$ is a select instruction. If pairs $C_1 = (i_1, i_4)$ and $C_2 = (i_2, i_3)$ are chosen, there is no feasible schedule for the transformed DFG. It is because $i_1$ must be scheduled after $i_2$ which implies $C_1 < C_2$. However, $i_3$ must be scheduled before $i_4$ which implies $C_2 < C_1$. Thus, no feasible order for $C_1$ and $C_2$ exists.

To respect partial order of operations, BRMap starts from a select instruction. Each select instruction has three inputs: an input from if-path of an ITE construct, another input from else-path, and a Boolean input to choose among former two. A select instruction along with two inputs from if-path and else-path are the first candidates to form a packed node. The necessary condition to form a packed node is that the schedule window ($t_{ALAP} - t_{ASAP}$, i.e. time between ASAP schedule till ALAP schedule) of the pair overlaps. Please note that if only one of those operations are to be executed at run-time, there is no need to have a select operation.

In Figure 6.4(a), select instruction $e$ and its inputs are packed to form a new node. The transformed DFG is shown in Figure 6.4(b) where three nodes are merged.

Next, BRMap finds the set of input nodes of the packed nodes. Similar to select instruction, there are inputs from if-path and else-path of an ITE constructs to those packed nodes. However, the number of inputs of packed nodes may vary. This is different from select instruction where the number of inputs is always three. At this step, we need to ensure that for any pair of instructions we choose, there is an instruction from if-path and one from else-path. For instance, in Figure 6.4(b), $j$ and $k$ cannot form a pack because they both are within the if-path.

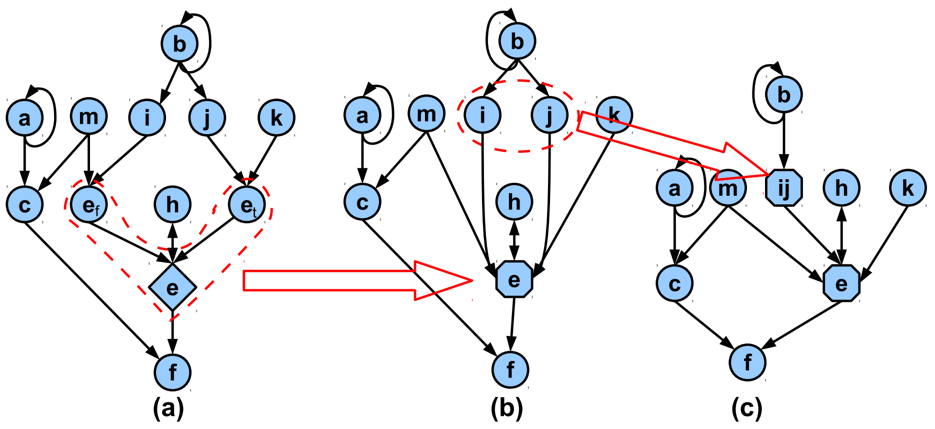


Figure 6.4. (a) shows the input DFG. The first candidate to form a packed node is select instruction $e$ and its inputs. (b) select node and its inputs are replaced with a new packed node. Nodes $i$ and $j$ are the best pair to form next packed node. Nodes $k$ and $m$ cannot be reduced because $m$ is used as an input to instruction $c$ that is not within the loop body. (c) shows the minimized DFG. The number of nodes in the final DFG is reduced by three.

If there are many possible pairs, BRMap attempts to pack a pair that it is easier to place at the mapping step. Let $I_a$ be the set of inputs of node $a$. Consider two candidate nodes to form a packed node $(a, b)$. BRMap finds the intersection of $I_a$ and $I_b$. It also finds the overlap in schedule window of $a$ and $b$. BRMap finds a pair with the maximum intersection and overlap in schedule windows. BRMap packs any pair of nodes that is possible to pack in a greedy manner $(O(n^2))$.

In Figure 6.4(b), nodes $i$ and $j$ in this DFG are packed next. They share input $b$ and they are scheduled at the same cycle. The minimized DFG is shown in Figure 6.4(c). Nodes $m$ and $k$ cannot be packed because node $c$ is not within an ITE construct and there is an arc from $m$ to $c$.

A packed node from a pair of nodes with common inputs is easier to map. For a packed node, the number of dependencies that must be satisfied to find a valid mapping is usually higher than a regular node. When a pair of nodes forming a packed node share an input, the number of dependent nodes that should be placed in neighbering PEs decreases. Therefore, it is easier to place such packs compare with the one without any common input. BRMap iteratively finds pairs of nodes to form packed nodes until no further pair can be found ($O(n^3)$).

**Step 3: Mapping Transformed DFG onto CGRA.** The number of data dependencies after forming packed nodes makes mapping of DFG significantly more challenging than a regular DFG. Thus, it is important to use a constructive CGRA mapping technique. REGIMap is a constructive CGRA mapping algorithm which efficiently utilized resources on CGRA. After DFG minimization, BRMap calls REGIMap to find a valid mapping of DFG on CGRA. We modified REGIMap to allocate registers both in PEs register files and predication network.

# Chapter 7

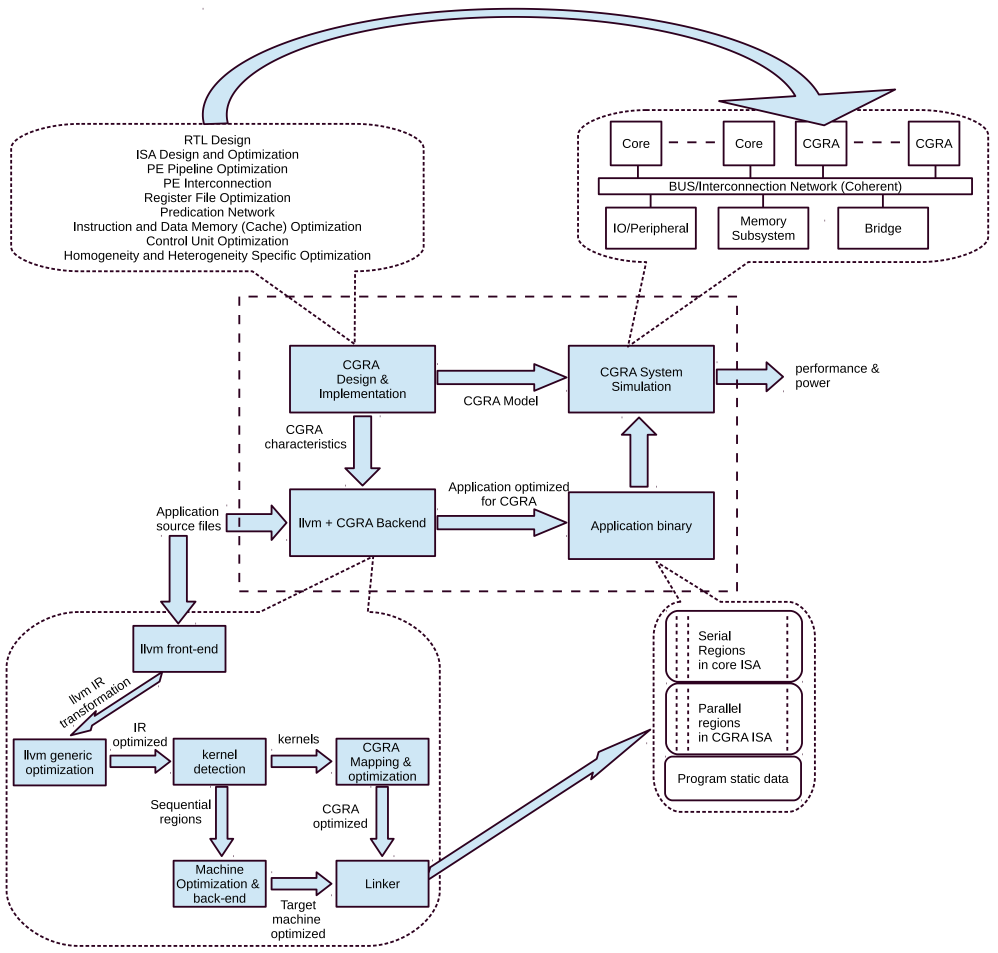
# A FRAMEWORK TO STUDY CGRAS

## 7.1 Introduction

This study requires an extensive infrastructure development to build a system that simulates a CGRA as an accelerator. It also requires to integrate the compiler techniques presented in previous chapters in a compiler framework. This infrastructure is necessary to conduct experiment on and ensure the correct execution of an application on a system equipped with a CGRA as an accelerator.

Three major requirements of this framework include an architectural design and implementation of CGRA, a system level, cycle accurate simulator to run applications, and a compiler to automatically and efficiently map kernels on CGRA. An overview of this framework is depicted in Figure 7.1. Architectural design and hardware implementation is necessary to extract physical characteristics of CGRA. This implementation enables an accurate modeling of CGRA in a computing system simulator. These properties are used to model CGRA as an accelerator in gem5 [12] system. Modeling CGRA in gem5, allows us to evaluate CGRA at system level and measure its benefits as a whole. In addition, it enables us to ensure that compiler techniques presented in this research result in correct execution of input application source codes.

Extracting CGRA characteristic from hardware implementation is necessary for compiler design and optimization too. Without detail information about characteristics of CGRA, latency of different operation, compilers cannot generate a precise and



RTL Design
ISA Design and Optimization
PE Pipeline Optimization
PE Interconnection
Register File Optimization
Predication Network
Instruction and Data Memory (Cache) Optimization
Control Unit Optimization
Homogeneity and Heterogeneity Specific Optimization
Core
Core
CGRA
CGRA
BUS/Interconnection Network (Coherent)
IO/Peripheral
Memory Subsystem
Bridge
CGRA Design & Implementation
CGRA Model
CGRA System Simulation
performance & power
CGRA characteristics
Application optimized for CGRA
Application source files
llvm + CGRA Backend
Application binary
llvm front-end
llvm IR transformation
IR optimized
llvm generic optimization
kernel detection
kernels
CGRA Mapping & optimization
Sequential regions
CGRA optimized
Machine Optimization & back-end
Linker
Target machine optimized
Serial Regions in core ISA
Parallel regions in CGRA ISA
Program static data

Figure 7.1. An overview of different aspects of this research. It includes a design of CGRA, simulation at system level, and compiler development.

correct mapping of application kernels onto CGRA. We integrated the compilation techniques developed in this research in llvm [19] compiler framework.

An important aspect of this framework is to design an interface between components in the system. Most importantly, the interface between CGRA accelerator and processor [30] is designed at system level. At this level, CGRA is envisioned as an accelerator in a computing system where it has to communicate with processor and

[30]Processor implies an active processing element in a computing system usually referred to as core or CPU.

memory subsystem. The interface provides the compiler a set of well-defined system calls to produce binaries so that a well-defined communication interface between hardware and software at one end, and between processor and accelerator at the other end is available. This chapter is devoted to present an overview of this infrastructure used to evaluate compiler techniques presented in this research.

## 7.2 System Simulation

gem5 [12] is a system simulation framework. It has a collection of models for components used to build modern computing systems. This collection includes models of processors, caches, memory, bus, disks, etc. Through a sophisticated software design, interface between components and their detail implementations are separated. Therefore, units such as processors can be configured to behave as one of many commercially available processors. Thanks to this elegant design, a new instruction set can be introduced into the framework using a domain-specific language. Most importunately, many commercial processors have already been modeled in gem5.

This rich simulation framework has motivated us to model CGRA in gem5. During execution, CGRA model is kept inactive until a new CGRA thread is initiated. At this point, CGRA is configured to start execution, that is to fetch instructions from its instruction memory. At every execution cycle, CGRA fetches a packet of instructions [31]. These instructions are sent to PEs for decoding and execution. All memory transactions are collected in the next step to be forwarded to data cache. At the end of a cycle, CGRA controller changes the execution state. Execution states include idle when CGRA is not active, configuration when CGRA is reading

[31] one instruction per PE.

its configuration for next execution phase, epilog where the first few iterations of a loop is executed, kernel when CGRA is executing loop pipeline in steady state, prolog when CGRA is finishing loop pipeline. When the computation is completed, CGRA thread interrupts the main processor. At this point, any thread-wait function that have been called on processor will be waken up and resume.

CGRA configuration plays an important role in this communication model. The most important component of this model is configuration memory. This region acts as a mail box where processor saves dynamic variables such as pointers, control loop related information such as epilog, prolog and kernel length.

In gem5, we instantiate CGRA instruction and data memory as cache. This enables us to use the existing interface for CGRA and the rest of the system. The rest of the system is connected as a regular gem5 configuration. It is important to note that CGRA cores, however, have to be explicitly specified at configuration so a CGRA unit would be instantiated at run-time. In the end of a simulation, we compare the result of a program executed on CGRA with the one that is only executed on the main processor. We verify whether the mapping technique presented in this dissertation maps the loops properly to the CGRA and the program output matches the expected outputs.

## 7.3 Compiler

llvm [19] is a rich set compiler libraries. These libraries can be connected together to form a complete compiler for a target system. Due to this flexibility in llvm, we have introduced several CGRA specific compiler optimizations in llvm. We have designed an instruction set for CGRA that is very close to ARM instruction set [36]. Therefore, we can easily adapt ARM instructions and generate CGRA instruction.

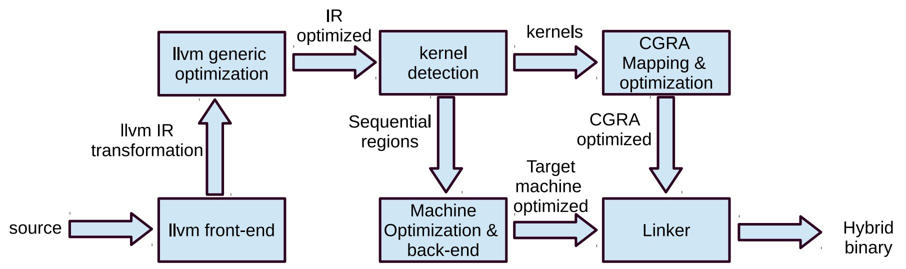


Figure 7.2. Compilation flow for mapping applications.

We adapted ARM backend to generate a new backend in llvm. In this backend, loops nests are analysed to identify the ones that can be efficiently accelerated on CGRA. Those loops are then optimized for CGRA execution.

A new instruction set can be introduced to llvm using a domain-specific language. We introduced CGRA instruction set and extended ARM backend in llvm. At the end of standard optimizations, when all instructions are still in single assignment form (SSA), a new CGRA specific pass is introduced.

A loop nest is represented as a control flow graph (CFG). We would like to construct a data flow graph for each loops nest. If a loop is found profitable for CGRA mapping, a thread is created which hold sequence of instructions generated from that loop to CGRA and that loop will be removed from the main thread CFG.

At the end of this step, CGRA binary generator creates a binary output for the mapping. This binary is inserted in the hybrid binary file generated by the compiler at the end of compilation.

The mapping algorithms presented in previous chapters have been integrated in llvm compiler framework. A new backend is introduced in llvm framework to support compilation for CGRA. This framework transforms loop kernels to CGRA ISA and optimize them for accelerations. The rest of the application is compiled with llvm ARM backend.

Several optimization is performed before modulo scheduling including *loop fission, fusion, interchange, peeling, and unrolling* as well as classical loop optimization is llvm framework. After those optimization, before *Phi* elimination, we construct the data flow graph for the loop. If there are conditional clauses within the loop, operations are predicated. The regular instructions are partially predicated and operations which can cause exception are fully predicated .

## 7.4 REGIMap Evaluation

In this section, we conduct different experiments to evaluate EPIMap and REGIMap for single cycle, multi-cycle, and pipelined CGRAs. Loops that are important for performance are selected from SPEC2006 [51] and digital signal processing applications. The profiling switch is enabled during compilation (gcc -pg and opt -insert-edge-profiling) of benchmarks. We used *gprof* to measure the running time of different regions in applications. Among hotspot regions in the code, we select and accelerate those regions contribute most to total execution time (10% or more). The source code of applications are modified and *C pragma* is used to assist compiler kernel detection unit to identify those loops.

We compare REGIMap with DRESC algorithm [78] and its register allocation extension [20], which we name RA-DRESC throughout this section. DRESC modulo scheduling algorithm is based on a simulated annealing search technique. It is shown [20] that DRESC accelerates loops better than all existing mapping schemes. However, DRESC mapping time is inherently slow due to its semi-exhaustive searching nature. The DFG constructed from optimized loops are sent to both EPIMap(and REGIMap) and DRESC(and RA-DRESC). The mapping results are compared against each other. We implemented EPIMap and REGIMap in C++ and compiled it using clang v3.4 with -O3 optimization. We assume a $4 \times 4$ mesh inter-connected CGRA with PEs capable of executing fixes-point and logical operations. We also assume that the latency of accessing memory is 1 cycle and data bus is shared among all PEs in a row.

### 7.4.1 Optimizing scheduling window factor

During modulo scheduling, a scheduling window is defined for an operation $u$ between the cycle ASAP and ALAP cycles associate to operation $u$. In Chapter 4, we introduced a scale factor ($C$) to scheduling window. This factor gives us more flexibility to associate an execution cycle to an operation. In our experiments, we observe that a scaling factor between 3 and 5 is sufficient to achieve more than 90% success rate in modulo scheduling operations at $MII$. Note that we measured this rate before placement, thus this number does not reflect the success rate of mapping at $MII$. When $C$ is reduced to 1, equivalent to the original scheduling window, the modulo scheduling success rate at $MII$ decreases dramatically to less than 40%. When the scheduling window factor is set to 5, it has less than 1% overhead in total compilation time. In the rest of our experiments, we use this configuration for EPIMap and REGIMap.

### 7.4.2 Optimizing the number of clique search attempts

Placement is reduced to finding the maximum clique is compatibility graph, which is a well-known NP-Complete problem. Exhaustive search for the maximum clique imposes a significant compilation time overhead. Instead, we use dynamic programming approach where we keep track of the best solution (largest cliques). Every time the

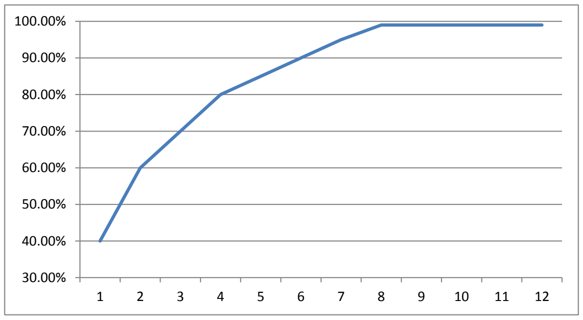


Figure 7.3. The success rate of fining maximum clique. The X-axis is the number of consecutive failed attempts in increasing the maximum clique size. When EPIMap fails to increase the size of maximum clique 8 consecutive times, it is 98% likely that the maximum clique size has already been reached.

upper bound on the size of a candidate clique is no better than best clique so far, we stop expanding that candidate clique. EPIMap stops searching for new candidates when in 8 consecutive attempts, the size of a newly found clique candidate is no better than best solution.

We selected 8 because in our experiments, we observed that in more than 98% of times, if we fail to increase the size of clique after 8 consecutive attempts, there is no clique with larger size. When we conduct an exhaustive search for largest clique, in 1 out of 50 loops, the size of maximum clique increased. When we increased this number to 16, we did not observe any improvement in final clique size while loop compilation time increased substantially. For the rest of our experiments, we use this configuration (8) in our placement algorithm.

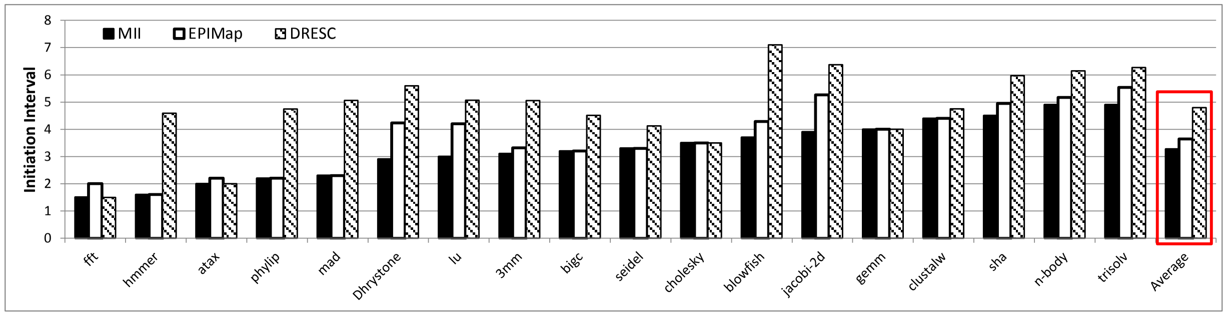
### 7.4.3 EPIMap performance

In this part, we conduct experiments to compare the performance of mapping loops using EPIMap and DRESC. In order to compare the performance of mapping those loops, $II$ is selected as our performance metric. In fact, the execution time of a loop is proportional to its mapping $II$. To make this comparison independent of register allocation, we set the size of local register files to 64. This number is selected because in our experiments, we observed that 64 registers are sufficient to map all of those selected loops. Note that such a large local register file is clearly not feasible in practice. We used EPIMap and DRESC mapping algorithms without any register allocation to compare their efficiency on base mapping only.

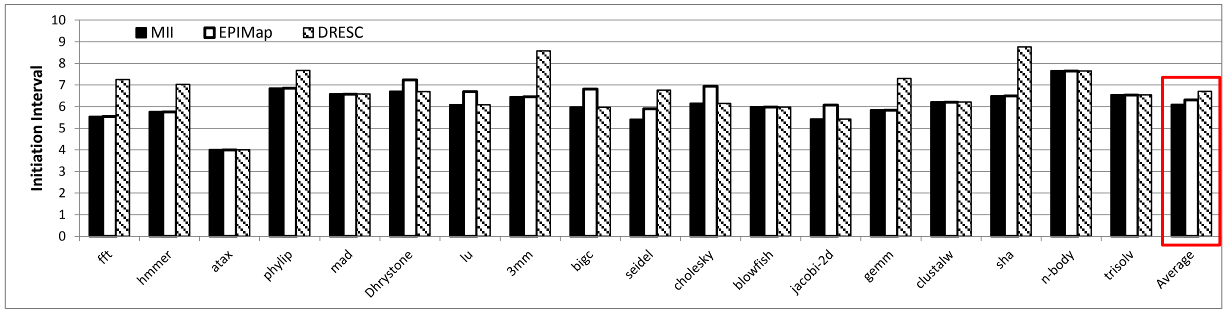
In this experiments, we first assume that all instructions have single cycle latency. Later, we assume multi-cycle implementation of a multiplier with latency of 4 cycles. Finally, multipliers are pipelined with latency of 4 cycles and 1 cycle throughput. We compare the $II$ of mapping loops using EPIMap and DRESC loops in various applications. Those results are compared with $MII$ for those loops too.

As depicted in Figure 7.4(a), loops mapped at $II$ that is very close to $MII$ when we use EPIMaps and it is much lower than mappings generated by DRESC when all instruction can be executed in 1 cycle. On average EPIMap finds mapping at an $II$ that is about 11% higher than $MII$. On the other hand, those loops mapped with EPIMap run on average 31% faster than those mapped using DRESC.

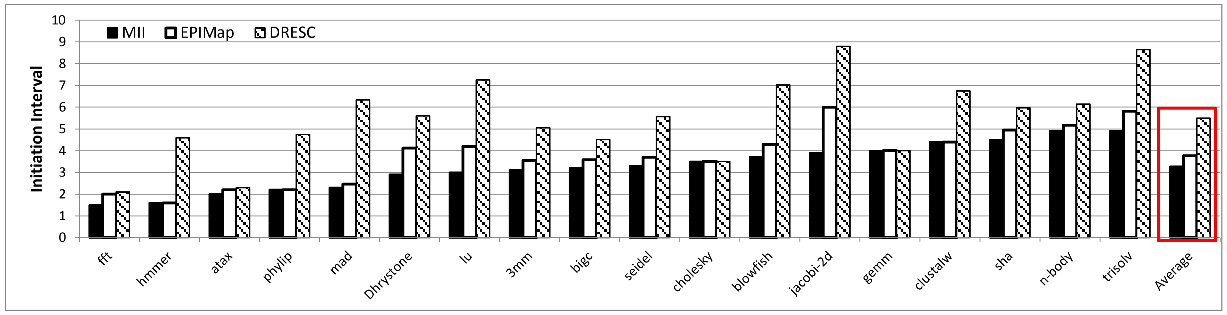
In Figure 7.4(b), a multi-cycle multiplier is integrated at each PE with latency of 4 cycles and EPIMap and DRESC are used to generate mappings. In a multi-cycle implementation, generally the $MII$ is limited by latency of multi-cycle operations. For such implementation, EPIMap performs even better than single cycle implementation and generates mapping at $II$s about 3% higher than $MII$. Loops mapped using DRESC algorithm in a multi-cycle ALU implementation also performs slightly better than single cycle case. This is because in a multi-cycle CGRA implementation, the resource (PE) utilization is much lower than single cycle implementation, thus mapping is much easier. On average, loops are accelerated 10% faster when EPIMap is used compare to when those loops are mapped using DRESC.



(a) Single cycle latency.



(b) Multi-cycle latency.



(c) Pipelined ALUs.

Figure 7.4. Performance of loops mapped using EPIMap vs DRESC for cases on single cycle latency, muti-cycle and pipelined ALUs.

In Figure 7.4(c), multipliers in ALUs are pipelined with latency of 4 cycles and 1 cycle throughput. Those loops are mapped at $II$s which is on average about 15% higher than $MII$ when EPIMap is used. Note that multi-cycle and pipeline ALU implementation imposes a significant communication overhead between operations. It is because the variation between completion of instruction increases significantly and intermediate results have to be communicated between various PEs over multiple cycles. Those loops mapped with EPIMap can be executed on average 45% faster compared to when they are mapped using DRESC. We conclude that EPIMap is more effective than DRESC in accelerating loops when local register files are sufficiently large.

### 7.4.4 EPIMap success rate at first mapping attempt

Scheduling plays an important role in CGRA mapping. An effective scheduling can actively avoid the cases for which a placement is not feasible. More importantly, a feasible minimum II can be accurately estimated when the order of operations and their communication is taken into account well. In fact, when data communication between operations is high, it imposes significant routing overhead which requires to include extra routing operation in DFG. This can increase $II$ while it is unknown when $MII$ is approximated initially. It is because this overhead is not observed in $MII$ calculation. The search time between lower bound $II$ calculated after scheduling and using $MII$ at the starting point for mapping plays an important role in making mapping time faster.

In our experiments we observe that about 59% of selected loops are successfully mapped in the first placement attempts as shown in Figure 7.5 in EPIMap. We conclude that scheduler assigns modulo cycle to operations which results in high success rate in placement. In addition, it does not impose large overhead by adding many routing nodes which increases effective minimum $II$. More importantly, this shows that resource conflicts between operation are effectively avoided during scheduling.

It is also important that the scheduling algorithm does not over estimate lower bound $II$. It is because this over estimation results in mapping loops at higher $II$ thus losing performance. To verify that, we implement an exhaustive mapping search that starts at $MII$ rather than starting from minimum $II$ computed during scheduling. We did not find any loop for which a mapping can be found lower than what scheduler outputs as lower bound $II$.

### 7.4.5 Re-scheduling is effective

Placement is a time consuming process, and a failure in allocating PEs imposes a significant overhead in loop compilation time. On the other hand, placement can provide important information to the scheduler about which nodes could not be placed as well as resource conflicts between operations. A scheduler can effectively use such information and re-schedule operations to actively avoid failures in next mapping attempts.

We measure the percentage of mapping success after a loop is scheduled until it is successfully placed (between line 11-16 in Algorithm 2). We observe that the placement success rate is about 86% in EPIMap (and REGIMap). The success rate without re-scheduling is only 59%. This is a direct impact of *re-schedule* function in EPIMap (and REGIMap) which enables EPIMap to adjust the priority of the nodes based on input DFG characteristics.

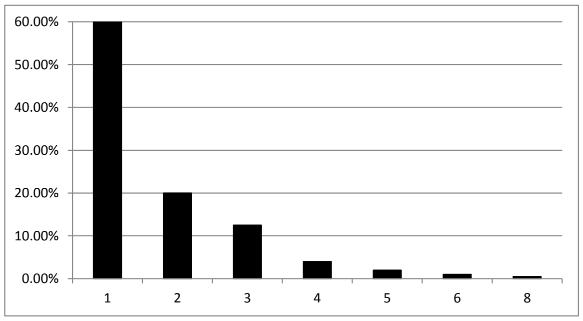


Figure 7.5. The success rate of placement. 59% of loops are successfully mapped in first placement attempt. In first 3 placement attempts, more than 90% of all loops are successfully mapped.

### 7.4.6 Constraining resources during scheduling is effective

When placement and rescheduling both fail, EPIMap uses another heuristic instead of directly increasing $II$. At this step, EPIMap conducts a new phase of scheduling while reducing the number of available PEs at all cycles by 1 (line 17 in Algorithm 2). This heuristic may increase the schedule length for operation not in critical cycle. However, increasing latency of one iteration of the loop does not directly increase $II$. In our experiments, we observed that 70% of loops failed in placement and re-scheduling attempts, could eventually be mapped without any increase in $II$.

### 7.4.7 Placement and register allocation search space

Here, we demonstrate the compilation time benefit of register allocation model introduced in REGIMap algorithm. As it is discussed in Section 5.3, PE and register allocation can be done simultaneously if registers, too, are replicated in resource graph. However, when registers are replicated in resource graph, it incurs a substantial overhead in the number of nodes both in resource graph as well as in compatibility graph.

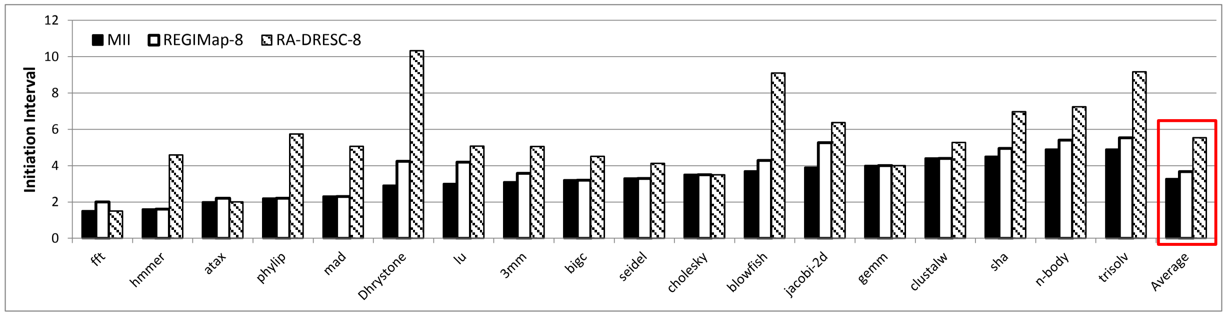
Formulating registers at the weight of arcs, as is done in REGIMap, enables us to effectively allocate registers with minimal overhead in placement. To show that, we slightly modified EPIMap to replicate registers in resource graph. In the end, the placement of operations would change to allocating PEs and registers during the course of searching for the largest clique. Our experiments show that such resource allocation imposes between 16X to 52X compilation time overhead when the size of local register file is only 4. When REGIMap is used, all loops are mapped at the same $II$ as the modified EPIMap did. We conclude that register allocation model in REGIMap significantly reduces the compilation time of loops compared to original technique based on MCS technique.

We also compared the overhead of integrated placement and register allocation using REGIMap with EPIMap placement only approach (arcs in compatibility graphs are binary). We observed that when the size of register files are set to 64 (64 registers are sufficient to allocate registers for all loops), the compilation times of EPIMap and REGIMap are close. In fact, on average, the compilation time increases only by 2.29% in REGIMap. We conclude that REGIMap does not impose significant compilation time overhead to placement while it extends the mapping operations to simultaneously allocating PEs and registers.

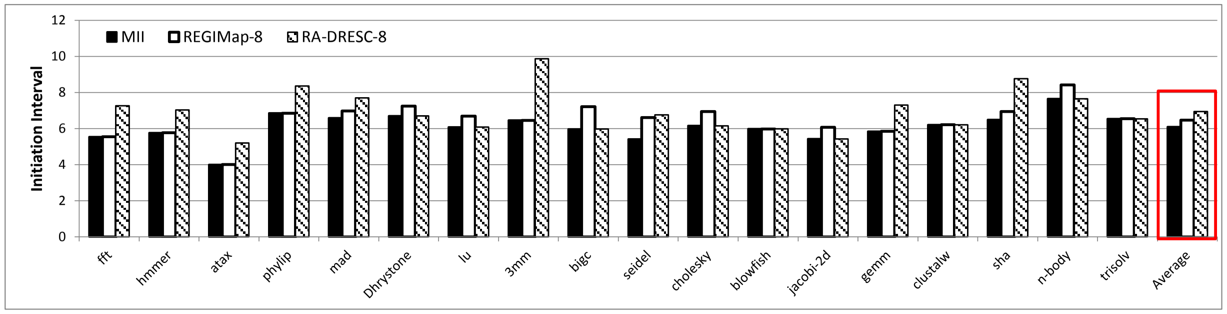
### 7.4.8 Register allocation performance

Here, we evaluate the performance of loops mapped using REGIMap and RA-DRESC with different number of registers. In this experiment, we would like to compare the effectiveness of mapping as well as register allocation together.

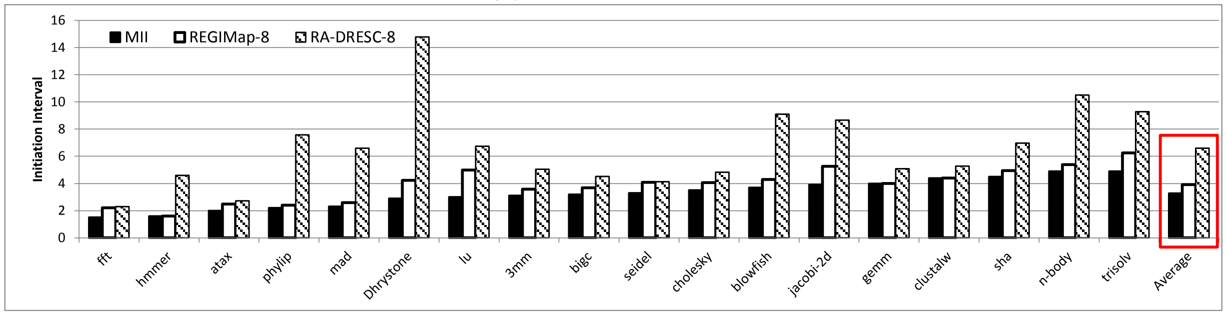
In the first configuration, we set the size of local registers to 8. Then, we evaluate the performance of those mapping techniques when PEs execute all instruction in one



(a) Single cycle latency.



(b) Muti-cycle latency.



(c) Pipelined ALUs.

Figure 7.6. Performance comparison of loops mapped using REGIMap and Register-Aware DRESC when size of register file is 8 for single-cycle latency, multi-cycle and pipelined ALUs.

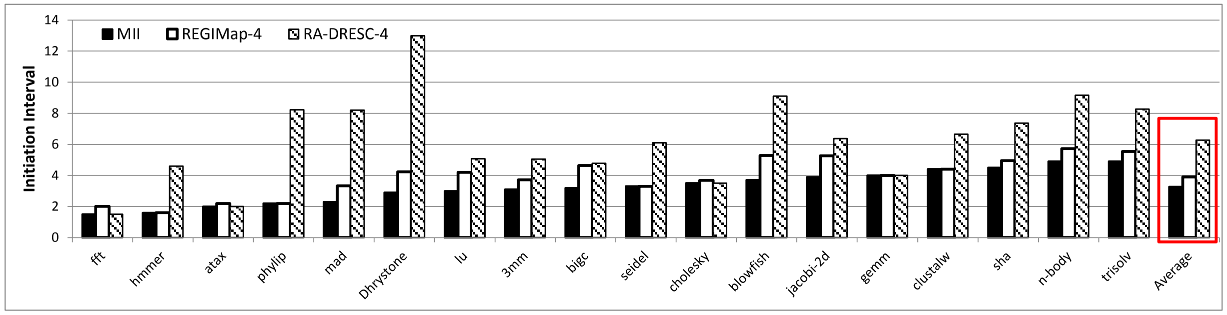
cycle. As shown in Figure 7.6(a), REGIMap accelerates loops at lower $II$ than DRESC across most benchmarks with exception of *fft*. On average, loops are accelerated 50% faster when we use REGIMap compared to RA-DRESC mappings.

In next configuration, multipliers use multi-cycle implementation to save area. For this CGRA configuration, loops are accelerated at relatively close performance when REGIMap and RA-DRESC are used. Details are depicted in Figure 7.6(b). We expected such results because the resource utilization of mappings on multi-cycle CGRA configuration is low. Thus, it is easier to find a mapping and both mapping schemes show consistently similar mapping results. Those mappings, in fact, heavily rely on routing nodes and registers to satisfy data dependency between operations. Since the are many spare resources in multi-cycle implementation case, it is relatively easy to find a mapping.

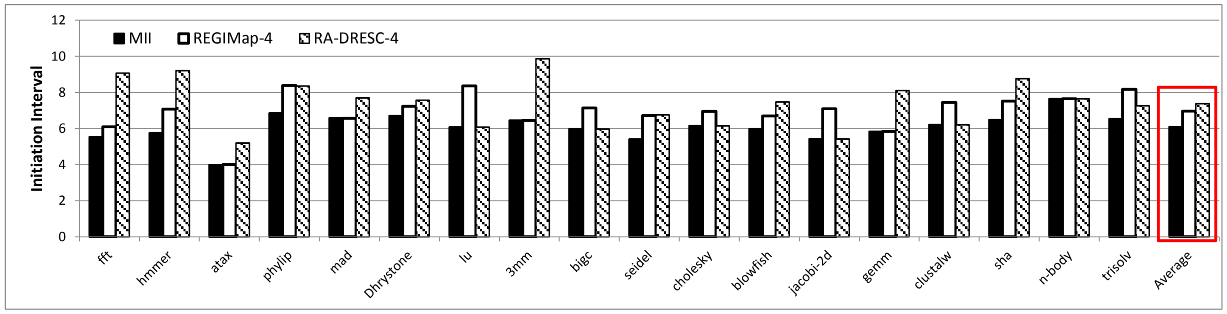
Finally, we compare both mapping schemes when ALUs are pipelined. For such configuration, REGIMap consistently maps loops at lower $II$ than RA-DRESC. On average, as is shown in Figure 7.6(c), loops mapped using REGIMap execute 68% faster than when RA-DRESC is used.

As the number of registers decreases in CGRA, effective register allocation becomes extremely important for a successful mapping. In next configuration, we reduce the number of registers to 4 and compare performance of mappings using REGIMap and RA-DRESC. For a single cycle CGRA implementation with 4 registers, REGIMap accelerate loops 1.6 times faster than when they are mapped using RA-DRESC. REGIMap accelerates on average loops 5% and 81% faster than RA-DRESC for multi-cycle and pipelined CGRA implementation, respectively. Results are presented in Figure 7.7.

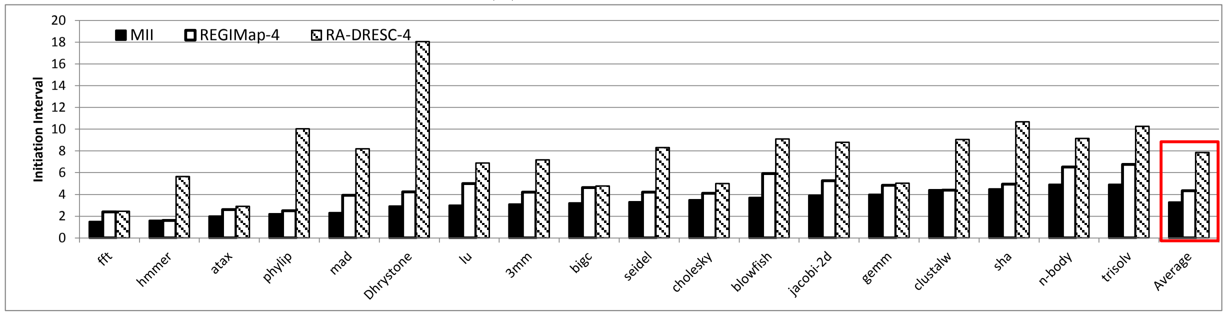
When registers are scarce, the performance ($II$) of mappings tightly depends on register allocation effectiveness. In such case, the mapping $II$ generally increases



(a) Single cycle latency.



(b) Multi-cycle latency.



(c) Pipelined ALUs.

Figure 7.7. Performance comparison of loops mapped using REGIMap and Register-Aware DRESC when size of register file is 4.

because data dependencies are satisfied with frequently using routing nodes in addition to registers. The results shown in Figure 7.8 demonstrate that REGIMap utilizes register much better than RA-DRESC. In a single cycle CGRA implementation, REGIMap accelerate loops on average 73% faster than RA-DRESC when each PE has 2 local registers.

Mappings generated for a multi-cycle implementation of CGRA generally require more registers than single cycle implementation. For such implementation, registers are usually used for transferring data between dependent operations. However, because $MII$ is mapping loops for such CGRA is limited by latency of multi-cycle operation, PEs are under-utilized. Thus, using PEs to route operation between dependent operation does not degrade performance. REGIMap and RA-DRESC accelerate loops for multi-cycle CGRA with similar acceleration factor as shown in Figure 7.8(b).

Mappings generated for pipelined CGRAs, too, require extensive register usage. REGIMap for such CGRAs accelerates loops on average about 83% faster than mappings generated by RA-DRESC as shown Figure 7.8(c). We conclude that REGIMap utilizes local registers better than RA-DRESC and accelerates loops better.

### 7.4.9 Multi-cycle implementation severely damages performance

Throughout our experiment, we observe that using multi-cycle functional units severely damages performance and increases mapping $II$s. Since CGRAs are generally used as accelerators, performance benefit is by far the most important factor. On average, $MII$ decreases by 53% when we use a pipelined implementation as compared to a multi-cycle one. Due to this fact, we continue our experiments with single cycle and pipeline CGRA configurations.

### 7.4.10 Compilation time of loops with EPIMap

Next, we compare the running time of EPIMap and DRESC when mapping loops on different CGRA implementations. We observe that REGIMap accelerates loops faster at significantly lower compilation time. On average, in a single cycle implementation, EPIMap finds mappings on average 138 times faster than DRESC. In a pipelined CGRA, EPIMap mapping time is 192X faster than DRESC. We observe a consistent faster compilation time when REGIMap and RA-DRESC are used for mappings. The gap between compilation time substantially increases as the number of registers decreases. Details are presented in Figure 7.9.

### 7.4.11 REGIMap Compilation time scales well with register file size

An important property of a good compilation scheme is to show a consistent execution time over different architectural configuration. In this part, we present the running time of REGIMap and RA-DRESC when the size of local register files changes for both single cycle and pipeline CGRA implementation. In Figure 7.10, the compilation time using REGIMap and RA-DRESC to map loops are shown for different register files sizes in single cycle CGRA implementation.

We observe that loops are mapped consistently at a significantly lower compilation time using REGIMap compared to RA-DRESC. In addition, we observe that as the size of register file decreases, the compilation time increases slightly in REGIMap. We expect this increase because when less number of registers are available to transfer data between dependent instruction, extra nodes are allocated for routing purposes. As the number of nodes increases, both $II$ and the compilation time increase because the size of the problem increases. We can observe the similar increase in compilation time when RA-DRESC is used. However, it occurs at a significantly higher overhead.

Mapping loops for pipelined CGRA imposes overhead in using registers. Therefore, we expect higher increase in compilation time as the number of registers decreases. The results are presented in Figure 7.11. We observe that the compilation time of loops using REGIMap and RA-DRESC both increases with decreasing the number of registers. However, compilation time using RA-DRESC algorithm is significantly higher than REGIMap. Note that the Y-axis is scaled exponentially.

### 7.4.12 Pipelining is effective

Pipelining is an effective optimization in hardware implementation to improve performance and frequency. Thus, a mapping algorithm should effectively map loops for such CGRA implementation. On the other hand, as discussed before, pipelining imposes a significant overhead in register usage. Therefore, we expect an increase in $II$ of mappings for a pipeline CGRA compared to a single cycle CGRA implementation.

The $II$ overhead imposed by pipelining CGRA implementation in EPIMap (and REGIMap) and DRESC (and RA-DRESC) are shown in Figure 7.12 for different register file sizes. EPIMap accelerates loop with on average only 0.11 increase in $II$. DRESC, on the other hand, compiles loop with an average 0.69 increase in $II$. We conclude that EPIMap supports pipelining much better than DRESC. There are cases that mapping in pipelined CGRA results in decrease in $II$ such as *Dhrystone*. This occurs in applications with imbalance data dependency between operations. Variation in latency of executing operations makes data dependencies in those loops more balanced.

This overhead in $II$ increases as the size of local register files decreases. However, loops mapped using REGIMap consistently show lower overhead of pipelining as compared to RA-DRESC. We conclude that REGIMap successfully handle communication overhead of pipelining while RA-DRESC shows less tolerance.

### 7.4.13 Pipelining compilation time overhead

Compilation time is an important factor in mapping. The compilation time overhead of supporting and mapping loops on pipelined CGRAs is shown in Figure 7.13. The Y-axis shows this overhead in percentage. Supporting pipelining imposes only 7% compilation time overhead on average to EPIMap. This overhead is about 78% on average in DRESC.

The average overhead is 32%, 45%, and 31% in REGIMap for CGRA with 8, 4, and 2 local registers. On the other hand, this overheads increases in RA-DRESC and on average is 105%, 147%, and 38%. We conclude that REGIMap compilation time increases modestly to support pipeline CGRAs.

## 7.5 Supporting Loops with Conditionals

In this section, we evaluate the performance impact of various transformations introduced in Chapter 6 to support acceleration of loop with conditionals. We conduct our experiments to explore the performance of the various architectural and compiler techniques for handling conditionals in CGRA. We map the loops on a $4 \times 4$ CGRA with sufficient instruction memory to hold all instructions within a loop body as well as sufficient data memory space to hold all the variables. Latency of all operations are assumed only one cycle. Load and store operations requires two CGRA operations, one for address bus transaction and the other for data bus transaction. The bus is shared among all PEs within a row; in other words, only one memory transaction can proceed at any cycle in a row. We conduct our experiments on mesh-interconnected CGRA and then we enrich interconnection further with diagonal connections between PEs.

### 7.5.1 Need for supporting Conditionals in Loops

Our first evaluation demonstrates the importance of supporting conditions within loops. If conditional constructs are not supported, many performance-significant loops cannot be accelerated on CGRAs. As shown in Figure 7.14, about 40% of loops that can be executed on CGRA have at least one ITE construct within the body of the loop. Here, the condition we are referring to is different from the main loop condition which controls the number of times a loop would be executed. We are referring to an ITE construct in the loop. This plot is extracted after -O3 optimization in llvm.

### 7.5.2 Performance of dual-issue scheme

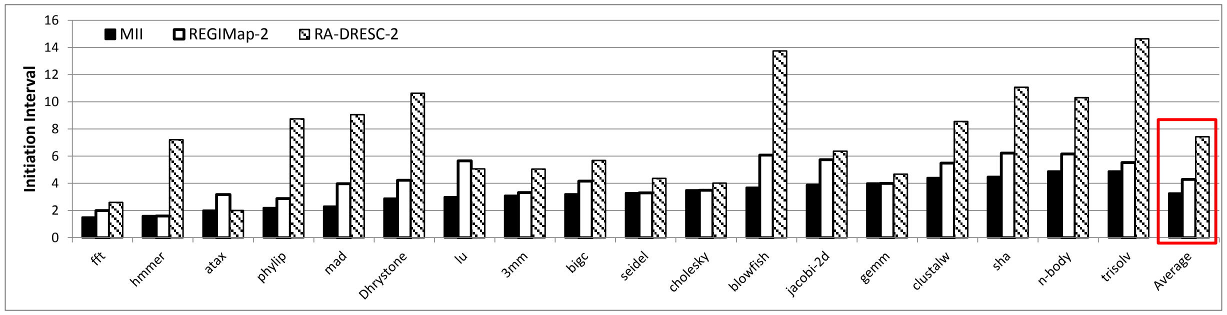
Next, we compare different techniques to accelerate loops with conditionals, namely: full predication approach presented in [42], full predication, partial predication, and BRMap for dual-issue. Figure 7.15 plots the achieved $II$ of the loops by different schemes. The bars on the right corner show the average $II$ achieved over all the loops by the techniques. This figure shows that the dual-issue approach leads to the best acceleration (least $II$) among all the techniques. The full predication approach presented in [42] performs the worst, since it is highly restrictive – all the instructions inside the conditionals have to be mapped to the same PE – this results in long schedule length, and long $II$. Our approach for full predication performs better, primarily because it is less restrictive. The restriction is that the operations in different branches that are generating the same output must be mapped to the same PE. Partial predication performs better than both of these, since it does not add restrictions in mapping, only adds more nodes to the graph. However, dual-issue scheme performs best, since it neither adds restrictions, nor extra nodes in the graph. Overall dual-issue architecture can improve $II$ by almost 42% as compared to the full predication scheme proposed in [42].

### 7.5.3 Dual issue scheme and CGRA interconnect

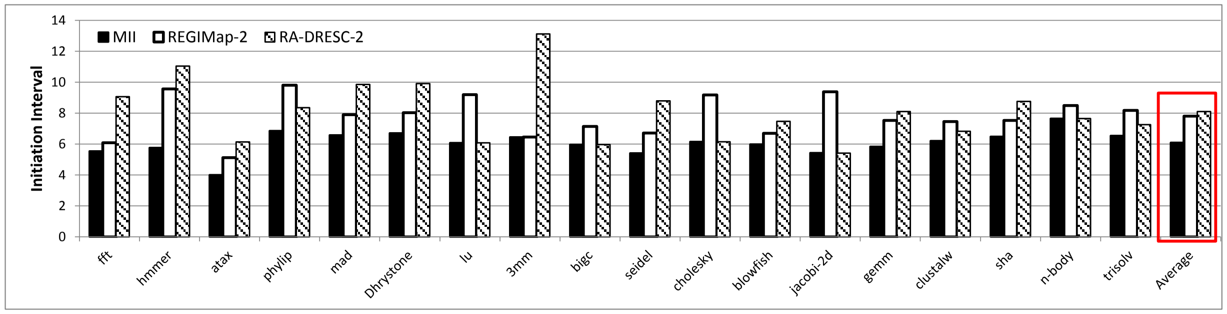
As interconnection is enriched with diagonal connections, the full predication scheme presented in [42] does not improve considerably (only about 0.7% on average). This is because this technique constrains ITE constructs to be executed sequentially, and does not benefit from either more PEs nor richer interconnect between PEs. Full predication gains the most performance benefit from diagonal connections, (about 7% on average). This is because many instructions in this scheme requires three dependencies to be satisfied in mapping. Because of the high data dependency between operations, it is more likely that all dependencies cannot be satisfied in mapping in a mesh-interconnected CGRA. Therefore, mapping fails and more routing nodes are needed to be inserted in the DFG. This eventually leads to more frequent increment in $II$. Partial predication achieves modest benefits from better interconnection (on average about 4.5%). Dual-issue architecture gains 6.7% performance benefit from better interconnection because dual-issue instructions have the highest data dependency among all instructions. Therefore, when more communication channels are available, data dependencies are more likely to be met in mapping. However, even in a richly connected CGRA, dual-issue architectures achieves the lowest $II$, and (therefore) the best performance.

## 7.6 Performance projection

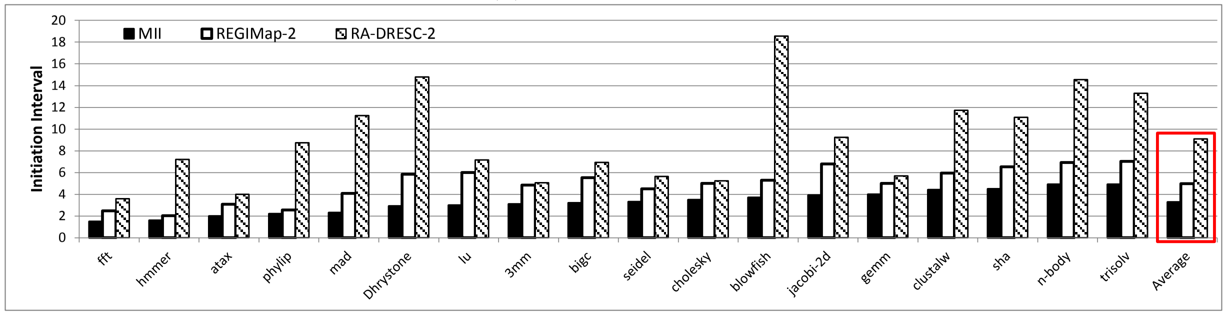
In many applications, a loop mapped using REGI/EPIMap algorithm can reach up to 14 instructions per cycle (IPC). However, this is an upper bound for IPC and memory subsystem performance and latency can greatly change that. For example, in inner-most loop of 2-dimensional matrix multiplication, there are 8 operations and it can be mapped with an $II = 1$, that is equivalent to one iteration per cycle. Therefore, the upper bound on IPC is 8, which is close to what is observed (7.9 IPC) during simulation. This is because the cache hit rate is 98.3% with Access Map Pattern Matching (AMPM) [53] prefetcher for a 16KB 4-way set associative cache. IPC linearly decreases with increasing memory access latency or decreasing cache hit rate. For example, we reach IPC of 0.96 when cache hit rate reduces to 12%. Due to this great sensitivity of CGRA performance to memory latency, it is very important to optimize memory subsystem for the target application with minimal access latency and sufficient bandwidth.



(a) Single cycle latency.

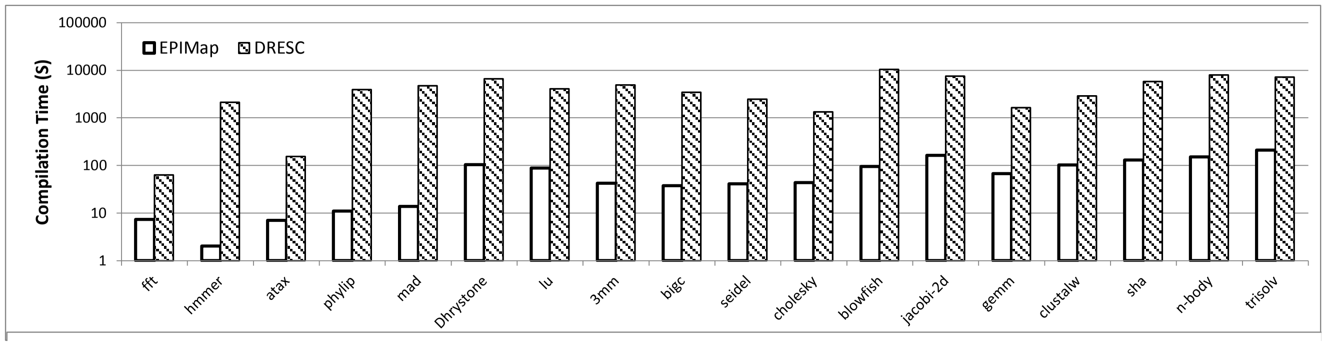


(b) Muti-cycle latency.

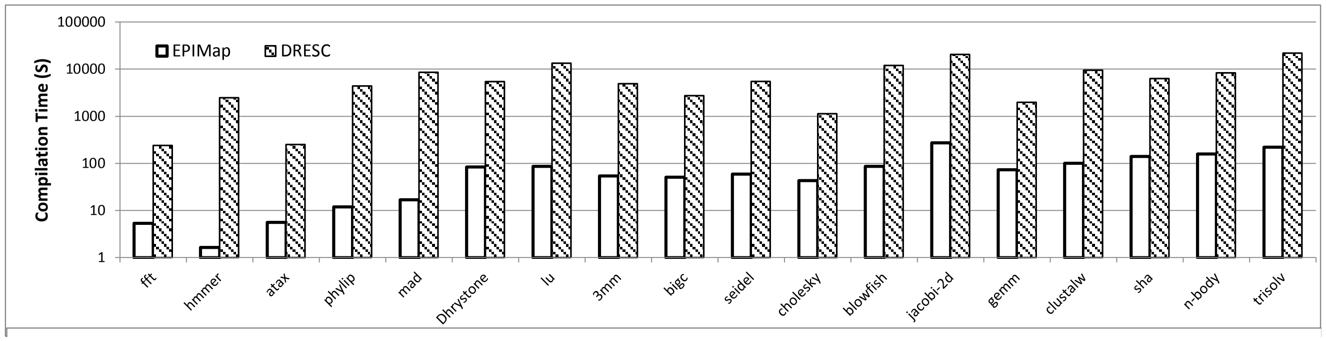


(c) Pipelined ALUs.

Figure 7.8. Performance comparison of loops mapped using REGIMap and Register-Aware DRESC when size of register file is 2.

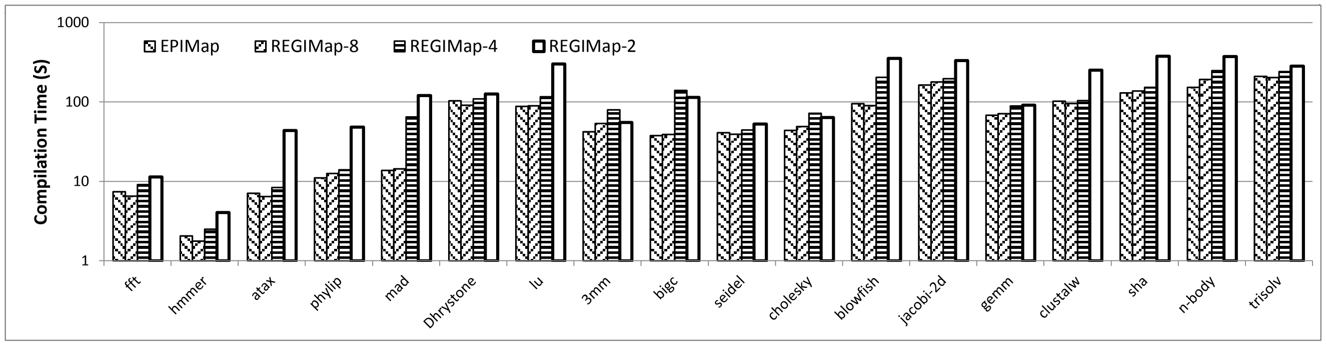


(a) The compilation time of loops using EPIMap and DRESC algorithms for a single cycle CGRA implementation.

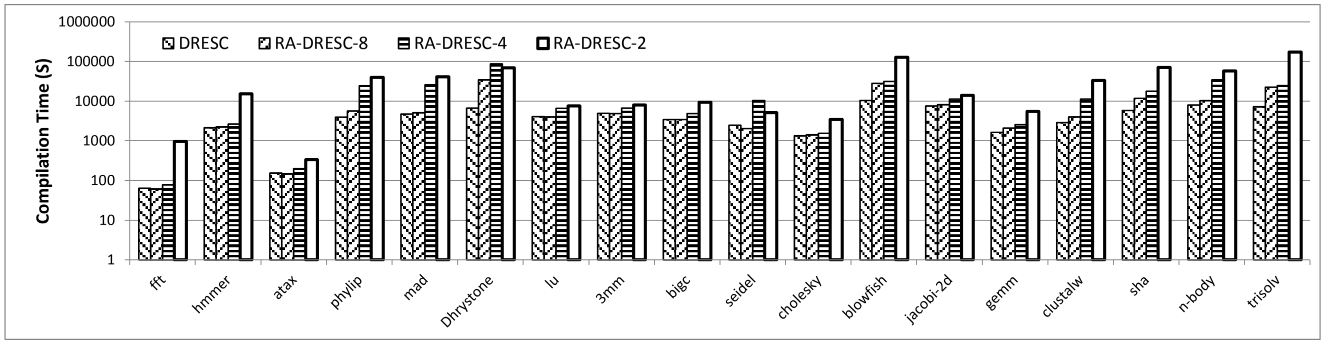


(b) The compilation time of loops using EPIMap and DRESC algorithms for a pipelined CGRA implementation.

Figure 7.9. The compilation time of loops using EPIMap and DRESC algorithms for different CGRA implementation.

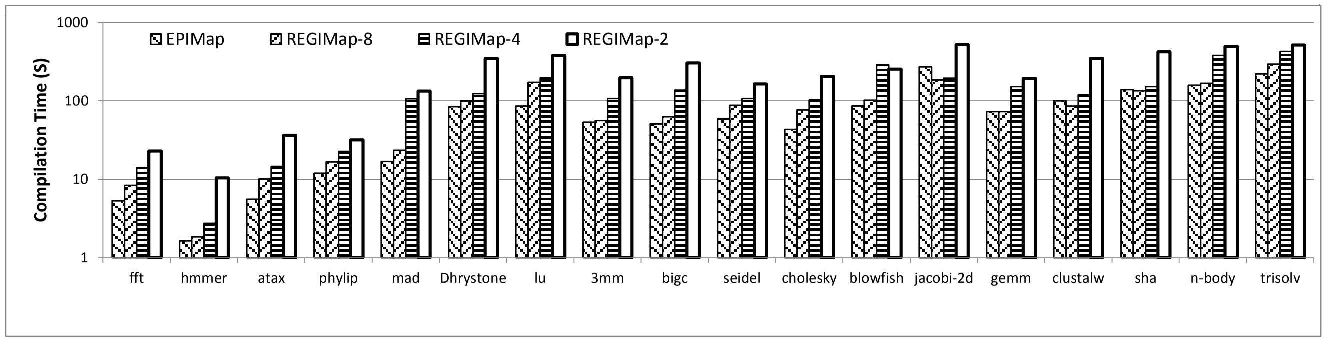


(a) The compilation time of loops using REGIMap for different register file size is shown in a single cycle CGRA implementation.

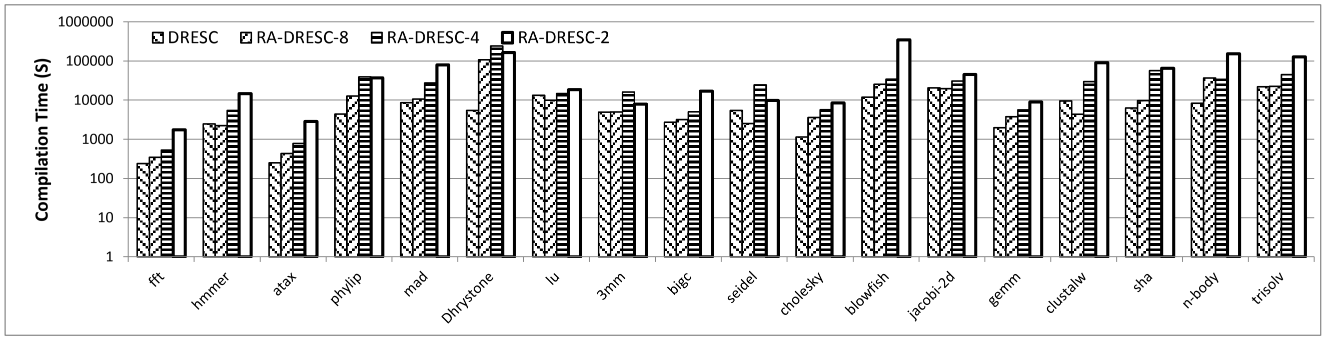


(b) The compilation time of loops using RA-DRESC for different register file size is shown in a single cycle CGRA implementation.

Figure 7.10. The compilation time using REGIMap and DRESC for loops in single cycle CGRA implementation is presented.

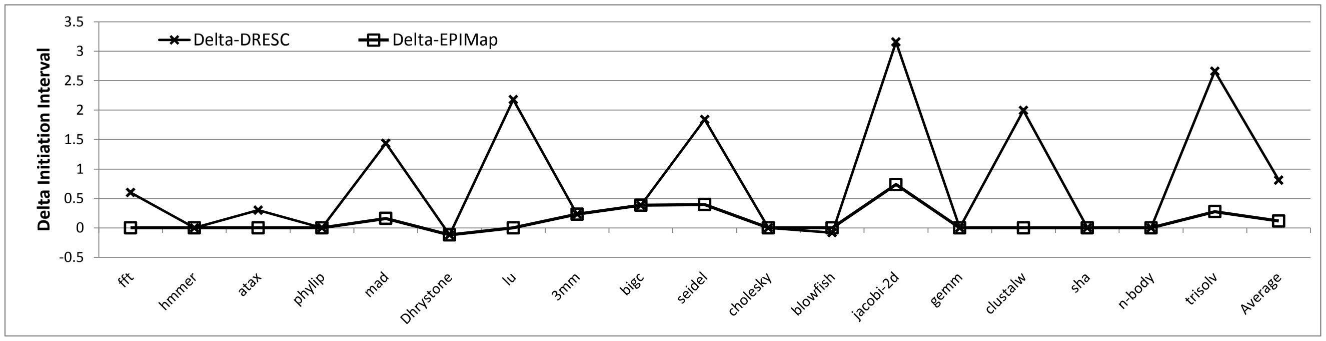


(a) The compilation time of loops using REGIMap for different register file size is shown in a pipeline CGRA implementation.

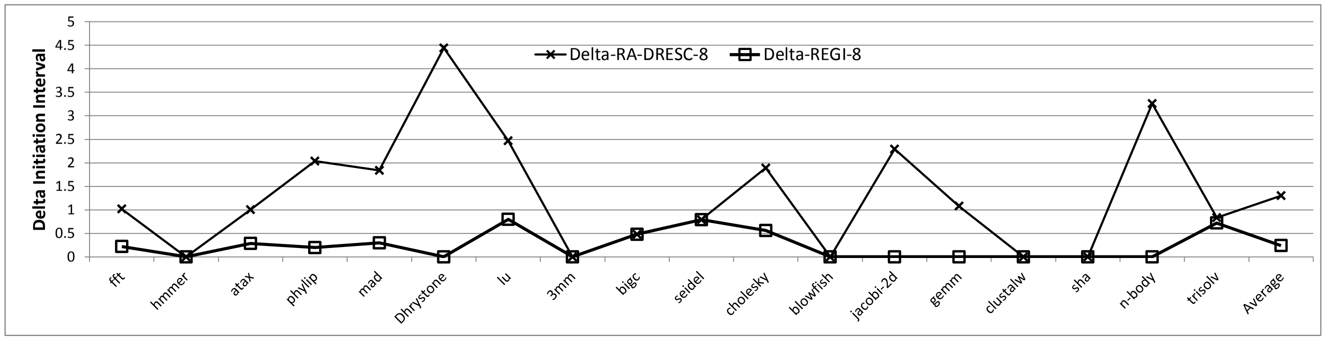


(b) The compilation time of loops using RA-DRESC for different register file size is shown in a pipeline CGRA CGRA implementation.

Figure 7.11. The compilation time using REGIMap and RA-DRESC for loops in pipeline CGRA implementation is shown.

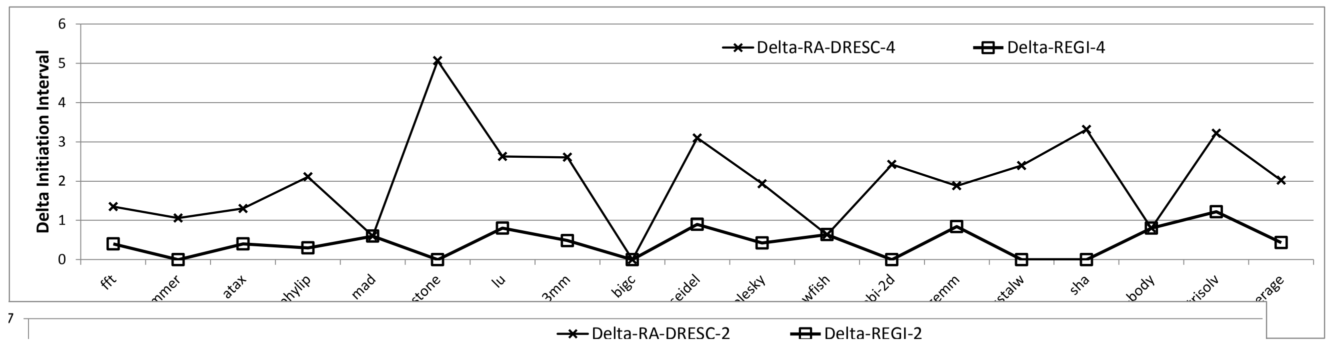


(a) A comparison between overhead of supporting pipeline CGRA in $II$ of mappings in EPIMap and DRESC.

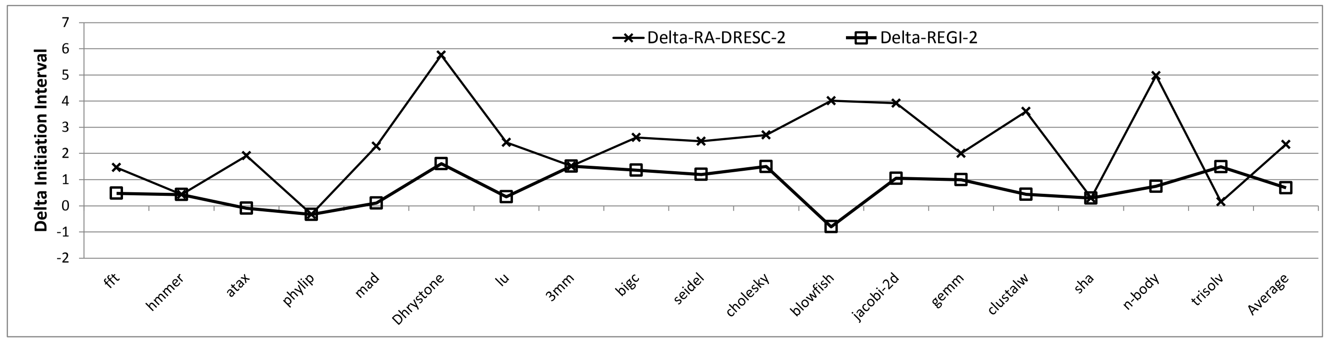


(b) A comparison between overhead of supporting pipeline CGRA in $II$ of mappings in EPIMap and DRESC when each PE has 8 registers.

*(continued)*

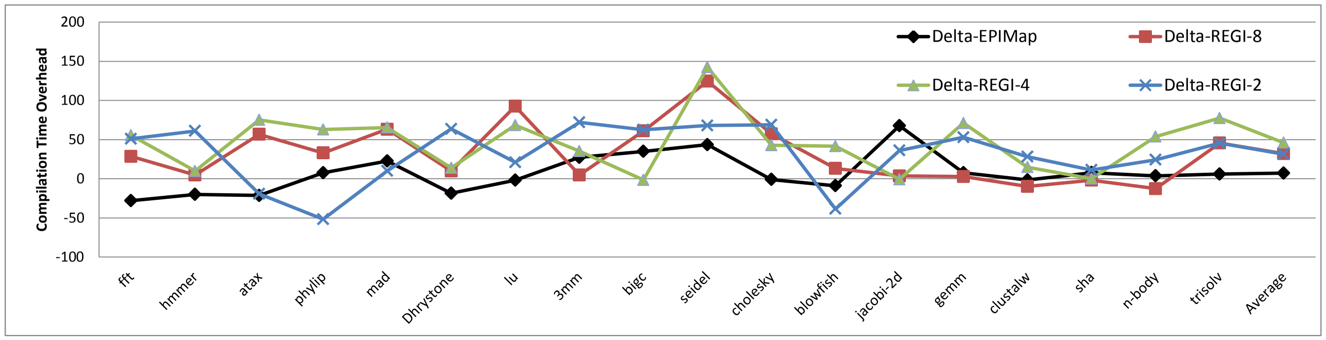


(a) A comparison between overhead of supporting pipeline CGRA in $II$ of mappings in EPIMap and DRESC when each PE has 4 registers.

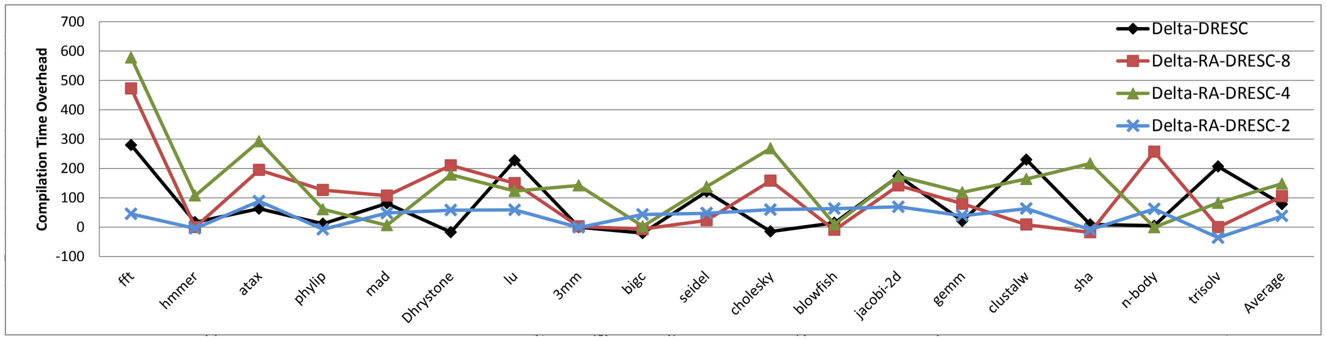


(b) A comparison between overhead of supporting pipeline CGRA in $II$ of mappings in EPIMap and DRESC when each PE has 2 registers.

Figure 7.12. The overhead of supporting pipelined CGRAs in $II$ of mappings.



(a) The overhead of supporting pipelined CGRA in compilation time of loops in REGIMap.



(b) The overhead of supporting pipeline CGRA in compilation time of loops in DRESC.

Figure 7.13. The overhead of mapping loops onto pipeline CGRA in percentage.

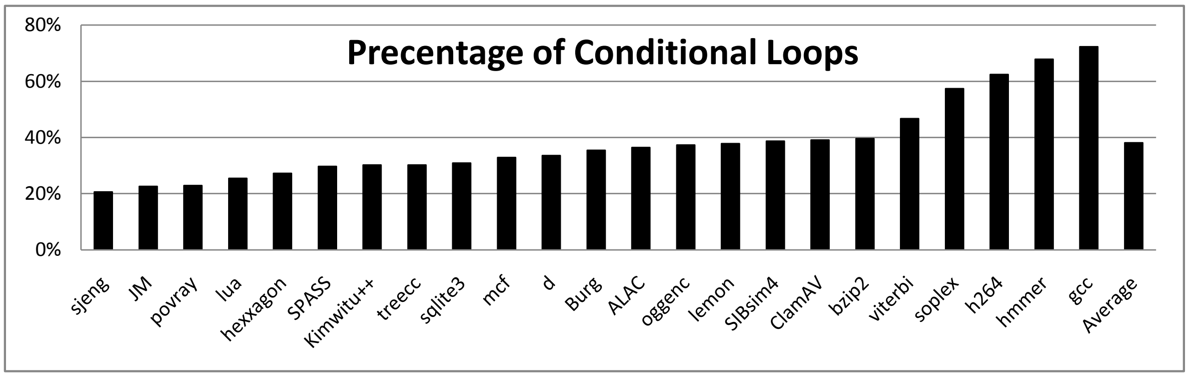


Figure 7.14. It is important to support loops with conditional constructs. Many important loops have at least one conditional clause in their body.

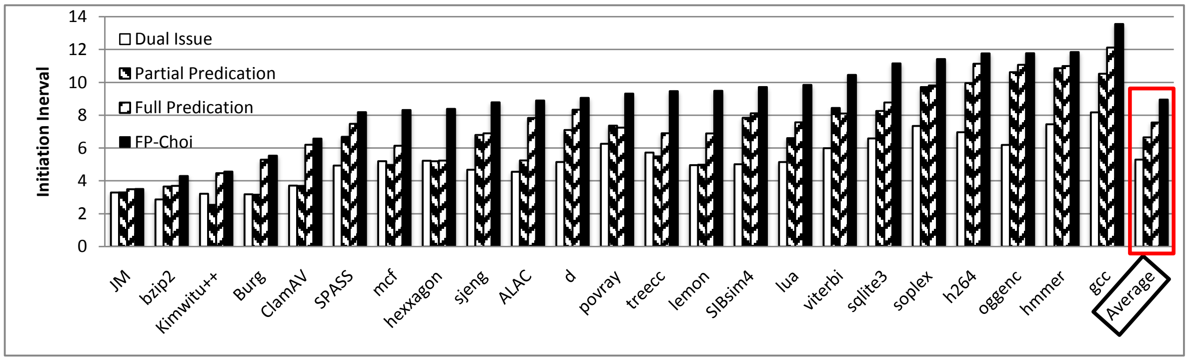


Figure 7.15. This figure plots the achieved $II$ for different loops with conditionals from various benchmarks. The graph shows that Dual-issue architecture with our proposed compiler technique results in the lowest $II$.

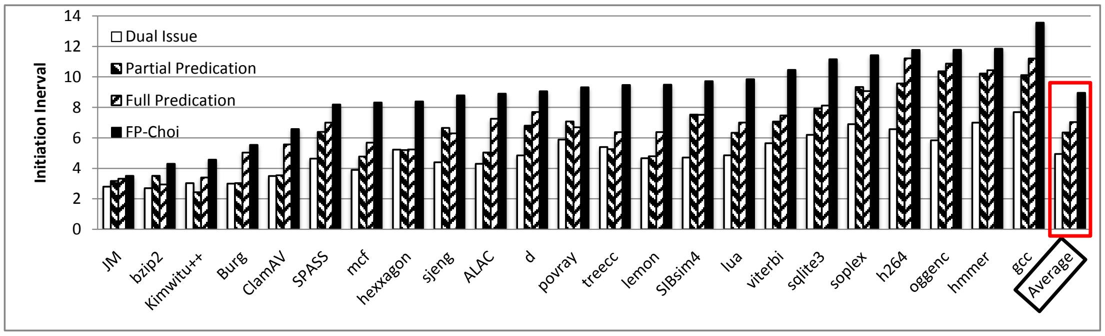


Figure 7.16. The Performance of compiled loops using different acceleration techniques in mesh and diagonal interconnected CGRA. Dual-issue scheme leads to the best acceleration among all. Partial predication and full predication show relatively close performance benefit. Better interconnection benefits all application but the benefit is narrow for loops limited by $RecMII$.

# Chapter 8

# SUMMARY AND FUTURE WORKS

Coarse-Grained Reconfigurable Architectures (CGRAs) are extremely attractive platform when both performance and power efficiency are paramount. However, the achievable performance and power efficiency of CGRAs critically hinges upon compiler capabilities. Several problems has to be addressed to construct an effective compiler for CGRAs.

In this dissertation, we formulate the problem of mapping application onto a CGRA and establish its complexity. We also characterize the necessary conditions for application specification to find a feasible mapping. To tackle the mapping problem, we proposed a heuristic algorithm called EPIMap. EPIMap is different from the existing methods in the sense that it systematically searches the solution space to find a valid mapping.

One of the main challenges in CGRA compilers is to efficiently utilize registers which is specially difficult due to their distributed nature. We formulated the problem of mapping loops on CGRAs while efficiently using registers, we present a unified and precise formulation of the problem of simultaneous placement and register allocation, and an efficient and effective heuristic solution, REGIMap is distilled from our problem formulation.

Another important problem in CGRA compilers is to accelerate loops that have if-then-else constructs. In this dissertation, we study different acceleration schemes for loops with if-then-else constructs and develop compiler techniques to efficiently accelerate such loops on CGRAs.

Through out this study, we present several mapping techniques. Initiation Interval, the performance metric in modulo scheduling, is used as the main performance metric. We have developed a compiler and simulation framework for CGRA. This framework enables us to run application on CGRA on a computing system. It will be very valuable to design an effective interface between CGRA and the rest of a computing system specially the processor. This enables us to accurately mesure the performance of the system equipped with a CGRA and measure the execution time as a whole.

Several optimization schemes are available which transforms loops to maximize loop performance based on memory access pattern observed during a loop execution such as polyhydral model [10, 77]. I will be interesting to integrate the proposed mapping schemes in this dissertation with those memory optimizations.

Just-in-time compilation of loops at run-time and offload them on CGRAs seems another interesting research directions.

# REFERENCES

[1] "Snapdragon s4 processors: System on chip solutions for a new mobile age", (Copyright © 2011 Qualcomm, Inc, 2011).

[2] Ahn, M., J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures", in "Proceedings of the Conference on Design, Automation and Test in Europe", pp. 363–368 (2006).

[3] Aiken, A. and A. Nicolau, "Perfect pipelining: A new loop parallelization technique", in "Proceedings of the 2Nd European Symposium on Programming", pp. 221–235 (1988).

[4] Akturan, C. and M. Jacome, "Caliber: a software pipelining algorithm for clustered embedded vliw processors", in "Computer Aided Design, IEEE/ACM International Conference on", pp. 112–118 (2001).

[5] Aleta, A., J. M. Codina, A. Gonzalez and D. Kaeli, "Heterogeneous clustered vliw microarchitectures", in "Proceedings of the International Symposium on Code Generation and Optimization", pp. 354–366 (2007).

[6] Ansaloni, G., L. Pozzi, K. Tanimura and N. Dutt, "Slack-aware scheduling on coarse grained reconfigurable arrays", in "Design, Automation Test in Europe Conference Exhibition", pp. 1–4 (2011).

[7] Bandishti, V., I. Pananilath and U. Bondhugula, "Tiling stencil computations to maximize parallelism", in "Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis", SC '12, pp. 40:1–40:11 (2012).

[8] Bansal, N., S. Gupta, N. Dutt, A. Nicolau and R. Gupta, "Network topology exploration of mesh-based coarse-grain reconfigurable architectures", in "Proceedings of the Conference on Design, Automation and Test in Europe", pp. 10474– (2004).

[9] Becker, J. and M. Vorbach, "Architecture, memory and interface technology integration of an industrial/ academic configurable system-on-chip (csoc)", in "Proc. ISVLSI", pp. 107–112 (2003).

[10] Benabderrahmane, M.-W., L.-N. Pouchet, A. Cohen and C. Bastoul, "The polyhedral model is more widely applicable than you think", in "Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction", pp. 283–303 (2010).

[11] Betz, V., J. Rose and A. Marquardt, eds., *Architecture and CAD for Deep-Submicron FPGAs* (Kluwer Academic Publishers, Norwell, MA, USA, 1999).

[12] Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, “The gem5 simulator”, SIGARCH Comput. Archit. News **39**, 2, 1–7 (2011).

[13] Bouwens, F., M. Berekovic, B. D. Sutter and G. Gaydadjiev, “Architecture enhancements for the adres coarse-grained reconfigurable array”, in “Proc. HiPEAC”, pp. 66–81 (2008).

[14] Cardoso, J. and P. Diniz, “Modeling loop unrolling: Approaches and open issues”, in “Computer Systems: Architectures, Modeling, and Simulation”, edited by A. Pimentel and S. Vassiliadis, vol. 3133 of *Lecture Notes in Computer Science*, pp. 224–233 (Springer Berlin Heidelberg, 2004).

[15] Carr, S. and Y. Guan, “Unroll-and-jam using uniformly generated sets”, in “Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture”, pp. 349–357 (1997).

[16] Chang, K. and K. Choi, “Mapping control intensive kernels onto coarse-grained reconfigurable array architecture”, in “Proc. ISOCC”, pp. I–362–I–365 (2008).

[17] Chen, D. and J. Rabaey, “A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths”, Solid-State Circuits, IEEE Journal of **27**, 12, 1895–1904 (1992).

[18] Chen, L. and T. Mitra, “Graph minor approach for application mapping on cgras”, ACM Trans. Reconfigurable Technol. Syst. **7**, 3, 21:1–21:25 (2014).

[19] Chris Lattner and Vikram Adve, “The LLVM Instruction Set and Compilation Strategy”, Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign (2002).

[20] De Sutter, B., P. Coene, T. Vander Aa and B. Mei, “Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays”, in “Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems”, pp. 151–160 (2008).

[21] De Sutter, B., P. Raghavan and A. Lambrechts, *Handbook of Signal Processing Systems*, chap. Coarse-Grained Reconfigurable Array Architectures, pp. 553–592 (Springer, 2013), 2 edn.

[22] Diestel, R., *Graph Theory (Graduate Texts in Mathematics)* (Springer, 2005).

[23] Dimitroulakos, G., M. D. Galanis and C. E. Goutis, “Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures”, in “Proceedings of the 20th International Conference on Parallel and Distributed Processing”, pp. 113–113 (2006).

[24] Dimitroulakos, G., S. Georgiopoulos, M. D. Galanis and C. E. Goutis, “Resource aware mapping on coarse grained reconfigurable arrays”, Microprocessors and Microsystems **33**, 2, 91 – 105 (2009).

[25] Du, J. and J. Y.-T. Leung, “Complexity of scheduling parallel task systems”, SIAM J. Discret. Math. **2**, 4, 473–487 (1989).

[26] Ebeling, C., D. Cronquist and P. Franklin, “Rapid — reconfigurable pipelined datapath”, in “Field-Programmable Logic Smart Applications, New Paradigms and Compilers”, edited by R. Hartenstein and M. Glesner, vol. 1142 of *Lecture Notes in Computer Science*, pp. 126–135 (Springer Berlin Heidelberg, 1996).

[27] Ebeling, C., L. McMurchie, S. Hauck and S. Burns, “Placement and routing tools for the triptych fpga”, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **3**, 4, 473–482 (1995).

[28] Esmaeilzadeh, H., E. Blem, R. St. Amant, K. Sankaralingam and D. Burger, “Dark silicon and the end of multicore scaling”, in “Proceedings of the 38th annual international symposium on Computer architecture”, pp. 365–376 (2011).

[29] Farooq, M. and L. John, “Loop-aware instruction scheduling with dynamic contention tracking for tiled dataflow architectures”, in “Compiler Construction”, edited by O. de Moor and M. Schwartzbach, vol. 5501 of *Lecture Notes in Computer Science*, pp. 190–203 (Springer Berlin Heidelberg, 2009).

[30] Fernandes, M., J. Llosa and N. Topham, “Distributed modulo scheduling”, in “High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On”, pp. 130–134 (1999).

[31] Friedman, S., A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling and S. Hauck, “Spr: An architecture-adaptive cgra mapping tool”, in “Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays”, pp. 191–200 (2009).

[32] Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman & Co., New York, NY, USA, 1979).

[33] Gnanaolivu, R., T. Norvell and R. Venkatesan, “Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization”, in “Soft Computing and Pattern Recognition (SoCPaR), 2010 International Conference of”, pp. 145–151 (2010).

[34] Goldstein, S., H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor and R. Laufer, "Piperench: a coprocessor for streaming multimedia acceleration", in "Computer Architecture, Proceedings of the 26th International Symposium on", pp. 28 –39 (1999).

[35] Golin, R., "Auto Vectorization", https://archive.fosdem.org/2014/schedule/event/llvmautovec/attachments/audio/321/export/events/attachments/llvmautovec/audio/321/AutoVectorizationLLVM.pdf/ (2014).

[36] Goodacre, J. and A. N. Sloss, "Parallelism and the arm instruction set architecture", Computer **38**, 7, 42–50 (2005).

[37] Govindaraju, V., C.-H. Ho and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing", in "High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on", pp. 503–514 (2011).

[38] Greggain, L., "Cost benefit tradeoffs for asic versus programmable logic device", in "ASIC Seminar and Exhibit, 1990. Proceedings., Third Annual IEEE", pp. 5.1–5.4 (1990).

[39] Gupta, S., N. Dutt, R. Gupta and A. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow", in "Proceedings of the Conference on Design, Automation and Test in Europe", pp. 10114– (2004).

[40] Hamzeh, M., A. Shrivastava and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras", in "Proceedings of the 49th Annual Design Automation Conference", pp. 1284–1291 (2012).

[41] Han, K., J. Ahn and K. Choi, "Power-efficient predication techniques for acceleration of control flow execution on cgra", ACM Trans. Archit. Code Optim. **10**, 2, 8:1–8:25 (2013).

[42] Han, K., K. Choi and J. Lee, "Compiling control-intensive loops for cgras with state-based full predication", in "PRoc. DATE", pp. 1579–1582 (2013).

[43] Han, K., J. K. Paek and K. Choi, "Acceleration of control flow on cgra using advanced predicated execution", in "Proc. FPT", pp. 429–432 (2010).

[44] Hannig, F., H. Dutta and J. Teich, "Regular mapping for coarse-grained reconfigurable architectures", in "Proc. ICASSP", pp. 57–60 (2004).

[45] Hartenstein, R., "A decade of reconfigurable computing: a visionary retrospective", in "Proceedings of Design, Automation and Test in Europe", pp. 642 –649 (2001).

[46] Hartenstein, R., M. Herz, T. Hoffmann and U. Nageldinger, “Using the kress-array for reconfigurable computing”, in “Proc. SPIE”, pp. 150–161 (1998).

[47] Hartenstein, R., M. Herz, T. Hoffmann and U. Nageldinger, “Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array architectures”, in “Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific”, pp. 163–168 (2000).

[48] Hatanaka, A. and N. Bagherzadeh, “A modulo scheduling algorithm for a coarse-grain reconfigurable array template”, in “Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International”, pp. 1–8 (2007).

[49] Hell, P. and J. Nesetril, *Graphs and Homomorphisms* (Oxford University Press, New York, NY, USA, 2004).

[50] Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003), 3 edn.

[51] Henning, J. L., “Spec cpu2006 benchmark descriptions”, SIGARCH Comput. Archit. News **34**, 4, 1–17 (2006).

[52] Iandola, F., D. Sheffield, M. Anderson, P. Phothilimthana and K. Keutzer, “Communication-minimizing 2d convolution in gpu registers”, in “Image Processing , 20th IEEE International Conference on”, pp. 2116–2120 (2013).

[53] Ishii, Y., M. Inaba and K. Hiraki, “Access map pattern matching for data cache prefetch”, in “Proceedings of the 23rd International Conference on Supercomputing”, pp. 499–500 (2009).

[54] Iskander, Y., C. Patterson and S. Craven, “High-level abstractions and modular debugging for fpga design validation”, ACM Trans. Reconfigurable Technol. Syst. **7**, 1, 2:1–2:22 (2014).

[55] Jacobsen, M., Y. Freund and R. Kastner, “Riffa: A reusable integration framework for fpga accelerators”, in “Field-Programmable Custom Computing Machines, Annual IEEE Symposium on”, pp. 216–219 (2012).

[56] Jacobsen, M. and R. Kastner, “Riffa 2.0: A reusable integration framework for fpga accelerators”, in “Field Programmable Logic and Applications, 23rd International Conference on”, pp. 1–8 (2013).

[57] Jain, S., “Circular scheduling: A new technique to perform software pipelining”, in “Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation”, pp. 219–228 (1991).

[58] Jump, J. and D. R. Fritsche, "Microprogrammed arrays", Computers, IEEE Transactions on **C-21**, 9, 974–984 (1972).

[59] Kennedy, J. and R. Eberhart, "Particle swarm optimization", in "Neural Networks, Proceedings., IEEE International Conference on", vol. 4, pp. 1942–1948 vol.4 (1995).

[60] Kim, W., D. Yoo, H. Park and M. Ahn, "Scc based modulo scheduling for coarse-grained reconfigurable processors", in "Field-Programmable Technology (FPT), 2012 International Conference on", pp. 321–328 (2012).

[61] Kirkpatrick, S., C. D. Gelatt and M. P. Vecchi, "Optimization by simulated annealing", Science **220**, 4598, 671–680 (1983).

[62] Kong, M., R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet and P. Sadayappan, "When polyhedral transformations meet simd code generation", in "Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation", pp. 127–138 (2013).

[63] Krishna, A., T. Heil, N. Lindberg, F. Toussi and S. VanderWiel, "Hardware acceleration in the ibm poweren processor: architecture and performance", in "Proceedings of the 21st international conference on Parallel architectures and compilation techniques", pp. 389–400 (2012).

[64] Kumar, A., A. Hansson, J. Huisken and H. Corporaal, "An fpga design flow for reconfigurable network-based multi-processor systems on chip", in "Design, Automation Test in Europe Conference Exhibition", pp. 1–6 (2007).

[65] Lam, M., "Software pipelining: an effective scheduling technique for vliw machines", in "Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation", pp. 318–328 (1988).

[66] Lee, G., S. Lee and K. Choi, "Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques", in "Proc. ISOCC", pp. 395–398 (2008).

[67] Lee, M.-H., H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho and V. C. Alves, "Design and implementation of the morphosys reconfigurable computingprocessor", J. VLSI Signal Process. Syst. **24**, 147–164 (2000).

[68] Levi, G., "A note on the derivation of maximal common subgraphs of two directed or undirected graphs", Calcolo **9**, 341–352 (1973).

[69] Li, S. and C. Ebeling, "Quickroute: a fast routing algorithm for pipelined architectures", in "Field-Programmable Technology, Proceedings of IEEE International Conference on", pp. 73–80 (2004).

[70] Li, W. and H. Kurata, "A grid layout algorithm for automatic drawing of biochemical networks", Bioinformatics **21**, 9, 2036–2042 (2005).

[71] Llosa, J., "Swing modulo scheduling: A lifetime-sensitive approach", in "Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques", pp. 80– (1996).

[72] Llosa, J., "Swing modulo scheduling: A lifetime-sensitive approach", in "Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques", PACT '96, pp. 80– (IEEE Computer Society, Washington, DC, USA, 1996).

[73] Mahlke, S., *Exploiting instruction level parallelism in the presence of conditional branches*, Ph.D. thesis, UIUC (1997).

[74] Mahlke, S., R. Hank, J. McCormick, D. August and W.-M. Hwu, "A comparison of full and partial predicated execution support for ilp processors", in "Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on", pp. 138–149 (1995).

[75] Maleki, S., Y. Gao, M. J. Garzarán, T. Wong and D. A. Padua, "An evaluation of vectorizing compilers", in "Proceedings of the International Conference on Parallel Architectures and Compilation Techniques", pp. 372–382 (2011).

[76] McMurchie, L. and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for fpgas", in "Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays", pp. 111–117 (1995).

[77] Mehta, S., P.-H. Lin and P.-C. Yew, "Revisiting loop fusion in the polyhedral framework", in "Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming", pp. 233–246 (2014).

[78] Mei, B., S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, "Dresc: a retargetable compiler for coarse-grained reconfigurable architectures", in "Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on", pp. 166–173 (2002).

[79] Mei, B., S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling", in "Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1", pp. 10296– (2003).

[80] Mei, B., S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix.", in "Proceedings of the Conference on Field Programmable Logic", vol. 2778, pp. 61–70 (Springer, 2003).

[81] Mercaldi, M., S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin and S. J. Eggers, "Instruction scheduling for a tiled dataflow architecture", in "Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems", pp. 141–150 (2006).

[82] Micheli, G. D., *Synthesis and Optimization of Digital Circuits* (McGraw-Hill Higher Education, 1994), 1st edn.

[83] Mirsky, E. and A. DeHon, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources", in "Proc. FPGAs for Custom Computing Machines", pp. 157 –166 (1996).

[84] Miyamori, T. and K. Olukotun, "Remarc: Reconfigurable multimedia array coprocessor", IEICE Trans. on Information and Systems pp. 389–397 (1998).

[85] Muchnick, S. S., *Advanced Compiler Design and Implementation* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997).

[86] Muck, T. R. and A. A. Frohlich, "Towards unified design of hardware and software components using c++", IEEE Transactions on Computers **99**, PrePrints, 1 (2013).

[87] Nageldinger, U., "Coarse Grained Reconfigurable Architectures", http://helios.informatik.uni-kl.de/papers/publications/Nageldinger2cgra.pdf, [Online] (????).

[88] Nuzman, D., I. Rosen and A. Zaks, "Auto-vectorization of interleaved data for simd", in "Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation", pp. 132–143 (2006).

[89] Oh, T., B. Egger, H. Park and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures", in "Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems", LCTES '09, pp. 21–30 (2009).

[90] Owens, J., M. Houston, D. Luebke, S. Green, J. Stone and J. Phillips, "Gpu computing", Proceedings of the IEEE **96**, 5, 879–899 (2008).

[91] Park, H., K. Fan, M. Kudlur and S. Mahlke, "Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures", in "Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems", pp. 136–146 (2006).

[92] Park, H., K. Fan, S. A. Mahlke, T. Oh, H. Kim and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures", in "Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques", pp. 166–176 (2008).

[93] Park, H., Y. Park and S. Mahlke, "Polymorphic pipeline array: A flexible multi-core accelerator with virtualized execution for mobile multimedia applications", in "Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture", pp. 370–380 (2009).

[94] Rau, B. R., "Iterative modulo scheduling: An algorithm for software pipelining loops", in "Proceedings of the 27th Annual International Symposium on Microarchitecture", pp. 63–74 (1994).

[95] Ravishankar, M., J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems", in "Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis", pp. 72:1–72:11 (2012).

[96] Sánchez, J. and A. González, "Modulo scheduling for a fully-distributed clustered vliw architecture", in "Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture", pp. 124–133 (ACM, New York, NY, USA, 2000).

[97] Sanders, J. and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming* (Addison-Wesley Professional, 2010), 1st edn.

[98] Sarkar, V., "Optimized unrolling of nested loops", Int. J. Parallel Program. **29**, 5, 545–581 (2001).

[99] Silberschatz, A., P. B. Galvin and G. Gagne, *Operating System Concepts* (Wiley Publishing, 2008), 8th edn.

[100] Smith, g., "Time is money", Design Test, IEEE **30**, 1, 55–57 (2013).

[101] Stock, K., M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam and P. Sadayappan, "A framework for enhancing data reuse via associative reordering", in "Conference on Programming Language Design and Implementation", (2014).

[102] Taylor, M. B., "Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse", in "Proceedings of the 49th Annual Design Automation Conference", pp. 1131–1136 (2012).

[103] Toi, T., N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi and L. Jing, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor", in "Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design", pp. 702–708 (2006).

[104] Vasilache, N., B. Meister, M. Baskaran and R. Lethin, "Joint scheduling and layout optimization to enable multi-level vectorization", in "IMPACT", (Paris, France, 2012).

[105] Vassiliadis, S. and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing* (Springer Publishing Company, Incorporated, 2007), 1st edn.

[106] Venkataramani, G., W. Najjar, F. Kurdahi, N. Bagherzadeh and W. Bohm, "A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture", in "Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems", pp. 116–125 (2001).

[107] Volkov, V. and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra", in "Proceedings of the ACM/IEEE Conference on Supercomputing", pp. 31:1–31:11 (2008).

[108] Waingold, E., M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, "Baring it all to software: Raw machines", Computer **30**, 9, 86–93 (1997).

[109] Wang, J. and C. Eisenbeis, "Decomposed software pipelining: A new approach to exploit instruction level parallelism for loop programs", in "Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism", pp. 3–14 (1993).

[110] Yeung, A. and J. Rabaey, "A 2.4 gops data-driven reconfigurable multiprocessor ic for dsp", in "Solid-State Circuits Conference, 1995. Digest of Technical Papers. 41st ISSCC, 1995 IEEE International", pp. 108–109 (1995).

[111] Zafar, N., "Managing risk in asic design cycle", in "ASIC Seminar and Exhibit, 1990. Proceedings., Third Annual IEEE", pp. 7.1–7.3 (1990).