Automatic Compilation of Field-Programmable Custom Compute Accelerators

by

Qiang Wang

A Thesis submitted in conformity with the requirements for the Degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering University of Toronto

© Copyright by Qiang Wang 1999
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

0-612-41334-9
Abstract

Automatic Compilation of Field-Programmable Custom Compute Accelerators

Doctor of Philosophy, 1999

Qiang Wang

Department of Electrical and Computer Engineering
University of Toronto

High performance can be obtained on field-programmable custom computers for applications by implementing computational kernels in hardware. Significant speedups can be obtained by mapping time critical computations in the source program in hard-wired logic, and creating multiple specialized functional units in FPGAs to exploit parallelism in the program. However, design of these machines is an expensive and time consuming task and must be performed once for each given application.

This dissertation describes a compiler, PECompiler, that generates both hardware and control software for field-programmable compute accelerators. The first goal of our compiler is to automate the design process of implementing a specialized computer for each given application. The second goal of the compiler is to exploit intrinsic parallelism in the source program that contains irregular patterns of computations. To attain these goals, we introduced a new design methodology for configurable custom computers, and created a parameterized architecture for modeling hardware features of accelerators. Through compiling the source code of an application together with part of its input, PECompiler can determine the specialized processing units required by an application program, create design descriptions, expose hidden parallelism by analyzing data dependencies in the source code, and produce instruction sequences to run on the customized accelerator.

Partial evaluation is the key technique that enables the compiler to optimize source code using part of the program’s input. In order to generate the instructions for each application, we developed a heuristic algorithm to determine an optimized sequence for the computations to execute on the accelerator. We also developed a new algo-
algorithm to handle register allocations during the process of scheduling operations to run on the accelerator.

We used PECompiler to generate designs of customized accelerators for two applications, and compared the performance of each program running on the accelerator with the program running on the general-purpose computer. The results of the experiments showed that significant speedups were achieved for both applications. To provide the insight for designers for implementing the accelerators on an FCCM, we also studied several architectural issues in the accelerators, and demonstrated the impact to the performance caused by these issues.
Acknowledgments

I would like to thank my supervisor, Professor David M. Lewis, for his guidance, advice, and encouragement. Through all my Ph.D. years, I have benefited from his sharp technical expertise, broad research experiences, and sober earnest attitude in seeking authentic answers to problems. I would also like to thank the members of my oral examination committee: Professors Paul Chow, P. Glenn Gulak, Brad Hutchings, and David A. Johns for their valuable comments and detailed suggestions on the dissertation. Special thanks are due to Professor Jonathan Rose for providing me with sources of technical information about FPGAs and industrial connections. I would also like to thank Ms. Sarah Cherian for her help.

It was my privilege to be surrounded by the collective wisdom of my fellow graduate students. I would like to thank Keith, Steve, Yaska, Vaughn, Javad, Ali, Minghui, Jianshe, Xiaoyun, Mazen, Khoman, Andy, Louis, Jordan, Jason, Marcus, Vincent, Mark, Warren, Sandy, Khalid, John, Heather, Kevin, Bob, Jianghong, Sean, Wai, Jason A., Ken, Nirmal, Rob, and the rest of the graduate students in rooms SF2202, SF2206, LP392, and LP476.

It had been a long journey before I reached this point, and there were many difficult times on the road. I gratefully thank my wife Li Zhang for her constantly supporting and understanding. Without her this dissertation would never have been completed. I also appreciate my parents, sister, and brother for their long time encouragement and support.

Finally, I appreciate the financial support for this project provided by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, Communications and Information Technology Ontario, Ricoh Corporation, and University of Toronto (both University of Toronto Open Fellowships and Teaching Assistantships).

This dissertation is dedicated to the memory of my grandparents. I managed to carry their hopes with my dream on the way through all these years, but I walked too slow...
# Contents

1 Introduction  :  1
  1.1 Motivation  :  1
  1.2 The Construction of FCCM Platforms  :  4
  1.3 Implementations of Customized Computers on the FCCM  :  5
  1.4 Research Approach and Dissertation Outline  :  6

2 FCCM Platforms and Customized Computers  :  7
  2.1 Architectures of Customized Computers  :  7
  2.2 Examples of FCCMs  :  11
    2.2.1 Splash 2 Custom Computing Machine  :  11
    2.2.2 Programmable Active Memories (PAM)  :  13
    2.2.3 Marc-1 System  :  15
    2.2.4 Transmogrifier-2 System  :  16
  2.3 Hardware Accelerators  :  18
    2.3.1 Yorktown Simulation Engine  :  18
    2.3.2 Awsim-3 Accelerator  :  22
  2.4 Partial Evaluation  :  23
    2.4.1 Principles of Partial Evaluation  :  24
    2.4.2 Online and Offline Partial Evaluation  :  27
    2.4.3 Binding-Time Analysis  :  29
  2.5 Objectives and Goals  :  29
  2.6 Summary  :  30
# The Architecture and Design Methodology of Field-Programmable Compute Accelerators

## 3.1 General Design Approach

## 3.2 Architecture of Configurable Accelerators
- 3.2.1 An Abstract Architecture
- 3.2.2 Parameterized Machine Model
- 3.2.3 The Time-Stationary Control Strategy

## 3.3 Overview of the Compiler Design
- 3.3.1 Design Considerations
- 3.3.2 Data Dependency Graph
- 3.3.3 Source Code Structures and Syntax Tree

## 3.4 Summary

## 4 Automated Hardware Design
- 4.1 Construction of the Compiler
- 4.2 The Partial Evaluator
  - 4.2.1 Application Programming Language
  - 4.2.2 User Annotations
  - 4.2.3 Deriving Variable Status Using BTA
  - 4.2.4 Data Dependency Graph Generation
- 4.3 Conditional Branches
- 4.4 Hardware Functional Units
  - 4.4.1 Extraction of FU Structures
  - 4.4.2 Generating VHDL Descriptions
- 4.5 DDG Optimization
  - 4.5.1 Removal of False Dependencies in DDGs
- 4.6 Summary

## 5 Instruction Sequence Generation
- 5.1 Overview of Approach
  - 5.1.1 Scheduling Operations on the Accelerator
5.2 A Heuristic Scheduling Algorithm .................................................. 70
5.3 Pattern Inference Vectors ................................................................. 72
5.4 Register Allocation ............................................................................ 76
  5.4.1 The Graph Coloring Method ......................................................... 76
  5.4.2 The Register Allocation Algorithm ............................................... 77
5.5 The General Scheduling Algorithm ................................................... 83
5.6 Summary ............................................................................................ 83

6 Examples .............................................................................................. 85
  6.1 Simple-Timing Simulation Program .................................................... 85
  6.2 Simple-Circuit Simulation Program .................................................. 93
  6.3 Compilation Time ............................................................................. 98
  6.4 Effect of Partial Evaluation ................................................................ 99
  6.5 Summary .......................................................................................... 100

7 Performance Analysis ........................................................................... 102
  7.1 The Top-Level Interconnection Network .......................................... 103
    7.1.1 The Full Crossbar Network ......................................................... 105
    7.1.2 The Mesh Network .................................................................... 107
    7.1.3 The Ring Network .................................................................... 109
    7.1.4 Performance Comparisons .......................................................... 110
  7.2 Register Files .................................................................................... 116
    7.2.1 Register File Implementation ...................................................... 116
    7.2.2 Register Usage of STS and SCS Applications ............................... 118
  7.3 Summary .......................................................................................... 123

8 Conclusions and Future Work ................................................................ 125
  8.1 Thesis Summary ............................................................................... 125
  8.2 Contributions ................................................................................... 126
  8.3 Future Work ..................................................................................... 127

References ............................................................................................. 129
A The Programming Language for Applications 135

A.1 The Source Programming Language ............................. 135
A.2 A Special Format .................................................. 136
# List of Figures

2.1 Splash 2 Architecture ........................................ 12  
2.2 DECPeRL-1 Architecture ................................. 14  
2.3 Marc-1 Architecture ........................................ 17  
2.4 The Yorktown Simulation Engine .......................... 20  
2.5 Awsim-3 Architecture ...................................... 23  
2.6 Both Sides of the Equation in Definition 2.1 .......... 25  
2.7 An Example of Program Specialization .................. 26  

3.1 Acceleration of Program Execution on FCCM Platform .... 34  
3.2 Top-Level Architecture ..................................... 36  
3.3 Processing Module Architecture ........................... 37  
3.4 Port Buffers of PM_Level Interconnection Network ...... 38  
3.5 The Circuit of a 4-Entry Port Buffer ...................... 39  
3.6 Data-Stationary and Time-Stationary Machines ........... 41  
3.7 General Function of the Compiler .......................... 43  
3.8 An Example Data Dependency Graph ....................... 45  
3.9 Syntax Tree Representations of Program Structures ..... 47  

4.1 The Compiler Structure ...................................... 50  
4.2 User Annotations in the Application Program ............. 53  
4.3 Deriving the Flattened DDG for a Loop Statement ........ 56  
4.4 Using Resolved-Write for a Conditional Writing Operation 58  
4.5 Designing a Functional Unit ................................ 60  
4.6 Breaking the False Dependency Chain ..................... 64
4.7 Removing False Dependencies ........................................... 65

5.1 The DDG and Customized Accelerator ............................... 69
5.2 A Delay Model for DDG Nodes ......................................... 71
5.3 Schedule Table and Interference Table ............................... 73
5.4 The Register Allocation Table ......................................... 78
5.5 Examples of allocating Registers ...................................... 80
5.6 Examples of Releasing Registers ...................................... 81

6.1 The STS Source Program .................................................... 87
6.2 The Optimized Program and DDG ....................................... 88
6.3 The Structures of the Five FUs ......................................... 90
6.4 Number of PMs versus Clock Cycle Ratio ............................ 92
6.5 The Structures of four FUs .............................................. 95
6.6 Number of PMs versus Estimated Speedup ........................... 97

7.1 Interconnection Network Using Full Crossbar Structure ........... 106
7.2 Interconnection Network Using Mesh Structure .................... 108
7.3 Interconnection Network Using Ring Structure ........................ 110
7.4 Performance of STS Program with the encoder circuit ............ 112
7.5 Performance of STS Program with the crc Circuit ................. 112
7.6 Performance of STS Program with the 8-bit counter Circuit ........ 112
7.7 Performance of SCS program with the 4-bit counter circuit ........ 115
7.8 Performance of SCS program with the 6-bit adder circuit ........ 115
7.9 Performance of SCS program with the 20-inverter circuit ........ 115
7.10 Performances of STS Program with the encoder Circuit .......... 120
7.11 Performance of STS Program with the crc Circuit ................. 120
7.12 Performance of STS Program with the 8-bit counter Circuit ........ 120
7.13 Performance of SCS Program with the 4-bit counter Circuit ........ 122
7.14 Performance of SCS Program with the 6-bit adder Circuit ........ 122
7.15 Performance of SCS Program with the 20-inverter Circuit ........ 122
| A.1  | The Statements Supported by the Compiler | 135 |
| A.2  | The Operators Supported by the Compiler | 135 |
| A.3  | The CiLF Format (Part I)                | 137 |
| A.4  | The CiLF Format (Part II)               | 138 |
| A.5  | The CiLF Format (Part III)              | 139 |
List of Tables

4.1 The List of Components in the Library ........................................ 62
6.1 The FUs Needed by the STS Program ........................................ 89
6.2 Number of PMs versus Execution Cycles .................................... 91
6.3 The FUs Needed by the SCS Program ........................................ 94
6.4 Total Numbers of the FUs in Different Machine Models ............... 96
6.5 Running the SCS Program to Simulate the Three Circuits on an UltraSPARC-Model170 workstation. ........................................... 96
6.6 Number of FUs versus Total Execution Cycles ............................ 97
7.1 Hardware Cost for Implementing the Multiplexers ........................ 107
Chapter 1

Introduction

1.1 Motivation

The introduction of field-programmable gate arrays (FPGAs) [BFRV92] in the early 1980s rapidly led to the development of field-programmable custom computing machines (FCCMs). An FCCM is typically constructed using FPGAs and SRAMs, and can be configured as a high-performance computer for compute-intensive programs by mapping computations onto hardware on the FPGAs. For each of the targeted programs, a customized computer is implemented on the FCCM with specialized processing units to speed up executions of the core computations. Since many programs offer abundant parallelism, speedup can be gained by increasing the total number of processing units on the configured machines, subject to hardware resource limitations in the FCCM.

From the standpoint of this dissertation, we classify modern high-performance computers into three categories: (1) supercomputers, (2) special-purpose computers, and (3) field-programmable custom computing machines (FCCMs).

Supercomputers, such as the Cray-1 [HB84] vector computer and the SGI Challenge [HP90] multiprocessor computer, have been built to speed up executions of massive computations in user programs. This kind of computer can achieve high performance by using a deep pipelined structure or a high degree of replication of the hardware to exploit parallelism available in the programs, but the cost to build a
supercomputer is very high.

Special-purpose computers, such as hardware accelerators YSE [Den82] and Ausim [Lew88], were built to implement a complete algorithm in hardware. This kind of computer was usually constructed with an array of dedicated processing elements (PEs) and hardwired interconnection among the PEs. Each of the PEs can perform a specific computation step for an application program, and these PEs can simultaneously execute multiple computations in the application. Special-purpose computers have been built for various application programs, but each of the machines can only offer high performance to a limited range of programs. As a result, special-purpose computers are not as widely used as general-purpose computers.

FCCMs, such as Splash 2 [ABD92] and PAM [VBR+96], are programmable custom computers. An FCCM can be configured as different custom computers for various application programs, and each custom computer that is implemented on the FCCM can offer high performance for a range of applications. This kind of computer system often has a high configurability to implement hardware functions to meet requirements of different programs, but users have to create hardware designs of the specialized custom computers for the programs.

Compared to supercomputers or special-purpose computers, FCCMs have following evident advantages:

- **Fast Implementation**: Using field-programmable hardware devices, designers can avoid the need to design and fabricate VLSI chips. The time needed to construct a hardware system and implement a custom machine can be greatly reduced, in contrast to using full custom design chips.

- **Reconfigurability**: SRAM-based FPGAs [XIL96] [ALT96] on an FCCM allow users to implement part of the computations in hardware. For a range of user programs, the same FCCM can be reconfigured as a high-performance custom computer for each of the programs, with specialized processing units that can speed up the execution of the particular program.

- **Low Cost**: The cost of building the hardware of FCCMs is much lower than
building vector computers or array processors.

On the other hand, there are also a number of disadvantages for building custom machines on FCCMs:

- **Lower area density:** The density of logic circuits on an FPGA chip is much lower than the density of logic circuits on a custom designed VLSI chip. Therefore, the number of FPGA chips is larger than the number of custom designed VLSI chips needed for implementing the same logic design.

- **Slower logic:** Two issues related to FPGAs cause FCCMs to have slower logic speed than conventional machines. First, FPGA devices are much slower than custom designed VLSI chips. Even using the same fabrication technology, circuits implemented on the FPGAs always experience extra delays caused by the use of the programmable logic and routing resources. Second, since multiple FPGA devices are used to implement a custom computing machine, the I/O delay time of FPGA chips and the delay for signals going through the interconnection network among the FPGA chips may increase latencies of individual operations performed on FCCMs.

Two means may be used to achieve speedup for program executions on an FCCM. One of them is implementing specialized processing units in the FPGAs to quickly perform part of the computations in the user program. A single operation executed on each of these hardware units can accomplish a function that would need execution of multiple instructions on a general-purpose computer. The second approach is to exploit the parallelism in the source program using replication of the hardware units to execute multiple operations at the same time. Therefore, the major design goals of customized computer design are to create specialized hardware units that can accelerate the operations in the targeted program and to expose hidden parallelism among the program operations that can be executed on the hardware units at the same time.

The process of building customized computing machines in general can be divided into two stages. The first stage is to construct an FCCM platform using FPGAs and
other VLSI chips. The same platform can then be configured as different customized computers for a variety of application programs, but a specific FCCM can only be used to implement one computer at a time. In the second stage, a customized computer is implemented for a given application program by configuring the FPGAs in the system. This may be performed many times for various applications on a single FCCM platform.

In this dissertation, we focus on the problem of creating the hardware and control software for customized computers that are to be implemented on an FCCM platform. We introduce a general architecture for customized computers and propose a new design methodology that can be used to build CAD tools to automatically generate the designs based on the source programs. Before presenting our design approach, we briefly describe the general construction of FCCM platforms and current methodologies of implementing customized computers for targeted applications.

1.2 The Construction of FCCM Platforms

An FCCM platform is usually constructed with FPGAs, RAM chips, and sometimes field-programmable interconnection devices (FPIDs). It may be built on a single board or multiple boards. The FPGAs in the system are used to implement arithmetic/logic units and control circuits, and the RAM chips are used as data memory blocks and control stores. Since each FPGA chip only has a limited number of pins available, it may only be connected to a few other FPGA or RAM chips. Often a RAM is directly connected to each FPGA. To meet the needs of different specialized machine structures, FPID chips may be used to provide reconfigurable connection topologies among FPGAs. For each kind of application, these FPIDs are programmed to connect all the hardware units implemented on the FPGAs and form a customized computer structure on the FCCM.

In the design of an FCCM system, one of the most important tasks is to decide the structure of the interconnection network connecting all the FPGA and RAM chips in the system. Although FPGAs and FPIDs are programmable, the wire connections
among chips on circuit boards cannot be changed after the hardware system is built. The FPIDs, as well as the wiring structure, need to provide flexibility for implementing all the different interconnections among the chips on the FCCM that are possibly required on the customized computer created for the targeted applications.

1.3 Implementations of Customized Computers on the FCCM

Once an FCCM platform has been built, it can be configured as a specialized computer for a range of application programs. For each of these programs, designers first need to determine the computations that may be mapped on hardware based on the analysis of the source program, and must then design hardware functional units to implement the computations. A tailored machine for the targeted program can then be constructed using a number of the designed functional units with some interconnection between the units. The decisions on the total number of the functional units mapped on the FCCM, the interconnect structure, and the sequence of operations performed on these functional units are determined according to the program execution flow and the available hardware resources on the FCCM.

Creating a customized computer requires a detailed hardware design based on the program requirements. To finish the design task in a modest time, current methodologies have only been applied to the programs that have simple, regular structures. The process of generating the design is a time consuming task, and needs to be performed once for every application program. It would be highly desirable to automate the process of generating a unique hardware design of the customized computer that is to be implemented on the FCCM for the targeted program. Our research is motivated to solve this difficult problem by creating a compiler that can find the functional requirements of application programs, explore inherent parallelism among operations, automatically design a specialized FCCM, and generate control software for operations in the application program to be executed on the customized machine.
The compiler should also be able to handle irregular patterns of computations in application programs, and improve the efficiency of program executions by eliminating statically determinable computations in the source code.

### 1.4 Research Approach and Dissertation Outline

This research explores design methods for building specialized FCCMs for different applications. The primary goal of the research work is to build a compiler that can compile a given source program and automatically generate a customized FCCM to speedup the execution of core computations in the program.

In Chapter 2, before presenting the thesis work, some typical FCCM designs are first reviewed. It also introduces a program optimization technique, partial evaluation, that plays a key role in the automated design flow.

This dissertation describes our design goals in Chapter 3. It first defines a parameterized architecture for hardware systems of FCCMs. Then it introduces a new methodology of automatically generating the design of customized computers, and further proposes a compiler structure that implements the methodology.

The first step of generating a specialized hardware configuration on an FCCM for a given program is to find the needs for specialized hardware structures for the program. For this purpose, a compilation tool, called a partial evaluator, is created to determine the part of the computations that can be mapped onto specialized functional units and generate the hardware design for the units. Chapter 4 describes how our compiler generates the design of specialized hardware units for given applications.

Chapter 5 shows how our compiler produces control software for executing operations of the application program on the customized FCCM.

Two examples of designing FCCMs for application programs are presented in Chapter 6. Results show high performance can be obtained for these programs on the automatically designed FCCMs.

Chapter 7 discusses some hardware design issues. It also presents a detailed analysis on determining several structural parameters for a customized FCCM.
Chapter 2

FCCM Platforms and Customized Computers

This chapter describes the background of the dissertation research. It presents an overview of previous design approaches of FCCM platforms and discusses architectural features of computing machines that can be implemented on FCCMs. Section 2.2 shows four examples of FCCMs and describes how to build specialized accelerators on the FCCMs to speed up executions of application programs. Section 2.3 discusses two specific examples of hardware accelerator designs. Section 2.4 describes a key technique, partial evaluation, that will be used to realize design automation for customizing given FCCMs as specialized computing machines for computationally intensive applications.

2.1 Architectures of Customized Computers

As described in Chapter 1, the process of building customized computers can be divided into two stages, (1) constructing an FCCM platform using FPGAs, FPIDs, and RAMs; (2) implementing a customized computer by hardware programming the FPGAs and FPIDs in the FCCM. Once an FCCM has been created, it can be configured as different computers for a range of application programs. Each of the computers is specialized for a particular application.
The architecture of an FCCM platform is characterized by the types and numbers of the component chips (FPGAs, FPIDs, and RAMs) and the structure of the hardwired connections among the chips. The architecture of a customized computer that is implemented on the FCCM is characterized by the functionalities of the processing units and the structure of the buses interconnecting all the units. The architecture of a particular accelerator built for a given application is determined based on the computation requirements of the program. To use an FCCM platform to implement a variety of customized computers for a range of applications, the FCCM platform must provide adequate resources for mapping the required processing units and the interconnect network.

A customized computer implemented on an FCCM is usually connected to a host computer. In general, the host computer is responsible for compiling the source program, generating the configurations of the FCCM platform, and controlling the configured computer to run user programs. During the execution of a program, the host also performs the I/O functions in the source program, sends operands to data memory on the FCCM, initializes the configured computer to perform the computation tasks in the program, and retrieves the results back from the FCCM memory.

Based on how the host computer controls the execution on the customized computers, the architectures of the customized computers can be classified into three categories: special-purpose processors, coprocessors, and attached-processors [BAK96]. Accordingly, design approaches of FCCMs for implementing these three kinds of architectures may vary from one to another.

Special-purpose processors have been built to implement the entire algorithm for application programs. The execution sequence of the operations on the processor is usually implemented using logic circuits, which are mapped on the FCCM together with the specialized functional units. To achieve high execution speed on special-purpose processors implemented on the FCCM, the interconnection among the FPGAs on the FCCM are usually hardwired.

Most FCCMs in this category have been built for rapid prototyping a system design. Although each FCCM can implement a variety of algorithms, the special-
ized hardwired connections on the FCCM confine the flexibility within a limited range. There have been several of this kind of FCCM built for a variety of applications. For example, special-purpose processors were built for neural network computations [CB91]. Other examples of FCCMs can be found in [CM94] [HWGG93].

Coprocessors have been built to extend the set of operations that can be performed in a conventional CPU. Each of the coprocessors is usually tightly coupled with a host CPU. For a given application, the FCCM is configured as a set of special functional units that target the fine-grained computations in the source program. Operations on the configured units are controlled by new instructions that are added into the instruction set architecture of the host processor. A compiler needs to be developed for the overall system of the combined host CPU and the processor mapped on the FCCM, so that the new instructions defined for the coprocessor can be inserted into the executable code to use the reconfigurable logic to speedup computations in the user program. The operation on the coprocessor is controlled by an individual instruction, which interweaves with other instructions that are executed on the host CPU. To use a coprocessor implemented on the FCCM, users also need to create a compiler that can select particular computations that frequently appear in the source program for executing on the coprocessor. The compiler also has to determine runtime reconfigurations of the coprocessor [Hau98]. Typical examples of the FCCMs proposed for implementing coprocessors are Garp [HW97], OneChip [WC96], and PRISM [AS93] [WAL+93].

Attached-processors have been built for accelerating the majority of the computations in the application programs. Such a processor in general is loosely coupled with a host computer and used as a compute accelerator. It performs the bulk of the coarse-grained computations, which need many steps on multiple functional units on the FCCM, in the application programs. Such an accelerator is constructed with an instruction unit and a datapath with multiple processing units. These processing units perform operations specified by the instructions executed on the accelerator. The flexibility of mapping different kernels onto the FCCM is realized by implementing specialized processing units to perform logic functions for each given program and also
generating the instruction routines to run on the accelerator, instead of hardwiring the execution sequence on the FCCM. Before starting to run the user program, the host computer configures the FCCM into a customized accelerator and stores these instruction routines in the instruction memory of the accelerator. During the execution, the main program running on the host machine sends commands to call the routines to run on the accelerator. The typical examples of FCCMs built for implementing attached processors are Splash 2 [ABD92] and DECPeRLe-1 PAM [VBR+96].

Recently, Researchers at MIT (Massachusetts Institute of Technology) Laboratory for Computer Science proposed a Raw microprocessor architecture [WTS+97] [AAB+97] for implementing high-performance computing machines. The design is based on the assumption that a billion transistors can be integrated on a single chip in the near future. The Raw architecture is make up of a set of identical tiles that are interconnected as a two-dimensional array. Each of the tiles contains an instruction memory (IMEM), a data memory (DMEM), an arithmetic logic unit (ALU), registers, configurable logic (CL) resources, and a programmable switch with its associated instruction memory (SMEM). A Raw chip can implement various microprocessor architectures by using the programmable switches in the tiles. For example, a Raw chip can be configured as a superscalar microprocessor with a single register file, a single memory, and multiple ALUs that communicate on a global bus. The chip can also be configured as a multiprocessor with a set of processors communicate through the memory subsystem. This kind of chip architecture relies on a compiler and run-time software to determine the best resource allocation for each application and exploit parallelism across tiles. The Raw architecture is suited for scientific applications and for processing regular streams of data. In contrast, our goal is to use partial evaluation to exploit statically determinable parallelism in the application programs with irregular patterns of the computations.

This dissertation focuses on the design of FCCMs used as compute accelerators (attached-processors). Before starting to describe our design work, the rest of the chapter will examine some designs of FCCMs and compute accelerators.
2.2 Examples of FCCMs

This section reviews four multi-FPGA systems. They are all constructed with multiple SRAM-based FPGAs and can be used to implement attached processors for a variety of applications. Three of the systems: Splash 2, DECPeRLe-1 PAM, and Marc-1 were specifically built as FCCMs. The last one, Transmogrifier-2, which was built as a rapid prototyping system, can also be used to implement customized computers.

2.2.1 Splash 2 Custom Computing Machine

Splash 2 [ABD92] is a large scale FCCM that can be used to implement application-specific attached-processors. The hardware system of Splash 2 is composed of \( n \) identical array boards and one interface board. The total number \( n \) can be up to 16. Figure 2.1(a) shows the top-level architecture. There are also a number of buses used to interconnect all the array boards. The interface board is the general controller of the Splash 2 and responsible for communication between the Splash 2 and a host computer. It receives commands from the host, generates clock signals for all the 16 array boards of the Splash 2, synchronizes operations among these array boards, and transfers data between the host and the Splash 2 boards.

Each of the 16 array boards in the Splash 2 is a reconfigurable computing board. Figure 2.1(b) shows the array board architecture. It is constructed with 17 Xilinx XC4010 FPGAs, each of which is directly connected to one 256K x 16 RAM block, and a crossbar switch that interconnects the 17 FPGAs. One of the 17 FPGAs (X0) and its associated connected memory block (M0) are used to form a control unit on the board. X0 is used to implement the logic circuit of the control unit and M0 is used as a control store. The other 16 FPGAs are individually coupled with the 16 memory blocks and used for data manipulation. Each FPGA together with the associated memory block can be seen as a processing element. The FPGA can be configured as a functional unit and the RAM block can be either used as a local memory in the element or accessed by a global bus on the array board. The crossbar switch provides
flexible interconnections among the 16 processing elements. The interconnect pattern of the crossbar is selected by the local control unit (X0) on the array board and can be switched dynamically during the execution. There are also direct connections between two neighbor FPGA chips that can be used to create a linear structure among the 16 chips.

A specialized computing machine implemented on the Splash 2 can be viewed as a SIMD machine. When a single instruction stream is decoded on the interface board,
all array boards cooperate on multiple data operands. The buses on the Splash 2 system can either transfer data through the array boards in serial or broadcast to all the boards in parallel.

To use the Splash 2 as a custom computer for an application program, users must not only provide a source program, but also create a hardware design of the attached processor that is required by the program. The source program is written in the C programming language, and the hardware design is described using VHDL code. In the VHDL code, users need to specify the hardware functions that are required to be implemented on all the processing elements of the Splash 2. These VHDL descriptions are used under the Splash 2 software environment to generate configurations of all the FPGAs in the system. The source program is compiled to generate instruction routines to prepare data operands, control the operations on the attached processor, and retrieve results generated by the processor. Using this design process, users must understand the Splash 2 system, have knowledge of circuit design, and design a processor for each application program. This makes the use of the Splash 2 system very difficult.

2.2.2 Programmable Active Memories (PAM)

DECPeRL-1 PAM [VBR+96] is a reconfigurable hardware system that can be customized as an application-specific processor. A processor implemented on the PAM is attached to a host computer and used to accelerate the execution of the computational kernels in the targeted application program. The DECPeRL-1 hardware system is built on a single board, which contains 23 Xilinx XC3090 [XIL96] chips and 4 MB static RAM. Figure 2.2 shows the architecture of DECPeRL-1. A 4 x 4 FPGA array (16 chips) forms the computational core of the board. The 16 FPGAs are interconnected using a mesh structure. Each chip has 16 bits of direct connections to its four neighbors. There are also four 64-bit buses across the FPGA array horizontally and vertically, with two 16-bit buses shared by four chips in the same row or in the same column. On each edge of the array, there is another FPGA, which is used as switcher, and a 1-MB RAM block. The shared buses across the array are connected.
to the RAM block through a switch FPGA on each array edge. Two other FPGAs are used to generate addresses for the four RAM blocks on the array edges. The four address signals are shown in the figure as \textit{AddrN}, \textit{AddrE}, \textit{AddrW}, and \textit{AddrS}. There is also another FPGA that is used as switcher to link the PAM to a host computer. To avoid the need to synchronize clocks, two \textit{FIFO} buffers are used for data transfers between the host and the customized processor on the PAM. At run time, the host sends data operands to the PAM through one of the two \textit{FIFO} buffers and receives results generated by the PAM processor from the other \textit{FIFO} buffer.

Many applications have been built on the DECPeRLe-1 system [VBR+96]. The source code of application programs are written using the C++ programming language. The PAM design tools can compile the source program and generate driving software that will be run on the host machine to control the operations on the configured processor on the PAM. The design methodology is similar to using the Splash 2
system. Users need to design the processor and provide the logic descriptions of the required functions that need to be mapped onto all the FPGA devices when using the PAM board as a specialized processor. Thus it is difficult for programmers to use the PAM system to accelerate executions of application programs.

2.2.3 Marc-1 System

Marc-1 [LvIW93] is a field-programmable compute accelerator. The hardware system of the Marc-1 consists of two identical processing modules and a programmable interconnect unit. Figure 2.3 shows the Marc-1 architecture. Each of the modules is constructed with nine Xilinx XC4005 FPGAs chips [XIL96], SRAM chips (256k x 64bits instruction memory, 256k x 32bits data memory), and a Weitek 3364 floating point unit. Three of the FPGAs are used to form an instruction unit for the module, the other six FPGAs are used to implement a datapath unit. These six FPGAs are organized as three pairs, with each FPGA belonging to either the upper or lower half of the 32-bit datapath. The instruction unit mapped onto the three FPGAs calculates memory addresses for both instruction and data memories and decodes fetched instructions. The two modules are interconnected through the programmable interconnect unit, which consists of another five Xilinx XC4005 FPGAs. One of the five FPGAs is used as a controller and each of the remaining four is used as a byte-wide slice of the 32-bit interconnect unit. The controller can change the connecting pattern implemented by the four FPGAs according to signals generated by the instruction units on the two boards. The interconnect unit provides five buses to link from each board and the other two buses from the outside world. All the buses are 32-bits-wide.

The design task of customizing an accelerator includes a hardware configuration of the Marc-1 and an instruction set that defines all the operations of the configured accelerator. When using the Marc-1 as a special-purpose accelerator, functional units are mapped onto the FPGA chips in the system to speed up the execution of critical operations in the application program. To implement the hardware functions for each specific application, users need to design the hardware units that are to be mapped on the FPGAs and provide an instruction set that defines all the operations
of these units. A compiler must be built to generate instruction code to run on every accelerator configured for its targeted source program. Based on the instruction set designed for the corresponding application program, the compiler can use the hardware features to generate efficient code for the Marc-1 system. This design method uses the compiled-code technique to generate instructions for applications to run on customized accelerators, but the compiler still needs to be crafted for each particular application. This method can be improved by using a parameterized architecture to implement a variety of accelerators, so that the compiler can generate instructions for each of the accelerators based on a set of architectural parameters.

2.2.4 Transmogrifier-2 System

The Transmogrifier-2 (TM-2) [LGvI'97] is a reconfigurable hardware system with a capacity of 1 million gates for user circuits. Although it was built for prototyping a wide range of applications, the architecture of the system is well suited for use as a custom computing machine. The TM-2 is composed of multiple FPGA boards. Each board is constructed with two Altera EPF10K50 [ALT96] FPGAs, four I-cube interconnect chips, and up to 8 MB of memory. The largest version of the system comprises 16 boards, which in total contains 32 FPGA chips. The interconnection among all the FPGAs in the system is built using a novel non-uniform partial crossbar structure.

The TM-2 system can be used to implement customized computing machines for application programs that are computationally intensive. Such a customized machine can provide a large number of specialized hardware units and high-speed execution of the programs by parallelizing computations on the hardware units. The TM-2 machine is attached to a host computer (A SUN SPARCstation) as a reconfigurable accelerator. At run time, the host machine is responsible for sending data and control signals to the accelerator.

The TM-2 system is still under development. Two of the FPGA boards have been built and used for several test examples. The latest version of the TM-2 board design will use 32 Altera EPF10K100 [ALT96] devices, instead of the EPF10K50 devices.
This increases the maximum capacity of the TM-2 to 2 million gates for user circuits.

To allow users to implement circuits on the TM-2 system, a compiler, called Transmogrifier C [Gal95] was created to produce a circuit, which is described in an xnf [XIL96] file, based on an input program written in a subset of the C programming language. The compiler was designed to compile a complete program, which is in general a short program, into a hardware circuit. In our design approach, the compiler uses partial evaluation to exploit intrinsic parallelism in large programs, determines computational kernels in the programs to be implemented on FPGAs in an FCCM platform, and generates control sequences to execute the kernel operations on the FCCM.
2.3 Hardware Accelerators

One way to speed up the execution of a program is to build a hardwired accelerator that has a dedicated hardware structure to implement the algorithm of core computations in the program. The high performance on this kind of accelerator is obtained by implementing specialized functional units to perform the computations, and a customized hardware structure to execute multiple operations at the same time. For the same reason, it is very expensive to gain speedup using the custom designed accelerators, as each unique algorithm requires the development of a specialized machine.

Research efforts have been made to create a single accelerator that can meet the requirements of a range of applications. There are two kinds of methods to solve the problem. One method is to use compiled-code techniques during compilation of source programs and seek the best way to use the hardware features of the given accelerator to improve performance for different application programs [App88] [Lew92]. Another method is to design a reconfigurable accelerator using SRAM-based FPGA devices. For each specific application, the specialized functional units and hardware structure can be implemented on the accelerator by fetching bit streams into the FPGA chips.

This section describes two examples of compiled-code hardware accelerators. Section 2.3.1 describes the design of Yorktown Simulation Engine (YSE), and Section 2.3.2 describes the design of the Awsim-3 accelerator.

2.3.1 Yorktown Simulation Engine

The YSE [Den82] [KP82] is a special-purpose, highly parallel, programmable computer that was built to provide high-speed for the gate level simulation of logic circuits. It was constructed with multiple processors that can perform unit-delay and rank-order [Den82] simulations. A full YSE configuration can simulate circuit networks up to two million gates. For a large circuit, the logical network can be partitioned into sub-circuit networks, and each sub-circuit can be simulated by one of the processors in the YSE. During simulation, all the processors compute the logical values for signals on the sub-circuits, store the values in their memory, and transfer values of the
signals between sub-circuits among the processors. The operations on each processor are specified by the instructions executed on the processor. The YSE is used as a hardware accelerator. An IBM 370/168 is used as the host computer for the YSE system.

Figure 2.4(a) shows the overview of the YSE system. It consists of logic processors, array processors, an inter-processor switch, a bus controller, and a control processor. The logic processors may be used to perform random logic simulation, with each of the processors simulating a portion of the targeted logic circuit. The array processors may be used together to simulate RAMs or ROMs in the targeted circuit. The inter-processor switch can provide communication among up to 256 logic and array processors during the execution of the system. The bus controller connects the control processor to all the logic and array processors in the YSE system, specifies the data transfers among the processors, and accesses logic values of signals stored in the data memories of the processors. A global bus is used by the bus controller to control all the processors and the switch in the system. The control processor, which is a standard Z8000 microcomputer system, is used to reduce the requirements for frequent communication between the host computer and the YSE system. It sets up paths for data transfers among the host, its own memory, and the bus controller. It can also handle interrupts from the processors in the YSE.

Each of the processors consists of four components: an instruction memory, a program counter, a data memory, and a function unit. Figure 2.4(b) shows the processor architecture. The instruction memory can store 8K 128-bit instructions. Every instruction contains five 14-bit addresses for fetching operands from the data memory, one 13-bit address for storing data to the data memory, five 4-bit Generalized DeMorgan memory (GDM) [Den82] codes, a 5-bit function code, and an interrupt mask. The program counter specifies the address of the current instruction, and it sequentially steps through the instruction memory from the starting address to the ending address of the instructions. The clock cycle time for loading one instruction from the instruction memory is called an instruction cycle. The execution of all the instructions from the starting to ending addresses is called a simulation cycle. The
Figure 2.4: The Yorktown Simulation Engine
data memory physically consists of five identical 16K 2-bit banks, which together implement a single 16K x 2 RAM with five read ports and two write ports. Each memory bank is divided into an 8K x 2 local data memory (LDM) and an 8K x 2 switch data memory (SDM). During the first half of each instruction cycle, the output of the function unit (generated by last instruction) is written into the LDM (in all the five banks), and data received from the inter-processor switch is written into the SDM (in all the five banks). During the second half of the cycle, five operands are read out from the five banks. For unit delay simulation, logic values are read from one half of each memory bank (LDM) and stored into the other half (SDM). The roles of the two halves (LDM or SDM) are reversed after every simulation cycle. The function unit is used to compute a 2-bit value every instruction cycle based on four 2-bit data operands, five 4-bit GDM codes, and a 5-bit function code.

The structure of the function unit is shown in Figure 2.4(c). It contains five 64 x 2-bit GDMs (RAMs) and an 8K x 2 Function RAM. Each of the GDMs can perform sixteen 2-bit-valued functions of a single 2-bit argument. The four high-order address bits of the GDM are used to select one of the sixteen functions, and the two low-order address bits represent a 2-bit operand. The function unit can generate a 2-bit output of a gate function of up to four 2-bit arguments. All six RAMs are loaded during YSE initialization. The purpose of the GDMs, as well as the 8K x 2 Function RAM, is to perform transformations to logical values according to De Morgan's theorems, so that arbitrary logic functions of 2-bit-valued variables in simulated netlist can be evaluated by looking up the contents of the RAMs. Each GDM RAM contains sixteen four-entry truth tables, and the Function RAM contains thirty-two 256-entry tables. The function RAM can implement several Boolean functions that are related by De-Morgan's theorems by using the GDM to complement some of the inputs and outputs. The Function Code and GDM codes fetched from the instruction memory are used to select particular truth tables to implement gate functions for circuit simulation. The compiled-code technique was used to generate a customized simulation engine on the YSE platform for each circuit that needs to be simulated. A compiler [KP82] was built based on the technique. It takes as input the logic descriptions of the cir-
cuit and generates both contents of all the GDM RAMs and instructions that are executed on the customized engine. The YSE is a successful example of using the compiled-code technique to generate customized hardware designs. Unfortunately, the YSE system is a special-purpose computer system, and was built only for logic simulations of digital circuits.

2.3.2 Awsim-3 Accelerator

Awsim-3 [Lew92] is a hardware accelerator designed for circuit simulation applications. By using compiled-code techniques, a range of algorithms for circuit simulation can be implemented in programs and the generated programs are executed with high performance on the accelerator.

Figure 2.5(a) shows the architecture of the Awsim-3 system. It was constructed with two main components, a special-purpose processor (SP) and a general-purpose processor (GP). The two processors are interconnected by a local bus, which can also handle multiple GPs for extended systems. The SP was designed to implement an approximation algorithm, called the tree-based model approximation (TBMA) [Lew90], for evaluating device models. Figure 2.5(b) presents the datapath of a single GP. It is a VLIW machine that consists of five identical processing slices. Each of the slices contains a 2K word register file, a 128K word data memory, an arithmetic/logic unit, three stages of full crossbars, and a 512K word by 64 bit instruction memory. Awsim-3 is a pipelined machine with five slices synchronously executing five independent instruction streams.

High performance is also a result of the compiler that was built for the machine. To take advantage of hardware features of the Awsim-3 accelerator, a compiler is capable of taking as input a circuit netlist and generating instruction streams that can parallelize operations on the five slices of the datapath.

Both of the previous accelerators operate by compiling a specific problem into code for a dedicated machine. This requires the design of a specialized compiler for each application, which is a time consuming task. For example, the compiler of Awsim-3 was written in the C programming language with fourteen thousand lines.
of code. One goal of this dissertation is to demonstrate that an automated form of

![Overview of Awsim-3](image)

(a) Overview of Awsim-3

![Datapath of General-Purpose Processor](image)

(b) Datapath of General-Purpose Processor

Figure 2.5: Awsim-3 Architecture

the transformation from a program into specialized code can be achieved through

the use of partial evaluation. By using this kind of transformation technique, we can

build one single compiler, and use it to compile different applications for targeting

specialized accelerators.

2.4 Partial Evaluation

Partial evaluation [JGS93] [Jon95] [BEe87] is a program optimization technique. It

is used to specialize source code with respect to part of the program input. This

technique is usually applied during compilation of a source program to generate ef-
ficient object code. The result of partial evaluation is optimized code that has a reduced number of computations and can thus run faster. In this dissertation, partial evaluation is used to exploit intrinsic parallelism in application programs during compilation and automatically create optimized designs of customized accelerators for specific programs.

This section provides a brief introduction to the terminology of partial evaluation.

2.4.1 Principles of Partial Evaluation

Partial evaluation, also referred as program specialization, is a transformation technique for optimizing a source program and generating a new version of the program. The transformation maintains the functional equivalence between the original program and transformed version of the code, but the latter is specialized with respect to part of the program input and often has a simplified code structure and reduced computation. The program transformation is completed through symbolically executing the program, unrolling loops, inlining subroutine calls, and reducing computations of expressions using the known input values. By applying partial evaluation to the source program, some variables are bound to concrete values, computable expressions and control conditions are evaluated, and many redundant computations are eliminated.

During partial evaluation, variables in the program may be classified into three categories: dynamic, statically computable, and constant. A dynamic variable can only be evaluated at run time, a static computable variable inside a loop can be bound to concrete values in all the loop iterations when the partial evaluation tool unrolls the loop, and a constant variable always holds the same value.

Before defining partial evaluation, we introduce the notation, $[Prog](InpList)$, to represent the meaning of a program. $Prog$ in the notation is a source program, $InpList$ is the program input, and notation $[Prog](InpList)$ represents the result of executing the program using the input. The result is not restricted only to be in the form of data values, but can be a program as well. For example, the notation can be used to represent an object program generated by compiling a source program. The compiler
Definition 2.1: Let \( p \) be a source program, let \((s, d)\) be a complete list of \( p \)'s input, then \( \text{PE} \) is called a transformation program using \textit{partial evaluation} technique iff the equation

\[
[s]([\text{PE}](p, s))(d) = [([\text{PE}](p, s))(d)](d)
\]

is always true.

In the above definition, the notation \([\text{PE}](p, s)\) presents the target program that is generated by \( \text{PE} \) through specializing \( p \) with respect to the partial input \( s \). The right hand side of the equation can be interpreted as the result of executing the target program using the input subset \( d \). Figure 2.6 depicts equivalence of the both sides of the equation in Definition 2.1. The left hand side, which is shown in Figure 2.6(a), can been viewed as a one-step evaluation, and the right hand side, which is shown in Figure 2.6(b), can been seen as a two-step evaluation.

Definition 2.2: In Definition 2.1, the \( \text{PE} \) is called the \textit{partial evaluator}, \( p \) is called the \textit{subject program}, and the target program represented by \([\text{PE}](p, s)\) is called the \textit{residual program}.

The primary goal of applying partial evaluation is to generate an efficient residual
Subject Program: \( p(n, x) \) is a two-input program to compute \( x^n \)

\[
p(n, x) = \\
\text{if } n = 0 \text{ then } 1 \\
\text{else if even}(n) \text{ then } p(n/2, x)^2 \\
\text{else } x \cdot p(n-1, x)
\]

Applying Partial Evaluation: \( \text{PE}(p, 5) = p_5(x) \)

Residual Program: \( p_5(x) \) is a one-input program to compute \( x^5 \)

It is generated by applying partial evaluation: \( \text{PE}(p, n) \) with respect to \( n=5 \)

\[
p_5(x) = x \cdot ((x^2)^2)^2)
\]

Figure 2.7: An Example of Program Specialization

program that can execute faster than its original subject program. Given a subject program together with a part of its input that always has the same value when the program is executed, a partial evaluator can generate a specialized program by eliminating the statically determinable computations using the partial input. Figure 2.7 shows an example how to generate a specialized program using partial evaluation. In this example, \( p \) is the original source code for calculating values of \( x^n \). The program input \( x \) and \( n \) may be bound to any positive integer values. If we use this program to compute values of \( x^5 \), which is really \( x^n \) specialized with respect \( n \) having the constant value 5, the source code can be transformed into a simplified program by eliminating redundant operations in the original program. In the resulting code, conditional branches and recursive function calls have all been removed. Therefore, the residual program can run faster.

Partial evaluation has been used for purposes of program optimization, compiling, and other forms of program generation [B+76] [Jon95] [Goa82]. In recent years, it has been used in compiling scientific programs [BW90]. A compiler was built to produce the specific program for a particular problem from a source program that is made for a class of applications by using partial evaluation. The compiler targets numerically oriented scientific problems. The purpose of using partial evaluation in
the compiler is to optimize the source program, so that the resulting code can execute efficiently for the particular problem. For example, the compiler can transform a program created for the n-body problem (involves computing the trajectories of the a collection of n particles that exert forces on each other) into a specialized program to solve the nine-body problem, which is of particular interest in astronomy. By applying partial evaluation, the compiler produces the specialized program with reduced computations and exposed parallelism among the computations. In our research, we use partial evaluation for not only reducing operations in source program, but also generating specialized hardware and control software of the customized accelerator for the given program.

Partial evaluation has also been applied to hardware descriptions in designing dynamic reconfigurable systems on FPGAs [SHM96]. This research effort was to investigate the potential method to change the structure of the circuit that is implemented on FPGAs at run-time. The assumed run-time system consists of a microprocessor and FPGAs that are used to implement circuits for applications. The microprocessor performs partial evaluation to the netlists of the circuits implemented on the FPGAs with respect to the current input conditions and status of the circuits, and generates optimized netlists for reconfiguring the FPGAs at run-time. This approach requires the CAD tools support the implementations of the run-time reconfigurable circuits, and also needs a suitable hardware description language to express possible reconfigurations of application circuits. In this research, partial evaluation is only applied to descriptions of hardware designs.

This dissertation explores methods for using partial evaluation in automated reconfigurable compute accelerator designs on FCCMs. A partial evaluator that is designed for this purpose will be described in Chapter 4.

2.4.2 Online and Offline Partial Evaluation

Partial evaluation is an automatic optimization process. Whether or not to conduct a specialization to a particular part of the source code is determined by the partial evaluator during program transformation. The decisions are made based on a general
control strategy that is employed by each partial evaluator. Two kinds of different strategies, online partial evaluation and offline partial evaluation, are commonly used to build partial evaluators. An online partial evaluator decides if an expression can be computed by checking if all involved variables are bound to concrete values, which either are part of static input or derived from earlier evaluations. An offline partial evaluator makes decisions based on a set of conditions that are independent of values of variables in subject programs. The conditions can be specified using annotations in the source code or driven from running status of the program specialization.

**Definition 2.3:** A strategy is called online if the concrete values computed during partial evaluation can affect the choice of action taken. Otherwise the strategy is called offline.

Online partial evaluation determines each individual computation to be either executed at partial evaluation time or executed in the residual program at specialization time. It usually accomplishes the program transformation in a single phase. This is because the online strategy can exhaustively eliminate static computations in the subject programs. This kind of partial evaluator always tries to execute all the operations in subject programs and only gives up on those that cannot be finished because of lacking values of the operands.

Offline partial evaluation usually uses an extra phase to preprocess subject programs before applying specializations to the programs. During the preprocessing phase, the partial evaluator analyzes the source program and decides whether each segment of computation in the source code is static or dynamic. The collected information is then used during the second phase, which is program specialization phase, by the offline partial evaluator to determine which computations should be evaluated. The technique used for analyzing source code in the preprocessing phase is called binding-time analysis, which will be described in Section 2.4.3. Comparing online and offline strategies, an online partial evaluator in general can produce more efficient specialized programs, but the time cost on the specialization is much longer. When source programs contain recursive function calls and loops that have large numbers of iterations, an online partial evaluator may run into overflow conditions.
On the other hand, an offline partial evaluator is much easier to build and runs faster without a risk of overflow.

### 2.4.3 Binding-Time Analysis

When choosing the offline strategy, a partial evaluator requires that programmers annotate explicitly which part of an original program should be specialized and when the program specialization should be terminated. In practice, user annotations for original programs can be very complicated, especially in programs with structures composed of nested loops, conditional branches, and subroutine calls. Consequently, many static operations, which can potentially be precomputed in a subject program, may slip into the residual program after the program specialization.

An explicit annotation in a subject program for offline partial evaluation can be automatically generated by a software tool based on the static input of the subject program. This software tool uses a technique called binding-time analysis (BTA). Before applying partial evaluation, BTA traverses the subject program, binds the initial static input values to the associated variables, examines expressions of computations and control flows, determines whether each variable is static or dynamic, and generate all the annotations in the subject program. With the annotations marked by BTA, offline partial evaluation can exploit all the static computations.

### 2.5 Objectives and Goals

This dissertation explores a general methodology for implementing customized compute accelerators on FCCM platforms for computationally intensive application programs. It describes a design flow using application independent algorithms that can find special functional requirements of a given program and automatically generate a customized hardware design of the accelerator. The design flow will be used to construct a compiler to produce both the hardware configuration and control software of the FCCM for each application through analyzing the source program.

The primary goal of our research is design automation. The approach is to create
an abstract architecture to describe reconfigurable hardware accelerators that may be mapped on FCCMs and to build a compiler to generate hardware designs for functional units of the architecture during compilation of each source program. To use the same architecture to model different accelerators customized for individual applications, a parameterized model is adopted to describe detailed hardware specifications of the accelerators. In the early stages of compilation, the compiler examines operations in the source program, finds functional requirements of the application, determines the values of the architectural parameters, and generates a concrete machine model using the parameter values. The resulting machine model, along with functional descriptions of hardware units, will be used by synthesis tools to construct a customized accelerator on the FCCM. This machine model is also needed in the later stages of compilation for scheduling computations on the implemented accelerator.

Our compiler is designed to target computationally intensive application programs that contain irregular computations, but which has statically determinable variables that can be used by the compiler to transform the source program into a specialized code by applying partial evaluation. During the transformation, the compiler will optimize the source code, exploit inherent parallelism among computations in the code, and find the operation patterns of the computations. From the set of patterns, the compiler will further generate the design descriptions of hardware units to be mapped onto the target FCCM and a precise machine model of the customized accelerator. The descriptions and the model will be used by synthesis tools to produce the configuration of the FCCM. The machine model will also be used by the compiler to generate the control software to run on the configured accelerator.

2.6 Summary

In this chapter, we have described the construction of FCCMs, and examined three categories of FCCMs that are used to implement special-purpose processors, coprocessors, and attached processors, respectively. In Section 2.2, we have then reviewed four examples of FCCM designs, and discussed the design methods for implementing
specialized processors on the FCCMs for application programs. In Section 2.3, we have also reviewed two previous designs of hardware accelerators. In Section 2.4, we have briefly introduced a program transformation technique, partial evaluation, that is the primary technique used in this dissertation. Finally, in Section 2.5, we have described the objectives and goals of this dissertation.
Chapter 3

The Architecture and Design Methodology of Field-Programmable Compute Accelerators

This chapter presents the framework of our research work. The automated compilation of the hardware design is based on a parameterized architecture of specialized accelerators that are implemented on FCCM platforms. First we introduce the architecture and describe the parameters that are used to characterize hardware constructions of different accelerators. Then we propose a methodology to generate the design of the accelerator that is specialized for each given application program. To achieve design automation, Section 3.3 describes design considerations for our compiler, PECompiler, that was built to apply the design methodology during compilation of a source program.

3.1 General Design Approach

The research described in this dissertation is aimed at creating a compiler to automatically design field-programmable compute accelerators (attached processors) to
speed up executions of application programs. The primary goal of the compiler is to use hardware resources and the configurability of FCCM platforms to increase the speed of program executions.

To generate the accelerator design, the compiler first divides a source program into two parts according to user annotations in the program. These are a main program to run on the host and a set of subroutines to run on the FCCM. Since conditional branches can severely degrade the performance on the accelerator implemented on the FCCM, the compiler creates subroutines that have a static execution sequence, and places the operations of determining major loop conditions into the main program together with the initializations and data I/O operations. Achieving a significant speedup requires that most of the conditional code be eliminated using partial evaluation. Figure 3.1 provides an overview of the design approach.

According to Amdahl's Law [HP90], the performance improvement to be obtained by improving some portion of a computer system for faster program execution is limited by the fraction of the time the faster mode can be used. Assume that $x$ is the fraction of execution time for computational kernels in a program, then for the overall program the relationship, $\text{speedup} < \frac{1}{1-x}$, is always true. This means that to achieve significant performance improvement, the value of $x$ needs to be close to 100%. When implementing field-programmable compute accelerators for application programs, we set the goal of mapping the computations in each program that take at least 90% of program execution time on a conventional CPU onto the accelerator.

Computationally intensive programs consist of segments that are initialization functions, data input/output, and major loops that contain computational kernels. Computational kernels represent operations that occur repeatedly in the execution flow of the programs and account for the vast majority of execution time on a conventional CPU. As shown in Figure 3.1(a), during execution of the program, most of the execution time for the program is spent in the kernel parts of the program, and only a small portion of execution time is spent in other parts of the program. In the figure, the darker blocks represent the segments of computational kernels in a source program, and the lighter blocks represent remaining segments. Figure 3.1(b) shows
the approach to accelerate the program execution using the FCCM platform.

Unlike other design methods that only map regular patterns of program segments to the FCCM with fixed control sequences, our compiler can handle irregular patterns of computations in the source code. The advantage of this approach can be seen by an example. Consider matrix inversion of sparse matrices, but with the property that the fill pattern of the matrix is known in advance, although the matrix entries are not known. In a fully hardwired accelerator that did not take advantage of the static fill pattern, considerable logic would be devoted to address generation and control logic. By using partial evaluation and an instruction-oriented model, the accelerator imple-
mented on the FCCM would contain an arithmetic datapath, and executes a fixed sequence of operations that contain precomputed addressing information. This can be efficiently implemented using instructions on the accelerator to specify addresses that directly reference the array elements. The intrinsic parallelism that is available can easily be exploited via knowledge of the fill pattern of the matrix. Our compiler achieves the goal of exploiting parallelism by applying partial evaluation to the source code using part of the program input. PECompiler has three key goals:

- To determine the functional requirements from the source program for the specialized hardware processing units on the FCCM.

- To design specialized hardware structures and functional units of customized computers for programs containing irregular patterns of computations.

- To exploit the intrinsic parallelism in programs with complex algorithms and generate instructions to schedule operations to execute on multiple functional units in parallel.

### 3.2 Architecture of Configurable Accelerators

Our compiler uses an abstract architecture and a parameterized hardware model to create customized accelerators for a variety of applications. The particular functionality of any given accelerator that is implemented on the FCCM for a given application program is characterized by parameters of the architecture. The architecture is designed to meet the needs of two primary goals. The first goal is to parameterize specialized requirements of different accelerators that are customized for a variety of applications, so that a compiler can be built to generate the hardware design for each of the accelerators. The second goal is to ensure that accelerators designed based on the architecture can be implemented on FPGAs. Methods that are used to implement various architectural components on FPGAs have direct impact on the way that architecture is determined. Section 3.2.1 presents the architecture that we designed for the configurable accelerators and the parameterized machine model.
3.2.1 An Abstract Architecture

The architecture of the accelerators comprises a set of components:

- a two-level hierarchical interconnect network
- functional units that are specialized for individual applications;
- register files;
- data memory blocks;
- buffers at the input/output ports of the network;
- control units to execute instruction routines on the accelerator.

Figure 3.2 shows the architecture at the top-level. It consists of a number of pro-

processing modules (PMs) and a top level interconnection network that provides data
connections among the PMs. The solid lines connecting the network and every PM
represents the data links used by the FCCM during execution. The shaded lines are
a bus between the host and the accelerator.

Each PM is a slice of the processor within the VLIW machine, and all the PMs
in the machine simultaneously execute a section of a single wide instruction.
Figure 3.3 shows the structure of the processing modules. A single PM consists of several functional units (FUs), storage units (SUs) that are register files and data memory blocks, and a PM-level interconnection block that transfers data among the FUs and SUs. Each PM may contain a different number and kinds of FUs and SUs compared to others. To achieve high performance, pipelined structures are used to implement the circuits of datapaths in all the PMs. There is also a control unit that consists of an instruction memory and control logic in every PM. The instruction memory stores all the instructions to be executed on the PM, the control logic decodes the instructions and controls all the hardware units of the PM. The control unit is a parameterized logic circuit designed as part of the porting process for a particular FCCM, with the specific parameters determined by PECompiler for each application. Every PM contains a program counter (PC), but the PCs across multiple PMs always point at the same address. Both data and instruction memory blocks are usually 32 or 64 bits wide. The specific configurations of the memory blocks for each application
also depend on the available memory resources on the given FCCM platform.

One problem with using this architecture is that each FU may need to load many operands that are stored in the register files before executing an operation, but each register file can only provide a limited number of ports for accessing data in the registers. When the register files cannot provide enough ports for an FU to fetch all the operands at the same cycle, extra cycles will be used to finish fetching the operands, and the issue rate of the operations on the FU will be reduced. As the register files are implemented using on-chip RAM or lookup tables, the number of ports that can be implemented in the register files is limited by the available hardware resources (details will be discussed in Chapter 7). For a given number of ports on the register files, data buffers are placed at the input/output ports of the PM-level interconnection block to improve the issue rate of operations on the FUs. Since they are placed at the input/output ports of the interconnection block, these buffers are called port buffers in this dissertation. Figure 3.4 shows an example of using port buffers to overlap the operand fetches of two operations executed on an FU. The three operands of operation \( a \) are \( a_{\text{data}1} \), \( a_{\text{data}2} \), and \( a_{\text{data}3} \). Assume that both \( a_{\text{data}1} \) and \( a_{\text{data}2} \) are stored in the same register file (\texttt{Regf.1}) and \( a_{\text{data}3} \) is stored in \texttt{Regf.3}. Since \texttt{Regf.1} has a single read port, it needs two cycles to fetch two of the operands, \( a_{\text{data}1} \) and \( a_{\text{data}2} \), while \( a_{\text{data}3} \) can be fetched during the first cycle.
To avoid the FU being occupied by one operation for two cycles, three port buffers are arranged to load operands before starting an operation on the FU. As shown in the figure, \texttt{a.data1} and \texttt{a.data3} are loaded into \texttt{Buffer.1} and \texttt{Buffer.3}, respectively, at cycle \texttt{i}, and \texttt{a.data2} is loaded into \texttt{Buffer.2} at cycle \texttt{i+1}. Operation \texttt{a} can start on the FU at cycle \texttt{i+2}. If operation \texttt{b} is also scheduled to run on the FU at the time, fetching operands can be interleaved for the two operations. Assume that \texttt{b.data1} is in \texttt{Regf.1}, \texttt{b.data2} is in \texttt{Regf.2}, and \texttt{b.data3} is in \texttt{Regf.3}. Then \texttt{b.data1} can be loaded into \texttt{Buffer.1} at cycle \texttt{i+2}, \texttt{b.data2} can be loaded into \texttt{Buffer.2} at cycle \texttt{i}, and \texttt{b.data3} can be loaded into \texttt{Buffer.3} at cycle \texttt{i+1}. Consequently, the operation \texttt{b} can start from cycle \texttt{i+3}. It is still possible to load operands for other operations, which are to be executed on the same FU, into the three buffers between cycles \texttt{i} and \texttt{i+2}, so that the issue rate of operations on the FU can be further increased.

Figure 3.5 shows the logic circuit of a port buffer that consists of four entries. It can be seen as a random-in-serial-out (RISO) shift register. Data items in the buffer move one stage ahead every clock cycle. An input data to the buffer can be directly latched at any chosen stage. Three select signals \texttt{S1}, \texttt{S2}, and \texttt{S3} can be generated based on two instruction bits, and at any time at most one of them is set to allow the input to be latched in the associated stage. If none of the signals is set, stage \texttt{L0} latches the input data. The choice of a particular stage to latch the input also determines the number of cycles that the data item can stay in the buffer. For example, when the select signal \texttt{S1} is set before cycle \texttt{i}, then the input data is latched by stage \texttt{L1} at cycle \texttt{i} and can stay in the buffer for three cycles. At cycle \texttt{i+2} the data is shifted into \texttt{L3} and can be fetched to the FU input.
3.2.2 Parameterized Machine Model

With the abstract architecture that is described in Section 3.2.1, the design task for a customized accelerator is to determine the set of FU designs, the numbers of PMs, FUs, and SUs, and generate the code that is to be executed. For the compiler to use the architecture in automated design, a parameterized hardware model is needed to describe the detailed hardware specifications of each accelerator created for an application program. The parameters include:

- the number of PMs configured in the customized accelerator;
- connections between the top-level network and each PM;
- the internal structure of the network;
- sizes of all the port buffers
- the total numbers of FUs and SUs inside each PM;
- the number of input/output ports on each FU;
- sizes of all the SUs;
- connections among the FUs and SUs within the same PM;
- the pipeline latency on every hardware unit.

The synthesis task for our compiler is to determine the set of FUs that are required to execute the program operations. Since the computations needed to run on the FCCM are organized as a set of subroutines (as described in Section 3.1), the compiler determines the functions of all the FUs when compiling these subroutines and generates VHDL descriptions of the FUs. After the compiler generates the designs of the FUs, users can determine the total number of FUs that can be arranged in the VLIW machine and a subset of FUs for each PM based on the information of given hardware resources on the FCCM, which includes the number of FPGA chips, number of lookup tables on each chip, available memory blocks, and the interconnection topology among all the chips. The description of the available hardware resources is created once for each unique FCCM platform. After customizing the structure
of the accelerator, the compiler schedules computations to execute on the accelerator and generates instructions for all the PMs to control the operations on the FUs. Operations executed on all the FUs are statically scheduled.

3.2.3 The Time-Stationary Control Strategy

Pipeline control strategies can typically be categorized as either time-stationary or data-stationary [Kog81]. Figure 3.6 shows machines constructed using the two control strategies respectively. In a data-stationary machine, each control word in an instruction specifies the control information for all of the operations applied to a single piece of data. This is commonly used in conventional CPUs. Less common is the time-stationary approach, in which a control word in an instruction applies to a single point in time, and hence may specify the control for many different pieces of data. It is difficult to generate code for a time-stationary machine executing a

![Figure 3.6: Data-Stationary and Time-Stationary Machines](image)
conventional program because the presence of branches makes it hard to determine what the execution trace through the program will be. Code generation for data stationary machines is easier, but requires additional pipeline registers for the delayed control information. Since PECompiler eliminates almost all branches for the code segments that are to be executed on the FCCM, it can use the less expensive time-stationary approach to schedule the operations and generate VLIW instructions for the customized accelerator.

3.3 Overview of the Compiler Design

A specific accelerator design on an FCCM platform is determined by the functional requirements of a given application program and available hardware resources that can be used to build the accelerator. Using the parameterized machine model, the same compiler can even generate instruction routines for different specialized accelerators that are implemented on the FCCM platform. This section discusses the design approach of the compiler.

3.3.1 Design Considerations

Before describing the compiler design, this subsection first examines the tasks that are needed to be accomplished by the compiler. Figure 3.7 shows the general function of the compiler. The input to the compiler includes the source program and hardware parameters of the targeted FCCM platform. The output produced by the compiler consists of VHDL descriptions for the hardware functional units of the specialized accelerator, instruction sequences to run on the accelerator, and the main program to run on the host machine.

To fully take advantage of the configurable hardware, the compiler is required to determine the program operations that should be executed using specialized hardware units, decide what kind of hardware functional units should be implemented on the FCCM platform, and obtain the maximum utilization of all the hardware units on the accelerator for executing as many of the operations as possible in parallel.
The compiler accomplishes the tasks in two stages. The first is source code preprocessing, and the second is operation scheduling. In the first stage, the compiler preprocesses the source program using partial evaluation to discover the special functions required by the program. Through analyzing the source code, the compiler can decide which segments of the source code may be accelerated on the FCCM machine and which should execute on the host machine. For the code segments to be executed on the FCCM, the compiler can also find out what custom functions should be implemented in hardware on the accelerator. One specific task of the compiler at this phase is to produce designs of hardware units for the required custom functions.
and generate VHDL descriptions of the designs. The VHDL code will be used by synthesis tools to generate the configuration of the customized accelerator.

After synthesis tools generate the configuration of the FCCM platform, a set of architectural parameters of the implemented accelerator are read into the compiler and used as an input to the second stage of the compiler. These parameters are used to create a precise machine model of the specialized accelerator. The model is further used by the compiler to generate instruction routines for controlling the execution of the computations on the accelerator.

In rest of this section, we first introduce a graph representation of program operations, which is used inside PECompiler, and then describe a syntax tree that is used by the compiler to represent program structures.

### 3.3.2 Data Dependency Graph

Since the computations are expressed in a sequential order in source programs, the compiler has to analyze the code expressions, determine sequences that data operations may be performed, and search for as many operations as possible to be executed at each cycle time. To exploit parallelism in an application program, the compiler at first needs to examine data relationships among all the operations and record the information in a format that can be easily retrieved during code analysis.

In this dissertation, a directed graph, which is called data dependency graph (DDG), is used to represent the relationship between operations. Each node in this graph can represent either an operation that can be performed by an FU or a data memory access. Every edge in the DDG represents a data value that is used as an operand by the node to which the edge points. The source end of the edge is either assigned with a constant or connected to an output port of a node that produces the value. The number of input ports on a DDG node shows the total number of operands that are required to conduct the operation(s) of the node, and the number of node output ports shows the total resulting data items that are produced by the node operation. Every single output port on a DDG node may be connected with any number of outward edges, with each of the edges connecting to an input port.
of another DDG node, but every input port on a DDG node may only be connected to one inward edge. The source of the inward edge can be either an output port of another DDG node or a constant value. A directed edge from a source node to a target node indicates that the target node is dependent on the source node.

There are two kinds of special DDG nodes. One of them has no input port, and represents the read of the value of a scalar variable. The other special node has no output port, and represents an assignment to a variable. To describe the data dependency between an earlier write node to a variable and a later read node from the same variable, a dashed directed edge is linked from the write node to the read node. Unlike other edges in the DDG, however, a dashed edge cannot be used to carry a data value. It simply indicates an execution order between the two linked nodes.

Figure 3.8 presents an example data dependency graph for a fragment of a source program. Figure 3.8(a) presents a fragment of a source program. Figure 3.8(b) shows the DDG that represents the operations, as well as the dependency information, of the code fragment. In the DDG, the node name is marked beside each node and the function is given inside every node. Figure 3.8(c) shows the structure of node f_1 that
appears in the DDG. The operations included in the $f.1$ are always implemented by an FU as a single operation with four source operands and two results.

When scheduling all the node operations in a DDG to run on a customized accelerator, the compiler can determine the execution sequence of the nodes based on the dependencies between the nodes. A DDG node is ready to be scheduled at a clock cycle when all its dependent nodes have finished their executions. During the scheduling process, the compiler will search for all the ready nodes at each machine cycle and choose the most urgent ones to run on the hardware units that are available at the time. Once a node is scheduled, the edges that connect from its output ports to the input ports of the dependent nodes are assigned with concrete values. A node becomes ready after all its inward edges at its input ports carry concrete values.

### 3.3.3 Source Code Structures and Syntax Tree

A subset of the C programming language is chosen as a source program language. Each source program is converted into an internal format before it is compiled. By using this format, we were able to build a parser for the language in a short time. Appendix A describes the subset of the language and internal format.

Once the source code is parsed by the compiler, a syntax tree, which is an internal format of the source program, is created to represent the program structures. Figure 3.9 shows the syntax tree construction. The format of the syntax tree introduced in this dissertation is similar to what is described in [ASU88].

The syntax tree representations for different kinds of code blocks are recursively used to describe structures of source programs. At the top level, the root of a syntax tree represents a given application program. Every child of the root corresponds to a program function or a set of declarations for global variables. Figure 3.9(a) shows the top-level syntax tree. For every application, there must be one child node dedicated to the main() function. Other child nodes are optional. A program function can be decomposed into a number of code blocks. The program function corresponding to each of the child nodes can also be represented by a syntax tree. Figure 3.9(b) shows the syntax tree of a program function.
For our subset of the C programming language, six major code blocks are defined. They are `decl.block`, `basic.block`, `if.block`, `call.block`, `for.block`, and `while.block`.

A `decl.block` may appear either in the top-level syntax tree as a child node for declaring global variables or in the second-level syntax tree for declaring local variables for a program function. A `basic.block` presents a segment of straight-line source code that contains computations and assignments in a fixed order of execution. It appears as a leaf node in syntax trees of program functions. As described in Section 3.3.2, a DDG can be used to represent operations in a piece of source code. In a syntax tree, operations inside each leaf node can be represented by a DDG. For a leaf node of type `basic.block`, Figure 3.8 shows an example of a DDG.

An `if.block` represents a segment of source code that contains a conditional expression, and two alternative sets of code blocks that need to be executed when the condition is either true or false, respectively. Figure 3.9(c) shows the sub-tree of the `if.block`.

In a similar way, a `call.block` is represented by a sub-tree shown in Figure 3.9(d). The sub-tree has three child nodes. The first child node represents operations that calculates values of parameters and passes them to the callee (a subroutine), the second node represents operations of the subroutine, and the third node passes the
return value to the caller.

Figure 3.9(e) shows the sub-tree of a for_block. It has four child nodes to represent the loop initialization, the condition for continuing iterations of the loop, the loop body, and the changes on loop variables made after each iteration, respectively. A while_block is a special case of the for_block when the first and the last child node do not contain any operation.

To create a syntax tree representation for a program, a sub-tree may be recursively used to represent a node in another sub-tree. For example, the node of code blocks in Figure 3.9(e) can be replaced by the sub-tree shown in Figure 3.9(b).

3.4 Summary

In this chapter, we have first described our general approach to implement field-programmable compute accelerators on FCCM platforms for application programs. We have then defined an abstract architecture and a parameterized hardware model for the accelerators. At last, we have proposed a new methodology for designing the customized accelerator for each given application program, and designed a compiler to apply the methodology during the compilation of each source program. We have introduced the DDG and syntax tree that are used by the compiler as an internal format of the source program. The construction of the compiler, as well as detailed design issues, will be described in Chapter 4 and Chapter 5.
Chapter 4

Automated Hardware Design

This chapter starts to describe the design of PECompiler. It first presents the construction of the compiler, and then focuses on how the compiler accomplishes the task of automated hardware design during compilation of the source program. PECompiler derives the hardware design of customized accelerators based on the requirements of targeted programs. To generate the optimized design, the compiler uses partial evaluation to analyze each source program, organize static traces of operations that are to be executed on the accelerator, and create descriptions of the specialized functional units that are required by the operations. This chapter also describes how the compiler creates a graph to represent the computations in the program that need to be executed on the FCCM. This graph is used by PECompiler to analyze data dependencies among computations during the compilation of each source program.

4.1 Construction of the Compiler

Figure 4.1 shows the structure of PECompiler. The source programs are written in a restricted subset of the C programming language. The compiler consists of five major blocks: Partial Evaluator, DDG Generator, Post-Map Optimizer, Machine Model Generator, and Operation Scheduler.

In the first step, PECompiler reads in the source code, and generates a syntax tree to represent the code structure and the operations in the program. Each inner
node of the syntax tree represents a code block, such as a for loop or if statement, and the leaf nodes represent arithmetic/logic operations or the reading/writing of variables. The compiler requires programmers to annotate which code blocks can be partially evaluated during compilation and which code blocks should be implemented using hardware resources. The problem of automatically selecting which parts of the application to accelerate is difficult, and is not addressed by our system. Then, PECompiler uses the Partial Evaluator to analyze data dependencies among operations and optimize the structures of the code blocks through traversing the syntax tree. As the result of this step, all the code segments that can be accelerated on the FCCM are organized as flattened traces of operations, and the remaining code is left in a main program to run on the host computer. This main program uses special hardware function calls to invoke the execution of the flattened traces on the FCCM. Each of the traces is seen by the main program as a discrete hardware subroutine.
The second step is to use a data dependency graph (DDG) to represent the operations in each flattened trace that is generated in the first step. It also determines what kinds of functional units (FUs) need to be implemented on the FCCM for executing the operations in the flattened traces that are derived in the previous step. The compiler finds the common patterns (described in Section 4.4) of operations in the traces and uses the patterns as the structures of the FUs. After determining the FU designs, clusters of operations in the flattened traces that are covered by the common patterns are replaced by new single operations implemented by the FUs. PECompiler also creates VHDL descriptions for all the FU designs. The VHDL code can be used by synthesis tools to map the FUs on the FCCM.

The third step optimizes the DDGs that are generated for the subroutines to run on the FCCM. It reduces memory accesses by removing redundant operations for reading/writing the same variables, and eliminates false dependencies among DDG nodes (described in Section 4.5.1).

In the fourth step, the Machine Model Generator takes as input a set of architectural parameters, and generates a machine model of the accelerator customized for the source program. The values of these architectural parameters (described in Section 3.2.2) are determined by users based on the information of available hardware resources on the FCCM and the cost to implement the FUs required by the program. This means that users need to manually generate the parameter values. As described in Section 8.3, automated generation of the architectural parameters for a given application program will be pursued as a future research topic.

In this step, PECompiler also creates VHDL descriptions for the Processing Modules (PMs) for synthesis tools to generate the configuration of the accelerator on the FCCM.

In the last step, the compiler uses the Operation Scheduler to schedule the DDG nodes to be executed on the customized accelerator and generate VLIW instruction routines for the optimized DDGs based on the scheduling results, with each routine corresponding to one DDG. These instruction routines are stored in the accelerator before the program starts to run. During the execution, the main program that is
run on the host uses function calls to start execution of these instruction routines.

4.2 The Partial Evaluator

This section describes the partial evaluator that we developed for PECompiler to perform specialization of the source program using part of the input values. The primary goal of applying partial evaluation is to optimize the source program and expose the intrinsic parallelism among computations in the program. Then, based on the specialized source program, the compiler can generate customized hardware units and control software that efficiently uses the implemented units to execute multiple operations simultaneously each clock cycle.

We used the offline strategy in the partial evaluator. As described in Section 2.4, an online partial evaluator tries exhaustively to complete the computations that can be performed at compile time. An online partial evaluator can generate better code than an offline partial evaluator, but the online partial evaluator runs slower than the offline partial evaluator because of the exhaustive effort to specialize the program. As application programs usually contain main loops that run for a large number of iterations, it is difficult to apply online partial evaluation to the programs. Instead, an offline partial evaluator, which relies on user annotations and the technique of binding-time analysis (BTA), was built to optimize the code segments that are to be accelerated on the FCCM.

4.2.1 Application Programming Language

The source code to PECompiler is an annotated program that only describes the computation of the application. The compiler can generate both the specialized hardware design of the accelerator and instructions to run on the accelerator. To realize automated design of the customized hardware, PECompiler first needs to determine the partition between the operations to be running on the host computer and the operations to be accelerated on the FCCM. The compiler makes the decision based on user annotations that indicate specific segments of the source program that
should be executed on the FCCM. According to user annotations, the compiler can organize hardware routines for code segments to be run on the FCCM.

### 4.2.2 User Annotations

Three kinds of user annotations for partial evaluation are supported by PECompiler. The first one is to specify the code segments that need to be compiled into hardware routines for running on the FCCM. HF (stands for hardware function) is the keyword for this kind of annotation, which is placed at the starting point of the annotated segment and a pair of '{' and '}' are used to enclose the segment. The second annotation allows the programmer to further specify the part of a hardware routine segment to be mapped onto specialized hardware units. The keyword for this kind of annotation is HwImp, and a pair of '{' and '}' are also used to enclose the block of code. The third annotation is to indicate a code segment to be partially evaluated. Keyword, PE, and a pair of '{' and '}' are used to delimit the block. Figure 4.2 gives an example of the application program with user annotations. Figure 4.2(a) shows the structures of the annotations in the source program. The keyword PE is used to annotate the code blocks that can have partial evaluation applied during compilation,
and these blocks are enclosed by a dashed line. The keyword HF is used to annotate the code blocks that need to be run on the FCCM, and these blocks are inside the shaded area. Within these blocks, two segments, which are represented by two white rectangles, are further marked by the keyword HwImp for the compiler to map the computations onto specialized hardware units. Figure 4.2(b) shows the syntax tree representation of the program.

By annotating the code blocks using the keyword HF, programmers separate a source program into two portions: one portion (P1) contains all the blocks that are annotated using HF, and the computations in these blocks will be performed on the FCCM; the other portion (P2) contains all the operations that will be executed on the host computer. The P1 portion will be converted to one or multiple hardware routines to run on the customized accelerator that is implemented on the FCCM for the source program. When running the program on the system comprising a host computer and an FCCM, the P2 portion executes on the host and invokes executions of the hardware routine(s) on the accelerator. To achieve high program performance, programmers should not only include the computations in the program into P1 as much as possible, but also reduce the data communications between P1 and P2 (data transfers between the host and the FCCM). To further improve parallelism among the operations in the P1 partition, programmers should explicitly use statically determinable indices for data arrays together with the input as much of P1 as possible. This will allow PECompiler to reduce the data dependencies caused by memory accesses, and schedule more operations to be executed on the FCCM at the same time.

4.2.3 Deriving Variable Status Using BTA

The efficiency of the offline partial evaluator relies on binding-time analysis (BTA). The main task of our BTA tool is to determine the status of variables in all the code blocks by using part of the program input and user annotations. Using the syntax tree constructed for the source code, the BTA tool uses depth-first traversal [ASU88] to trace the definitions on all the variables inside the tree nodes. For a tree node without any user annotation, the BTA tool labels as dynamic the variables that are
assigned with new values inside the code block corresponding to the node. When the node is the root of a subtree, all the variables that are written inside the nodes of the subtree are dynamic to the rest of the program. For a node that is annotated with the keyword PE, the BTA tool tries to complete the computations inside the code block that corresponds to the annotated node using the available part of the program input. If the node is the root of a subtree, the BTA tool recursively applies partial evaluation to all the nodes included in the subtree. During the binding-time analysis, PECompiler also records all the identifiers of the nodes that are annotated using the keyword HF. These nodes will be processed after the binding-time analysis.

For example, when applying partial evaluation to the program shown in Figure 4.2, PECompiler first uses the BTA tool to traverse the syntax tree. The BTA tool can determine the variable status in the nodes that are not annotated and use the given part of the program input to derive the values of variables in the nodes that are enclosed in the PE subtree. After finishing the BTA for the whole syntax tree, the compiler applies partial evaluation to the nodes that are annotated with the keyword HF.

4.2.4 Data Dependency Graph Generation

As described in Section 3.3, data dependency graphs (DDGs) are used to represent computations in basic blocks of the source program. The basic blocks are included in other code blocks, and always appear as leaf nodes in the syntax tree of the whole program. For example, a basic block can be the loop body of a for statement. When applying partial evaluation to the for statement, PECompiler may unroll the loop into a flattened trace of operations, and derive a single DDG that contains the DDG nodes needed to represent operations in all the loop iterations. During the compilation of each source program, PECompiler creates two versions of DDGs. Before applying partial evaluation, the compiler creates the first version of DDGs for all the leaf nodes in the syntax tree of the source program. These DDGs are called template DDGs, and the nodes of template DDGs are called template nodes. After applying partial evaluation to the source program, PECompiler generates the second version of DDGs,
main()
{
    int A[4], B[5];
    int i;
    int c, v;
    int x;
    i = 0;
    i < 4; i++;
    x = A[i] * c;
    B[i+1] = x - v;
}

Figure 4.3: Deriving the Flattened DDG for a Loop Statement

which are resulting DDGs derived from the template DDGs. Each of the resulting
DDGs represents a flattened trace of operations that are to be executed on the FCCM.
This trace may be organized as a hardware routine that will be called to run on the
FCCM by the main program running on the host machine. Figure 4.3 presents an
example of how the compiler derives a flattened DDG for a for loop. Figure 4.3(a)
shows the code block of a for statement in the source code. Figure 4.3(b) shows the
syntax subtree of the code block. Figure 4.3(c) shows the template DDG created
for the loop body. This template DDG is used for generating a flattened DDG when
PECompiler unrolls the loop. The loop is repeated for four iterations before it finishes
execution. For each of the iterations, the compiler first tries to perform the operation on each template node. If the operation associated with a template node cannot be executed by the partial evaluator, the compiler maps the template node onto a node in the resulting DDG. Since variables c and v in the loop are bound to the same constant values (2 and 3, respectively) for all iterations, both nodes tn.3 and tn.6 can be eliminated. The variable x is assumed to be used only within the loop allowing nodes tn.5 and tn.7 to be eliminated. Although variable i changes each iteration, it can always be bound to a static value. Therefore, nodes tn.1 and tn.9 can be replaced by constant values. During the process of partial evaluation, PECompiler creates a table to trace values of all the variables in the program. Figure 4.3(d) shows the resulting DDG. It contains sixteen nodes in total. During the four iterations, node tn.2 in the template maps into nodes n.1, n.2, n.3, and n.4 in the resulting DDG; node tn.4 in the template maps into nodes n.5, n.6, n.7, and n.8 in the resulting DDG; node tn.8 in the template maps into nodes n.9, n.10, n.11, and n.12; node tn.10 in the template maps into nodes n.13, n.14, n.15, and n.16. In this example, by using partial evaluation, the compiler not only eliminates redundant DDG nodes, but also reduces the number of the data dependencies among the operations. The optimized resulting DDG allows the customized accelerator to exploit the parallelism using multiple functional units.

Basic blocks may also be child nodes in subtrees of other compound code blocks, for example, if statements or subprograms. In the same way, when applying partial evaluation to these code blocks, PECompiler uses the templates of the basic blocks to generate the resulting DDGs.

For simplicity, in the rest of the dissertation, the resulting DDGs are simply called DDGs.

4.3 Conditional Branches

One of the biggest obstacles to high performance on a VLIW machine is conditional branches during the execution of programs. Although PECompiler uses partial eval-
uation to eliminate the conditional branches that are statically determinable during compilation of the program, many branch conditions for if statements can only be decided during execution. To avoid the penalty of these branches, the method of resolved-writes is applied to assignments in clauses of conditional branches. By using this method, PECompiler can generate flattened traces for code segments containing if statements. Figure 4.4 depicts the method used to flatten a conditional branch.

![DDG Diagram](image)

**Figure 4.4: Using Resolved-Write for a Conditional Writing Operation**

Since conditional branches also exist in loops, PECompiler unrolls loops that have the static computable loop conditions at compilation time, and places computations of loop conditions for other loops, especially for the loops involving I/O operations in the main program to run on the host machine.

After loop unrolling and conditional branch flattening, there are many memory access nodes in the DDG that read and write the same variables multiple times. These memory accesses require high bandwidth between the FPGAs and RAM chips in the FCCM to support the parallel execution of the DDG nodes. To gain the highest parallelism in the execution, PECompiler analyzes data dependencies among the read and write operations, and eliminates redundant memory accesses.

### 4.4 Hardware Functional Units

One main task of the hardware design for a customized accelerator is to create the specialized functional units (FUs) needed for executing the traces of operations. These FUs will be used to perform computations inside loop bodies or subroutines that are executed repeatedly during the execution of a program on the FCCM. To achieve high
performance on the FCCM, PECompiler generates an optimized design for each of the FUs by examining all the computations in the source code that need the FU, and determines the hardware design of the FU. This section describes how PECompiler generates hardware designs of the FUs.

4.4.1 Extraction of FU Structures

PECompiler designs FUs based on common patterns of DDG nodes that represent operations in the flattened traces to be run on the FCCM. A common pattern is a structure in a group of nodes that repeatedly appear in the traces. We use an example shown in Figure 4.5 to illustrate the definition of common patterns, and describe how PECompiler determines a common pattern for a code segment and uses it to create an FU. Since the pattern will be implemented in hardware as a single FU, the pattern must not include any node that accesses memory. To create patterns for the FUs, the compiler uses the memory access nodes to separate one set of nodes that are included in a pattern from nodes in other patterns.

Figure 4.5(a) shows a part of a source program containing a loop structure. The loop body is annotated to be mapped onto a hardware unit. During the compilation, PECompiler will unroll the loop and generate a flattened trace of code that contains 100 copies of the loop body. The hardware unit designed for the annotated code will be used by all the copies of code in the flattened trace. Since the compiler will optimize the code of the loop body during partial evaluation, the resulting code for each iteration may be different. To create an optimized design of the FU to execute all the loop iterations, the structure of the FU needs to be able to implement the most general form of computation in the DDG node pattern that represents the loop body in all the iterations. Instead of searching for common patterns in a flattened DDG, which would be a complicated procedure, PECompiler determines all the common patterns appearing in the flattened trace when it applies partial evaluation to the code. Figure 4.5(b) shows the template DDG created for the loop body. PECompiler finishes the design of the FU in three steps. The first step is to find the patterns of DDG nodes for every loop iteration. In the process, the loop body will be executed
A Source Program

\[ s = 1; t = 2; \]
\[ \text{for (i=0; i<100; i++)} \{ \]
\[ \}

(a) A Kernel Segment

(b) The Pattern Template

(c) Patterns After Loop Unrolling

(d) A Common Pattern

(e) The FU for the Pattern

Figure 4.5: Designing a Functional Unit

repeatedly for 100 times. For each iteration, the template DDG of the loop body will be examined. If a node operation cannot be statically executed, the node will be mapped onto the resulting DDG and a mark will be made on the original node in the template. For the annotated loop body shown in the figure, PECompiler can determine that \( vI \) is statically computable, and therefore eliminates the adder node that computes \( vI \) from the resulting patterns. Figure 4.5(c) shows the resulting DDG patterns after unrolling the loop in the source code. In the second step, PECompiler derives the minimized common pattern for each of the pattern templates by simply including the nodes that are marked in the previous step. These common patterns can be used to produce the VHDL descriptions for the FU. Figure 4.5(d) shows the resulting common pattern for the DDG created for the loop body. This pattern will
be implemented using an FU with three inputs and one output. In the third step, \textit{PECompiler} replaces node clusters covered by the resulting patterns with new \textit{fu} nodes. The operation of each \textit{fu} node can be executed by the FU on the accelerator.

### 4.4.2 Generating VHDL Descriptions

The VHDL description of each designed FU is created based on the common pattern of operations that is extracted during partial evaluation. \textit{PECompiler} generates a structural VHDL module for the FU by mapping each of the nodes in the pattern to a component entity [LSU89] and creating connections between DDG nodes represented by the edges in the pattern using signals in the structural VHDL module. For example, the common pattern shown in Figure 4.5(d) needs three kinds of components (a multiplier, an adder, and a subtracter). Therefore, the VHDL module of the FU contains four instantiations of the components (two for the multiplier, one for the adder, and one for the subtracter). To allow the compiler to automatically generate VHDL descriptions of the designed FUs, we have created in advance a library (\texttt{Lib\_VHDL\_For\_C\_Operators}) of VHDL components for all the operators that may be appeared in the source programs. Table 4.1 lists the components in our library. When generating the VHDL module for a given DDG pattern, \textit{PECompiler} can allocate the entities of the component VHDL modules that are associated with the nodes in the pattern, and describes the structure of the FU using the instantiated entities. The compiler also allows programmers to define new operators in the source code by adding VHDL modules into the library, and can map nodes of the new operators in the DDG patterns to the VHDL components.

To allow \textit{PECompiler} to generate VHDL models for FUs that can be mapped on different kinds of FCCM platforms, the VHDL library must be able to support mapping the VHDL models onto the FPGAs that are used to construct the FCCMs. Since \textit{PECompiler} only uses instantiations of entities of the library components to describe the FU structures, the structural VHDL modules generated by \textit{PECompiler} are technology independent and may be compiled by a variety of synthesis tools that can handle compilation of standard VHDL code. The specific circuits needed to
Library: **Lib_VHDL_For_C_Operators**

<table>
<thead>
<tr>
<th>Library Component</th>
<th>Arithmetic/Logic Function</th>
<th>Hardware Cost (LEs* on Altera 10K chips)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_Add.vhd</td>
<td>Integer addition</td>
<td>30 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Sub.vhd</td>
<td>Integer subtraction</td>
<td>30 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_LSL.vhd</td>
<td>Logical shift left</td>
<td>61 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_LSR.vhd</td>
<td>Logical shift right</td>
<td>61 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Mux2To1.vhd</td>
<td>2-to-1 multiplexer</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_BwAnd.vhd</td>
<td>Bitwise AND</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_BwOr.vhd</td>
<td>Bitwise inclusive OR</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_BwXor.vhd</td>
<td>Bitwise exclusive OR</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_BwNot.vhd</td>
<td>Bitwise one's complement</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Equ.vhd</td>
<td>Equal operator</td>
<td>11 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Gt.vhd</td>
<td>Greater than operator</td>
<td>21 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Ge.vhd</td>
<td>Greater or equal operator</td>
<td>21 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Lt.vhd</td>
<td>Less than operator</td>
<td>21 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Le.vhd</td>
<td>Less or equal operator</td>
<td>21 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_And.vhd</td>
<td>Logical AND</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Or.vhd</td>
<td>Logical OR</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_Not.vhd</td>
<td>Logical negation operator</td>
<td>16 LEs (16 bits)</td>
</tr>
<tr>
<td>Comp_FpAddSub.vhd</td>
<td>IEEE floating-point add/sub</td>
<td>450 LEs (32 bits)</td>
</tr>
<tr>
<td>Comp_FpMul.vhd</td>
<td>IEEE floating-point multiplier</td>
<td>1232 LEs (32 bits)</td>
</tr>
<tr>
<td>Comp_FpDivBy1.vhd</td>
<td>IEEE floating-point 1/x</td>
<td>1100 LEs (32 bits)</td>
</tr>
</tbody>
</table>

* LEs: Logic Elements.

Table 4.1: The List of Components in the Library

implement the functions of the FUs are described in the architecture bodies of our VHDL modules. Since FPGA manufacturers usually provide target libraries [KA95] to allow users to use enhanced hardware features (typical examples are fast-carry circuitry, Hard Macros [XIL96], and EAB [ALT96]) on the FPGA devices, the cells from a specific vendor library are invoked in the architecture bodies [LSU89] of the components in our library. When mapping the FU designs onto the FCCM, the structural VHDL modules of the FUs are synthesized together with the architecture
bodies that are created for the specific kind of the FPGAs used in the FCCM.

4.5 DDG Optimization

As we have described in Section 3.3, PECompiler uses the Post-Map Optimizer to optimize the DDG before it generates control software for the node operations in the DDG. The primary goal of the optimization is to exploit parallelism among the node operations.

4.5.1 Removal of False Dependencies in DDGs

Since multiplexer nodes are used to implement resolved-writes in the DDG, they may also appear inside the DDG node patterns that are mapped as specialized FUs for the application program. The use of the multiplexers inside the FUs may cause long chains of false dependencies in the DDG. Figure 4.6(a) shows part of a DDG that contains FU nodes (..., n_i, n_i+1, n_i+2, n_i+3, ...) and other simple nodes, such as n_a, n_b, n_c, n_d, n_e, n_f, n_g, and n_h. For simplicity, we assume that each of the FU nodes only contains a multiplexer and the select line on each multiplexer is static (has a value of either 0 or 1). We also assume the longest data dependency chain in the DDG is formed by all the FU nodes that are linked in a series. To increase the degree of parallelism in the DDG, the Post-Map Optimizer in PECompiler optimizes the DDG by using the concrete values on the port of the FU nodes to remove the redundant edges on the false dependency chain. The main goal of this optimization is to eliminate all the redundant edges. To eliminate these edges, the compiler traverses all the DDG nodes, applies partial evaluation to the operations inside each FU node by using the concrete value at each input port on the FU nodes, and determines redundant edges in the DDG. Figure 4.6(b) shows the resulting DDG after PECompiler finishes the optimization. The long dependency chain has been broken into short pieces. In this example, some FU nodes also can be eliminated by the optimizer. In general, internal structures of FU nodes in a DDG are usually more complicated, and PECompiler may not always be able to eliminate the FU nodes. However the compiler can still optimize
the data dependencies in the DDG.

Figure 4.6: Breaking the False Dependency Chain

Figure 4.7(a) shows part of another DDG that contains an FU node (node_4). The FU contains three internal nodes (nd1, nd2, and nd3), one of which (nd2) is a multiplexer node. The first input port of the FU node, which is connected to the select line of the multiplexer, is bound to the constant value 1. Assume that the same FU is used by other nodes in the DDG, but the first input port on these nodes is not always bound to value 1. Although PECompiler cannot eliminate the FU node, it can still use the constant value on the input port to optimize the data dependencies related to the specific nodes. For this node, the compiler can determine that the second input of the FU is not used because the multiplexer is specified by the select line to use the value generated by internal node nd1. Therefore the edge between nodes node_1 and node_4 in the DDG can be eliminated. Figure 4.7(b) shows the part of the DDG after removing the redundant edge. In the resulting DDG, node node_4 no longer takes the
result of node node_1 as input. Then PECompiler can schedule node_4 more flexibly. When the node is executed on the customized accelerator, loading the operand from a register file for the second input of the FU can be avoided. Furthermore, if the third input port is also bound to a constant value, the compiler can compute the value of the first output and forward it to node node_5. Then the edge between nodes node_4 and node_5 can also be eliminated. After removing the redundant edge, the restriction that node_5 can only be executed after node_4 is no longer valid, and a write to a register file for the first output of the FU node can be avoided. In a similar way, if the first input of the FU node is bound to 0, PECompiler can determine that the third FU input is redundant and remove the edge connecting to the third input.
4.6 Summary

In this chapter, we have first presented the construction of PECompiler. We have then described the design of our partial evaluator. The main issues include input language format, user annotations, binding-time analysis, and data dependency graphs. These issues have been discussed in Section 4.2. We have described how PECompiler uses resolved-writes to flatten conditional branches when generating DDGs. We have described in Section 4.4 how PECompiler derives designs of specialized functional units and generates VHDL descriptions of the designs for the given application program. Finally, we have described an important optimization that can break long dependency chains in the DDG, so that the degree of parallelism among the node operations can be increased. In Chapter 5, we will describe how PECompiler generates schedules of DDG node operations to run on a customized accelerator.
Chapter 5

Instruction Sequence Generation

This chapter describes how PECompiler generates an instruction sequence to control the operations on the customized accelerator. As described earlier, PECompiler creates the design of specialized hardware units and generates optimized DDGs for flattened operation traces of the computational kernels that need to be accelerated on the FCCM. The instruction sequence is derived based on the generated DDG and a given machine model of the customized accelerator that is constructed using the specialized hardware units.

5.1 Overview of Approach

The execution of DDG node operations on a customized compute accelerator is controlled by a VLIW instruction stream executed on all the PMs in the accelerator. To generate the VLIW instruction stream for a DDG, PECompiler needs to select a particular FU to perform the operation for each DDG node and schedule the operation to be executed at a specific clock cycle. The scheduling results of all the DDG nodes can be used to derive an instruction sequence, with each VLIW instruction specifying the operations on all the hardware units on the accelerator at a clock cycle. A scheduler (the Operation Scheduler shown in Figure 4.1) in PECompiler was built to determine the schedules for all the nodes in a given DDG. Since the scheduler was built to target accelerators with different hardware structures, it takes as an input a
machine model that characterizes the architectural specifications of each customized accelerator. The goal of the scheduler is to minimize the number of cycles that are used to finish all the node operations.

The rest of the chapter describes the design of the Operation Scheduler. It also describes the major algorithms that are used in the scheduler to choose candidate nodes and allocate hardware resources to each of the operations. This section starts with the method that is used to generate the instruction sequence.

5.1.1 Scheduling Operations on the Accelerator

The overall VLIW machine is partitioned into a number of PMs, and each PM contains a subset of the components. The FUs and register files within the same PM are called local to that PM, and otherwise are called remote. The accelerator is designed as a load/store machine, so that the FUs can only access local registers for the data operands and results. By using pipelined structures, the accelerator can execute one VLIW instruction every clock cycle.

When scheduling a DDG node to run on the VLIW machine, PECompiler assigns a specific FU to execute the node operation, and allocates SUs to store the data operands and results of the operations. When an operand is in a remote register file, PECompiler schedules a data transfer to copy the data into a local register. There is usually more than one node in the DDG ready for scheduling and more than one FU can be assigned to each node to execute the operation. During scheduling, PECompiler picks one ready node in the DDG at a time and chooses a suitable FU and the earliest feasible cycle time to schedule the node operation. PECompiler uses a delay model (described in Section 5.2) to determine a schedule that attempts to complete all the node operations in the DDG in the shortest possible time. After each node is scheduled, the scheduler calculates the ready time of the other nodes that depend on the current one based on the finishing cycle of the scheduled operation. The same procedure is conducted repeatedly until all the DDG nodes are scheduled. A VLIW instruction routine for the DDG is derived from the scheduling result of all the nodes, with each field of the instruction controlling a specific hardware unit on
Figure 5.1 shows an example of a DDG and the accelerator customized for executing the operations. In Figure 5.1(a), assume that a, b, and c are current ready nodes and node a is determined to be scheduled first. The accelerator, which is shown in Figure 5.1(b), consists of four PMs with each containing two FUs. The accelerator contains four kinds of specialized FUs (FU-1, FU-2, FU-3, and FU-4) to perform operations for four kinds of DDG nodes (F-1, F-2, F-3, F-4). Since the result of node a will be used as an operand for both node e and node f, it is better to put node a on the PM that can also be used for node e and node f so as to avoid a data transfer between different PMs. Therefore, the FU-3 inside PM-1 is the better choice for node a than the FU-3 inside PM-4.

The above example displays a simple case in which the scheduler only compares two choices and only considers the nodes one step ahead. To handle more complex scheduling tasks and to achieve good results for all the DDG nodes, the scheduler determines the particular FU for each node operation with considerations of finishing executions of all the node operations on the accelerator in the shortest time. In the next section, Section 5.2, we will present a delay model of DDGs that can be used to estimate the impact that is caused by the particular choice of the FU for each node.
to the total time needed to finish all the node operations in a DDG.

5.2 A Heuristic Scheduling Algorithm

The goal of scheduling DDG nodes is to finish all the operations within the shortest time. In a DDG, the delay time on a node corresponds to the latency of the FU that executes the node operation. The delay time on an edge may be composed of the delays of data transfer between an FU and an SU within a PM and data transfer between two PMs. Extra delays will be inserted during scheduling when structural hazards (multiple nodes competing for a hardware unit at the same cycle) force some ready nodes to be scheduled at later cycles. These extra delays can only be computed through exhaustive analysis of all possible ways to schedule all the nodes in a DDG. Since it is infeasible to obtain the concrete values of these extra delays during the scheduling process, PECompiler uses a heuristic algorithm to make the scheduling decisions. It determines the schedule of each node based on the estimate of total cycles needed for finishing operations of all the DDG nodes.

During scheduling, particular choices of a DDG node and FU not only affect when the node operation can be finished, but also influence the schedules of other DDG nodes. To calculate the impact of each scheduling decision, a delay model of the DDG is developed based on the delay information of the nodes and edges in the DDG. Figure 5.2 describes how the delay model is derived for a DDG with respect to an accelerator containing four PMs. In the figure, \(d(x, i)\) represents the time needed to finish node operations on the longest path from the current node to the last directly or indirectly connected nodes, where \(x\) is the index of the current node and \(i\) is the index of the PM on which the current node is executed. It is called the Estimated Path Delay (EPD). For any node \(x\) in the DDG, a vector of EPDs \([d(x,1), d(x,2), d(x,3), d(x,4)]\) represents the time delays when the current node is executed on PM\_1, PM\_2, PM\_3, and PM\_4, respectively. The smallest value, \(d(x, j)\), in the EPD vector of a node indicates that the best FU for the node \(x\) is in PM\_j. The EPD vector of each DDG node can be computed using the equations shown at the
\[ d(a, i) = \begin{cases} \infty, & \text{if node}_a \text{ cannot be executed on PM}_i; \\ \max\{m(a,b), m(a,c), m(a,e)\} + dfu(a), & \text{if node}_a \text{ can be executed on PM}_i. \end{cases} \]

where, \( m(a,y) = \min\{d(y,1), d(y,2), d(y,3), d(y,4)\} + l(x,y) \) is a node index, \( dfu(a) \) is the latency on the FU used to execute the operation for node \( a \), and \( l(x,y) \) represents the time delay for transferring data from the PM that executes the node \( x \) to the PM that executes the node \( y \):

\[ l(x, y) = \begin{cases} \text{a global delay, if node}_x \text{ and node}_y \text{ are not executed in the same PM}, \\ \text{a local delay, if node}_x \text{ and node}_y \text{ are executed in the same PM}. \end{cases} \]

Figure 5.2: A Delay Model for DDG Nodes

PECompiler uses this delay model to schedule a DDG to run on an accelerator. Based on the model, we created a heuristic algorithm to determine schedules of the DDG nodes. The first step of the algorithm is to create a Ready_Node_List, which contains all the nodes that are initially ready to be scheduled. The second step is to use a loop to schedule all the nodes in Ready_Node_List. Based on the EPD vectors of the nodes, PECompiler chooses the most urgent node to be scheduled currently by comparing the path lengths of the ready nodes. The path length of each node is the sum of the minimum value in the EPD vector of the node and the earliest cycle time at which the node can possibly finish the execution. Then, the compiler further determines the best choice of PM to execute the node. The third step of the algorithm is to remove the node from Ready_Node_List. If other nodes become ready after the current node is executed, the compiler adds these nodes into the Ready_Node_List. Then the algorithm goes back to the second step and continues to schedule other nodes, until Ready_Node_List becomes empty when all the nodes are scheduled. By
always giving the most urgent nodes (the nodes on the critical paths in the DDG) the highest priority to use the earliest available hardware resources, the compiler can reduce the total number of clock cycles that are needed for all the DDG nodes to be executed on the accelerator. Using this heuristic algorithm, the partition of the DDG and schedules of the node operations are generated at the same time.

5.3 Pattern Inference Vectors

Table structures are commonly used in operation scheduling for pipelined computers. A schedule table can be formed with each row representing a cycle time and each column representing a hardware unit. A hardware unit is the finest-grained element that can be allocated by the scheduler. Each grid cell in the table is called a schedule entry and used to indicate whether or not the hardware element corresponding to the column is vacant at the time cycle corresponding to the row. A basic operation, such as fetching an operand from a register file to an input port of an FU, usually needs more than one hardware element to cooperate in successive clock cycles. The utilization of these hardware elements can be presented by a set of schedule entries in the schedule table. The term schedule pattern is defined as:

\[[(u_1, t_1)(u_2, t_2)...]\]

where \(u_i\) or \(u_j\) is a hardware unit and \(t_i\) or \(t_j\) represents a cycle time, the pair \((u_i, t_i)\) indicates that hardware unit \(u_i\) is used at clock cycle \(t_i\), the complete list of pairs of hardware units and clock cycles together represent an operation performed by using these hardware units at corresponding clock cycles.

We use an example that is shown in Figure 5.3 to illustrate the above terms and define notations for them. Figure 5.3(a) shows a partial structure of a PM that contains a single FU, three register files (Regf_1, Regf_2, and Regf_3), and the bus connections for fetching operands from the read ports of the register files. A port buffer is placed at each read port to increase the issue rate of operations on the FU. There are eight hardware elements, which are \(u_1, u_2, u_3, u_4, u_5, u_6, u_7,\) and \(u_8\) in the structure. Each element contains a latch to support pipelining for the operand
Eight hardware elements used in scheduling; each has a latch and a latency of 1 cycle:

- u1, u2, u3, u6, u7, u8 are network ports;
- u4, u5 are multiplexers.

Five patterns for fetching operands:

- P1: [(u1, 0), (u6, 1)]
- P2: [(u1, 0), (u4, 1), (u5, 2), (u7, 3)]
- P3: [(u2, 0), (u4, 1), (u5, 2), (u7, 3)]
- P4: [(u3, 0), (u5, 1), (u7, 2)]
- P5: [(u3, 0), (u8, 1)]

Fetches in this PM. There are five different patterns that can be used to fetch data to three FU input ports when the operands are stored in different register files. The first pattern is to fetch an operand from Regf1 to Inp1, the second pattern is to fetch an operand from Regf1 to Inp2, the third pattern is to fetch an operand from Regf2 to Inp2, the fourth pattern is to fetch an operand from Regf3 to Inp2, the fifth pattern is to fetch an operand from Regf3 to Inp3. The five patterns (P1, P2, P3, P4, and P5) are also shown in the Figure 5.3(a). Each of the patterns is presented by a list of (element_i, cycle_j) sets in which the element_i specifies a hardware element and the
cycle_\text{j} specifies a pipeline stage. For example, the second pattern (\text{P2}) fetches an operand from \text{Regf1} to \text{Inpl} through four hardware elements in four clock cycles, \text{u1} at stage 0, \text{u4} at stage 1, \text{u5} at stage 2, and \text{u7} at stage 3. In the figure, the pattern is represented by the list: [(\text{u1,0}), (\text{u4,1}), (\text{u5,2}), (\text{u7,3})]. Figure 5.3(b) shows the schedule table built for the structure. Each column of the table corresponds to a hardware element, and each row corresponds to a clock cycle. Every table grid cell is a \textit{schedule entry}, and the $\times$ marked on a table grid cell indicates that the hardware element corresponding to that column is used during the clock cycle corresponding to that row. All the marked grid cells in the table record a data fetch of the \text{P2} pattern scheduled at cycle $i$. When scheduling a DDG node to run on the FU, \text{PECompiler} first determines all the patterns that are needed for fetching the operands to the FU from the registers that store the data. It then allocates the cycles to schedule the patterns. To determine if a pattern can be scheduled at a particular cycle, all the table grid cells corresponding to the set of elements in the pattern need to be checked. Only when none of the grid cells is occupied, the cycle can be assigned to the pattern. This is potentially a slow operation, but can be sped up by use of efficient data structures.

To simplify the scheduling procedure for \text{PECompiler}, a different table-based scheduling method is defined to reduce the number of comparisons. A new table is created to record the interferences among all the patterns that are used in scheduling. As shown in Figure 5.3(b), after a \text{P2} is scheduled at cycle $i$, it may conflict with other patterns scheduled around the time. When trying all five patterns inside the schedule table, the \text{P2} (scheduled at cycle $i$) will conflict with a \text{P1} scheduled at cycle $i$, or another \text{P2} scheduled at cycle $i$, or a \text{P3} scheduled at cycle $i$, or a \text{P4} scheduled at cycle $i+1$. The whole set of pattern conflicts with respect to a scheduled pattern can be collected into a vector $\{(\text{P1,0}), (\text{P2,0}), (\text{P3,0}), (\text{P4,1})\}$, with each element defined as (pattern_index, offset_cycles). In this paper, the vector is called the \textit{Pattern Interference Vector (PIV)} of \text{P2}.

\textit{Definition 5.1:} Let $p_\text{i}$ and $p_\text{j}$ be identifiers of any two patterns, let $n_\text{i}$ be a number of clock cycles, let $t_\text{i}$ and $t_\text{j}$ be two cycle times, assume that $p_\text{j}$ is scheduled at cycle
time \( t_j \), let \( \text{PIV}[j] \) be the PIV of \( p_j \). Then

\[
\text{PIV}[j] = \bigcup \{ (p_i, n_i) \}
\]

where each \( (p_i, n_i) \) pair meets following three conditions:

1. pattern \( p_i \) is scheduled at cycle time \( t_i \);
2. \( n_i = t_i - t_j \);
3. patterns \( p_i \) and \( p_j \) share at least one hardware unit at the same cycle.

Using this definition, we can derive a set of PIVs for all five patterns as follows:

- \( \text{PIV}[1] \): \( \{(P2, 0)\} \)
- \( \text{PIV}[2] \): \( \{(P1, 0), (P2, 0), (P3, 0), (P4, 1)\} \)
- \( \text{PIV}[3] \): \( \{(P2, 0), (P3, 0), (P4, 1)\} \)
- \( \text{PIV}[4] \): \( \{(P2, -1), (P3, -1), (P4, 0), (P5, 0)\} \)
- \( \text{PIV}[5] \): \( \{(P4, 0)\} \)

With these pattern interference vectors, we define an *Interference Table* for *PECompiler* to find suitable cycles for patterns in scheduling and recording scheduled patterns. Each row of the interference table corresponds to a cycle time and each column of the table corresponds to a pattern. An unoccupied grid cell in the table indicates that the pattern associated with the column index of the grid cell can be scheduled at the cycle associated with the row index. When recording a scheduled pattern, all the PIV elements of the pattern will be marked in the corresponding grid cells in the interference table. Therefore, other patterns will not be scheduled at cycles with conflicts. Figure 5.3(c) shows the interference table created for the five patterns. When \( P2 \) is scheduled at cycle \( i \), the PIV of \( P2 \) is marked in the interference table. Using the interference table, only one comparison is needed to determine if a pattern can be scheduled at a particular cycle. Compared to the method based on the conventional schedule table, which is shown in Figure 5.3(b), four comparisons may be required to determine if a pattern can be scheduled at cycle \( i \). By using this interference table based method, the time for *PECompiler* to generate schedules of DDG node operations can be significantly reduced.
5.4 Register Allocation

Register allocation is one of the most important steps performed during compilation of the source programs. It is usually performed at a late stage of the compilation, typically immediately before the code generation. However, during the compilation of an accelerator, PECompiler performs register allocation at the same time as it generates schedules of all the node operations. The decision on assigning a particular register to a source operand or a result of a node operation may affect not only the schedule of the node operation, but also the schedules of many other nodes. To meet the requirement of scheduling operations and allocating registers at the same time, this section introduces a new algorithm for register allocation. Before describing our new algorithm, Section 5.4.1 briefly reviews the general approach that is used in recent compiler designs.

5.4.1 The Graph Coloring Method

In modern compilers, register allocation is often performed using graph coloring algorithms [ASU88] [CH90] [CAC+81]. This kind of algorithm is based on an interference graph, which is constructed for the program. Each node in the graph represents a variable that needs a register, and an arc between two nodes represents the overlapped live ranges between the variables corresponding to the nodes. A live range is defined as the time period between the cycle that a data value is written into the register and the cycle that another data value is written. The approach to perform register allocation is to assign the nodes in the interference graph with a number of colors, so that any two adjacent nodes, which are linked by an arc, must be assigned with different colors. This method is equivalent to assigning different registers to the variables that have overlapped live ranges. The number of different colors that are assigned to the nodes is equal to the number of registers that can be assigned to the variables in the program. The goal of the graph coloring is to find the best way to assign different colors to adjacent nodes, so that the total number of register spills can be minimized. Register allocation is a hard problem, and the solution is often
derived using heuristic algorithms.

Register allocation in PECompiler has different criteria from conventional computers. There are two primary differences between accelerators and conventional computers. First, the hardware structures are different. In conventional computers, there is only one single register file, and register allocation is based on the available registers in the register file. In the accelerator, there are multiple register files that can be accessed by an operation executed on every FU in the same PM. To perform register allocation, PECompiler has to first choose a register allocation for the variable, and then allocate a register in the register file. The decision on a particular register file is dependent on not only the available registers in the register file, but also the available read/write ports on the register file (no other FU can use the same port to access the register file at the same cycle). Second, in a conventional compiler, register allocation is performed after the sequence of all the operations is decided, and based on the fixed sequence, the compiler can derive an interference graph for the program. However, PECompiler performs register allocation before the sequence is determined. As a result, the compiler cannot create an interference graph for the program, thus it is impossible to apply the graph coloring method to solve the problem of register allocation.

5.4.2 The Register Allocation Algorithm

When creating the DDG, PECompiler assigns a unique symbol to each data item at a node output port. Each of the symbols in the DDG will be allocated to a register by the compiler. Similar to graph coloring, the foundation of our register allocation algorithm is based on analysis of the live ranges of data items that need registers.

As described earlier, by using partial evaluation, PECompiler exposes parallelism among all the node operations in a DDG, and breaks long dependency chains into shorter pieces. As a result, a large number of DDG nodes can be executed in parallel. During the execution, a large number of registers are needed to store their operands and results, but most of the data items stored in the registers have a short lifetime.

To effectively use available registers, PECompiler uses a register allocation method
that is based on calculations of live ranges of data items. The basic technique relies on a register allocation table on which each column represents a register and each row corresponds to a cycle time. The number of columns in the table is equal to the total number of registers in the register file. Since the customized accelerator consists of multiple register files, one register allocation table is defined for each of the register files. Figure 5.4 shows the table structure and depicts how PECompiler uses the table to perform the register allocation for the DDG nodes. For simplicity, assume that each register file contains eight registers. Since the register allocation problem is NP-complete, the algorithm uses heuristic decisions to derive the solution that is likely a good one.

### Recording live ranges in the register allocation table

When allocating a register to a data symbol, PECompiler marks the table entries in the column corresponding to the register. As shown in Figure 5.4(a), the shaded vertical bars represent live ranges of register contents. The letter \( w \) indicates that a data item is written into the register file at the cycle corresponding to the table row, and the letter \( r \) specifies that the data item in the register is read at the cycle.
corresponding to the row. Every write starts a live range in a register, and the last cycle of a live range is always a read. Since a data value stored in a register is often read more than once, multiple \( r \)'s may appear within a live range marked in the table. Figure 5.4(b) shows the register allocation for the data symbols in the DDG. Symbols \( S0, S1, S2, \) and \( S3 \) represent the data values that are assigned with registers inside the register file that is associated with the table in the figure. The edges that are drawn in dashed lines represent data values that are stored in other register files.

When scheduling node \( a \), \textit{PECompiler} allocates register \( R0 \) to store the result that is represented by symbol \( S0 \), and allows the result to be written into the register at cycle \( i+1 \). The allocation is recorded in the table using a bar in the column of \( R0 \). The live range starts from cycle \( i+1 \) with the letter \( w \). The downward arrow indicates that the register contents are active and the end of the live range is indefinite. The second node scheduled is \( b \), and register \( R1 \) is allocated to store its result \( S1 \). The data value is written into the register at cycle \( i \), and read twice as a source operand for both nodes \( d \) and \( f \) at cycles \( i+2 \) and \( i+5 \), respectively. Since the contents of \( R1 \) are no longer needed, the compiler releases the register and terminates the live range of data item stored in \( R1 \). Then register \( R1 \) becomes available at cycle \( i+6 \).

Similarly, for symbol \( S2 \), register \( R2 \) is first allocated to store the results of node \( c \) and later released after node \( e \) finishes reading the data at cycle \( i+3 \). The released \( R2 \) is allocated again for node \( g \) at cycle \( i+5 \). At this point, two registers are still occupied by symbols \( S0 \) and \( S3 \), and the other six registers are free and can be assigned to other data symbols.

\textbf{Allocating a register}

When a register is needed for a data symbol that is used as a source operand or an execution result of a node operation, \textit{PECompiler} will look for a column that is not assigned to an active symbol. The decision of the particular column is made according to the cycle in which the data item becomes valid. The best column is the one that becomes available in the cycle closest to the valid time of the data item. \textit{PECompiler} first tries to find the best one among the columns that become available earlier than
the data valid time. If no column is available earlier than the data valid time, then PECompiler chooses the column that has the earliest available time.

Figure 5.5 gives examples of allocating registers to two data symbols (S1 and S2), one after another. The value of the first data symbol (S1) can be written into the register file at a cycle time as early as \( i+4 \), and the data value of the second symbol (S2) can be written into the register file at a cycle as early as \( i+1 \). The current status of the Register Allocation Table is shown in Figure 5.5(a). Three registers (R0, R2, and R4) are occupied by active data items, and five registers (R1, R3, R5, R6, and R7) can be assigned to data symbols. The available cycles of the five registers are \( i+6, i+2, i+5, i+3, \) and \( i+7 \), respectively. Since R6 has the latest available cycle \( i+3 \) that is not later than cycle \( i+4 \) (the valid cycle of symbol S1), PECompiler assigns register R6 to S1. For symbol S2, PECompiler cannot find a register available at cycle \( i+1 \), then the compiler assigns register R3, which becomes available at the earliest cycle \( i+2 \), to the data symbol. Figure 5.5(b) shows the table status after recording the usages of the registers.
Releasing a register

When \textit{PECompiler} finishes scheduling all the nodes that use the contents of a register as a source operand of their operations, the register will be released. The live range of the register contents ends at the cycle time that the register contents are last read. Starting from the next cycle, the register becomes available and can be assigned to another data symbol. When a register is released, \textit{PECompiler} ends the live range at the table row corresponding to the last read cycle in the table column corresponding to the register.

When releasing a register, \textit{PECompiler} also rearranges the live ranges in the table, so that the number of unused cycles in all the table columns is minimized. By applying this optimization, the compiler can increase the utilization of the limited register resource. Figure 5.6 gives examples of releasing registers and rearranging live ranges in the \textit{Register Allocation Table}. In the table shown in Figure 5.6(a), the contents of registers \textit{R0} and \textit{R4} are last read at cycles \(i+3\) and \(i\), respectively. Then the live range of the data item in register \textit{R0} ends at cycle \(i+3\), and the register becomes available starting from the next cycle. Then the open end of the bar in the column

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.6.png}
\caption{Current Status of the Table (a) and After Allocating Two Registers (b)}
\end{figure}

range of the data item in register \textit{R0} ends at cycle \(i+3\), and the register becomes available starting from the next cycle. Then the open end of the bar in the column
corresponding to \( R0 \) is terminated at cycle \( i+3 \), and the live range from cycle \( i+4 \) to cycle \( i+6 \) in the column of \( R1 \) is merged into the column of \( R0 \). The purpose of merging live ranges is to reduce the number of empty cells between live ranges in each register allocation table column. Each empty cell in the table indicates an unused cycle of a register. By merging the live range in the \( R1 \) column with the live range in the \( R0 \) column, two unused cycles of register \( R1 \) can possibly be used by some other live range. In the same way, register \( R4 \) is first released after cycle \( i \), and then the live range in the column of \( R2 \) is merged into the column of \( R4 \). Figure 5.6(b) shows the status in the table after the registers are released. After merging the live ranges, the register file can provide two more registers to other data symbols at earlier cycles.

**Spilling a register**

When a data symbol in the DDG needs a register after all the registers are occupied by other data items, \textit{PECompiler} may choose to free one of the registers by spilling the contents of the register into memory, and then assign the register to the symbol. \textit{PECompiler} uses the least-recently used (LRU) [HP90] algorithm to choose a register to be spilled.

To spill the register, \textit{PECompiler} first checks if there is another copy of the data item stored in a memory block or a register file in another PM. When there is another copy, the compiler simply releases the register after the cycle at which the contents are last read. Otherwise, the compiler first schedules an extra operation to save the data copy into the memory block, which is in the same PM with the register file. It then releases the register. After the register is released, \textit{PECompiler} also rearranges the live ranges on the table and tries to fill the unused cycles in all the columns.
5.5 The General Scheduling Algorithm

The general algorithm for scheduling the node operations for a DDG is given as follows:

```
Schedule_Algorithm {
    1. Read DDG.Node.File;
    2. Read Machine_Model.File;
    3. Build the Interference_Table;
    4. Compute critical paths in the DDG using the Delay Model;
    5. Create a Ready_Node_List and Collect Initial Ready Nodes;
    6. While (Ready_Node_List is not empty) {
        6.1. Choose the most urgent node from the Ready_Node_List;
        6.2. Loop: Search for the best way to schedule the current node {
            6.2.1 Choose one of all candidate FUs for node operation;
            6.2.2 Allocate registers for the operands and results;
            6.2.3 Compare the finish cycle of the node operation with the best one:
                if an earlier finish cycle is obtained, record it as the current best.
        }
        6.3. Update the Ready_Node_List:
            6.3.1. Delete the scheduled node;
            6.3.2. For each node that becomes ready, add to the ready node list;
        6.4. Record the schedule results;
    }
}
```

5.6 Summary

In this chapter, we have described the generation of instruction sequences for computations in application programs to be accelerated on the FCCM. We have illustrated the general approach of producing the sequences based on the schedules of all the
DDG nodes (in Section 5.1). We have introduced a heuristic algorithm to determine the schedule of each node operation in the DDG. In Section 5.3, we have also proposed a new table structure, which is called the *Interference Table*, for *PECompiler* to quickly determine the best schedule for each node operation. In Section 5.4, we have described the algorithms that are used to perform register allocation in *PECompiler*. 
Chapter 6

Examples

This chapter presents two examples of using PECompiler to generate the hardware designs and control sequences of customized accelerators. Various design considerations for the major architectural parameters will be further discussed in Chapter 7.

6.1 Simple-Timing Simulation Program

The STS (simple-timing simulation) program performs timing simulation for digital circuits. The program has two parts to its input. One of them is the netlist of the circuit (including transistor models used in the circuit), and the second is the test vectors needed for simulations. For this example, PECompiler will read the first part of input during compilation and apply partial evaluation to the source code. The same simulation algorithm of the program was also used to build a hardware accelerator [Lew88], however the input to PECompiler is an annotated C program.

As illustrated in [Lew88], digital MOS circuits can be modeled by the nodal formulation \( C(v, u) \times \dot{v} = G(v, u) \) where \( v \) is a vector of nodal voltages, \( u \) is a vector of external inputs, \( C \) is a matrix of capacitances, and \( G \) is a matrix of currents. After discretizing time and approximating \( \dot{v}_k = (v_{k+1} - v_k)/h \) using the forward Euler integration method, the vector of node voltages at the end of time step \( k \) can be calculated by \( v_{k+1} = v_k + h \times C^{-1} \times G(v, u) \) where \( h \) is the integration time step and the matrix \( C \) is assumed diagonal and constant (i.e. nodes only have capacitance
to ground). The computation of $G(v, u)$ is decomposed into several $G_i(v, u)$ where $G = \sum_{i=1}^{N} G_i$ and $N$ is the number of transistors in the circuit. Each $G_i$ corresponds to one transistor, and there are three node voltages (one for each of the gate, source, and drain of the transistor) associated with each $G_i$. The algorithm of the computation kernel was expressed as

$$\text{for } (i \leftarrow 1 \text{ to } N)$$

$$\Delta V_S \leftarrow B_{TS_i} \times I_{DS}(V_{k,TG_i}, V_{k,TS_i}, V_{k,TD_i})$$

if ($F_{Si}$)

$$V_{k+1,TS_i} \leftarrow V_{k,TS_i} + \Delta V_S$$

else

$$V_{k+1,TS_i} \leftarrow V_{k+1,TS_i} + \Delta V_S$$

... similar code for drain

where $V$ represents node voltages in the circuit, $F_{Si}$ indicates if transistor $i$'s $I_{DS}$ is the first current that affects $V_{k+1,TS_i}$; $k$ specifies a time step; $TG_i$ and $TS_i$ are the indices of the nodes connected to the transistor, and $B_{TS_i}$ is a constant. To reduce the hardware cost and delay time for the computations in the algorithm, we can first convert the expression: $B_{TS_i} \times I_{DS}(V_{k,TG_i}, V_{k,TS_i}, V_{k,TD_i})$ into a logarithmic representation: $\exp(\log(B_{TS_i}) + \log(I_{DS}(V_{k,TG_i}, V_{k,TS_i}, V_{k,TD_i})))$, and then calculate the value of the new expression by looking up two tables (ids_table and exp_table) that contain the precomputed values of $I_{DS}$ and exponents.

Figure 6.1 shows the source code of the computational kernel of the STS program. The main program contains three steps: (1) read the parameters of the transistors used in the simulated circuit and construct the lookup tables (arrays ids_table and exp_table) that are used to reduce model evaluation time for calculating node voltages; (2) read the circuit netlist and build index vectors that associate transistors with their connected nodes in the circuit; (3) read the test vectors and calculate the node voltages of the circuit in discrete time steps. For each time step, subroutine sim_step is invoked once to evaluate the voltages of all the nodes. The execution of sim_step updates node_voltage by accumulating the voltage changes of every node which is affected by all of the transistors in the circuit. The bulk of this work is done by
/* Source program of the STS application: */

/* global program: */
int node_voltage[max_node]; /* store voltage on nodes in circuit */
int exp_table[exp_tab_size]; /* lookup table used in computations */
int ids_table[ids_table_size]; /* lookup table for computing Ids values */

main()
{
  read_config(); /* Step 1: read transistor parameters */
  read_net(); /* Step 2: read circuit netlist */
  while(read_test_vector() != EOF) /* Step 3: read test vectors */
  {
    sim_step(); /* simulate circuit for a time step */
  }
}

void sim_step()
{
  int j;
  vbase_old = 0; vbase_new = n_node;
  for (j=0; j<trans; j++)
    do_trans(j); /* compute impact for each transistor */
  vbase_old = n_node; vbase_new = 0;
  for (j=0; j<trans; j++)
    do_trans(j); /* compute impact for each transistor */
}

void do_trans(it)
{ /* implement computation kernel */
  ................
  vold = node_voltage[vbase_old + trans_gate[it]];
  vs_old = node_voltage[vbase_old + trans_src[it]];
  vd_old = node_voltage[vbase_old + trans_drain[it]];
  vs_new = node_voltage[vbase_new + trans_src[it]];
  vd_new = node_voltage[vbase_new + trans_drain[it]];
  if (vs_old > vd_old)
    vds = vs_old - vd_old; vmax = vs_old;
    vmin = vd_old; dir_flag = 1;
  else
    vds = vd_old - vs_old; vmax = vd_old;
    vmin = vs_old; dir_flag = 0;
  if (config_trans_type[trans_type[it]])
    vgs = vmax - vs_old;
    vbs = node_voltage[vdd_node] - vmax;
  else
    vgs = vg_old - vmin;
    vbs = vmin - node_voltage[gnd_node];
  if (vbs < 0) vbs = 0;
  if (vgs < 0) vgs = 0;
  log_ids = ids_table[trans_type[it]]
    [vgs >= trans_vgs_shift]
    [vds >= trans_vds_shift]
    [vbs >= trans_vbs_shift];
  dvs = exp_table[log_ids + trans_k_over_c_src[it]];
  if (dir_flag) dvs = -dvs;
  if (trans_first_src[it])
    vs_new = vs_old + dvs;
  else
    vs_new = vs_new + dvs;
  if (node_is_input[trans_src[it]])
    node_voltage[base_new + trans_src[it]] = vs_new;
  vdd = exp_table[log_ids + trans_k_over_c_drain[it]];
  if (dir_flag) vdd = -vdd;
  if (trans_first_drain[it])
    vd_new = vd_old + vdd;
  else
    vd_new = vd_new + vdd;
  if (node_is_input[trans_drain[it]])
    node_voltage[base_new + trans_drain[it]] = vd_new;
}

Figure 6.1: The STS Source Program

subroutine do_trans. This routine is called for each transistor to compute the voltage changes on its source and drain nodes. In the source code, subroutine do_trans is annotated (Section 4.2.2 describes how to make user annotations) to be compiled into a hardware routine (HwRoutine) that is to run on the FCCM.

By applying partial evaluation, PECompiler determines that five FUs (FU_1, FU_2, FU_3, FU_4, and FU_5) are needed for computational fragments in do_trans. Each kind of unit implements a series of computations, which are shaded in Figure 6.1. The partitions among the five code segments are derived by PECompiler,
/* Main Program of STS Application: */
int node_voltage[max_node];
int exp_table[exp_tab_max];
int ids_table[ids_tab];
...........................

main()
{
...............................
while(read_test_vector() != EOF) {
    for (i=0; i<Total_time_steps; i++) {
        /* invoke hardware routine running on an accelerator */
        HwRoutine(i);
    }
   ...............................
}

(a) Optimized main program

<table>
<thead>
<tr>
<th>FU</th>
<th>LUTs</th>
<th>% of EPF10K100</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU-1</td>
<td>272 (LCs)</td>
<td>6%</td>
</tr>
<tr>
<td>FU-2</td>
<td>20 (LCs)</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>FU-3</td>
<td>102 (LCs)</td>
<td>2%</td>
</tr>
<tr>
<td>FU-4</td>
<td>20 (LCs)</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>FU-5</td>
<td>102 (LCs)</td>
<td>2%</td>
</tr>
</tbody>
</table>

(b) Logic Cells Needed for Three FUs

(c) Operations Contained in 'HwRoutine()'

Figure 6.2: The Optimized Program and DDG

as the indices for accessing the two array (ids_table and exp_table) are dynamic. Figure 6.2(a) shows the optimized program generated by PECompiler. The first two steps in the original main program have been accomplished when partial evaluation is applied. The call to sim_step has been replaced by an invocation of the hardware routine HwRoutine. HwRoutine is the flattened code of sim_step after PECompiler has fully unrolled its loop and inlined all the calls to do_trans. Figure 6.2(b) shows the utilization of logic cells for mapping the five FUs onto Altera FLEX10K devices [ALT96]. To show the contents of HwRoutine, we used an inverter that contains two transistors as an example circuit. Figure 6.2(c) gives a DDG representation of all
the operations contained in HwRoutine generated for the inverter. There are three kinds of fu nodes (FU-1, FU-2, and FU-3) in the DDG, and only the three FUs (FU_1, FU_2, and FU_3) are needed to execute the nodes. In the inverter circuit, the drain terminals of both transistors connect to either vdd or gnd, so PECompiler eliminates the computations of voltages on the two terminals. As a result, FU_4 and FU_5 are not needed for simulating the inverter. There are two other kinds of nodes, read and write, in the DDG. The write nodes update specific elements of the array node_voltage. The input to each read node is an array index. The read nodes without an input, read elements of array node_voltage with fixed addresses. Other read nodes read elements of array ids_table or exp_table with dynamic addresses. After applying optimizations, 33 operations are needed to simulate the two transistors for a time step.

When simulating three large circuits, the compiler generates a customized accelerator for each netlist using the five kinds of FUs. The first of the three circuits is an encoder, which contains 1346 transistors. The second circuit is a crc circuit, which contains 478 transistors. The third circuit is an 8-bit counter, which contains 246 transistors. Table 6.1 shows the input/output ports and the utilization of the five FUs for the STS program with the three circuits. Figure 6.3 shows the five FU

<table>
<thead>
<tr>
<th>FU/Op.</th>
<th>Ports on FU</th>
<th>FU Utilization of Three Circuit Netlists</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>input</td>
<td>output</td>
</tr>
<tr>
<td>FU-1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>FU-2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>FU-3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>FU-4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>FU-5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Memory Accesses</td>
<td>8641</td>
<td>2980</td>
</tr>
</tbody>
</table>

Table 6.1: The FUs Needed by the STS Program

designs. For example, by using the netlist of the encoder circuit, the compiler generates the trace of operations that need to be executed on the accelerator. The trace
Figure 6.3: The Structures of the Five FUs

is represented by a DDG containing 19053 nodes. As a comparison, for the same circuit (encoder) the original sim_step needs to call do_trans 2692 (to simulate each transistor, do_trans is called twice) times to finish the same computation task, and a C compiler compiles do_trans into 325 instructions with a shortest execution path of 252 instructions (on SPARCstation). To simulate the circuit, the program needs to execute approximately 874,900 instructions. When using the netlist of the crc circuit (478 transistors), the DDG generated by the compiler contains 6705 nodes, and the program needs to execute approximately 310,700 (478x2x325) instructions. When using the netlist of the 8-bit counter circuit (246 transistors), the DDG generated by
the compiler contains 3573 nodes, and the program needs to execute approximately 159,900 (246x2x325) instructions.

We conducted a series of experiments for the three circuit netlists based on machine models of the accelerators with different numbers of PMs. Each PM contains a set of the five FUs, six register files and one data memory block. When more than 1 PM is implemented in the accelerator, a full crossbar network is used to interconnect all the PMs. The speed of execution on the FCCM is also determined by other hardware parameters of the customized accelerator, and more detailed analysis of the design tradeoffs between the hardware cost for implementing the hardware specifications and the performance of the accelerator will be presented in Chapter 7. Table 6.2 presents the number of cycles needed to finish the executions of simulating the three circuits on an FCCM platform when different numbers of PMs are used to implement the customized accelerators. It also shows the estimated speedup (clock cycle ratio) that can be obtained on each accelerator. On a clock cycle basis, the

<table>
<thead>
<tr>
<th>Total PMs/FUs</th>
<th>encoder</th>
<th>crc</th>
<th>8-bit counter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TEC*</td>
<td>CCR**</td>
<td>ES***</td>
</tr>
<tr>
<td>1/5</td>
<td>14065</td>
<td>62</td>
<td>2.1</td>
</tr>
<tr>
<td>4/20</td>
<td>1438</td>
<td>608</td>
<td>20.3</td>
</tr>
<tr>
<td>8/40</td>
<td>819</td>
<td>1068</td>
<td>35.6</td>
</tr>
<tr>
<td>12/60</td>
<td>616</td>
<td>1420</td>
<td>47.3</td>
</tr>
<tr>
<td>16/80</td>
<td>477</td>
<td>1834</td>
<td>61.1</td>
</tr>
<tr>
<td>20/100</td>
<td>403</td>
<td>2171</td>
<td>72.4</td>
</tr>
<tr>
<td>24/120</td>
<td>393</td>
<td>2226</td>
<td>74.2</td>
</tr>
</tbody>
</table>

* TEC: Total Execution Cycles; ** CCR: Clock Cycle Ratio; *** ES: Estimated Speedup.
For encoder: CCR = 874900/TEC; for crc: CCR = 310700/TEC; for 8-bit counter: CCR = 159900/TEC.
For all the three circuits: ES = CCR/30.

Table 6.2: Number of PMs versus Execution Cycles

speedup gained by using the FCCM is up to $2226 \times$ for the encoder circuit, $1046 \times$ for the crc circuit, and $828 \times$ for the 8-bit counter circuit. To estimate the speedup, assume that the accelerator implemented on the FCCM can run at a clock rate of 10MHz and a modern general-purpose CPU (e.g., SPARC CPU) can run at a clock
rate of 300MHz. The FCCM will then lose a factor of 30 due to the slower clock cycle, but a speedup of about $70 \times$ should be possible. Although we conducted the tests using models of accelerators with a full crossbar network and containing up to 24 PMs, the limited number of pins on the Altera 10K100 FPGA cannot provide enough data ports to implement a full crossbar network to interconnect 24 PMs (more details will be given in Chapter 7). However, implementations of the models with up to 24 PMs will be feasible when using FPGAs with more logic resources and a larger number of input/output pins. Figure 6.4 shows the speedups that can be obtained for simulating the three circuits by implementing different numbers of PMs on the FCCM. The results show that a higher speedup can be achieved when increasing the number of PMs in the accelerator. The performance has a super-linear improvement when the total number of PMs in the accelerator is increased from 1 to 4. This is because the accelerator with a single PM cannot provide enough registers to cache data from memory before using the data as operands of node operations. The results also show that the speedup for running the operations of a DDG with a larger number of nodes is higher than that of a DDG with a smaller number nodes. This is because the DDG that contains more nodes has a higher degree of parallelism among the
node operations and allows the compiler to schedule more operations to be executed simultaneously on the FCCM each clock cycle. When compiling the STS program with a variety of netlists, the longest dependency chains in most DDGs generated for simulating the netlists contain only 22 nodes. This means that a large number of DDG nodes can be executed at the same time. The high degree of the parallelism in the DDGs is obtained by PECompiler through applying partial evaluation.

6.2 Simple-Circuit Simulation Program

Our second example is the SCS (simple-circuit simulation) program. The SCS program performs a transient analysis of a circuit, which is similar to the SPICE program [Nag75] [MS90]. It uses nodal analysis with trapezoidal integration, and octal-tree based device models that can accelerate simulations [Lew90]. Like the STS program, the source code of the SCS example is also an annotated C program. The main program also contains three subroutine calls: the first subroutine reads the device models and sets up the tree-based table models for computing the voltages on circuit nodes; the second routine reads the netlist of the circuit to be simulated; the third routine performs simulations to the circuit based on input test vectors. The simulation subroutine uses the lookup table that has been created in the second step to evaluate the node voltages in the circuit. To accelerate the execution of circuit simulations, two code blocks (subroutines sim_step_trap_setup() and sim_step_trap_iter()) in the source program are annotated to be executed on the FCCM. For every time step of the simulations, sim_step_trap_setup() will be called once to set up the variables that are used in the simulations, then sim_step_trap_iter(), which implements the algorithm of the tree-based device model approximation, will be called iteratively in a loop until the computations converge. When using the program to simulate circuits, sim_step_trap_iter() takes most of the execution time. By implementing a specialized accelerator to speed up executions of both sim_step_trap_setup() and sim_step_trap_iter(), we can achieve high performance for the SCS program.

To generate the accelerator design, PECompiler compiles the SCS program to-
<table>
<thead>
<tr>
<th>FU/Op.</th>
<th>FU Ports</th>
<th>Functionality of FU</th>
<th>FU Utilization of Three Circuit Netlists</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>input</td>
<td>output</td>
<td>6-bit adder</td>
</tr>
<tr>
<td>FU-1</td>
<td>3 1</td>
<td>(see Figure 6.5)</td>
<td>15875</td>
</tr>
<tr>
<td>FU-2</td>
<td>1 1</td>
<td>output = (input * 2.796E+6) &lt;&lt; 24</td>
<td>1398</td>
</tr>
<tr>
<td>FU-3</td>
<td>4 1</td>
<td>(see Figure 6.5)</td>
<td>3191</td>
</tr>
<tr>
<td>FU-4</td>
<td>4 3</td>
<td>(see Figure 6.5)</td>
<td>2725</td>
</tr>
<tr>
<td>FU-5</td>
<td>1 1</td>
<td>output = l/input</td>
<td>149</td>
</tr>
<tr>
<td>FU-6</td>
<td>2 1</td>
<td>(see Figure 6.5)</td>
<td>298</td>
</tr>
<tr>
<td>FU-7</td>
<td>2 1</td>
<td>output = input1 + input2</td>
<td>25571</td>
</tr>
<tr>
<td>FU-8</td>
<td>2 1</td>
<td>output = input1 * input2</td>
<td>6748</td>
</tr>
<tr>
<td>Memory Accesses</td>
<td></td>
<td></td>
<td>29112</td>
</tr>
</tbody>
</table>

Table 6.3: The FUs Needed by the SCS Program

gether with the inputs of device models and a netlist, applies partial evaluation to the source code with respect to the input, and generates the VHDL descriptions for the FUs that are required by the program. It also generates two DDGs for the two annotated code blocks in the simulation routine based on the given netlist. One of the DDGs (DDG_setup) is created for code block sim_step_trap_setup(), and the other DDG (DDG_iter) is created for code block sim_step_trap_iter(). Table 6.3 shows a list of FUs that are designed for the annotated code and the three versions of DDG_iter that are generated for three test circuits (a 6-bit adder, a 4-bit counter, and a 20-inverter chain). There are eight kinds of FUs in the list, the numbers of input and output ports on the FUs are shown in the second and third columns in the table, the function of each FU is given in the fourth column. Each of the last three columns shows the numbers of nodes in the DDG that use the FUs. The total numbers of nodes in the three DDGs are also shown in the table. Figure 6.5 shows designs of four FUs (FU-1, FU-3, FU-4, and FU-6).

Using the eight kinds of FUs, we created different machine models for the accelerators, each of which contains a different number of PMs. Since the eight kinds of FUs are not needed by the same number of nodes in the DDGs, the total number of FUs for each kind may vary. Table 6.4 shows the total numbers of the FUs in five
Figure 6.5: The Structures of four FUs

different models. Based on each of the machine models, we generated an execution sequence for all the node operations in each of the three DDGs. Table 6.6 shows the total execution cycles that are needed to finish all the node operations for the three DDGs based on the five machine models. To calculate the speedups that are achieved by using the FCCM for the circuit simulations, we compared the time that is spent on running the program on a general-purpose computer with the time that is needed when using the FCCM to accelerate the simulations. When simulating the three netlists (6-bit adder, 4-bit counter, and 20-inverter chain), the program executions\footnote{Dr. David Lewis wrote the SCS program. He ran the program to simulate the three circuits, and recorded the CPU time that was spent on an UltraSPARC-Model170 workstation to simulate each of the circuits.} using the three netlists took about 11.66 seconds, 106.76 seconds, and 2.72 seconds of CPU time respectively. For simulating the 6-bit adder circuit, subroutine
Table 6.4: Total Numbers of the FUs in Different Machine Models

<table>
<thead>
<tr>
<th>Machine Model</th>
<th>Total Number of PMs</th>
<th>FU-1</th>
<th>FU-2</th>
<th>FU-3</th>
<th>FU-4</th>
<th>FU-5</th>
<th>FU-6</th>
<th>FU-7</th>
<th>FU-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MachModel_1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MachModel_2</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>MachModel_3</td>
<td>12</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>MachModel_4</td>
<td>16</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>MachModel_5</td>
<td>20</td>
<td>17</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

sim_step_trap_setup() had been called 94 times, and subroutine sim_step_trap_iter() had been called 366 times; for simulating the 4-bit counter circuit, sim_step_trap_setup() had been called 1049 times, and sim_step_trap_iter() had been called 4673 times; for simulating the 20-inverter chain circuit, sim_step_trap_setup() had been called 171 times, and sim_step_trap_iter() had been called 694 times. Table 6.5 lists the CPU time and the numbers of the subroutine calls. Based on the assumption that the accelerator implemented on the FCCM can run at a clock rate of 10MHz, we can estimate the total time that is spent by the program to simulate each circuit on a given machine model. The estimated execution time for all the experiments are also shown in Table 6.6.

Figure 6.6 shows the changes on the estimated speedups that are achieved by implementing different numbers of PMs on the FCCM. When more PMs are added
T: the total number of seconds needed to execute the program for simulating one circuit.
t-1: the number of cycles needed to execute DDG_setup;
t-2: the number of cycles needed to execute DDG_iter;

Table 6.6: Number of FU's versus Total Execution Cycles

<table>
<thead>
<tr>
<th>Total PMs</th>
<th>6-bit adder</th>
<th>4-bit counter</th>
<th>20-inverter chain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t_1</td>
<td>t_2</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>34967</td>
<td>35171</td>
<td>1.62</td>
</tr>
<tr>
<td>8</td>
<td>28876</td>
<td>21184</td>
<td>1.05</td>
</tr>
<tr>
<td>12</td>
<td>30222</td>
<td>23282</td>
<td>1.14</td>
</tr>
<tr>
<td>16</td>
<td>25739</td>
<td>20331</td>
<td>0.99</td>
</tr>
<tr>
<td>20</td>
<td>27422</td>
<td>19631</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Figure 6.6: Number of PMs versus Estimated Speedup

to the accelerator, the speedup obtained for all the three cases rises, but the rate of changes becomes smaller after the total number of PMs in the accelerator is increased. There are several reasons that cause the saturation on the increase rate of the speedup. The first one is because the SCS program requires 32-bit floating-point FUs to perform circuit simulations, each of the FUs needs more logic resources to be implemented.
To map each PM on a single FPGA chip, we can only place two or three FUs on each PM. As a result, a large number of data transfers are performed on the top-level network among all the PMs. The second reason is because the DDGs generated for simulating the three circuits contain long dependency chains. For example, the longest dependency chain in the DDG that is generated for simulations of the 4-bit counter netlist contains 841 nodes. This kind of long chain determines the highest speed of execution on the FCCM that can be possibly achieved. Like the accelerator built for the STS program, the limited available registers and memory bandwidth in each PM also cause the saturation of speedup increase that can be obtained by adding more PMs.

6.3 Compilation Time

For a given application program, the speedup obtained by using a customized accelerator should be calculated based on both the compilation time and execution time of the program. In Sections 6.1 and 6.2, we demonstrated that significant speedups of execution time can be achieved on the customized accelerators for both the STS and SCS programs. However, the compilation time for PECompiler to compile each of the two application programs, create customized hardware, and generate execution sequence is longer than the execution time of the examples in software. The time for PECompiler to compile the STS program with the encoder netlist is about 18 minutes, and the time to compile the SCS program with the 4-bit counter netlist is about 39 minutes.

The compile time is a very important issue in generating the hardware and control software of customized accelerators for applications. However, in our current research, we did not set as the primary goal to achieve the high speed of compilation for building PECompiler. As part of future work, the compile time can be improved by optimizing the algorithms used in PECompiler, especially the algorithms used to partially evaluate the source program. A two-step scheme of partial evaluation can be adopted to first create the hardware design through the first step of partial evaluation.
based on part of program input, and then generate the control software by applying partial evaluation in the second step to the resulting program, which is produced after the first step of partial evaluation, using an extra set of the program input. By using this two-step scheme, the hardware compilation, which is conducted when the first step of partial evaluation is applied, is only performed once for a source program, and the process of generating an instruction stream, which is conducted when the second step of partial evaluation is applied, may be repeatedly performed for different input values.

For a set of input values, the compiler performs the process once to generate an instruction stream to run on the FUs that are determined during the first step of partial evaluation. For example, when compiling the STS (or SCS) program, the compiler can first determine the designs of the five FUs (eight FUs for the SCS program) required by the program in the first step of partial evaluation, and then compile the source program with a netlist input to generate the VLIW instructions. When using the same program to simulate various netlists, the compiler will only repeat the second step of partial evaluation to the program for each given netlist and generate a VLIW instruction stream for the simulations of the netlist.

In this dissertation, we do not intend to pursue the goal of fast compilation, but propose it as a topic for future work to improve the compilation time of PECompiler.

### 6.4 Effect of Partial Evaluation

The speedups on both STS and SCS program executions have been achieved through using the customized accelerators that are created by PECompiler. As described in Chapter 4, partial evaluation is the key technique used in PECompiler. The prime advantages of using this technique in our approach are to generate flattened traces of operations and expose intrinsic parallelism among the operations. As a result, PECompiler can generate the optimized hardware design of the accelerators for the applications and schedule multiple operations to be executed in parallel.

Partial evaluation can also be applied to programs executed on general-purpose
microprocessors. Compared to our approach, the use of partial evaluation for these processors can also improve the program performances by reducing the static computations during compilation of the source code. However, there are also two disadvantages to using partial evaluation in a compiler for the microprocessor. One of them is that the long flattened traces of instructions, which are generated after applying partial evaluation to the source code, can cause a high rate of instruction cache misses. The other disadvantage is that a larger number of extra variables will be needed to store the intermediate results in the flattened traces of instructions. As a result, the frequency of register spills may be increased, and a higher memory bandwidth may be needed to support the data exchanges between the microprocessor register file and data memory. These disadvantages restrict the overall speedup of each program. These two disadvantages do not occur in our design approach. On an accelerator implemented on an FCCM, the VLIW instructions are statically scheduled, and an adequate bandwidth for loading the instructions can be provided by the memory resources on the FCCM. Furthermore, PECompiler reduces the memory accesses by using a large number of data registers inside each PM of the customized accelerator.

6.5 Summary

In this chapter, we have shown two examples of using PECompiler to generate designs of specialized functional units that can be implemented on the FCCM for the STS and SCS programs. We have described the source programs, and presented the structures of the FUs generated by the compiler. Based on the assumption that the TM-2 can be used as the FCCM and each of the Altera 10K100 chips, which are used to construct the TM-2 system, can be used to implement a PM, we have also estimated the total execution cycles needed for each accelerator to accomplish the computation tasks of each application program, and showed the potential speedup that can be obtained on the FCCM compared to executions of the programs on a SUN workstation. In Chapter 7, we will examine how the performance of the accelerators changes when architectural parameters (such as topology of the top-level interconnect structure and
the structures of register files) vary.
Chapter 7

Performance Analysis

Chapter 6 demonstrated the compilation of two applications, the STS and SCS programs, using PECompiler. We also showed how the performance of both applications can be improved when the total number of PMs in the accelerators is increased. In this chapter, we discuss how other architectural features can affect the performance of the accelerators. We did not intend to exhaustively examine the impact to the performance caused by every architectural parameter, but focused on two of the most important parameters. The purpose of this chapter is to provide insight to designers for implementing both of the STS and SCS accelerators on FCCMs. We describe the hardware cost for implementing each of the architectural features, and show the impact on the performance that can be caused by varying each of the features. To conduct experiments with different architectural features and analyze the results, we assumed that TM-2 system [LGvI+97] is the FCCM platform for implementing the accelerators.

In this chapter, we focus on two architectural features: (1) the network architecture used for interconnecting all the PMs in the accelerators; and (2) the organization of the register files. These two issues are among the most important decisions of all the architectural features when implementing an accelerator on the FCCM, and have great impact on performance of the program that is executed on the accelerator.

As described in chapter 3, the architectural features can be described using our parameterized hardware model. To conduct experiments on the two issues, we created
a variety of machine models for the STS and SCS accelerators. Each of the models was constructed with a different set of parameters. Then, each of these models was used by PECompiler to generate the schedules for all the operations in a DDG that was generated for either the STS or SCS program. For each of the two programs, we conducted the tests using three DDGs that are generated by PECompiler for the simulations of three circuit netlists, respectively.

In this chapter, we use the total execution cycles (TEC), which is the number of clock cycles that are needed by the accelerator to finish all the node operations in a DDG, to evaluate the performance of a customized accelerator created for a given DDG. A lower TEC implies better performance. Section 7.1 presents three network structures that are used to construct the top-level interconnection network in the two customized accelerators. It compares the TEC values obtained for accelerators constructed using the three kinds of networks, and discusses the hardware cost for implementing the networks on the FCCM platform. Section 7.2 describes how to arrange data registers in the customized accelerator. It shows how different organizations of the registers can affect the TEC values of the accelerators.

7.1 The Top-Level Interconnection Network

The top-level interconnection network is used to provide connections for the data transfers during execution of operations on the FCCM. The performance of a network depends on the total number of input/output ports connected to all the PMs and the interconnect topology among the PMs. The design of a network is restricted by a limited number of pins on each FPGA and the available routing resources among all the FPGAs in the FCCM. On the other hand, the number of input/output ports that can be placed on each PM is also limited by the available logic resource on each FPGA. Although adding more input/output ports on each PM can improve the bandwidth of data transfers among all the PMs, the added ports will also increase the complexity of the PM-level interconnection block (see Figure 3.3) and require more logic resources on FPGAs to implement the interconnection block and more register
In this section, we discuss structures of the top-level interconnection network among all the PMs in each of the accelerators that are created for the STS and SCS applications. We also examine the hardware cost to implement each kind of network, and compare the performance of the accelerators that are constructed using the networks.

In an accelerator, the I/O capacity between each PM and the top-level interconnection network is limited by the number of available pins on the FPGA chip that is used to implement the PM. Each FPGA in the FCCM has a fixed number of pins connected to other FPGAs, and only these pins can be used to implement data ports on a PM. Furthermore, the topology inside the top-level network affects the average bandwidth of data transfers among the PMs. For the accelerators that are required by the STS and SCS applications, we designed three structures for the top-level interconnection network. The three network structures are: (1) full crossbar; (2) mesh; and (3) ring. These networks were designed for implementing the accelerators on the TM-2 system. The designs are based on the assumption that each PM of the accelerators is always mapped on one of the FPGA (Altera 10K100) chips in the TM-2 platform. The FPGA has 407 I/O pins, and there are about 200 pins that can be used to connect the chip to other FPGAs in the TM-2 after using about 200 pins for accessing the two memory banks that are connected to the FPGA.

Since each of the FPGA chips may be used to implement a single PM, each PM has 200 pins to connect to the top-level interconnection network. The 200 pins can be used as 12 16-bit I/O ports or 6 32-bit I/O ports on each PM. For the accelerator that was created for the STS program, the data width is 16 bits. Thus each of the PMs has 12 data ports to connect to all other PMs in the accelerator. For the accelerator created for the SCS program, the data width is 32 bits, and each of the PMs has six data ports connecting to the other PMs. By using time-multiplexing [BTA93] on the FPGA pins, the physical number of pins on the FPGA chip that are required to implement the data ports on the PM can be reduced. For example, if each of the physical pins is multiplexed by two, each 16-bit data port only needs eight physical
pins. Assume that the operations inside a PM are running at a clock rate of 10MHz, the I/O operations on the FPGA pins are performed at the clock rate of 20MHz. Thus, the 200 pins can be used to implement up to 25 16-bit data ports (for the STS program) or 12 32-bit data ports (for the SCS program).

In this section, we present the designs of the three networks, describe the bandwidth that each of the networks can offer, and show the performance comparisons.

### 7.1.1 The Full Crossbar Network

Figure 7.1 shows a full crossbar network architecture that interconnects $n$ PMs. This network can provide a direct connection for transferring a data item from any one PM to another PM. As it is shown in Figure 7.1(a), each PM has one input port, which connects to an output port of the network, and one output port, which connects to an input port of the network. Figure 7.1(b) presents the internal structure of the network and all possible connecting patterns between the input ports and the output ports on the network. Each of the input ports can directly link to an output port that connects to a different PM, and each output port can be chosen to link from one of the input ports that connects to a different PM. Using the full crossbar network, each PM can simultaneously send a data item to a PM and receive another data item from a PM, and the direct connections can offer low latency on the data transfers.

Although the full crossbar network can offer high bandwidth and low latency for data transfers, the hardware cost for implementing the network on the FCCM is very high. This kind of architecture may only be used to interconnect a limited number of PMs in the customized accelerator. Figure 7.1(c) depicts the physical constraints of implementing the accelerator with a full crossbar network and $n$ PMs. To provide the high bandwidth and direct data transfers between the PMs, the full crossbar network requires a great amount of routing resources between the FPGA devices on the FCCM and a large number of pins on each of the FPGA chips. Since the operations executed on the same PM may get data items from other PMs at the running time, the connection patterns between the PM input and the other $n-1$ PM outputs may be switched every clock cycle. Although FCCMs provide programmable interconnect
structures, most of them are not able to support run-time reconfiguration. Hence, the circuit to implement switching among multiple patterns (represented by the multiplexer in the figure) is usually implemented together with the associated PM on the same FPGA chip. The dashed line that is linked from the Control Unit to the multiplexer provides the select signal, which is decoded from the VLIW instruction, to choose a connecting pattern every clock cycle. Each chip needs to provide I/O pins for $n$ data ports (each data port needs a multiple number of pins) that connect to the other $n-1$ PMs, and also needs to provide pins for accessing both of the data and instruction memories. Table 7.1 shows the hardware costs of implementing
Table 7.1: Hardware Cost for Implementing the Multiplexers

<table>
<thead>
<tr>
<th>total number of PMs in accelerator</th>
<th>number of inputs on multiplexer</th>
<th>16-bit multiplexer (for STS program)</th>
<th>32-bit multiplexer (for SCS program)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LUTs</td>
<td>% of 10K100</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>32</td>
<td>0.6%</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>80</td>
<td>1.6%</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>112</td>
<td>2.2%</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>160</td>
<td>3.2%</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>208</td>
<td>4.2%</td>
</tr>
<tr>
<td>24</td>
<td>23</td>
<td>240</td>
<td>4.8%</td>
</tr>
<tr>
<td>28</td>
<td>27</td>
<td>288</td>
<td>5.8%</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
<td>336</td>
<td>6.7%</td>
</tr>
</tbody>
</table>

multiplexers with different numbers of inputs on the Altera 10K100 FPGAs.

### 7.1.2 The Mesh Network

Figure 7.2 shows the mesh network architecture that is used to construct both of the STS accelerators and SCS accelerators. All the PMs in each of the accelerators are interconnected using a two-dimensional array. Figure 7.2(a) shows the mesh network that interconnects 16 PMs in an accelerator. Each PM can directly exchange data with four neighbor PMs (one per geographical direction north, south, east, or west). For data transfers between two remote PMs that are not directly linked, the data items may pass through other PM(s). For example, a data transfer from PM2 to PM9 may be carried through PM6 and PM10, which is the shortest path. There are usually more than one shortest path between two PMs, and there are often multiple data transfers between different pairs of PMs that are carried at the same time. To achieve the maximum bandwidth on the network, PECompiler determines the best path for every data transfer between two remote PMs.

Figure 7.2(b) describes the connecting structure between a PM and its four neighbors. In the accelerator, each PM has one input port and one output port, and can send out one data item and receive one data item at a time. For the same reason that
is described in Section 7.1.1, the multiplexer needs to be mapped together with the associated PM on the same FPGA chip. When implementing each PM on a single FPGA chip, the chip needs to provide I/O pins for five data ports (four inputs and one output). Comparing to the full crossbar network, the advantage of the mesh network is that it needs less routing resources to connect the PMs and the number of data ports on each PM (FPGA chip) is independent of the total number of PMs in
the accelerator. One disadvantage of the mesh network is that the maximum latency for data transfers between remote PMs increases when more PMs are added to the accelerator. Since a PM is often on the path of data transfers among other PMs, another disadvantage of the mesh structure is that the bandwidth of the network is lower than the full crossbar. Figure 7.2(c) shows another connection structure for the mesh network. In the accelerator, each PM has two input and two output ports, but each input port only links from two other PMs and each output port only links to two other PMs. When implementing the accelerator, the two multiplexers are mapped on the same FPGA with the associated PM, and each FPGA has six data ports that connect to other FPGA chips. As it is described before, adding more input/output ports on each PM will also increase the complexity of the PM-level interconnection block and require more logic resources to implement the PM. In our experiments, the mesh structure shown in Figure 7.2(b) (each PM has one input port and one output port connected to the PM-level interconnection block) was used.

7.1.3 The Ring Network

Figure 7.3 shows the ring network architecture that is used. In each of the accelerators, each PM has a single input port and a single output port, and all the PMs are linked in a serial ring. Figure 7.3(a) shows the connections between the I/O ports on the network and the ports on the PMs. Through the network, every PM can directly send data to or receive data from two (left and right) neighbor PMs. Figure 7.3(b) shows the connecting structure among all the PMs. By using dual-direction links, the longest path between any two PMs traverses half of the total PMs in the accelerator. Compared to the full crossbar and mesh networks, the hardware cost to implement the ring network is much lower, but the latency of data transfers on the ring network is longer.

When implementing the accelerator on FCCMs, the multiplexer connected to the PM input is also mapped together with the PM on the same FPGA. The FCCM is only required to provide the routing resource to connect one FPGA chip with two neighbor FPGAs, and each FPGA chip, which is used to map a PM, needs to
provide three data ports connecting with two other PMs, no matter how many PMs are contained in the accelerator.

### 7.1.4 Performance Comparisons

To compare the performance of the three network architectures, we tested a variety of accelerators that are constructed using the three kinds of networks and different numbers of PMs. For each of the accelerators, a machine model was created by PECompiler to generate the schedules for all the operations. The TEC value derived by the compiler based on each machine model denotes the speed of the accelerator represented by the model. The rest of this section shows the comparisons of the TEC values among the three networks for the STS and SCS programs, respectively.

#### The STS Application

As described in Chapter 6, three circuits (encoder, crc, and 8-bit counter) were used to conduct the experiments of using PECompiler to generate the accelerator for the STS program. For each of the three circuits, we show the comparisons of TEC values of the application running on the accelerators that were constructed using a full crossbar,
mesh, and ring networks, respectively. Figure 7.4 shows the length of schedule results that are generated by PECompiler for three kinds of the STS accelerators when the encoder is used as the circuit netlist. Each type of accelerator is constructed using one of the three network architectures. For each network architecture, four accelerators that consist of four, eight, twelve, or sixteen PMs were tested. Each PM has one input port from the top-level interconnection network and one output port to the network. As shown in the figure, the TEC values for the cases of the full crossbar architecture are always lower than that of the other two architectures, which means finishing all the operations within the shortest time. In the cases of the mesh network, the TEC values are lower than that of the ring network. The executions on the accelerators with the ring network are always the slowest. When containing four PMs, the TEC values for all three kinds of the accelerators are about the same. When increasing the total number of PMs to eight, all the TEC values decrease because the extra FUs can be used to execute more operations in parallel. The differences among the TEC values of the application running on the three accelerators become apparent because of differences of the latencies of data transfers and the bandwidths. When adding four more PMs in the accelerators, the TEC values of the application running both on the accelerator with the full crossbar network and the accelerator with the mesh network are decreased by 27% and 20%, respectively, but the TEC value of the operations running on the accelerator constructed using the ring network decreases by less than 3%. When further increasing the number of the PMs from twelve to sixteen, the TEC values of the application running on the accelerators using the full crossbar or mesh networks are only reduced by less than 1%, and the TEC value of the application running on the accelerator using the ring network is increased by about 24%. The main reason for little improvement on both of the full crossbar and the mesh cases is because the total number of the data transfers on the networks increases when adding more PMs in the accelerators, and the bandwidth of the networks becomes the primary constraint for data communications among all the PMs. For the ring network case, both the bandwidth and the long latency of data transfers make the performance even worse. Therefore, for the STS program with the circuit netlist
Figure 7.4: Performance of STS Program with the encoder circuit

Figure 7.5: Performance of STS Program with the crc Circuit

Figure 7.6: Performance of STS Program with the 8-bit counter Circuit
of the encoder, the accelerator should be constructed with 12 PMs. Although the performance of the accelerators with a mesh network is lower than the performance of the accelerators with a full crossbar network, the hardware cost to implement the mesh network is lower than the full crossbar network. When implementing an accelerator, designers may choose the mesh network as the tradeoff between performance and the cost. For example, the 12-PM accelerator with a full crossbar network needs eleven input ports and one output port on each PM, and its TEC value is 765; a 12-PM accelerator with a mesh network needs four input ports and one output port on each PM, and its TEC value is 957. The latter takes about 25% more cycles to finish the execution than the former, but the former needs seven more input ports on each PM compared to the latter. Increasing the number of ports on PMs can significantly increase the complexity of the PM-level interconnection block, which means a much higher hardware cost to implement the PM.

Similarly, Figure 7.5 shows the performance comparisons among the different accelerators that were constructed for the STS program with the crc circuit as netlist input. For the accelerators that consist of four, eight, twelve, or sixteen PMs, the accelerators with a full crossbar network always have the lowest TEC values, the accelerator with the ring network have the highest TEC values. The results of the experiments show that the accelerator constructed with 12 PMs is a good design choice. The performance can be further improved by adding more PMs (using the full crossbar network), the I/O pin limit (about 200 pins) on each FPGA chip can only support 12 data ports. Another interesting point is that the performance of the accelerator with 16 PMs and a mesh network is slightly better than the one with 12 PMs and a full crossbar network, the former has a TEC value of 456, and latter has a TEC value of 475. However, the former needs more FPGAs and RAM resources to implement four more PMs. Considering the cost of both accelerators, the accelerator with 12 PMs and a full crossbar is a better choice. Figure 7.6 shows the performance comparisons for the STS program with the 8-bit counter the as netlist input. The changes of the performance with respect to the three network architectures with different numbers of PMs are similar to the cases of simulating the circuit of encoder. The
accelerator constructed with 12 PMs is a good choice for the application. The results in both Figure 7.5 and Figure 7.6 also show that the performance of a full crossbar network is only slightly better than the performance of a mesh network. Therefore, the latter can be a good tradeoff between the performance and the hardware cost when implementing an accelerator.

The SCS Application

In the same way, a variety of customized accelerators were created for the SCS program using the three network architectures and four different numbers of PMs. We also tested the accelerators using three DDGs generated by PECompiler using three circuit netlists (4-bit counter, 6-bit adder, and 20-inverter chain). Figure 7.7 shows the TEC values of the SCS accelerators when scheduling the DDG generated for the 4-bit counter circuit. It is similar to the STS accelerators, the full crossbar network has the best performance and the ring network takes the longest time to finish the execution of all the operations on the accelerators. A system containing four PMs produces the TEC values that are about the same for all three kinds of the accelerators. The TEC values decrease significantly after adding four more PMs in the accelerators, and the decreases saturate when twelve PMs are used in the accelerators. When the number of PMs increases further, performance can no longer be improved. The main reason for this saturation is that the limited bandwidth on the top-level interconnection network cannot handle increased requirements of data transfers among different PMs. For example, in the full crossbar network, each PM has one output port, and can only provide one operand for the data transfers to other PMs. To solve the problem, more I/O ports on each PM will be needed, and the cost to implement the network will be increased. These results suggest that the accelerator with 12 PMs and a full crossbar network is the best choice for the program to simulate the 4-bit counter.

Figure 7.8 shows the performance comparisons of TEC values for the SCS program to simulate the 6-bit adder circuit using accelerators constructed with different numbers of PMs and three kinds of networks. For accelerators with the same number
Figure 7.7: Performance of SCS program with the 4-bit counter circuit

Figure 7.8: Performance of SCS program with the 6-bit adder circuit

Figure 7.9: Performance of SCS program with the 20-inverter circuit
of PMs, the one with a full crossbar network has the lowest TEC value, and the one with a ring network has the highest TEC value. The results also suggest that the accelerator with 12 PMs and a full crossbar network is the best choice for simulating the circuit. Figure 7.9 shows the same comparisons for the SCS program to simulate the 20-inverter circuit. The best choice is still the accelerator with 12 PMs and a full crossbar network. The figure also shows that when using the ring network the accelerator with 12 PMs has performance worse than both of the accelerator with 8 PMs or the accelerator with 16 PMs. This is because the memory access pattern for the circuit simulation application requires a higher network bandwidth to support the data transfers among all the PMs in the accelerator.

Similar to the STS application, in most of the test cases, the performance of an accelerator with a mesh network is slightly worse than the performance of an accelerator with a full crossbar network. Therefore, the mesh network may be used to implement an accelerator for the SCS application.

### 7.2 Register Files

This section discusses the considerations of the register files for designing the customized accelerators for both the STS and SCS programs. It first describes different ways to arrange the register files in the accelerators, and then shows how different register file structures affect the performance of the accelerators. A variety of register file structures are tested and the results are analyzed. It also examines the hardware costs for implementing the different register file structures on the FCCM.

#### 7.2.1 Register File Implementation

Register files can be implemented using RAM resource or lookup tables in FPGA chips. For example, each Altera 10K100 FPGA provides 12 embedded array blocks (EABs) that are on-chip RAM blocks. These EABs can be used to implement register files. Each EAB block can be in one of four configurations: (1) 256x8bits; (2) 512x4bits; (3) 1024x2bits; (4) 2048x1bit. To implement multiple register files on each
FPGA, we use the first configuration (256x8bits). For the STS program, the data width is 16 bits, thus the maximum number of register files that can be implemented on each 10K100 chip is six; for the SCS program, the data width is 32 bits, then each chip can implement up to three register files. Another example is that Xilinx XC4000 FPGAs allow users to use all the lookup tables as RAM resources. Each lookup table on the XC4000 chip can be used as a 16x1bit register file. Thus the cost for implementing a 16x16bits register file for STS program is 16 lookup tables (8 CLBs). Since the TM-2 is used as the FCCM platform, we consider designs of register files in the STS and SCS accelerators based on the available EABs on the Altera 10K100 FPGA chip. As each FPGA only has a fixed number of EABs (12 EABs in 10K100 chip), implementing register files for the accelerator is expensive, and the maximum number of registers that can be implemented in each PM is limited to 12 EABs.

The structure of register files in a PM can be characterized by two parameters. The first parameter is the total number of data registers. The second parameter is the total number of read/write ports on the register files that can be accessed for loading source operands of the operations and storing the execution results. Both of the parameters have direct impact on the schedules of operations executed on the PM. The total number of available registers in a PM determines the maximum number of data items that can be stored inside the register files at a time. At any time when all the registers are used by previously scheduled operations, PECompiler may spill the contents of the registers into memory and allocate the registers to the next operation. However, register spills can postpone the execution of the next operation, and consequently increase the TEC value of the application.

The total number of read/write ports on the register files determines the maximum number of source operands and execution results that can be loaded from or stored into the registers at each clock cycle. When scheduling a DDG, there are usually a number of operations that need to fetch the operands at the same time, and often more than one of these operands is stored in the same register file. When the number of ports cannot meet the needs of all the operations, PECompiler has to schedule some of the operations at later cycles. This kind of delay can increase the TEC value.
of the accelerator.

When adding more read/write ports on each register file, the delay cycles due to multiple operations competing on accessing the same register files can be reduced. Then, the number of clock cycles that are required to finish all the DDG operations will be reduced, but the hardware cost to implement these ports will be increased. First, the clock cycle time will be increased because of using time-multiplexing to access the EABs multiple times. Second, the number of instruction bits that are required by all the read/write ports of the register files will be increased. As a result, a higher memory bandwidth will be used to fetch the longer instructions. The decision on the register file structure may be derived based on the tradeoff between the performance and the hardware cost. Section 7.2.2 will show the TEC values for both the STS and SCS accelerators when different register file structures are used.

### 7.2.2 Register Usage of STS and SCS Applications

A variety of ways to arrange data registers have been tested by creating the machine models for the accelerators using different register file structures. These models are used by PECompiler to generate the schedules of all the operations on the FCCM. For each of the STS and SCS examples, four kinds of machine models have been created. The first kind (case 1) contains three register files in every PM and each register file has two read ports and two write ports; the second kind (case 2) contains three register files in every PM and each register file has four read ports and four write ports; the third kind (case 3) contains six register files in every PM and each register file has a single read port and a single write port; and the fourth kind (case 4) contains six register files in every PM and each register file has two read ports and two write ports. For each of the four kinds, five different sizes of register files (16, 32, 64, 128, or 256 registers in each register file, respectively) have been tested. All the machine models are constructed with four PMs that are interconnected by the full crossbar network.
The STS Application

For the STS program, we conducted the experiments using the three netlists (encoder, crc, and 8-bit counter). PECompiler schedules the DDGs (described in Table 6.1) that were generated for the simulations of the netlists based on the machine models. The TEC values derived by PECompiler for the three DDGs are shown in Figure 7.10, Figure 7.11, and Figure 7.12 respectively.

In each of the three figures, the four curves show the TEC values of four different cases for various sizes of register files. The points marked on both of the lines corresponding to case 1 and case 2 represent the TEC values of the accelerators that contain three register files in each PM, and the total numbers of registers on the accelerators of both cases are always equal no matter what size of register files are used. The points on the other two lines (case 3 and case 4) represent the TEC values of the accelerators that contain six register files in each PM, and the accelerators of the two cases also have the same number of registers. On the other hand, all the accelerators associated with the points on both solid lines have six ports for reading data and six ports for writing data in each PM, and all the accelerators associated with the points marked on both dashed lines have twelve read and twelve write ports in each PM.

As shown in the three figures, when each register file contains only 16 registers, the TEC values of the four cases are all very high. This is because most of the cycles are spent on spilling register contents. All the TEC values of the four cases are reduced when more registers are implemented in each register file. However if the total number of registers in each PM is more than a threshold, which are represented by points A and B in each figure, the TEC values on four curves show only minor changes when further increasing the total registers. The point A represents the model of an accelerator that each PM contains six register files, each of which has two ports and 64 registers for the DDG of the encoder circuit or 32 registers for the other two DDGs. The point B represents the model of an accelerator where each PM contains three register files, each of which has four ports and 128 registers for the DDG of the
Figure 7.10: Performance of STS Program with the encoder Circuit

Figure 7.11: Performance of STS Program with the crc Circuit

Figure 7.12: Performance of STS Program with the 8-bit counter Circuit
encoder circuit or 64 registers for the other two DDGs. Both model A and model B have the same number of registers and the number of read/write ports in each PM. The experimental results show that the TEC values of both cases are close to each other, but the hardware costs to implement the two cases are different. To implement the register file structures on the Altera 10K100 chips in the TM-2, the accelerator of model A implements 2 read/write ports in each register file by time-multiplexing the accesses on the EABs by 2, and the accelerator of model B needs to implement 4 read/write ports by time-multiplexing the accesses on the EABs by 4, in which a longer clock cycle time is needed. As a result, the accelerator of model B may run at a lower speed compared to the accelerator of model A. Since the size of the three register files in model B is twice the size of the six register files in model A, each of the twelve ports on the register files in model B needs one more instruction bit for addressing registers. That is, model B requires 12 more bits than model A in each instruction. Therefore, model A is a better choice than model B.

The results also show that a total number of 12 register file ports in each PM is a good design choice for the application. Although adding more ports on the register files can reduce the TEC value, the number of instruction bits for register address lines would increase, and the clock cycle time of the accelerator would be longer. Comparisons between accelerators of case 1 and case 2, and between accelerators of case 3 and case 4 indicate that the TEC values were only reduced by a small portion even when the number of ports on the register files was doubled.

The SCS Application

We also conducted the same experiments for the SCS program using three circuits (4-bit counter, 6-bit adder, and 20-inverter) as the netlist input. Figure 7.13, Figure 7.14, and Figure 7.15 show the results generated by PECompiler for scheduling the DDGs (described in Table 6.3) of the three circuits.

The same four structures of register files are tested and the TEC values are also represented by the points marked on the four curves. These four curves show similar behaviors of the accelerators when different sizes of register files are used for the four
Figure 7.13: Performance of SCS Program with the 4-bit counter Circuit

Figure 7.14: Performance of SCS Program with the 6-bit adder Circuit

Figure 7.15: Performance of SCS Program with the 20-inverter Circuit
cases, but less register pressure because each PM in the SCS accelerator contains a smaller number of FU input ports. The points A and B also suggest choices for the structures of the register files. Notice that the number of registers in the SCS accelerator is 32 bits wide. To implement design choice A or B, all the EABs in each of the FPGA chips must be time-multiplexed by 4. However, to implement accelerator B, twelve more instruction bits will be needed compared to the implementation of accelerator A. These extra instruction bits will be used for addressing the register files that are twice as big as the register files in accelerator A. Therefore, model A is still a better choice than model B.

The experimental results of both the STS and SCS applications have demonstrated that the performance of an accelerator constructed with six register files, each of which has one read port and one write port, is about the same as that of an accelerator with three register files, each of which has two read ports and two write ports. The total numbers of registers and read/write ports are the same in both accelerators, but the former contains more register files with fewer ports and the latter contains less register files with more ports. For implementing register files on FPGAs, the model containing more register files with fewer ports is capable of running at a clock speed that is higher than the clock speed for the model containing fewer register files with more ports. This means that PECompiler is successful in using the types of hardware resources available on FPGAs, and tolerates multiple register files with few ports, achieving high performance on the FCCM for the two applications.

7.3 Summary

In this chapter, we have examined three architectural features of the two customized accelerators that are built for the STS and SCS applications, respectively. In Section 7.1, we have first proposed three network architectures, and described the hardware cost for implementing them on the FCCM. We have then compared the performance of the accelerators that are constructed using the three networks. For the case of using the TM-2 to accelerate executions of both applications, the results indi-
icated that full crossbar networks always have the best performance among the three kinds of networks. However, since the performance of mesh networks is just slightly lower than the performance of full crossbar networks, a mesh network can be a good tradeoff between the performance and the hardware cost. Since these results were generated by PECompiler for the specific application programs based on the TM-2 FCCM, it is difficult to predict if mesh networks would be good design choices for other applications. In Section 7.2, we have discussed structures of register files inside PMs of the accelerators. We have described two important parameters: the total number of registers in each register file; and the total number of read/write ports on register files in each PM. We have shown the results of experiments that were conducted using different structures of register files. For both the STS and SCS applications, the results suggest that six register files, each of which has one read port and one write port, is a good design choice. Each register file should contain 32 to 64 registers. When implementing a larger number of registers in each register file, or more read/write ports on each register file, only minor improvements can be achieved on the performance of the accelerators.
Chapter 8

Conclusions and Future Work

8.1 Thesis Summary

Designing a configurable computer system is a complex and time-consuming task. To achieve high execution speed of an application program on such a system, designers need customized hardware structures and specialized processing units to be implemented on an FCCM platform. For a given application, tremendous effort is needed to design the customized hardware structure of the configured computing machine and the control software for executing the program operations.

To automate the design process, we have proposed a new methodology based on partial evaluation for generating designs of customized accelerators for computationally intensive programs. In this dissertation, we have described a VLIW machine architecture that we designed for configurable compute accelerators. This architecture can be used to build different accelerators for a variety of applications. To realize the automated design process, we have created a parameterized machine model to describe the hardware designs. Using this model, users can use parameters to specify specialized architectural features in every customized accelerator. We have designed and implemented a compiler, PECCompiler, that can create the specialized hardware design and control software for each given application through compiling the source program of the application. By applying partial evaluation, PECCompiler can exploit parallelism in the source program, generate designs for specialized functional units.
and optimized data dependency graphs (DDGs) for computations to be accelerated on the FCCM, and schedule operations to be executed on the customized accelerator. As examples of using PECompiler to generate specialized designs, we have shown the STS and SCS applications and the designs obtained by compiling the source programs with part of program input. We have also discussed how different architectural parameters can affect the performance of application programs.

### 8.2 Contributions

The major contributions of the thesis are:

- A new design methodology using partial evaluation to achieve automated design of field-programmable custom compute accelerators. Unlike previous uses of partial evaluation in FPGA systems, our approach uses part of the input to perform specialization to the source program.

- A compiler, PECompiler, that is based on the partial evaluation technique was developed to generate the hardware designs of the customized compute accelerators for various application programs.

- A parameterized VLIW architecture was designed for field-programmable compute accelerators that can be implemented on FCCM platforms.

- A design flow was created to determine the hardware configuration and control software of a customized accelerator during compilation of each given source program.

- A method was developed for extracting FU designs during the compilation of source programs.

- A heuristic algorithm was developed for scheduling DDG nodes to run on the accelerator, and an efficient method to search for the best schedule for each node operation was created.
Since register allocations for the operations need to be performed at the same time as the operations are scheduled, we developed a new algorithm for allocating registers to the operations executed on the accelerator.

We demonstrated speedup for two application programs obtained on the customized accelerators that were designed by PECompiler.

Three architectures (full crossbar, mesh, and ring) for the interconnect network among all the PMs were designed and their impact on performance of the accelerator was studied. The experimental results on the two applications showed that the mesh network can offer performance nearly as good as the full crossbar and much better than the ring network, the cost to implement the mesh is much lower than the full crossbar network.

The structures of register files were studied. The experimental results for the two applications showed that the performance of the accelerator containing more register files with fewer ports is about the same as that of the accelerator containing fewer register files with more ports, while the cost for implementing the former in FPGAs is lower than the cost of the latter.

8.3 Future Work

There are several goals to be pursued in future research:

- A current on-going research effort is to implement the two customized accelerators that are designed for the two examples described in Chapter 6. The accelerators will be implemented on the Transmogrifier-2 system [LGvI+97] based on the design results that are generated by PECompiler.

- It is important to design a configurable structure for the control unit in the field-programmable compute accelerators. With this structure, a code generator can be built to produce instructions for every customized accelerator designed by the compiler.
• One of the most urgent tasks is to find an efficient algorithm to realize automated generation of all the architectural parameters for the machine models based on the requirements of source programs and information about available hardware resources on the provided FCCM platform.

• It is also important to improve the Hardware Functional Unit Extraction block in the PECompiler to determine optimized sizes and number of I/O ports of functional units and handle the partition task when it is necessary.

• The algorithms that are used by PECompiler for applying partial evaluation to source code need to be improved to reduce the compile time of application programs. The compilation of hardware will only be performed once for a given application program, and the control software will be generated once for each set of input values.

• More research is needed to maximize the bandwidth of the data communications among the memory blocks, register files, and functional units within each PM, while keeping the hardware cost of the PM-level interconnection block relatively low.

• Another interesting topic is run-time reconfiguration of the FCCM platforms. There may be multiple hardware routines that are invoked to run on the customized accelerator at different times. Each of them usually needs different kinds of specialized processing units. To efficiently use the available hardware resources on FCCM platforms to exploit whatever parallelism existing among operations, the compiler can schedule the partial or complete reconfiguration of the customized accelerator mapped on the FCCM platform.

Work in the above research fields is very important for improving the design of PECompiler, and the results that are expected to be produced will enable the compiler to generate designs of customized accelerators with a better quality for application programs.
References


Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1996.


[WTS+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev

Appendix A

The Programming Language for Applications

A.1 The Source Programming Language

The source code of applications is written in a restricted subset of the C programming language. Figure A.1 shows a list of the statements that are currently supported by PECompiler.

| 1. assignment; | 4. while statement |
| 2. if statement; | 5. function call |
| 3. for statement; | 6. printf statement |

Figure A.1: The Statements Supported by the Compiler

Figure A.2 shows a list of operators currently supported by PECompiler.

| Arithmetic Operators: | +, -, *, /, %; |
| Relational and Logic Operators: | >, >=, <, <=, !=, ==, &&; |
| Bitwise Operators: | &!, ^, <<, >>, ~. |

Figure A.2: The Operators Supported by the Compiler

The data types that are currently supported by PECompiler include integer and float-
point. The compiler can handle the data types both in scalar and array formats.

A.2 A Special Format

Currently, we use a special format for the source program. This format is similar to the Lisp programming language, but the semantics of the format remains the same as the C programming language. The format is called C in Lisp Format (CiLF). The reason for using CiLF is mainly because part of PECompiler, including the partial evaluator, is written in lisp. By using CiLF, we were able to easily create a parser for PECompiler in a short time.

The definitions of the CiLF format are given in Figures A.3, Figures A.4, and Figures A.5. Definition 1 shows that a program consists of a list of declarations for global variables and a number of functions. Definition 2 shows the format of these global declarations. Since a program may or may not have global variables, this part of the declarations are optional. Definition 3 shows how a single variable is declared in the CiLF format. Definitions 4, 5, and 6 describe the format of a function. Definition 7 shows the format of local variable declarations. Definitions 8 to 16 illustrate formats of six code blocks that are currently supported by PECompiler. Definition 17 shows the formats of three kinds of user annotations in the application program. A program written in the CiLF format is organized as a recursive list, and based on this list structure the compiler can be easily create a syntax tree for the program.

Before using PECompiler to compile the source code of each application, we converted the code from the original format of the C programming language to the CiLF format. The user annotations are also inserted into the source code at this stage. This conversion is currently done by hand, but it can be implemented by a program tool.
1. Format of a source program is defined as:

\[
\text{(<global-declarations> <function-1> <function-2> .......)}
\]

where \text{<global-declarations>} is a list of global variable declarations (optional);
each of \text{<function-1>}, \text{<function-2>}, ....... is a function.

2. Format of \text{<global-declarations>} is defined as:

\[
\text{(declg <declaration-1> <declaration-2> .......)}
\]

where \text{declg} is a keyword, indicating that the declared variables are global;
each of \text{<declaration-1>}, \text{<declaration-2>}, ....... is a variable declaration.

3. Format of a variable declaration is defined as:

3.1. For a scalar variable: \text{(Vscn VarName VarType InitValue)}

where \text{Vscn} is a keyword to indicate a scalar variable;
\text{VarName} is the name of the variable;
\text{VarType} is the type of the variable;
\text{InitValue} is an initial value of the variable.

3.2. For an array variable: \text{(Varn VarName (s_1 s_2 ... ) VarType InitValue)}

where \text{Varn} is a keyword to indicate an array variable;
\text{VarName} is the name of the variable;
\text{(s_1 s_2 ...)} specifies sizes of all the array dimensions;
\text{VarType} is the type of array elements;
\text{InitValue} is initial values of all the array elements;

4. Format of a function is defined as:

\[
\text{(Func-Name} <Parameter-List> <Function-body> <Return-Value>)}
\]

where \text{<Func-Name>} is the name of the function;
\text{<Parameter-List>} is a list of parameter declarations;
\text{<Function-body>} the body of the function;
\text{<Return-Value>} is the data type of the return value (void, int, or float).

5. Format of \text{<Parameter-List>} is defined as:

\[
\text{((p_1 type_p1) (p_2 type_p2) .....)}
\]

where \text{p_1}, \text{p_2}, ... are names of parameters;
\text{type_p1}, \text{type_p2}, ... specify data types of the parameters.

6. Format of a function body is defined as:

\[
\text{(<local-declarations> <codeblock-1> <codeblock-2> .......)}
\]

where \text{<local-declarations>} is a list of local variable declarations (optional);
each of \text{<codeblock-1>}, \text{<codeblock-2>}, ....... is a code block.
7. Format of `<local-declarations>` is defined as:

\[(\text{decl} \ <\text{declaration-1}> \ <\text{declaration-2}> \ldots)\]

where `decl` is a keyword, indicating that the declared variables are local.

8. There are six kinds of code blocks that are defined in CiLF format:

1. basic block;
2. if block;
3. for block;
4. while block;
5. printf block;
6. subroutine calls

9. A basic block is defined as:

\[(\text{bblk} \ \text{assign}_1 \ \text{assign}_2 \ldots)\]

where `bblk` is a keyword, indicating a basic block; each of `assign_1`, `assign_2`, \ldots is an assignment.

10. An assignment is defined as:

\[= (\text{Vscn VarName}) \ \text{NewValue}\] for assigning to a scalar variable;

\[= (\text{Varn VarName (idx}_1 \ idx}_2 \ldots)) \ \text{NewValue}\] for assigning to an array.

where `NewValue` represents the value to be assigned to the variable `VarName`;
each of `idx_1`, `idx_2` is a value that is used as an index of an array dimension.

11. A value can be in one of following formats:

1. a constant (the value is C): `(\text{Valu C})`;
2. a scalar variable: `(\text{Vscn VarName})`;
3. an array variable: `(\text{Varn VarName (idx}_1 \ idx}_2 \ldots))`;
4. an expression: `(\text{Expr Op operand}_1 \ \text{operand}_2)`.

which `Value` and `Expr` are keywords;
`Op` specifies an operator;
`operand_1` and `operand_2` are two operands, each of them is recursively defined by the above four formats.

12. An if block is defined as:

\[\text{(if ConditionExpr TrueBranch FalseBranch)}\]

where `if` is a keyword;
`ConditionExpr` is an expression, specifying the branch condition;
`TrueBranch` is a list of code blocks that are executed when condition is true;
`FalseBranch` is a list of code blocks that are executed when condition is false.

Figure A.4: The CiLF Format (Part II)
13. A for block is defined as:

\[(\text{for } \text{lnits } \text{LoopCondition } \text{LoopBody } \text{LoopVariation})\]

where \textit{for} is a keyword;
\textit{lnits} is a basic block with any number of assignments;
\textit{LoopCondition} is an expression, specifying the loop condition;
\textit{LoopBody} is a list of code blocks, representing any number of statements;
\textit{LoopVariation} is a basic block with any number of assignments.

14. A while block is defined as:

\[(\text{while } \text{LoopCondition } \text{LoopBody})\]

where \textit{while} is a keyword;
\textit{LoopCondition} is an expression, specifying the loop condition;
\textit{LoopBody} is a list of code blocks, representing any number of statements;

15. A printf block is defined as:

\[(\text{printf } \text{OneVariable})\]

where \textit{printf} is a keyword;
\textit{OneVariable} specifies either a scalar variable or an array element.

16. A subroutine call is defined as:

\[(\text{call } \text{FuncName } \text{ParameterList})\]

where \textit{call} is a keyword;
\textit{FuncName} is the name of the called subroutine;
\textit{ParameterList} is a list of values that are passed to the called subroutine.

17. Three user annotations are defined as:

(1) applying partial evaluation:
\[(\text{PE } \text{CodeBlocks})\]

(2) mapping to the FCCM platform:
\[(\text{HF } \text{CodeBlocks})\]

(3) designing FUs for annotated code:
\[(\text{Hwimp } \text{CodeBlocks})\]

where \textit{PE}, \textit{HF}, and \textit{Hwimp} are keywords;
\textit{CodeBlocks} is a list of annotated code blocks.

Figure A.5: The CiLF Format (Part III)