COMPILING PARALLEL APPLICATIONS TO COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

MOHAMMED ASHRAFUL ALAM TUHIN

Your file Votre référence
ISBN: 978-0-494-33454-6
Our file Notre référence
ISBN: 978-0-494-33454-6

Compiling Parallel Applications to Coarse-Grained Reconfigurable Architectures

by

© Mohammed Ashraful Alam Tuhin

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

M.Sc.

Department of Computer Science

Memorial University of Newfoundland

July 2007

St. John's Newfoundland and Labrador

Abstract

Reconfigurable computing has been an active field of research for the past two decades. Coarse-Grained Reconfigurable Architectures (CGRAs) are gaining interest for embedded systems and multimedia applications, which demand a flexible but highly efficient platform. A CGRA comprises a network of simple programmable processing elements (PEs). CGRAs exploit the inherent parallelism and repetitive computations found in these applications and can adapt themselves to diverse computations by dynamically changing configurations. Although CGRAs have the potential to exploit both hardware like efficiency and software like extensibility, the absence of proper compilation approaches is an obstacle to their widespread use.

In this thesis a novel approach for compiling parallel applications to a target CGRA will be presented. The application will be written in HARPO/L, a parallel object oriented language suitable for hardware. HARPO/L is first compiled to a Data Flow Graph (DFG) representation. The remaining compilation steps are a combination of three tasks: scheduling, placement and routing. For compiling cyclic portions of the application, we have adapted a modulo scheduling algorithm: modulo scheduling with integrated register spilling, which incorporates register spilling with instruction scheduling. For scheduling, the nodes of the DFG are ordered using the hypernode reduction modulo scheduling (HRMS) method. The placement and routing is done using the neighborhood relations of the PEs.

Acknowledgments

First of all I would like to express my deep and sincere gratitude to the almighty ALLAH who showed me the way throughout this thesis.

It is my immense pleasure to thank all those people who helped in various ways in preparation of this thesis.

I would like to thank my supervisor, Dr. Theodore S. Norvell, whose guidance, support, ideas, stimulating suggestions and encouragement helped me in all the time of research and for writing this thesis.

I am grateful to the administrative staffs and instructional staffs of the Department of Computer Science for assisting me in many different ways.

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Thanks to NSERC for providing the funding for this project.

Finally, I wish to express my love and gratitude to all my family and friends.

I am grateful to my friends, Rajibul Huq and Momotaz Begum, for their encouragement and continuous support.

Especially, I would like to give my special thanks to my wife, Isheeta Nargis, for her endless love and encouragement. Without whom I would have struggled to find the inspiration and motivation needed to complete this work.

Lastly, and most importantly, I wish to thank my parents, Abdul khaleque Bhuiyan and Meherun Nahar. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation	**Meaning**
ASICs	application specific integrated circuits
GPPs	general-purpose processors
FPGAs	Field Programmable Gate Arrays
CGRAs	Coarse-Grained Reconfigurable Architectures
RAW	Reconfigurable ArchitectureWorkstation
DReAM	Dynamically Reconfigurable Architecture for Mobile Systems
ADRES	Architecture for Dynamically Reconfigurable Embedded Systems
DFG	Data Flow Graphs

Chapter 1

Introduction

1.1 Reconfigurable Computing

Reconfigurable computing has been an active field of research for the past two decades. The main concept of reconfigurable computing is to avoid the von Neumann bottleneck (the bandwidth limitation between processor and memory) through direct mapping of a portion of an application into hardware to exploit the implicit data parallelism in the application. These systems are also capable of dynamically changing the hardware logic, which they implement. Thus, an application can be partitioned for execution on the hardware which enables it to execute designs which are larger than the available physical resources.

Reconfigurable computing has the potential to become a vital component in the next generation of computation devices. Reconfigurable architectures bridge the gap between application specific integrated circuits (ASICs) and general-purpose processors (GPPs) as seen in Figure 1.1. They achieve better performance by combining the

cost and power efficiency of customized hardware like ASICs with the extensibility of microprocessors.

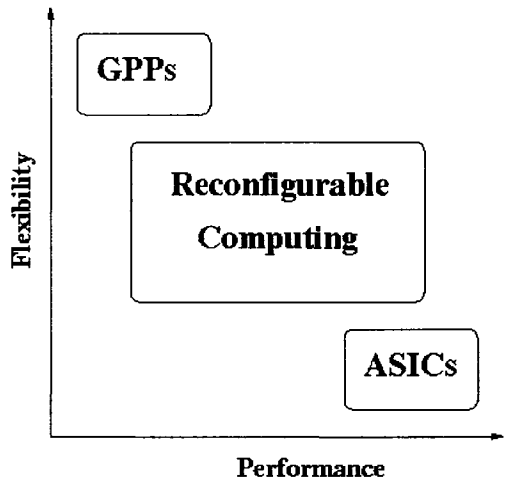

Figure 1.1: Bridging the gap.

1.2 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs), which are the most widely used reconfigurable architectures, are more capable of exploiting the parallelism than traditional processors. So naturally some applications on FPGA-based reconfigurable architectures perform much better than processor-based alternatives. However, for applications where the data path is coarse-grained (8 bit or more), the performance and power consumption on FPGAs are inefficient. Also the compilation time and the reconfiguration time on FPGAs are long. Standard arithmetic computation is less efficient on FPGAs. FPGAs have the following inherent disadvantages:

- **Logic granularity:** FPGAs are designed for logic replacement. As a result, applications with coarse-grained data path perform inefficiently.

- **Compilation time:** Compiling applications (typically written in a hardware description language like VHDL or Verilog) to FPGA consists of logic synthesis, technology mapping, placing and routing. These process requires a long time for some applications.

1.3 Coarse-Grained Reconfigurable Architectures (CGRAs)

To overcome the disadvantages of FPGAs, many coarse-grained or ALU-based reconfigurable architectures have been proposed as an alternative between FPGA-based systems and fixed logic CPUs. Due to the coarse granularity of CGRAs, they substantially reduce the overhead for configurability at the cost of reduced flexibility. Coarse-grained reconfigurable architectures (CGRAs) provide massive parallelism, high computational capability and they can be configured dynamically, making them attractive in the years to come especially in embedded system design. They also have such advantages as flexible topology, predictable timing, small instruction storage space, etc. Since CGRAs have more computation resources than other programmable devices such as RISC and VLIW processors, higher performance or better performance/energy efficiency is demanded from them and they are ready to deliver it. Due to the partial connectivity of CGRAs, they are scalable, yet cost- and power-efficient unlike coarse-grained VLIW. Examples of some CGRAs are the

KressArray [Hartenstein and Kress 1995], RaPiD [Ebeling *et al.* 1997], MorphoSys [Singh *et al.* 2000], CHESS [Marshall *et al.* 1999], PipeRench [Goldstein *et al.* 2000], REMARC [Miyamori and Olukotun 1998], **R**econfigurable **A**rchitecture **W**orkstation (RAW) [Waingold *et al.* 1997], Matrix [Mirsky and DeHon 1996], **D**ynamically **Re**configurable **A**rchitecture for **M**obile Systems (DReAM) [Alsolaim *et al.* 2000], **A**rchitecture for **D**ynamically **R**econfigurable **E**mbedded **S**ystems (ADRES) [Mei *et al.* 2003a], Chameleon Systems [Cha], MathStar [Mat], etc.

1.4 Compilation Techniques for CGRAs

For programming coarse-grained reconfigurable architectures, diverse tools and programming approaches have been used. However, due to the familiarity of developers with traditional programming languages, compiler-based approaches have been popular over the years. Compiling applications written in a high-level language to hybrid systems containing coarse-grained reconfigurable platforms has been an active field of research in the recent past. The work in this domain is mostly highly dependent on the target architecture.

Although CGRAs have the potential to exploit both hardware like efficiency and software like flexibility, the absence of proper compilation approaches is an obstacle to their widespread use. There has not been much work on compiling applications directly on to systems containing only coarse-grained reconfigurable architectures. Although there is extensive research and even commercial tools for FPGAs, the techniques developed for them are not directly applicable to CGRAs, because of the substantial differences in PEs and interconnection architectures among others.

The compiler plays a critical role in the success of a coarse-grained reconfigurable architecture (CGRA). The compiler must carefully schedule code to make the best use of the multiple resources available in a CGRA. Compiling applications to CGRA, after the source code of the target application has been transformed and optimized to a suitable intermediate representation, is a combination of three tasks: scheduling, placement, and routing. Scheduling assigns time cycles to the operations for execution. Placement places these scheduled operation executions on specific processing elements. Routing finds routes to data from producer PE to consumer PE using the interconnect structure of the target architecture. Our goal is to compile parallel applications to a given target architecture with near optimal execution time.

1.5 Motivation of this thesis

Computationally intensive applications demand a great deal of performance and flexibility. These applications such as image recognition and processing, streaming video, and highly interactive services are asking more of the processing components. These applications have some code segments that need most of the time required for the whole application. Moreover, there is a high degree of parallelism in those code segments, enabling concurrent execution of operations. On the other hand, decreases in power consumption, cost, and the time-to-market of the processing components are demanded by these applications. Fulfilling all these demands has posed a new challenge in the area of embedded system and multimedia applications. CGRAs can accept the challenge and promise to meet all the demands by combining the benefits of ASICs, DSPs, and Microprocessors. Moreover, CGRAs are equipped with abun-

dant computational resources to exploit the concurrency in those applications. But the absence of proper automatic compilation techniques is a significant obstacle in accepting this challenge.

This motivates us in designing an automatic compilation approach that can achieve high performance.

1.6 Contribution of this Thesis

The main contribution of this thesis is to propose a compilation approach for a family of CGRAs. The target architecture will be specified by the user. The intended application will be written in HARPO/L [Norvell 2006], a parallel, object-oriented, multithreaded programming language. The input of the compilation is the intermediate representation of the target application in the form of a Data Flow Graphs (DFG) and a description of the target architecture; the output will be executable code. HARPO/L is first compiled to a Data Flow Graph (DFG) representation [Zhang 2007]. The remaining compilation steps are a combination of three tasks: scheduling, placement and routing. For compiling cyclic portions of the application, we have adapted a modulo scheduling algorithm: modulo scheduling with integrated register spilling (MIRS) [Zalamea *et al.* 2001a], which incorporates register spilling with instruction scheduling. We have also simplified the MIRS method for acyclic portions of the given application. For scheduling, the nodes of the DFG are ordered using the hypernode reduction modulo scheduling (HRMS) [Llosa *et al.* 1995] method. The placement and routing is done using the neighborhood relations of the processing elements (PEs).

1.7 Organization of this Thesis

The rest of the thesis is organized as follows:

Chapter 2 gives background on coarse-grained reconfigurable architectures. First an overview of development of CGRA will be outlined by briefly describing several academic and commercial CGRAs. Then the compilation techniques adopted in some of those CGRAs will be discussed. After that some traditional scheduling methods, which are useful during compilation, are discussed.

Chapter 3 gives an overview of the related work of compilation of applications to coarse-grained reconfigurable architectures that are in line with our work.

Chapter 4 discusses how the input application will be presented to the compilation process. The input application is written using an explicitly parallel language HARPO/L and then after some transformation and optimization the intermediate representation is obtained in the form of executable data flow graph.

Chapter 5 first describes our target architecture. Then an overview of our compilation process is presented.

Chapter 6 describes the modulo scheduling algorithm for cyclic portions of the target application that is the central part of the compiler.

Chapter 7 describes the placement and routing steps performed during mapping from the input DFG to the input target architecture.

Chapter 8 gives a summary of the thesis, presents some final conclusions with possible future work.

Chapter 2

Background

This chapter gives background on coarse-grained reconfigurable architectures. First an overview of development of CGRA will be outlined by briefly describing several academic and commercial CGRAs. Then the compilation techniques adopted in some of those CGRAs will be discussed. After that some traditional scheduling methods, which are useful during compilation, are discussed.

2.1 Introduction

Over the last one and a half decade, the rapid development of circuit densities and speed of VLSI systems has brought about a radical growth in both computer architectures and microprocessors. Some computation-intensive applications, previously feasible only on supercomputers, are presently feasible on workstations and PCs. Similarly, the use of reconfigurable architectures has also widened during the past decade or two. The principle attributes of these reconfigurable architectures are the capability of dynamic mapping of a portion of a program to the hardware to exploit

the implicit data parallelism in the program.

2.2 Coarse-Grained Reconfigurable Architectures Overview

Coarse-Grained Reconfigurable Architectures (CGRAs) are the most promising among reconfigurable architectures. CGRAs are gaining interest for embedded systems and multimedia applications, which demands flexible but highly efficient architecture platforms. An important characteristics of CGRAs is that they do not support as many different instructions as a general purpose processor does. So the instructions in the target applications for the target architecture should be as regular as possible. That is, these architectures demand to have the same instruction being repeated for many times instead of having many different instructions being used once. CGRAs can enhance the performance of these applications by exploiting the inherent parallelism and repetitive computations found in these applications and adapting themselves to diverse computations by dynamically changing configurations of its internal processing elements (PEs) and their interconnections. They can achieve close to the performance of DSPs or ASICs, mostly used for embedded systems and multimedia applications, due to their abundant parallelism, high computational density, and flexibility of runtime reconfiguration. They can outperform DSPs on many applications owing to greater parallelism. CGRAs can not compete with ASICs on performance, but rather on time-to-market and flexibility.

2.3 Selected Overview of Some CGRAs

In the past one and a half decades, different types of CGRAs with different granularity, fabrics, and compilation techniques, and intended for different applications have been developed. They have identical processing elements (PEs), even though wide variation exists in the number and functionality of components and the interconnections between them. These architectures often consists of tens to hundreds of PEs intended to execute word-level operations as opposed to the bit-level ones common in FPGAs. The coarse granularity of CGRAs drastically reduces the power, area, delay, and configuration time compared with FPGAs, at the expense of flexibility. As a result, we have seen the emergence of a wide range of CGRAs over recent years. Hartenstein [Hartenstein 2001] has surveyed the development in the field of Reconfigurable Computing. He first briefly outlined the major aspects of various types of reconfigurable architectures and then demonstrated possible methods for programming them.

2.3.1 PipeRench

PipeRench [Goldstein *et al.* 2000] is a coprocessor, which acts as an accelerator for pipelined applications. PipeRench provides a virtual hardware of several reconfigurable pipeline stages and relies highly on fast partial dynamic pipeline reconfiguration as well as runtime scheduling of both configuration streams and data streams. The architecture comprises a 256 by 1024 bit configuration memory, a state memory, an address translation table (ATT), four data controllers, a memory bus controller and a configuration controller. The state memory stores the current register contents

of a stripe. The ATT stores the address in the state memory for the state of a given stripe. All the controllers are used for data I/O. The data controller generates address sequences for both input and output data streams. The configuration controller interfaces the fabric to the host, maps configuration to the hardware, does runtime scheduling and manages the configuration memory.

The reconfigurable fabric allows the configuration of a pipeline stage in every cycle, while executing all other stages simultaneously. The fabric comprises several horizontal stripes, each stripe consisting of interconnect and processing elements with registers and ALUs.

The interconnect network of PipeRench has local interconnect inside a stripe as well as local and global buses. The global interconnect input and output data to and from the pipeline. The local interconnect facilitates each PE to have outputs from the previous stripe and from any other PE in the same stripe.

2.3.2 MorphoSys

The MorphoSys [Singh *et al.* 2000] architecture consists of a core processor, a frame buffer, a DMA controller, context memory, and a 8 by 8 reconfigurable cell (RC) array. The execution of an application is split into the two part: the core processor and the reconfigurable part. The core processor executes the sequential part while the RC array, capable of multithreading, takes care of any embedded parallelism. The core processor is a TinyRISC processor with special instructions for controlling the DMA controller and the RC array. The DMA controller uses DMA instructions for transferring data between the main memory and the frame buffer and for loading

configuration from the main memory to the context memory. The context memory stores the configuration for the RC array.

The frame buffer works like a data cache by storing blocks of intermediate results. The frame buffer is logically divided into two sets, by using the two sets independently, load and store can be overlapped. Each set has two banks of 128 16-bit words. A horizontal bus of 128 bit connects the frame buffer to the RC array, allowing all cells of a row to share a 16-bit segment of the frame buffer.

Each cell of the RC array features an ALU-multiplier, a shifter, two input multiplexers, a register file with 16 4-bit registers and a 32 bit context register. The ALU multiplier has four inputs, two from the input multiplexers, one from the output register and one from the context register. The ALU-multiplier is capable of performing the standard arithmetic and logical operations, as well as a multiply-accumulate operation in a single cycle. The input multiplexers select inputs from any neighbor cell, from the data bus, from the frame buffer, or from the register file.

The RC array has a 3-layer interconnection network. In the first layer all the cells are connected to the nearest neighbors. In the second layer, the cells are connected to cells in the same row or column in the same quadrant. In the third layer, the cells are connected to cells in the same row or column of the neighbor quadrant.

2.3.3 ADRES

Architecture for **D**ynamically **R**econfigurable **E**mbedded **S**ystems (ADRES) [Mei *et al.* 2003a] is a power-efficient flexible architecture template that combines a very long instruction word (VLIW) processor with a coarse-grained array. This

architecture has such advantages as high performance, low communication overhead and ease of programming. The array, containing many functional units, accelerates data-flow loops by exploiting high degrees of loop-level parallelism. The VLIW DSP efficiently executes the part of the code which can't achieve so large parallelism. The VLIW and the array are coupled and communicate by a shared VLIW register file. The architectural flexibility of ADRES, combined with the design flow from the application written in C, allows a designer to rapidly explore architectural options for an application domain. The architecture template allows designers to specify the interconnection, the type and the number of functional units.

2.3.4 RAW

The RAW (**R**econfigurable **A**rchitecture **W**orkstation) micro-architecture [Waingold *et al.* 1997] consists of an array of inter-connected tiles, each of which resembles a RISC processor. Each tile contains an ALU, register file of 32 general purpose and 16 floating point registers, data memory, and instruction memory, configurable logic (CL) and a programmable switch that can support both static and dynamic routing.

The RAW architecture provides both a static and a dynamic network. The processors lack hardware for register renaming, dynamic instruction issuing or caching (which is found in current super scalar processors). As a result, the compiler generates statically scheduled instruction streams and it is the responsibility of the software to handle all the dynamic issues. If the compiler fails to find a static schedule, there is a backup dynamic support in the form of possible flow control.

2.3.5 RaPiD

The RaPiD architecture [Green and Franklin 1996] is a linear array of data path units (DPUs) which is configured to form a mostly linear pipeline. The linear array is divided into identical cells which are replicated to construct the whole array. Each cell consists of an integer multiplier, three integer ALUs, six general purpose data path registers and three local 32 word memories. Each memory has a specialized data path registers having an incrementing feedback path. The routing and configuration structure of RaPiD consists of several parallel segmented 16 bit buses, which span the length of the data path. The bus segments in different tracks have different lengths. The interconnections are used most efficiently using this feature. In some tracks, adjacent bus segments can be joined together by configurable bus connectors.

For propagation of I/O streams, RaPiD provides a stream generator which contains address generators. Address generators are optimized for nested loop structure, associated with FIFOs. The address sequences for the generators are determined at compile time. The cells of the linear array are interconnected and controlled using a combination of static and dynamic control.

2.3.6 KressArray

KressArray (also known as rDPA) can be considered as the first actual CGRA as it has a 32 bit wide data path [Hartenstein and Kress 1995]. It is a generalization of the systolic array. It is basically a regular array of reconfigurable processing elements, known as reconfigurable Data Path Units (rDPUs).

Figure 2.1 shows the KressArray architecture. Each rDPU has two inputs and two outputs. Each rDPU consists of an ALU with a data path of 32 bits wide. One important characteristics of rDPU is that they are data-driven. So, an rDPU computes an operation whenever all its inputs are available. The rALU controller attached to the rDPU mesh controls the communication of data streams to and from the array and the configuration.

The KressArray has a global bus for longer connection. Although the topology of its local interconnect is static, global interconnection may be dynamic. The local interconnects follow a unidirectional approach that provides efficiency in area management. The global data bus is used for data I/O to the array and for propagation of intermediate values to other rDPUs or between the rDPU array and the register file in the controller.

2.4 Compilation Techniques for CGRA Overview

For programming coarse-grained reconfigurable architectures diverse tools and programming approaches have been used. However, due to the familiarity of developers with traditional programming languages, compiler-based approaches have been popular over the years. Compiling applications written in a high-level language to hybrid systems containing coarse-grained reconfigurable platforms has been an active field of research in the recent past. The work in this domain is mostly highly dependent on the target architecture.

Although CGRAs have the potential to exploit both hardware like efficiency and software like flexibility, the absence of proper compilation approaches is an obstacle

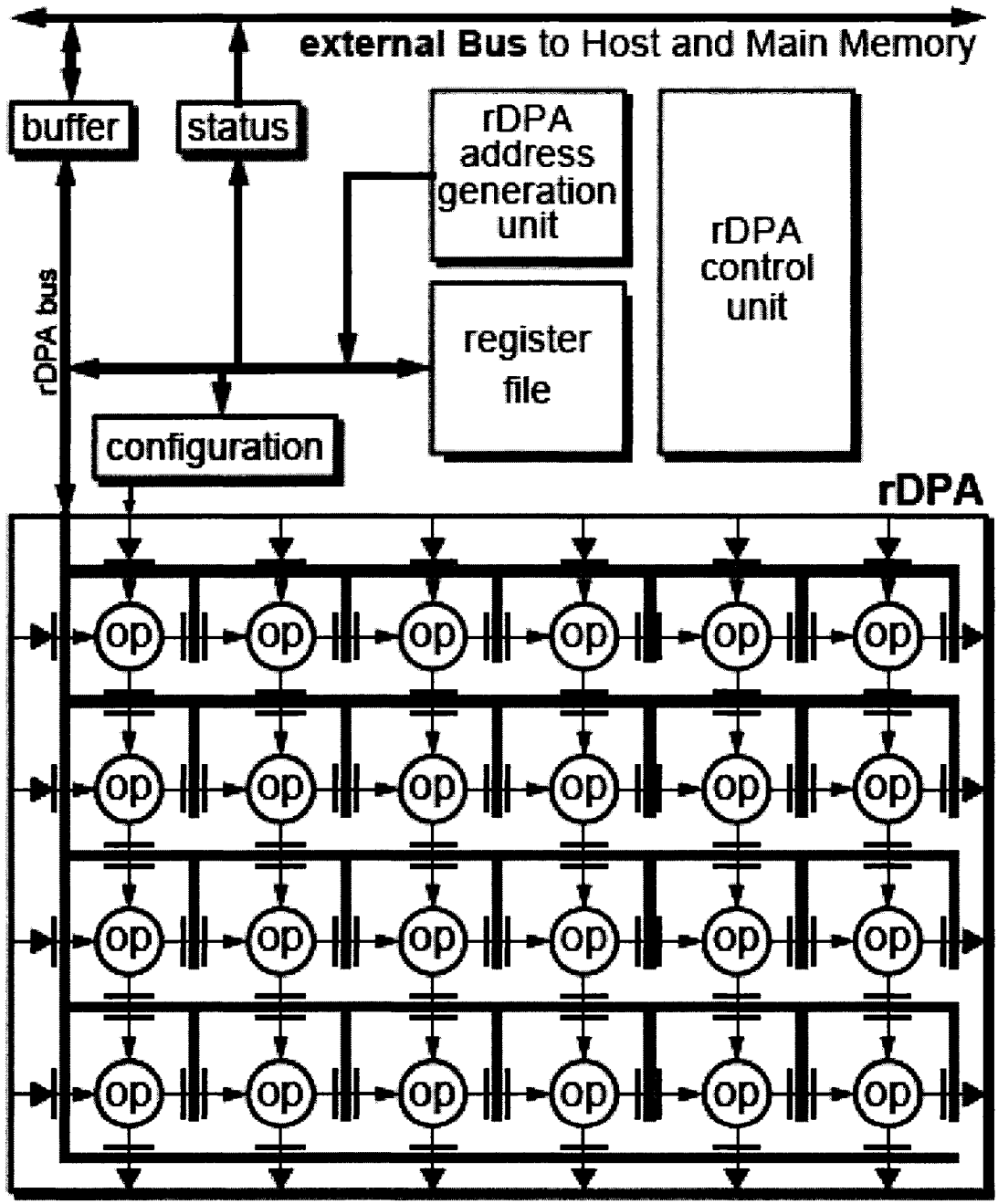

Figure 2.1: The Kress Array Architecture [Hartenstein and Kress 1995].

to their widespread use. There has not been much work on compiling applications directly on to systems containing only coarse-grained reconfigurable architectures. Although there is extensive research and even commercial tools for FPGAs, the techniques developed for them are not directly applicable to CGRAs, because of the substantial differences in PEs and interconnection architectures among others. This section investigates various mapping and scheduling approaches that are needed during compilation of applications for different CGRAs.

2.4.1 Compilation Techniques

The compiler plays a critical role in the success of a coarse-grained reconfigurable architecture (CGRA). The compiler must carefully schedule code to make the best use of the multiple resources available in a CGRA. Compiling applications to CGRAs, after the source code of the target application has been transformed and optimized to a suitable intermediate representation, is a combination of three tasks: scheduling, placement, and routing. Scheduling assigns time cycles to the operations for execution. Placement places these scheduled operation executions on specific processing elements. Routing plans the movement of data from producer PE to consumer PE using the interconnect structure of the target architecture.

We will give an overview on the scheduling methods found in the literature in the next section.

The placement of operations onto the processing elements is largely dependent on the target coarse-grained reconfigurable architecture. The placement problem is an NP-hard problem. As a result different heuristic approaches have been used over

time for approximating an optimal placement. Some of the approaches taken are:

- Partitioning
- Simulation
- Analytic

Out of the above, simulation based placement is the most widely used approach. Simulated annealing, force-directed placement, and genetic algorithms are three approaches for simulation. Instead of using any one of the above approaches, combining two or more to have the benefit of them often leads to better performance.

We will now give an overview of the compilation approach taken by the CGRAs outlined in the previous section.

2.4.1.1 PipeRench

The approach taken for compilation of applications on PipeRench is to analyze the application's virtual pipeline, which is mapped onto physical pipe stages to maximize execution throughput [Goldstein *et al.* 2000].

The compiler uses a greedy place-and-route algorithm to map these pipe stages onto the reconfigurable fabric. PipeRench uses a technique called pipeline reconfiguration to improve compilation time, reconfiguration time, and forward compatibility. A reconfigurable system partitions computations between the fabric and the system's other execution units. The fabric does reconfigurable computations whereas the processor does system computations. The system performs reconfigurable computations by configuring the fabric to implement a circuit customized for each particular re-

configurable computation. The compiler embeds computations in a single static configuration rather than an instruction sequence, reducing instruction bandwidth and control overhead. However, their technique is limited to very specific architectures, and thus cannot be applied to other coarse-grained reconfigurable architectures.

2.4.1.2 MorphoSys

MorphoSys [Singh *et al.* 2000] aims at applications which have inherent data-parallelism, high regularity, and high throughput requirements. Venkataramani et al. Presented a compiler framework for mapping loops to the MorphoSys architecture [Venkataramani *et al.* 2001]. Application are written using SA-C, which is an expression oriented single assignment language.

2.4.1.3 ADRES

For compiling applications on the ADRES architecture template [Mei *et al.* 2003a], the Inter-university Microelectronics Center (IMEC) of Belgium has designed a compiler framework DRESC (Dynamically Reconfigurable Embedded System Compiler). The DRESC [Mei *et al.* 2002] retargetable C compiler targets both the VLIW processor and the array. Application source code can therefore be compiled directly onto the coarse-grained reconfigurable processor. DRESC [Mei *et al.* 2002] framework uses a novel modulo scheduling algorithm, which is capable of pipelining a loop onto the partially interconnected array to achieve high parallelism. The task of modulo scheduling is to map the program graph to the architecture graph and try to achieve optimal performance while respecting all dependencies.

We will briefly describe DRESC in the next chapter as DRESC is quite similar

with what we are trying to design and implement.

2.4.1.4 RAW

The RAW compiler uses the SUIF compiler infrastructure [Waingold *et al.* 1997]. The compiler partitions the program into multiple, coarse-grained parallel threads, each of which is then mapped onto a set of tiles. The RAW compiler views the set of N tiles in a RAW machine as a collection of functional units for exploiting ILP. The compiler can generate unoptimized code for a small set of programs.

2.4.1.5 RaPiD

RaPiD has a linear data path that is a different approach compared with 2-dimensional meshes of processing elements (PEs). Its Functional Units (FUs) communicate in nearest-neighbor fashion. This constraint simplifies application mapping but restricts the design space dramatically.

The VLIW compiler front-end is used to transform programs written in a high-level language like C or Java to a control/data flow graph that is then scheduled to the configurable data path. The programmer has to explicitly specify the parallelism, the data movement, and partitioning using special constructs like signal-wait for synchronization and conditions for identifying the first or last iteration of a loop. For nested loops, the outer loops are translated into sequential code handled by the address generators. The innermost loops are transformed into structured code which are handled by the RaPiD architecture. The compilation procedure is composed of four steps: netlist generation for the structured code, dynamic control extraction, instruction stream generation for the programmed controller, and I/O configuration

data generation for the stream units. The scheduling problem is formulated as a place and route problem that maps data flow graphs from the program control/data flow graph to a computing substrate comprising multiple instances of the data path unrolled in time. The placement is done using a simulated annealing algorithm, considering the routing simultaneously.

2.4.1.6 KressArray

In KressArray, the data path synthesis system (DPSS) [Hartenstein and Kress 1995] maps statements of a high level language description onto the reconfigurable Data Path Architecture (rDPA). Configuring the rDPA is composed of logic optimization and technology mapping, placement and routing, and I/O scheduling. DPSS uses simulated annealing to simultaneously solve the placement and routing sub-problems. However, it does not support multiple configurations for one loop. The routing is restricted to direct neighbor connections and neighbors connections with a delay of one cycle. Global buses are used for routing all other connections.

2.4.2 Scheduling

Scheduling is a well defined and studied problem in the research area of high-level synthesis. It is actually an optimization problem. Scheduling rearranges instructions by filling the gap created by the delay due to dependence between instructions. If other instructions were not scheduled in this gap, the processor would stall and waste cycles. Scheduling is normally applied after machine independent optimizations, and either before or after register allocation.

The objective of scheduling is to create an optimal schedule, a schedule with the

shortest length. Schedule length is measured as the total execution time in cycles. Moreover, this optimal schedule must be obtained in a reasonable amount of time. Scheduling algorithms must satisfy both resource and dependence constraints when producing a schedule. Two instructions are dependent on each other if one uses an operand defined by another. If a schedule obeys the resource constraints, it will require resources that will be supported by the available resources of the target architecture.

Scheduling algorithms generally use a resource reservation table. A resource reservation table has columns equal to the number of resources, and rows equal to the cycles of the schedule. Placing an instructions in an entry of the table (suppose $< c, r >$) indicates that the instruction will use that resource (r) in that cycle (c).

Scheduling may be broadly classified into three main categories depending on the constraints:

- **Unconstrained Scheduling**: it do not consider timing or resource usage during scheduling.
- **Time-Constrained Scheduling**: it minimizes the number of required resources when the number of clock cycles is fixed.
- **Resource-Constrained Scheduling**: it minimizes the number of clock cycles when the number of resources is given.

Scheduling is often classified into three categories in terms of basic blocks it schedules: **Local Scheduling, Global Scheduling**, and **Cyclic Scheduling**. Local scheduling deals with single basic blocks, regions of straight line code with a single entry and exit. Global scheduling deals with multiple basic blocks having acyclic control flow. Cyclic scheduling deals with single or multiple basic blocks with cyclic control flow.

Since local scheduling handles only single basic blocks, which are generally not so large, it has some limitations. It can obtain optimal schedules locally, but the global schedule might not be optimal. Global scheduling, on the other hand, handles multiple basic blocks and can overlap execution of instructions from different basic blocks.

Most scheduling problems are NP-hard problems. Over the years, to solve scheduling problems, exact algorithms (capable of giving optimal solutions), or heuristic algorithms (capable of giving feasible and suboptimal solutions) have been applied. Some of the commonly used scheduling algorithms are briefly described below.

2.4.2.1 As-Soon-As-Possible (ASAP) or As-Late-As-Possible (ALAP) Scheduling

The ASAP algorithm schedules each node on the earliest possible clock cycle. The ALAP scheduling is similar, but it schedules each node to the latest possible clock cycle. ASAP and ALAP algorithms are used for solving the unconstrained scheduling problem [Walker and Chaudhuri 1995].

2.4.2.2 List Scheduling

The List scheduling is the most commonly used scheduling algorithm for resource-constrained scheduling problems [Pangrle and Gajski 1987]. It falls in the category of local scheduling. List scheduling schedules instructions starting at cycle 0, until all the instructions have been scheduled. A Conventional list based algorithm maintains a candidate list of candidate nodes, i.e., nodes whose predecessors have already been scheduled and which have no resource conflicts. The candidate list is sorted according

to a priority function of these nodes. In each iteration nodes with higher priority are scheduled first and lower priority nodes are deferred to a later clock cycle. Scheduling a node within a clock cycle may make its successor nodes candidates. ASAP, ALAP, mobility, height-based priority, etc. are used as the priority functions. In a dynamic list scheduling algorithm, the list changes every clock cycle. In a static list scheduling, a single large list is constructed statically only once before starting scheduling. The complexity is decreased by fixing the candidate list.

2.4.2.3 Trace Scheduling

Trace scheduling, a global scheduling algorithm, locates frequently executed traces (paths) in the program and treats the path as an extended basic block. This extended basic block is then scheduled using a list scheduling approach.

2.4.2.4 Superblock Scheduling

Superblock scheduling is a branch of global scheduling algorithms. Superblocks are basically a subset of traces having a single entry and multiple exits. Superblocks are scheduled using list scheduling approach.

2.4.2.5 Hyperblock Scheduling

Hyperblock scheduling, a global scheduling techniques, removes excessive control flow to simplify scheduling. It eliminates conditional branches using a technique known as If-Conversion [Allen *et al.* 1983].

2.4.2.6 Forced Directed List Scheduling

This is the common choice for solving the time-constrained scheduling problem. It minimizes the resource for a given time by balancing the concurrency of operations, the value to be stored, and data transfers [Paulin and Knight 1989a]
[Paulin and Knight 1989b].

2.4.3 Software Pipelining

Software pipelining is a family of global cyclic scheduling algorithms. It is an instruction scheduling technique that exploits the instruction level parallelism (ILP) available in loops by overlapping operations from various successive iterations and executing them in parallel. The idea is to look for a pattern of operations (often termed as the kernel) so that when repeatedly iterating over this pattern, it produces the effect that an iteration is initiated at fixed intervals, termed as initiation interval (II), before the preceding ones are finished. This way multiple iterations of a loop can be in execution simultaneously, each iteration in different stages of their computation.

After the schedule is obtained, the loop is reconstructed into a prologue, a kernel, and an epilogue. The prologue consists of code from first few iterations. Once a steady state is reached, a new iteration of the kernel is initiated every II cycles. An interesting feature of the steady state is that an iteration of the kernel consists of instructions from multiple iterations of the original loop. This feature is the central idea of software pipelining. The last few iterations, after the steady state, constitute the epilogue. Generally the majority portion of a loop is spent while executing in the kernel.

The objective of software pipelining is to minimize the II. As a result, we can perform more iterations using software pipelining in the same amount of time during which the original loop is scheduled than without software pipelining.

Figure 2.2 shows an example illustrating software pipelining of a loop with no intra-iteration dependency. The body of the loop loads a value from memory, increments the value by a constant, and stores the value back to memory again. Here each iteration takes 4 time cycles to execute, add requiring two cycles. If there is no inter-iteration dependency a new iteration can be initiated each time cycle, i.e., II = 1, then 7 iterations of the loop will take only 10 cycles instead of 28 cycles if there was no software pipelining. The execution of the software pipelined loop in time cycle 4 is termed the kernel, while from time cycle 0 to 3 is termed the prologue, and time cycle 8 to 10 as the epilogue. The execution from time cycle 4 to 7 is known as the steady state. During this state, after every II time cycles an iteration finishes its execution and the execution of a new iteration is initiated.

Figure 2.3 shows another example illustrating software pipelining of a loop with intra-iteration dependency. The body of the loop loads a value from previous iteration, increments the value by a constant, multiplies the added value by another constant, and stores the value back to memory again. The multiplied value is loaded in the first instruction of the loop body. In this case each iteration also takes 4 time cycles to execute, each instruction requiring one cycle. Due to the inter-iteration dependency between the first and the third instructions, a new iteration can be initiated every three time cycles, i.e., II = 3, then 7 iterations of the loop will take 22 cycles instead of 28 cycles if there was no software pipelining. The execution of the software pipelined loop from time cycle 4 to 6 is termed the kernel, while from time cycle 0

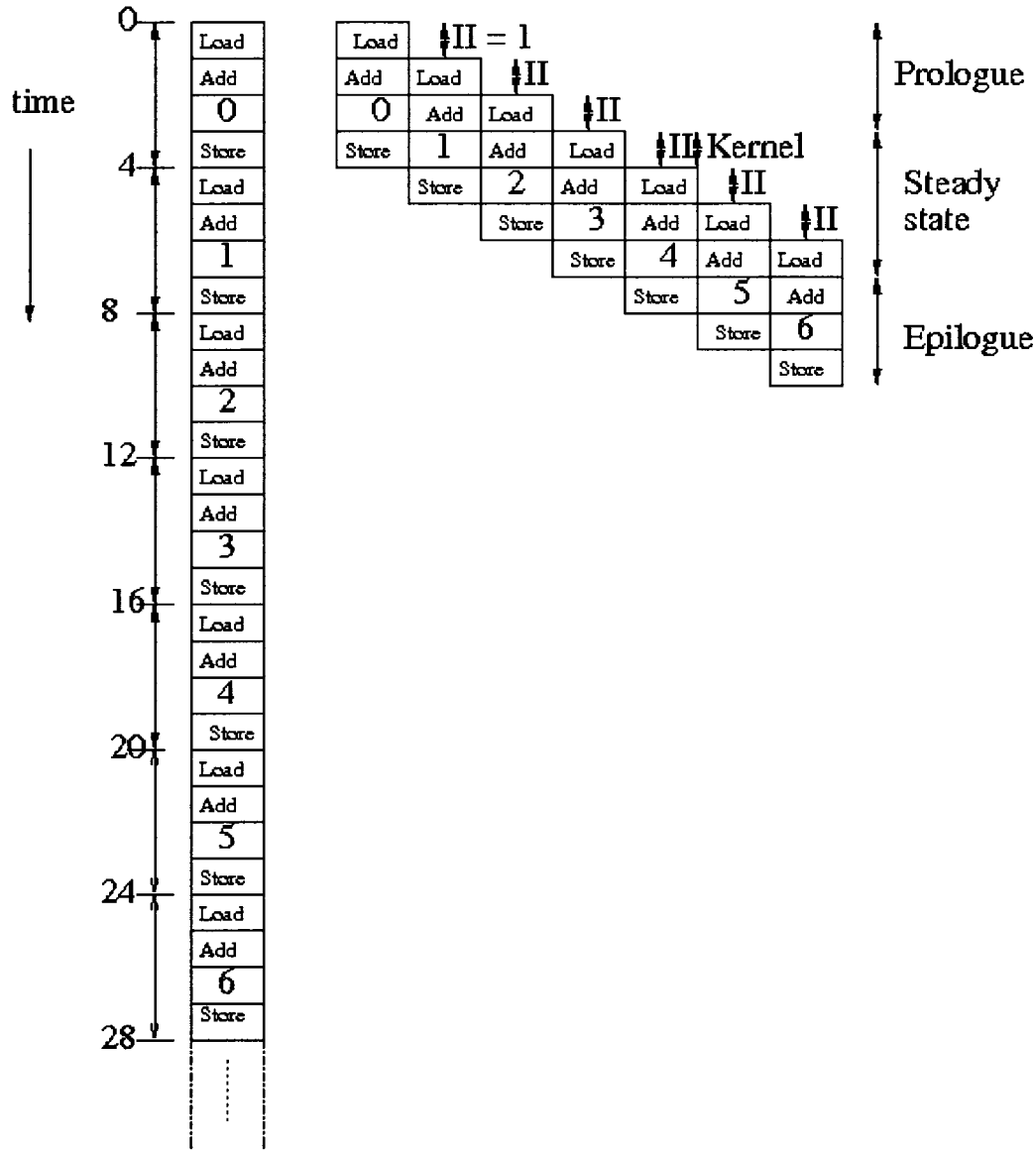

Figure 2.2: Software Pipelining Example for loops with no dependency.

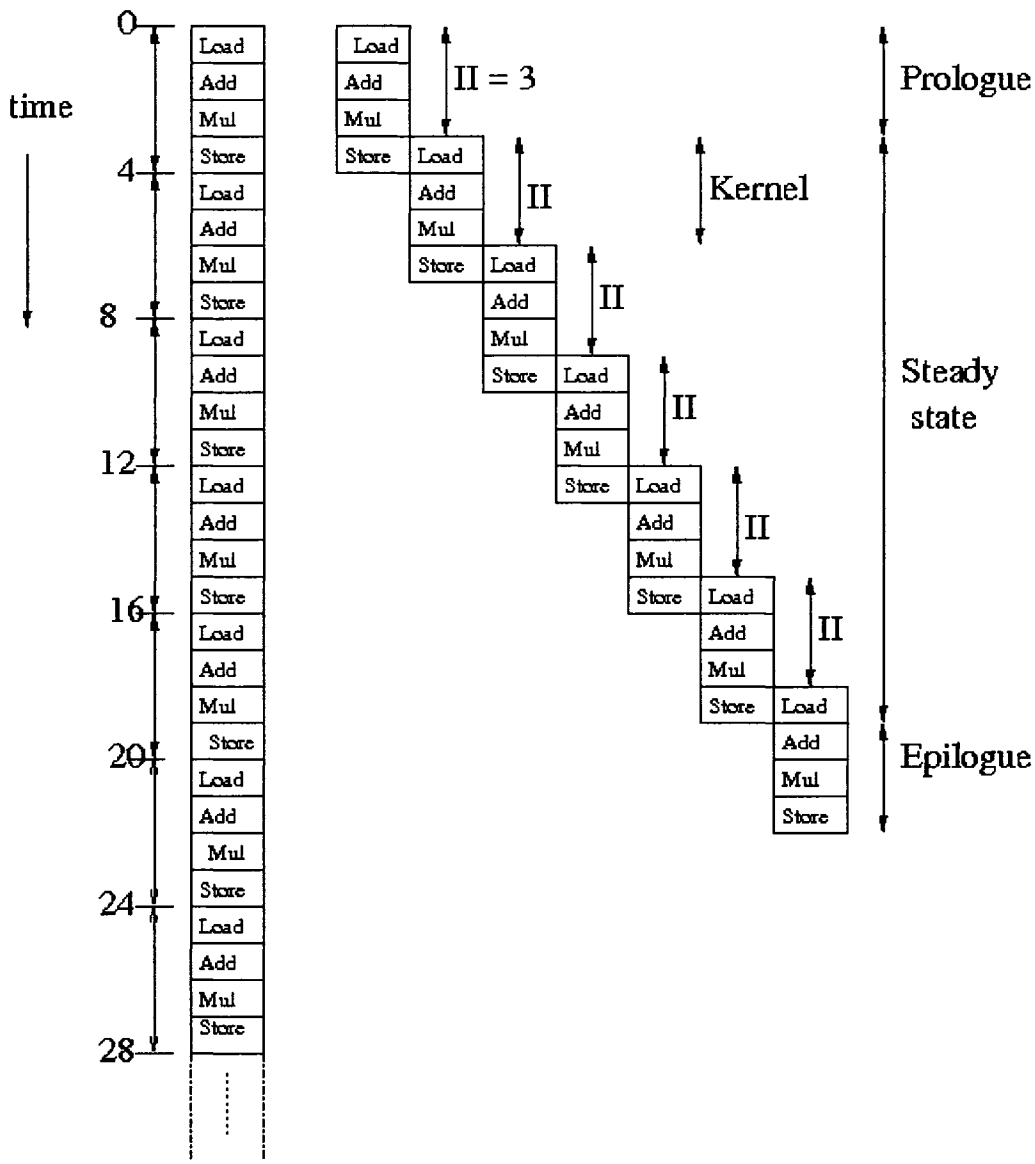

Figure 2.3: Software Pipelining Example for loops with dependency.

to 3 is termed the prologue, and time cycle 20 to 22 as the epilogue. The execution from time cycle 4 to 19 is known as the steady state.

There are two ways of software pipelining: **move-then-schedule** and **schedule-then-move**.

In the "move-then-schedule" (also known as code motion) approach, operations of a loop are moved, one by one, across the back edge of the loop, in either the forward or the backward direction [Moon and Ebcioğlu 1992][Jain 1991]. But the difficulty with this approach is the determination of operations, direction, and frequency of move around the back edge to obtain the best performance.

In the "schedule-then-move" approach, schedules are constructed to maximize performance and then moves are done as needed [Codina *et al.* 2002]. There are two ways for this approach. In the first, "unroll-while-schedule" [Aiken *et al.* 1995], the loops are unrolled and scheduled simultaneously, until the schedule becomes a repetition of an existing schedule. The partial schedule must have some parameters that should be the same if one would consider the total schedule. Among these parameters are: the number of iterations in execution, and for each iteration: the operations that have been scheduled, the time of the availability of their output, resources that will be consumed by them until completion, and the allocated register for each output. This approach often leads to high time complexity. The second is "modulo scheduling", which is outlined in the next subsection.

2.4.3.1 Modulo Scheduling

Modulo scheduling is the most popular approach for software pipelining [Rau and Glaeser 1981][Dani 1998][Llosa 1996][Llosa *et al.* 1995][Rau 1994]. It sim-

plifies the process of software pipelining by using the same schedule for all iterations. It is used in ILP processors such as VLIW to improve parallelism by executing different loop iterations in parallel. It uses a modulo technique (instead of maximal unrolling) to place operations in the schedule such that when iterations are overlapped there are no resources or data conflict.

The objective of modulo scheduling is to generate a schedule for an iteration of the loop such that this same schedule is repeated at constant intervals without violating intra- and inter-iteration dependency and without arising any resource uses conflict between operations of either the same or distinct iterations. The schedule for an iteration is divided into stages so that the execution of consecutive iterations, each one in a different stage, overlaps. The number of stages in one iteration is named stage count (SC), and the number of cycles per stage is termed the initiation interval (II). II, basically, reflects the performance of the scheduled loop. The lower the II, the higher the amount of parallelism existing in a loop is exploited. Modulo scheduling attempts to reduce the II associated with a loop.

Modulo scheduling take as input the loop to be scheduled, represented by its data dependence graph, and a description of the architecture, and produce a schedule for this loop. The II is constrained either by recurrences in the dependence graph, i.e., cycles created by loop-carried dependences (RecMII) or by resource constraints of the architecture (ResMII). ResMII is calculated from the resource usage requirements of the computation, while RecMII is obtained from the latency associated with the cycles in the dependence graph of the loop body. The lower bound on the II is known as the Minimum Initiation Interval (MII) and it is computed as MII = max(RecMII, ResMII). The scheduling starts with MII as the II. If an optimal schedule cannot be

obtained, II is increased and the algorithm tries to produce the schedule again. This process is repeated until a valid schedule is obtained or the algorithm gives up (when II reaches a value greater than the original loop's length in cycle).

Figure 2.4 shows an example illustrating software pipelining of a loop using modulo scheduling. Suppose each iteration takes 6 time cycles to execute. If the dependencies of the operations for the loop are such that a new iteration can be initiated after every two time cycles, i.e., II = 2, then 5 iterations of the loop will take only 14 cycles instead of 30 cycles if there was no software pipelining. The execution of the software pipelined loop from time cycle 4 to 6 is termed the kernel, while from time cycle 0 to 4 is termed the prologue, and time cycle 10 to 14 as the epilogue. The execution from time cycle 4 to 10 is known as the steady state.

Modulo schedules are constructed with two different approaches. The first is to find a global optimal solution using Integer linear programming. It is implemented using a mathematical formulation of the scheduling objectives and the constraints. But the problem with this approach is its time complexity, since scheduling, in general, is an NP-hard problem.

Another approach is based on heuristics, but they may not always give the optimal solution. But due to its time complexity with respect to the other approach, this has been adopted in many existing compilers.

Some heuristic based modulo scheduling algorithms are Iterative Modulo Scheduling [Rau 1994], Enhanced Modulo Scheduling [Warter *et al.* 1992], Integrated Register Sensitive Iterative Software Pipelining [Dani 1998], Hypernode Reduction Modulo Scheduling [Llosa *et al.* 1995], Modulo scheduling with integrated register spilling [Zalamea *et al.* 2001a], Slack Modulo Scheduling [Llosa *et al.* 2001], etc.

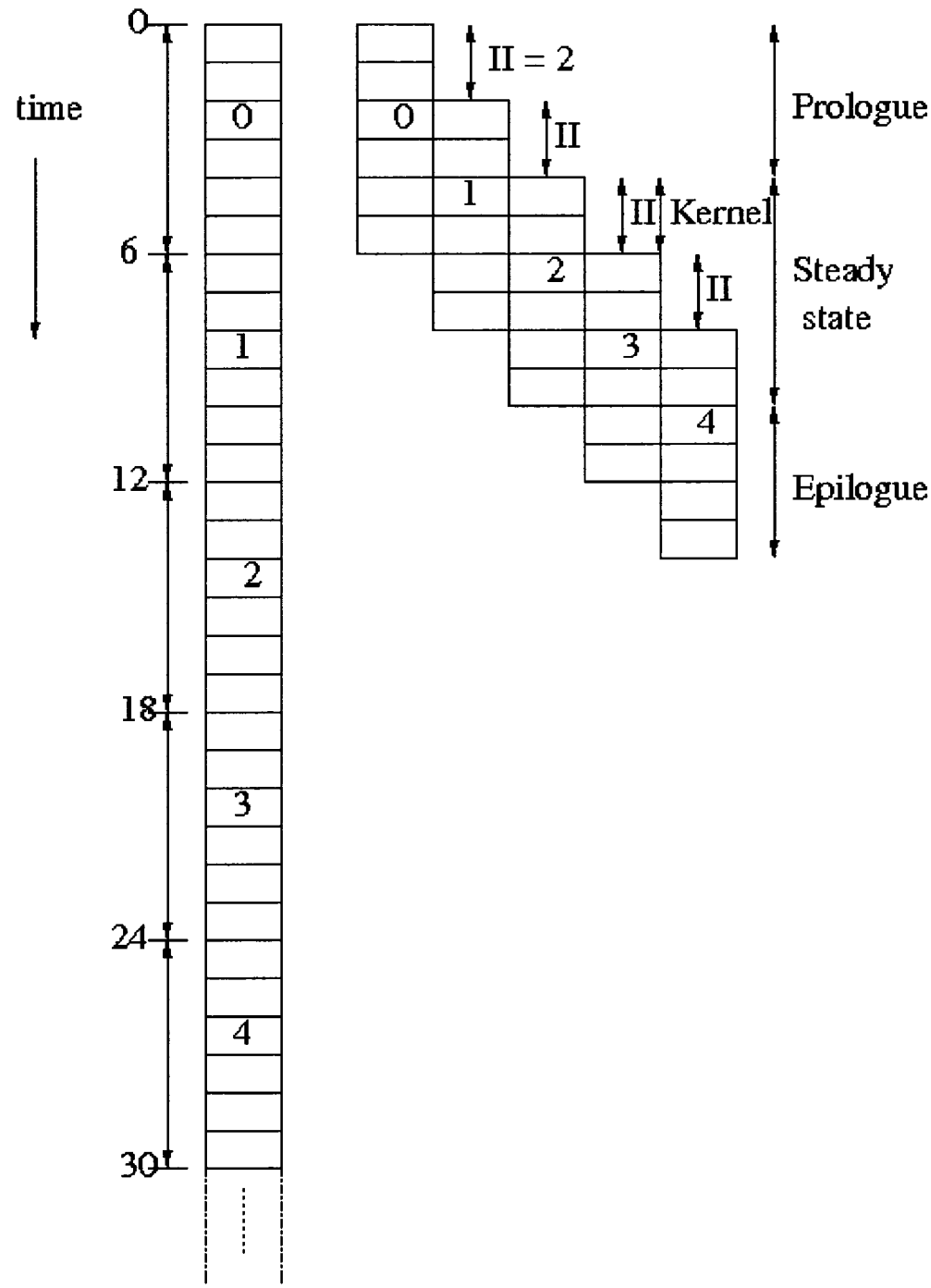

Figure 2.4: Modulo Scheduling Example.

Chapter 3

Related Work

This chapter gives an overview of the related work of compilation of applications to coarse-grained reconfigurable architectures.

3.1 DRESC Compiler

In [Mei *et al.* 2002] DRESC (Dynamically Reconfigurable Embedded Systems Compiler), a retargetable compiler for a family of coarse-grained reconfigurable architecture ADRES [Mei *et al.* 2003a] has been proposed. Modulo scheduling algorithm, which exploits loop level parallelism (LLP), is the main driver of DRESC. It can solve the placement, scheduling, and routing of operations simultaneously in a modulo constraint 3D space. DRESC is capable of parsing, analyzing, transforming, and scheduling any application written in C to a family of CGRAs. Their work is a combination of FPGA placement and routing, and modulo scheduling used for compilation for VLIW.

3.1.1 Target Architecture

Since DRESC is a retargetable compiler, the target architecture is a family of CGRAs [Mei *et al.* 2002]. There is flexibility as to the number of functional units (FUs), the number of register files (RFs), and the interconnection topology. Actually, in DRESC the architecture is an array of FUs and RFs. An FU can have inputs from neighboring nodes and outputs are saved to register. Each FU is supported with configuration RAM for storing multiple configurations locally. The configuration RAM provides control signals for the FUs, MUXes and RFs. The FU can handle more than one operation and the operations can be heterogeneous among different FUs. RF acts both as local storage and routing resource. Figure 3.1 shows an example of FU and RF of DRESC compiler.

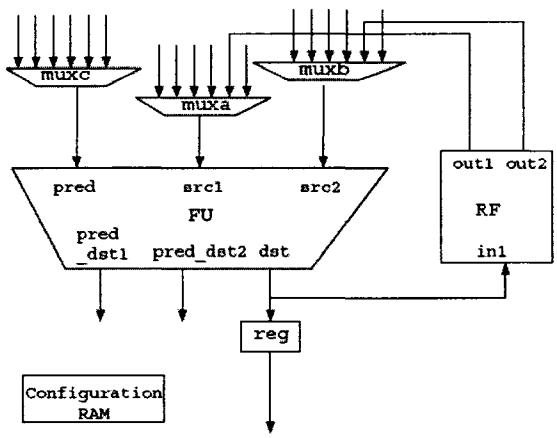

Figure 3.1: An Example of the organization of a FU and RF in DRESC target architecture [Mei *et al.* 2002].

3.1.2 Structure of DRESC Compiler

Figure 3.2 shows the overall compilation flow of the DRESC compiler [Mei *et al.* 2005]. It uses IMPACT compiler framework as the front end. The output is lcode, an intermediate representation. The architecture parser transforms the target architecture description into an architecture abstraction, which produces a modulo routing resource graph (MRRG). The modulo scheduling algorithm uses this MRRG for the compilation. The analysis and transformation phase generates a data flow graph for loop pipelining during scheduling taking lcode as the input. In the program anal-

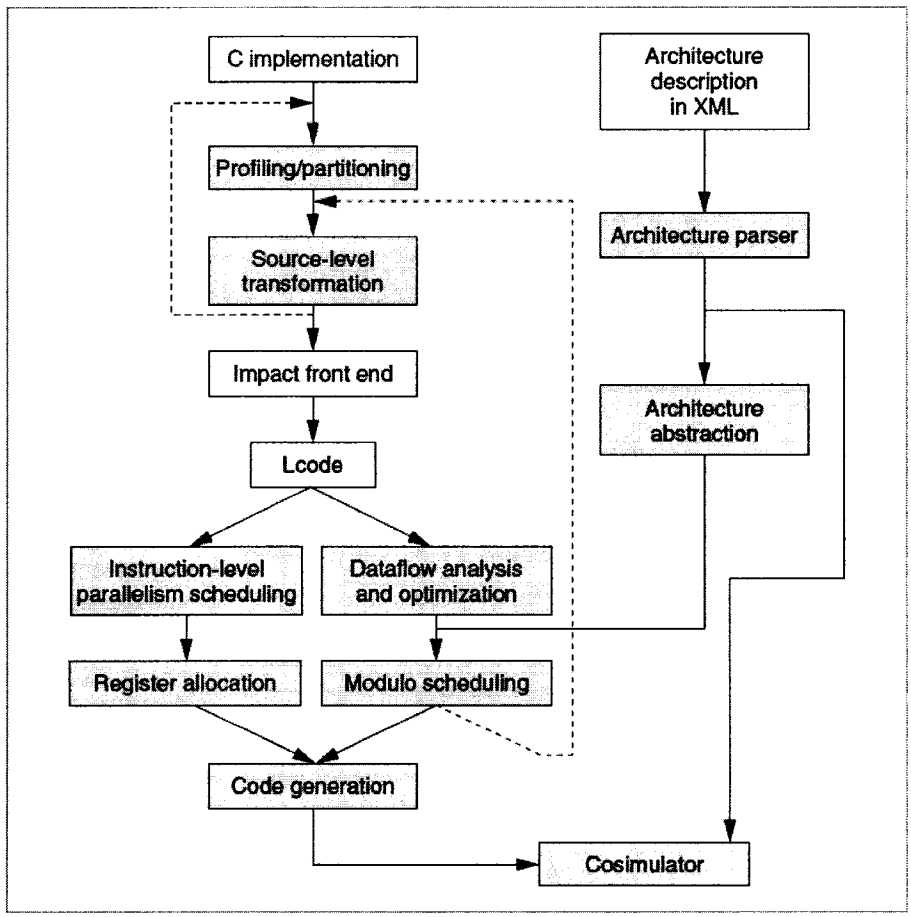

Figure 3.2: Compilation Flow of the DRESC Compiler [Mei *et al.* 2005].

ysis and transformation phase, several steps are performed such as identification of pipelinable loops, construction of the data dependence graph, transformation to a normalized static single assignment (SSA) form (SSA form will be discussed in chapter 4), etc. In normalized SSA form, unlike SSA form, each ϕ-function has two inputs and predicate analysis produces the signal for selection. Live-in and live-out variables are detected in the live-in and live-out analysis step. This step reduces the communication overhead on the register file. Prologue and epilogue, generated by pipelining, are removed. MII, which is the larger of resource-constrained MII, ResMII, and the recurrence-constrained MII, RecMII, is computed like iterative modulo scheduling [Rau 1994]. Moreover, ASAP (As-Soon-As-Possible), ALAP (As-Late-As-Possible), and mobility are computed for scheduling, and operation ordering operations are ordered following some priority. For example, operations on the critical path are assigned higher priority. To exploit spatial locality, operations are placed close to both its producer and consumer during routing.

3.1.3 Modulo Routing Resource Graph

DRESC uses modulo routing, therefore, to enforce modulo constraints, a modulo reservation table for software pipelining is required [Mei *et al.* 2002]. When targeting the CGRA, in the DRESC compiler, a graph named the modulo routing resource graph (MRRG) is introduced. MRRG transforms the target architecture to be used for the modulo scheduling algorithm. According to [Mei *et al.* 2002], an MRRG is defined as a directed graph $G = (V, E, II)$, which is a three-dimensional architecture graph representation generated by replicating the two-dimensional spatial architec-

ture across time. V is the set of nodes representing ports (input, output) or wires (bus) or two artificially created nodes (source, sink). A time t is associated with each node $v \in V$. E is the set of edges $\{(V_i, V_j)|t_{V_i} \leq t_{V_j}\}$ representing the switches connecting the nodes in V. In other words, MRRG is a graphical representation of the scheduling space, where nodes represent routing resources, and edges describe the connectivity among these resources.

There are two asymmetric aspects of MRRG indicated by the following two properties [Mei *et al.* 2002]. First, if an operation is scheduled using node R at time T, then all the nodes that have the same $[(T \bmod II), R]$ are also used, where II is the modulo time representing initiation interval. Second, if $(t_{Vi} > t_{Vj})$, then there must not be a route from V_i to V_j. Figure 3.3 shows the MRRG for some part of the DRESC architecture.

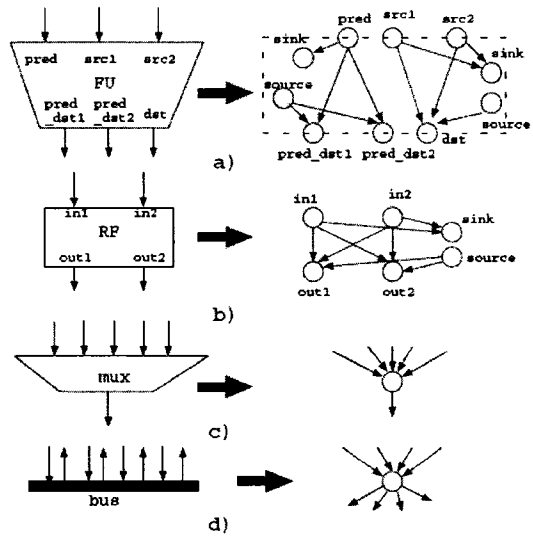

Figure 3.3: MRRG Representation of DRESC architecture part [Mei *et al.* 2002].

3.1.4 Modulo Scheduling Algorithm

3.1.4.1 Problem Formulation

For CGRAs, scheduling is a placement and routing problem in a modulo-constraint 3D space [Mei *et al.* 2002]. The reason is we have to determine the time, the place, and the connection of the operations. The scheduling problem is formulated as a mapping one from the data flow graph $G_1 = (V, E)$ to the MRRG graph $G_2 = (V, E, II)$. There are some constraints such as all MRRG nodes can not be used more than once, the exception is for source and sink nodes, all the edges E of G_1 have to be routed on G_2 without violating any resource constraint.

3.1.5 Algorithm Description

Figure 3.4 shows the modulo scheduling algorithm for CGRA [Mei *et al.* 2002]. Initially all the operations are ordered using the technique of [Llosa *et al.* 2001]. The algorithm starts with a minimal II and the outermost loop increments this II by 1 until a valid schedule is found. The MII is calculated as in [Rau 1994]. For each II, the algorithm creates an initial schedule respecting all dependency constraints. But this schedule may place more than one operation onto a single FU at the same cycle. The inner loop reduces this resource overuse iteratively and searches for a valid schedule. In each iteration an operation is removed from the existing schedule and placed randomly. The necessary routing is done simultaneously. The new placement and routing is evaluated by a cost function. A simulated annealing algorithm chooses whether the new placement and routing is taken or not. If the cost of the new placement and routing is smaller than the previous one, the operation will be placed in the

```
SortOps();
II   := MII(DDG);

while not scheduled do
  InitMrrg(II);
  InitTemperature();
  InitPlaceAndRoute();

  while not scheduled do
    for each op in sorted operation list
      RipUpOp();

      for i := 1 to random_pos_to_try do
        pos := GenRandomPos();
        success := PlaceAndRouteOp(pos);

        if success then
          new_cost := ComputeCost(op);
          accepted := EvaluateNewPos();
          if accepted then
            break;
          else
            continue;
          endif
        endfor

        if not accepted then
          RestoreOp();
        else
          CommitOp();

        if get a valid schedule then
          return scheduled;
    endfor

    if run out of time budget then
      break;

    UpdateOverusePenalty();
    UpdateTemperature();

  endwhile
  II++;
endwhile
```

Figure 3.4: Modulo Scheduling Algorithm for CGRA [Mei *et al.* 2003b].

new location. The new location is still chosen if the new cost is larger depending on temperature, which is decreased gradually. Thus local minimas are avoided. The cost function is comprised of overused resources. The weights of the overused resources are increased each iteration.

This process is repeated until a valid schedule is found. Otherwise the II is increased and the whole process is restarted with the new II. The objective of the algorithm is to schedule more than one operation using as few resources as possible. When each resource has at most one operation bound to it, a valid schedule is found.

3.1.6 Limitations

The modulo scheduling algorithm of the DRESC compiler has some limitations [Park *et al.* 2006]. It is time consuming due to the use of simulated annealing approach and has long convergence time for loops with large loop bodies. The algorithm do not use any information from the structure of the DFG in taking scheduling decisions. It does not scale well with respect to the size of the DFGs. Moreover, if there is sparse interconnection among the FUs, the algorithm converges with low probability. The algorithm only considers the innermost loop of a nested loop construct. Due to these limitations, the overall performance of an application is greatly affected.

3.2 Compilation Using Modulo Graph Embedding

Park *et al.* uses a graph theoretic technique, Modulo Graph Embedding, for compiling applications to CGRAs [Park *et al.* 2006]. One of the advantages of using this technique is that it utilizes the information about the structure of the DFG during

scheduling. Moreover, the technique is scalable with the size of the DFG. Modulo Graph Embedding is also adaptable to various CGRA configurations like sparse connectivity and register files configurations. So their technique takes care of most of the limitations found in the DRESC compiler [Mei *et al.* 2002].

3.2.1 Target Architecture

Modulo Graph Embedding uses a target architecture which is 16 homogeneous FUs arranged in a 4x4 CGRA in a mesh interconnection network as shown in Figure 3.5. In this design we can see that each FU has a dedicated register files. There is no register file sharing and there is no central register file.

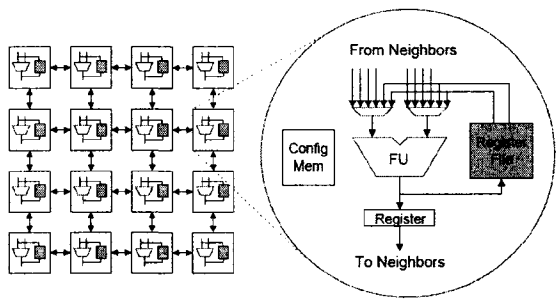

Figure 3.5: A target CGRA configuration with dedicated register files [Park *et al.* 2006].

3.2.2 Modulo Graph Embedding

Like [Rau 1994] Modulo Graph Embedding(MGE) uses a Modulo Reservation Table (MRT), having II time slots. Like [Mei *et al.* 2002], MGE also have a Modulo Routing Resource Graph (MRRG). But the approach for MRRG construction was simplified.

There are nodes for each FU and RF, but there is no node for ports for FUs and RFs. Moreover, an MRT is associated with each node of the modified MRRG. For efficient register usage, modulo graph embedding approach keeps an MRT for each register file. Register allocation and assignment is done during modulo scheduling. Registers keep the same value for up to II cycles. If the life is more than II, the register value is saved to another register in the same or different register file. Unlike [Mei *et al.* 2002], the original architecture description is replicated across time for II cycles with wraparound edges resembling a toroidal topology.

For minimizing routing cost [Park *et al.* 2006] uses two techniques. One is height-based where operations with larger height are scheduled before smaller height operations. Operations with the same height are treated together for better performance. The other is affinity-based, in which an affinity graph is constructed using the affinity information. The nodes of this graph are the operations and the edges are the affinity value between the corresponding operation pair.

Graph Embedding is done for each height level of the DFG, scheduling one level at one time. The overall layout is obtained iteratively. Partial layout makes the remaining layout construction easier and quicker as the available resources are reduced at every step.

One of the important characteristics of modulo scheduling is backtracking. But applying backtracking affects the performance of compiling to CGRAs significantly. The reason is the need for routing from producer to consumer(s) in CGRA. If backtracking is applied during scheduling, operations have to be routed to both its consumers and from its producers. For solving this problem MGE does clustering of the CGRA dynamically. The clusters span from left to right of the FU array. Leftmost

available FUs are given highest priority, whereas rightmost available FUs are given lowest priority. A skewed scheduling space is used to implement dynamic clustering. Each FU has an original start time equal to the time cycle in which it is placed. The start time of unused FUs at a particular time cycle is increased, the same is done for all the FUs located right and later time cycles. We can see examples of some skewed scheduling space in Figure 3.6. In modulo graph embedding operation at the same

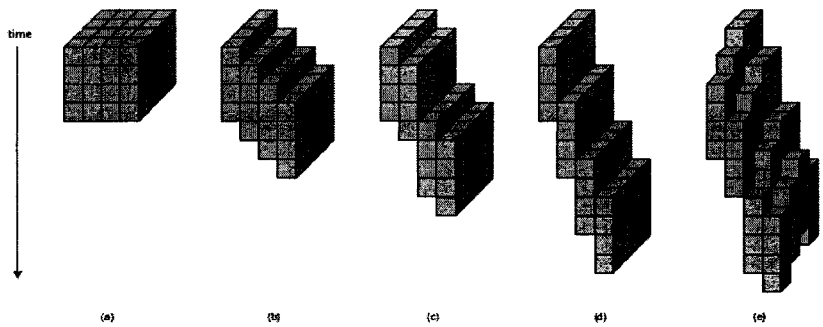

Figure 3.6: Variations of CGRA skewing spaces (a) Normal scheduling Space, (b) Variations of skewed scheduling space [Park *et al.* 2006].

height are considered together to obtain an optimal layout. Therefore, the parallelism in an application in a particular height determines the shape of the skewed scheduling space.

3.2.3 The Algorithm Description

Figure 3.7 outlines the framework of modulo graph embedding with an example. Initially some preprocessing is performed to analyze the DFG and to construct the skewed scheduling space. The DFG analysis computes the heights of all operations,

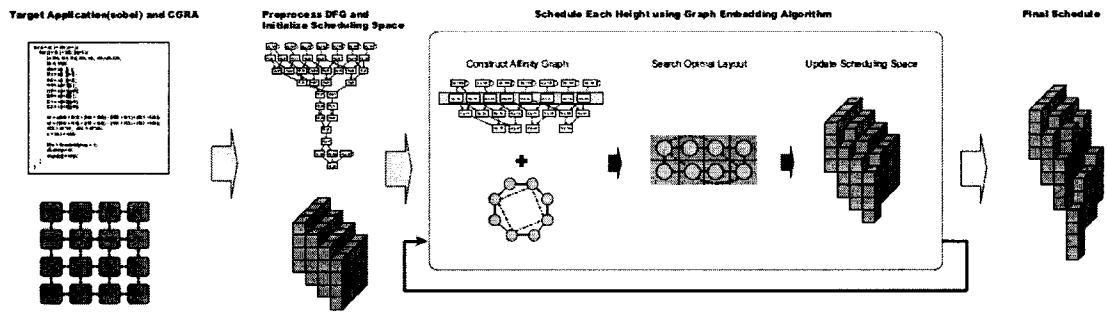

Figure 3.7: Overview of the Modulo Graph Embedding Approach. [Park *et al.* 2006].

i.e. the distance from the terminating operation. Scheduling uses this height and the live range of the intermediate values are estimated from the height difference of the producer and the consumer. The basic scheduling is done by considering the operations of all the levels, placing all the operation at a particular height using modulo graph embedding. Scheduling is formulated as a graph embedding problem. The graph embedding problem is treated as an optimization problem. Optimization is done by minimizing a discrete cost function of a layout. This cost function is computed by summing the cost of all node pairs. Simulated Annealing is used for this purpose. The affinity graph is mapped to the skewed scheduling space.

According to [Park *et al.* 2006], scheduling is done for achieving the following objectives.

- Placing operations with a common consumer close to each other.
- Minimization of the routing cost for values from producers.
- Ensure the routability of values to consumers.

The cost function is computed from three factors: routing cost, affinity cost, and

position cost. The variable $num\_cons(A, B, d)$ indicates the number of common consumers of A and B whose distance from A and B is the DFG is d.

The $affinity(A, B)$ between two operations A and B is computed as follows:

$$affinity(A, B) = \sum_{d=1}^{max\_dist} 2^{max\_dist-d} \times num\_cons(A, B, d)$$

The overall grid layout cost is computed as follows:

$$routing\_cost(A) = \#FUs \text{ used for routing values from producers to } A$$

$$affinity\_cost(A, B) = distance(FU(A), FU(B)) \times affinity(A, B)$$

$$position\_cost(A) = \text{column\#} \text{ of } FU(A) \times BASE\_COST$$

$$layout\_cost = \sum_{A \in ops} (routing\_cost(A) + position\_cost(A)) + \sum_{A,B \in ops} affinity\_cost(A, B)$$

The scheduling process for operations at each successive dependence height is carried out as shown as Figure 3.8. From the experiment done by [Park *et al.* 2006],

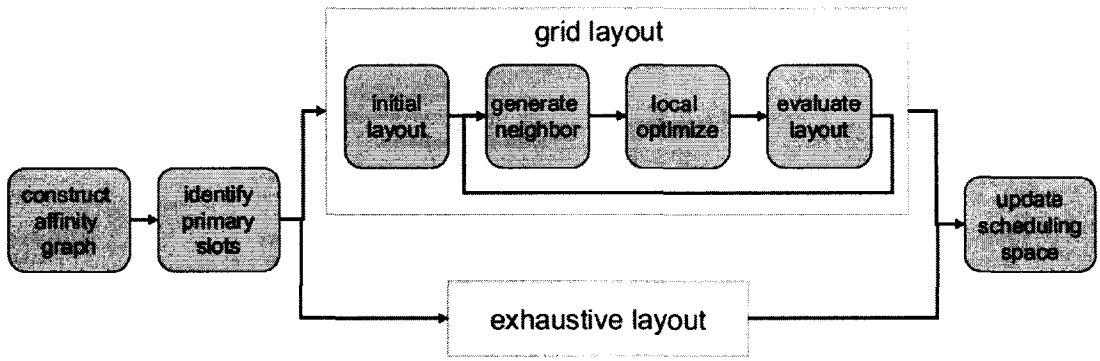

Figure 3.8: Modulo Graph Embedding for operations at each successive dependence height [Park *et al.* 2006].

it can be seen that register file plays an important role in the quality of the schedule. Register files with a central register file achieves highest quality. Moreover, shared register file design can give more utilization than dedicated register file design. The reason behind the discrepancy of utilization among various designs is that FUs having shared register file can use them as routing resource, so more routing opportunity is created.

3.2.4 Advantages

The modulo graph embedding method achieves 56% overall utilization and a maximum utilization of 69% [Park *et al.* 2006]. The average utilization is similar to DRESC compiler [Mei *et al.* 2002]. So it can be concluded that modulo graph embedding approach is cost-effective, as it uses dedicated register files and sparser network connectivity compared to the central register file and denser connectivity of the target architecture of DRESC. Modulo graph embedding deploys systematic placement decisions based on producer-consumer and uses a skewed scheduling space to achieve better convergence and faster compilation times, as the search space is limited only to the operations residing at the same height.

3.3 Compilation Using Graph Covering Algorithm

Guo *et al.* [Guo *et al.* 2005b] presented a mapping and scheduling technique for a coarse-grained reconfigurable processor tile, MONTIUM. The compilation method consists of four steps: transformation, clustering, scheduling, and allocation. These steps are applied to the control data flow graph (CDFG), which is translated from

the source code of a given application. The compilation method exploits maximum parallelism and locality of reference to achieve high performance and low power consumption. The main objective of their work is to compile DSP applications, written in a high level programming languages like C, to a MONTIUM tile with maximum throughput. Code efficiency and power consumption are the major concerns of the compiler.

3.3.1 Target Architecture

The target architecture [Guo *et al.* 2005b] used is a part of a heterogeneous system of the CHAMELEON/GECKO project. The coarse-grained reconfigurable part of the whole system containing several MONTIUM processing tiles is the target architecture of their work. The MONTIUM tiles take care of the highly regular computational intensive DSP kernels. Figure 3.9 shows a MONTIUM processor tile. Within each

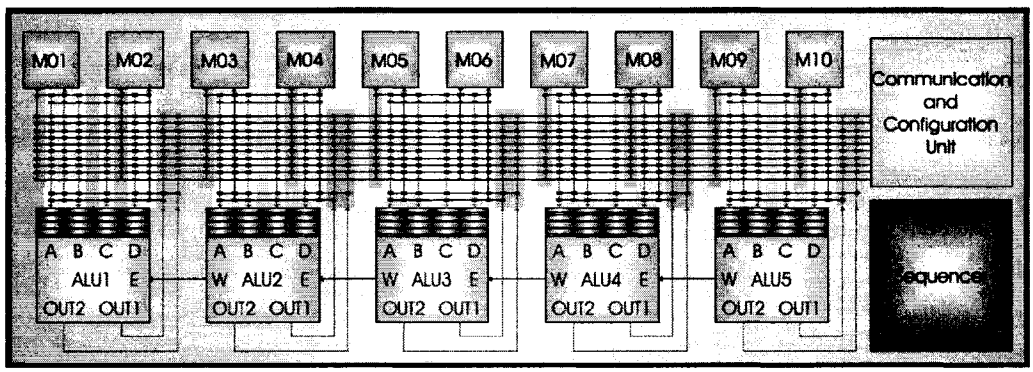

Figure 3.9: A MONTIUM tile [Guo *et al.* 2005b].

tile there are 5 ALUs, 10 local memories, a communication and configuration unit (CCU), a sequencer, and local and global interconnections. The ALUs can improve

performance by using spatial concurrency. Each ALU has four 16-bit inputs and two 16-bit outputs. Each input is associated with an input register file, storage capacity of which is four operands. The CCU communicates with outside the tile. The sequencer controls everything except the CCU inside a tile. The memory is also used as a lookup table to compute complex functions, which the ALU is not capable of.

3.3.2 Control Data Flow Graph

Guo *et al.* modeled the directed acyclic CDFGs, which includes both the control and data flow information, as hydra-graphs [Guo *et al.* 2005b]. A hydra-graph is represented by G = (N_G, P_G, A_G), where N_G is a finite non-empty set of nodes, P_G is a finite non-empty set of ports, and A_G is a set of hydra-arcs. A hydra-arc $a = (t_a, H_a)$ has one tail $t_a \in N_G \cup P_G$ and a non-empty set of heads $H_a \subset N_G \cup P_G$. Guo *et al.* represented the operations of the CDFG by N_G, the inputs and outputs of the CDFG by P_G, while the hydra-arc (t_a, H_a) either indicate that an input is used by an operation (if $t_a \in P_G$), or that the output of the operation $t_a \in P_G$ is the input of the operation, H_a, or that this output is merely an output of the CDFG (if a port of P_G is contained by H_a).

An important characteristics of a CDFG is that in CDFG an "if then else" construct is represented by a multiplexer and iterations are modeled by recursions. Loops are modeled as subgraphs of a CDFG with a multiplexer and a recursion. This simplicity of CDFG allow more flexibility to the mapping approach.

3.3.3 Structure of the Compiler

The compiler in [Guo *et al.* 2005b] is divided into four phases, namely **translation, clustering, scheduling**, and **resource allocation**. These four phases are performed in a sequential manner. But to negate any overhead due to this sequential nature, each phase considers the requirements of the later phases. Each of these phase is briefly described below:

3.3.3.1 Translation

This is the intermediate representation generation phase. The main task of this phase is to translate the target C application into a control data flow graph (CDFG). This phase is mostly independent of the target architecture. Since CDFGs may be cyclic, first the CDFG is partitioned into acyclic and cyclic blocks. The compilation process is only concerned with the acyclic portion. The cyclic portion is handled by the sequencer of the MONTIUM. Some other behavior-preserving transformations such as hierarchy expansion and optimizations are performed on the CDFG to extract information needed for the compilation phase.

3.3.3.2 Clustering

The transformed and optimized CDFG is partitioned into clusters in this phase. Clustering is done to minimize the number of ALUs, distinct ALU configurations, and length of the critical path of the CDFG. A cluster on a clustered graph is represented by an ALU configuration. The distinct templates are represented by distinct configurations and the distinct matches are represented by clusters. Figure 3.10 shows

an example CDFG and its two templates [Guo *et al.* 2005b]. These clusters are then

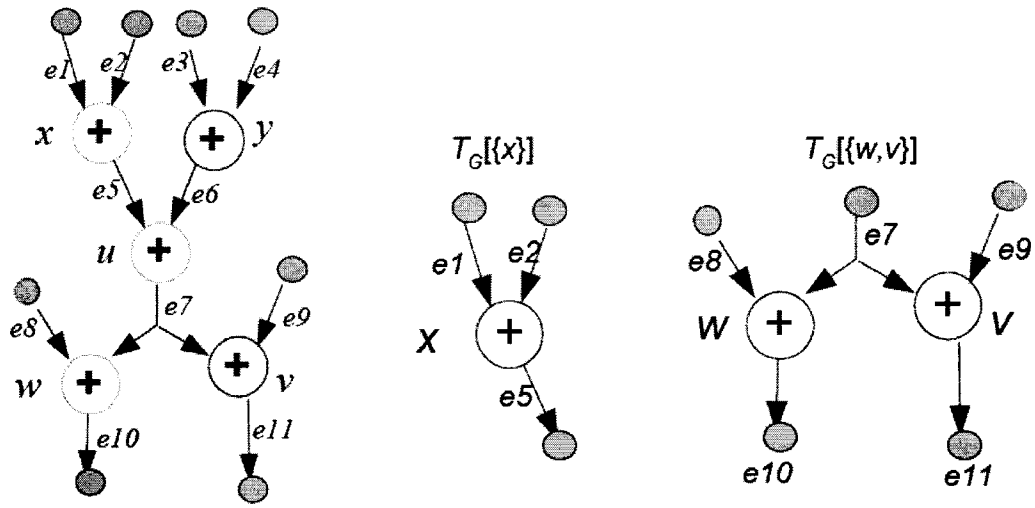

Figure 3.10: A small CDFG and its two templates [Guo *et al.* 2005b].

mapped to the ALUs.

A graph covering algorithm is used for clustering the nodes of the CDFG [Guo *et al.* 2003]. During clustering a cover of the CDFG is obtained. The clustering phases is composed of two steps: **template generation** and **template selection**.

In the template generation step the DFG is analyzed to extract functionally equivalent structures (templates). Here all the possible non-isomorphic templates and their corresponding matches are generated.

In the template selection step an optimal cover of the given CDFG is obtained such that the number of distinct templates and that matches are minimized. Templates are generated such that they represent one-ALU configuration. Templates actually model portions of a given application. The objective of template generation is to

generate a small number of larger templates, that can be mapped to an ALU and a small number of matches that partitions the nodes of the DFG. Templates should be smaller in number since the smaller the number of templates, the higher the execution rate of compilation. During generation of non-isomorphic templates, the nodes are labeled to minimize necessary calculations. For the template selection approach, Guo *et al.* adopted a heuristic based on maximum independent set, and then applied it to a conflict graph related to their problem.

3.3.3.3 Scheduling

The output graph of the clustering phase is scheduled in this phase. The objective is to obtain an optimal number of distinct configurations of ALUs of a tile. The task of scheduling is to order and schedule the clusters and allocate physical ALUs for each cluster such that the execution time is as small as possible. The scheduling phase also considers the constraints and limitations of the target architecture. So the output of this phase is two values for each cluster, a clock cycle and an ALU. These values determine in which clock cycle a particular cluster will be executed by which ALU.

Guo *et al.* used three algorithms for scheduling: the multi-pattern scheduling algorithm [Guo *et al.* 2005a], the column arrangement algorithm [Guo *et al.* 2006a] and the pattern selection algorithm [Guo *et al.* 2006b]. Figure 3.11 shows their mapping approach [Guo 2006]. The pattern selection algorithm selects a set of non-ordered patterns from the DFG. The column arrangement algorithm minimizes the number of configurations for each ALU. The multi-pattern scheduling algorithm schedules the DFG using the patterns selected by the other two algorithms. The algorithms use heuristics based on height-based priority functions.

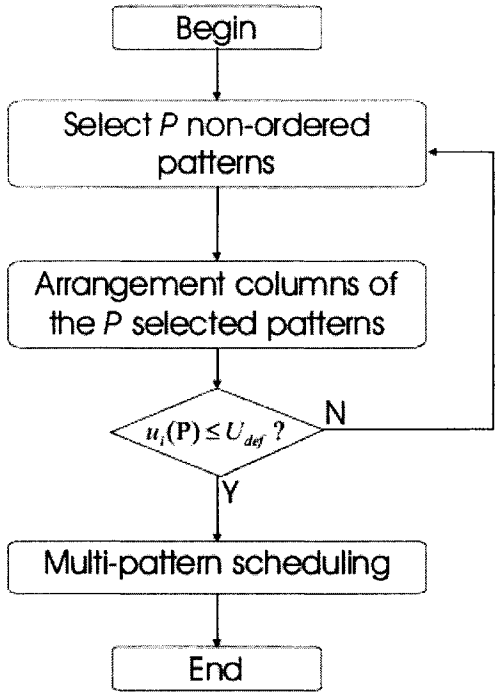

Figure 3.11: The Scheduling approach [Guo 2006].

3.3.3.4 Resource Allocation

In this phase, resources other than ALUs are allocated for the clusters and data moves are scheduled considering the constraints of resource limitations (such as reading/writing ports of memories, number of registers, crossbar, size etc.). This is the last step before generating code and it handles the inputs and outputs of the clusters. During this allocation, locality of reference is exploited to achieve high performance and low power consumption. The output of this phase is the executable assembly code.

In the allocation phase each intermediate value is assigned to appropriate memories or registers. Crossbars, address generators are arranged so that the outputs of the ALUs are stored in the proper registers and memories and resources are arranged

such that the inputs of ALUs are positioned in the appropriate register for the next cluster that will be executed on that ALU. A heuristic allocation algorithm is used for all these purposes [Guo *et al.* 2005b]. Data movements are performed for preparing inputs for and storing outputs of clusters. Resource constraints might increase the clock cycles needed for the schedule to achieve this.

There are several sub-steps of this phase like variable allocation, memory allocation, register allocation, scheduling data moves, and crossbar allocation. Each sub-step is performed one by one, considering the requirements of the later sub-steps. In this way an overall optimization is tried to be achieved.

3.3.4 Limitations

The limitation of this work is inherent in the limitation of the CDFG [Guo 2006]. For example, in CDFGs, one can not easily identify loops. Also the there is no clear specification of the number of iterations of a loop in the CDFG. Another limitation of this work is that it does not consider the cyclic parts of an application during compilation.

Chapter 4

Executable DFG from Source Language

This chapter will show how the input application will be presented to the back end of the compilation process. The input application is written using an explicitly parallel language HARPO/L and then after some transformation and optimization the intermediate representation is obtained in the form of executable data flow graph.

4.1 Introduction

The target application must be presented to a suitable target Coarse-Grained Reconfigurable Architecture (CGRA) for execution. That means, the application must be transformed into some form that is suitable for configuring to the CGRA. First the application must be written in some programming language. Then after compiler optimization we will have the intermediate representation in the form of data flow graph. This chapter gives an overall picture of this transformation.

4.2 Source Language Description

To implement any given application on a CGRA, the first step is to write it using a programming language. Various standard and modified programming languages have been used for that purpose. But almost every one of them has one problem or another. Some are not object-oriented, some are complex in structure, many of them do not have provision for concurrent execution [Norvell 2005]. In [Norvell 2006] a language named HARPO/L (**HAR**dware **P**arallel **O**bjects **L**anguage) has been designed that targets CGRAs as well as microprocessors. HARPO/L is a structured language with a co/co construct used to express parallelism. It is a parallel, object-oriented, multi-threaded programming language. The language design allows explicit parallelism. It also enables the compiler to extract inherent parallelism.

4.2.1 Overview

HARPO/L facilitates both object-orientation and parallelism. Both processes (threads) and shared resources can be expressed using objects. HARPO/L is more suited than either C++ or JAVA, both of which are object-oriented languages, to hardware implementation. It also facilitates concurrency for explicitly parallel programming. One of the differences with JAVA is that JAVA is mostly dynamic, whereas HARPO/L is purely static. In JAVA, objects are created and destroyed during runtime. Only classes are available at compile time. In HARPO/L both classes and objects are available at compile time. HARPO/L does not allow dynamic object creation. There is no use of pointers in HARPO/L.

4.2.2 Language Syntax

This section briefly summarizes HARPO/L according to [Norvell 2006]. The syntax of HARPO/L is given in Appendix A.

4.2.2.1 Classes and Objects

In HARPO/L a program is a set of classes, interfaces, and objects. Objects are named instances of types where type may be any of the following:

- Primitive types: The basic types in any programming languages fall in this category. Examples are int8, int16, int32, int64, int, real16, real32, real64, real, bool etc.
- Classes: Classes are used to generate objects.
- Interfaces: Interfaces are Classes without implementation.
- Arrays: Arrays of primitive types or arrays of objects are allowed.
- Generic Types: Instantiated generic types may be used.

In summary types are names of classes, array types or generic types.

Initialization of an object is in the form of an expression or an array initialization.

A class defines a type. Classes are either generic (having one or more generic parameters etc.) or nongeneric. Class members can be fields (objects within objects), methods and threads. Method declarations only declare the method, not its implementation. The implementation of a method must be embedded within a thread.

4.2.2.2 Threads

Threads are blocks of code whose execution is triggered by object creation. Each object can have zero or more threads within it. The programmer must manage the coordination of the threads within the same object. If there are multiple threads in a class, concurrency must be maintained within that object. The threads share the same address space and execute concurrently with each other.

A block is a sequence of statements where each statement may be an assignment statement, a local variable declaration, a method call statement, a sequential control flow statement, a parallel statement, or a method implementation statement, etc. In parallel statements, blocks of statements are separated from one another by ||. Multiple threads are synchronized and communicate using rendezvous construct using 'accept'.

Sequential consistency, an important correctness criteria for concurrent execution of a program, is implemented by atomic statements. The idea is that any two statements labeled 'atomic' within the same object cannot execute at the same time, unless they can not interfere with each other.

4.2.2.3 Genericity

Like JAVA's generic class or C++'s template class, HARPO/L allows classes and interfaces to be parameterized by generic parameters. They may be parameterized by other classes and interfaces, values of primitive types or objects. In general generic parameters may be nongeneric types, nongeneric classes, objects, or values.

4.2.3 Some Examples

In this section some examples illustrating the strength of HARPO/L are given. The example given below, taken from [Norvell 2006], implements the producer-consumer relation in a FIFO.

```
(class FIFO [in capacity : int, type T extends primitive]

    public proc deposit(in value : T)
    public proc fetch(out value : T)

    private obj a : T(capacity)
    private obj front := 0
    private obj size := 0

    (thread
        (wh true
            (accept
                deposit( in value : T ) when size < capacity
                    a( (front + size) % capacity ) := value
                    size := size + 1
            |
```

```
                fetch( out value : T ) when size > 0
                    value := a(front)
                    front := (front + 1) % capacity
                    size := size - 1
            accept)
        wh)
    thread)
class)
```

Below is an example Finite Impulse Response(FIR) Filter. The code is first written in C, and then using HARPO/L.

```
void FIR(int maxInput, int Taps) {
    for (i=0; i < maxInput; i++) {
        y[i] := 0;
        for (j=0; j < Taps; j++)
            y[i] := y[i] + x[i+j] * w[j];
    }
}
```

```
(class FIR [in maxInput : int, in Taps : int, obj x : real(), obj y : real(), obj w : real()]
    public proc run()
```

```
(thread
    (wh true
        (accept run()

            (for i : maxInput
                y(i) := 0
                (for i : Taps
                    y(i) := y(i) + x(i+j) * w(j)
                for)
            for)
        accept)
    wh)

thread)

class)
```

We will now give another example of Infinite Impulse Response(IIR) Filter. First we give the pseudocode and then we write the code using HARPO/L.

```
proc IIR() {

    for i=1 to 100 do
        for j=1 to 80 do
```

```
            sum = 0
            for k=1 to 10 do
                sum = sum + a[i,k]*y[i,j-k-1]
            end for
            y[i][j] = x[i,0] * x[i,i] - sum
        end for
    end for
}
```

```
(class IIR [obj a : real(), obj x : real(), obj y : real()]

    public proc run()

    (thread
        (wh true
            (accept run()

                (for i : 100
                    (for j : 80
                        sum := 0
                        (for k : 10
                            sum := sum + a(i,k) * y(i,j-k-1)
                        for)
```

```
                    y(i,j) := x(i,0) * x(i,j) - sum
                for)
            for)
        accept)
    wh)

thread)

class)
```

4.3 Intermediate Representation : Input of compilation

The target applications will first be written in HARPO/L, which is a parallel, object-oriented language. We are targeting parallel applications. For that our intermediate representation will be in the form of Data Flow Graph (DFG). This will be the input of our compilation and must have capability to express parallelism and concurrency.

Some well known representation of intermediate program representation are Control Flow Graphs (CFG) [Aho *et al.* 1986], def-use chains [Aho *et al.* 1986], Program Dependence Graphs [Ferrante *et al.* 1987], Dependence Flow Graphs [Pingali *et al.* 1991], Static Single Assignment [Cytron *et al.* 1989], etc. Of these SSA has been popular over the last decade or two. We will first give a brief overview of SSA.

4.3.1 Static Single Assignment Form

For sequential programs, Static Single Assignment (SSA) form has been an established and efficient intermediate representation [Cytron *et al.* 1991]. In SSA form each variable has exactly one definition point during its lifetime which can reach its uses. Multiple reaching definitions are combined into a single definition by using ϕ-functions at the merge points of the program flow. Moreover, each defintion of a variable is renamed and the appropriate change is carried out at the uses of the variable to reflect the appropriate (single) reaching definition.

Conversion of a sequential program into SSA from uses the Control Flow Graph (CFG) of the program. After conversion of a program into SSA representation, the resulting program exhibits two important characteristics:

- Every use of any variable in the program has exactly one reaching definition. This removes unrelated uses of the same variable name from the original source code and

- Merge functions known as ϕ-functions are inserted at appropriate confluence points in the CFG. A ϕ-function for a variable combines the values of the variable from distinct incoming control flow paths at control flow merge points, thus preserving the property that each variable has a unique definition site. A ϕ-function has the form $V'=\phi(v_1, ..., v_n)$ where $V', v_1, v_2, ..., v_n$ are variables and n is the number of incoming control flow edges for the node where the ϕ-function is placed.

Figure 4.1 shows the SSA form of the simple program on the left.

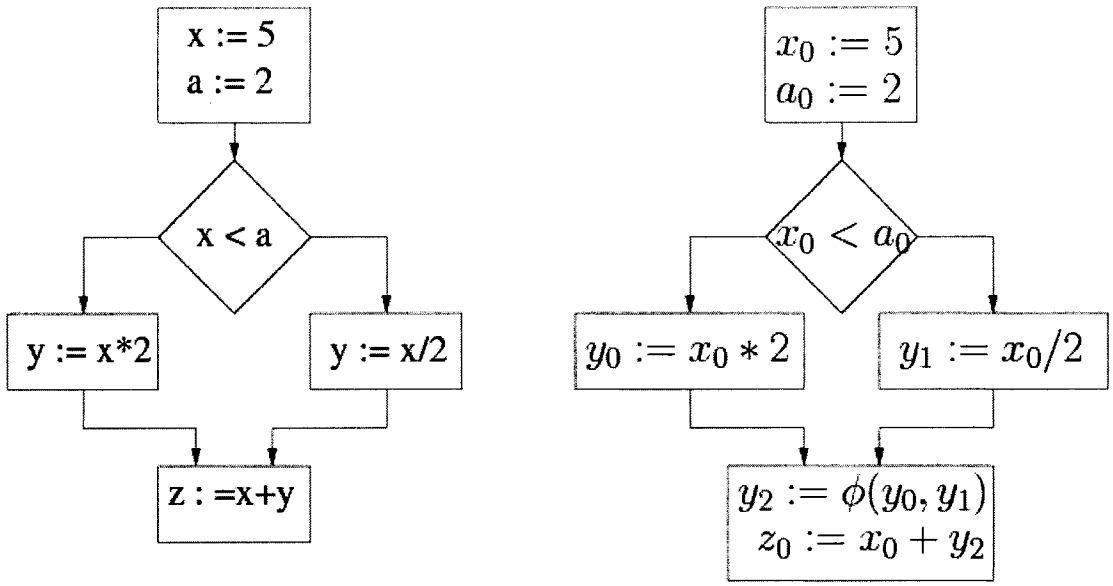

Figure 4.1: A simple program (a) and its single assignment version (b).

4.3.2 Concurrent Static Single Assignment Form

Since we are dealing with Parallel Programs, SSA will not meet our needs. Concurrent Static Single Assignment (CSSA) for was built for that purpose [Lee *et al.* 1997, Lee *et al.* 1999]. Like SSA, every use of a variable has exactly one definition in CSSA. The Concurrent Control Flow Graph (CCFG), which is the parallel counterpart of the control flow graph of a sequential program, is used as the intermediate representation for explicitly parallel programs in order to convert them into CSSA form. CCFG contains information about conflicting statements along with control flow and synchronization information.

The CSSA form has three confluence functions: ϕ, π, ψ. They, according to [Lee *et al.* 1997, Lee *et al.* 1999], are defined as follows: ϕ-function in CSSA is similar to that in SSA. The ϕ-function merges all the control reaching definitions to create

a new definition for the variable. It distinguishes values of variables coming from distinct incoming control flow edges.

A ϕ-function has the form $\phi(v_1, v_2, ..., v_n)$, where n is the number of incoming control flow edges of the node where it is placed. The value of $\phi(v_1, v_2, ..., v_n)$ is one of the v_i's and the selection depends on the control flow path followed by the program.

The π-function distinguishes values of variables coming from incoming control flow edges and distinct conflict edges labeled DU. It summarizes the interleaving of defining assignments of shared variables in *Cobegin/Coend* and parallel do constructs from different threads. They are placed at the point where there is a use of a shared variable with DU conflict edges.

A π-function for a shared variable v has the form $\pi(v_1, v_2, ..., v_n)$, where n is the number of reaching definitions to the use of v through the incoming control flow edges and incoming conflict DU edges. The value of $\pi(v_1, v_2, ..., v_n)$ is one of the v_i's and the selection is dependent on the interleaving of statements in the threads computing v_i's.

ψ-functions, on the other hand, are placed at such confluence points as *Coend* and *Endpdo* nodes. They are basically introduced by *Cobegin/Coend* and parallel do constructs.

A ψ-function for a shared variable v has the form $\psi(v_1, v_2, ..., v_n)$, where n is the number of threads merging at a *Coend* node or an *Endpdo* node where the ψ-function is placed. The value of $\psi(v_1, v_2, ..., v_n)$ is one of the v_i's and the selection depends on the interleaving of statements in the threads merging at the node.

4.3.3 Dependence Flow Graphs

We can not use intermediate representation like control flow in the form of CSSA in parallelizing compilers for execution on CGRA. Rather data flow graphs serves this purpose. The reason is data flow graphs act as an executable intermediate representation in parallelizing compilers [Beck *et al.* 1991]. Beck *et al.*'s work shows how to translate control flow into data flow. Their idea for the transformation was to use tokens along the flow path. A variable definition and its use points are termed as a token producers and token consumers respectively. They implement the def-use chain in the CFG by flow of tokens from producer to consumer. They proposed an executable Dependence Flow Graph.

4.3.4 Static Single Information Form

Static Single Information [Ananian 1999], a generalization of SSA form, is another intermediate program representation. SSI extends SSA without adding unnecessary complexity to allow efficient predicate analysis and backward data flow analysis. SSI form improves backward flow analysis. In SSI form, if the definition of a variable at program point x reaches two uses of the same variable at program points y and z, then either all paths from x to y contain z or all paths from x to z contain y. To satisfy this SSI condition, statements of the form $x_0, x_1 := \phi^{-1}(x)$ are used in the transformed program to create multiple copies of a single variable.

4.3.5 Static Token Form

Static Token (ST) [Teifel and Manohar 2004] form is an intermediate representation, which extends the SSI form. ST is used for for converting sequential programs to concurrent data flow graphs. Teifel and Manohar present a compiler framework that automates this synthesis method and also showed how it can be used to synthesize logic for pipelined asynchronous FPGAs.

The ϕ and the $\sigma = \phi^{-1}$ functions of SSI are also present in ST form. But ST form re-interprets their execution. For example, the statement $x := \phi(x_0, ..., x_{n-1})$ in SSI form will be $x := \phi_g(x_0, ..., x_{n-1})$ in ST form where g is an integer depending on whose value ϕ function is executed. A statement $x_0, x_1 := \phi_g^{-1}(x)$ generates one of two possible copies of a variable x using the condition g – one when the guard is false and the other when the guard is true.

In static token form, variables are transformed into tokens and the pipelining of asynchronous computations is simplified following the data flow graphs. As in [Beck *et al.* 1991], variable definitions are treated as data token producers and variable uses as data token consumers. It is the responsibility of the compiler to connect token producers and token consumers to correctly represent the original sequential program. After the transformation, each data flow graph node represents a concurrent asynchronous pipeline stage and is one of seven simple process types: *copy*, *function*, *split*, *merge*, *source*, *sink*, and *initializer*.

Their seven types of concurrent data flow nodes are shown in Figure 4.2 and their functionality in CHP is given below. The CHP notation used by Teifel and Monohar is based on Hoare's CSP [Hoare 1978]. A short and informal description of CSP is

given in [Teifel and Manohar 2004].

$Copy \equiv *[A?a;\ Z_0!a,\ ...,\ Z_{n-1}!a]$

$Function \equiv *[A_0?a_0, ...,\ A_{n-1}?a_{n-1}; Z!f(a_0, ...,\ a_{n-1})]$

$Split \equiv *[C?c,\ A?a;\ [c = 0 \rightarrow Z_0!a\,|...|\,c = n-1 \rightarrow Z_{n-1}!a]]$

$Merge \equiv *[C?c;\ [c = 0 \rightarrow A_0!a\,|...|\,c = n-1 \rightarrow A_{n-1}!a];\ Z_a]$

$Source \equiv *[Z!"constant"]$

$Sink \equiv *[A?a]$

$Initializer \equiv a := "constant"; Z_0!a,\ ...,\ Z_{n-1}!a; *[A?a; Z_0!a, ..., Z_{n-1}!a]$

Now we will show an example of a control flow graph (expressed in SSI) and its corresponding data flow. Figure 4.4 shows a HARPO/L statement on the left and its data flow graph on the right. Its SSI form is shown in Figure 4.3.

4.3.6 Static Token for Parallel Programs

We cannot directly apply the static token approach, since static token deals with sequential form, and we are dealing with parallel concurrent forms. Zhang [Zhang 2007] extends the work of Teifel and Monohar [Teifel and Manohar 2004] so that static token form works for parallel programs also. According to him the parallel program must first be analyzed for synchronization. Multiple threads running in a parallel program need this synchronization during execution in order to maintain sequential consistency. Sequential consistency according to Lamport [Lamport 1979] is defined as follows:

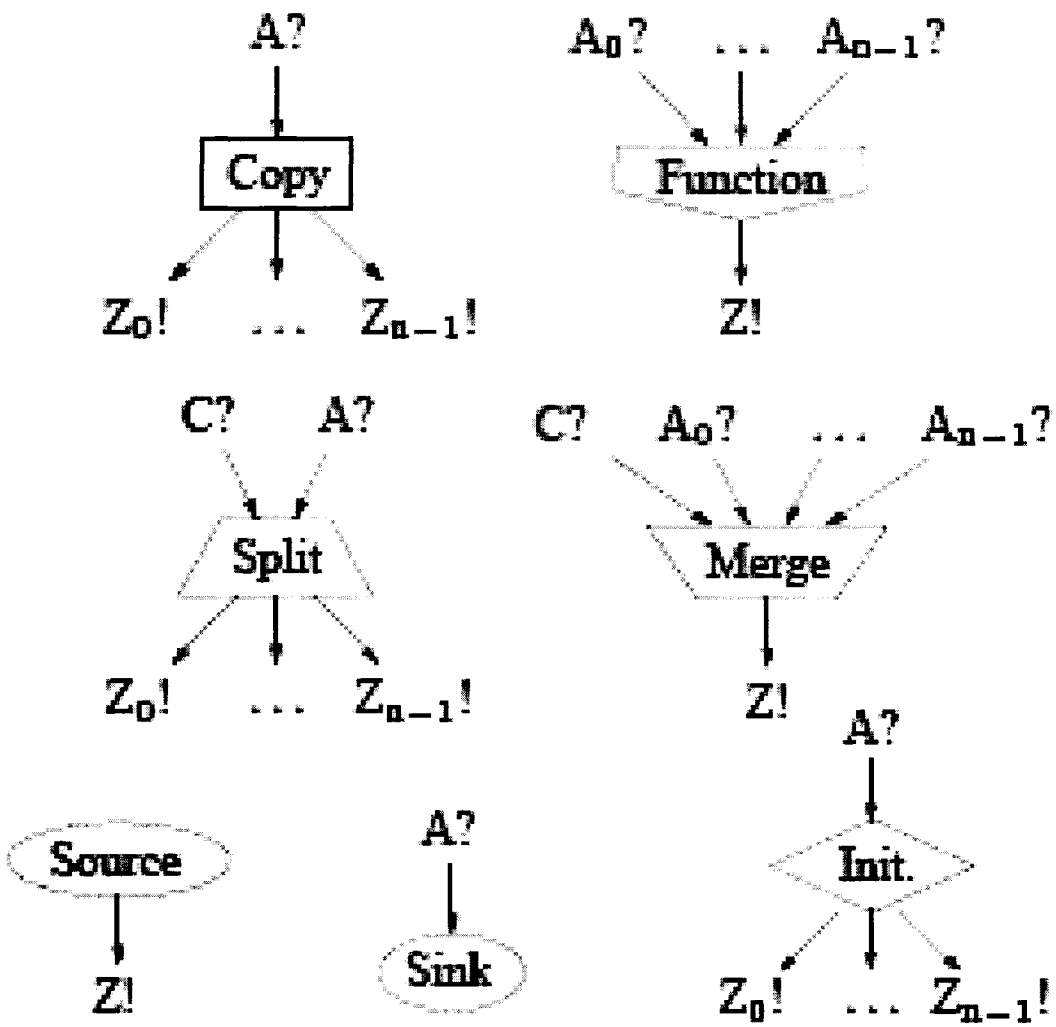

Figure 4.2: Concurrent Data Flow Nodes [Teifel and Manohar 2004].

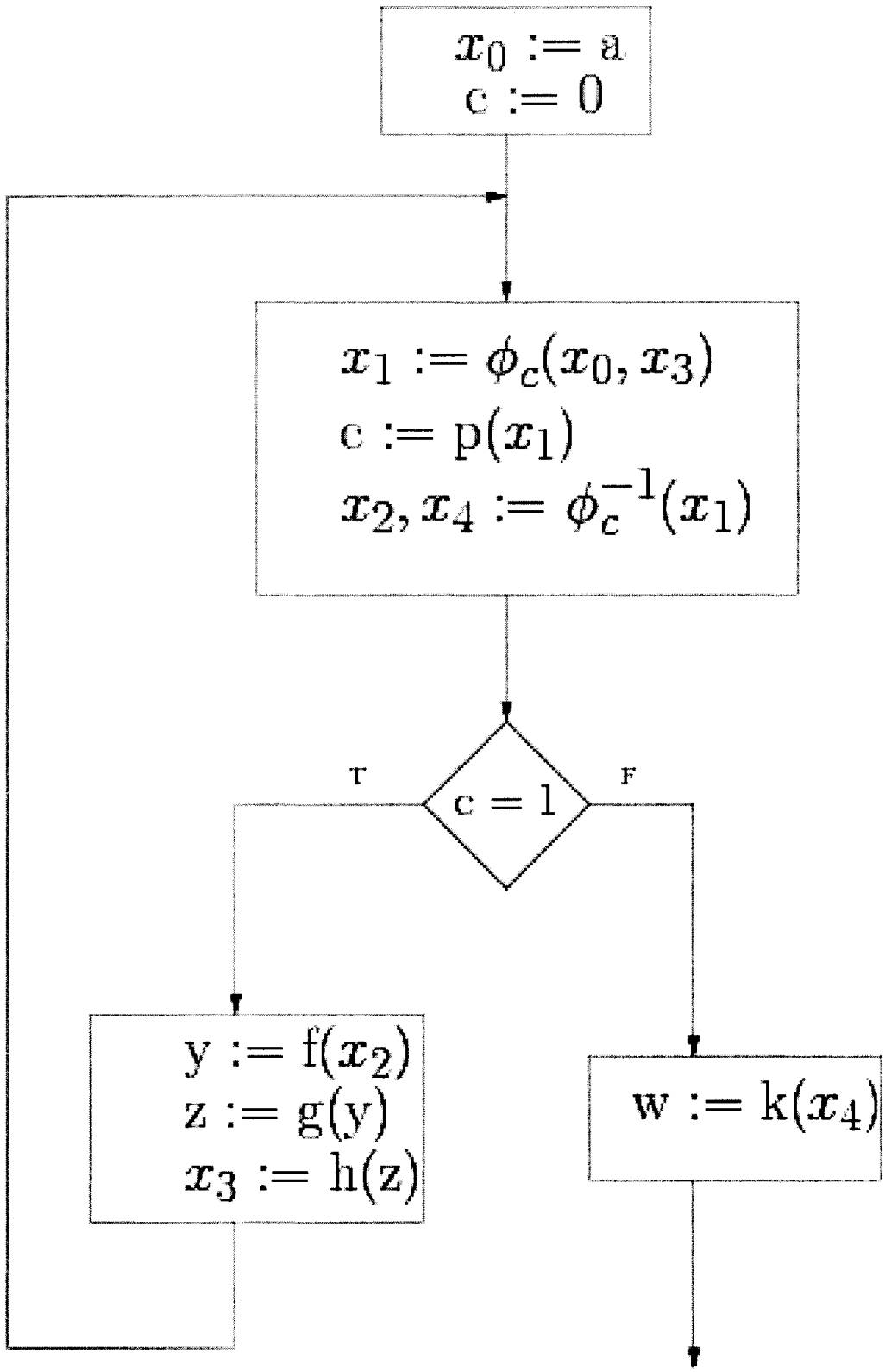

Figure 4.3: Single Static Information form.

```
x := a;
(wh p(x) = 1
    y := f(x);
    z := g(y);
    x := h(z);
wh)
w := k(x);
```

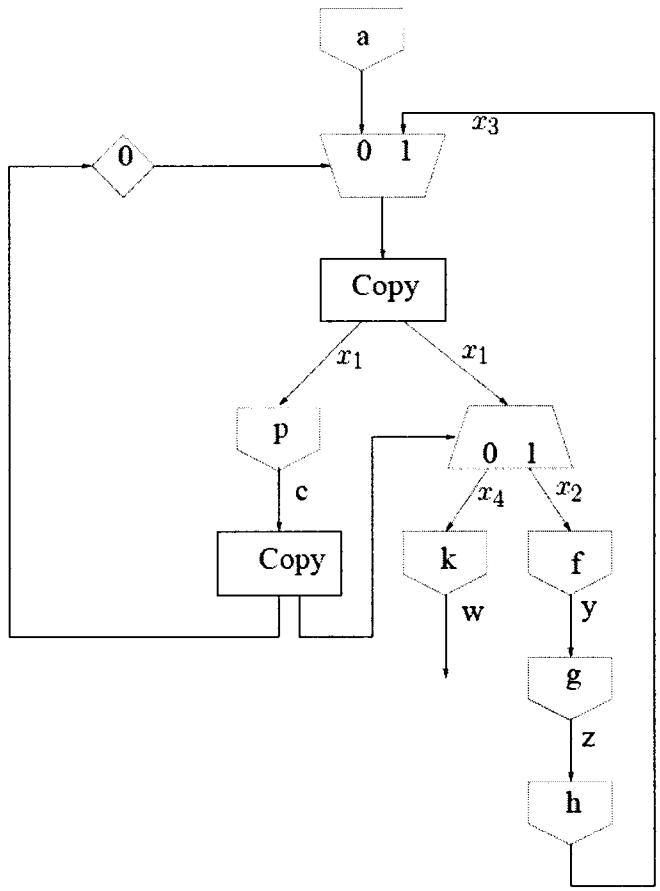

Figure 4.4: Data flow Graph using Static Token form.

A result of an execution of a parallel program P is sequentially consistent if it is the same as the result of an execution where all operations were executed in some sequential order, and the operations of each individual thread occur in this sequence in the order specified by P.

Zhang also implements the ψ, π functions and introduced a new function ζ for procedural coordination.

4.3.7 Executable DFGs: The Input

We will use the executable data flow graphs, represented by static token form for parallel program [Zhang 2007], as input for scheduling, placement and routing. We can represent our input executable data flow graph as a Graph $DFG = (N, E, op, inRole, outRole)$ such that:

- N is the set of nodes.
- E is set of edges labeled by *roles*. $e^{\leftarrow}$ represents the source of an edge, while $e^{\rightarrow}$ represents the target of that edge. *inRole* is a function $E \longrightarrow roles$, while *outRole* is a function $E \longrightarrow roles$.
- *op* is a function: $N \longrightarrow operations$, which labels nodes with operations.

Chapter 5

Target Architecture and Compilation Framework

In this chapter we first describe our target architecture and then we will present an overview of our compilation process. We will also outline some issues during compilation.

5.1 Target Architecture Description

5.1.1 Architecture Overview

Our aim is to design and implement a retargetable compiler in such a way that it is compatible with a large range of target architectures. The user is responsible for providing a description of the target architecture that will be used as an input to the compiler. The compiler should be flexible enough to cope with the target architecture given by the user. But what the compiler demands from the architecture describer is

some architectural parameters and characteristics in the target architecture description. The user will be given a guideline by following which he/she can define the target architecture. Among those parameters and characteristics are size, topology, component functionality, memory requirements for each component, register file, etc. We will briefly describe each of them below.

5.1.1.1 Size

By size we mean total number of processing components. Size can be any number that the user wishes with respect to the overall cost of the architecture. By overall cost here we mean the total cost for each node, the interconnection cost, and the memory cost.

5.1.1.2 Component Functionality

It is up to the user to choose different types of components with different functionality. The simplest will be regular (i.e., homogeneous) where all the processing components do the same computation. In other words, the functionality of each processing component can be homogeneous or heterogeneous. Though from a practical point of view, heterogeneity is better, it makes the mapping process more challenging. For example, a processing component can be a single functional unit (e.g., adder or subtracter), an ALU, a complete processor, etc.

5.1.1.3 Topology

This is one of the most important factors of target architecture description. The reason is if every processing component is connected with every other, then routing is

trivial. Otherwise, we need to consider routing of values from producers to consumers. The fewer the number of connections, the harder mapping is. The reason for it is that with fewer connections, the possibility of finding unoccupied connections is less. Although with larger number of connections, a choice between several alternate routes has to be made. This process may be time consuming. Topology of the target architecture can be anything like mesh based, star shaped, crossbar, linear, array, hexagonal (like CHESS [Marshall *et al.* 1999]).

5.1.1.4 Memory Requirements

Memory serves as the storage location for variables, arrays, etc. A PE fetches its input from memory and stores its output into memory. So the amount of memory specified by the architecture definer plays an important role for mapping. The reason is: if there is less storage than needed in a particular clock cycle, then the execution time will be increase for the overall application.

5.1.1.5 Register File

Register files are another important component of a target architecture. The performance of compilation is affected greatly by its amount and its availability. Register files may be embedded in a target architecture in different ways. In one approach, each functional unit (FU) may have a register file attached with it. In another approach there may be sharing of register files among some number of FUs. From performance point of view, the first approach is efficient, but it also is expensive with respect to cost. In another approach, there may be a central register file along with the shared ones (like VLIW). But that approach has a bottleneck due to a limited

number of memory ports. A register file can have variable number of registers and read/write ports.

5.1.2 Framework of Target Architecture

This section gives some overview of the sample target architecture. Our goal is to design and implement a retargetable compiler for any target architecture. What will be the description of the target architecture depends on the architecture designer. But there should be a framework following which he/she will describe it. We now describe the framework of Target Architecture.

Number of Type of different functional units (t)

$Type_1$

$Type_2$

...

...

...

$Type_t$

Number of functional units (f)

FU_1[: $Type$]

FU_2[: $Type$]

...

...

...

$FU_f[: Type]$

Number of registers (r)

Total number of interconnections among functional units (c)

$IC_1 : FU_x - FU_y$

$IC_2 : FU_x - FU_y$

...

...

...

$IC_c : FU_x - FU_y$

Total number of interconnections between functional units and registers (k)

$IC_1 : FU_x - Reg_y$

$IC_2 : FU_x - Reg_y$

...

...

...

$IC_k : FU_x - Reg_y$

Number of Central Register File(CRF)

If there is only one type of functional unit, then specifying the type of each FU is optional. The interconnection is usually bidirectional. That means, if there is a connection between two FUs from one to the other, then this information is specified by giving the FU-pair once. This follows from the fact that if there are a total n FUs, and each FU is connected with each other, then the total number of interconnections will be $\frac{n(n-1)}{2}$, not n^2. Unidirectional interconnections can also be incorporated. If

the total number of registers is equal to the total number of functional units, then one can easily conclude that there are dedicated register files and there is no sharing of register files. Otherwise, by looking at $\frac{f}{r}$ one can determine how many FUs share a register file. For central register file, CRF will be specified by 0 in the architecture description if there is no central register file present. For example following this framework one might describe a possible target architecture as follows:

1

ALU

4

FU_1

FU_2

FU_3

FU_4

$IC_1 : FU_1 - FU_2$

$IC_2 : FU_1 - FU_3$

$IC_3 : FU_2 - FU_4$

$IC_4 : FU_3 - FU_4$

0

0

A pictorial view of the given target architecture is shown in Figure 5.1. There is no register or central register file. In this way the target coarse-grained reconfigurable architecture can be described. For more clarity, we conclude this section with some more representative target architectures as shown in Figure 5.2, Figure 5.3,

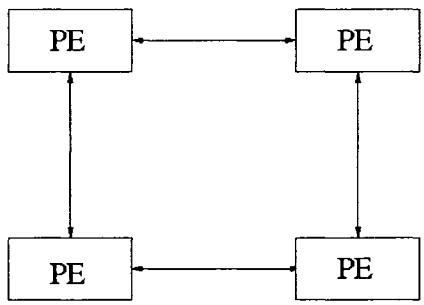

Figure 5.1: A simple target architecture.

Figure 5.4. The description of the target architecture of Figure 5.2 is given below.

1

ALU

5

FU_1

FU_2

FU_3

FU_4

FU_5

$IC_1 : FU_1 - FU_2$

$IC_2 : FU_1 - FU_3$

$IC_3 : FU_1 - FU_5$

$IC_4 : FU_2 - FU_4$

$IC_5 : FU_2 - FU_5$

$IC_6 : FU_3 - FU_4$

$IC_7 : FU_3 - FU_5$

$IC_8 : FU_4 - FU_5$

0

0

The descriptions of the target architecture of Figure 5.3 and Figure 5.4 are similar.

The characteristics of these target architecture is that the functional units are the same, i.e., they are homogeneous, although there may be heterogeneous PEs in a given target architecture.

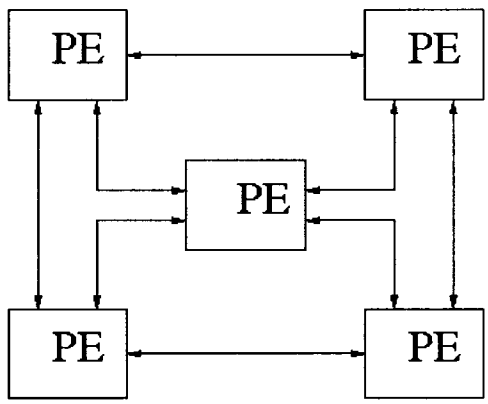

Figure 5.2: Another simple target architecture.

5.1.3 Our Sample Target Architecture

To compare our work and demonstrate the proposed compilation technique, we are considering 16 processing elements (PEs) arranged in a 4x4 design and each PE has connection with its four neighboring PEs as shown in Figure 5.5. The description of

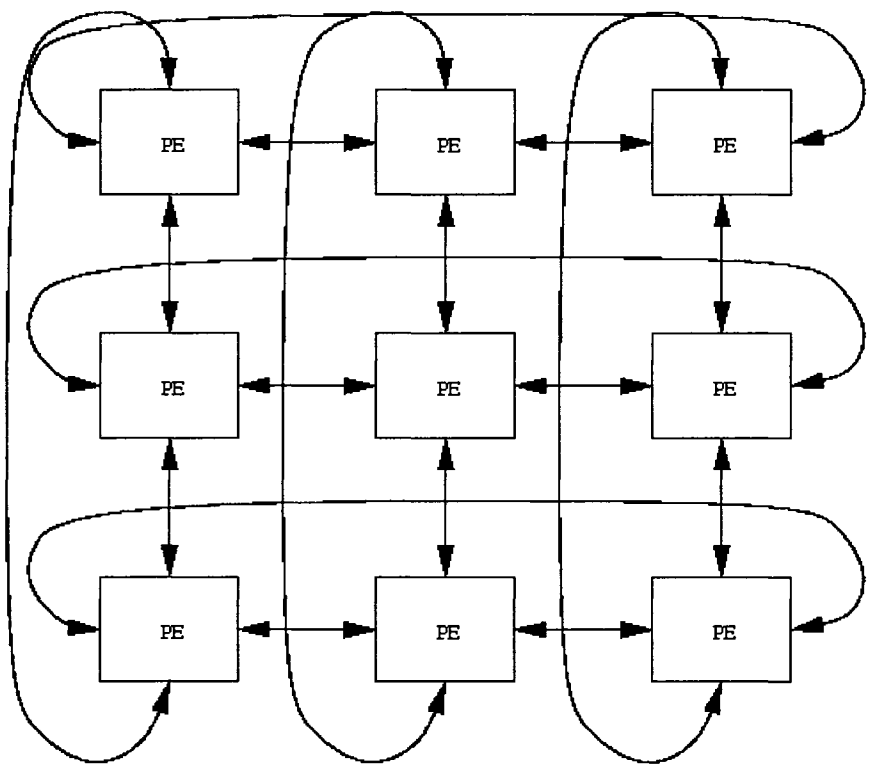

Figure 5.3: Another simple target architecture.

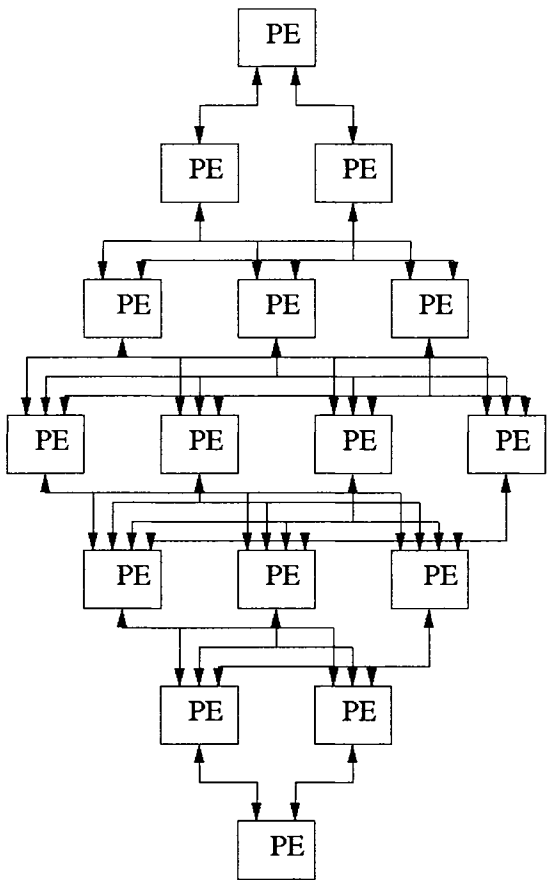

Figure 5.4: Another simple target architecture.

our target architecture is given below:

1

ALU

16

PE_1

PE_2

...

PE_{15}

PE_{16}

$IC_1 : PE_1 - PE_2$

$IC_2 : PE_2 - PE_3$

$IC_3 : PE_3 - PE_4$

$IC_4 : PE_1 - PE_4$

$IC_5 : PE_1 - PE_5$

$IC_6 : PE_2 - PE_6$

$IC_7 : PE_3 - PE_7$

$IC_8 : PE_4 - PE_8$

$IC_9 : PE_1 - PE_{13}$

$IC_{10} : PE_2 - PE_{14}$

$IC_{11} : PE_3 - PE_{15}$

$IC_{12} : PE_4 - PE_{16}$

...

$IC_{29} : PE_{13} - PE_{14}$

$IC_{30} : PE_{14} - PE_{15}$

$IC_{31} : PE_{15} - PE_{16}$

$IC_{32} : PE_{13} - PE_{16}$

0

0

Figure 5.6 shows a detailed view of one PE. Each PE has a functional unit (FU), a register file, an output register, a configuration RAM, and some MUXes (MUltipleXers). For the sake of simplicity we are assuming homogeneous functional units (FUs) although the proposed algorithms can easily be generalized for heterogeneous FUs. MUXes serve as the selection of inputs from various sources such as neighboring PEs. The inputs of a PE can come from neighbor PEs or its own register file. Output of a FU is saved temporarily to the output register from where it can either go to a neighbor PE or be saved to the register file of the same PE. Each PE uses input values generated in earlier cycles, possibly by other PEs. It outputs a new value that is routed to the places where it is needed. Configuration memory are used to provide control signal for the FUs and the MUXes. There is a dedicated register file for each PE, but there is no central register file or register file sharing. The reason is central register files increase the cost of a target architecture. Moreover, if our compilation techniques can be applied without a central register file, it can also be adapted for a target architecture having a central register file. Besides performing operations, each PE is responsible for routing values whenever needed, although a PE cannot do both at the same time.

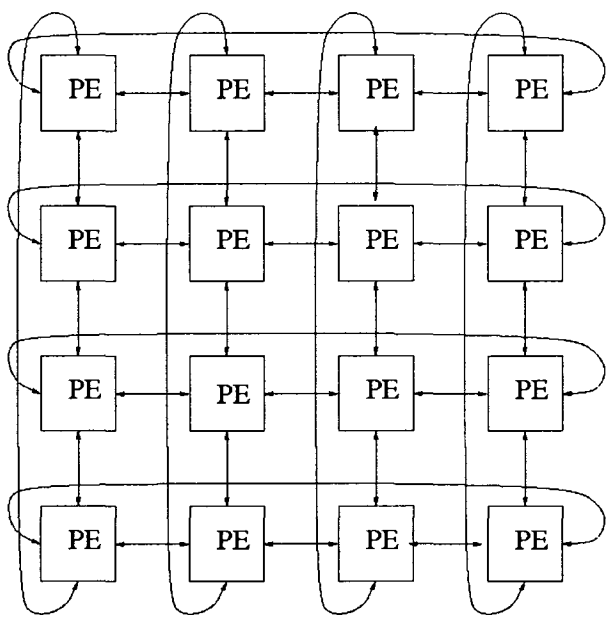

Figure 5.5: Organization of our target architecture for experimental purposes.

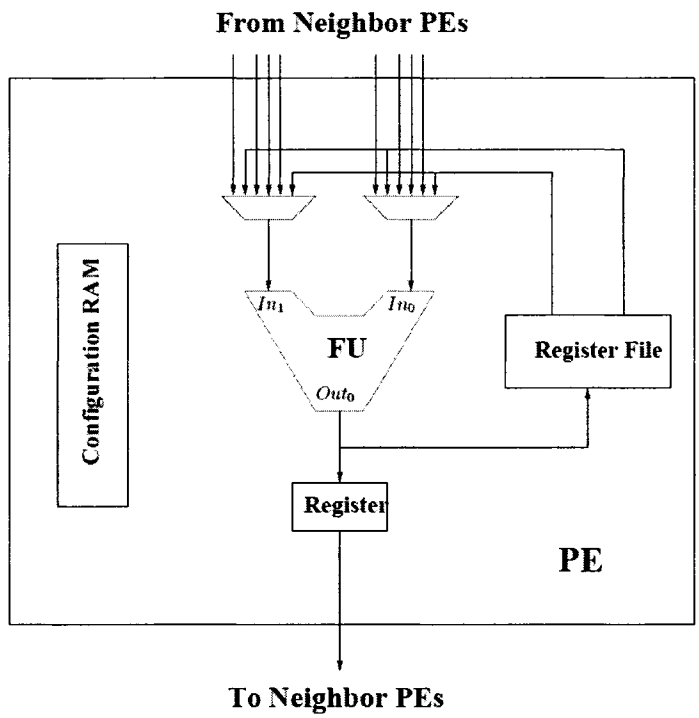

Figure 5.6: A detailed view of a Processing Element (PE).

5.1.4 Transformation of Architecture Description

The target architecture designer will describe various aspects of the target architecture. But that description cannot be used directly for compiling applications. We need to somehow transform the description into a form that is compatible with other input of the compilation, the executable data flow graph transformed form the target application written in HARPO/L. [Mei *et al.* 2002] introduced Modulo Routing Resource Graph (MRRG) for their compiler and [Park *et al.* 2006] slightly simplified it. But MRRG can be used when only considering the loop body. Since we are targeting the whole application, we cannot use MRRG directly. But we can consider the transformation of the target architecture into MRRG whenever we consider only a particular loop portion of the application. We can represent the target architecture specified by the architecture designer by a graph $TA = (C, R)$, such that:

- $C = FU \bigcup RF$ is the set of functional units and registers.
- R is the set of interconnections where $r^{\leftarrow}, r^{\rightarrow} \in C$ for each $r \in R$.

For compilation purposes we have modeled our target architecture with routing resource graph (RRG). RRG is basically obtained by replicating the target architecture graph TA an infinite number of times and giving necessary interconnections across time cycles.

An *RRG* is a directed graph $RRG = (C \times \mathbb{N}, A \bigcup B \bigcup D)$. $A \bigcup B \bigcup D$ is the set of interconnections in a time cycle and across time cycles in a forward direction. Here $C \times \mathbb{N}$ is the vertices of the graph, i.e., resources of the target architecture replicated across time.

The set A, B, and D and the interconnection relations for their edges can be expressed as follows:

- $A = \{(i, r) | r^{\rightarrow} \in FU, i \in \mathbb{N}\}$

  $(i, r)^{\leftarrow} = (r^{\leftarrow}, i)$, for all $(i, r) \in A$

  $(i, r)^{\rightarrow} = (r^{\rightarrow}, i)$, for all $(i, r) \in A$

- $B = \{(i, r) | r^{\rightarrow} \in RF, i \in \mathbb{N}\}$

  $(i, r)^{\leftarrow} = (r^{\leftarrow}, i)$, for all $(i, r) \in B$

  $(i, r)^{\rightarrow} = (r^{\rightarrow}, i + 1)$, for all $(i, r) \in B$

- $D = \{(i, f) | f \in RF, i \in \mathbb{N}\}$

  $(i, f)^{\leftarrow} = (f^{\leftarrow}, i)$, for all $(i, f) \in D$

  $(i, f)^{\rightarrow} = (f^{\rightarrow}, i + 1)$, for all $(i, f) \in D$

In the above, set B represents information being stored, while set D represents information being retained. So overall, a functional unit may be connected to another functional unit of the same time cycle. It is also connected to a register file of the next cycle. A register file may be connected to a functional unit of the same cycle. A register file is also connected to a register file of the next cycle.

Figure 5.7 illustrates the routing resource graph (on the right) for the sample target architecture (on the left). We can see from this figure that the target architecture graph has been replicated four times and the corresponding connections has been incorporated to model the flow of information across time cycles. Here the dashed lines mean connections in the same time cycle, whereas the solid lines mean connections across time cycles.

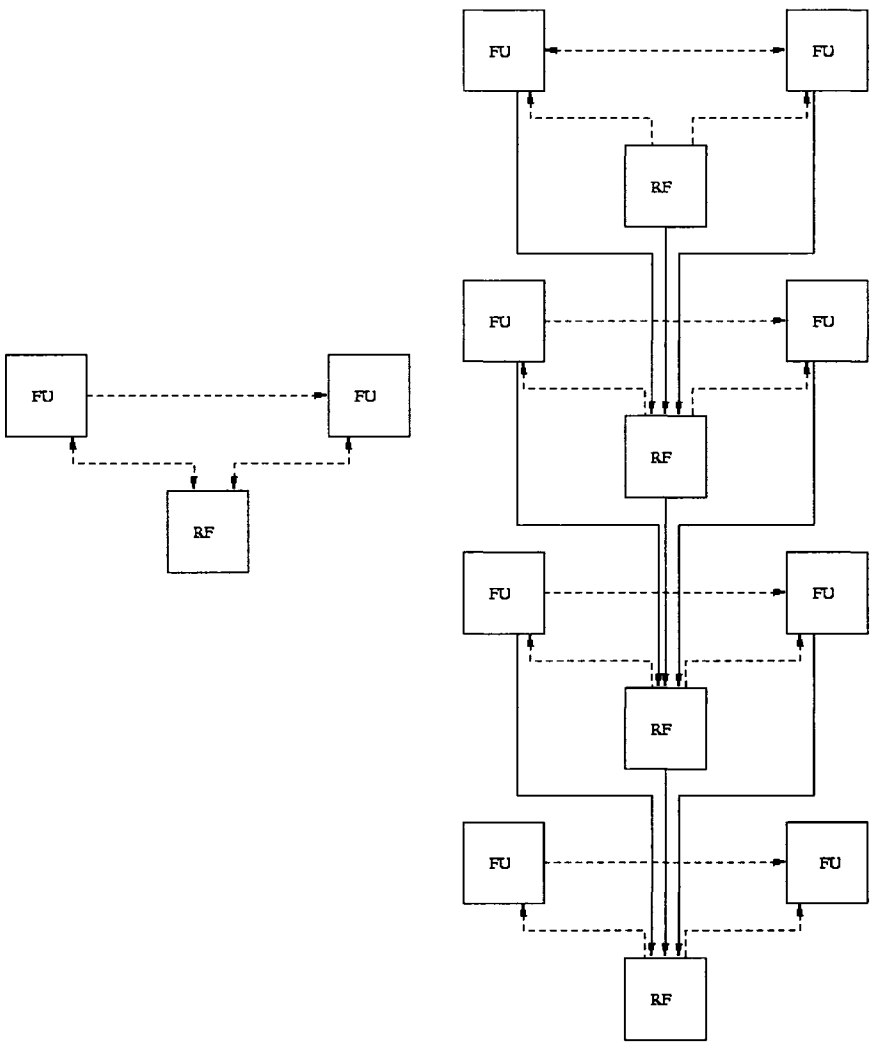

Figure 5.7: A sample Routing Resource Graph.

5.2 Traditional Compilation Approach

Compilation of applications onto CGRAs is highly dependent on the target architecture. Since there are some similarities among all the CGRAs, there must be some similarities among the compilation approach also. Initially the given application is written in a high level programming language. Then the source code is analyzed, transformed and optimized to obtain the desired intermediate representation (IR) in the form of data flow graphs. During IR steps possible parallelism is extracted. This process is beneficial because the more parallelism we can extract from the source code, the better will be the performance of compilation. After the DFG is available, it is analyzed during the clustering phase to extract a set of clusters with respect to the target architecture. Then these clusters are mapped to the target architecture during the mapping phase. The scheduling phase determines which cluster will be executed in which clock cycles. During this phase the available resources and memory are utilized properly to extract maximum benefit. During the routing phase operand values are routed using the available interconnection network from producer PE to consumer PE. After this phase the desired code is generated and converted to binary form for execution. The overall compiler flow of a traditional compiler for CGRA is shown in Figure 5.8.

5.3 Framework of Overall Compilation Approach

Our objective is to compile applications that have parallelism onto coarse-grained reconfigurable architectures (CGRAs). For that the target application is first written

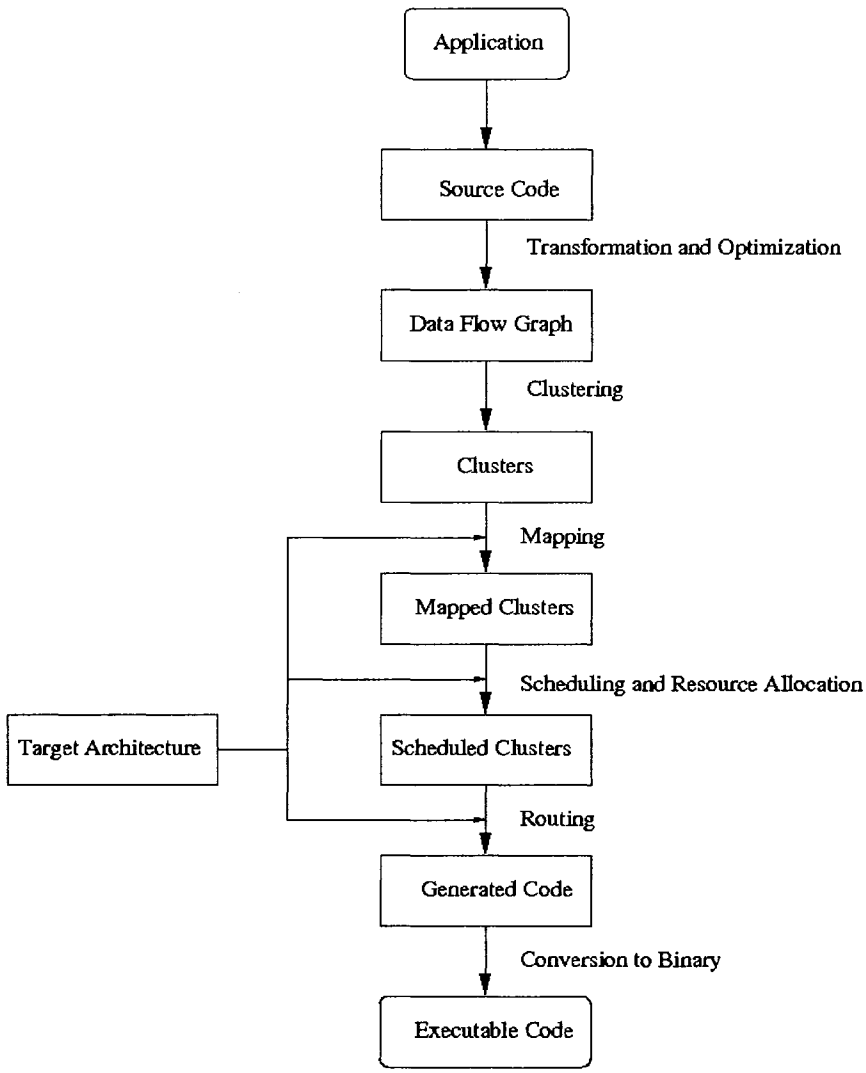

Figure 5.8: General Structure of Compilation flow for CGRA.

in HARPO/L, a parallel object-oriented concurrent programming language suitable for CGRAs. Then the source code is transformed and optimized to get the intermediate representation in the form of an executable data flow graph. Here we have used the static token form for parallel programs. This executable DFG and the target architecture in the form of RRG are the two inputs of the back end of our compilation procedure. The output of compilation will be the executable code for the given CGRA and the given application.

Figure 5.9 shows the framework of our overall compilation approach.

5.4 Overview of our Compilation flow

The compiler plays a critical role in the success of a coarse-grained reconfigurable architecture (CGRA). The compiler must carefully schedule code to make the best use of the abundant resources available in a CGRA. Compiling applications to CGRAs, after the source code of the target application has been transformed and optimized to a suitable intermediate representation, is a combination of three tasks: scheduling, placement, and routing. Scheduling assigns time cycles to the operations for execution. Placement assigns these scheduled operation executions to specific processing elements. Routing plans the movement of data from producer PE to consumer PE using the interconnect structure of the target architecture.

Our target is to compile parallel applications to a given target architecture with near optimal execution time. After target architecture transformation and intermediate representation of the target application (written in HARPO/L [Norvell 2006]) we have two input graphs, an RRG and an executable DFG. Now our tasks is to map the

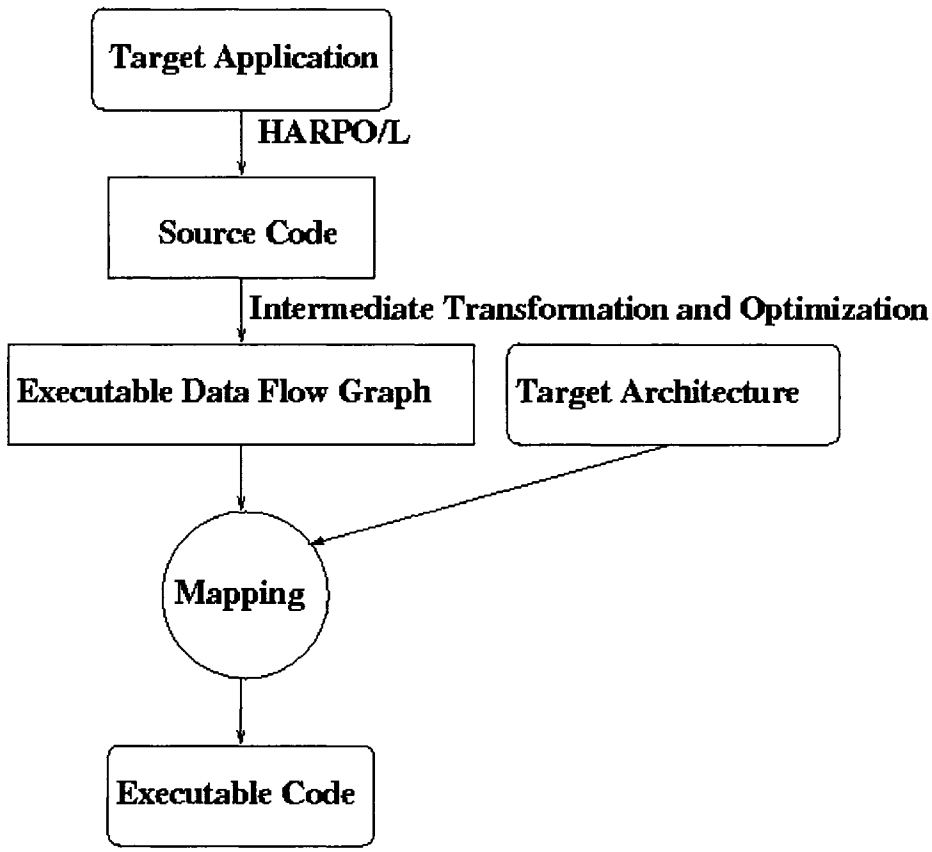

Figure 5.9: Overall framework of our Compilation approach.

DFG onto the RRG as efficiently as possible so that the number of execution clock cycles is as small as possible. We can consider the RRG as the source graph and the DFG as the target graph.

We will now give an overview of our compilation process. Our idea is to first analyze the DFG to extract some information that may be useful in the later phases. We are assuming here that the DFG given as input using static token form for parallel programs [Zhang 2007] has been optimized using various common optimization techniques. The DFG is transformed by removing conditional branches and thus control dependences. For doing this we adopted the if-conversion method using predicates of Park and Schlansker [Park and Schlansker 1991] as is done in [Warter *et al.* 1992]. In If-conversion control dependences are converted to data dependences by computing a condition for executing each operation.

Since the input DFG can be cyclic, we need some approach for partitioning the cyclic and acyclic parts from the DFG. Then we will apply mapping (from DFG to RRG) for both the parts separately and integrate them for mapping as a whole.

For mapping acyclic parts we will use the most commonly used list scheduling algorithm for resource constrained scheduling problems, which will be discussed in the next chapter. In the list scheduling algorithm, instead of using the conventional priority functions, we will use the hypernode reduction modulo scheduling approach [Llosa *et al.* 1995].

Cyclic parts will be mapped using a register-constrained modulo scheduling method, improved modulo scheduling with integrated register spilling (MIRS) algorithm [Zalamea *et al.* 2001a], which will be discussed in the next chapter. MIRS is a simultaneous instruction scheduling and register spilling modulo scheduling method. So

MIRS does scheduling considering the number of registers available at a particular clock cycle. For both the cyclic and acyclic parts, the nodes of the DFG will be mapped to the processing elements of the target architecture and necessary routing is done accordingly. The placement and routing phase will be discussed in chapter 7.

So we can represent our compilation flow as shown in Figure 5.10.

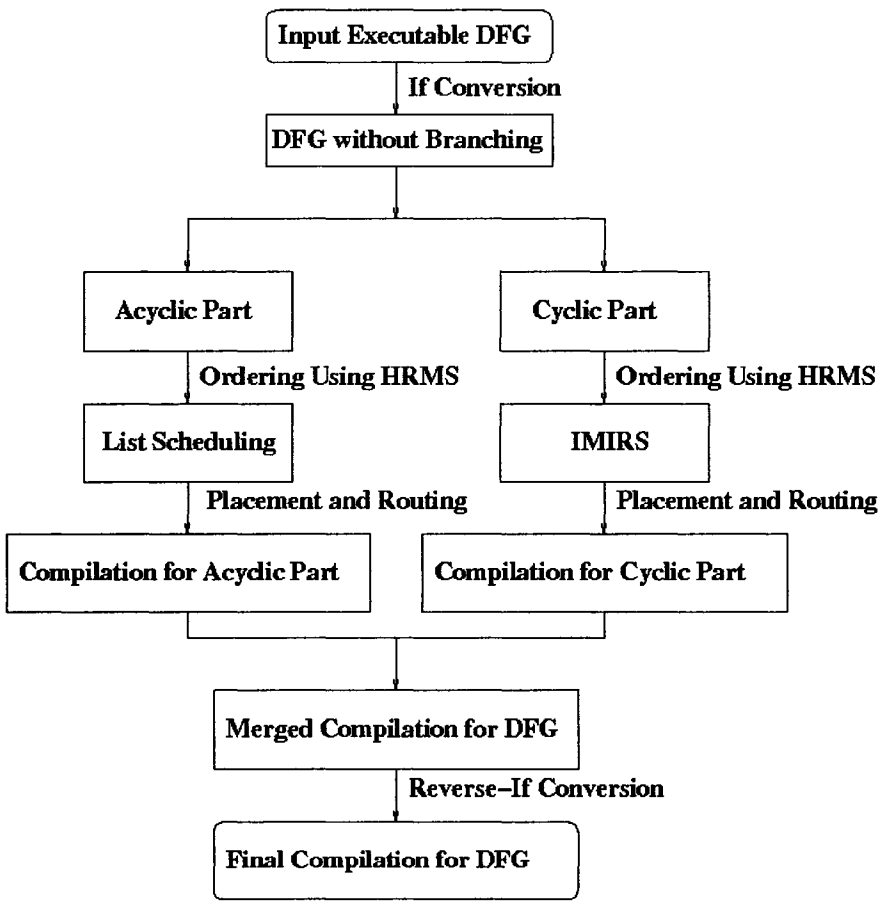

Figure 5.10: Overview of our Compilation flow.

5.4.1 Some Definitions

A subgraph homeomorphism between two directed graphs G_1 and G_2 is a pair of one-to-one mapping (f_1, f_2), where f_1 is from the vertices of G_1 to the vertices of G_2, and f_2 is from the edges of G_1 to simple paths of G_2 [LaPaugh and Rivest 1978]. For each edge (x, y) in G_1 $f_2(x, y)$ should be a path in G_2 from vertex $f_1(x)$ to vertex $f_1(y)$. This homeomorphism is node disjoint, if the image of the edges of G_1 is a set of paths that are node disjoint up to endpoints.

A subdivision of a graph G is a graph constructed by subdividing the edges in G [Wik 2007]. The subdivision of an edge e with endpoints u, v yields a graph containing one new vertex w, with an edge set formed by replacing e by two new edges (u, w) and (w, v). For example, the edge in Figure 5.11(a) with endpoints (u, v) can be subdivided into two edges, e_1 and e_2, connecting to a new vertex w (shown in Figure 5.11(b)). Each node disjoint subgraph homeomorphism from G_1 to G_2 corre-

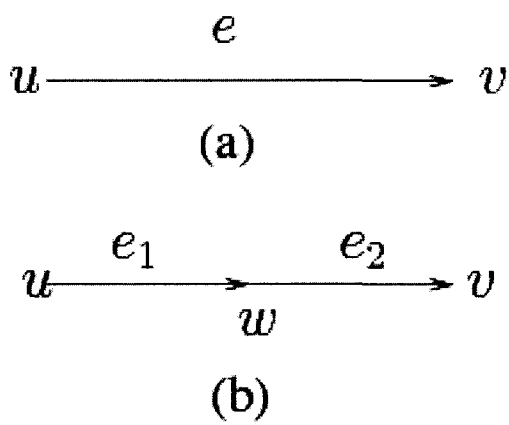

Figure 5.11: Subdivision of Graphs: (a) An edge, (b) Subdivision of the edge.

sponds to an isomorphism between a subdivision of G_1 and a subgraph of G_2.

5.4.2 Compilation Problem Formulation

Assuming each operation has latency of 1 clock cycle, we can formulate the scheduling, placement, and routing problem as one of finding a node disjoint subgraph homeomorphism (f_1, f_2) between the input executable $DFG = (N, E, op, inRole, outRole)$ and the RRG, modeled from the input target architecture, such that:

- $f_1(n) = (k, t)$ Here k is the resource (functional unit) which will execute n's operation. t is the execution time of n. $f_1(n)$ must be capable of executing n's operation, for all nodes $n \in N$. For any two nodes $u, v \in V$, $f_1(u) \neq f_1(v), if u \neq v$.

- $f_2(e) = f_2(n_0, n_1) = P$, such that $start(P) = (k_0, t_0)$ and $end(P) = (k_1, t_1)$. $f_2(e)$ must be capable of carrying e's information, for all edges $e \in E$. For any two edges $e_0, e_1 \in E$, $f_2(e_0)$ is disjoint from $f_2(e_1)$ apart from endpoints, if $e_0 \neq e_1$.

But an operation can have latency greater than 1. So we need to generalize the above formulation. We can formulate our scheduling, placement, and routing problem as one of finding a pair of functions (f_1, f_2) between the input executable DFG and the RRG such that:

- $f_1(n) = \{(k, t), (k, t+1), ..., (k, t+\lambda_n - 1), \}$ Here k is the processing element which will execute n's operation. t is the start time when n will start executing and λ_n is the latency of n's operation. $f_1(n)$ must be capable of executing n's

operation, for all nodes $n \in N$. For any two nodes $u, v \in V$, $f_1(u) \cap f_1(v) = \phi$, if $u \neq v$.

- $f_2(e) = f_2(n_0, n_1) = P$, such that $start(P) = (k_0, t_0 + \lambda_{n_0} - 1)$ and $end(P) = (k_1, t_1)$. $f_2(e)$ must be capable of carrying e's information, for all edges $e \in E$. For any two edges $e_0, e_1 \in E$, $f_2(e_0)$ is disjoint from $f_2(e_1)$ apart from endpoints, if $e_0 \neq e_1$.

Our formulated compilation problem has the following desired properties:

- n must be scheduled to be processed on a unique processing element $f(n) = c \in C \times \mathbb{N}$ starting at a unique time.
- A processing element k can process at most one node's operation at a given time t.
- If a node $n_1 \in N$ is a predecessor of another node $n_2 \in N$, then n_1 must complete its operation's execution before n_2's operation starts.

5.4.3 Partitioning DFG

In general any DFG, as a whole, can be considered acyclic unless there is a cycle between the start node and the end node. A DFG having both acyclic and cyclic parts can be considered as a combination of zero or more of both parts and they may be interleaved. Moreover, cyclic parts may be nested, i.e., there may be one or more level of nesting cyclic parts in a outer cyclic part. Figure 5.12 shows an example of this cyclic-acyclic part of a program. Here we can see that the whole program has more than one of both cyclic and acyclic parts and also a cyclic part contains another

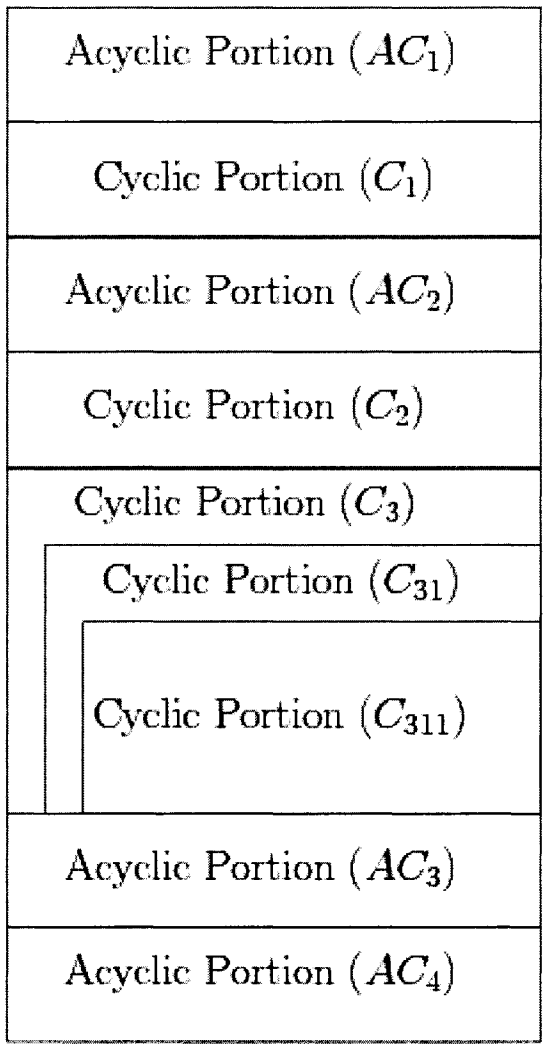

Figure 5.12: Sample of Cyclic and Acyclic Portions in an application.

cyclic part which contains even another. Also a cyclic portion may have two other cyclic portions one after the other inside it.

5.4.4 Mapping Nested Loops

We will apply modulo scheduling to a nested loop structure using a hierarchical fashion similar to the method proposed by [Lam 1988]. We can represent all the loops in a program using a tree. The root of the tree will the outermost loop. The leaves will be the innermost loops. The loops will be scheduled in a bottom-up fashion. When at a particular level more than one loop will be merged to be scheduled, resource requirements of each of them will be considered during scheduling the merged loops.

Chapter 6

Scheduling

The most critical part of compiling parallel applications for CGRAs is scheduling, placement, and routing, which maps software implementations of the applications, especially the compute-intensive loops, onto the target architecture and onto time. Traditional schedulers are not sufficient for this purpose since they do not consider the routability of operand values. They just assign a resource (Functional Unit (FU) or register) and time to every operation in the program. There must be an explicit routing of operand values from producing FU to consuming FU. In VLIWs routing is not needed because the intermediate values are stored in the central register file, and routing is thus implicitly ensured. So scheduling does not imply only binding operations to time slots and resources, but also the explicit routing of operands from producers to consumers.

6.1 Introduction

The scheduling problem is concerned with associating nodes and edges of the DFG to resources and clock cycles such that certain conditions are met. The operations are scheduled on the processing elements of the target coarse-grained reconfigurable architecture. During the scheduling phase, we have to take the constraints and limitations of the architecture resources (memory, registers, ALUs, etc.) into consideration. Also we need to consider the organization of the data flow graph. In this phase the operations are assigned to designated processing components and operations are scheduled as required. This phase determines the relative execution order of the operations and determines which portion of the target architecture will execute which operations. The objectives of scheduling are:

- To generate a valid schedule so that each operation has a resource to execute it and a time cycle, in which that resource and corresponding routing resources are not otherwise occupied.
- To minimize the execution time of the whole application by using the available resources efficiently.

The following properties are maintained during scheduling:

- If an operation (say o_1) is dependent on another operation (say o_2), then o_2 must be scheduled in a time cycle prior to that of o_1.
- Operations that are independent can be scheduled to be executed in parallel.
- Operations without any dependency can be scheduled as early as possible providing resource availability.

The output of the scheduling phase will be the pair of mappings f_1, f_2 described in chapter 5. The first assigns a resource and a block of times to each DFG operation. The second assigns a route to each DFG edge.

In this chapter first the scheduling for cyclic part (loop) will be shown. We start with a motivating example to illustrate scheduling (section 6.2). Then some definitions and concepts are introduced that will be used in the subsequent algorithms (subsection 6.3.1). Then all the phases of the scheduling algorithm for the cyclic part will be described (subsections 6.3.3 to 6.3.8). Next we present the complete algorithm (subsection 6.3.9). Finally we present the scheduling algorithm for the acyclic part of an application (section 6.4).

6.2 Motivating Example

We will first present the modulo scheduling approach for cyclic parts using a motivating example. Figure 6.1(a) shows a sample input data flow graph representing a loop body. The DFG consists of five different nodes. The operations performed by each node, the latency of each operation, and the earliest time in which a node can be scheduled (ASAP) is shown in Figure 6.1(b). We schedule the nodes following the order of increasing ASAP values. This ordering is actually the same as the HRMS ordering (to be discussed in subsection 6.3.4) for this example DFG. The schedule for an iteration is divided into several stages. The schedule will be such that the execution of consecutive iterations overlaps and each iteration will be in a different stage. The number of stages in one iteration is named the stage count (SC), and the number of time cycles per stage is termed the initiation interval (II). Figure 6.1(c)

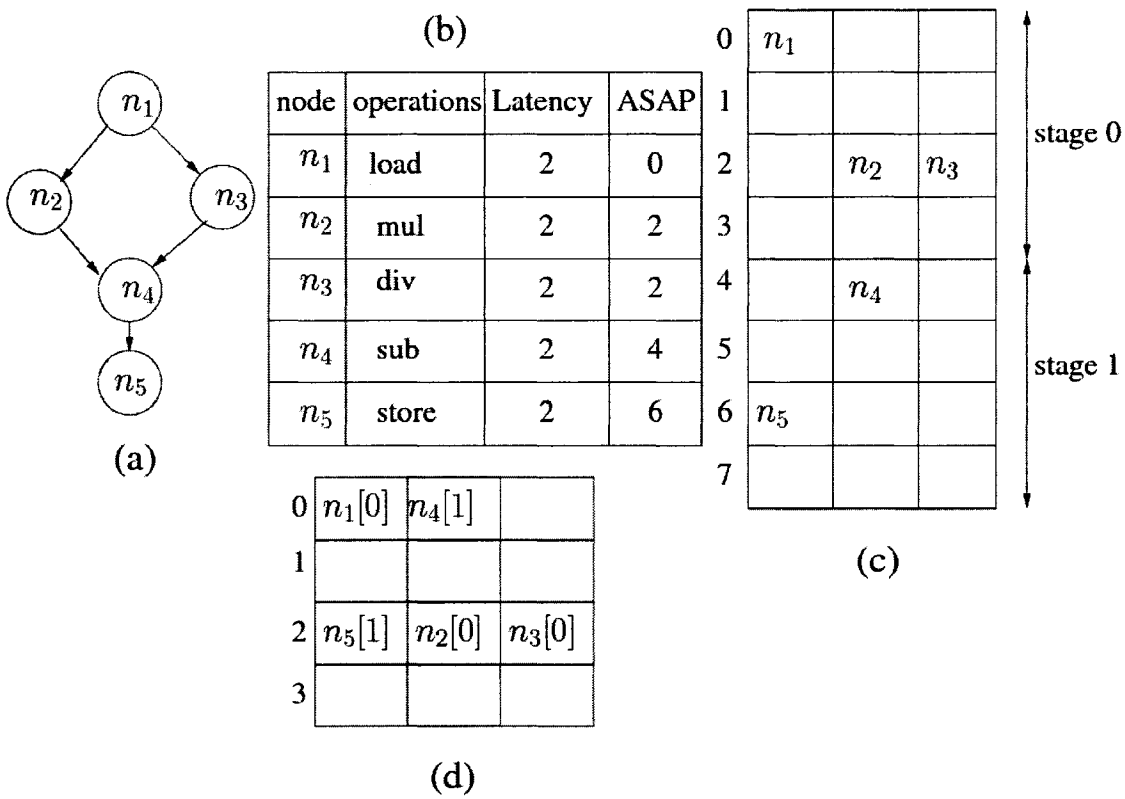

Figure 6.1: A motivating example for the Modulo Scheduling approach. (a) A simple data flow graph. (b) A table showing the properties of each node of the DFG. (c) Scheduling for an iteration. (d) Modulo Scheduling of the kernel.

shows the scheduling for one iteration of the DFG assuming the value of II as 4. The length of the schedule is 8. We can see from Figure 6.1(c) that the stage count of the schedule, which is the ratio of the schedule length to the initiation interval, is 2. Figure 6.1(d) shows the modulo scheduling of the kernel of the loop. Here the number inside the bracket indicates stage number. Figure 6.2 shows four iterations of the loop. The determination of stages in Figure 6.1(c) is obvious from Figure 6.2.

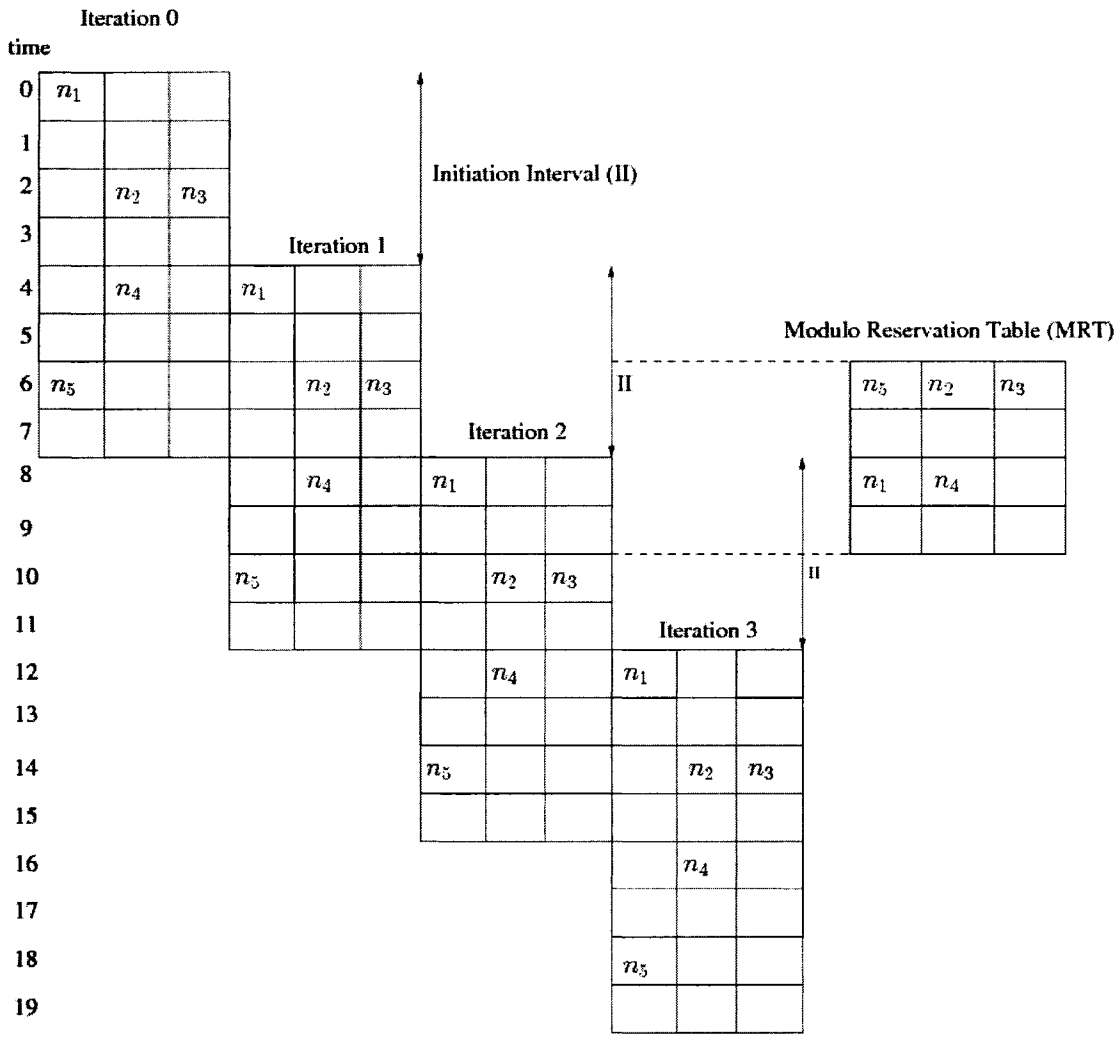

Figure 6.2: Four iterations of the loop.

Modulo Scheduling algorithms generally use a modulo reservation table. A modulo reservation table has columns equal to the number of resources, and rows equal to the cycles per stage (II). Placing an instruction in an entry of the table (suppose $< c, r >$) indicates that the instruction will initiate on that resource (r) in that cycle (c). Figure 6.2 shows the state of the modulo reservation table during the steady state. Actually this table reflects the usage of resources of the target architecture in each time slot of the steady state of the modulo scheduling algorithm.

6.3 Scheduling for Cyclic Parts

In this section we will discuss how to schedule, map and route cyclic parts, especially loops of an application. We will adapt a register constrained modulo scheduling technique, Modulo Scheduling with Integrated Register Spilling (MIRS) [Zalamea *et al.* 2001a]. MIRS is a software scheduling method that is capable of instruction scheduling with reduced register requirements, register allocation and register spilling in a single phase, unlike many other modulo scheduling techniques that schedule instructions and allocate registers in two phases. But MIRS alone cannot do the required compilation for our problem. The reason is that MIRS does only scheduling and placement, it does not consider routing. As with FPGA placement and routing, we need to do routing during placement. The reason is as follows. During placement a cost function is computed to evaluate the quality of placement. While calculating that cost function, we need to incorporate routing cost. So we have modified the MIRS algorithm to incorporate this feature, that is, to consider routing and its associated cost. MIRS has been applied to clustered VLIW architectures [Zalamea *et al.* 2001b]. Because of

the presence of central register file, routing is not necessary in VLIW.

Another factor that we have incorporated into the MIRS algorithm is the consideration of loops with conditional branches. For doing this we have adapted the if-conversion and reverse-if-conversion idea from [Park and Schlansker 1991] that was used in Enhanced Modulo Scheduling [Warter *et al.* 1992].

The input of the algorithm will be an executable data flow graph (DFG) representing the cyclic part (loop body) and the routing resource graph (RRG) (introduced in chapter 5) representing the target architecture replicated across time. There are two outputs of the algorithm. One is the initiation interval (II) (defined in chapter 2). Another is a schedule of the nodes of the DFG, which is the pair of functions f_1 and f_2 described in chapter 5. This schedule will enable each node of the DFG to execute at its time cycle in its resource.

The steps followed in the improved modulo scheduling with integrated register spilling algorithm are shown in Figure 6.3 for a quick overview. Table 6.1 explains the variables used in the algorithm. In the next subsections we will introduce some definitions and concepts useful for the scheduling of cyclic parts (subsection 6.3.1). Then all the phases of the scheduling algorithm for cyclic parts will be described with suitable examples (subsections 6.3.3 to 6.3.8). Finally we present the complete algorithm (subsection 6.3.9).

6.3.1 Some Definitions and Concepts

In this section we define some terms that will be used for our algorithm (improved from MIRS).

Variables	Meaning
II	Initiation Interval initialized to MII
f_1	A Partial function that maps each node of the DFG scheduled so far to a set of a pair of values: a time cycle and a resource
f_2	A partial function that maps each edge of the DFG to a path in the RRG
Priority_List	List of nodes of the DFG ordered by the HRMS method
Budget	Number of times the algorithm will try to schedule with the current II. It is initialized to Budget_Ratio times the number of nodes of the DFG
u	Next node from the Priority_List, which will be scheduled
Start	A time cycle for a node starting from which a node can be scheduled
End	A time cycle for a node up to which a node can be scheduled

Table 6.1: Variables used in the IMIRS Algorithm.

1. Initialize II with MII and the partial schedule (f_1, f_2) to be empty.
2. Order the nodes of the DFG using HRMS method and initialize the *Priority_List* with these ordered nodes.
3. Initialize *Budget* to *Budget_Ratio* times the number of nodes of the DFG.
4. Select and remove a node u with the highest priority from the *Priority_List*
5. Place u in a unoccupied processing element i such that a valid route exists from its previously placed predecessors or successors.
6. Determine a time cycle for node u.
7. If a valid cycle cannot be found, force u in a particular time cycle and eject some nodes if necessary.
8. Check the numbers of available and required registers and spill values to memory if required. Update the DFG and *Priority_List* accordingly.
9. If the Budget is 0 or the memory cannot support memory traffic:
    - Initialize the partial schedule (f_1, f_2) to be empty.
    - Initialize the *Priority_List* with the ordered nodes of the DFG.
    - Go to step 4 with II := II + 1.
10. If all the nodes are scheduled, placed, and routed then go to step 11, otherwise, decrease Budget by 1 and go to step 4.
11. Allocate registers and generate the CGRA configuration with f_1, f_2 and II.

Figure 6.3: Phases of the IMIRS algorithm.

- **Dependent Distance:** $\delta_{(u,v)}$ is a value p between nodes v and u, if the execution of v in iteration i depends on the execution of u in iteration $(i-p)$. If there is no dependency between u and v, then $\delta_{(u,v)}$ will be 0. If u and v are in the same iteration then the value of $\delta_{(u,v)}$ will be 0 (intra-iteration dependency). If $p > 0$ then it is called a loop-carried dependency.

- **Latency:** λ_u is the delay associated with each operation u of the DFG. An operation takes this number of time cycle to produce a result.

  Figure 6.4 illustrates dependent distance and latency, Figure 6.4(a) shows a simple loop. We assume that *load, mul* and *store* have latencies of 1, 4, and 1 respectively. Figure 6.4(b) shows two iterations of the loop in a graph. Each edge of the graph is labeled with the latency of the operation located at the tail of that edge. Since each iteration uses the output of the previous iteration as its input, there is a labeled edge between the corresponding operations. Figure 6.4(c) shows another graph for the loop with each edge being labeled with a $< \delta_{(u,v)}, \lambda_u >$ pair. We can see from this figure that any operation that depends on another operation of the same iteration has a dependent distance of 0 (intra-iteration dependency). An edge from an operation of the immediate previous iteration to an operation of the current iteration has a dependent distance of 1 (loop-carried dependency). Figure 6.4(d) shows another simple loop with dependent distance of 2 as shown in the Figure 6.4(e)-(f).

- t_v is the time-cycle in which a node v in f_1 has been scheduled.

- $PSP(u)$: It is the set of predecessors of u that have been scheduled. A node v is a predecessor of u if u is reachable from v. The set of predecessors of a node u

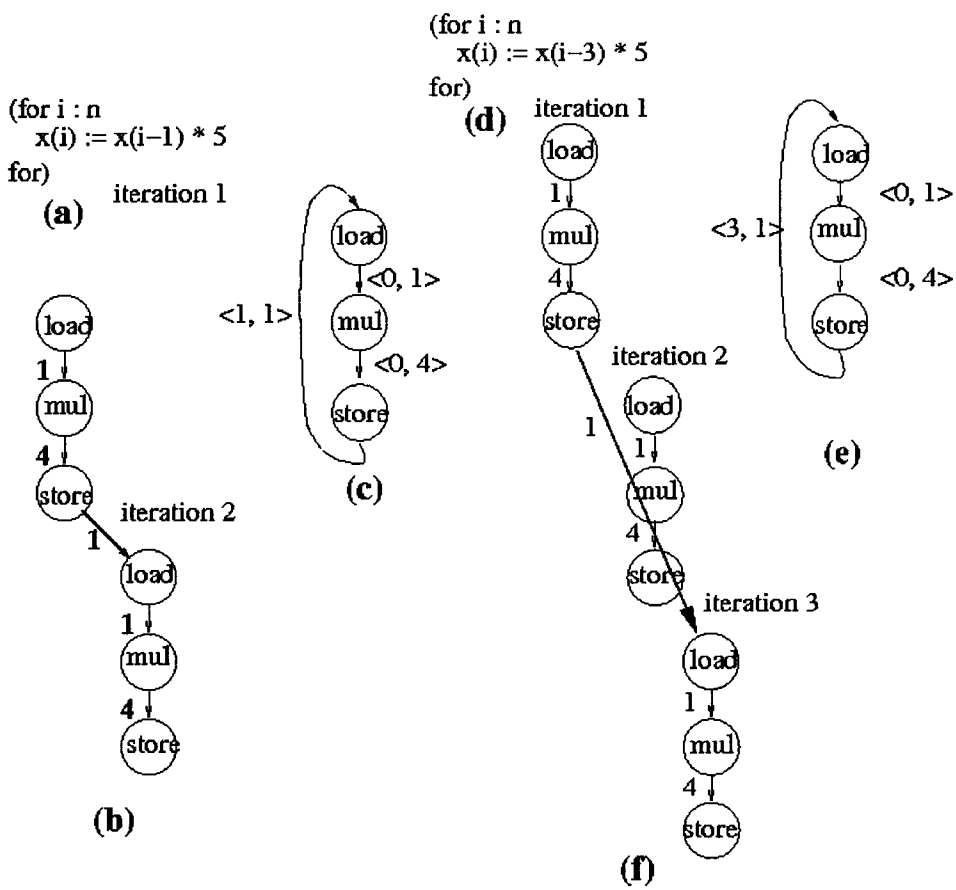

Figure 6.4: Example illustrating dependent distance and latency.

is represented by Pred(u) such that Pred(u) = $\{v|v \in V$ and $(v,u) \in E\}$. Here V is the vertex set and E is the edge set. So $PSP(u) = \text{domain}(f_1) \cap Pred(u)$.

- $PSS(u)$: It is the set of successors of u that have been scheduled. A node v is a successor of u if v is reachable from u. The set of successors of a node u is represented by Succ(u) such that Succ(u) = $\{v|v \in V$ and $(u,v) \in E\}$. So $PSS(u) = \text{domain}(f_1) \cap Succ(u)$.

- **Early Start:** The Early Start for a node u is the earliest time at which it can be scheduled so that all the scheduled predecessors of u have completed their execution [Rau 1994]. It is calculated as follows:

$$Early\_Start_u = max_{v \in PSP(u)}(t_v + \lambda_v - \delta_{(v,u)} \times II)$$

- **Late Start:** The Late Start for a node u is the latest time at which it can be scheduled so that it can complete its execution before all the scheduled successors of u have started their execution [Llosa *et al.* 1995]. It is calculated as follows:

$$Late\_Start_u = min_{v \in PSS(u)}(t_v - \lambda_u + \delta_{(u,v)} \times II)$$

- **ASAP**: ASAP for a node u is the earliest time in which it can be scheduled. It is calculated as follows:

$$ASAP_u = max_{v \in PSP(u)}(ASAP_v + \lambda_v - \delta_{(v,u)} \times MII)$$

- **Life Time:** The life time (LT) of a variable r is from the beginning of the definition of r (producer FU) to the beginning of the last use of r (last consumer

FU). When considering a cycle, i.e., loops, variables may be of two kinds: loop-variant and loop-invariant. Each loop-invariant variable has a single value for all iterations of a loop. Loop-variant variables have separate lifetimes for each iteration. They are called loop-variant because a variable may be initialized to different values at each iteration of the loop.

- **MaxLive**: MaxLive is the maximum number of simultaneously live variables. If we consider the lifetime of each variable of the DFG then the number of variables spanning in a particular clock cycle can be termed as the live values of that clock cycle. It is a relatively accurate approximation of the number of registers that are required for the schedule of a loop

- **Critical Cycle**: The Critical Cycle is the scheduling cycle in which the number of live values is equal to MaxLive.

6.3.2 Necessity of considering register usage

Registers act as intermediate locations for values when compiling applications onto coarse-grained reconfigurable architectures. During compilation, a software pipelining technique in the form of iterative modulo scheduling is applied to the cyclic parts of an application, such as loops. Overlapping of loop iterations imposes high register requirements. While finding the optimal schedule is the main objective, scheduling must also be aware of the potential increase of register pressure by reordering operations. Register pressure is the maximum number of live values at any point in the program. A live range is the range

from when a value is defined to its final use in the program. A value is live at a given point if the last use of that value has not occurred. Because CGRAs have limited registers, the number of live values at a particular point should not exceed the total number of registers available in the CGRA. A schedule that uses at most the number of registers available in a CGRA is considered valid. When a schedule requires more registers than available, the register pressure must be reduced for the execution of the loop. Possible ways to do this are to reschedule the loop with a larger II, to spill some values to memory, or to split the loop into several smaller loops (each one with fewer operations than the original one).

Register pressure is, in a sense, proportional to the number of simultaneously executed iterations. An increase in II decreases the stage count (SC); thus the number of concurrently executed overlapping instructions is decreased. Thus an increase in II reduces register pressure at the expense of execution time.

In register spilling, some values are spilled from the registers to memory and loaded again when needed, although these loads and stores will incur some cost. The scheduler first determines the spilling candidates and orders them. Then the necessary number of nodes are selected and necessary loads and/or stores are added. Selection is based on priorities of the spilling candidates. Either the life time of the candidate or the ratio of life time to the memory traffic caused due to spilling is used. The second heuristic produces better spilling candidates. In register spilling, the loop is scheduled with the same II, unlike the previous method.

6.3.3 Calculation of Minimum Initiation Interval (MII)

The minimum Initiation Interval (MII) is a lower bound on the minimum number of cycles required between initiations of successive iterations of a loop. The Initiation Interval (II) of the modulo scheduling algorithm is constrained by the most heavily utilized resource and the worst-case recurrence for the loop. The minimum II (MII) is the maximum of the lower bounds for both of these constraints. MII is thus a lower bound on the II. If there are not enough resources available, instructions will be delayed from issuing until the required resources are free. If there are dependence constraints, instructions cannot complete until all their operand values are available. IMIRS uses the MII as the starting value for II when generating a schedule, which is the lowest value that can be achieved given the resource and recurrence constraints.

6.3.3.1 Calculating ResMII

ResMII is computed by taking the summation of the resource usage required for one iteration of the loop. This resource usage patterns for each cycle during one iteration of a loop is represented by a reservation table [Rau 1994]. There are II rows and one column per resource in this table. The exact ResMII is obtained by performing a bin-packing of the reservation table for all instructions. This method often leads to exponential complexity as bin-packing is an NP-hard problem [Rau 1994]. So ResMII is typically approximated in modulo scheduling algorithms.

ResMII is approximated from the most heavily used resource count along any

execution path and total usage count for each resource. If an execution path p uses a resource r for C_{pr} cycles and there are n_r copies of this resource, then ResMII is calculated as follows:

$$ResMII = max_{p \in P}(max_{r \in R} \left\lceil \frac{C_{pr}}{n_r} \right\rceil),$$

where P is the set of all execution paths and R is the set of all resources.

6.3.3.2 Calculating RecMII

Inter-iteration dependences can induce recurrences that causes a maximum delay for the operations on the recurrence path or dependence cycle. Memory operations (load/store) are mostly the cause of a recurrence. These loop-carried dependences have a distance property, which is equal to the number of iterations separating the two instructions involved. If a dependence edge, e, in a cycle has latency l_e and connects operations that are at a distance of d_e iterations, then $RecMII$, the minimum II for recurrence constraints is calculated as follows:

$$RecMII = max_{c \in C} \left\lceil \frac{\sum_{e \in E_c} l_e}{\sum_{e \in E_c} d_e} \right\rceil,$$

where C is the set of all dependence cycles and E_c is the set of edges in dependence cycle c.

After calculating $ResMII$ and $RecMII$, MII is calculated as the maximum of these two values. For example, Figure 6.5 shows a DFG. We assume all the operations require 2 cycles, except store, which requires 1 cycle. Also the target architecture has two load/store units, a multiplication unit and an add unit. Since the DFG has no recurrence circuits, its MII is constrained only by

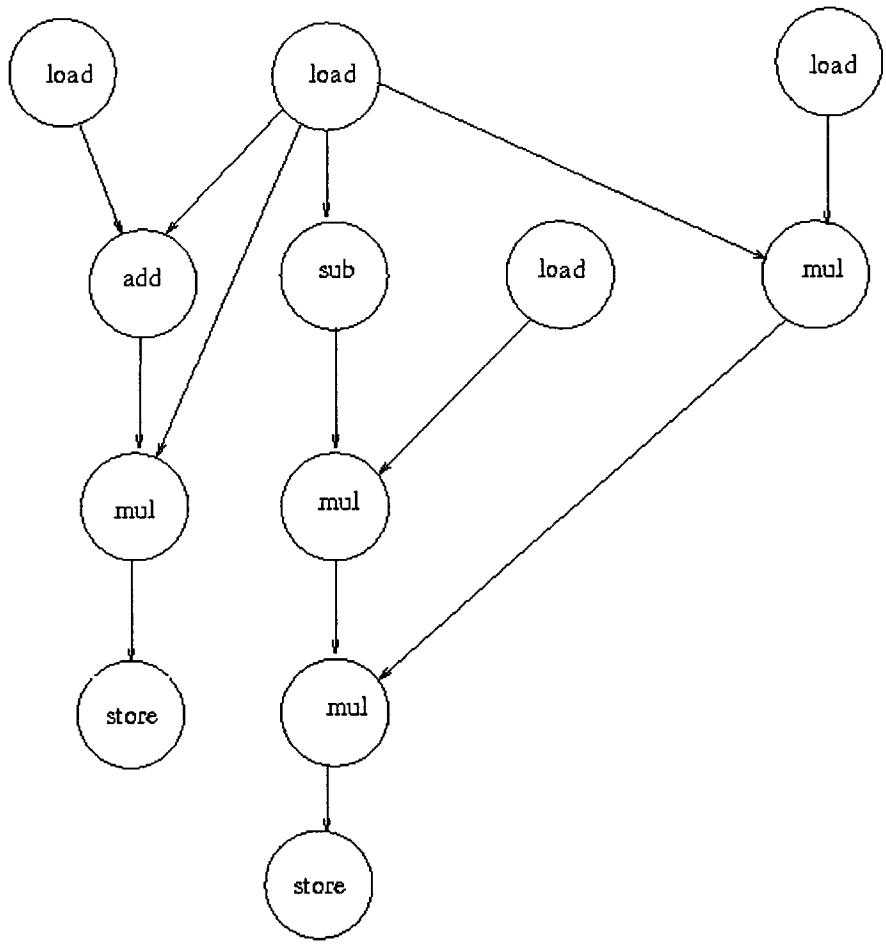

Figure 6.5: A simple DFG for illustrating the calculation of MII.

the available resources in the target architecture. In this case multiplication is the most heavily used resource class, which limits the $ResMII$. The value of $ResMII$ is $\frac{4}{1} = 4$. Therefore, $MII = max(ResMII, RecMII) = max(4, 0) = 4$.

6.3.4 Ordering Using HRMS Method

All the nodes of the DFG are ordered following a bidirectional approach, Hypernode Reduction Modulo Scheduling (HRMS), described in [Llosa *et al.* 1995]. The nodes are scheduled one by one following this order. The objective of this ordering is to obtain a schedule that uses minimum II and minimum number of registers. For that operations on the critical path are given priority so that the cycle is executed in as few clock cycles as possible.

The ordering phase guarantees that when a node is scheduled, the node will have only its predecessors or its successors in the partial schedule, but not both. The only exception is when the node to be scheduled is the first or the last node of the cycle. This ordering is done only once, even if II increases later on.

During the scheduling phase, the nodes are scheduled as soon as possible (as late as possible) if predecessors (successors) have been scheduled already. The idea of HRMS is, first, to decrease the lifetime (LT) of loop variant variables by ordering the nodes so that each operation to be scheduled has a previously ordered reference point (predecessors or successors) and second, to decrease the schedule length. The ordering phase is an iterative algorithm. At first a node is selected as the hypernode. A hypernode is a single node that represents a

single node or a subgraph of the DFG. In each iteration, the neighbors of the hypernode are ordered, and the neighbors, along with the new hypernode, are reduced to a new hypernode.

For a given DFG without cycles, the first node or any other node may be chosen as the initial hypernode. The predecessors and the successors of a hypernode are alternatively ordered following the steps given below:

1. The nodes on all paths between the predecessors/successors are collected.

2. The predecessor/successor nodes of step 1 and the current hypernode are transformed into a new hypernode.

3. A topological sort is performed on the subgraph represented by the nodes obtained in step 1. The resulting sorted node list is appended to the final ordered list.

4. Step 1-3 are repeated until the DFG is reduced to a single hypernode.

In HRMS strategy for a given DFG with cycles, the cycles are processed first before considering the nodes not in the cycles. In this case no single node is chosen as the initial hypernode. The objective is to order all the nodes of all the cycles and reduce them to a single hypernode. After that the resulting DFG is acyclic with a hypernode. So the ordering for the rest of the nodes is like the ordering for DFG without cycles. All the cycles are first sorted according to their RecMII, with the highest RecMII having highest priority. This results in a list of sets of nodes, where each set is a cycle. If two cycles share a back

edge, the corresponding node sets are combined into the one with the highest priority. If more than one set has a node, that node is included into the cycle with the highest RecMII, and excluded from the rests. Unlike DFG without cycles where predecessors and successors are ordered alternatively, the ordering phase for DFG with cycles follow the steps given below starting with the first cycle of the list:

1. All the nodes from the current cycle to the next in the list is obtained. All the back edges are removed for avoiding cycles in this process.

2. The current cycle, the nodes obtained from step 1 and the current hypernode (if one exists) are reduced to a new hypernode.

3. A topological sort is performed on the subgraph represented by the current cycle and the nodes obtained from step 1. The resulting sorted node list is appended to the final ordered list.

4. Step 1-3 are repeated until the DFG is reduced to a single hypernode.

An example is shown in figures 6.6 and 6.7, illustrating the HRMS method for DFG without cycles. Figure 6.6(a) shows the input DFG. Figure 6.6(b) shows the state of the DFG after one iteration. Only the node v_1 is consumed as the hypernode and v_1 is the first node of the priority list to be scheduled. Next the predecessors/successors are ordered. Since v_1 has no predecessor, only its successors are ordered using a topological sort and appended to the priority list. Figure 6.7 shows the remaining steps of the HRMS ordering. The final output

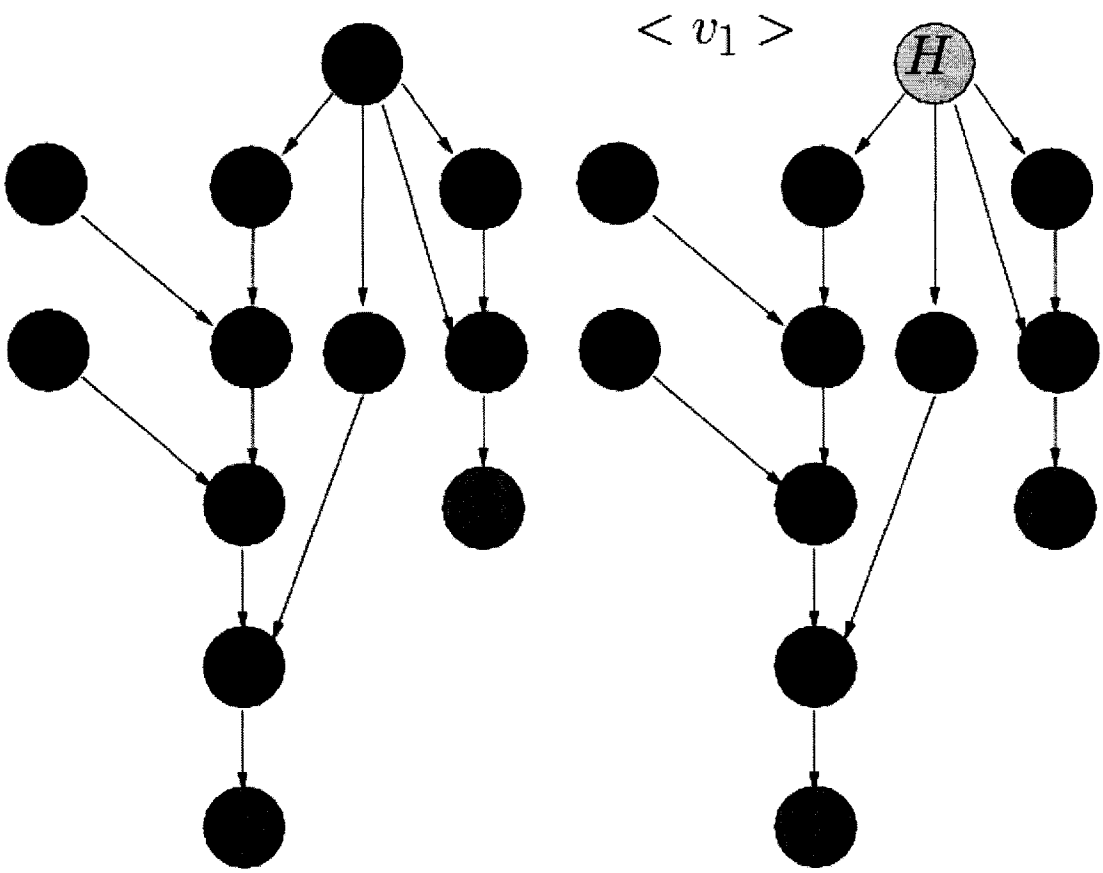

Figure 6.6: Illustration of HRMS method for DFG without cycles. (a) Input DFG. (b) Ordered list and DFG after iteration 1.

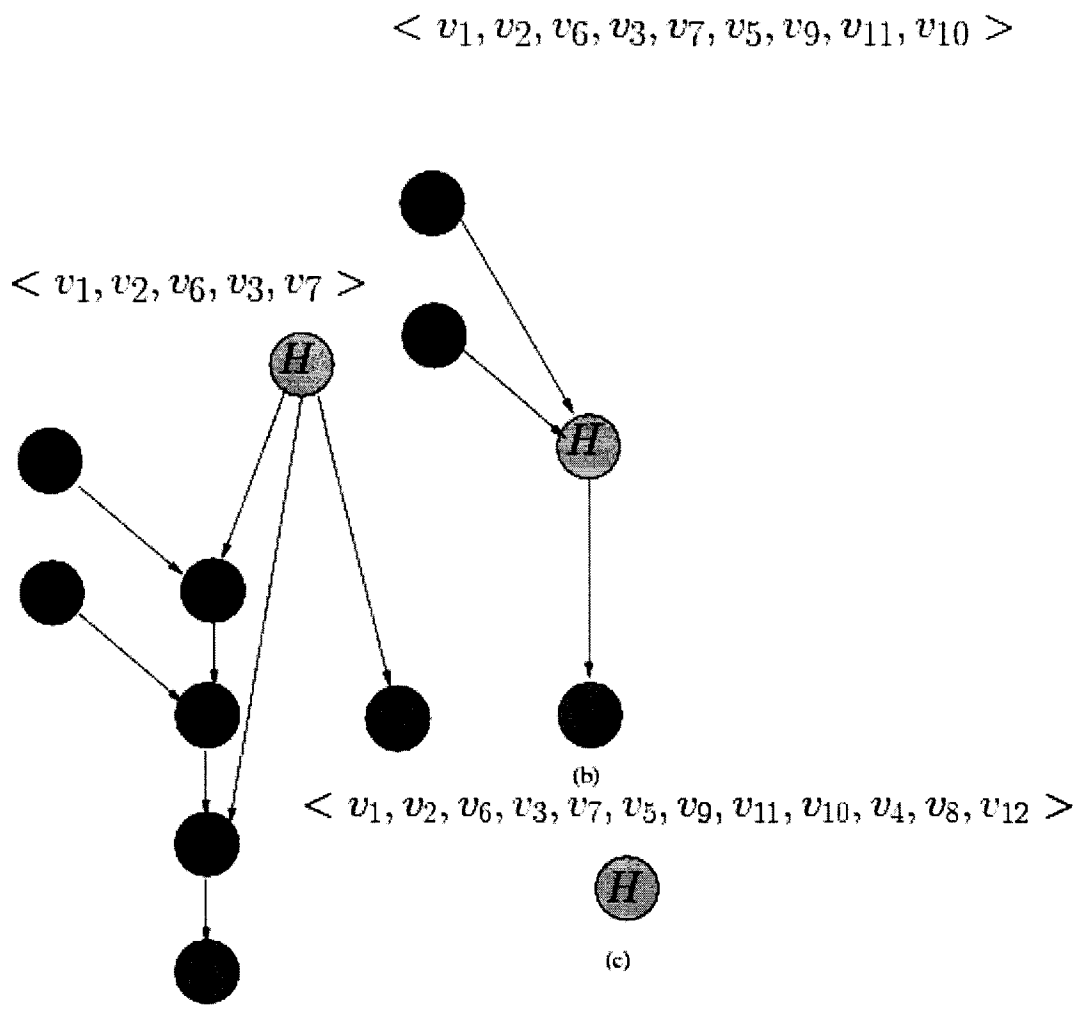

Figure 6.7: Illustration of HRMS method for DFG without cycles. (a) Ordered list and DFG after iteration 2. (b) Ordered list and DFG after iteration 3.

of the ordered list is $< v_1, v_2, v_6, v_3, v_7, v_5, v_9, v_{11}, v_{10}, v_4, v_8, v_{12} >$. The nodes of the input DFG will be scheduled, mapped, and routed using this order.

6.3.5 Schedule_Place_Route

Figure 6.8 shows the actual scheduling, placement, and routing step of the IMIRS algorithm. In this phase a node is scheduled starting from a particular clock cycle in one or more resource(s) (functional unit(s) and/or register(s)). It first calculates the $Early\_Start_u$ and $Late\_Start_u$ of the node u to be scheduled, which produces a time frame in which that node can be scheduled legally. Suppose the *Start* and *End* defines this time frame. For scheduling node u, Mapping() (to be discussed in chapter 7) determines one or more resources in the RRG within this time frame starting from Start that produces optimal cost. During this checks are done so that there are valid routes from/to the predecessors/successors of u to u. Checks are also done so that there is no violation of dependence or no resource conflict. If such free resources are found, u is scheduled to the time cycles indicated from the position of the resource(s) in the RRG. Necessary updates are made to the partial schedule, resources, and registers. However, if no valid cycle is found, then the Force_and_Eject heuristic is applied.

The partial schedule is scanned forwards or backwards depending on the values of *Early_Start*, *Late_Start*, II, and whether predecessors or successors of the node to be scheduled are already placed in the partial schedule. The partial schedule is explored according to the following rules [Llosa *et al.* 1995]:

```
Procedure Schedule_Place_Route(DFG, RRG, f1, f2, u) {
        var Start, End;
        if (Pred(u) is in Partial Schedule) {
            Start = Early_Start_u;
            End = Early_Start_u + II − 1;
        }
        else if (Succ(u) is in Partial Schedule) {
            Start = Late_Start_u;
            End = Late_Start_u − II + 1;
        }
        else if (both Pred(u) and Succ(u) are in Partial Schedule) {
            Start = Early_Start_u;
            End = min(Late_Start_u, Early_Start_u + II − 1);
        }
        else {
            Start = ASAP_u;
            End = ASAP_u + II − 1;
        }
         if (not Mapping(DFG, RRG, f1, f2, u, Start, End))
            Force_And_Eject(i, u);
}
```

Figure 6.8: Scheduling Phase of the IMIRS algorithm.

1. If a node u of the DFG has only predecessors in the partial schedule f_1, then HRMS strategy maps u to the resources of RRG as soon as possible to reduce the lifetime of any associated loop-variant variable. For doing that the cycle search range for node u is starting from $Early\_Start_u$ to $Early\_Start_u + II - 1$. If a legal cycle is not found in this range due to resource conflicts, it is impossible to find one outside this range. So it is not necessary to consider more than II contiguous cycles starting from $Early\_Start$.

2. If a node u of the DFG has only successors in the partial schedule f_1, then HRMS strategy maps u to the resources of RRG as late as possible to reduce the lifetime of any associated loop-variant variable. For doing that the cycle search range for node u is starting from $Late\_Start_u$ to $Late\_Start_u - II + 1$.

3. If node u has both predecessor(s) and successors(s) in the partial schedule f_1, then a potential cycle is searched for starting at $Early\_Start_u$ to min($Late\_Start_u$, $Early\_Start_u$ + II - 1).

4. If node u has no predecessor or successors in the partial schedule f_1, then potential cycle is searched for starting at $ASAP_u$ to ($ASAP_u$ + II - 1).

6.3.6 Force_And_Eject Heuristic

While scheduling a node in a particular cycle, if a resource conflict occurs, then the scheduler makes two decisions. First, a cycle is chosen to force the schedule of that node, and one or more nodes are chosen to be ejected from

the current partial schedule. The nodes to be ejected is the node or set of nodes which caused the node to be scheduled to violate a dependence relation or make resource conflicts. The cycle in which the node in question is forced is calculated such that there is a forward progress in the schedule. If *Early_Start* is less than the cycle in the last previous partial schedule, then the node is forced at *Early_Start*. Otherwise, it is forced one cycle later than it was previously scheduled. If the node is forced in the cycle *Forced_Cycle* then it is calculated as follows [Zalamea *et al.* 2001a]:

$$Forced\_Cycle = max(Early\_Start, (Prev\_Cycle(i) + 1)),$$

if the search is made from *Early_Start* to *Late_Start*. If the search is made in the opposite direction (from *Late_Start* to *Early_Start*) then

$$Forced\_Cycle = min(Late\_Start, (Prev\_Cycle(i) - 1)).$$

In each case $Prev\_Cycle(i)$ is the cycle in which the node in question was scheduled in the last previous partial schedule (before a possible ejection).

When ejecting, the algorithm ejects only one node, unlike [Rau 1994] which ejects all the nodes causing resource conflicts with the forced node. According to [Zalamea *et al.* 2001a], the node to be ejected is the one that was placed first in the partial schedule. Besides, for forcing the node in a particular clock cycle, there might be some more resource conflicts. The Force_And_Eject heuristics ejects all the successors and predecessors that violate the dependence constraint for the placement of that forced node. The ejected nodes are all inserted into the Priority_List to be considered for rescheduling, perhaps at the next round of scheduling.

6.3.7 Check_and_Insert_Spill Heuristic

This heuristic is responsible for inserting spill codes. It first determines whether there is any need to do that. If the number of available registers is enough for the current partial schedule, this heuristic does nothing and scheduling continues with the next node of the Priority_List. Zalamea *et al.* [Zalamea *et al.* 2001a] inserts spill code whenever $RR > SG \times AR$, where RR is the required number of registers in the current schedule, AR is the number of registers available and SG, $SG \geq 1$, is a parameter, named spill_gauge.

To illustrate how spilling is done, let us assume a variable v has a single definition and five uses at five different program points as shown in Figure 6.9. The definition of a variable is called the producer of a value, while the uses of that value are its consumers. The life time (LT) of v is from the beginning of the definition of v (producer FU) to the beginning of the last use of v (last consumer FU) as shown in the figure. Variables are spilled by inserting a store operation after the definition of the variable and a load operation before every use, as in Figure 6.10. For a particular variable, the lifetime from the time the variable is available (the variable definition time cycle + the latency of the variable) to the first use of that variable is termed as $use1$. For other use points, the lifetime from one use of the variable to the next use of the same variable is termed as *use* for the second use. For example, the lifetime from the first use to the second use is termed as $use2$. Uses are spilled by inserting a store operation after the definition of the variable and a single load operation before the corresponding use as shown in Figure 6.11. Variable spilling and uses spilling are the same

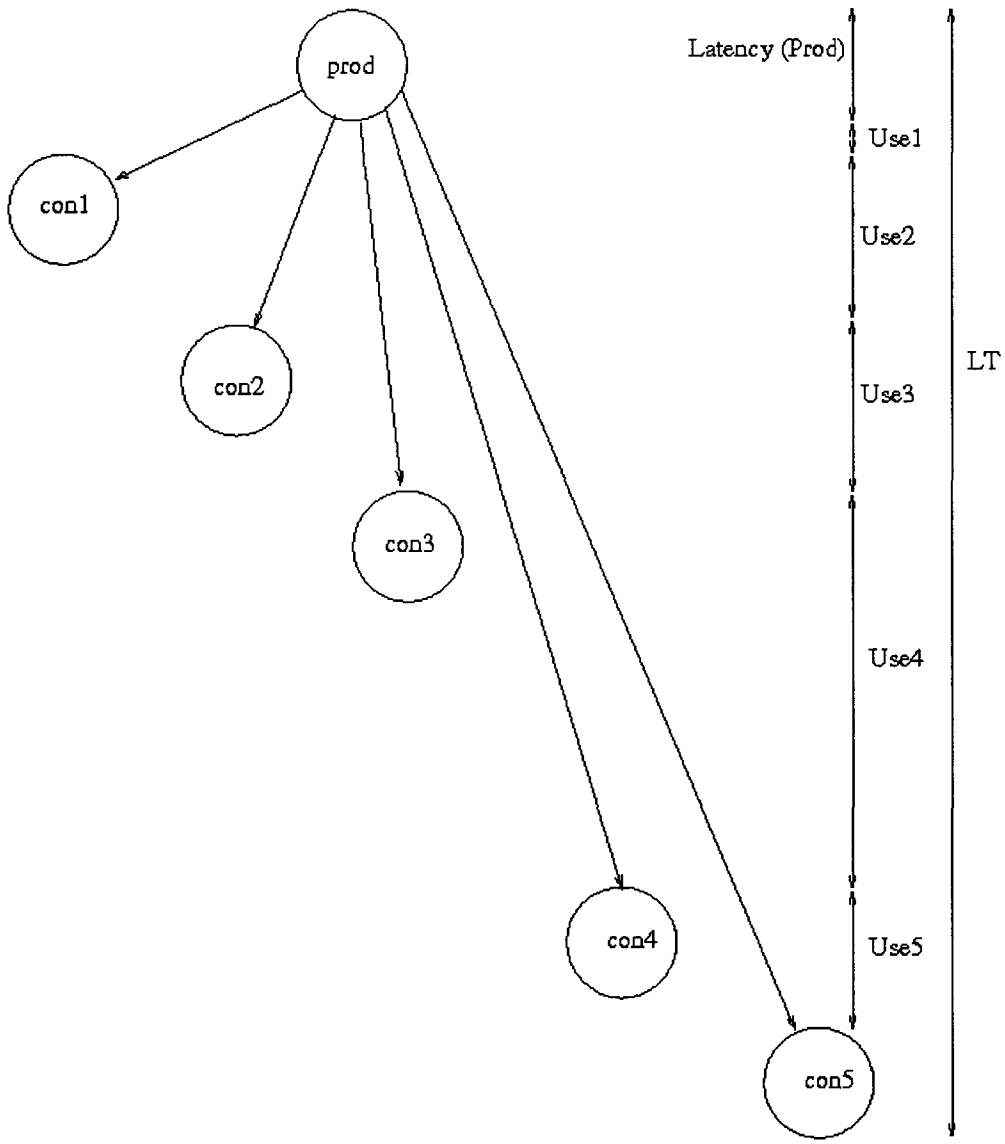

Figure 6.9: A sample DFG illustrating the lifetime and producer-consumer relations of a variable.

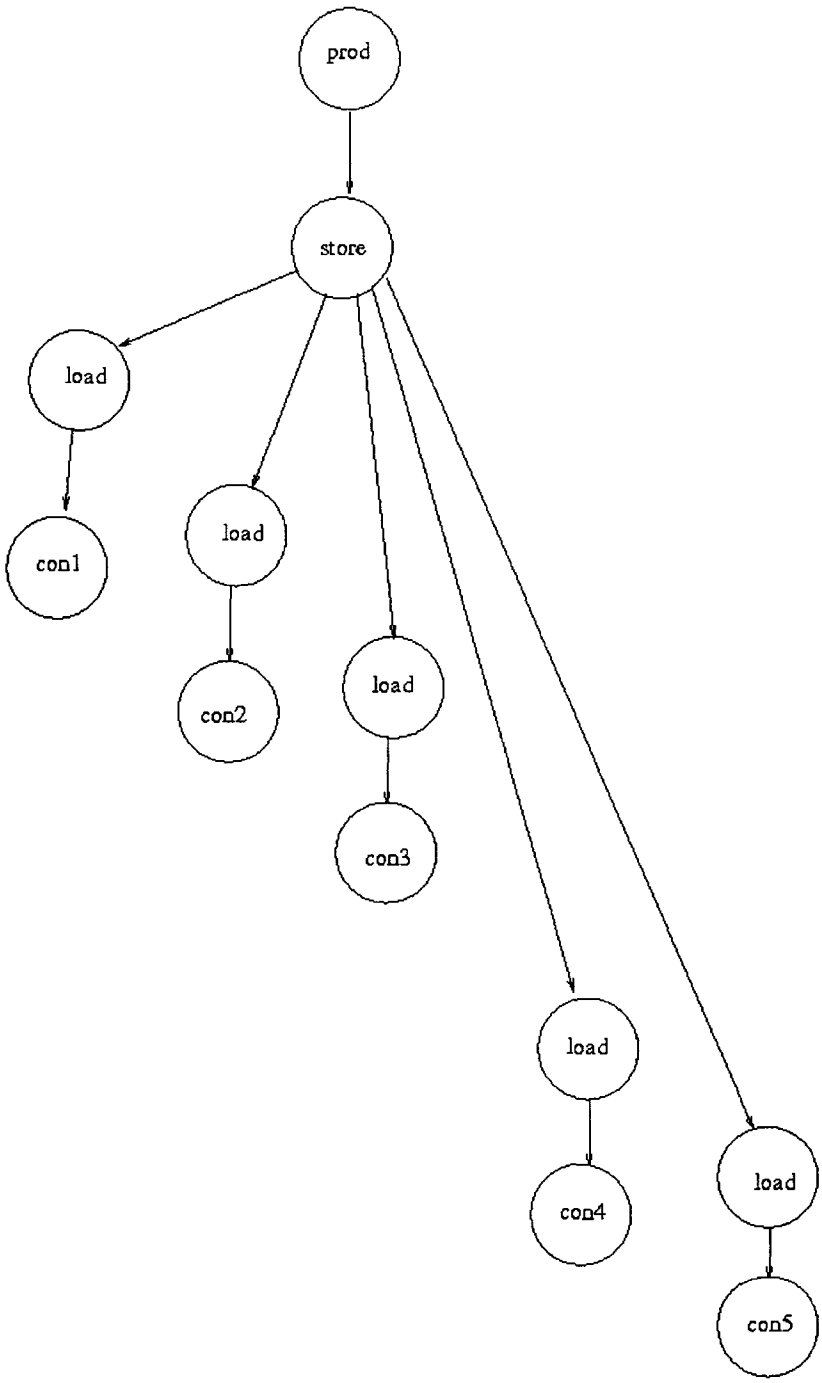

Figure 6.10: Examples illustrating the spilling of variables.

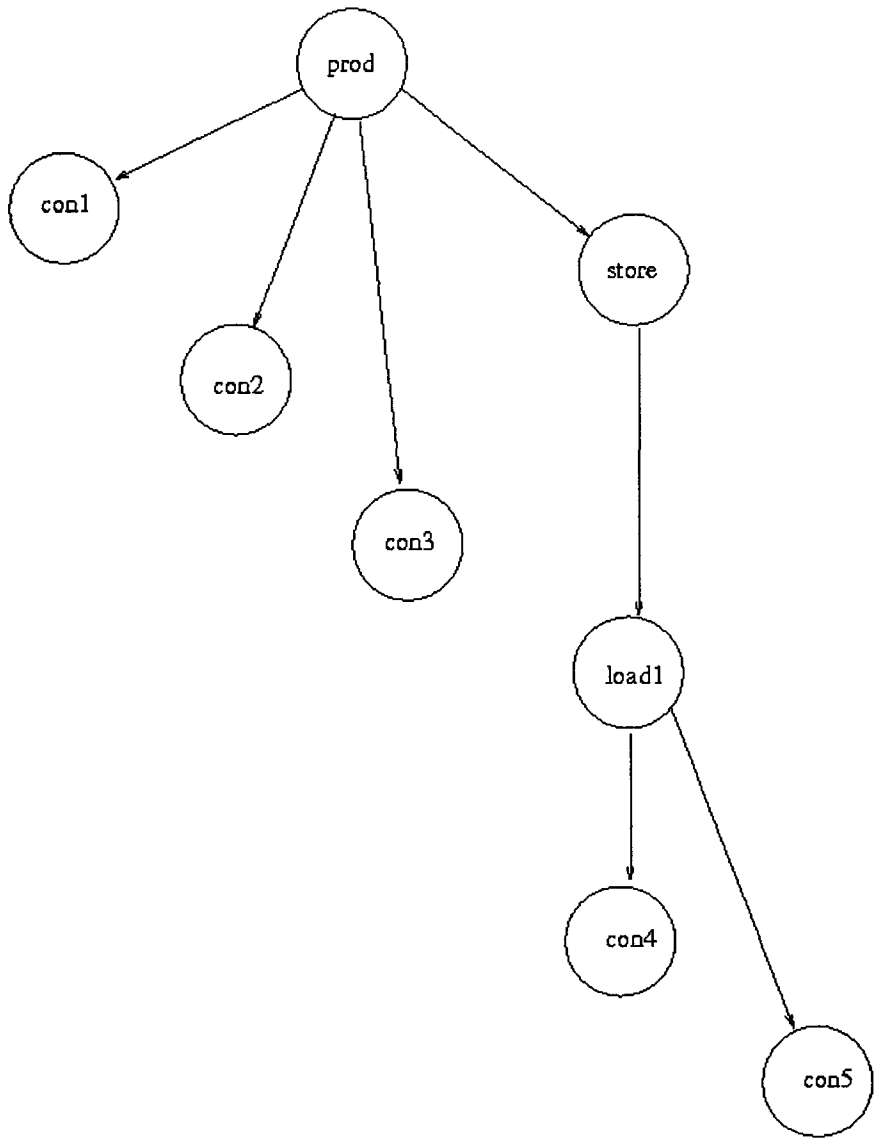

Figure 6.11: Examples illustrating the spilling of uses.

when there is only one definition of a variable and only one use of it.

The heuristic first determines the use with the highest ratio of its lifetime and memory traffic (increased number of load from and store to memory operations due to spilling) and that use must cross the critical cycle in the partial schedule. If such a use does not exist or the use does not span a minimum number of cycles, ejection is done again. A node which is already scheduled in the critical cycle is ejected from the partial schedule and is inserted into the priority_list. This action reduces register requirements in the critical cycle by moving the non-spillable sections of the lifetime outside this cycle.

The inserted spill code adds some nodes in the DFG. These nodes are also inserted in the priority_list. Their priority is set such that they are placed as close as possible to their producer or consumer node. This is done by setting *Early_Start* and *Late_Start* of the node as follows:

$$Early\_Start = Late\_Start - DG$$

and

$$Late\_Start = Early\_Start + DG.$$

where DG is a parameter named distance gauge.

For the newly added nodes due to spilling, the budget is increased in proportion to the number of newly added node.

6.3.8 Restart_Schedule Heuristic

IMIRS abandons the current partial schedule and restarts the whole procedure with an increased II ($II_{Current} + 1$) whenever there is not enough resource left to continue. Increasing II means there are fewer iterations overlapping, so more time slots are available for scheduling nodes. Among these resources that are responsible for increase in II are budget (if it is 0 at a particular time) and available memory (whenever the available memory cannot support the traffic of the newly inserted spill operations for the current II). If there is no such resource constraint then the IMIRS continues with the next highest priority node from the Priority_List.

6.3.9 Improved MIRS for Compilation on CGRA

Figure 6.12 shows the pseudocode of improved MIRS algorithm for adapting to CGRA for cyclic parts. This algorithm uses the node ordering strategy introduced by [Llosa *et al.* 1995] for assigning priority to the nodes of the DFG. *Mapping*(), to be discussed in chapter 7, is used for placement of operations and routing them from producer FU to consumer FU in the available time cycles. The basic steps of the algorithm are summarized below.

At first the algorithm initializes the II with MII. f_1 and f_2 are initialized to empty functions. After the algorithm is completed f_1 will map all the nodes of the DFG with each node having one or more time cycles and a resource depending on the latency of the node's operations. *Budget* is initialized to the number of nodes of the DFG times the *Budget_Ratio*, where *Budget_Ratio* is

```
Procedure IMIRS(DFG, RRG) {
        var II := MII(DFG);
        var f1 := empty();
        var f2 := empty();
        var Priority_List := Order_HRMS(DFG);
        var Budget := Budget_Ratio× Number_Nodes(DFG);
        while (!Priority_List.empty()) {
            var u := Priority_List.highest_Priority();
            Priority_List.remove(u);
            Schedule_Place_Route(DFG, RRG, f1, f2, u);
            Check_and_Insert_Spill(DFG, f1, f2, Priority_List);
            if (Restart_Schedule(DFG, Budget))
                Re_Initialize(II++, f1, f2, Priority_List);
            else
                Budget--;
        }
        Register_Allocation(DFG, f1, f2);
        Generate_Code(f1, f2, II);
}
```

Figure 6.12: Improved MIRS algorithm for Compilation on CGRA.

the average number of times that each node of the DFG can be attempted to be scheduled with a fixed value of II.

After these initializations all the nodes of the DFG are ordered according to Hypernode Reduction Modulo Scheduling [Llosa *et al.* 1995]. The ordered nodes are inserted into *Priority_List*. Then the algorithm iteratively tries to schedule, place, and route operations from the *Priority_List*. In each iteration, the operation with the highest priority is removed from the list and Schedule_Place_Route() tries to find a functional unit (FU) for its execution using a route of free edges of the RRG that minimizes a cost heuristic. If such a FU and time cycle is found without violating any intra-iteration or inter-iteration dependency and resource constraints then those FU and time cycle are reserved for that operation. However, if no such cycle exists, then the algorithm employs the Force_And_Eject technique in which the node to be scheduled is forced to a specific cycle. Force_And_Eject, at the same time, ejects some nodes that were the reasons for dependency violations or resource conflicts.

Then the algorithm determines whether there is any need to spill values to memory to reduce the register pressure. The algorithm also detects the lifetime of a variable or its use which needs spilling. Then Restart_Schedule validates the current partial schedule with the current II. If the current partial schedule is valid then the algorithm continues with the next node of the *Priority_List*, otherwise II is increased and the whole procedure is restarted with the new II.

After all the nodes of the *Priority_List* have been scheduled, the algorithm allocates registers for them. Then the configuration for executing the target

application on the target coarse-grained reconfigurable architecture is generated using the II and the mapping function f_1 and f_2.

When a node u has got its time cycle(s) and FU(s) to execute the required, resources are reserved so that they cannot be utilized by any subsequent nodes until u is finished with utilizing them.

6.4 Scheduling for Acyclic Parts

We will use a simplified version of the IMIRS algorithm for scheduling the acyclic parts (i.e., the application unless the whole application is a loop itself). The input of the algorithm will be an executable data flow graph (DFG) representing the acyclic part and the routing resource graph (RRG) representing the target architecture replicated across time. The output of the algorithm is a schedule of the nodes of the DFG. This schedule assigns two values to each node of the DFG. One is a starting time cycle and another is a functional unit in that time cycle. This schedule will enable each node of the DFG to execute at its time cycle(s) (more than one consecutive time cycles if latency of the node's operation is greater than 1) in its functional unit. The algorithm is shown in Figure 6.13. Like IMIRS(), the nodes are first ordered using the HRMS method. Then the nodes are placed, routed and scheduled one by one starting with the highest priority nodes. While scheduling the acyclic part, cyclic parts embedded in it will be treated as individual nodes. Their priority will be calculated using all the nodes in a particular cyclic part, and the resource requirements of that node

```
Procedure IMIRSA(DFG, RRG) {
        f1 := empty();
        f2 := empty();
        Priority_List := Order_HRMS(DFG);
        while (!Priority_List.empty()) {
            u := Priority_List.highest_Priority();
            Schedule_Place_Route(DFG, RRG, f1, f2, u);
        }
        Generate_Code(f1, f2);
}
```

Figure 6.13: Scheduling algorithm for acyclic Parts of an application.

will be the combined resource requirements of all the nodes of that part.

Chapter 7

Placement and Routing

7.1 Introduction

This chapter introduces our strategy for mapping from DFG to RRG used in the IMIRS algorithm discussed in chapter 6. We have proposed a new placement method to be adapted to CGRA. This method uses the neighborhood relations among the functional units (FUs) and registers. We will denote both FUs and registers as processing elements (PEs).

This phase is the final step before code generation and uses the details of the given target architecture. During HRMS ordering, the relative scheduling order of the nodes of the DFG is determined. During placement the objective is to place the nodes so that the routing of edges among the PEs is minimized following a valid route and also the scheduling can be done so that the overall execution time of the target application can be optimized. So mapping of

nodes and edges of the DFG to the resources in RRG should be such that the producer-consumer relations among the nodes are utilized properly to obtain a near optimal schedule length. For example, if two nodes' operations produce results that are used by another node's operation, the first two nodes should be placed close to one another so that the routing cost is minimized. Also nodes should be placed such that there is a route from the producer PE to the consumer PE. If this criteria is not maintained, one or more extra clock cycle might be needed to ensure routing.

7.2 Idea

Mei *et al.* [Mei *et al.* 2002] use a simulated annealing approach for deciding placement of operations to processing elements. To do that for a particular operation, their method places that operation randomly into a position and evaluates its cost. If a particular position cannot be accepted as the location for that operation, another position is randomly tried, and this process continues for a certain number of times until a valid position is found. In our approach, instead of trying random positions for a random number of times, we start with the unoccupied nearest neighbors of the already scheduled predecessor (or successor) nodes' PEs in an incremental fashion within the range defined by allowable time cycles as shown in chapter 6.

7.3 Placement

The properties of placement are given below:

- The placement is routable. That means all the shortest path edges between the start PE and the end PE should be unoccupied.
- Every processing element deals with at most one operations at a time, i.e., no two operations of the DFG are mapped to the same processing element at the same time.
- The objective function of placement is optimized.

The objective function that has to be minimized is generally the length of the interconnection. This length, in turn, is dependent on the routability and the performance of placement. Since the exact value of the objective function can not be determined before routing is done, the objective function, in most of the cases, is estimated. Another approach which does not adopt approximation of the objective function, is to do routing during the placement. This simultaneous placement and routing approach tends to provide better solutions than the separate steps at the expense of compilation time (in a sequential manner). We will adapt this simultaneous placement and routing to CGRAs.

7.4 Necessity of considering routing during placement

Routing is an essential component of compiling applications to CGRA. It affects the performance of compilation in the form of execution time directly. If it is not considered during scheduling, it might be possible that no PE will be available for placing the operations in a particular clock cycle. We will now illustrate how placement of operations impacts routing by using an example. Figure 7.1 shows a part of a DFG with 7 operations, Op1 to Op7. Op7 is dependent on Op5 and Op6, which in turn are dependent on Op1 and Op2, and Op3 and Op4 respectively. Figure 7.2 shows that after placing Op1 to Op4 in the first row

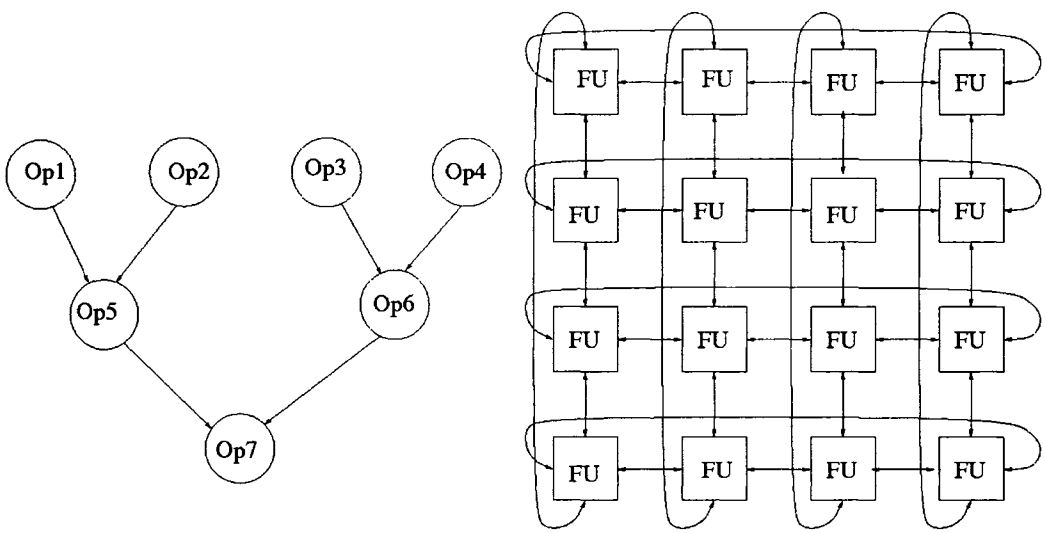

Figure 7.1: Example of a part of a DFG and the sample target architecture.

of the PE array in the first cycle, we need to place Op5 and Op6 in PEs such that there is no routing needed. Now if we place Op5 and Op6 as in Figure 7.3

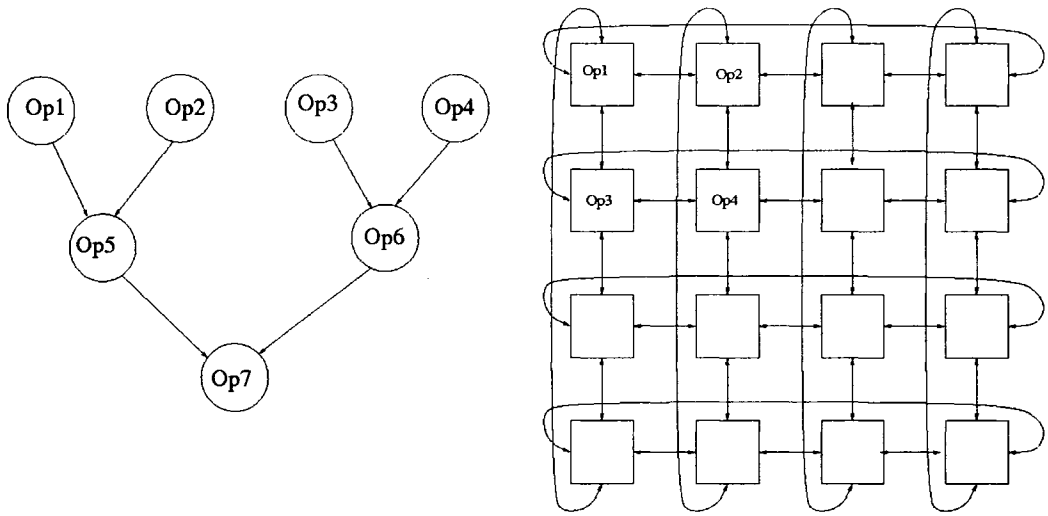

Figure 7.2: Placing the operations at cycle 0.

in clock cycle 2, we cannot place Op7 in clock cycle 3 (as in Figure 7.6). The reason is: the operands of Op7 will not be available at clock cycle 3 to the PE where Op7 will be placed. Op7, in that case, can be executed in clock cycle 4 if either Op5 or Op6 is routed to a PE such that Op5 and Op6 are neighbors of each other as shown in Figure 7.4. If, however, we place Op5 and Op6 as in Figure 7.5, then Op7 can be scheduled in the third cycle as in Figure 7.6. So, routing plays a crucial role in the performance of scheduling (compilation).

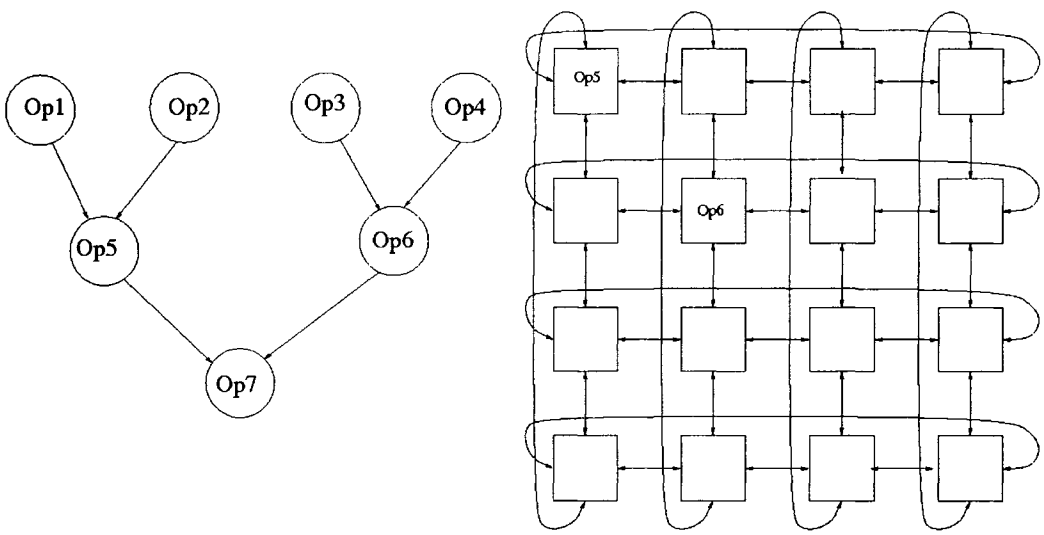

Figure 7.3: Placing the operations at cycle 2 that will induce a delay later.

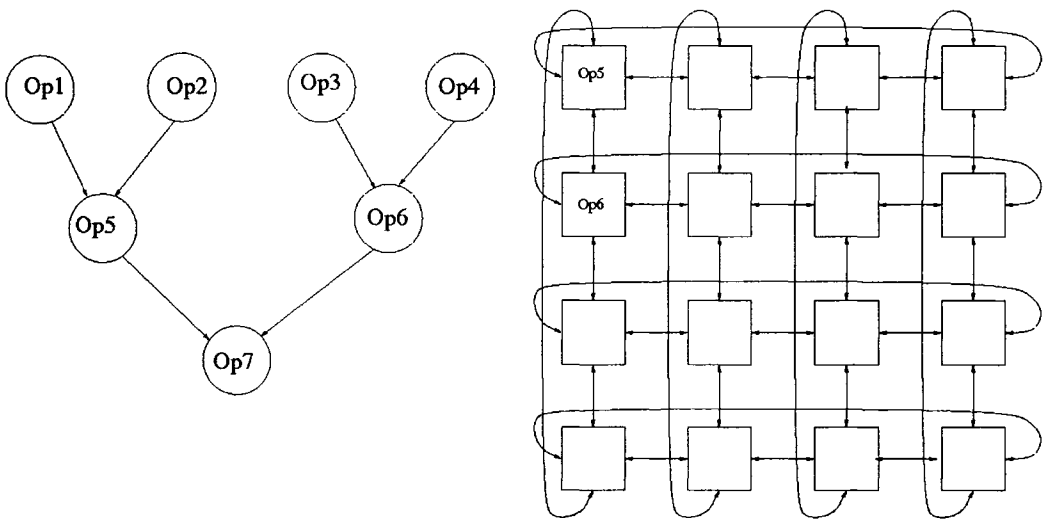

Figure 7.4: Routing needed for executing Op7.

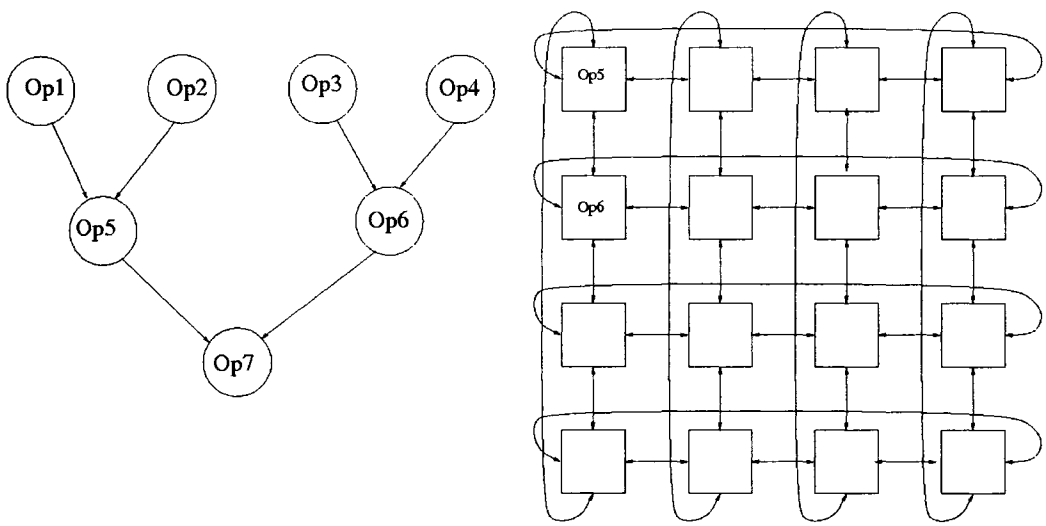

Figure 7.5: Placing the operations at cycle 2 that will induce no delay later.

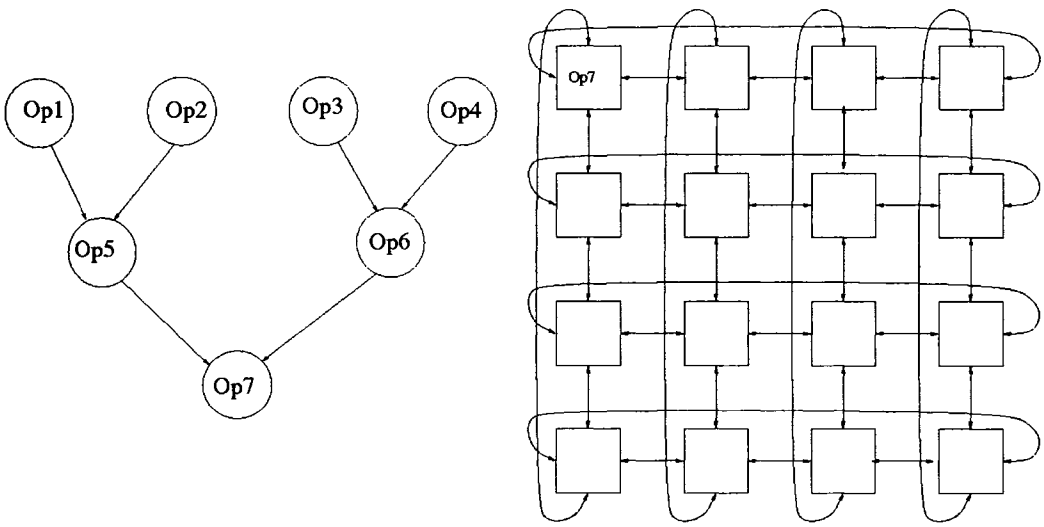

Figure 7.6: Placing the operations at cycle 3.

7.5 Routing

Routing moves data from producer PE to consumer PE using the interconnect structure of the target architecture. For proper routing some extra clock cycles might be added due to the limitations of resources (such as interconnecting bus restrictions, reading or writing port limitations of a memory). Our objective is to decrease the extra clock cycle for the routing.

The routing problem can be formulated as obtaining a set of interconnected paths between the producer processing elements and the consumer processing elements using the interconnection structure of the target architecture. These paths may consist of both the functional units and register files. Like the placement problem, routing should also consider some constraints and minimize an objective function. Typically the length of the routing path is considered as the objective function.

7.6 Placement and Routing Method

We can view the routing resource graph as the given target architecture (composed of processing elements) replicated across time. The interconnections among the PEs in a particular time and across time boundaries define regions with incrementing distances. All the unoccupied PEs in time cycle *Start* can be viewed as the PEs of first choice. The reason for this highest priority is that those PEs can be reached from the producer/consumer PEs in the fewest possible clock cycle. Our idea is to look for a potential PE for a particular node

from these PEs first, provided all the shortest route edges from producers or consumers of a node u to u are also unoccupied. All the possible PEs considering all the predecessors/successors of the node to be placed are tried and a cost function is evaluated for each of them. The PE with the lowest cost is selected for placing the node in question. The cost function evaluated for each path will be discussed in a later section. If such a PE cannot be found, we will explore the PEs at time cycle $(Start + 1)$ and so on. That is, we choose the shortest paths connecting the producer PEs and the consumer PEs.

The algorithm is outlined in Figure 7.8. The first for loop of the algorithm iterates for the number of times equal to the time span the node can be scheduled indicated by *Start* and *End*. This loop determines the possible candidates for mapping the current node. In each iteration unoccupied PEs at time cycle d, such that for a PE all the edges along the shortest path from PSP(u) or PSS(u) to u in the RRG are unoccupied, are elements in the neighbor set. Then the while loop selects the best PE from the candidates. Each neighbor candidate is considered for mapping and a cost function is evaluated for each of them considering all the elements in its PSP(u) or PSS(u). The PE that contributes the lowest cost is selected for placing the operation in question. Then the selected PE is marked occupied at *Selected* from time cycle d to $d+\lambda_u$-1. Using the available interconnections, necessary routing is done that causes optimal routing. In this case, we follow Dijkstra's shortest path algorithm for finding the optimal route. All the edges in RRG along the path P that corresponds to the edges between the selected PE at d and the PE occupied by each node in

PSP(u) or PSS(u) is marked occupied.

7.7 Cost Evaluation

We use a greedy approach for evaluating the cost function of a particular placement. Some related work uses simulated annealing for placement [Mei *et al.* 2002]. Their cost function is a function of overused resources. Our cost function is composed of two components. They are the delay cost, and the interconnect cost. The delay cost of a node u is contributed by the time cycle in which the nodes $\in$ $Pred(u)$ are scheduled. It is equal to the maximum of such delays. The interconnect cost comes from the interconnections that must be dedicated in order to route the node from the producer PE to the consumer PE. The longer the interconnections are occupied, the larger the *Early_Start* of the successor nodes will be. In other words, there will be delays in scheduling successor nodes. The PE with the lowest total of these costs will be selected for executing the current node.

```
Procedure Mapping(DFG, RRG, f1, f2, u, Start, End) {
    var selected, j;
    var time := -1;
    var Old_Cost := Max_Num;
    bool found := false;
    for d := Start to End do {
        if found break;
        var Neighbors := Unoccupied PEs at time cycle d such that
            for a PE all the edges along the shortest path
            from PSP(u) or PSS(u) to u in the RRG are unoccupied
        found := !Neighbors.empty();
    }
    if not found return false;
    while (!Neighbors.empty()) do{
        j := Select a neighbor from Neighbors();
        Neighbors.remove(j);
        var New_Cost := 0;
        for each v ∈ PSP(u) or PSS(u) do
            New_Cost += Evaluate_Cost(DFG, RRG, f1, f2, u, v, j);
```

Figure 7.7: Algorithm for Mapping from DFG to RRG.

```
        if (New_cost < Old_Cost) {
                Old_cost := New_Cost;
                time := d;
                Selected := j;
        }
    }
    f1 := f1 ∪ {u ↦ {(Selected, d), (Selected, d + 1),
                        ..., (Selected, d + λu − 1)}};
    for each v ∈ PSP(u) or PSS(u) do{
        Let P be the shortest path from the last (first) node in f1(v)
        to the first (last) node in f1(u)
        f2 := f2 ∪ {(v, u) ↦ P}
    }
    return true;
}
```

Figure 7.8: Algorithm for Mapping from DFG to RRG (Contd.).

Chapter 8

Conclusion and Future Work

This chapter first summarizes this thesis. Then some possible future work is outlined. Last we compare our work with some related work.

8.1 Summary

The objective of coarse-grained reconfigurable architectures is to achieve close to the performance of customized hardware such as ASICs while capturing most of the flexibilities of general purpose processors. In this thesis a novel compilation approach for parallel applications to coarse-grained reconfigurable architectures has been proposed. The target architecture is specified by the user as shown in chapter 5. The intended application is written in HARPO/L [Norvell 2006], a parallel, object-oriented, multithreaded programming language, as shown in chapter 4. The input of the compilation is the intermediate representation of the target application in the form of Data Flow Graphs (DFG) using static token

(shown in chapter 4) and a description of the target architecture; the output is executable code. HARPO/L is first compiled to a Data Flow Graph (DFG) representation [Zhang 2007]. The remaining compilation steps are a combination of three tasks: scheduling, placement and routing, which are described in chapters 6 and 7. For compiling cyclic portions of the application, we have used a modulo scheduling algorithm: modulo scheduling with integrated register spilling (MIRS) [Zalamea *et al.* 2001a], which incorporates register spilling with instruction scheduling. We have also simplified the MIRS method for acyclic portions of the given application. For scheduling, the nodes of the DFG are ordered using the hypernode reduction modulo scheduling (HRMS) [Llosa *et al.* 1995] method. The placement and routing is done using the neighborhood relations of the processing elements (PEs).

8.2 Future Work

We want to extend the work of this thesis as follows:

- Implementing the proposed compilation method for some benchmark parallel applications in the area of multimedia and embedded systems. Initially we had plan of implementation. But due to lack of time we could not implement the proposed compilation method.

- Comparing the compilation result with some of the related works. In this case we can use some available benchmark applications. We can compare various attributes of the compilation method among some related work. Among those

attributes may be II, schedule length. We can also compare our work with related work by using different target architecture descriptions.

- Making the compiler retargetable across a wide range of target architectures.

8.3 Comparison with Related Work

Our goal was to compile parallel applications to coarse-grained reconfigurable architectures (CGRAs) with near optimal schedule length. Some of the related works [Mei *et al.* 2002][Park *et al.* 2006][Guo *et al.* 2005b]
[Hatanaka and Bagherzadeh 2007] compiled applications to CGRAs. We have tried to improve on their work in several aspects.

- We have targeted parallel applications. For expressing the target applications we have used HARPO/L, a parallel object oriented language suitable for hardware. HARPO/L is also suitable for software implementation. Moreover, it can express explicit parallelism present in the applications. Another important characteristic of HARPO/L is the use of generic parameters. We believe by using HARPO/L we can present the target applications to the target architecture in a more suitable way that will make compilation on CGRAs efficient.

- One of the inputs of our compilation process is the intermediate representation in the form of a data flow graph (DFG) expressed using static token form for parallel programs. Although most of the related work has used DFG as the input (except [Guo *et al.* 2005b], which used CDFG), we think that using

static token for parallel programs to express the DFG facilitates mapping to CGRAs easily.

- Most of the related work has used some variations of modulo scheduling algorithms for scheduling purposes. But none of the works has dealt with register usage during the compilation process. [Hatanaka and Bagherzadeh 2007] assumes there are sufficient number of registers in the target architecture. This may not always be the case. We have adapted a modulo scheduling algorithm that considers register usage during scheduling (Chapter 6).

- Compilation approaches on CGRAs mainly focus on cyclic portions of the target application. The reason is: CGRAs are intended for accelerating the time consuming portion of the applications (generally loops). But most of the related works have not mentioned how to deal with the whole application. We have given an approach in this regard (Chapter 5).

- We have used an ordering (HRMS [Llosa *et al.* 1995]) which orders the operations of the DFGs in such a way that after scheduling them in this order we will have a schedule with near optimal schedule length (i.e., execution time). The reason is: nodes are ordered such that a node is scheduled to its predecessors/successors as close an possible. As a result, the life time of all the variables are optimized. This, in turn, reduces the register usage. Other related works have used other ordering techniques. But HRMS is better than those [Llosa *et al.* 1995].

- [Mei *et al.* 2002] uses a simulated annealing approach for scheduling, map-

ping, and routing. But the process is very time-consuming. We have used a systematic search range for the whole process determined by the *Early_Start* (and/or *Late_Start*), II, and the predecessors/successors in the partial schedule. This approach is less time-consuming than simulated annealing.

Appendix A

HARPO/L Syntax

In Chapter 4 we use HARPO/L for writing the target application. Here we will briefly describe the syntax of HARPO/L from [Norvell 2006].

Meta notation

$N \rightarrow E$	Nonterminal N can be an E
$\underline{(}E\underline{)}$	Grouping
E^*	Zero or more
E^{*F}	Zero or more separated by Fs
E^+	One or more
E^{+F}	One or more separated by Fs
$E^?$	Zero or one
$\underline{[}E\underline{]}$	Zero or one
$E \mid F$	Choice

A.1 Classes and Objects

A.1.1 Programs

A program is a set of classes, interfaces, and objects.

$$Program \rightarrow \underline{(}ClassDecl \mid IntDecl \mid ObjectDecl\underline{)}^*$$

A.1.2 Types

Types come in several categories.

- Primitive types: Primitive types represent sets of value. As such they have no mutators. However objects of primitive types may be assigned to, to change their values. Primitive types represent such things as numbers. They include
  - int8, int16, int32, int64, int
  - real16, real32, real64, real
  - bool
- Classes: Classes represent sets of objects. As such they support methods that may change the object's state.
- Interfaces. Interfaces are like classes, but without the implementation.
- Arrays: Arrays may be arrays of primitives or arrays of objects.

– Generic types. Generic types are not really types at all, but rather functions from some domain to types. In order to be used, generic types must be instantiated.

Types are either names of classes, array types or specializations of generic types

$$Type \rightarrow Name \mid Type(Bounds) \mid Type\,[GArgs]$$

Arrays are 1 dimensional and indexed from 0 so the bounds are simply one number

$$Bounds \rightarrow ConstIntExp$$

A.1.3 Objects

Objects are named instances of types.

$$ObjectDecl \rightarrow \textbf{obj}\ Name[\underline{\ } : Type\underline{\ }] := InitExp$$

The Type may not be generic. The type and preceeding colon may be omitted when the type can be inferred from the initialization expression, for example **obj** f := **new** A is the same as **obj** $f : A$:= **new** A

Initialization of an object can be an expression or an array initialization

$$\begin{aligned} InitExp &\rightarrow Exp \mid ArrayInit \mid \textbf{new}\ Type \\ &\mid \left(\textbf{if}\ Exp\ InitExp\ \textbf{else}\ InitExp\ [\underline{\textbf{if}}]\right) \\ ArrayInit &\rightarrow \left(\textbf{for}\ Name : Bounds\ InitExp\ [\underline{\textbf{for}}]\right) \end{aligned}$$

- If the object to be initialized is of a primitive type (such as **int32** or **real64**), the *initExp* should be a compile-time constant expression of a type assignable to the type of the object.

- If the object to be initialized is an array, then the *initExt* should be an *ArrayInitExp.*

- If the object to be initialized is an object of non-primitive type, then the InitExp should be of the form **new** *Type* where the *Type* is a non-generic class type.

- In any case, the InitExp can be an if-else structure in which the expression is a compile-time constant assignable to **bool**.

A.1.4 Classes

A class defines a type. Classes may be generic or nongeneric. A generic class has one or more generic parameters

$$ClassDecl \rightarrow \Big(\textbf{class}\ Name\ GParams^{?}\ \underline{(}\textbf{implements}\ Type^{*,}\underline{)}^{?}\ \underline{(}ClassMember\underline{)}^{*}\ \underline{[}\textbf{class}\underline{]}\Big)$$

- The *Name* is the name of the class.

- The *GParams* is only present for generic classes, which will be presented in a later section.

- The *Type*s are the interfaces that the class implements.

A.1.5 Class Members

Class members can be fields, methods, and threads. [Nested classes and interfaces are a possibility for the future.]

$$ClassMember \rightarrow Field \mid Method \mid Thread \mid ;$$

Fields are objects that are within objects. Field declarations therefore define the part/whole hierarchy.

$$\begin{aligned} Field &\rightarrow Access\ \textbf{obj}\ Name[\,:Type\,] := InitExp \\ Access &\rightarrow \textbf{private} \mid \textbf{public} \end{aligned}$$

Method declarations declare the method, but not its implementation. The implementation of each must be embedded within a thread.

$$\begin{aligned} Method &\rightarrow Access\ \textbf{proc}\ Name([\,Direction\ [Name :]\ Type\,)^{*,}\,)] \\ Direction &\rightarrow \textbf{in} \mid \textbf{out} \end{aligned}$$

The types of parameters must be primitive.

Recommended order of declarations is

- public methods and fields, followed by
- private methods and fields, followed by
- threads.

There is no 'declaration before use rule'. Name lookup works from inside out.

A.2 Threads

Threads are blocks executed in response to object creation.

$$Thread \rightarrow (\textbf{thread}\ Block\ [\textbf{thread}])$$

Each object contains within it zero or more threads. Coordination between the threads within the same object are the responsibility of the programmer. All concurrency within an object arises from the existence of multiple threads in its class. Thus you can write a monitor (essentially) by having only one thread in a class.

A.2.1 Statements and Blocks

A block is simply a sequence of statements

$$Block \rightarrow (Statement \mid ;)^*$$

Statements as follow

– Assignment statements

$$\begin{aligned} Statement &\rightarrow ObjectId := value \\ ObjectId &\rightarrow Name \mid ObjectId(Expression) \end{aligned}$$

The type of the ObjectId must admit assignment, which means it should be a primitive type, like **int32** or **real64**.

– Local variable declaration

$$\mathit{Statement} \rightarrow \textbf{obj}\ \mathit{Name}[\ :\ \mathit{Type}]\ :=\ \mathit{InitExp}$$

Same restrictions as fields.

– Method call statements

$$\begin{aligned}\mathit{Statement} &\rightarrow \mathit{ObjectId.Name}(\mathit{Args})\\ &\mid \mathit{Name}(\mathit{Args})\end{aligned}$$

– Sequential control flow

$$\begin{aligned}\mathit{Statement} &\rightarrow \Big(\textbf{if}\ \mathit{Expression}\ \mathit{Block}\ (\textbf{elseif}\ \mathit{Expression}\ \mathit{Block})^{*}(\textbf{else}\ \mathit{Block})^{?}\ [\textbf{if}]\Big)\\ &\mid \Big(\textbf{wh}\ \mathit{Expression}\ \mathit{Block}\ [\textbf{wh}]\Big)\\ &\mid \Big(\textbf{for}\ \mathit{Name}\ :\ \mathit{Bounds}\ \mathit{Block}\ [\textbf{for}]\Big)\end{aligned}$$

– Parallelism

$$\begin{aligned}\mathit{Statement} &\rightarrow \Big(\textbf{co}\ \mathit{Block}\ (||\ \mathit{Block})^{*}\ [\textbf{co}]\Big)\\ &\mid \Big(\textbf{co}\ \mathit{Name}\ :\ \mathit{Bounds}\ \mathit{Block}\ [\textbf{co}]\Big)\end{aligned}$$

In the second case, the *Bounds* must be compile-time constant.

– Method implementation.

$$\begin{aligned}\mathit{Statement} &\rightarrow \Big(\textbf{accept}\ \mathit{MethodImp}\ (|\ \mathit{MethodImp})^{*}\ [\textbf{accept}]\Big)\\ \mathit{MethodImp} &\rightarrow \mathit{Name}(\ (\mathit{Direction}\ \mathit{Name}\ :\ \mathit{Type})^{*,}\ )\ [\mathit{Guard}]\ \mathit{Block}_0\ [\textbf{then}\ \mathit{Block}_1]\\ \mathit{Guard} &\rightarrow \textbf{when}\ \mathit{Expression}\end{aligned}$$

* Restrictions

  - The directions and types must match the declaration.
  - The guard expression must be boolean.
  - Each method may only be implemented once per class

* Possible restrictions:

  - The guard may not refer to any parameters.
  - The guard may refer only to the in parameters.

* Semantics: A thread that reaches an accept statement must wait until there is a call to one of the methods it implements and the corresponding guard is true. Once there is at least one method the accept can execute, one is selected. Input parameters are passed in, $Block_0$ is executed and finally the output parameters are copied back to the calling thread. If there is a $Block_1$ it is executed next.

– Sequential consistancy

$$Statement \rightarrow \left(\textbf{atomic } Block \; [\underline{\textbf{atomic}}]\right)$$

The block is executed as-if atomically. That is, any two atomic statements within the same object can not execute at the same time unless they can not interfere with each other.

A.3 Genericity

Classes and interfaces can be parameterized by "generic parameters". The effect is a little like that of Java's generic classes or C++'s template classes. Classes and interfaces may be parameterized, in general, by other classes and interfaces, values of primitive types, for example integers, and objects.

Programs using generics can be expanded to programs that do not use generics at all. For example a program

---

```
(class K ... class)
obj k : K
(class G[ x : K ] ...x... class)
obj g : G[k]
```

Expands to

```
(class K ... class)
obj k : K
(class G0 ...k... class)
obj g : G0
```

Generic parameters may be one of the following

- Nongeneric Types

- Nongeneric Classes
- Objects
- Values

$$
\begin{aligned}
\mathit{GParams} &\rightarrow \mathit{GParam}^{+,} \\
\mathit{GParam} &\rightarrow \textbf{in}\ \mathit{Name} : \mathit{Type} \\
&\mid \textbf{obj}\ \mathit{Name} : \mathit{Type} \\
&\mid \textbf{type}\ \mathit{Name}\ \underline{[}\textbf{extends}\ \mathit{Type}^{+,}\underline{]}
\end{aligned}
$$

$$
\mathit{GArgs} \rightarrow \underline{(}\mathit{Type} \mid \mathit{Expression}\underline{)}^{+,}
$$

Bibliography

[Aho *et al.* 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Aiken *et al.* 1995] Alexander Aiken, Alexandru Nicolau, and Steven Novack. Resource-constrained software pipelining. *IEEE Trans. Parallel Distrib. Syst.*, 6(12):1248–1270, 1995.

[Allen *et al.* 1983] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM Press.

[Alsolaim *et al.* 2000] Ahmad Alsolaim, Janusz Starzyk, Jurgen Becker, and Manfred Glesner. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. *fccm*, 00:205, 2000.

[Ananian 1999] C. S. Ananian. The static single information form. Master's thesis, MIT, 1999.

[Beck *et al.* 1991] M. Beck, R. Johnson, and K. Pingali. From control flow to data flow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, 1991.

[Cha] http://www.chameleonsystems.com.

[Codina *et al.* 2002] Josep M. Codina, Josep Llosa, and Antonio González. A comparative study of modulo scheduling techniques. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 97–106, New York, NY, USA, 2002. ACM Press.

[Cytron *et al.* 1989] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[Cytron *et al.* 1991] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Dani 1998] A. Dani. Register-sensitive software pipelining. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on Inter-*

national Parallel Processing Symposium, page 194, Washington, DC, USA, 1998. IEEE Computer Society.

[Ebeling *et al.* 1997] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping Applications to the RaPiD Configurable Architecture. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 106–115, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[Ferrante *et al.* 1987] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[Goldstein *et al.* 2000] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *Computer*, 33(4):70–77, 2000.

[Green and Franklin 1996] C. Ebeling D. C. Green and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, 1996. Springer-Verlag.

[Guo *et al.* 2003] Yuanqing Guo, Gerard J.M. Smit, Hajo Broersma, and Paul M. Heysters. A graph covering algorithm for a coarse grain reconfigurable system. *SIGPLAN Not.*, 38(7):199–208, 2003.

[Guo *et al.* 2005a] Y. Guo, C. Hoede, and G. J. M. Smit. A multi-pattern scheduling algorithm. In T. P. Plaks, R. DeMara, M. Gokhale, S. Guccione, M. Platzner, G. J. M. Smit, and M. Wirthlin, editors, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05), Las Vegas, Nevada, USA*, pages 276–279, USA, June 2005. CSREA Press.

[Guo *et al.* 2005b] Y. Guo, G. J. M. Smit, H. J. Broersma, M. A. J. Rosien, P. M. Heysters, and T. Krol. Mapping applications to a coarse grain reconfigurable system. In *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pages 93–104, Dordrecht, 2005. Springer.

[Guo *et al.* 2006a] Y. Guo, C. Hoede, and G. J. M. Smit. A column arrangement algorithm for a coarse-grained reconfigurable architecture. In T. P. Plaks, R. DeMara, M. Gokhale, S. Guccione, M. Platzner, G. J. M. Smit, and M. Wirthlin, editors, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'06), Las Vegas, Nevada, USA*, pages 117–122, USA, June 2006. CSREA Press.

[Guo *et al.* 2006b] Y. Guo, C. Hoede, and G. J. M. Smit. A pattern selection algorithm for multi-pattern scheduling. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - 12th Reconfigurable Architecture Workshop (RAW 2006), Rhodes Island, Greece*, pages 198–205, Los Alamitos, CA, USA, April 2006. IEEE Computer Society.

[Guo 2006] Y. Guo. *Mapping Applications to a Coarse-Grained Reconfigurable Architecture.* PhD thesis, Univ. of Twente, Zutphen, September 2006.

[Hartenstein and Kress 1995] Reiner W. Hartenstein and Rainer Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *ASP-DAC '95: Proceedings of the 1995 conference on Asia Pacific design automation (CD-ROM)*, page 77, New York, NY, USA, 1995. ACM Press.

[Hartenstein 2001] R. Hartenstein. A Decade of Reconfigurable Computing: A Visionary Retrospective. In *Design, Automation and Test in Europe*, pages 642–649, Munich, Germany, Mar 2001. IEEE Computer Society.

[Hatanaka and Bagherzadeh 2007] Akira Hatanaka and Nader Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *IPDPS 2007: Proceedings of the IEEE symposium on Parallel and Distributed Processing*, pages 1–8, 2007.

[Hoare 1978] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Jain 1991] Suneel Jain. Circular scheduling: a new technique to perform software pipelining. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 219–228, New York, NY, USA, 1991. ACM Press.

[Lam 1988] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988*

conference on programming language design and implementation, pages 318–328, New York, NY, USA, 1988. ACM Press.

[Lamport 1979] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[LaPaugh and Rivest 1978] Andrea S. LaPaugh and Ronald L. Rivest. The subgraph homeomorphism problem. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 40–50, New York, NY, USA, 1978. ACM Press.

[Lee *et al.* 1997] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Languages and Compilers for Parallel Computing*, pages 114–130, 1997.

[Lee *et al.* 1999] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Principles Practice of Parallel Programming*, pages 1–12, 1999.

[Llosa *et al.* 1995] Josep Llosa, Mateo Valero, Eduard Ayguadé, and Antonio González. Hypernode reduction modulo scheduling. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 350–360, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[Llosa *et al.* 2001] Josep Llosa, Eduard Ayguadé, Antonio González, Mateo Valero, and Jason Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. Comput.*, 50(3):234–249, 2001.

[Llosa 1996] Josep Llosa. Swing modulo scheduling: A lifetime-sensitive approach. *pact*, 00:0080, 1996.

[Marshall *et al.* 1999] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143, New York, NY, USA, 1999. ACM Press.

[Mat] http://www.mathstar.com/.

[Mei *et al.* 2002] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures, 2002. In International Conference on Field Programmable Technology.

[Mei *et al.* 2003a] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *FPL*, pages 61–70, 2003.

[Mei *et al.* 2003b] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE '03:*

Proceedings of the conference on Design, Automation and Test in Europe, page 10296, Washington, DC, USA, 2003. IEEE Computer Society.

[Mei *et al.* 2005] Bingfeng Mei, Andy Lambrechts, Diederik Verkest, Jean-Yves Mignolet, and Rudy Lauwereins. Architecture exploration for a reconfigurable architecture template. *IEEE Des. Test*, 22(2):90–101, 2005.

[Mirsky and DeHon 1996] E. Mirsky and A. DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[Miyamori and Olukotun 1998] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable multimedia array coprocessor (abstract). In *FPGA*, page 261, 1998.

[Moon and Ebcioğlu 1992] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 55–71, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Norvell 2005] Theodore S. Norvell. CGRA Project Proposal. www.engr.mun.ca/ theo, 2005.

[Norvell 2006] Theodore S. Norvell. HARPO/L: Language Design for CGRA project. www.engr.mun.ca/ theo, 2006.

[Pangrle and Gajski 1987] B.M. Pangrle and D.D. Gajski. Design tools for intelligent compilation. *IEEE Transactions Computer-Aided Design*, 6(6):1098–1112, 1987.

[Park and Schlansker 1991] J. C. H. Park and M. S. Schlansker. On predicated execution. Hewlett Packard Laboratories, 1991. Technical Report HPL-91-58.

[Park *et al.* 2006] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146, New York, NY, USA, 2006. ACM Press.

[Paulin and Knight 1989a] Pierre G. Paulin and John P. Knight. Algorithms for high-level synthesis. *IEEE Design and Test of Computers*, 6(6):18–31, 1989.

[Paulin and Knight 1989b] Pierre G. Paulin and John P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[Pingali *et al.* 1991] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: an Algebraic Approach to Program Dependencies. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 445–467. MIT Press, Cambridge, MA, 1991.

[Rau and Glaeser 1981] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.

[Rau 1994] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[Singh *et al.* 2000] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49(5):465–481, 2000.

[Teifel and Manohar 2004] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *ASYNC '04: Proceedings of the 10th International Symposium on Asynchronus Circuits and Systems*, pages 17–27, 2004.

[Venkataramani *et al.* 2001] Girish Venkataramani, Walid Najjar, Fadi Kurdahi, Nader Bagherzadeh, and Wim Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *CASES '01: Proceedings of the 2001 international conference on Compil-*

ers, architecture, and synthesis for embedded systems, pages 116–125, New York, NY, USA, 2001. ACM Press.

[Waingold *et al.* 1997] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *Computer*, 30(9):86–93, 1997.

[Walker and Chaudhuri 1995] Robert A. Walker and Samit Chaudhuri. Introduction to the scheduling problem. *IEEE Des. Test*, 12(2):60–69, 1995.

[Warter *et al.* 1992] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *MICRO 25: Proceedings of the 25th Annual International symposium on Microarchitecture*, pages 170–179, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Wik 2007] July 2007. http://www.wikipedia.org.

[Zalamea *et al.* 2001a] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. MIRS: Modulo scheduling with integrated register spilling. In *Proc. of 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 239–253. Springer-Verlag, August 2001.

[Zalamea *et al.* 2001b] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *MICRO 34: Proceedings of the 34th annual*

ACM/IEEE international symposium on Microarchitecture, pages 160–169, Washington, DC, USA, 2001. IEEE Computer Society.

[Zhang 2007] Dianyong Zhang. An Intermediate Representation for CGRA Implementation. Master's thesis, Memorial University of Newfoundland, 2007.