1-1-2016

A Framework for the Design and Analysis of High-Performance Applications on FPGAs using Partial Reconfiguration

Richard D. Anderson

A framework for the design and analysis of high-performance applications on FPGAs using partial reconfiguration

By

Richard D. Anderson

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2016

A framework for the design and analysis of high-performance applications on FPGAs

using partial reconfiguration

By

Richard D. Anderson

Approved:

Donna S. Reese
(Major Professor)

Susan M. Bridges
(Committee Member)

David A. Dampier
(Committee Member)

Edward A. Luke
(Committee Member)

Gerald R. Morris
(Committee Member)

T.J. Jankun-Kelly
(Graduate Coordinator)

Jason M. Keith
Dean
Bagley College of Engineering

Name: Richard D. Anderson

Date of Degree: August 12, 2016

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Donna S. Reese

Title of Study: A framework for the design and analysis of high-performance applications on FPGAs using partial reconfiguration

Pages of Study: 190

Candidate for Degree of Doctor of Philosophy

The field-programmable gate array (FPGA) is a dynamically reconfigurable digital logic chip used to implement custom hardware. The large densities of modern FPGAs and the capability of the on-the-fly reconfiguration has made the FPGA a viable alternative to fixed logic hardware chips such as the ASIC.

In high-performance computing, FPGAs are used as co-processors to speed up computationally intensive processes or as autonomous systems that realize a complete hardware application. However, due to the limited capacity of FPGA logic resources, denser FPGAs must be purchased if more logic resources are required to realize all the functions of a complex application. Alternatively, partial reconfiguration (PR) can be used to swap, on demand, idle components of the application with active components. This research uses PR to swap components to improve the performance of the application given the limited logic resources available with smaller but economical FPGAs.

The swap is called ”resource sharing PR”. In a pipelined design of multiple hardware modules (pipeline stages), resource sharing PR is a technique that uses PR to improve the performance of pipeline bottlenecks. This is done by reconfiguring other pipeline stages, typically those that are idle waiting for data from a bottleneck, into an additional parallel bottleneck module. The target pipeline of this research is a two-stage “slow-to-fast” pipeline where the flow of data traversing the pipeline transitions from a relatively slow, bottleneck stage to a fast stage. A two stage pipeline that combines FPGA-based hardware implementations of well-known Bioinformatics search algorithms, the X! Tandem algorithm and the Smith-Waterman algorithm, is implemented for this research; the implemented pipeline demonstrates that characteristics of these algorithm.

The experimental results show that, in a database of unknown peptide spectra, when matching spectra with 388 peaks or greater, performing resource sharing PR to instantiate a parallel X! Tandem module is worth the cost for PR. In addition, from timings gathered during experiments, a general formula was derived for determining the value of performing PR upon a fast module.

Key words: Bioinformatics, FPGA, Partial Reconfiguration, Proteomics, Smith-Waterman, Tandem Mass Spectrometry, Virtex-6, Xilinx, X! Tandem

DEDICATION

To my elders that had begun this journey with me and went to live among their ancestors along the way.

ACKNOWLEDGEMENTS

I would like to thank God for blessing me with the opportunity to continue my education, for providing a strong base of loving and caring family and friends, for placing elders around me to help guide me down the right path, and for giving me the strength and desire to push forward even when I wanted to quit. Without God, this would have never happened.

To my elders Inez Shelton, Helen Breckenridge, Mary Boudreaux, and Paul Boudreaux, although you are no longer here, thank you for your prayers and encouragement. To Helen Boudreaux, mom I strive to make you proud. You and I are mirrors so I know that you know what I am thinking. Thank you and I love you. To Rik Anderson, dad your random calls to see I how was doing always made the stress of this research work not so bad. To Kenneth "Pops" Boudreaux, our Saturday morning conversations about the Lakers gave me something to look froward to. They were great beginnings to long days of work. To Dr. Daniel Omotosho Black, Mwangi Kwesi, Mawu Olumoroti, Azikewe Nzuriwatu, Mombé Banga, and Tcheseret Ifadaré, thank you for constantly pushing me to persevere and to not let my work be in vain. To Jeremy Davis, who knew that the man I met in an AI class would become one of my best friends. Thanks for your support and your help with the data conversion for this research. The code you wrote came in handy many times over. To Dr. Cindy Bethel, thank you for your optimism and motivational support. To family

and friends not mentioned above, thank you for your love and for your desire to see me succeed. You all have helped me keep my motivation.

To Dr. Donna Reese, years ago a friend affectionately called you “mama Donna” and when you became the Computer Science and Engineering Department head, it was clear why. Since you hired me as an Instructor and then later became my major adviser, “mama Donna” is the best way to describe the capacity in which you have been to me these past four years. You were the mama sandpiper constantly pushing me toward the beach so that I could feast on the clams of a PhD. Dr. Bridges, thank you again for your encouragement and affirmations and, thank you for remaining a member of my dissertation committee even while in retirement. Your input was extremely valuable. To Dr. David Dampier and Dr. Edward Luke, thank you for identifying the technical details that I should have expanded upon or have overlooked. Jerry thank you for agreeing to serve on my dissertation committee, your hardware insight was invaluable. As a side note, you were right to say that trying to graduate in two years was ambitious. However in hindsight, your prediction of five years was also ambitious. Therefore sir, I mean Dr., I do not concur. To Natividad “Lisa” Morris, thank you for sharing Jerry with me. I love and miss you. Yogi, thanks. You were my standard for patience and a good friend.

Lastly, to the National Science Foundation, CyberCorps: Scholarship For Service program, thank you for funding two years of my PhD study and enabling the opportunity for me to intern at the Sandia National Laboratory.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

INITIALISMS AND ABBREVIATIONS

Below is a list of initialisms and abbreviations used throughout this dissertation.

AES Advanced Encryption Standard

ALM Adaptive Logic Modules (Aletra Corp.)

ASIC Application Specific Integrated Circuit

bps Bits per second

BRAM Block Random-Access Memory

CLB Configurable Logic Block (Xilinx Inc.)

CMD Command Register (Xilinx Inc.)

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DCM Digital Clock Manager

DDR Double-Data Rate

DSP Digital Signal Processor

DU Data Unit

EST Earliest Start Time

EFT Earliest Finish Time

F+V Fixed-plus-Variable

FAR Frame Address Register (Xilinx Inc.)

FDRI Frame Data Register, Input Register (Xilinx Inc.)

FIFO First-In First-Out

FPGA Field Programmable Gate Array

Gbps Gigabits Per Second; 10^9 bits per second

HDL Hardware Description Language

HMAC Hash-based Message Authentication Code

HPC High Performance Computer/Computing

ICAP Internal Configuration Access Port

IPDP Intra Post-Dominator Path

IP Internet Protocol

LAB Logic Array Block (Aletra Corp.)

LUT Look-Up Table

LZSS Lempel-Ziv-Storer-Szymanski

LZW Lempel-Ziv-Welch

Mbps Megabits Per Second; 10^6 bits per second

MD5 Message-Digest version 5

MGT Multi-Gigabit Transceiver

MHz Mega Hertz; 10^6 Hertz

μ**s** Microseconds; 10^{-6} seconds

ms Milliseconds; 10^{-3} seconds

MUX Multiplexer

ns Nanoseconds; 10^{-9} seconds

PAP Placement-Aware Probability

PCIe Peripheral Component Interconnect Express

PR Partial Reconfiguration/Reconfigurable

PRM Partial-Reconfiguration/Reconfigurable Module

PRR Partial-Reconfigurable Region

RAM Random-Access Memory

ROM Read-Only Memory

SATA Serial Advanced Technology Attachment (or Serial ATA)

SHA Secure Hash Algorithm

SRAM Static Random-Access Memory

SODIMM Small Outline Dual In-Line Memory Module

SSD Solid-State Drive

TCP Transmission Control Protocol

UDP User Datagram Protocol

VAPRES Virtual Architecture for Partially Reconfigurable Embedded Systems

VEH Vulnerability Emulation Handlers

WSN Wireless Sensor Network

XST Xilinx Synthesis Technology

CHAPTER 1

INTRODUCTION

The use of field programmable gate array (FPGA) technology in high-performance computing (HPC) is increasing because the computational capacity and I/O connectivity of FPGA devices has improved in recent years [6, 58, 60]. Historically FPGAs were restricted to a small set of HPC applications because acquiring the quantity of FGPAs required to support large-scale HPC applications was economically infeasible. Regardless of their known advantages, FPGAs were originally considered an unlikely option for HPC because the performance of the microprocessor and custom arithmetic hardware solutions (*i.e.*, Application Specific Integrated Circuits) was superior to the processing capabilities of FPGAs [58, 60]. However, advances in technology have allowed for architectural enhancements to the logic structure and communication infrastructure of FPGAs thus improving their overall performance and reducing manufacturing expense. The result is the availability of high performing FPGAs at relatively low cost [58].

In the past, the progression of performance capabilities and clock frequencies of general-purpose central processing units (CPUs) has scaled in accordance with Moore's law, where the advances in semiconductor technology allowed frequencies to double approximately every 18-24 months [6, 32, 58, 60]. In response, the demand for the development of ap-

plications has risen with the expectation of using these advances in CPU technology to perform complex computations and produce results faster than what was previously possible. A common objective of high-performance computing (HPC) is to complete large scale, computationally intensive, tasks within a short amount of time [6], which is achieved by utilizing pipelined code (*i.e.*, code having well-defined execution stages) that is distributed across multiple processing units executing in parallel.

Initially, HPC systems consisted of single-core CPUs arranged as a cluster (*i.e.*, a network of workstations). As CPU technologies have continued to follow Moore's law, the scaling has resulted in complex circuits (increasing development costs) and as CPUs' clock frequency have risen, power dissipation has risen to levels where the heat generated by these CPUs can no longer be sufficiently dissipated [6, 58]. Given the restriction in CPU technology, a solution to increasing computation capacity is the adoption of multi-core CPU architectures, where CPUs are designed with two or more processing units (*i.e.*, cores) for all processing duties. Clearly, multi-core CPUs have the potential to out perform single-core CPUs. However, without adopting a parallel programming design model that supports multi-core, multi-processor systems these systems do not perform as anticipated [58]. Furthermore multi-core CPUs are also susceptible to the same scaling limitations as single-core CPUs (*i.e.*, excessive power consumption and heat generation).

An alternative solution is a CPU-based system augmented with an application-specific, hardware-based, co-processor. In such a system, computationally intensive tasks are offloaded from the CPU to a special-purpose co-processor relieving the CPU to execute other application tasks (*e.g.*, I/O) until the co-processor has completed the required computation

and delivered the result back to the CPU. When accelerators such as GPUs and FPGAs with architectures capable of both pipelining and parallelism are used as co-processors, have the potential to advance HPC beyond what traditional systems are capable of achieving [6, 58].

Modern FPGAs are manufactured to contain multi-millions of transistors resulting in very large densities of logic cells. Logic cells (or just logic) are the FPGAs' building blocks for implementing customized processing elements in hardware. Each logic cell is independent of other logic cells and is capable of executing concurrently with other cells. From 2000 to 2010, architectural enhancements improving FPGA clock frequencies (average improvement of 25% each generation) and the increased logic densities per FPGA generation have contributed to an approximate 92-fold increase in logic compute performance of the FPGA [58].

The increasing logic densities provide a medium for the realization of custom processing elements with deep pipelines to exploit fine-grained parallelism. The large number of resources available in modern FPGAs enable the construction of multiple parallel pipelines that can be arranged to execute concurrently to exploit course-grained parallelism. The improved logic densities enable the realization of hundreds of complex pipelined and parallelized hardware functions capable of producing results with high throughput [6, 58, 60], an FPGA feature highly attractive in HPC environments. Furthermore, while CPU frequency scaling has peaked, FPGA operating frequencies continue to improve [60]. Therefore, there is potential for the continued increase in FPGA performance for HPC applications.

1.1 Motivation for this Research

The FPGAs' logic enables custom hardware where end-users can design special purpose hardware to meet the processing needs of HPC applications. Given the ability to support pipelining and parallelism, FPGAs are used to realize multiple pipelined processing elements (or modules) in parallel, increasing throughput as a result. In HPC applications, parallel modules go idle on occasion. Field programmable gate arrays can be reconfigured to replace idle modules with other modules needed for the current stage of execution. This on-the-fly reconfiguration enables FPGAs to perform complete hardware logic changes in circuit. This is the advantage FPGAs have over fixed logic hardware chips, such as ASICs, whose logic is permanently etched into the chip. The FPGA performance is not as good as fixed hardware, but the ability to reconfigure and especially to dynamically reconfigure the FPGA makes it a viable alternative.

Additional features of FPGAs include built-in multi-gigabit transceivers (MGT), built-in storage memory, and embedded processors. The FPGAs' relatively low operating frequencies (speeds in the MHz range) result in low power utilization and also low heat generation. The built-in MGTs enable high-bandwidth communication channels, peaking at speeds of 28Gbps [58], useful for clusters and modules performing I/O intensive computations. Given the speeds of the MGTs, inter-FPGA communication is limited by the production rate of the realized hardware and the communication rate of connected hardware components (*e.g.*, hard disks, expansion slot buses) and not by the MGTs. For memory intensive computations, modules can utilize the built-in memory storage. Field programmable gate arrays can instantiate their internal memory banks for storage needs

to avoid using the random-access memory (RAM) that is shared with the host CPU and peripheral devices in the host system. Lastly, modern FPGAs contain embedded general-purpose processor cores or can be configured to realize logic-based processor cores. The embedded cores enables FPGAs to run software and perform general-purpose processing operations as a complete autonomous system-on-chip (SoC) [16, 50]. Additionally, an embedded processor core can initiate on-the-fly reconfiguration to swap in additional cores (or special purpose hardware modules) to meet the processing needs of the HPC application.

The on-the-fly reconfiguration capabilities of FPGAs are convenient for swapping their function (*e.g.*, from AES hashing to MD5 hashing) by replacing one configuration with another. However, the limiting factor is that the reconfiguration process requires the FPGA to stop all logic execution until reconfiguration is complete. When the FPGA halts, all applications depending on the FPGA to perform a computation must also halt. The resulting delay is an overhead that is unacceptable in HPC environments. An alternative is to load all computation modules as independent hardware modules onto the FPGA simultaneously. However, this may necessitate using larger FPGAs to support all modules simultaneously or require distributing these modules across multiple FPGAs. Both options are feasible but can be expensive in terms of procurement and maintenance costs in addition to being expensive in terms of space, power, weight, and cooling requirements.

With the advent of partial reconfiguration (PR) capable FPGAs, it is possible to reconfigure parts of the FPGA while the remaining parts of the FPGA retain their current configuration and continue execution. Partial reconfiguration provides for fine grained reconfiguration of the HPC applications where independent hardware processing elements can be

identified and swapped in and out on demand [72]. Partial reconfiguration is achieved by providing a library of interchangeable hardware components, called partial reconfiguration modules (PRMs), to be loaded by the application onto a specified region of FPGA logic resources called the partial reconfiguration region (PRR).

Regardless of the benefits of using PR, wide ranged adoption depends on the availability of a general design flow that could be applied to any application and FPGA. Field programmable gate array vendor Xilinx provides a PR design and tool flow that outlines rules and steps for designing applications that use PR for Xilinx FPGAs [43, 72, 73]. However, the flow is basic and teaches one specific hardware design: the design of an application that changes its function by swapping only one module for another.

The Xilinx design flow provides little instruction for achievable PR designs outside of the specified design flow structure nor does it provide a model for the analysis and design of PR applications optimally tuned for any given FPGA. In other words, the Xilinx PR design approach is "ad hoc", where the discovery of efficient modularization that supports an arbitrary hardware design in general is left to the logic designer. Factors such as module layout, resource utilization, clock speed requirements, tool configuration settings, and other design factors are left for the designer to learn on-the-fly via trial and error. There is no model that provides a means for the designer to analyze the cost relative to space (*i.e.*, resources needed) and the execution time (clock cycles) required by an application using PR to complete its task without at first, implementing the PR application and then testing it.

This research extends Xilinx's basic PR design flow by introducing design techniques for two advanced PR design scenarios that incorporate the use of multiple PRMs. This dissertation presents a PRM design architecture enabling multiple independent PRMs to be mapped as one unit to a single PRR, a PRM design option not provided by Xilinx. The dissertation presents an advanced PR application design that will swap idle modules with additional active modules to increase computation throughout. Additionally, using Xilinx PR design methods, a PR application was implemented and tested to determine the usefulness of PR for a given application. This dissertation provides a generalized technique for estimating configuration bitstream (*i.e.*, a binary representation of the hardware application) size and PRM configuration time allowing designers to perform a cost analysis of the application before implementing a complete PR design.

1.2 Contribution of this Dissertation

A framework for designing and analyzing PR applications on FPGAs is presented in this dissertation. In typical non-PR implementations, FPGA application designers are concerned about space (*i.e.*, the on-chip resources required by the application), operating frequency, and the correctness of their designs. In PR applications, designers additionally consider how the application will be split between fixed and PR modules (PRMs), how resources will be shared between the various PRMs, what communication paths exist between all the application modules, what storage medium (internal or external) to use for unloaded PRM configuration bitstreams, and if the space (on-chip resources) and time overheads make the PR implementation feasible/useful. Correctness and operating fre-

quency are determined by the application designer and place-and-route software, and are therefore beyond the scope of this dissertation.

In order to enable the designer to make good decisions about using PR in applications, the dissertation presents a mathematical PR model of PR computation that incorporates resource utilization (for bitstream size estimation) and time (clock cycles) required to complete the computations associated with PR modules. The model is incorporated into a PR application design framework that can be used for analyzing PR application design decisions.

The design framework includes formulae for estimating configuration bitstream sizes and PR configuration times (in terms of clock cycles). By estimating the configuration bitstream size based on some known parameters (*i.e.*, known/estimated set of required resources) of the proposed application, an approximate PR configuration time can be estimated. Furthermore with known configuration times, a cost analysis of the application execution time (data transference + processing + configuration overheads + other overheads) can be made prior to implementing the design.

The framework also presents a multi-function PRM design technique where multiple functional modules can be loaded as a single PRM module. The technique enables the PR application to perform multiple functions using a single PRM without using PR to swap PRMs for each function change. To support applications with limited FPGA resources, an advanced PR technique is presented that temporarily reassigns resources via PR to improve execution efficiency. Techniques used for the analysis, design, and implementation of PR

applications are developed and presented in this dissertation research using the presented model as the blueprint.

The hypothesis of this research effort is as follows: In a hardware application pipeline where data traversing two interconnected modules transitions from a "slow" processing module to a relatively "fast" processing module. The PR model presented in this dissertation is a mathematical formula for determining the value of performing PR upon the fast module to improve the production of the slow module.

1.3 Background

The discussion below presents a brief history of reconfigurable computing, FPGAs, and the FPGA's architecture and resources. Field programmable gate array-based hardware applications and PR are discussed in Sections 1.3.2 and 1.3.3.

The concept of reconfigurable computing was first introduced in 1960 by Dr. Gerald Estrin with his Fixed-plus-Variable (F+V) Structure Computer [25]. The F+V structure paired a fixed general purpose computer (*e.g.*, microprocessor) with a variable structure that allowed the user to manually change the function of the system on the fly. Currently, reconfigurable computing is realized in the FPGA.

The FPGA is essentially a programmable digital logic chip that can be configured to implement applications in hardware. Similar to most static hardware chips such as the ASIC, which is custom designed to implement a specific function, the FPGA can be configured to implement a specific task (or hardware module) and then used as a component of a large scale hardware application; the FPGA can also be configured to implement the

application itself. However, as opposed to ASICs, if the hardware application had the need to adapt to a change in its environment, the FPGA can be reconfigured with a new design to support this change.

There are two types of FPGAs, non-volatile (*e.g.*, flash-based) and volatile (*e.g.*, SRAM-based). The flash-based FPGAs can maintain the state of the FPGA when powered off and are designed to operate at low power (*e.g.*, 1.2Volts to 1.5Volts) for hardware devices that have minimal power requirements [44]. The SRAM-based FPGA, such as the Xilinx Virtex family [74] and Altera Stratix family [8], is the most commonly used FPGA and is frequently the target FPGA in HPC/reconfigurable computing research projects that attempt to find hardware (HW)-based solutions for a range of problems [10, 20, 24, 46].

1.3.1 The FPGA Functional Blocks

The target FPGA in this research is a Xilinx Virtex family FPGA [65]. The internal fabric of the Virtex FPGAs is a collection of available resources (*i.e.*, "building blocks") that can be used to realize a variety of specialized functions [10], full applications, and complete autonomous systems. The fabric is an infrastructure of functional blocks capable of either data communication, data computation, or data storage. These blocks include configurable logic blocks (CLB), block RAM (BRAM), digital signal processors (DSP), digital clock managers (DCM), multi-gigabit transceivers (MGTs), general purpose I/O buffers, Ethernet blocks, signal routing resources, and in some Virtex FPGAs, dedicated PCI Express (PCIe) controller blocks.

The CLB is a logic function block that includes the look-up-table, the main logic function generator of the FPGA. The BRAM is a dedicated on chip memory storage block and the DSP is a dedicated arithmetic and logic function block. The CLB, BRAM, and DSP are the typical functional blocks used for PR and are discussed in further detail in Chapter 3.

The DCM is a clock management block that provides a means for designers to configure the clock driving their hardware application. The DCM receives an input clock signal, typically the FPGA's global clock, and presents as output a modified clock relative to the configurations specified by the designer. Digital clock manager features include frequency synthesis and phase shifting. With frequency synthesis, the DCM provides an output frequency that is either double the input clock frequency, or a fraction of the input clock frequency, or a derived clock frequency that is the result of a simultaneous frequency multiplication and division of the input clock [65]; the multiplier and divisor values applied are provided by the designer. The DCM supports fine grained and course grain phase shifting, such as, a 90, 180, and 270 degree phase shift of the input clock. For fine grained shifts, the DCM can be configured to dynamically shift the clock frequency forward or backward by 1/256 of the input clock period [65].

Multi-gigabit transceivers are used for data communication needs. The MGT, GTP or GTX in Virtex-5 and later FPGAs is a high throughput transceiver capable of transmitting a stream of bits at a peak rate of 6.5Gbps. The MGT supports both analog and digital signaling and is capable of delivering 8b/10b encoded transmissions. The MGT can be configured to support various communication protocols including SATA, Infiniband, Gigabit Ethernet, and PCIe.

1.3.2 FPGA-Based Hardware Applications

Historically, hardware designs that target FPGAs include a wide range of applications including circuit interconnect logic ("glue logic"), data communication (*e.g.*, switched networks), cyber security, applications for HPC areas such as Bioinformatics, and HPC co-processors. Field programmable gate array-based applications are commonly implemented for the purpose of gaining an advantage in computational speedup. There are research efforts that use the FPGA to accelerate existing software solutions by implementing the solution (or a fraction of the solution) in hardware [14, 20, 45]. High performance FPGA-based applications are typically implemented as interacting parallel modules consisting of high-throughput pipelines [47]. In general, an FPGA-based hardware application is a modular design that includes several parallelized, pipelined modules that are a classified as one of following categories:

1. Processing operations
2. Input and output (I/O) operations

Processing modules are the main data processors of the application. They are responsible for implementing an algorithmic procedure on input data elements and returning the computed results. Processing modules are typically designed as a set of pipelined sub-modules operating in parallel and controlled by a state machine. The I/O modules are responsible for the transference of data into and out of the FPGA. These modules are commonly designed to interface with external memory devices and the MGTs are the components used for data communication. An example FPGA application design with both processing and I/O modules is found in Dandass *et al.* [20].

When implementing an application for FPGA platforms, the set of all hardware-hosted modules is mapped to the resources on the FPGA and converted into a configuration bitstream. The configuration bitstream is the binary representation of a hardware application that includes everything that is required for the hardware application to be realized such as, clock management settings, dedicated hardware (*e.g.*, BRAM, DSP, MGT) configurations, logic functions (*i.e.*, functions/modules represented by look-up-tables), wires that interconnect these components and logic functions to make complete circuits, and the physical placement of these circuits onto the FPGA fabric. Configuration bitstreams generated for dynamically reconfigured FPGA resources (*e.g.*, PRMs) are generated as partial bitstreams. Partial bitstreams are not entire FPGA configurations; they are smaller bitstreams that configure a subset of FPGA resources and are loaded during run time using PR techniques discussed below.

1.3.3 Partial Reconfiguration

Figure 1.1 is a typical non-PR design of an FPGA-based hardware application designed to dynamically modify the application functionality. Processing elements constant to the application (labeled "Fixed") are implemented as modules directly connected to I/O modules (shown) or as a pipeline of interconnected modules. The dynamic modules of the application are labeled Processing Modules 1-3. These modules can be any combination of processing units for increased computation throughput and/or independent modules that implement a specific function (*e.g.*, different network protocols). Note that all these mod-

ules are present even when only one is being actively used. The multiplexer (MUX) is a common technique for connecting multiple data sources to one destination.

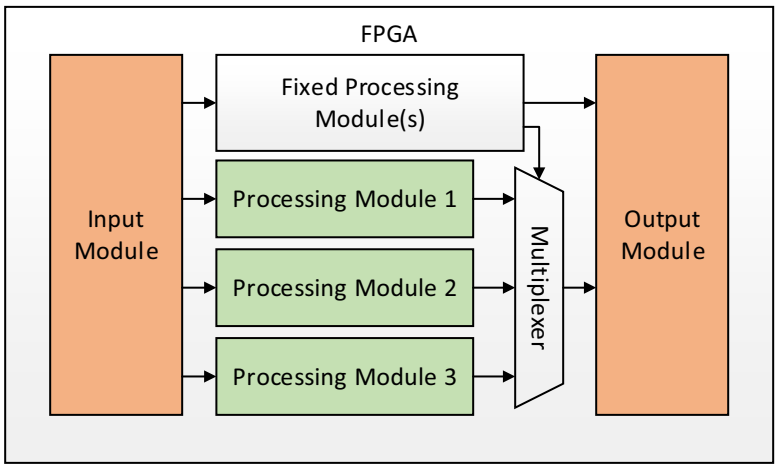

Figure 1.1

FPGA-based hardware application example

The disadvantage of a non-PR design is that all function modules used by the application must be resident simultaneously in logic. Having all modules resident in logic costs a resource utilization overhead, which is the increase in resources required to support a non-PR design versus a PR design. In addition, instantiating all modules increases timing overheads due to the creation of more complex circuits connecting them and increases the application's power needs to support the resident modules. If on-the-fly reconfiguration is used, modules not required in all stages of the application execution will not to be resident in the FPGA and can be swapped in, replacing other non-critical modules at the cost of halting the HPC application to reconfigure the entire FPGA.

The number of modules that can be added to an FPGA-based application is limited by the set of FPGA resources that are available for implementing logic. When a non-PR application's resource demands exceed the target FPGA's available resources, the application must be redesigned to use fewer resources at the cost of increased computation times and/or a loss in some of the functional capabilities of the original design. Alternatively, the application can be partitioned across multiple FPGAs. However, inter-FPGA communication links can increase communication overheads. In addition, the HPC system must be capable of supporting multiple FPGAs, which can be an expensive system modification or may even be architecturally infeasible. Mapping the application to a larger FPGA scatters the placement of modules and communication routing (the placement of wires and buses between communicating components), which results in longer communication paths that decrease the application clock frequency. Furthermore, the monetary cost of FPGA devices is traditionally dictated by the amount of resources the FPGA provides for implementing logic. The greater the number of resources, the more expensive the FPGA.

An alternative PR-based approach is to design an application that replaces inactive modules with other modules allowing the use of the idle resources to construct additional processing elements to increase throughput (by reclaiming resources from idle non-critical modules) and to decrease wasteful power and resource consumption (by eliminating inactive modules) without interrupting the execution of the application. A non-PR implementation requires more FPGA resources (with a concomitant increase in cost) as compared to a PR implementation. In contrast, a PR implementation requires additional development cost as compared with a non-PR implementation.

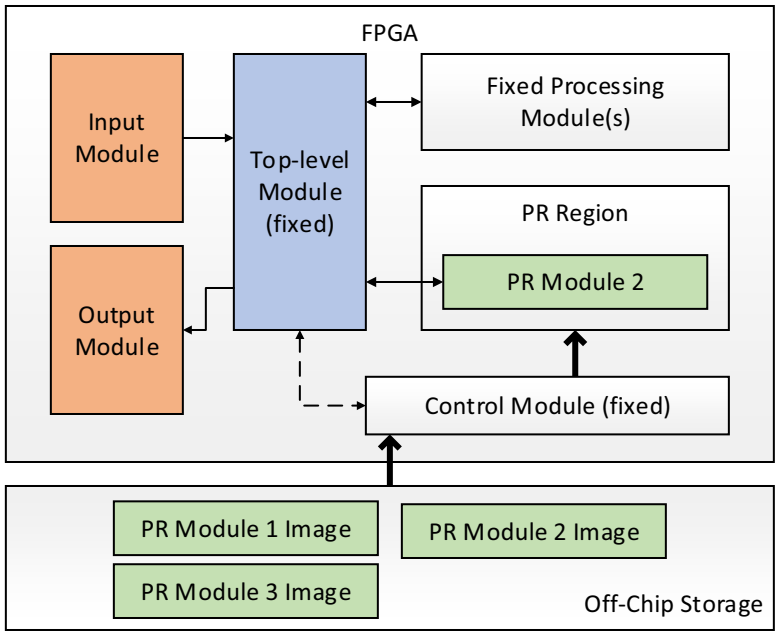

Figure 1.2

Basic PR design example

Figure 1.2 is a design perspective of a hardware application employing a general form of partial reconfiguration. The application architecture includes fixed processing modules, a fixed control module for loading PRMs, a fixed top-level module, I/O modules, and a single PRR for loading a PRM. The Xilinx PR design flow requires all PR implementations to contain a top-level module. The top-level module combines the PRMs, fixed modules, fixed I/O modules, and control module into a complete PR application. The primary purpose of this module is to establish data communication channels between the modules of the application. The PRR reserves resources for PRMs loaded via PR. The position in the FPGA fabric and dimensions (height and width) of the region is set by the designer. A

PRM is either implemented as a single hardware module or a collection of interconnected modules that make up a complete circuit.

During PR, configuration bitstreams of PRMs are loaded from either off-chip storage (typical to conserve logic) or on-chip storage populated before or during FPGA initialization (*i.e.*, loading the initial application configuration onto the FPGA). The PR application can be designed to execute with or without PRMs dictated by the application's control logic through the control module. The control module's purpose is to invoke PR according to application parameters set by the designer. It monitors the system, loads or unloads PRMs if necessary, and enables access to PRMs when they are available for use. The control module includes logic that interfaces with the internal configuration port of the FPGA and establishes the tie between the configuration port, the on/off-chip storage, and the PRR that receives a loaded PRM. The dashed connection to the top-level module in Figure 1.2 marks designs where the control module has logic that throttles communication channels between PRMs and fixed modules. The fixed processing modules complete the application. Fixed modules are logic that is unmodified during PR. Fixed (or static) modules implement unchanging modules essential to the application (in function or utility) and functions that cannot be partitioned into PRMs.

The Xilinx general PR design focuses on using PR to change application functionality. This research focuses on advanced PR design techniques that leverage the Xilinx design flow to realize hardware applications that use PR to improve performance.

1.4 Research Summary

This research has developed a mathematical PR model for the development and use of partial reconfiguration in applications. This research has also extended the Xilinx PR design flow by introducing techniques for the development of advanced PR designs as described in Chapter 4.

The expected result of this research is a PR model that aids the designer in making informed decisions on incorporating PR into a hardware application, its feasibility relative to application performance and consumed resources.

1.5 Organization of Dissertation

Chapter 2 is a survey of related research on partial reconfiguration techniques. In Chapter 3 (p.52) details on the FPGA architecture and the basic partial reconfiguration design flow is provided. This chapter is for readers that are unfamiliar with the FPGA and/or partial reconfiguration and wish to read more about them. The dissertation PR framework is discussed in Chapter 4 (p.74) and the design for experimentation is discussed in Chapter 5 (p.102). Chapter 6 (p.144) discusses the hardware configurations used in the experiments and timing results of the hardware execution. Chapter 7 (p.156) discusses hardware design decisions made during this research and presents the mathematical PR model (Chapter 7.4) mentioned in hypothesis. Lastly, the dissertation concludes with Chapter 8 (p.175).

CHAPTER 2

LITERATURE REVIEW

This chapter introduces various techniques and models used for implementing FPGA-based hardware applications capable of supporting partial reconfiguration. This chapter provides background information on the Xilinx partial reconfiguration design flow. The chapter will also discusses some proposed configuration prefetch, compression, and caching approaches used to minimize reconfiguration overheads. This chapter also summarizes published works that apply the partial reconfiguration design flow in the implementation of several hardware applications.

2.1 The Partial Reconfiguration Design Flow

Current Altera Stratix family and Xilinx Virtex family of FPGAs support the ability to be reconfigured during execution in a process called *dynamic reconfiguration* [8, 49, 74]. At any point after being powered up, the FPGA can be invoked to stop all operations and load a new configuration, and then begin the execution of that new configuration [11, 42, 49]. The usefulness of dynamic reconfiguration is adaptability, where, in an ever-changing environment, the FPGA can change its configuration to meet the requirements of that environment. The term, dynamic reconfiguration, has been used to represent almost any type of on-the-fly reconfiguration during hardware application execution, including

tile-based [36] and partial reconfiguration [27, 39, 53, 66] that target and reconfigure only portions of the FPGA fabric. In this document, the term partial reconfiguration (PR) is used to refer to dynamically reconfiguring a user-defined subset of the overall FPGA fabric; the term dynamic reconfiguration will be used to refer to uploading a full hardware configuration bitstream during execution to reconfigure the FPGA entirely. Below is a brief introduction to the PR design flow. The design flow is discussed in further detail in Chapter 3.

Partial reconfiguration is the support for modules in an FPGA design to be dynamically reconfigured by uploading a configuration bitstream file. The Xilinx partial reconfiguration design flow is summarized in the following steps [72]:

1. Hardware description language (HDL) design and bottom-up synthesis
2. Floor planning of PRRs and timing constraints
3. Generate FPGA implementation files (and configuration bitstreams) of all modules in the design and merge static modules and PRMs into a complete application

The FPGA manufacturer Altera has its own PR design flow for its Stratix family of FPGAs [4, 5] but the Xilinx Virtex FPGA is most commonly used in research literature involving PR and the Virtex FPGA was also chosen for the research documented in this dissertation.

Step 1, bottom-up synthesis, is essentially a syntax check and optimization of all modules in the HDL-based hierarchical design beginning with the synthesis of module dependencies, and then the synthesis of upper-level modules stopping at the top-most module in the application. This phase generates files that contain details needed in the next two steps. In step 2, the floor planning step, the designer manually specifies static and PR region geometries (*i.e.*, size and shape) and physical placement on the FPGA fabric. A static region

is where fixed modules may be placed. A hardware application can have several static and PR regions. Step 3 results in a complete PR application that can be deployed to the FPGA.

2.1.1 Two Design Flows for Partial Reconfiguration

There are two design flows for partial reconfiguration; *module-based* PR and *difference-based* PR [26, 62, 72, 79]. Wu and Madsen outline the benefits and pitfalls of both module-based and difference-based PR design flows based on their experience with PR in "Run-time Dynamic Reconfiguration: a Reality Check Based on FPGA Architectures From Xilinx" [62]. Zeineddini and Gaj also discuss some differences between the design flows in "Secure Partial Reconfiguration of FPGAs" [79]. Xilinx outlines the modular-based design flow in "Partial Reconfiguration User Guide" [72], including the steps highlighted above. Xilinx also discusses the difference-based PR design flow in "Difference-Based Partial Reconfiguration" [26]. The following sections are some highlights of the module-based and difference-based design flows.

2.1.1.1 Module-based Partial Reconfiguration

Module-based design is the design flow recommended by Xilinx for partial reconfiguration [72, 79]. In module-based PR flow, FPGA resources are partitioned into static regions for unchanging hardware modules and PRRs that are designated for deployed PRMs. During implementation of a PRM and in a phase called place-and-route, PRM components are placed on the FPGA fabric and wires that connect the components are routed to make complete circuits [69]. The PRM place-and-route process is confined to the resources reserved by the PRR (*i.e.*, the PR module is fitted to the PR region) [62, 72].

Wire connections between PRR and static regions travel through partition pins. Partition pins are specialized hardware components that are necessary to guarantee complete circuits between regions and are placed at region borders by the available Xilinx tools [72]. Partition pins replace the bus macro, a multi-state 4-bit buffer used as a fixed data path that allows intermodule communication between regions [43, 62, 63, 72].

Bus macros are implemented logic devices that must be instantiated and placed across region borders. Bus macro placement is done manually via user-defined constraints [63]. The 4-bit data width requires several modules to be instantiated and placed to accommodate larger widths. Bus macros are required for PR designs that target Xilinx's Virtex-II and earlier model FPGAs [72]. The maturation of the Xilinx partial reconfiguration tools and the capabilities of the latest Virtex family FPGAs allow for the replacement of bus macros with partition pins [72]. One of the pitfalls of the module-based design flow is that the procedure outlined in this flow is repetitive and scales with the number of PRMs in the design. However, this design approach is commonly used since it does not require the designer to know the low level details of the target FPGA [62, 79].

2.1.1.2 Difference-based Partial Reconfiguration

Difference-based design flow is used for making low level changes to a design such as logic equations (*i.e.*, LUT contents), I/O standards, and routing information by directly editing the synthesized netlist using tools such as the Xilinx *FPGA_Editor* or *PlanAhead* [26, 62, 79]. Block RAM contents can be changed using a different specialized tool [26]. The difference-based design flow does not require user-defined PR regions or region con-

necting logic and is considered to be more space efficient as a result [26]. However, the user must have some knowledge of the low level components of the FPGA resources.

The design flow begins with the netlist of a full design and some knowledge as to which components of the design are to be partially reconfigured [26]. Wu and Madsen's approach is to implement several full designs and do a frame-by-frame contrast to highlight and record the differences between configurations [62]; a frame is a 1-dimensional column of configurable logic blocks. These frame differences are used to modify the design and eventually generate partial bitstreams for on-the-fly PR.

Design modifications for difference-based PR are done manually. Different architectural configurations for PR can be generated by modifying LUT content, routing, I/O, and low level components that are provided in the design netlist—done by using netlist editing tools. All modifications in this flow are low level; there is no high level approach for difference-based PR using HDLs. Therefore, large changes such as an algorithm change are not recommended due to the large number of low level components that will require modification [26].

Difference-based partial reconfiguration is largely dependent on the configuration that is currently loaded on the FPGA. Since changes made to a configuration are specific to the low level hardware components, some partial bitstreams may be incompatible with different configurations of the same application. Therefore, for every n possible configurations of an application to be partially reconfigured by $n-1$ configurations, there must be $n(n-1)$ partial bitstreams generated to support all possible reconfigurations [62]. An alternative suggested by Wu and Madsen is to choose a neutral configuration where all

partial configurations are based. This way the number of required partial bitstreams for an application is reduced to $2n$ [62]. The trade-off, however, doubles the number of reconfigurations since, for each PR, the design must first revert back to the neutral configuration before the desired configuration change can be applied.

2.2 Reducing Reconfiguration Overheads

Both Altera and Xilinx provide a wide range of FPGAs of various densities (*i.e.*, the amount of available resources). Configuration bitstream sizes ranging from smallest FPGA to the largest FPGA of the same series may differ by an order of magnitude [18, 66]. For example, the Xilinx Virtex-5 LX series FPGA bitstream size range from a typical 8.3 Mbits for the smallest LX30 FPGA to a 79.7 Mbits for largest LX330 FPGA [66]. In a changing environment that requires the FPGA to adapt via dynamic reconfiguration, the speedups gained by FPGA hardware are limited by overheads due to the time it takes to perform reconfiguration [42, 49]. A larger bitstream also increases the memory resources needed for storage and is thus not optimal in environments where memory resources are limited [54]. Several researchers have developed techniques to reduce reconfiguration overheads and/or the memory footprint of the configuration bitstreams that includes configuration prefetch [11, 35, 55] and bitstream compression [18, 40, 42, 49, 54]. These techniques are outlined in the following sections.

2.2.1 Configuration Prefetch

An advantage of partial reconfiguration is that modules of an FPGA-based hardware application can be reconfigured without interrupting the execution of the application's other

modules. To exploit this advantage, designers employ a configuration prefetch technique to minimize any overheads associated with the reconfiguration stage of PR. Configuration prefetch is a technique used to amortize the configuration time cost of PR by parallelizing the configuration of PRMs with the execution of other modules in the application [11, 35, 55]. A down side to configuration prefetch is that the reconfiguration time can exceed the execution times of the remaining modules and, if not initiated at the right time, can result in idling. Therefore, the success and failure of configuration prefetch can be determined by the accuracy in predicting what time, during application execution, is the best time to initiate reconfiguration. There are several methods for configuration prefetch that attempt to find the earliest time to initiate a prefetch that results in minimal configuration overheads [11, 35, 55]. This section with will discuss a few approaches to configuration prefetch.

2.2.1.1 Prefetch Using Data Control Flow

Scott Hauck introduces the configuration prefetch technique in "Configuration Prefetch for Single Context Reconfiguration Copocessors" [35]. The target system architecture for this algorithm is a microprocessor with a reconfigurable hardware co-processor. The microprocessor executes software instructions and is responsible for executing instructions (*i.e.*, function calls) to initiate prefetch operations and instructions to invoke hardware functions on the co-processor. The co-processor is assumed to be a reconfigurable hardware device where portions (or even the entire device) can be dynamically reconfigured. The co-processor is also assumed to implement a single configuration, *i.e.*,, a single computation function. If a different computation is desired, the co-processor must be reconfigured

to implement this computation. Applications that target this system are sequential programs written in assembly language instructions. Throughout these programs are portions of the software selected (in a previous process) for hardware processing and therefore, replaced by instruction calls to different hardware computation functions implemented by the co-processor. These hardware instruction calls (RFUOPs) contain the function ID of the hardware configuration needed for the desired computation.

During program execution without prefetch, the microprocessor will execute instructions until a hardware instruction is reached [35]. At this point, if the current co-processor configuration is not the requested function, reconfiguration is initiated and the microprocessor shuts down program execution until reconfiguration is complete; otherwise the co-processor executes its function. Hauck's prefetch algorithm attempts to limit (or eliminate) shutdown lengths by inserting hardware prefetch instructions into the program code when necessary and at the earliest possible time.

The prefetch algorithm analyzes a control flow graph of the program executable and attempts to find the best possible point in the flow to insert hardware prefetch instructions; an example control flow graph is presented in [35]. Each node in the control flow graph represents an instruction in the program. The prefetch insertion algorithm considers only forward paths in the graph and edges that lead backward from a RFUOP to preceding instructions. The basis of the prefetch insertion algorithm is a directed shortest-path algorithm [35]. The belief is that, for any RFUOP in the graph, the RFUOP with the shortest path (*i.e.*, reached in the least number of clock cycles) from the current point of execution is the RFUOP that should be configured [35]. Instructions close (via shortest path) to

an RFUOP instruction in the graph are considered to be *owned* by the RFUOP. Owned instructions are candidate locations for prefetch instruction insertion, allowing the associated configuration to be loaded prior to executing the RFUOP instruction. Instruction ownership simplifies the prefetch insertion process by partitioning the control flow graph into *ownership regions*. Each region includes all the instructions owned by a single RFUOP. Once ownership regions of the program are established, Hauck's algorithm simply inserts hardware prefetch and region borders where the predecessor of an instruction in a region is an instruction that belongs to another ownership region.

2.2.1.2 Probability-based Prefetch

Sim *et al.* uses probabilities to determine the best possible time for the configuration prefetch of PR modules [55]. The target system is a CPU paired with an FPGA-based co-processor. The FPGA resources are organized into several slots of six configurable logic blocks to be used for PR. The CPU is responsible for controlling module execution including initiating PR by following a precompiled schedule. The application is a heterogeneous (*i.e.*, software and hardware-based) sequential program that uses the FPGA as a co-processor to implement hardware designated tasks. Like Hauck, Sim *et al.* represents the hardware as an interprocedural control flow graph [55]. Each node in the graph is either idle resources (*i.e.*, prefetch destinations in the FPGA fabric) or a hardware module (*i.e.*, non-idle resources). These nodes are one of three different types: general nodes, exit nodes, and call site nodes that lead to the execution of its descendant nodes.

Sim *et al.* generates a reconfiguration schedule in the following steps [55]:

1. Use profiling to determine how often each edge in the graph is executed (*i.e.*, traveled) and eliminate those edges that have not been executed. Assign weights to remaining edges, e, using the formula $w(e)$ where:

$$w(e) = \frac{total\ number\ of\ times\ the\ edge\ e\ is\ traveled}{total\ number\ of\ times\ the\ node,\ head(e),\ is\ traversed} \tag{2.1}$$

2. Compute the post-dominators for each node. A node, m, is *post dominate* of a node, n, if all paths to the exit node of a graph starting from n must go through m.

3. Compute the intra post-dominator path (IPDP) for each node, m, in the graph that have to following properties: a) starting from m, there are no other nodes in the path to an exit node that is a post-dominator of any other node along the path and b) the product of the weights for each edge in the path must be greater than a threshold value of 0.0005.

4. Using the IPDP and post-dominator information, compute the estimated *placement-aware probability* (PAP) of reaching each hardware node for every node in the graph.

5. Insert bitstream load instructions into FPGA functional blocks that have a non-zero PAP.

Step 4 is separated into two algorithms, $A1$ and $A2$. Algorithm $A1$ is an iterative process that calls different PAP estimation functions for general nodes and call site nodes; PAP values are not computed for exit nodes. In addition to estimating the PAP, these functions also check for hardware *placement conflicts*. A placement conflict occurs when the physical resource requirements of two or more modules overlap, preventing proper placement. If a placement conflict for a node is discovered, its PAP calculation is skipped. Placement-aware probability calculations for call site nodes are estimated by taking the maximum value between the sum of $w(e)$ values for all its called nodes and the PAP of its immediate post-dominator. Algorithm $A2$ takes a general node in the graph and estimates its PAP using its IPDP and the PAP that the node will reach a hardware module through all possible paths. In general, it produces a temporary vector of new PAPs and compares it to the current vector of PAPs, replacing the current with the new if there is a difference.

Modules selected for reconfiguration are determined by the CPU at run time using the reconfiguration schedule generated by the algorithms in step 4. For example, if the most probable (highest PAP) module has not been configured to the FPGA and is not currently being configured, then this module is selected for reconfiguration. Otherwise, if this module has already been loaded to the FPGA, the CPU selects the module with the next highest PAP if there are no placement conflicts and if no other module is currently being reconfigured.

2.2.1.3 Priority-based Prefetch

Banerjee *et al.* uses a placement-aware list-scheduling algorithm to determine the best possible time to initiate configuration prefetch of hardware modules by using a prioritization function to rank each potential module [11]. Like Sim *et al.*, to create an *execution schedule* of an application, Banerjee generates a dependency task graph to represent the order of execution and clearly illustrate any data dependencies between tasks (*i.e.*, nodes) in the application. A node is a dependent if it must wait on the results produced by its ancestor nodes to begin execution.

The target system architecture is in a simulated environment that pairs a Xilinx Virtex-II FPGA with an external (*i.e.*, not embedded) IBM PowerPC microprocessor. The application is a heterogeneous application where tasks are partitioned to be implemented in software on the PowerPC or in hardware; Sim *et al.* focuses primarily on the assignment of task (*i.e.*, PR modules) to hardware using configuration prefetch. The PR design flow for a Virtex-II FPGA suggested by Xilinx is to place hardware modules at column boundaries

where a column is a vertically aligned set of configurable logic blocks; a placed PRM must consume the entire length of the columns it spans. Therefore, the researchers' prioritization function of the list-scheduling algorithm is fully aware of the physical layout of the FPGA including the current placement of previously configured modules. The algorithm is also fully aware of the application's task execution schedule.

The list-scheduling algorithm proposed by Banerjee is a simple selection algorithm that first ranks each PRM in the dependency task graph using a prioritization function Equation 2.2. The algorithm then selects for reconfiguration the module that has the highest priority value. The reconfiguration process is then scheduled to begin at the earliest possible time. The priority value of a PRM is a function of the module columns, earliest start time (EST) for computation, earliest finish time (EFT) of computation, and the module critical path length:

$$-A * moduleColumns - B * EST + C * criticalPath - D * EFT \qquad (2.2)$$

The moduleColumns parameter is the total columns the PRM will consume. The EST parameter is computed by a heuristic method described below. Parameters criticalPath and EFT can be obtained by observing the physical and execution characteristics of the PRM. Constants *A*, *B*, *C*, and *D* are weight values. The negative weights (*A*, *B*, *D*) lower priority values of a module with relatively high values of the associated parameters; smaller values of these parameters (*i.e.*, fewer columns and earlier start and stop times) are preferred.

The earliest computation start time heuristic takes into account several physical properties such as available resource, reconfiguration time, and the availability of the FPGA reconfiguration controller. The heuristic is as follows [11]:

start heuristic

Using the application execution schedule, search for the earliest time to start PR;
// the earliest time in the schedule is where the FPGA resources are available for PR

IF(*the reconfiguration controller is available*)
SET *PR_start_time* to the earliest time instance;
// *PR_start_time* is a time slot in the execution schedule where PR can began

IF((*PR_start_time* + *reconfiguration_time*) < *dependency_time*)
SET EST to the time where the module's dependency values are available;
// the cost for partial reconfiguration is amortized.
ELSE
SET EST to (*PR_start_time* + *reconfiguration_time*);
// the cost for partial reconfiguration is not completely hidden

end heuristic

The resulting EST calculation is the best possible time, during execution, to initiate the configuration of the associated PRM. The PRM's associated priority will determine the point in the execution order where it will be selected for hardware implementation.

2.2.2 Bitstream Compression

Bitstream compression is a technique used to condense the bitstream in an attempt to reduce reconfiguration overheads and memory requirements. The idea of bitstream compression comes from analyzing the structure of the configuration bitstream and determining which bits actually contribute to configuring the FPGA resources. Researches have discovered that given the entire configuration bitstream and knowledge of the bitstream structure,

only a subset of the total bits actually have unique information used to configure the FPGA resources. Another set of bits include several identical (and smaller) subsets that are used to configure the smallest configurable component of the FPGA. The remaining bits contain no configuration data and are used as padding. Bitstream compression attempts to minimize redundant and duplicate subset to decrease reconfiguration time and memory storage footprint.

2.2.2.1 Dictionary-Based Compression

Researchers Li and Hauck [42, 49] and, in a separate effort, Dandalis and Prasanna [18, 19, 49] use a dictionary technique with readback based on the Lempel-Ziv-Storer-Szymanski (LZSS) [57] and Lempel-Ziv-Welch (LZW) compression algorithms. The compression technique developed by Li and Hauck uses LZSS and is specific to Xilinx Virtex FPGAs. A more general technique developed by Dandalis and Prasanna uses LZW and applies to all SRAM based FPGAs [19]. However, to support this generalization, Dandalis' approach processes raw data and does not consider the individual semantics of the configuration bitstream [19, 33, 42]. As a trade-off, the compressed bitstreams derived by Dandalis and Prasanna are not efficient in compression ratio relative to FPGA series specific compression techniques [33, 42, 49]; it has been stated that Li's technique is superior in this respect [49].

The FPGA fabric layout is an array (or matrix) of resources with a fixed number of like resources placed in a column called a frame. This frame is the smallest reconfigurable block of a Xilinx FPGA; Altera has an equivalent construct but named differently. A

frame extends the height of a row in the fabric and each row consists of several frames positioned side-by-side. Li states that because of the similarity of resources in frames contained within the array, developers can expect some regularity between frames [42]; *i.e.*, frames that consist of the same resource are likely to have similar, if not identical, configurations. This similarity is called *inter-frame* regularity [33, 42, 49]. A second regularity, *intra-frame* regularity, categorizes similarities of resources within the frame, (*i.e.*, the configurations of the individual resources within the frame may also be similar or identical to each other). Li's compression technique attempts to leverage these regularities by storing similar bit sequences and then replacing all occurrences of the sequences with a reference to the location of the stored sequences using a dictionary technique of the LZSS algorithm. In general, when parsing an input string for compression, Lempel-Ziv-based algorithms store sequences (preferably long) of strings in a dictionary where each string in the dictionary is referenced by a unique index (or encoding) [19, 42]. As the input string is being processed, each sequence in the input string that exists in the dictionary, is replaced by the index, otherwise, the new sequence is added to the dictionary.

During configuration, the configuration bits of the FPGA are loaded into the Frame Data Register, Input register (FDRI). This register is a shift register where configuration bits are loaded prior to being transferred to the configuration memories of the FPGA resources. The LZSS algorithm used by Li is an extension of the Lempel-Ziv 77 (LZ77) [80] algorithm that is designed to improve the coding efficiency of the LZ77 [42]. The LZ77 is a sliding window compression algorithm that tracks the last n symbols in a string sequence where n is the size of the sliding window. In the case of an FPGA configuration

bitstream, a symbol is the smallest sequence of bits that can be placed in the dictionary and the FDRI is used as the sliding window for detecting similar symbol sequences. A symbol size of either 6 or 9-bits was chosen to support other compression algorithms (*e.g.*, Huffman), preserve potential regularities and also limit any additional overheads that may arise with larger symbol sizes [42].

Li's compression technique employs several algorithms including the LZSS algorithm and a directed minimum spanning tree algorithm. This technique also introduces a frame readback algorithm and methods for rearranging the order in which frames are configured to allow for like frames to be configured consecutively. The frame readback and frame configuration order rearranging aid in compression by allowing the FPGA to temporarily store the configuration bits of a frame and then recall and reuse the same sequence to configure the next consecutive frame with a similar configuration. This frame reuse allows for like sequences (*i.e.*, symbols) to be removed from the original bitstream. The overall procedure is somewhat summarized in Li's readback algorithm [42]:

1. Preprocessing: break the bitstream up into a stream of symbols
2. For each frame in the stream of symbols use it as a fixed dictionary and perform LZSS upon each other similar frame
3. Build an inter-frame regularity graph from the result of LZSS
4. Find the minimum spanning tree of the inter-frame regularity graph to create a configuration sequence graph
5. Execute a preorder traversal of the tree and for each node traversed, do:
    (a) If the node has multiple children, copy the node into an empty BRAM slot
    (b) If the node's parent is not in the FDRI read the parent from BRAM back into the FDRI
    (c) Perform LZSS compression

(d) If the node is the final child of the parent node, release the BRAM slot taken by the parent

To find both inter-frame and intra-frame regularities, the FDRI was increased from 32-bits to the size of two frames by implementing additional hardware on the FPGA to realize this extension. The two frame size FDRI allows frame-size similarities to be detected and encoded. Detecting intra-frame regularities came naturally to LZSS since the regularities can be found by just comparing individual symbols. However, for inter-frame regularities, Li found it necessary to rank each frame by their similarities, which is computed by comparing the set of symbols enclosed by one frame, say frame A, to the set of symbols enclosed by another frame, say frame B. The ranking is a numerical weight representing the differences between the symbols in the *reference frame* [49], *e.g.*, frame A, when compared to the *beneficiary frame*, *e.g.*, frame B, in that direction. In the set of all similar frames, each frame is rated as if it was a reference frame and all other frames are beneficiary frames. After ranking each frame, the end result is a weighted directed graph of configuration frames (nodes) and the similarities between other frames (the weighted directed edges) where the larger the weight of the edge represents greater differences between related frames [42]. The minimum spanning tree algorithm removes all reference frame to beneficiary frame edges that are not optimal. It is clear that all reference frames are the parent nodes in the resulting tree and leaf nodes are beneficiary frames. During configuration, reference frames are configured to the FPGA but they are also used as a fixed dictionary where similar symbols in the dictionary are reused to configure the beneficiary frame. The preorder traversal in the algorithm implements the reuse of frame configuration

bits and explains the readback procedure. All parent nodes (*i.e.*, reference frames) contain symbols that can be used to configure the resources associated with a child frame (*i.e.*, beneficiary frame). Therefore, the parent frame is stored in memory until it is time to configure a child. When a child frame is configured, the parent frame is loaded into a frame slot of the FDRI and used as a dictionary to configure the resources of all of its children. Implied in this algorithm is a sequential ordering of like frames. Li reorganizes similar frames to be configured consecutively to eliminate more redundant symbols and achieve a greater compression of the original bitstream [42].

In the Dandalis and Prasanna approach [19] the dictionary is constructed off chip using a modified version of the LZW algorithm; the modification is a change in how the compression ratio used by the algorithm is calculated. The entire dictionary, including the index (*i.e.*, the set of encoded strings), is stored in memory. To reduce the memory requirements of the dictionary, Dandalis removes unused symbols and merges a few common prefix strings. Bitstream decompression is executed by the FPGA and therefore Dandalis added the hardware components that are needed to complete this process; the added hardware components had a minimal impact on the amount of available resources. At power-up, the decompression of the bitstream begins with reconstructing the dictionary by parsing it with respect to the index. The decompressed bitstream is fed to the internal configuration interface of the FPGA.

For partial bitstream compression, Gu and Chen [33] used techniques from both Li and Dandalis to implement a compression technique for partial bitstreams. Gu and Chen used the LZW algorithm to build the dictionary, attempted to exploit the inter-frame and

intra-frame regularity, and employed dictionary reuse. The main contribution by Gu and Chen approach is the consideration of the *inter-bitstream regularity* in partial reconfigurable designs. Inter-bitstream regularity is identified as the symbol similarities between the full configuration bitstream that must be uploaded to the FPGA at initial configuration, and the partial bitstreams that are loaded into the FPGA at any time thereafter. Since the full configuration bitstream is a complete hardware design, Gu and Chen state that a partial bitstream will likely have some remnant symbols of the full configuration bitstream [33]. In other words, some portions of the design are persistent in all "modes" of the application (*e.g.*, static modules) [49] that can be reused to aid in configuring the PRM; the term "mode" refers to the change in the application that occurs due to a (or several) PRM swap(s). Gu and Chen extend the LZW algorithm to also detect inter-bitstream regularity and therefore place the related symbols (or set of symbols) into the dictionary allowing redundant symbols in the partial bitstreams to be removed.

2.2.2.2 Compression Via Frame Removal and PR

In current FPGA boards, it is common for the SRAM-based FPGAs to be paired with non-volatile flash memory and, in some cases, DDR RAM slots to be used as external storage. The flash memory is typically loaded with an initial configuration for the FPGA to be used at power up. One side effect of the dictionary-based bitstream compression discussed in Section 2.2.2.1 is the need to store the dictionary and index in non-volatile memory so that the bitstream can be decompressed and used to configure the FPGA at start up. As the size of bitstreams increase due to larger FPGAs and their available resources,

the size of the compressed bitstreams will also increase. Therefore researchers developed additional techniques to reduce the required memory space [18, 19, 42, 49, 54]. However, in environments where memory capacities are greatly limited, the memory space savings achieved by these techniques may also not reduce bitstream storage needs to within the capacity limitations of these environments. Another side effect of bitstream compression is the additional hardware needed to perform FPGA-based (or online) decompression [54]. Again, in certain environments where smaller sized FPGAs are used, dedicating hardware resources for bitstream decompression may not be an acceptable option for designers.

Sellers *et al.* [54] developed a bitstream compression technique that does not require dedicated bitstream decompression hardware for FPGA boards with a limited memory. The technique targets FPGAs used in harsh radiation environments. Non-volatile memory used in these environments must be radiation hardened and typically have lower capacities than memory that is not hardened. Because of the low capacities, several radiation hardened memories may be needed to provide an adequate amount of storage for potential FPGA bitstreams and radiation hardened FPGA boards with greater available memory units are more expensive than those with non-hardened memories [54].

Sellers *et al.* propose a PR solution that uses bitstream compression via frame removal. The FPGA-based application proposed by Sellers *et al.* follows the Xilinx PR design flow; that is, the application is broken down into a static module and PRMs. The static module contains a microprocessor and additional hardware needed to initiate PR and route partial bitstreams to the FPGA internal configuration access port. Bitstream compression via frame removal is used on the static module bitstream to minimize the amount

of radiation-hardened flash memory space needed to store the module. Partial bitstreams are not compressed and are read from disk and or loaded via PCI or network transactions. A complete configuration process begins at power up where the static module is uploaded into configuration memory of the FPGA from flash memory and then used to bring in all remaining modules of the application via partial reconfiguration [54]. Once completed, the hardware application is intact and can execute its designed operations.

The compression via the frame removal technique proposed by Sellers *et al.* exploits the actions that are taken to prepare the FPGA for the receipt of its initial configuration bitstream (*i.e.*, the bitstream loaded at startup). At any time where the FPGA is initialized, the configuration memory of the FPGA is sequentially cleared (*i.e.*, made to have an empty configuration or basically "zeroed" out). Therefore, any frame in the initial configuration bitstream (*i.e.*, the static module) that contains only zeros can effectively be removed since the target frame already has an empty configuration. The removal process utilizes a tool developed by Sellers *et al.* that parses the original bitstream and detects which portions of the bitstream actually contain logic [54]. The logic frames are then extracted from the original bitstream and a bitstream is created with write instructions that support the new initialization sequence of frames that only contain logic. During the removal process, the extracted frames are also reorganized to place similar frames in sequence with each other. This reorganization allows for a multiple-frame-write command to be issued where frame data in the FDRI can be written to multiple frame addresses (where frame data similarities exists) before shifting a new frame into the FDRI [54].

2.3 PR Architecture Models and Applications

The Xilinx partial reconfiguration design flow allows some flexibility in the design of a PR application. A PR design includes design options such as partial bitstream storage options, region selection and placement, and PR configuration mode. In PR configuration mode, partial reconfiguration is either host, embedded processor, or hardware triggered and also includes configuration port selection. There are several research efforts that implement PR applications using models that employ different combinations of the above design options. This section presents published works that apply these design models in PR application implementation and PR related designs.

2.3.1 Partial Bitstream Encryption

A general architecture of a PR application includes a PR capable FPGA partitioned into static and PRRs, and an external storage to hold the partial bitstreams of PRMs. The external storage can be either memory (RAM of flash) or disk (magnetic or SSD), depending on how often PR occurs and the speed (*i.e.*, bandwidth) the bitstream can be delivered. partial reconfiguration applications (and SRAM FPGA-based applications in general) that use an external storage device for run-time reconfiguration are vulnerable to bitstream cloning, reverse-engineering, and tampering [13, 38, 61, 79].

When a system is compromised by an entity such as a virus, malware, or attacking user, software may be installed that is capable of monitoring internal and external communication traffic (*e.g.*, ethernet or hardware bus), or capable of recording data (*e.g.*, keystrokes, memory contents), or capable of augmenting the system; the remote user may choose to

perform these actions directly versus installing a software. Control commands sent across hardware communication buses (*e.g.*, PCIe, SATA, etc.) may be intercepted by a malicious software/attacking user, modified, and then rerouted to the specified hardware device. During PR, where a partial bitstream is read from external memory and loaded onto the FPGA, an attacker (or installed malicious software) can monitor this transmission and extract the configuration bitstream; alternatively, the attacker may opt to augment the bitstream while it is in transmission. Once extracted, the attacker can reverse engineer the bitstream to get the design of the encoded intellectual property if the bitstream is not secured; the malicious software may transmit the bitstream to a remote destination for the same purpose. Configuration vulnerabilities and potential attacks on SRAM-based FPGAs are outlined in Wollinger *et al.*, "Security on FPGAs: State-of-the-Art Implementations and Attacks" [61].

Bossuet *et al.* [13] present a bitstream encryption model to secure stored bitstreams using a PR capable FPGA (called secure PR). The method uses the FPGA to encrypt partial bitstreams in a two step process. The initial step of the secure PR process begins by loading a full non-PR encryption algorithm onto the FPGA. Decryption PRMs and other unencrypted PRMs (to be loaded during execution) are stored in external memory in the same step. In the second encryption step, all partial bitstreams to be secured are encrypted by the application and stored in external memory. If different encryption schemes are used, the FPGA is reconfigured with the different encryption algorithm for bitstream encryption. Bossuet *et al.* propose permanent storage of a secret key into the fabric of the FPGA [13] as a non-volatile core; an alternative is to use the FPGA internal memory and external battery

power to maintain the memory's state after powered off [13]. Once all partial bitstreams are encrypted, the FPGA is cleared of its configuration.

Bitstream decryption is handled during execution by the decryption PRM. Partial reconfiguration is used to load application components (encrypted or non-encrypted) onto the FPGA in two phases. The first phase configures the decryption module and other non-encrypted application parts onto the FPGA. The second phase decrypts the partial bitstreams and initiates the PR of the decrypted modules. If a different decryption module is needed, the current decryption module is reconfigured to another decryption PR module.

2.3.2 Partial Bitstream Encryption with Authentication

Zeineddini and Gaj present a model for secure PR focused on implementing secure PR after the initial configuration [79]. This bitstream encryption model is specifically designed for applications that benefit from PR. The PR architectural model is a single embedded processor with a single PRR and DDR external memory for encrypted bitstream storage. The embedded processor is responsible for initiating PR, decrypting partial bitstreams, and validating the authenticity of partial bitstreams. Zeineddini and Gaj use AES for decrypting partial bitstreams and the HMAC-SHA1 method for authentication [79]. However, the inclusion of the embedded processor in this model adds the flexibility of using any arbitrary encryption and authentication algorithm deemed necessary for any given PR application. All that is required is to upload C programs to the processor (during the initial configuration) that implement the desired decryption and authentication algorithms; the keys used by these algorithms are embedded in the algorithm's program. In this encryption model,

partial bitstreams are encrypted and signed on an external host system and then downloaded to the DDR memory using a Xilinx debugger tool. The initial configuration of the FPGA is triggered by the host system. Bitstream decryption and PR is executed by the embedded processor at run time after a successful authenticity check of the encrypted partial bitstreams.

Hori *et al.* [38] expanded the bitstream encryption model presented in Zeineddini and Gaj [79] by eliminating the embedded processor and introducing a pipelined and parallelized authenticated encryption hardware module. The PR architectural model of this design is a single PR region design with an external memory to store encrypted partial bitstreams. Partial reconfiguration is initiated by the authenticated encryption hardware module. The researchers' model includes an AES-GCM authenticated encryption module that is a combination of the AES block cipher with the Galois/Counter Mode (GCM) of operation. The GCM is a counter mode of encryption (used to convert a block cipher into a stream cipher) combined with the Galois mode of authentication [38]. Hori *et al.* uses the AES-GCM module for its pipeline and parallelizable properties that are optimal for hardware implementations. The module allows for simultaneous partial bitstream decryption and authentication, ultimately resulting in a shorter delay before initiating PR than a similar sequential method. This parallelization outperforms the authentication and decryption embedded processor-based approach in Zeineddini and Gaj [38].

2.3.3 Fingerprint Image Processing

Fons *et al.*, [29], implements a PR design to accelerate fingerprint image processing for a personal recognition security system. Currently, fingerprint-based personal recognition systems are used in several fields of application that require some form of identity managed access control [29]. There are two system architectures where the fingerprint recognition system is currently in use: the personal computer (*e.g.*, laptops) and embedded systems (*e.g.*, a low-cost microprocessor paired with an ASIC or an FPGA for acceleration tasks) [29]. Fingerprint recognition is executed in two phases, the enrollment phase and authentication phase. In the enrollment phase, legitimate users register an image of their finger by using a fingerprint device. The characteristics of the fingerprint are then extracted and stored for authentication [29]. During the authentication phase, an image is taken of the user's fingerprint, its characteristics extracted and then compared with the fingerprint data stored at user enrollment [29]. The fingerprint characteristic extraction and the matching process include a set of complex processing procedures. The focus of Fons' work is to accelerate these complex processing procedures through FPGA partial reconfiguration.

The PR architectural model used by Fons *et al.* is a single PR region for module reconfiguration, a static Xilinx Microblaze soft processor core [74], off-chip DDR and flash memories, a static memory management unit, and other static hardware units. The architecture also includes a PR controller that uses the ICAP configuration interface to load partial bitstreams at a peak rate of 3.2 gigabits per second [29, 71]. The ICAP configuration interface is an interface internal to the FPGA used for PR. This interface is discussed in Chapter 3. The Microblaze processor is responsible for driving the recognition application

flow, initiating PR, and monitoring the other static hardware components. The flash memory is the initial storage location for partial bitstreams and recognition application code to be executed by the Microblaze processor. The memory management unit is responsible for providing a buffered communication interface (via FIFO) between static and PR regions and also DDR memory. The DDR memory stores the application data and partial bitstreams transferred from flash at boot up, during application execution. The DDR memory also stores fingerprint images processed by the application. The fingerprint recognition application is a multi-stage image processing algorithm. Each stage in the algorithm is implemented as a different PR module. During execution, specific PR module configuration occurs only at the algorithm stage where the module is needed.

2.3.4 Network Virtualization

In a network based PR application, Yin *et al.* implement a heterogeneous FPGA-based virtualization platform that is comprised of several software-based and hardware-based virtual routing components (or routers). Virtual networks offer the flexibility of implementing various different types of network infrastructures to meet the requirements of a range of different network applications [78]. Software-based virtual networks have limited packet forwarding capabilities with performances, in technologies such as OpenVZ, of up to 300Mbps [78]. Yin *et al.* include a PR virtual network design to implement multiple PR virtual routers and for better packet forwarding performance in instances where a higher throughput is needed. To limit the gaps in hardware routed network traffic precipitated by a full FPGA shutdown for run-time reconfiguration, Yin *et al.* used partial reconfiguration.

With PR, the number of delayed traffic routes is relative to the number of reconfigured virtual routers.

The application model is a host driven PR architecture that uses a NetFPGA Virtex II Pro 50 PCI expansion slot card and a Linux based host system. The host system communicates with the FPGA board via the PCI port. OpenVZ user space instances (or containers) are used to implement the software-base virtual routers. The hardware-based virtual routers are implemented in logic as PR modules individually placed in their own PR region. The static region of the FPGA includes the communication hardware such as the input arbiter, the output queue, and I/O ports. A full system includes both software (SW) and hardware (HW) independent virtual networks monitored by the host system OS. The operating system is also responsible for monitoring the network and executing incoming network requests.

Partial reconfiguration depends on the condition of system and type of modification request received (more below). The system can be modified by making changes to a content addressable memory table that stores the virtual IP to virtual router mappings and then reconfigures the routers. Changes to the table can be done manually by the virtual network operator. Partial reconfiguration initialization is determined by the host system for the following three scenarios:

1. **Virtual network removal**: *A network change request requires a virtual network to be removed*. If a request to remove a virtual network is received, the host system either removes a SW virtual network by destroying an OpenVZ container or a HW virtual network by loading a blank partial bit stream into its PR region.
2. **Virtual network addition**: *There is sufficient bandwidth available that another virtual network can be added*. Because of the time it takes to program the FPGA hardware, the host system will first create a SW virtual network upon request. If it does

not create a SW virtual network, a HW virtual network is created using partial reconfiguration.

3. **Virtual network bandwidth adjustment**: *A request to increase or reduce virtual network bandwidth.* In either scenario, the host system determines if modifications to the network should be applied to the software module or the hardware module. If a request is received to reduce bandwidth, the bandwidth of the target virtual network is modified without affecting the other virtual networks. If the request is to increase the network bandwidth, the allocation of all HW and SW virtual networks modules is rebalanced. Yin *et al.* use a greedy algorithm to rebalance the network. For example, to make room for the target resource(s), the lowest bandwidth HW virtual network is migrated to software and vice versa for high bandwidth SW virtual network to HW.

2.3.5 Wireless Sensor Networks

Leligou *et al.* present some use cases for PR in wireless sensor networks [41]. Wireless sensor network (WSN) devices are limited in resources and power. Leligou *et al.* suggest that partial reconfiguration can be used when the WSN application has to adjust its behavior upon operator request or a change in the network or environment. A communication module may be reconfigured to regulate transmission power for a more efficient use of the remaining battery power. Authentication and encryption modules may be reconfigured to adjust security parameters (*e.g.*, key lengths and algorithms) for communication transmissions in the network. Partial reconfiguration can be used to implement algorithms that use large keys for a higher level of security, or smaller keys to reduce encryption/decryption computation and lower battery power consumption; a module may also be reconfigured to implement different block ciphers to accommodate outgoing messages lengths. The trade-off is the level of security versus the performance (*i.e.*, minimal power consumption) of the WSN device. Partial reconfiguration can be used to switch between high security to low power modes in support of the demands of the network.

The environment monitored by a wireless sensor network node, in general, is dynamic in nature. Therefore selecting which modules to use on a sensor node to account for these changes can be situational. Garcia *et al.* [30] propose a partial reconfiguration model that allows WSN nodes to swap in modules to adjust to changes in the monitored environment. Garcia *et al.* demonstrate this model by implementing a partial-reconfigurable situation-based Kalman filter target tracking kernel for a WSN [30].

The Kalman filter kernel is a path prediction method that is capable of estimating travel paths of sensed objects in noisy environments. Path estimations are done by filtering out the noisy elements and calculating the target's destination using its travel characteristics (*e.g.*, past trajectory, acceleration, speed, etc.) sensed by the WSN. The situation-based Kalman filter makes adjustments to the sensor modules in the WSN that deviate from the base Kalman filter settings to compensate for varying conditions. For example, if a sensor module is tracking a slow moving target (or on standby), it can switch to low power Kalman module that runs at a lower clock frequency and sample rate relative to the base Kalman filter setting. Additional examples can be found in Garcia *et al.* [30].

The PR architectural model proposed by Garcia *et al.* is called the Virtual Architecture for Partially Reconfigurable Embedded Systems (VAPRES). This architecture is a module-based PR design that uses a single soft processor with multiple PR region partial reconfiguration model. The model also includes flash-based partial bitstream storage accessed by a flash memory controller implemented as a static module for VAPRES. All PR regions represent a different clock domain for the flexibility of having several modules operating concurrently on their own independent clock. The processor is responsible for initiating

PR of Kalman filter modules at run time. A typical scenario starts with the acquisition of a target sensed by a target detection module on the sensor. At this time the processor initiates the PR of the appropriate Kalman filter target tracking module. Once configured, target tracking begins.

2.3.6 Cognitive Radio

Cognitive radio is essentially a reconfigurable radio that adjusts its communication modules in response to the condition of its associated network; changes may occur upon user request also. The concept of cognitive radio is considered the final step to software-defined radio (SDR) evolution [21]. Software-defined radio describes a technique of changing a communication system by reconfiguring the system without replacing the hardware components of the system. The goal is a universal wireless terminal that can support a wide range of wireless networks simply by making a change in its hardware configuration. Delorme *et al.* demonstrate the feasibility of an SDR terminal as a baseband modem. The baseband modem implements the multi-carrier code division multiple access (MC-CDMA) modulation technique using a mixed hardware board of two ASICs and two FPGAs [21]. The FPGA implements a FAUST network-on-chip (NoC). The FAUST NoC is a worm-hole switched embedded networked infrastructure that interconnects multiple independent functional units in the baseband modem. The NoC is the main media of communication in the baseband modem board and is also the target for partial reconfiguration [21].

The proposed PR architectural model is a single embedded processor with one PR region implemented on a stand alone ASIC-FPGA baseband modem board and connected to

a host system via an Ethernet communication port. On the host system is a radio communication application that monitors the signal-to-noise ratio (SNR) of the radio traffic handled by the baseband modem. If the SNR ratio reaches a certain threshold, the application signals (via Ethernet connection) the PR of the NoC to implement a different channel coder rate. The partial reconfigurable NoC design uses a single PR region and a MicroBlaze embedded processor. Partial bitstreams are stored externally in RAM and loaded into the FPGA using the ICAP configuration interface. The processor is responsible for initiating PR upon the receipt of an order received from the radio communication application.

2.3.7 Malware Collection

Mühlbach and Koch implement a malware-collection honeypot system entirely in hardware [48]. The purpose of this system is to emulate vulnerabilities for different Internet protocols (*e.g.*, IP, TCP, and UDP) and capture packets that attempt to exploit these vulnerabilities. The PR architectural model for the honeypot system, called MalCoBox, includes static Internet protocol intellectual property cores, a management controller, a PR controller with the ICAP configuration interface, and several PR vulnerability emulation handlers (VEH). Each VEH is an individual PR module designed to emulate a specific vulnerability or application; the resource requirements for each individual VEH can vary per implementation. Mühlbach and Koch use FIFO buffers for data to cross region borders to connect the VEH to the Internet protocol cores. A busy signal between the VEH and management controller is used to signal that the VEH is still processing data; PR may be delayed to wait for the VEH to finish before beginning. During execution partial recon-

figuration is used to configure or reconfigure VEH PR modules to add or remove different vulnerabilities. Partial reconfiguration initialization is done manually by console request through a dedicated network link (or PCI Express) to the management controller. The management controller, upon console request, signals the PR controller to begin PR using ICAP. The MalCoBox architecture has 20 different sized PR regions to meet the resource needs of the different VEH modules. These twenty regions also enable MalCoBox to emulate several different vulnerabilities and capture incoming packets using VEH modules operating in parallel.

2.4 Relation to this Research

The research efforts discussed in this chapter employed various PR application architectures ranging from host system-controlled PR to embedded processor-controlled PR. The design approaches follow the Xilinx basic PR design flow: perform PR to change the application function with design flaws and optimizations discovered by trial and error *after* the design has been written, built, implemented, deployed, and tested. These approaches do not perform PR to improve the performance of the application. In addition, an analysis is not done to determine the benefit of performing PR until after the application is fully implemented.

This dissertation presents a design technique that uses PR to improve the performance of a PR application pipeline. This dissertation also provides a mathematical PR model that will enable designers to make informed design decisions by using estimated (or measured) execution and reconfiguration times, pipeline buffer capacities, and bitstream sizes.

CHAPTER 3

BACKGROUND ON FPGA CONFIGURATION

An FPGA-based hardware application goes through several phases (*i.e.*, design, synthesis, map, place-and-route, and bitgen) before it is deployed onto the FPGA. In the design phase, the structure and behavior of the application are described by the designer using a hardware description language or schematic design. Application performance parameters (*e.g.*, driving clock frequency) and functional block configurations are also set by the designer in this phase. The synthesis phase essentially checks if the design is syntactically correct (like a compiler for software), translates the HDL code into an intermediate level netlist form, and estimates the resources needed by the design. After synthesis, the synthesis resource estimate is used to map the design to actual resources available in the FPGA fabric during the map phase. This is not a physical mapping, but the functional blocks of the design are translated into logic, and the number of CLBs and other resources on the FPGA that will be needed to represent these blocks is determined during this phase.

The map step produces a similar but more fine grained resource utilization estimate of the design. Since map, in essence, is another resource estimation phase, it is often excluded when discussing hardware implementation steps. However, the map step can require a long time to finish and therefore should not be excluded when considering the time

to complete building a hardware implementation. After map, the design (i.e. set of required resources) is mapped onto the fabric of the FPGA in the place-and-route phase. The FPGA is a large array of resources and, because it has so many resources, there are several different physical mappings that can be derived for any given design. Place-and-route is a process where the place-and-route tool selects a placement of the design functional blocks and routes signals to connect them. Place-and-route utilizes multiple passes where the tool derives several place-and-route solutions with varying performance capabilities within (or not within) the performance parameters specified by the designer. The place-and-route solution with the best achievable performance is then selected and encoded into a configuration bitstream during the bitgen phase. Place-and-route is actually the final step in the design implementation process (design, synthesis, map and place-and-route). When comparing time elapsed for each step to complete, map and place-and-route often complete in a time that is significantly greater than the time needed for the synthesis step, making synthesis relatively negligible. Furthermore, since place-and-route typically takes the longest time of all phases to complete, the term "place-and-route" is often used to represent the entire implementation process.

This chapter presents details on the FPGA fabric, layout, and the PR design flow framework. Section 3.1 outlines the basic building blocks of the Xilinx Virtex family of FPGAs. Although this dissertation focuses on Virtex FPGA devices, the techniques described in this chapter can be extended to other FPGA architectures. This section describes the basic functional blocks of the FPGA. Section 3.2 describes the organization of the functional

blocks of the FPGA. Section 3.4 discusses the basic steps that are performed by an application designer in order to create an application that utilizes PR technology.

3.1 Functional Blocks in the FPGA

The FPGA fabric is a finite set of CLBs, BRAMSs, DSPs, DCMs, MGTs, Ethernet and PCIe blocks, and general purpose I/O buffers, all of which are the functional blocks of the FPGA. Signal routing resources, also a part of the FPGA fabric, are responsible for connecting the various functional blocks together. Although any arbitrary FPGA-based hardware application (PR or non-PR) may consist of a combination of these functional blocks and signal routing resources, this dissertation focuses on the CLB, BRAM, and DSP blocks. These three functional blocks are the building blocks utilized by PRMs and may contribute to the invocation and execution of PR.

For PR applications, there are several functional blocks that must be designated as fixed blocks [72]. The MGT, Ethernet, PCIe, and general purpose I/O blocks are a part of the fixed configuration of a PR application. They are fixed because the design of the board that the FPGA is mounted on statically determines the I/O configuration of any application deployed on that FPGA. That is, the signal lines on the board that lead to its single purpose (*e.g.*, PCIe, Ethernet, SATA) I/O connectors are permanently affixed to the I/O pins on the FPGA and therefore dictate the application's I/O configuration.

Clock signals are also fixed. The driving clock of an application is typically statically defined by the designer, and therefore DCMs are a part of the fixed configuration of a PR application. Utilization of the signal routing resources is determined automatically by the

place-and-route tool and therefore the signal routing resources are not explicitly controlled by the application designer, and by extension are not discussed in this dissertation.

3.1.1 The Configurable Logic Block

Figure 3.1 is an example of a CLB. The CLB contains a pair of slices, with carry-in (labeled CIN) and carry-out (labeled COUT) signals, connected to a matrix of routing resources. The CLB is the FPGA's main logic source for implementing circuits [65]. The *slice*, within the CLB, is the lowest level hardware component of the FPGA. Each slice is composed of *look-up-tables* (LUTs), storage elements (edge-triggered or level-sensitive memory), multiplexers, and carry logic (CIN and COUT) [65], all used by the slice to provide logic, ROM, and arithmetic functions.

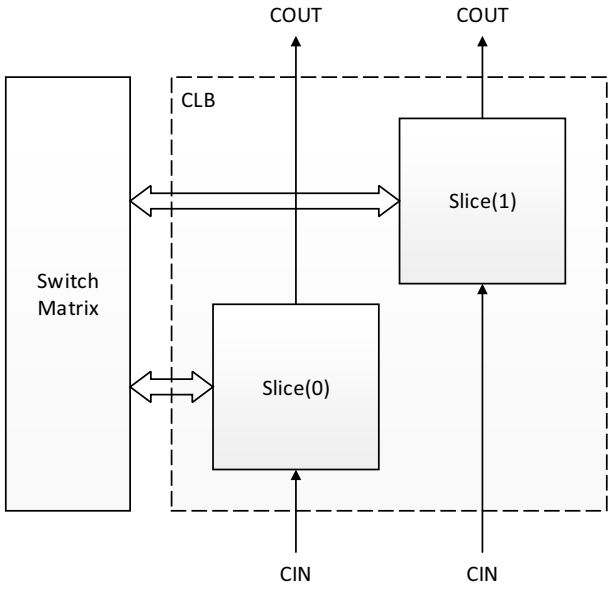

Figure 3.1

A CLB and connected routing resources

In the Virtex-5 and later FPGAs, the slice contains four 6-input LUTs that return a single output for any combination of six independent input values. The look-up-table of the slice is a logic function generator [65], which is discussed further below. The other components of the slice provide temporary synchronous storage (one cycle) and logic for transferring data between the slices of connected CLBs.

The LUT is capable of implementing any arbitrary 6-input Boolean logic function programmed to the FPGA. The LUT can be configured to implement a synchronous distributed RAM with data storage capacities of 32, 64, 128, and 265 bits that supports single-, dual-, and quad-port configurations (one LUT per maximum 64 bits single-ported). When configured as a ROM, the LUT shares the same capacity (and port) options as the RAM configuration beginning at 64 bits.

The LUT can also be configured to implement a shift register capable of delaying serial data from one to 32 clock cycles. The LUT also has shiftin pin and shiftout pins that allows the LUT in a slice to be cascaded together to implement larger shift registers that allow delays of up to 128 bits (32 bits per cascaded LUT). The shift register configurations are useful for hardware applications that require the use of delays or latency compensation for efficiency purposes. They are also useful for synchronous FIFO and content addressable memory designs [65].

Manipulating the configuration of the slice is not advised except for advanced designers with an in-depth knowledge of the FPGA architecture and the use of Xilinx's FPGA editing tools. However, the various slice functions can be inferred by the behavior(s) described in the hardware design or by the instantiation of special purpose "soft" hardware cores in the

design. The term "soft" means the core is realized by the CLBs and its slices and is not a dedicated special purpose "hard" core built into the FPGA (*e.g.*, BRAM and DSP).

3.1.2 The Block RAM

The block RAM on Virtex FPGAs is a hard embedded dual ported random-access memory slice. A single BRAM can store up to 36K bits (36×2^{10} bits) of data and can be configured as a single RAM (the RAMB36) or as two independent 18K bit RAMs ($2\times$RAMB18) [65]. Each configuration can be organized to implement memories of varying data widths and depths where width is the number of bits per stored data item and depth is the number of items that can be stored.

The dual ported BRAM has two sets of data input, data output, address, clock, and enable ports. Both sets of ports are independent from the other and some ports within a set are independent from other ports in the same set. The dual clock ports allow for independent driving clocks. This means that a transaction on one port (*e.g.*, read/write) can be driven at a different clock frequency from another transaction on the other port. The BRAM also contains dual cascade-in and cascade-out ports. These ports enable multiple BRAMs to be cascaded together to implement larger memories.

Block RAM read and write operations are synchronous and the read (*i.e.*, data output) and write (data input) ports are independent, sharing only the stored data [65]. Each read and write port may also be configured to different data widths independently of each other as long as they are one of the available bit widths achievable by the implement memory. Two concurrent read or write operations (or a read-write combination) may be exe-

cuted by the BRAM using the dual input/output ports, address ports, and enable ports. To avoid read/write collisions, the BRAM can be configured to operate in read-before-write, or write-first, or no-change modes, meaning the data output will reflect the previous data at the specified address, or the newly written data, or remain unchanged, respectively.

In a hardware application, a single BRAM or a cascade of BRAMs can be used to implement embedded dual/single-ported RAM, ROM, and synchronous FIFO memory configurations [65]. The dual clock support for the BRAM contributes to the traditional use of FIFOs as a means for transmitting data across the boundaries of differing clock regions in the FPGA fabric.

3.1.3 The Digital Signal Processor Block

The digital signal processor (the DSP48E slice) of the Virtex FPGAs provides dedicated hardware support for arithmetic/logic functions without the use of the general FPGA fabric (*i.e.*, the CLB) [67]. Like the BRAM, the DSP48 is also a special-purpose hard core embedded as a slice in the FPGA fabric. The DSP48E supports several functions such as multiply add, multiply accumulate, multiply, and three-input add functions. In addition, the DSP48E can also be configured to perform bit-wise logic functions, wide-bus multiplexing and, by cascading multiple DSP48Es together, designers can build modules to support complex arithmetic, wide math functions and DSP filters [67].

3.2 Physical Organization of the FPGA Resources

This section discusses the resource organization and resource structure sizes that are specific to the Xilinx Virtex-5 FPGA. However, the values are only used to provide a

tangible point of reference to aid the reader in conceptualizing the structures that make up the physical organization of the FPGA resources. How resources are organized in the FPGA fabric has been fairly consistent with little deviation (except for specific resource quantities) between FPGA models and generations. Therefore, the resource organization of the Virtex-5 FPGA discussed here can also be extended to apply to FPGAs in general.

The Xilinx Virtex family FPGA fabric is an addressable configuration memory space where FPGA resources (*e.g.*, CLBs, BRAMs, DSPs, I/O buffers, clock managers) are mapped and arranged in a matrix of rows and columns. The basic organizational block of a Virtex FPGA is a homogeneous array of vertically stacked resources called a *major column*; signal routing and clocking resources are also included within this major column.

A major column in a Virtex-5 is organized as array of either 20 CLBs, or four BRAMs, or eight DSPs, or 40 I/O buffers, or a fixed set of any other resource [66]. The major column is further organized to contain an upper, lower, and center partition. The resources contained by the array are spilt evenly into the upper and lower partitions, for example, the upper and lower partitions of a CLB major column consist of 10 CLBs each. The upper and lower partitions in the major column are not mutually exclusive. This means that if a configuration loaded onto the FPGA only uses the resources in the upper partition, the entire major column is consumed. This is because the major column is the lowest level addressable configuration memory space of the FPGA called a *configuration memory frame* or *reconfigurable frame* [66, 72].

The center partition of the major column is reserved for signal routing and clocking resources that drive the resources contained by the array. If the resources in the array are

cascadable or contain carry signals, the cascade/carry-out signal of one resource is tied to the cascade/carry-in signal of the resource above it. Figure 3.2 shows a subset of two CLB major columns with connected slice carry signals. The slices use a XY position labeling increasing from left-to-right and bottom-up.

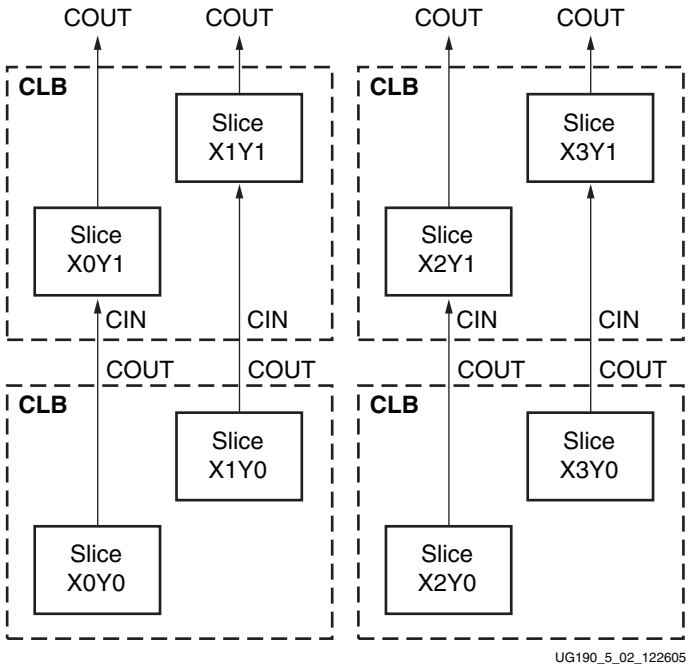

Figure 3.2

Row and column relationship between the CLBs (Xilinx UG190 v5.3)

A row in the FPGA fabric is a set of major columns placed horizontally across the FPGA. An FPGA has several rows organized vertically. The Virtex-5 FX200T FPGA, for example, has 12 rows of major columns. The height of a row is the height of a major column and the height of all major columns are equivalent, regardless of the type of resource

the column contains. A row consist mostly of CLB major columns with BRAM and DSP columns interspersed within.

The CLB and BRAM major columns are relatively uniformly distributed throughout the row. However, the DSP major column is clustered closer to the center of the FPGA. The irregularity of the major column distribution within a row is exacerbated by the presence of other resource types such as I/O columns and embedded hard processors that take the place of CLB and BRAM major columns. But, with the exception of the other resources, the relative positioning of the CLB, BRAM, and DPS major columns within a given row is the same in all rows. For example, if a DSP major column in an FX200T FPGA, located in row one is at column position 47, there is also a DSP major column located at column position 47 of rows 2-12.

3.3 Basic PR

Traditionally, non-PR hardware applications are designed as multi-module applications that consist of a module for each functional capability the application is to provide (see Figure 1.1 (p.14)). Furthermore, the application has logic that dynamically selects these modules when a functional change is necessary. The additional modules are used to process the various types of input the application is expected to receive from its operating environment. However, the added modules require more FPGA logic resources to realize and more routing resources to connect.

Basic PR design is the partial reconfiguration design approach outlined in the Xilinx PR design tool flow and is the most commonly applied technique used for PR applica-

tions. The flow highlights the design of a PR application that contains fixed logic paired with a single partial reconfiguration region for loading PRMs (Figure 1.2 (p.16)). A basic PR application is a replacement for the non-PR multi-module application designed that adds dynamic module selection. Like the non-PR multi-module application, the basic PR design approach is suited for hardware applications tasked to provide multi-function capabilities by dynamically modifying itself to meet the demands of a changing computational demands.

The objective of the basic PR is to reduce the application time overheads (frequency and propagation delays) and space complexity (signal routing and resource utilization) that result from the need to provide modules for all supported functions in a non-PR approach. This objective is achieved by implementing the supported functions as interchangeable PRMs where, when a change in function is necessary, the associated PRM is loaded into the PRR on the FPGA. For example, Anderson *et al.* [9] used the basic PR design to realize an FPGA-based autonomous system. The system was developed to detect and counter various jamming signals. The system determined the type of jamming signal that was being used and invoked PR to load the appropriate filter onto the FPGA. When the jamming signal changed, the system loaded the corresponding filter on-the-fly.

The PRR of a PR application is selected to contain a union of the resources needed by all PRMs that can be resident within the PRR. Each PRM to be loaded into a PRR must have the identical I/O port interface. This means that if one PRM is designed with 10 I/O ports, all other PRMs must have a 10 port I/O interface regardless if these ports will or will not be used.

This research forgoes the basic design approach for an advanced PR design technique focused on improving the performance of a PR application pipeline (Chapter 4). However, a description of the basic PR design flow is provided in the following section.

3.4 State of the Art in PR Design Flow

The basic partial reconfiguration (basic PR) software tools design flow [72] is a multi-step process that includes the synthesis, map, place-and-route, and bitgen phases of the hardware application development process. Basic PR also requires the use of the Xilinx design, analysis, and implementation tools to build a complete PR application.

The Xilinx basic PR design flow can be broken into a series of three procedures:

Step 1: HDL design, module organization, configuration interface integration, and module synthesis

Step 2: PRR floor planning, communication channel insertion, and timing constraints

Step 3: Application implementation and debugging, configuration bitstream generation

In the basic PR design flow, the tasks outlined in the above procedures (or steps) are primarily manual tasks with the occasional use of Xilinx automated tools to generate the necessary files for a succeeding task or step. The tools used in the process are the Xilinx Synthesis Technology (XST) tool (available in the Xilinx ISE Design Suite) and the Xilinx PlanAhead tool [73, 75].

The ISE Design Suite (ISE) is a GUI that provides a development environment and tools for designers to design and build non-PR hardware applications. The ISE design suite is used for Step 1 of the design flow. It does not have the necessary tools to build PR applications, but it can be used to develop and synthesize modules for PR applications.

The PlanAhead tool (also a GUI) is a development environment used for Steps 2 and 3. Files generated from ISE are used by PlanAhead to build a PR application. Designers may choose to used the Xilinx console-based tools as an alternative over these GUIs. The instructions for the console-based approach, which includes invoking commands and arguments, are provided in the basic PR design flow documentation [72].

3.4.1 Step 1

Design parameters/restrictions critical to Step 1 of the basic PR design flow are outlined below and elaborated in the following subsections:

- HDL design must be hierarchical and contain a top-level module
- Modules must be either fixed or dynamic and dynamic modules must not contain I/O resources
- The ICAP configuration interface used for performing PR is considered asynchronous and should be constrained to achieve a synchronous behavior
- PRMs must be synthesized as a single individual unit with all sub-modules synthesized first
- Synthesis and HDL: PRMs that share the same PRR must have the same name and I/O ports
- Netlist files are generated for use in Step 2

3.4.1.1 HDL Design

A hardware description language is a high-level language used to describe a *behavioral* implementation (synchronous/asynchronous actions taken per a given input, *e.g.*, a finite state machine), and/or *architectural* implementation (black-box elements combined with connecting wires to create circuits) of a designed hardware. Designers that possess only a

functional knowledge of circuit design can use an HDL to describe the behavior of hardware without having to be an expert in circuit design. The hardware build tools infer these circuits from the HDL. An HDL is analogous to using the C high-level language to write software versus writing software using assembler/machine code.

Partial reconfigurable applications are written as hierarchical designs. In a hierarchical design, the functions of the application are *modularized*, where each function is broken down into sub functions and written as individual testable logic modules. The sub function modules are instantiated in another module and combined with the HDL of the instantiating module to realize a larger function. The instantiating module is the *parent* module. The parent module is then instantiated by another module and so on until the application design is fully written.

The upper-most parent module, the module that instantiates all remaining parents, is called the *top-level module*. For PR, the top-level module must be an architectural module, *i.e.*, it must not contain any behavioral HDL. The PR application's top-level module HDL must only instantiate other modules and the signals that connect the modules together. In contrast, PRMs can contain behavioral HDL but they cannot contain instances of static modules. Furthermore, PR applications must contain control logic that handles closing off or rerouting the communication channels established by the wired connections between the PRMs and other logic. Control logic is needed to prevent corrupting the data across these channels during PR.

However, knowledge of function and operating environment will not always clarify that a PR design approach is worth the effort without conducting an analysis and observing the

execution of a non-PR design first. The analysis of a non-PR design can expose points where PR can improve upon (or extend) the capabilities of the application or show that a benefit is unobtainable.

When using PR, an application can have a set of different configurations. That is because each unique implementation of a distinct module combination is itself a *configuration*. In a previous version of the basic PR design flow [63], Xilinx included a recommended step for designers to implement and test non-PR versions of the potential application configurations. The step was listed as critical for debugging, initial timing and placement analysis, and for determining the placement of communication channels. Validating application configurations is proposed applications configurations are still recommended by Xilinx but not included as a step in the design flow.

3.4.1.2 Module Organization

This step partitions the modularized design into either fixed (static) modules or dynamic modules (*i.e.*, PRMs). Whether a module is either a static or a dynamic candidate is determined by the designer who should consider factors such as resources used, criticality to the application, communication, and, if possible, discovery of design options determined by conducting a preliminary analysis of a non-PR implementation. Modules that use resources that are not CLBs, BRAMs, or DSPs must be static [72]. This means global clocking, clock modifying, and I/O related resources must all be static. Major columns of I/O buffers are static resources because they connect to the static I/O pins of the FPGA.

The communication channels between modules are either static-to-static or PRM-to-static. A PRM cannot directly connect to another PRM. PRM-to-PRM communication paths must be routed through a static parent module. The designer must also consider how the modules are to communicate, determine channel lengths when partitioning the design, and consider how the application modules will be arranged on the FPGA fabric (floorplanning in Step 2).

Module criticality is determined by how the module (particularly its absence) affects the execution of the application. Modules that have any part in the application's basic function should be made static. The basic function can be considered the application's invariant path where modules within this path never change. Therefore, these modules should be fixed and designated as static. Independent modules not a part of the invariant path are dynamic module candidates.

Xilinx does not provide a tool for partitioning modules into invariant and dynamic groups nor for determining how critical the module is to the basic function of the application, nor for determining if two or more modules are independent or mutually exclusive. This responsibility is left to the designer. It is because of this responsibility that is it common to construct and analyze a non-PR implementation first, before considering a PR equivalent.

3.4.1.3 ICAP and Bottom Up Synthesis

There are several configuration interfaces with varying data transfer rates that can be used for PR. However, instructions for using the Internal Configuration Access Port are

provided in the basic PR design flow; it is also the chosen interface for this dissertation research. The Internal Configuration Access Port (ICAP) is a configuration interface primitive that can only be utilized directly from the FPGA fabric. The ICAP interface for Virtex-5 FPGAs is the ICAP_VIRTEX5. The primitive is a six-port black-box that allows users to access the configuration memory (fabric) of the FPGA, read back the configuration data, or perform PR upon the FPGA after the initial configuration has completed [66, 70]. The primitive also has a 32-bit data-bus width for configuring the FPGA. Therefore, when operating at a clock frequency of 100MHz, the ICAP_VIRTEX5 has a maximum bandwidth of 3.2Gbps when loading a bitstream. The controller module of a PR application will interact with this primitive for executing PR.

Bottom-up synthesis is a manual repetitive process that starts with the synthesis of lowest-level modules of a hierarchical design. The process continues up the hierarchy with the synthesis of the parent modules, and grandparent modules, up until synthesis of the top-level module (*i.e.*, the entire application) has completed. This procedure ensures that for each module all *dependencies* are synthesized first.

Any design of an instantiated dependence that is absent during the synthesis of the parent module is assumed implemented and synthesized as a black box. Partial reconfiguration module instantiations must always be synthesized as a black box, *i.e.*, the PRMs' HDL design must not be provided when synthesizing the parent static module. Static modules of a PR application may be synthesized collectively as long as the called static dependencies are synthesized first to avoid being designated as black boxes.

3.4.2 Steps 2 and 3

Design parameters/restrictions critical to Step 2 and 3 of the basic PR design flow are outlined below and elaborated in the following subsections:

- Netlists from Step 1 are required for PlanAhead to build PR applications
- PlanAhead is used for designating PRRs and building PR applications
- PRRs must be set at CLB boundaries
- A single resource reserved in a major column reserves the entire column
- The PR application build process generates multiple bitstreams

3.4.2.1 PlanAhead for PR

The Xilinx PlanAhead software is a GUI application used to complete Steps 2 and 3 of the basic PR design flow. Designers can create projects in PlanAhead enabled with PR support. Static logic and PRMs are added to the PR project by importing the corresponding netlist. The static logic netlist is imported at project creation. User-defined I/O and timing constraints can also be imported at project creation. The netlist for the PRMs and constraints are imported later when floorplanning the PRRs. Once the project is created, the PlanAhead toolset enables designers to create and position PRRs on the FPGA fabric, modify or generate user-defined I/O and timing constraints, run PR design rule checks, implement and verify the different application configurations, and generate the configuration bitstreams for each application configuration including partial bitstreams that are generated for PRMs.

3.4.2.2 PRR Floorplanning and Building the Application

When using the Xilinx ISE and PlanAhead tools for the design and implementation of non-PR and PR hardware applications, a unique FPGA model and package (fabric and I/O pin layout) is specified by the designer as the target for deploying the application. This information is provided during project creation of the respective tool and/or provided in a user-defined constraints file called the UCF. The UCF is where I/O, timing, and location constraints are defined for the hardware application. *Constraints* are rules set by the designer that the synthesis, place-and-route, and implementation tools must attempt to follow when building the application. The UCF can be written by the designer and included as a part of the application project or generated by using ISE or PlanAhead's UCF creation tool.

Partial reconfiguration regions are created by specifying location constraints in the UCF, or creating partition blocks using the floorplan view in PlanAhead. The floorplan view is a visualization of the target FPGA fabric that shows the rows, major columns, I/O pins, and other resources that are a part of the FPGA package. Designers create PRRs by drawing a rectangular section in the floorplan view, encapsulating a set of FPGA resources, and exporting the dimension of the section (*i.e.*, location constraints in slices) to the UCF.

Designers determine the geometry and location of PRRs by looking at the resource utilization of the static modules and PRMs and using the floorplan view to visually inspect the physical organization of the FPGA. Module resource utilization is provided by importing PRMs' netlists. The height and width of a PRR can vary, as long as it encompasses all the

resources utilized by all PRMs to be configured in that region. If one PRM uses BRAMs and another PRM uses DSPs, the PRR must enclose both resources.

The basic PR design flow requires that all PRRs be constrained to CLB boundaries. This means that both the bottom left and upper right of the PRR must encapsulate a single CLB. Furthermore, any one resource or a major column enclosed by a PRR reserves the entire major column, making the remaining unenclosed resources unavailable to any module. Therefore, designers should assign PRRs that encompass entire major columns.

Once floorplaned, the application is built by first performing a PR design rule compliance check and then implementing the application. The next step is verifying that the implemented application configurations meet the PR design rules. The last step is generating the fixed and dynamic logic configuration bitstreams. The user constraints, with PRR designation and other defined constraints are applied in the application implementation step. All four steps are handled by the Xilinx PlanAhead tool.

3.5 The Need for Advanced PR Design

The basic PR design flow builds PR applications that swap/replace PRMs to change the function of the application [15, 72]. This dissertation focuses on building PR applications that swap PRMs to improve performance of the application. In this research, two advanced PR design techniques (see Chapter 4) were developed for implementing applications that use PR to improve performance.

The basic PR design flow works well when PR is used to replace modules with alternative functionality and when performance/timing is not a factor. However, when modules

need to be swapped/replaced in order to improve performance, the need for estimating the time taken to perform the reconfiguration (*i.e.*, reconfiguration overhead) must be taken into account. Any performance gained by swapping the PRM of an application for another can be nullified by the process if the reconfiguration overhead is greater than the time gained by the improved performance.

The Xilinx PR tools do not estimate the reconfiguration time needed for PRMs. The designer must conduct this estimate by implementing the PR application and measuring the reconfiguration time during execution or performing reconfiguration time calculations using the size of the bitstreams generated from the realized PR application. However, as a part of the PR application development cost, it is typical that a PR application's initial design is derived from the HDL of a non-PR application. In an instance where the non-PR design does not fit on the FPGA, a case for using PR to swap in functional modules upon demand can be made. Otherwise, the feasibility of a PR design and potential PRMs are often discovered by analyzing an already completed hardware application. Designers may speculate on which functional components of the application are candidates for becoming PRMs, and in theory, a PR application is no different from a non-PR equivalent. However, knowledge of function and computational demands will not always clarify that a PR design approach is worth the effort without conducting an analysis and observing the execution of a non-PR design first. The analysis of a non-PR design can expose points where PR can improve the application's performance or show that a performance improvement is unobtainable.

To develop a PR application that swaps PRMs for the purpose of improving performance, knowing the reconfiguration time of a PRM is a necessity. To gain this knowledge, a complete PR application must be built. However, given the time required to design and build, it is impractical to realize PR applications just to discover that the reconfiguration overhead for swapping PRMs outweighs the performance gained from the swapped-in PRM.

This research has developed a formula for estimating the bitstream size of a PRM using the resource utilization estimate provided by the synthesis step of the hardware implementation process. This research had also developed a formula for estimating the reconfiguration overhead of a PRM given the size of the PRM's bitstream. The PR model provided by this research requires the bitstream size and reconfiguration time to be known. These estimation formulae have been developed to give designers a means to estimate reconfiguration overheads of an application without having to build the PR application. These formulae are presented in Chapter 4.

CHAPTER 4

FRAMEWORK

The application of partial reconfiguration can be categorized into the following use cases:

Case 1: The need to change application functionality to support environment changes, and

Case 2: The need for improved compute performance.

Hardware applications designed to support Case 1 perform basic PR, a technique (described in Chapter 3) recommended in the Xilinx PR design flow [72]. These applications use PR to change functionality upon demand by swapping PRMs. For example, Anderson *et al.* [9] developed an adaptable signal processing autonomous system using the basic PR design. The system is designed to combat signal jamming by selecting a filter module based on detected jamming signals. Because of physical and hardware constraints, only one filter module instance can exist on the FPGA at a time. During execution, the appropriate filter module is loaded on-the-fly as needed. This basic PR design approach is the most commonly used design for PR applications.

This dissertation focuses on Case 2, the use of PR to improve compute performance. In this case, modules are dynamically replaced for other modules based on computing conditions and demands. For example, a compute intensive application can replace I/O modules with compute modules in order to repurpose the I/O module resources for computational

needs with the goal of reducing compute time. However, this approach is viable only if the time taken to perform the PR operations (exchanging an I/O for a compute module and back) is less than the time saved from the added compute performance. In this example the I/O module should not be confused with the static I/O block resources of the FPGA. The I/O modules are user-defined functional modules that can be reconfigured using PR.

4.1 PR for Improved Performance

This dissertation targets two different scenarios where PR can be used to improve performance:

- Bottleneck mitigation, and
- Mismatched processing capacity.

The sections below provide further details on how PR can be used to improve performance.

4.2 Bottleneck Mitigation Using PR

Hardware applications are typically implemented as a parallel computation pipeline. In general, a computation pipeline is two or more processing operations interconnected in a line of successive processing stages. Data passes from the first processing stage to last stage where, with the exception of the first, each processing operation is performed upon the data it receives from the previous processing stage. Computation pipelines use FIFO memories to interconnect and buffer data between modules. The use of FIFOs as a buffer is a classic technique for controlling the flow of data between producing and consuming modules that may not operate in sync with each other. First-in first-out memories reduce the chance of pipeline stalls due to idle waiting.

Figure 4.1 is an example hypothetical two stage pipeline using a single FIFO buffer. Data traverses the pipeline from left to right beginning with the Process$_1$ (P_1) operation, through the FIFO, and then the Process$_2$ (P_2) operation. Assume, without loss of generality, that input data is available for P_1 upon demand and the P_2 operation result data is posted (*i.e.*, sent to its destination without waiting for an acknowledgement of receipt but arrival is guaranteed). Both processing operations are performed concurrently, each taking the time needed for the corresponding operation to complete.

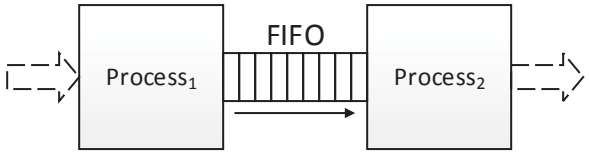

Figure 4.1

A general two process pipeline

Assume that the production rate of P_1 is r_{P1} and the rate for P_2 is r_{P2} where $r_{P1} \neq r_{P2}$. Furthermore, assume that the P_2 operation takes significantly longer to complete than P_1, such that $r_{P2} \ll r_{P1}$. This means that as data traverses the pipeline, the rate in which a result can be generated is determined by the production rate of P_2. Therefore, the P_2 operation is the pipeline bottleneck. Additionally, as execution continues, P_1 will eventually fill the FIFO due to its faster production rate, forcing it to wait (*i.e.*, stall) until P_2 consumes data from the FIFO thereby freeing up space for additional output from P_1.

In an FPGA-based hardware application, the processes in a pipeline are performed by custom logic-based hardware modules. To reduce the effects of a pipeline bottleneck, PR can be used to reallocate the logic resources of a stalled module and repurpose (*i.e.*, reconfigure) them to realize an additional bottleneck process/module. The reallocation and repurposing of FPGA resources is a technique called "resource sharing PR" that has been developed by this research to improve the compute performance of pipelines.

In Figure 4.1, if PR reallocates P_1 resources to realize an additional P_2, it is clear that the PR process breaks the pipeline by removing P_1 from the chain. Therefore, the resource sharing PR technique is a two step process: 1) the reallocation of resources to realize additional processing units to mitigate pipeline bottlenecks, and 2) once enough data has been processed, the reconfiguration of these resources back to their original module. Step 2 is possible because of the FIFO between pipeline stages. Because the FIFO buffers data between the two process stages, the application can determine when to perform PR by monitoring the rate in which the FIFO is filled (to perform step 1) and emptied (to perform step 2). In addition to monitoring the FIFO, the reconfiguration overhead of the PR process must be also considered when determining a threshold with respect to FIFO vacancy for performing resource sharing PR. Knowing the reconfiguration overhead of swapping PRMs also aids designers in determining whether using resource sharing PR to improve performance is a feasible option. If the reconfiguration takes so long that the input FIFO is empty (or nearly empty) when the additional PRM comes online, then performing resource sharing PR provides no benefit.

This dissertation research has developed (1) a formula for estimating the bitstream length of a PRM, (2) a formula for estimating the reconfiguration overhead (using the bitstream length), and (3) a formula for using the reconfiguration overhead and FIFO fill/empty rates to determine full/empty thresholds for executing PR. These formulae are provided in Sections 4.4 and 4.5. Details of the resource sharing PR architectural module is provided in Section 4.2.2.

4.2.1 Module Selection Options for Bottleneck Mitigation

The fill rate of the FIFO and estimated reconfiguration overhead can be used to determine thresholds for executing resource sharing PR. However, designers must also select which module(s) in the pipeline to reconfigure. In Figure 4.1 (p.76) where the processing rate relationship of P_1 and P_2 is $r_{P1} \gg r_{P2}$, it is clear that in attempt to improve performance, P_1 will be swapped for an additional P_2 (vice versa for P_2 if $r_{P1} \ll r_{P2}$); the additional Process$_2$ module will also read input values from the FIFO.

As the number of processes/stages in the pipeline increase, it may not be clear which module is an ideal candidate for resource sharing or which module the candidate should realize. Figure 4.2 is a hypothetical three stage pipeline where processing operations P_1, P_2, and Process$_3$ (P_3) produce results at rates r_{P1}, r_{P2}, and r_{P3} respectively. There are 27 different processing rate sequences achievable in a three stage pipeline. These sequences can be reduced to the rate relationships shown in Table 4.1.

In the relationship of column 1 in Table 4.1, the rates for the three processing operations are so close that it would be inefficient to use resource sharing PR for bottleneck

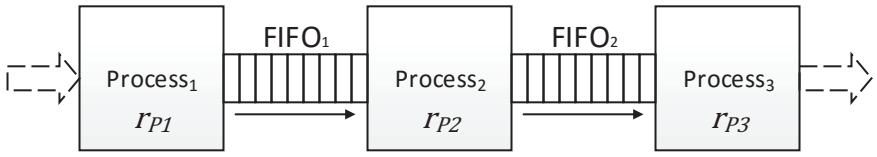

Figure 4.2

A general three process pipeline

mitigation. Reconfiguring one processing operation module into an additional bottleneck module can be achieved, but, because their processing rates are similar, the reconfiguration overhead would negate any gains in performance. In the case where the production rates for the processing operations are identical (*e.g.*, all three perform the same accumulation operation), FIFOs and resource sharing PR are not needed because each processing operation performs at the same production rate. If the process modules go out of synchronization, the register that is normally placed between stages grants the successor module time to retrieve data from its predecessor without stalling the predecessor module.

Table 4.1

Processing rate relationships of a three stage pipeline

1	**2**	**3**
$r_{P1} \approx r_{P2} \approx r_{P3}$	$(r_{P1} \approx r_{P2}) \ll r_{P3}$	$r_{P1} \ll r_{P2} \ll r_{P3}$
	$r_{P1} \ll (r_{P2} \approx r_{P3})$	$r_{P1} \ll r_{P3} \ll r_{P2}$
	$(r_{P1} \approx r_{P2}) \gg r_{P3}$	$r_{P2} \ll r_{P3} \ll r_{P1}$
	$r_{P1} \gg (r_{P2} \approx r_{P3})$	$r_{P2} \ll r_{P1} \ll r_{P3}$
	$(r_{P1} \approx r_{P3}) \ll r_{P2}$	$r_{P3} \ll r_{P1} \ll r_{P2}$
	$(r_{P1} \approx r_{P3}) \gg r_{P2}$	$r_{P3} \ll r_{P2} \ll r_{P1}$

The relationships of column 2 in Table 4.1 represent pipelines similar to Figures 4.3(a) and 4.3(b), where processing operations of identical/similar rates appear adjacent to each other, or are split by another processing operation. For the same reason as for the column 1 pipeline in Table 4.1, successive process modules performing at identical/similar rates (Figure 4.3(a)) are not good candidates for resource sharing PR because the reconfiguration overhead would negate the performance gained.

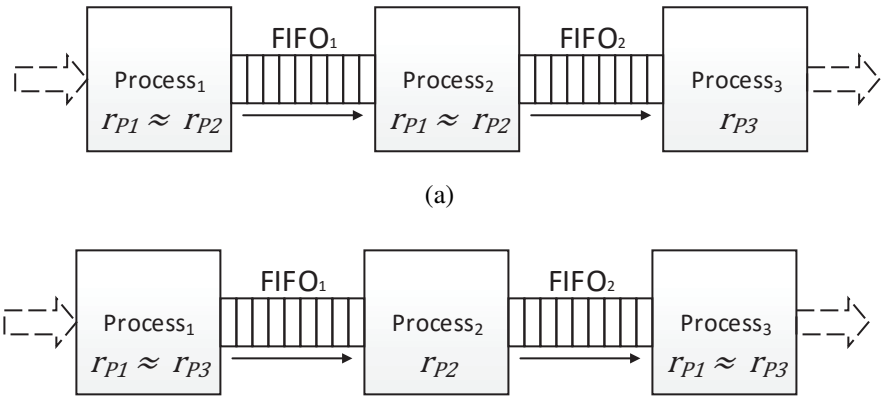

Figure 4.3

Two example three-process general pipelines

(a) Two processes in succession with identical/similar rates
(b) Two processes with identical/similar rates split by another

Therefore, in the relationship $(r_{P1} \approx r_{P2}) \ll r_{P3}$ the obvious candidate for resource sharing PR is P_3 because both P_1 and P_2 are bottlenecks. However, it is not ideal to use resource sharing PR for this pipeline because of the successive bottleneck process modules. Regardless of which additional bottleneck module (*i.e.*, P_1 or P_2) is realized, the

performance gained will be limited by the production rate of the other bottleneck module. For example, if an additional P_1 was realized, the inbound rate of FIFO_1 (*i.e.*, the rate in which the FIFO is filled) will double but its outbound rate (*i.e.*, the rate in which the FIFO is emptied) will remain the same, limited to the rate of P_2 (assuming P_2 consumes data at the rate r_{P2}); an additional P_2 would be limited by r_{P1}. Similarly, in the relationship $r_{P1} \gg (r_{P2} \approx r_{P3})$ where P_2 and P_3 are the bottlenecks, an additional P_2 will be limited by r_{P3} and an additional P_3 will be limited by r_{P2}.

The relationships $(r_{P1} \approx r_{P2}) \gg r_{P3}$, $r_{P1} \ll (r_{P2} \approx r_{P3})$, and $(r_{P1} \approx r_{P3}) \gg r_{P2}$ of column 2 in Table 4.1 are ideal for using resource sharing PR to gain a performance improvement. The relationships are ideal because the two processing operations with identical/similar rates are not the bottlenecks. This means that one or both of these process modules can be used to realize an additional bottleneck module. For example, in the $(r_{P1} \approx r_{P2}) \gg r_{P3}$ relationship, when the FIFO filled threshold for PR is reached (*i.e.*, FIFO_1 and FIFO_2 are almost full), the P_1 module can be reconfigured to realize an additional P_3 module. This approach can speed up the production of P_3 results without stalling the flow of data in the previous stages. Even though no new P_1 results are produced, the stage would have stalled when FIFO_1 and FIFO_2 reach capacity because of the production rate of P_3. The additional P_3 by way of resource sharing PR will prevent the stalls.

An alternate bottleneck mitigation approach to the $(r_{P1} \approx r_{P2}) \gg r_{P3}$ relationship is to realize two addition P_3 modules from the repurposed resources of P_1 and P_2 and then stagger the reconfiguration of the additional P_3 modules to reinstantiate P_2 first and then P_1. The two additional P_3 modules will increase the outflow of FIFO_2 and the stagger

technique allows for a continued performance improvement with one additional P_3 when P_2 is reinstantiated in response to the near-empty threshold of FIFO_2 being reached. The stagger technique allows for FIFO_1 to be emptied and P_2 results produced and processed without stalling the pipeline due to an empty FIFO_2. The P_1 process will be reinstantiated when FIFO_1 is close to empty.

The pipeline of Figure 4.3(b), where the relationship is $(r_{P1} \approx r_{P3}) \gg r_{P2}$, can also realize up to two additional P_2 modules to mitigate the bottleneck. The designer must choose, when reconfiguring only one additional P_2, which process module (P_1 or P_3) will be used. It may be ideal to repurpose P_1 to increase the inflow of FIFO_2 and have an affect on the outbound production rate of the pipeline. It may be more beneficial to repurpose P_3 to increase the inflow of FIFO_2, increasing the outflow of FIFO_1 as a result. This option can reduce the frequency P_1 will stall waiting for FIFO_1 to have space to receive more data. The benefit of either approach can be determined by observing the input and output rates of the pipeline. If faster input processing is desired, the designer can choose the resource sharing configuration that estimates to increase the data consumption rate of the pipeline. If a faster production rate is needed, the designer can choose a design that estimates to produce a higher outgoing data flow.

Figure 4.4 is an example of the $(r_{P1} \approx r_{P3}) \gg r_{P2}$ relationship (Figure 4.3(b)) if two additional P_2 modules where realized using resource sharing PR. The resources of P_1 are repurposed to realize P_2'', and the resources for P_3 become P_2'. All three P_2 modules execute concurrently. The additional P_2 modules increase the bottleneck processing rate to approximately $3 \times r_{P2}$.

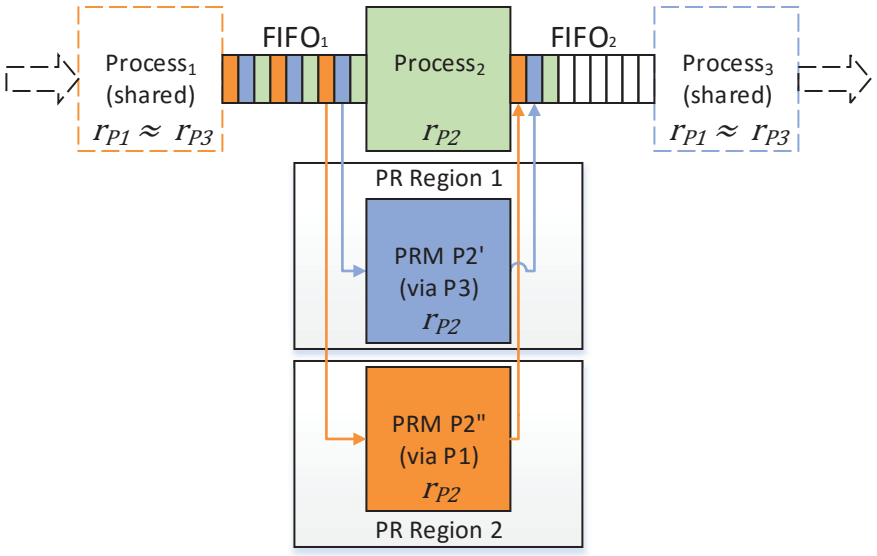

Figure 4.4

Possible resource sharing configuration of Figure 4.3(b)

The $(r_{P1} \approx r_{P3}) \ll r_{P2}$ relationship in column 2 in Table 4.1 (p.79) involves a similar resource sharing PR technique as shown in Figure 4.4. In this relationship both a bottleneck (*e.g.*, P_3) module and the non-bottleneck module P_2 can be used to improve performance. Two additional P_1 modules can be realized (using P_2 and P_3) to increase the inflow of FIFO_1. Additionally, P_1 can be repurposed into a P_3 to increase the outflow of FIFO_2. The use of both bottleneck and non-bottleneck modules to improve performance is the same type of technique that would be used for the relationships of column 3 in Table 4.1.

As discussed above, there are several options for resource sharing PR that can be used to mitigate the bottleneck of the three stage pipeline that realizes a column 2 relationship in Table 4.2. However, the two process modules that have identical/similar processing rates simplifies these options to using resource sharing PR between two pipeline stages

where the fastest process module is used to realize a bottleneck module similar to that of Figure 4.1 (p.76). An exception is the pipeline that realizes the relationship $(r_{P1} \approx r_{P3}) \ll r_{P2}$. In this relationship, the P_1 bottleneck module can be used to realize an additional P_3 bottleneck module or vice versa. This technique is a feasible option for improving the performance of the pipeline. Furthermore, the technique must be used for mitigating bottlenecks in column 3 pipeline relationships of Table 4.2.

The column 3 relationships represent pipelines of Figure 4.5. In these pipeline, the processing rates of the three process modules differ greatly from each other. Therefore, simply choosing the fastest process module to realize an additional bottleneck module (*i.e.*, the module with the lowest processing rate), may not always yield the best possible performance improvement. A better option may be to realize the middle-rate process module (*i.e.*, the module that is neither the fastest nor slowest). Another alternative would be to repurpose the bottleneck module to increase the production rate of the middle-rate process module. A third option is to use the fastest process module and middle-rate process module to realize two additional bottleneck modules. Depending on what stage in the pipeline the bottleneck is located (*e.g.*, stage 1) a performance improvement may be achieved.

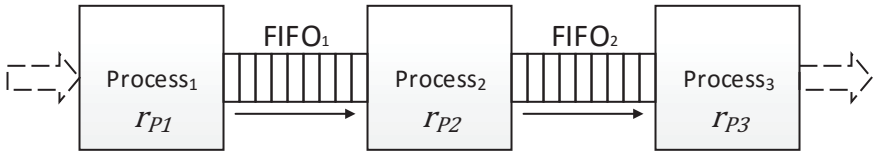

Figure 4.5

A general three process pipeline

Table 4.2

Processing rate relationships of a three stage pipeline

1	**2**	**3**
$r_{P1} \approx r_{P2} \approx r_{P3}$	$(r_{P1} \approx r_{P2}) \ll r_{P3}$	$r_{P1} \ll r_{P2} \ll r_{P3}$
	$r_{P1} \ll (r_{P2} \approx r_{P3})$	$r_{P1} \ll r_{P3} \ll r_{P2}$
	$(r_{P1} \approx r_{P2}) \gg r_{P3}$	$r_{P2} \ll r_{P3} \ll r_{P1}$
	$r_{P1} \gg (r_{P2} \approx r_{P3})$	$r_{P2} \ll r_{P1} \ll r_{P3}$
	$(r_{P1} \approx r_{P3}) \ll r_{P2}$	$r_{P3} \ll r_{P1} \ll r_{P2}$
	$(r_{P1} \approx r_{P3}) \gg r_{P2}$	$r_{P3} \ll r_{P2} \ll r_{P1}$

It is clear that the number of resource sharing PR options for a column 3 relationship in Table 4.2 is greater than the options presented by a relationship for column 2. This is due to a varying number of possible configurations that can yield a performance improvement. Additionally, determining which resource sharing PR configuration yields the best performance becomes an equally greater task when compared to the possible configurations of the column 2. Furthermore, as more stages are added to the pipeline, the possible rate relationships become more complex in addition to selecting modules for resource sharing PR. As the pipeline increases, designers will have to rely more heavily on estimating the FIFO fill rate and reconfiguration overheads to determine candidates for resource sharing PR. In addition, selecting the proper candidate will be done by trial and error and verified via application testing. Therefore, to reduce the frequency of realizing a full PR application for testing candidates, designers can use the bitstream length and reconfiguration formulae (developed by this research) to perform preliminary bitstream and reconfiguration overhead calculations to eliminate infeasible PR configurations.

4.2.2 Resource Sharing PR Architectural Model

Figure 4.6 shows the architectural model of a complete resource sharing PR application. The hardware application is comprised of a pipeline of computation modules, I/O modules, FIFOs, a demultiplexer (DMUX)-like selection module, and a controller module. Each module in the pipeline operates independently and performs operations to consume input data units (DUs) and produces output DUs for the next module in the pipeline. The modules may also operate at different clock frequencies with some modules capable of operating at higher clock speeds than others.

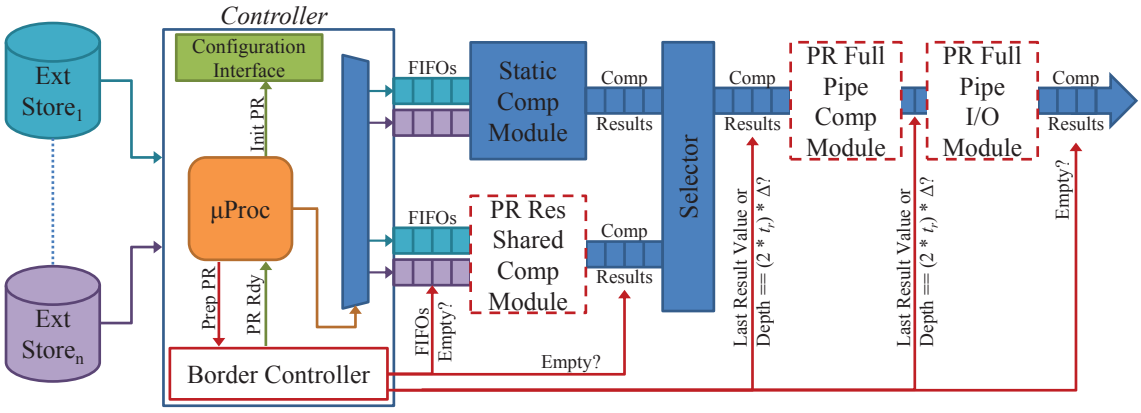

Figure 4.6

Resource-sharing partial-reconfigurable hardware application model

In this model, the I/O modules provide the controlling logic for the FPGA's I/O interface that is connected to an external I/O device. Input I/O modules not shown in this model are a part of the controller. The computation modules (labeled "Comp") process data provided by the I/O modules or other computation modules in the pipeline. The PRMs labeled

"PR Full Pipe" are modules that complete the full application pipeline and are uploaded as a part of the initial configuration of the resource sharing PR application. These modules are candidates for resource sharing PR; not shown in Figure 4.6 are other Full Pipe PRMs that come before the static module in the pipeline and are also candidates for resource sharing.

The selector is a DMUX that selects and forwards DUs stored in FIFOs filled by the computation modules in a previous pipeline stage. The selector implements any selection algorithm the hardware designer deems optimal for forwarding DUs in the application. With the full pipeline intact (no modules missing), the selector forwards DUs produced by the static computation module. However, when implementing resource sharing, where an additional computation module (labeled "Shared") is realized using the repurposed resources of an idle Full Pipe module, the selector implements the selection algorithm to forward DUs produced by the static and the additional compute module.

The static computation module is a fixed application module this will not be reconfigured using PR. This module is the bottleneck computation module whose computation time (*i.e.*, time to produce DUs) is significantly longer than all other modules in the pipeline; other static modules that have a minimal effect on performance are implied but are not presented in Figure 4.6. The "PR Res Shared" computation module is an additional bottleneck module implemented using the reallocated and repurposed resources of an idle Full Pipe PRM located on the outbound end of the selector. Clearly, once the resources of a Full Pipe PRM are repurposed, the application pipeline is broken. The controller determines when to reconfigure the additional bottleneck module back to the repurposed Full Pipe PRM.

The controller is a processor driven unit: (1) that initiates resource sharing PR by sending commands to the FPGA configuration interface, (2) that controls the the flow of data between the FPGA and external storage components (*e.g.*, Ext Store_1), (3) that monitors the execution status (idle or not) of computation and I/O modules, FIFO capacities, and determines the best time to initiate PR; and (4) that invokes the border controller, a user-defined hardware module that closes off input and output connections to PRMs that are to be reconfigured.

4.3 Mitigating Mismatched Processing

Some hardware application are designed with a fixed-sized array of PEs to perform computations. A PE array is a collection of logic-based processing elements either interconnected to realize a parallel computation pipeline, or are arranged as independent processors that operate upon a subset of the application input in parallel (*i.e.*, divide and conquer). The PE arrays are often designed with memories that store input data elements to be processed. The memories grant the PEs repeated access to the stored input data to perform its computation operations. The memory storage of the PE array can vary, but it is often sized to store a minimum number of data elements as the number of PEs in the array (*i.e.*, one data element for each PE).

Hardware applications that employ a fixed array of logic-based processing elements to perform computations will not always achieve a one-to-one input-to-PE pairing. A mismatch can occur when the application is processing a collection of input data sets where each input data set varies in size, that is, the number of contained data elements (*i.e.*, DUs).

The varying input data sets can result in an inefficient utilization of computing resources. For example, in a hardware application equipped with a large PE array for processing, some PEs will be idle when the received input data set contains fewer DUs than the number of PEs in the array (#DUs $<$ #PEs).

Hardware applications with large PE arrays can be redesigned to split the PE array into smaller PE arrays to process smaller sized input in parallel, increasing computation efficiency. However, when the application receives large input data sets, the smaller PE arrays may not be capable of storing the input due to their limited memory capacity. Random-access memory storage on the FPGA is limited and require BRAM and logic resources to realize. The limitation is a result of the fixed resources of the FPGA. Each PE array with memory realized, reduces the quantity of available resources for implementing additional PE arrays. Therefore, an application designed with several PE arrays often includes smaller memory blocks per array to allow for additional PE arrays to be configured.

An alternative shared memory structure can be designed for applications with multiple PE arrays, but the FPGA memories are dual ported where only two PE arrays can access the memory at a time. This approach is not optimal for applications that use more than two PE arrays. Therefore, a distributed memory design where each PE array has a dedicated memory is best suited for these multiple PE array application architectures.

To mitigate the effects of input-to-PE mismatches, an application can use PR to load a set of varying sized independent PE arrays as a single PRM. The effect is a PE array capable of processing data elements of different sizes (large and small) or an array capable

of processing both large and small data inputs concurrently. This PRM design technique has also been developed by this research and is called a “multi-module PRM”.

4.3.1 Multi-Module PRMs

The architecture of the application using a multi-module PRM looks similar to Figure 4.7. The figure presents a PRR with multiple PRMs. A multi-module PRM design will consist of a single PRM (*i.e.*, a wrapper PRM) encapsulating multiple internal PRMs. The case for implementing the multi-module PRM design approach is for applications that provide multi-function capabilities and swap PRMs that differ greatly in architecture and/or function, including allocated FPGA resources.

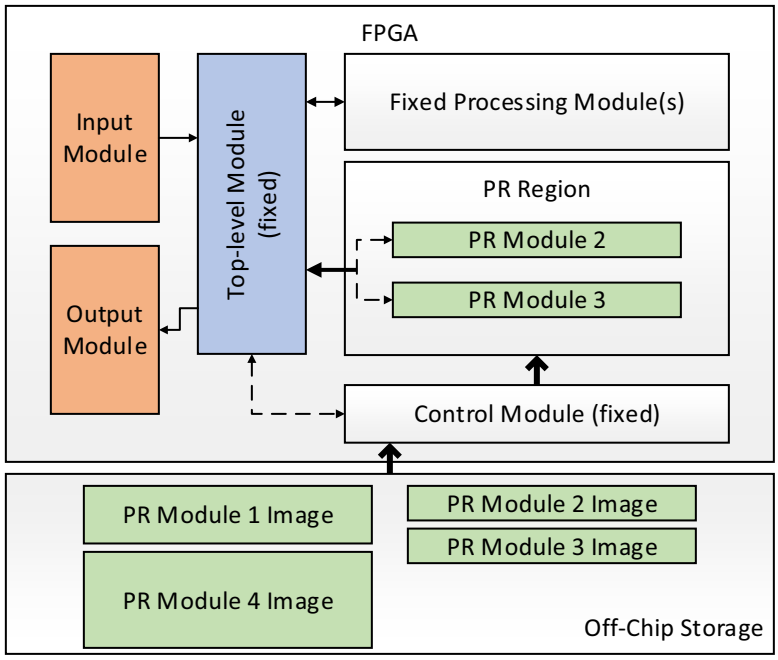

Figure 4.7

PR with dissimilar PRMs

The Xilinx basic PR restricts application designs to single module/single function PRMs to be loaded upon demand. Basic PR applications use PR to perform function changes by swapping single module PRMs. The function change is performed to process a different set of input data. However, in an environment where the input data sets presented to the application often vary (*e.g.*, in size). The overhead of performing PR to load the supporting functional PRM may nullify any benefits gained from a PR design.

An alternative is to load multiple PRMs into a single PRR as shown in Figure 4.7. However, the FPGA does not support loading PRMs into a PRR without clearing the configuration memory that implements PRMs previously loaded into the PRR. Therefore, during the PR process, all modules in the PRR are disabled and the configuration memory of the PRR resources is cleared. This means for any PRM to be loaded into the PRR, all modules must be loaded at once, including reloading any cleared modules that were intended to remain with the PRR.

The multi-module PRM design enables PRMs to provide simultaneous multi-function capabilities to be swapped for other multi-function PRMs on-the-fly. Using the application designed with a fixed-sized PE array for example, in Figure 4.7 module 2 can be a 4-PE array, module 3 a 6-PE array, module 1 a 12-PE array, and module 4 a 20-PE array. Using the multi-module PRM technique, any combination of these arrays can be represented as a single PRM and used as independent individual units, provided that FPGA resources are available. It will be possible to swap a 12-PE array for a PRM with two independent 6-PE arrays for processing small input sizes or four 4-PE arrays for smaller inputs. A combined 12-PE, 4-PE array PRM can be loaded to process both a large and small input data set in

parallel and, if necessary, the PRM can be exchanged for a 20-PE PRM to process larger input data sets.

Similar to frequently using PR to change application functionality, PR applications using multi-module PRMs will regularly reconfigure PRMs to meet the demands of varying inputs. However, PRMs designed strategically to meet observed patterns can reduce the frequency in which a PRM is reconfigured.

4.4 Bitstream Length Estimation

Chapter 3 states that the FPGA-based hardware application undergoes several phases before it is deployed onto an FPGA. These phases include the design, synthesis, map, and place-and-route phases. Of these phases, the place-and-route phase requires the longest time to complete. This is because, given the resources provided by the FPGA fabric, an unknown amount of distinct physical placements of the application's functional blocks, with varying performances, can be derived. Furthermore, because of the unknown number of placements, finding a placement that achieves an optimal performance above all possible placements within our lifetime is also unknown for any non-trivial design.

Place-and-route derives a set number of placements and chooses the global optima among them, failing if no placement is possible. The search for the optimal placement of elements in a given space if often called the "placement problem". The placement problem associated with place-and-route is commonly referred to as an NP-complete (in some cases NP-hard) problem that can take from several minutes to hours to complete [51, 77, 34, 12, 76, 28].

Bitstream estimation provides a means for determining a design's configuration bitstream length without having to complete the hardware implementation place-and-route process by using the resource estimate provided by the synthesis or map steps. For PR, the bitstream length estimation can aid in estimating the configuration time of PRMs and bypass the need to the complete the full PR implementation process.

In general, a the Xilinx Virtex configuration bitstream consists of handshaking and synchronization packets for the configuration interface, configuration header and data packets for the FPGA functional blocks (*e.g.*, CLB, BRAM, DSP), RAM memory configuration header and data packets (*i.e.*, memory function and contents), clocking, and interconnect routing packets. Field programmable gate array vendor Altera does not provide enough details relative to the construction of their bitstream to allow designers to calculate an approximate bitstream size of their hardware designs. However, as an alternative, Altera provides the uncompressed raw binary file size for their Stratix family FPGAs with the caveat that fully implemented designs may increase or decrease the binary file size depending on whether bitstream compression is used and/or may vary depending on the hardware design [1, 2, 3, 5]. The raw binary file is used to estimate the file size before compiling the design for hardware and is also used to estimate the minimum configuration time of the FPGA.

In addition to synchronization packets and other initializing overhead bits, the Xilinx Virtex family FPGA bitstream is broken down into two packet types: Type 1 packets and Type 2 packets [64, 66, 68, 54]. Type 1 packets are used for read and write transactions to the configuration registers of the FPGA. Configuration registers receive bitstream

commands that are used to instruct the configuration control logic to perform a specified configuration function upon the FPGA functional blocks.

Type 1 packets include a 32-bit Type 1 header (*i.e.*, one word) followed by a Type 1 data section with a maximum size of approximately 2^{11} words. Type 1 packets provide the bitstream commands that are written to (or read from) the dedicated FPGA configuration registers called "packet registers". Packet registers specify the type of configuration function to execute and the address location (row and major column) where this function is to be performed.

The Type 2 packet is used for writing long blocks (*e.g.*, BRAM contents) and includes its own header and data sections. The 32-bit Type 2 header specifies the type of operation (*e.g.*, NO-OP, read/write) and the number of 32-bit data words to be written (up to 2^{27} words), but it does not include the destination address. Therefore, the Type 2 packet must always follow a Type 1 packet because this packet provides the frame address location where the Type 2 data is to be loaded.

For writing configuration data to a Virtex FPGA (and also for bitstream size estimation), the packet registers of interest are the Frame Address Register (FAR), the Command register (CMD), the Frame Data Register Input register (FDRI), and the Cyclic Redundancy Check (CRC) register for data integrity checking [64, 66, 68, 54]. The FAR specifies the row and major column to configure as well as the specific frame within the major column to begin writing the configuration data. The CMD includes a set of commands that must be written prior to writing to the FDRI. The FDIR is the configuration data that is written, starting at the frame specified by the FAR. Each register requires a Type 1 packet with data

except for the FDRI. The FDRI requires Type 1 packet header and then a Type 2 packet following the Type 1 header. Using the above packet register and packet types, the estimated bitstream length of a major column is the following:

$$bits_{col} = T1_{FAR} + O_{CMD} + T1H_{FDRI} + T2_{FDRI} + T1_{CRC} \tag{4.1}$$

where $T1_{FAR}$ and $T1_{CRC}$ are Type 1 packets of a minimum size of approximately 64-bits. The overhead value, O_{CMD}, is the set of Type 1 command register packets required to be uploaded prior to uploading the FDRI packets. The value $T1H_{FDRI}$ is the single Type 1 header (32-bits) that is paired with the Type 2 packet, $T2_{FDRI}$. The following equation is a derivative of Equation 4.1 using the number of frames in a major column of a specific resource type:

$$bits_{col} = T1_{FAR} + O_{CMD} + T1H_{FDRI} + T2H_{FDRI} + (f_{res} \times bits_f) + T1_{CRC} \tag{4.2}$$

where $T2H_{FDRI}$ is the Type 2 packet header, f_{res} is the number of frames in the column relative to the resource type of the major column and, $bits_f$ is a constant that is the number of bits in a frame. Using Equation 4.2, the bitstream length of a hardware design is:

$$bitstream_{HW\,App} = O_{bitstream} + \sum_{i=1}^{k} (n_i \times bits_{col_i}) \tag{4.3}$$

The value $O_{bitstream}$ is a configuration overhead that exists in all configuration bitstreams including partial bitstreams. This overhead is a fixed value for all devices in the same Virtex FPGA series (*e.g.*, Virtex-5) [64, 66, 68]. In summation, the value k is the total number of different resources used by the design. The value n_i is the total number of major columns of resource i used by the design. Equation 4.3 assumes that the hardware design (or PRM)

is constrained to a fixed region of resources at major column borders. Equations 4.2 and 4.3 assume that the resources used by the hardware design are taken from the synthesis report generated by the Xilinx Synthesis Technology (XST) tool.

4.5 Configuration Time Estimation

The Xilinx PR design flow process requires the synthesis, map, and place-and-route of all PRMs before combining all static and PRMs into one complete PR application. The process produces a bitstream length estimate for each static and PRM. The bitstream lengths can be used to determine the configuration time of the modules using clock period or the data transfer rate (clock frequency times data width) of the selected FPGA configuration interface.

The configuration bitstream upload time can be estimated by using one of the following formulae:

$$t_{upload} = bitstream_{HWApp} \times \left(\frac{clock\ period \quad \text{ns}}{dataWidth \quad \text{bits/cycle}} \right) \tag{4.4}$$

using the clock period of the configuration interface, or

$$t_{upload} = \frac{bitstream_{HWApp} \quad \text{bits}}{(freq_{interface} \times dataWidth) \quad \text{bps}} \tag{4.5}$$

using the interface clock frequency.

The $dataWidth$ value is the maximum number of bits the configuration interface can transmit per clock cycle, $freq_{interface}$ is the clock frequency of the interface in Hertz and the expression ($freq_{interface} \times dataWidth$) is the data transfer rate of the configuration interface in bits per second (bps); the value of this expression in some cases has been called the configuration interface bandwidth.

In a self-reconfiguring autonomous PR application, PR is initiated by the application controller (see Figure 4.6). The controller monitors the state of the PR application and determines when PR should occur. The controller is also responsible for delivering the PRM bitstreams to the FPGA configuration interface by issuing I/O operations to the hardware that stores the bitstream and then routing the received bitstream to the configuration interface.

The ICAP configuration interface does not signal when it completes uploading the a bitstream nor when the configured logic (*i.e.*, the PRM) arrives at a ready state for execution. The application must include logic to detect the completion of the PR process and signal the controller when the PRM is ready. Equation 4.6 shows the complete configuration time estimate, t_r, for a PR region:

$$t_r = t_{read} + O_{proc} + t_{upload} + t_{detect} \text{ seconds} \tag{4.6}$$

The value t_{read} is the time (seconds) between the issuance of a read I/O operation and the receipt of a DU of a PRM bitstream from storage; in this instance a DU is the largest possible set of bits that can be transferred at once by the I/O interface. The remaining DUs of the bitstream are assumed to be streamed in immediately after the first DU is returned (*i.e.*, one DU per I/O interface clock cycle); otherwise t_{read} is multiplied by the total number of DUs in the bitstream. The value O_{proc} is the overhead for any internal processing time taken by the controller to route a DU to the configuration interface. This overhead may be counted as part of I/O read time, t_{read}, or considered so small to be valued

as negligible. The t_{detect} is the time between the bitstream upload completion and when the PRM is determined ready for execution.

4.6 Performance Estimations

The rate at which a module in the application consumes DUs can differ from the rate that same module produces DUs. For example, a module can consume five DUs per second and produce only one DU in the same second after processing the five DUs, meaning the module has five-to-one DUs per second consumer-producer ratio. Both the I/O and computation modules can have separate production and consumption rates. The I/O modules include read and write data transmission rates for the connected external I/O interfaces:

Input/Read module data rate

$$t_{IOread} = t_{read} + t_{IOprod} \quad \text{DUs/second} \tag{4.7}$$

Output/Write module data rate

$$t_{IOwrite} = t_{IOcons} + t_{write} \quad \text{DUs/second} \tag{4.8}$$

Time t_{read} in Equation 4.7 is the rate DUs are delivered to the I/O module during a read operation and is primarily dependent upon the external storage transmission rate and data width. The time t_{IOprod} is the time (DUs/second) the I/O module processes received data to prepare the DU for the next module (*e.g.*, a computation module) in the pipeline. The time t_{IOcons} in Equation 4.8 is the consumption and processing time needed to prep the DU (*e.g.*, packaging or framing) for write operations. Time t_{write} is the rate the I/O module can submit write operations to the external I/O interface; transmission rate and data width of

the external device may also affect this value. It should be noted that the times t_{read} and t_{write} are affected by the processing time of the I/O module. For example, if the processing time t_{IOprod} is slower than the DU delivery rate t_{read}, then the I/O read controller will have to throttle incoming data to match the rate of t_{IOprod}.

The consumption rate of a computation module is the number of clock cycles between retrievals of input DUs. For example, a computation module may retrieve a DU for processing every five clock cycles. The time it takes for the consumption of DUs is determined by the rate at which the DU is consumed and the speed at which the module operates. The production rate of a computation module is the number of computation cycles (or algorithm iterations) it takes to produce a single result DU; the production time is also relative to the operating speed of the computation module. Equation 4.9 is a general formula for computing the DU consumption and production rates of a computation module:

$$\begin{aligned} t_{CM} &= \frac{n \quad \text{cycles/DU}}{freq_{CM} \quad \text{Hz}} \\ &= m \quad \text{seconds/DU} \end{aligned} \tag{4.9}$$

The value, n, is the rate DUs are consumed or produced in cycles. The value t_{CM} is the time, in seconds, it takes to produce or consume DUs given the execution frequency ($freq_{CM}$) of the computation module.

The DU production rate of a module in the pipeline can differ from the consumption rate of the next module in the pipeline (*e.g.*, I/O module versus computation module).

Equation 4.10 is the delta between the two rates where t_{prod} is the production rate of a module and t_{cons} the consumption rate of the next module in the pipeline.

$$\Delta = t_{cons} - t_{prod} \quad \text{seconds/DU} \tag{4.10}$$

The condition for using PR to reconfigure idle resources is determined by the delta between the production and consumption rates of connected modules. With a positive delta from Equation 4.10, the production rate is the faster rate and therefore the producing module is a candidate for PR upon going idle after the connecting FIFO is filled. If the delta is negative the consumption rate is faster and the consuming module is a candidate for PR.

Which module is selected for reconfiguration is designer defined and realized through selection operations executed by the controller. The designer may chose to throttle input data and perform resource sharing PR to reconfigure frequently idle input I/O modules into compute modules increasing the production of outgoing result data. Alternatively, resource sharing PR can be used on idle output I/O modules to increase computation and avoid throttling incoming data. Furthermore, the designer may choose to reconfigure an idle computation module to speed up DU processing and the designer may also choose to reconfigure multiple modules (I/O and computation) to improve the overall DU computation rate. The performance gain of either approach can be observed during execution.

The reconfiguration process of a PRM can take several microseconds to complete, depending on the size of the partial bitstream. To amortize this cost, PR can be initiated during the execution of the application prior to the time the reconfigured PRM is needed. In a

resource sharing PR design, where idle modules are temporarily reconfigured to increase computational performance through parallelism, FIFOs are used to buffer enough DUs to avoid idling computation modules during PR. Equation 4.11 estimates the minimum work (in DUs) required to amortize the reconfiguration time of I/O modules.

$$FIFOdepth = (2 * t_r) * \Delta \tag{4.11}$$

In this equation, the time t_r, calculated from Equation 4.6 (p.97), is multiplied by two to account for the reconfiguration of an idle PRM into an additional critical section computation bottleneck module, and a second reconfiguration step to return the bottleneck module back to the originating PRM. Time t_r is assumed to be the greatest of the reconfiguration times between the swapped PRMs.

CHAPTER 5

EXPERIMENT DESIGN

The identification of similar subsequences between a pair of biological sequences (nucleic acid and protein sequences) is a common search problem in Bioinformatics. The X!Tandem [59] and Smith-Waterman [56] algorithms are known search algorithms commonly used for identifying similar nucleic acid (or protein) subsequences. For this research, non-PR hardware implementations of these algorithms have been realized. The algorithms with a brief background and description of their hardware solutions are presented in this chapter.

In this research, PR solutions of the algorithms—using resources sharing PR and/or the multi-module PRM technique—have been implemented and tested. In the sections below, background details on proteomics, protein identification, the algorithm for X!Tandem, and the non-PR hardware design are presented. The Smith-Waterman algorithm and non-PR hardware implementation follows. Lastly, the chapter concludes with an outline of the experiments that have been performed upon the sequence identification PR solutions.

5.1 Proteomics

Proteomics is the study of subsets of *proteins* present in different parts of an organism and how they change over time and in varying conditions [23]. Proteins are molecules

in the cell that are responsible for executing various cellular functions. A protein is a sequence of *amino acids* linked together in a chain. The amino acid consists of a central carbon atom with a connecting amino group, carboxyl group, and a distinguishing side chain [23]; there are 20 distinct naturally occurring amino acids identified by the structure of their side chain. An amino acid that is a part of a chain is called a *residue*. A chain of two or more amino acid residues is a *peptide*.

A peptide is the chain of two or more amino acids with a bond between the carboxyl group of one and the amino group of next consecutive amino acid. The end of the chain with an unbound carboxyl group is the C-terminus and the end with an unbound amino group is the N-terminus. The bond that connects the carboxyl group to the amino group is called a *peptide bond*. The sequence of the amino acids is a fundamental attribute of a protein and is used as a means to identify the protein [23]. Therefore, to classify an unknown protein, a set of peptide sequences from a known protein can be of an unknown protein. This process is called bottom-up *protein identification* [23]. Relative to the protein, a peptide is a small subset of the amino acid residues of a protein and in practice, the preferred peptide length for protein identification is about 6–20 amino acid residues.

5.2 Protein Identification

Protein identification is a method used to identify an unknown protein using peptide sequences from the proteins of known protein sequences. Protein identification can be done computationally or through laboratory experimentation and observation using physical samples; this research uses computational methods for conducting protein identifi-

cation. In general, protein identification is a two step procedure that uses *tandem mass spectrometry* in step one, and then peptide sequence matching using a database of known proteins as the final step.

5.2.1 Tandem Mass Spectrometry

Tandem mass spectrometry is a three step process that includes the use of two mass spectrometers in tandem to determine the mass and the sequence of amino acid residues in peptides constituting the protein [23]. Mass spectrometers are devices used to measure the mass of charged molecules. The tandem mass spectrometry, MS/MS, process begins with separating an unidentified protein sample into peptides. The separation is done by breaking the peptide bond at certain cleavage points using an enzyme called a *protease*. An example is trypsin, a protease that cleaves the peptide bond that follow arginine or lysine amino acid. The cleavage process is called *digestion* (trypsin digestion in this example). After digestion, the resulting peptide samples are then fed to the tandem mass spectrometers.

Upon the receipt of a peptide sample, the first mass spectrometer charges the molecules of the peptide through ionization—the addition or loss of protons—producing a peptide *ion*. During ionization the mass of the peptide molecules are increased (or decreased) by the number of protons added (or lost). The number of these protons is denoted by z. The mass of a charged molecule is indirectly determined by measuring the mass (m)-to-charge ratio, m/z [23]. The second mass spectrometer measures the m/z value of the ions in attempt to derive the sequence information of the sample peptide. The final result is a mass spectrum (*i.e.*, a data set) of the peptide ions' m/z values and the intensity of their charge.

Figure 5.1 is a visual of a returned mass spectrum with the m/z values of the ions along the x-axis and the ion intensities along the y-axis. It should be noted that the sample may include particles that are not a regular part of a peptide chain but are ionized during MS/MS. These additional particles are considered as noise and extra measures are taken to clean the noise or to verify that known peptides matched to these particles do not incorrectly identify the unknown protein.

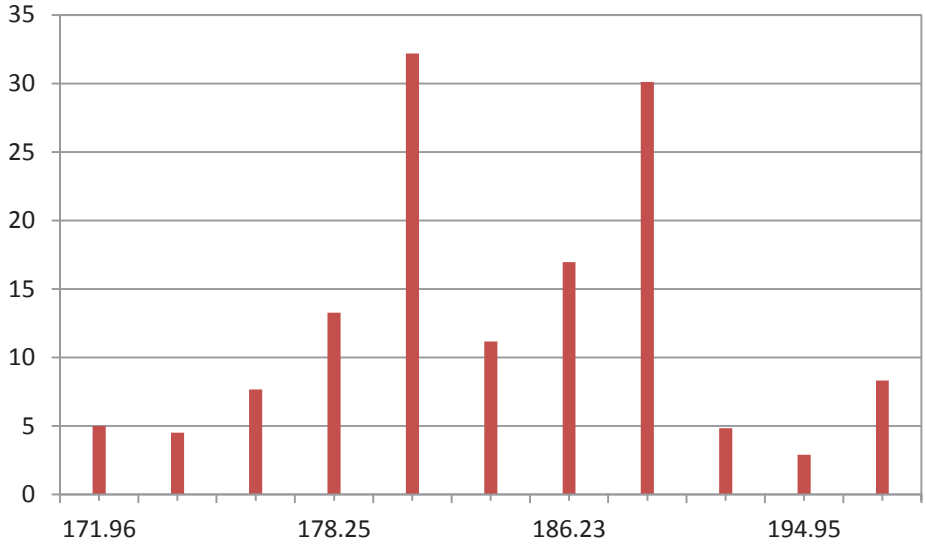

Figure 5.1

A mass spectrum of an unidentified protein

5.2.2 Peptide Sequence Matching

The matching step of protein identification is a search to find an identified peptide that has the most number of ions that match with ions of a peptide in the unidentified protein's peptide spectra (*i.e.*, the protein's set of ionized peptides). Identified proteins

undergo MS/MS *in silico* (*i.e.*, in silicon, computationally) by simulating digestion and the ionization of its cleaved peptides. There are six different types of peptide ions, a, b, c, x, y, and z. The m/z calculation of these ions is also performed computationally and differ per ion type. The formulae for determining the number of ions a peptide contains and for computing the m/z of these ions is provided in Eidhammer *et al.* [23].

The matching process is iterative and compares the m/z values of an identified peptide spectrum to the m/z values of the unidentified peptide's spectra; this matching is repeated for all identified peptide spectra. An ion match is confirmed if the m/z of an unidentified peptide ion is within range of the identified peptide ion's m/z. The match range is a window of surrounding m/z values centered on the identified peptide ion m/z. The surrounding m/z values differ by a small fractional value where upper and lower bounds of the m/z window area are set by a user-defined delta value (*e.g.*, a percentage of the known m/z) in units of *parts per million* (PPM). After iterating through the m/z values in a single identified peptide spectrum the match results are scored to illustrate the likelihood that the identified and unidentified spectra are similar. High scores suggest a greater probability that the unidentified peptide has been identified.

5.3 Software-based Protein Identification

The X!Tandem tool is an open source protein identification software [59] commonly used to perform peptide sequence matching. The software provides several user configurable options, which include specifying what ion types to match (*e.g.*, b, y), PPM, the charge for identified ions, and other user specified search parameters.

The protein identification process performed by X!Tandem can be generalized into the following three steps:

1. The computation of the identified-to-unidentified peptide match score
2. Rank each match score generated from matching identified and unidentified protein spectra resulting in a set of *hyperscores*
3. Probability calculations for verifying the likelihood that the highest hyperscore represents the identifying peptide sequence

From this point on, unidentified proteins, peptides, and ions will be referred to as *Observed*. Observed proteins undergo the MS/MS and are proteins currently being observed but the identity is unknown. Identified proteins, peptides, and ions will be referred to as *Hypothetical*. Hypothetical proteins undergo MS/MS *in silico* and come from a database of known proteins sequences. The Hypothetical peptide spectrum database is used to identify Observed proteins.

Equation 5.1 performs the match score calculation. Peptides are matched by traversing the ions of the paired Observed-Hypothetical peptide spectra and comparing them. A match occurs when an Observed peptide ion is within the PPM range of a Hypothetical peptide ion; the Hypothetical peptide is essentially mapped to the Observed peptide. The intensities of the matched ions are used to compute the match score.

$$MatchScore = \sum_{i=0}^{n-1} ObsIntensity_i \times HypIntensity_i \tag{5.1}$$

The value of n is the number of the m/z plots (*i.e.*, ions) in the Observed peptide spectrum. The intensities of the m/z values in the Observed spectrum are provided with the MS/MS generated spectrum. The intensity of the m/z values in the Hypothetical spectra are set to

one. All m/z values in the Observed spectrum that are not within PPM range of any Hypothetical m/z are represented with a intensity of zero (*i.e.*, skipped). Hence, Equation 5.1 is basically the sum of the intensities of matched ions in the Observed spectrum.

The match score rank (*i.e.*, the hyperscore) for each peptide in the Hypothetical protein is computed using Equation 5.2. The formula multiplies Equation 5.1 by the factorial of the number of each type of matched ion.

$$HyperScore = \left(\sum_{i=0}^{n-1} ObsIntensity_i \times HypIntensity_i \right) \times N_a! \times N_b! \times \cdots \times N_z! \quad (5.2)$$

Values $N_a!$, $N_b!$, and $N_z!$ are the factorial values of the total number ion types that have been matched to the Observed spectrum. For example, if 5 a ions were matched, then $N_a! = 5!$. The hyperscore represents how similar the Observed spectrum is to the current Hypothetical peptide spectrum under consideration.

X!Tandem generates a histogram of the calculated hyperscores for all matched Hypothetical peptides. The x-axis of the histogram is an integer $\log_{10}$ (or just $\log$) of the computed hyperscores to create "buckets" of similar hyperscores, $\log(HyperScore_i)$. The y-axis of the histogram is the count of similar hyperscores. The Hypothetical peptide(s) in the histogram with the highest hyperscore (*i.e.*, the bucket with the highest x-axis value) are assumed to be the potential protein identifying sequence that was found in the Observed spectrum.

5.4 Verifying Protein Identification Scores

The likelihood that the highest hyperscore is the identifying sequence of the Observed spectrum is verified using linear regression on a logarithm scale of the histogram. This

procedure is called *escoring*. Equation 5.3 computes the estimated regression line of the histogram [22]:

$$\begin{aligned} b_1 &= \frac{n\sum x_i y_i - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - (\sum x_i)^2} \\ b_0 &= \frac{\sum y_i - b_1 \sum x_i}{n} \\ y &= b_0 + b_1 x \end{aligned} \tag{5.3}$$

Prior to computing the regression line, the $\log$ is taken of hyperscore counts in each bucket of the histogram. The logarithm of these counts are the y_i values in Equation 5.3. The x_i values in Equation 5.3 are the individual bucket values, $\log(HyperScore_i)$, along the x-axis. The starting, x_0, point for the estimated regression line computation is the x-axis point with the highest y-axis peak. The value y in Equation 5.3 is the estimated regression line y-axis plot point at position x.

Figure 5.2 shows an estimated regression line (in red) plot of a hyperscore histogram [52]. The y points of interest (for verification) are the points to the right of where the estimated regression line intersects the x-axis at $y = 0$. These points are the x points that represent the highest value hyperscores that are assumed to identify the Observed spectrum.

The escore value of a y plot in the estimated regression line is the exponential function e^y, which is the probability that the scored Hypothetical peptide is the identifying sequence of the Observed spectrum.

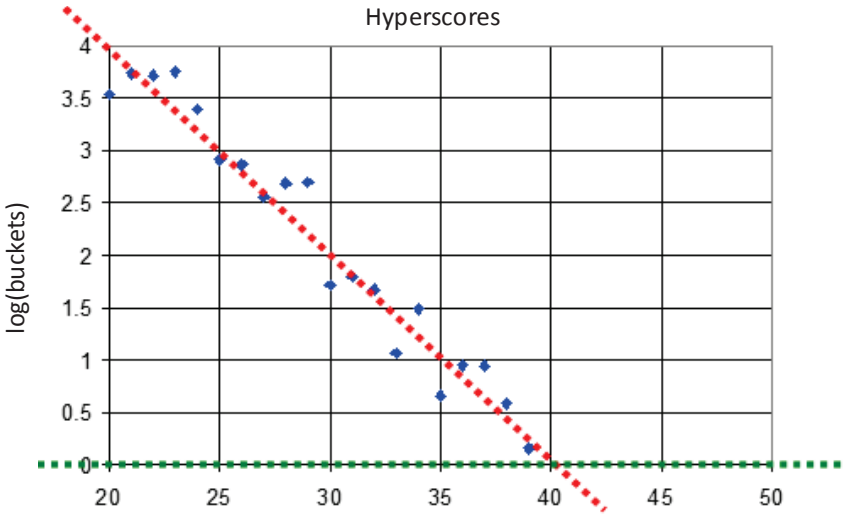

Figure 5.2

An estimated regression line of a hyperscore histogram [52]

5.5 FPGA-based Protein Identification

The hardware protein identification application designed for this research uses a multi-module pipeline that includes I/O modules, five computation modules, logarithm and factorial look-up tables, BRAM memory management modules, and an application *Controller* module for routing data. Dual ported FIFO structures are placed between pipeline stages to buffer data units for the next sequential module.

The flow of data traverses the five computation modules in the following order:

1. A *MinMax* module computes the Hypothetical peptide ion PPM range.
2. A *Match* module implements Equation 5.1 by executing the matching process and computing the $MatchScore$.
3. A *Hyperscore* module implements Equation 5.2 by computing the $HyperScore$ and the x-axis $\log$ values (*i.e.*, "buckets") for the *Histogram* module.
4. A *Histogram* module that generates a histogram of $HyperScore$ values by computing the $HyperScore$ count (*i.e.*, y-axis value) per bucket.

5. An *Escore* module implements Equation 5.3 by generating the escore for a known sequence found in the Observed spectrum.

The following subsections provide additional detail about the pipelined modules.

5.5.1 Data Representation

All m/z, intensities, and score values used by X!Tandem are Base 10 floating point values with up to 6 points of precision. The Virtex-5 FPGA does not have built in floating point hard cores and therefore, floating point operators must be implemented in logic. To simplify the hardware design, all m/z, scores, logs, and intensity values are represented as 36-bit fixed-point binary values with a 16-bit fractional part (*i.e.*, a 20 and 16-bit whole and fraction part respectively). Fixed-point arithmetic operations are integer operations, which is fully supported by the Virtex-5 FPGA.

5.5.2 The *MinMax* and *Match* Modules

The *MinMax* module computes the PPM window for each Hypothetical ion m/z. It receives a single Hypothetical ion m/z and computes the upper and lower-bound m/z values of the ion. This module implements a three stage pipeline that uses two parallel built-in DSPs for computing the upper and lower bounds. The three pipeline stages are needed because the DSP requires two cycles to produce a result. The DSP takes one cycle to register presented input data and a second cycle to perform its operation and register the result in the DPS output register. In the third *MinMax* pipeline stage the registered DSP output is forwarded to the FIFO of the *Match* module. At each clock cycle, the DSP is

presented a next m/z. Therefore, at the same time data is forwarded to *Match*, the next DSP result is registered, and the next input m/z is written to the DSP input register.

The *Match* module is a finite state machine (FSM) that performs the matching process and computes a match score using Equation 5.1. This module receives the PPM bounds from *MinMax* and Observed ion data from BRAM memory. The BRAM is initialized at FPGA startup.

The *Match* module uses Algorithm 5.1 (p.113) for ion matching and $MatchScore$ computations. The algorithm skips all Observed m/z that are less than the lower bound of the Hypothetical m/z window. A new Hypothetical m/z window is retrieved if the Observed m/z is greater than the window's upper boundary. Because the Hypothetical m/z received from *MinMax* is a range of similar m/z values, there is a chance for several Observed m/z values to fit within the Hypothetical m/z window. In the event of multiple matches to the same m/z window, *Match* chooses the matched Observed m/z with the greatest intensity value for scoring (Algorithm 5.1 lines 16-19).

The *Match* module uses four DSPs in parallel for the $MatchScore$ accumulation and for executing the upper bound (line 7), lower bound (line 13), and max intensity (line 16) relational comparisons. The *Match* module stops searching for matches when it has attempted to match the last m/z of the Observed spectrum or has iterated through all Hypothetical ions. At this point, the $MatchScore$ is inserted into the FIFO for the *Hyperscore* module and scoring begins for the next Hypothetical peptide spectrum.

In addition to computing the $MatchScore$, the *Match* module tracks the count of the different ion types in the Hypothetical peptide spectrum that have been matched with the

Algorithm 5.1 An m/z match, search and scoring algorithm

repeat
  let obs = first Observed spectrum ion data
  let hyp = first Hypothetical m/z window of a peptide
  let $max = 0, MatchScore = 0$

  while (!$obs.end$ **AND** !$hyp.end$) **do**
    if ($obs.mz > hyp.upperBound$) **then**
      // retrieve new Hypothetical windows until $obs.mz < hyp.upperBound$
      $hyp = hyp.next$
      // add the max intensity found from the previous cycles
      $MatchScore = MatchScore + max$
      $max = 0$
    else if ($obs.mz > hyp.lowerBound$) **then**
      // $obs.mz$ is within the m/z window; a match is found
      increment corresponding ion count
      if ($obs.intensity > max$) **then**
        // store the maximum matched intensity value
        $max = obs.intensity$
      end if
      $obs = obs.next$
    else
      $obs = obs.next$
    end if
  end while
  if (!$hyp.end$) **then**
    remove remaining Hypothetical m/z windows of current peptide
    $MatchScore = MatchScore + max$
  end if
  insert $MatchScore$ into FIFO
until (all Hypothetical spectra have been scored)

Observed spectrum. These individual ion counts are incremented every time a match is found for the ion type(s) of the Hypothetical m/z. The ion counts are also forwarded to the *Hyperscore* module along with the $MatchScore$.

5.5.3 The *Hyperscore* and *Histogram* Modules

The *Hyperscore* module realizes Equation 5.2. The module also takes the $\log_2$ (or $\lg$) of the computed hyperscores to generate x-axis values for the *Histogram* module. Equation 5.2 is a computationally intensive function because of the potential number of multiplication operations, which includes those that will be required to compute the factorial of each ion count. However, since the logarithm of the hyperscore will be taken to generate the x-axis values of the histogram, some multiplication operations can be removed from Equation 5.2 by using the following simplification:

$$\begin{aligned} \lg(HyperScore) &= \lg(MatchScore \times N_a! \times N_b! \times \cdots \times N_z!) \\ &= \lg(MatchScore) + \lg(N_a!) + \lg(N_b!) + \cdots + \lg(N_z!) \quad (5.4) \end{aligned}$$

The *Hyperscore* module uses two BRAM-based look-up tables (LUTs) to produce the $\lg$ values in Equation 5.4. One LUT is used to look up the $\lg(MatchScore)$ value and the other—a factorial logarithm LUT—is used to produce the $\lg$ of the ion counts. The factorial logarithm LUT returns the $\lg$ value of factorial of its input value. For example, the value returned by the factorial LUT for an input value n is $\lg(n!)$. This factorial $\lg$ look-up table eliminates the multiplication operations used to compute the factorial since the factorial values are computed prior to hardware configuration. Both logarithm LUTs are populated when the FPGA is initialized with the application bitstream.

The *Histogram* module sets the y-axis values for the histogram of $HyperScore$ values. *Histogram* receives x-axis values from the *Hyperscore* module and increments the y-axis bucket value at the corresponding x-axis location. The $HyperScore$ histogram is a BRAM addressed by x-axis values. A y-axis value update is a two cycle process, one cycle to submit the x-axis address for a memory read operation and the other cycle to increment and write the new y-axis value back to BRAM. The *Histogram* module also initiates escoring by signaling the *Escore* module once all Hypothetical spectra have been parsed for matching via *Match*, and the match results scored by the *Hyperscore* module.

5.5.4 The *Escore* Module

The *Escore* module implements Equation 5.3. The escoring process is executed in four passes of the histogram. The first pass takes the general $\lg$ of y-axis values. The second and third passes compute b_1 and b_0 respectively. The fourth pass computes the y plots of the estimated regression line. The multiply accumulate (MAC) and some summation operations in Equation 5.3 are computed using DSPs. The division operations are implemented in logic. The MAC, summation, and division operation results are registered for future usage by the next sequential pass. For example, results $\sum x_i$, $\sum y_i$, and b_1 are registered for computing b_0 in the next pass.

During the first pass of the escoring process, the summations, $\sum x_i$ and $\sum y_i$ are computed for each x_i and $\lg(y_i)$ using the FPGA built-in DSPs. The products $(\sum x_i)(\sum y_i)$ and $(\sum x_i)^2$ are also computed using a DSP. The final values of these summations and the products are registered for use in the next pass. During the second pass, *Escore* uses

DSPs to compute the MAC operations $\sum x_i y_i$ and $\sum x^2$. The corresponding multiplications of these MAC operations with n are computed after the completion of each MAC. *Escore* then computes b_1 using the DSPs for subtraction and then the logic-based division operation. The result b_1 is then stored for use in pass three. The third pass uses DSPs for the multiplication and subtraction and the logic based division for computing b_0, which is registered for the next pass. The fourth pass uses the operands b_1, b_0, and DSPs for the multiplication and addition operations to compute y.

5.5.5 Resource Sharing PR Prototype

Figure 5.3 shows the a resource sharing Tandem protein identification application prototype. The prototype is a pipeline of multi-module PRMs labeled as "Match Wrap" and "Escore Wrap". The Match Wrap PRM consists of the *MinMax*, *Match*, and *Hyperscore* modules. The *Match* module requires the most computation time and therefore the Match Wrap PRM is the application bottleneck. The *Histogram* and *Escore* modules make up the Escore Wrap PRM and it is the module whose resources will be repurposed to realize an additional Match Wrap PRM.

The *Controller* module controls the flow of data into and out of the application, distributes data units to computation module pipelines, monitors the module and FIFO status for PR, and prepares the application (*e.g.*, data flow throttling) for and initiates partial reconfiguration.

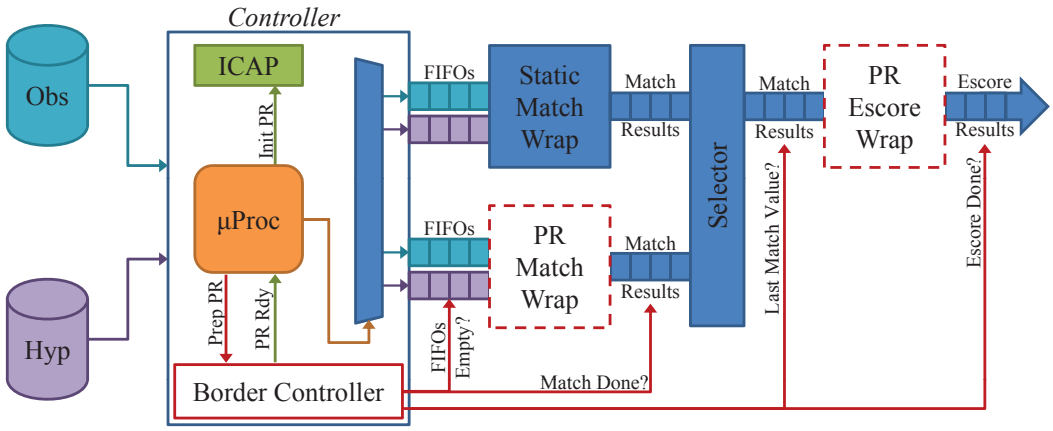

Figure 5.3

A resource sharing PR Tandem prototype

5.6 The Smith-Waterman Algorithm

The Smith-Waterman algorithm is a dynamic programming recursive algorithm designed for identifying subsequences [56]. The algorithm attempts to find a maximally homologous pair of subsequences, between two distinct biological sequences, where no other segment pair share a similarity greater than the pair found by the algorithm (*i.e.*, a homologous pair) [56]. In other words, the algorithm attempts to find the longest alignment of characters in the compared sequences that are either identical or contain few mismatches and/or character insertions/deletions. The identified homologous pair is said to have maximum similarity over all possible subsequence pairs.

To find the homologous pair, the Smith-Waterman algorithm conducts a similarity search between two biological sequences, S and T. It conducts this search by traversing S and T and comparing the s_i and t_j characters of these sequences, where $1 \leq i \leq n = |S|$ and $1 \leq j \leq m = |T|$. The similarity between s_i and t_j is a similarity weight value

extracted from a look-up-table of character pairs computed *a priori*. The algorithm uses the s_i, t_j similarity to compute a maximum similarity value, $V_{i,j}$. This $V_{i,j}$ is the maximum similarity of subsequences of S and T ending in s_i and t_j, respectively. The algorithm to compute $V_{i,j}$ is as follows [7]:

$$\text{Initialization:} \quad \begin{cases} V_{i,0} = E_{i,0} = 0, 0 \leq i \leq n \\ V_{0,j} = F_{0,j} = 0, 0 \leq j \leq m \end{cases} \tag{5.5}$$

$$\text{Recursion:} \quad V_{i,j} = \max \begin{cases} V_{i-1,j-1} + \sigma(s_i, t_i) \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} , 1 \leq i \leq n \text{ and } 1 \leq j \leq m \tag{5.6}$$

$$E_{i,j} = \max \begin{cases} V_{i,j-1} - \alpha \\ E_{i,j-1} - \beta \end{cases} , 1 \leq i \leq n \text{ and } 1 \leq j \leq m \tag{5.7}$$

$$F_{i,j} = \max \begin{cases} V_{i-1,j} - \alpha \\ F_{i-1,j} - \beta \end{cases} , 1 \leq i \leq n \text{ and } 1 \leq j \leq m \tag{5.8}$$

O. Gotoh developed this algorithm as an extension of the Smith-Waterman algorithm [7, 31]. Gotoh's extension employs an affine gap model to compensate for insertions and deletions. Insertions and deletions identify a mutational event that results in the addition or removal of one or more residues (*i.e.*, characters) in a biological sequence, typically called gaps [31]. The event misaligns the biological sequences, that is, the compared sequences are not of the same length. The affine gap model inserts additional gaps into to sequences to

realign them. The model compensates for these gaps by applying a penalty in the similarity calculations, one for the first inserted gap (the gap open), and another for any extending gaps (gap extension). In Equations 5.7 and 5.8 constants α and β are the gap open and gap extension penalties (respectively) assigned by the affine model.

The result of the similarity search performed by Equation 5.6 is an $n \times m$ matrix of maximum similarity values, where each computed $V_{i,j}$ value is an element in the matrix. Values $E_{i,j}$ and $F_{i,j}$ in the equation are the maxima computed from either opening a new gap or extending a current one [7]. The value σ is the similarity weight, between characters s_i and t_j, retrieved via look-up.

The calculation of $V_{i,j}$ uses five input values and $V_{i,j}$, $E_{i,j}$, and $F_{i,j}$ are all derived from previously calculated V, E, and F values. Figure 5.4 shows an example of a similarity matrix. The matrix element labeled 1 receives data input from the above, the upper left diagonal, and from the left. Those inputs are used to compute the first V value ($V_{0,0}$); the zeros are used as inputs to elements along the upper and left matrix borders. All computations of $V_{i,j}$ receive input from these three locations. For example, the bottom-most element 7 in Figure 5.4 receives a V, E, and F from the upper and left elements (labeled 6) and the left diagonal (labeled 5).

The line perpendicular to the diagonal of the matrix (*i.e.*, the anti-diagonal), in Figure 5.4, show the elements whose $V_{i,j}$ can be computed concurrently. All elements in the matrix with the same non-zero label can be computed in parallel.

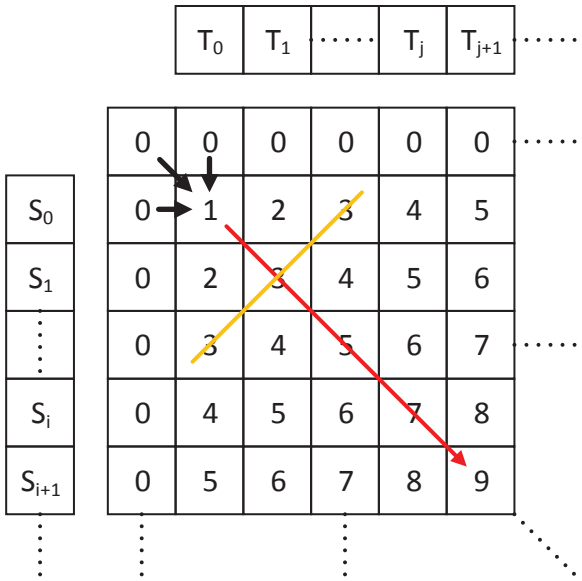

Figure 5.4

Smith-Waterman similarity matrix

5.7 FPGA-Based Smith-Waterman Design

An FPGA-based Smith-Waterman hardware design has been developed for this research by Dr. Y. Dandass. The device architecture includes FIFOs (for input, output, intermediate storage), an array of processing elements (performs similarity calculations), shift register, and control hardware (*i.e.*, controller). The input FIFO stores characters of both input biological sequences S and T, where the characters of S are first read from the input FIFO.

The processing element (PE) array in the design is mapped across the columns of the matrix, one PE per column. Each PE performs similarity calculations of its column of the matrix. Similarity calculations traverses the PE array (*i.e.*, the matrix columns) from left to right. The leftmost PE receives input data from the intermediate FIFO, performs its

similarity calculations, and passes its results to the PE on the right. This PE is the only PE that receives data from the intermediate FIFO. The rightmost PE (*i.e.*, the last PE in the array) inserts its calculated results into the intermediate FIFO. All middle PEs perform their calculations and pass the results to the PE on the right.

To perform the similarity calculation outlined in the Equation 5.6 a PE needs five data items, S and T characters and V, E, and F values. Figure 5.5 is an example of a PE. The PE uses several registers to store data needed by the PE and to store data to be given to the next PE (*i.e.*, the PE on the right). The values $V_{i,j-1}$, $E_{i,j-1}$, S_i character, and a start-of-frame/end-of-frame (SoF/EoF) marker are received from the PE on the left. The received V, E, and S values are used to compute the $V_{i,j}$ and $E_{i,j}$ values.

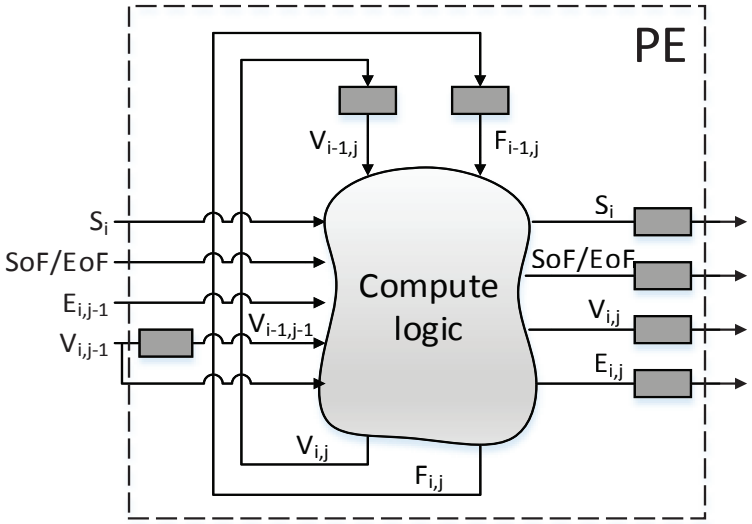

Figure 5.5

Model of a PE

The computed $V_{i,j}$ and $E_{i,j}$ (along with S_i and SoF/EoF) are written to registers to be used by the PE on the right; $V_{i,j}$ and $E_{i,j}$ becomes the $V_{i,j-1}$ and $E_{i,j-1}$ for the right PE. The registered $V_{i,j}$ value is also used by the PE as its $V_{i-1,j}$ value when executing the similarity calculation upon the next S character, S_{i+1}. The value $V_{i-1,j}$ is used for calculating the Fi,j value that, like $V_{i,j}$, is also registered for use as the above $F_{i-1,j}$ value for similarity calculations of S_{i+1}.

The $V_{i,j-1}$ value received from the left is stored in a register and also fed directly to the PE. The registered $V_{i,j-1}$ is used as the left diagonal value, $V_{i-1,j-1}$, for the similarity calculation of S_{i+1}. Using $V_{i,j-1}$ as $V_{i-1,j-1}$ is possible because a PE performs the computations of a column in the similarity matrix. For example, in Figure 5.4, the $V_{i,j-1}$ of a PE performing computations on an S_i during clock cycle m would clearly become the $V_{i-1,j-1}$ value for the PE performing computations on S_{i+1} in clock cycle $m+1$. The same argument can be made for values $V_{i,j}$ and $F_{i,j}$ for S_i becoming the $V_{i-1,j}$ and $F_{i-1,j}$ for S_{i+1}.

The T character, T_j, needed for the computations is read from a memory set by the controller prior to the PE receiving the SoF/EoF marker. The read T_j is registered by the PE and used for all S characters until the PE receives a set SoF marker, where it would read the next T_j character, T_{2j}. A set SoF marker initiates the beginning of a column calculation. Once received, the PE clears all registers to a zero value and reads in its values from the left and registers the T_j (T_{2j}, T_{3j}, ...) for the column. The receipt of a SoF is the only time the PE clears its registers; all values in the registers thereafter are the results of its computations or values received from the left.

Figure 5.6 shows the architectural design of the Smith-Waterman processing engine. The input FIFO stores all S and T characters with the S characters being first in the order. The output FIFO stores the final result of a completed similarity calculation. The PEs are arranged as an array of interconnected elements. The shift register is used to store the T characters to be given to the PEs. The depth of the shift register is equal to the number of PEs in the array. This memory is filled by the controller.

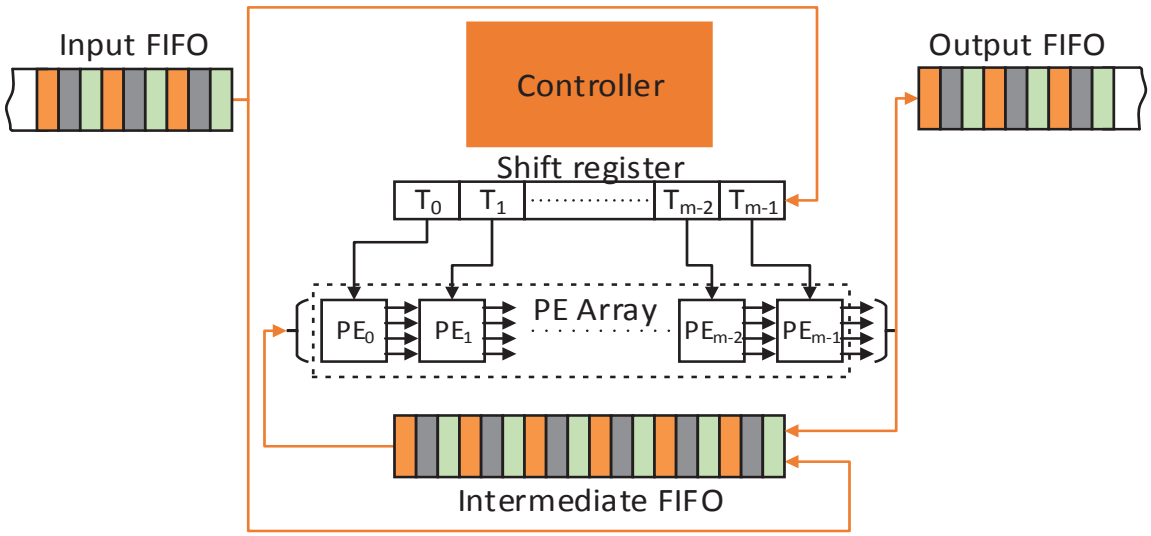

Figure 5.6

Smith-Waterman hardware architecture

The intermediate FIFO stores four-tuples of data to be processed. The four-tuple (or tuple) consists of a character of S, the SoF/EoF marker, a V value and an E value. These tuples are the left input values passed to the leftmost PE of the PE array. Before similarity calculations begin, the controller loads the intermediate FIFO with all S characters stored in the input FIFO. When reading the S characters from the input FIFO, the controller

creates a tuple for each S with initial V and E values set to zero and places them into the intermediate FIFO. The first tuple placed in the FIFO contains the first S character, S_0, and has the SoF value of the SoF/EoF marker set. The last tuple placed into the FIFO contains the S_{n-1} character and has the EoF value of the SoF/EoF marker set. Tuples between the start and ending tuple contain their subsequent S_i character and the SoF/EoF marker unset.

The flow of data begins with the initial loading of S characters by the controller. The controller then fills the shift register with T characters read from the input FIFO. This process takes m clock cycles where m is the number of PEs in the array. Next, the first tuple (the SoF tuple) is read into the leftmost PE. The PE clears its registers, loads in the S_0, $V_{i,j-1}$, $E_{i,j-1}$, and SoF from the tuple and registers the T_j character it reads from the shift register. Once computation is done, the PE registers the S_0, SoF, and computed $V_{i,j}$ and $E_{i,j}$ values for the PE on the right. At this time the leftmost PE reads in the next tuple with S_1 and the next $V_{i,j-1}$ and $E_{i,j-1}$ values. This process continues for all S_i tuples until EoF is encountered, where the PE waits until the controller signals it to begin reading from the intermediate FIFO again.

All PEs in the array perform computations concurrently. However, the data that the PE computes is not valid until it receives the SoF marker from the left. Once received, all computed data is valid until the PE finishes computing the i^{th} data with the EoF marker set. When the rightmost PE in the array receives the SoF marker, the controller recognizes the receipt and begins shifting the next $m \times T$ characters into the shift register. The controller also places the SoF, S_0 and the rightmost PE's computed $V_{i,j}$ and $E_{i,j}$ values as a tuple into the intermediate FIFO for the left most PE to use for similarity computation with

its new T_j character, T_{2j}. Once all $m \times T$ values are shifted into the shift register, the controller signals the leftmost PE to begin processing, where it reads the SoF tuple from the intermediate FIFO. This data flow continues until all T characters are read from the input FIFO and processed. Once completed, the computation results are placed by the controller into the output FIFO.

5.7.1 Smith-Waterman PR Application Prototype

Figure 5.6 shows a single Smith-Waterman similarity matching engine. The non-PR Smith-Waterman hardware application designed by Dr. Y. Dandass instantiates several identical, independent engines. Each independent engine has an n-PE array (*e.g.*, 24-PE array), controller, shift register, and intermediate, input, and output FIFOs. The number of Smith-Waterman engines that can be instantiated is limited by the available FPGA logic resources *and* the routing resources connecting all application modules and Smith-Waterman engines.

Before similarity matching begins, the entire S sequence is loaded into the intermediate FIFO. Therefore, the maximum size S character sequence that can be processed by a single Smith-Waterman engine is limited by the capacity of the intermediate FIFO. To process larger S sequences, multiple Smith-Waterman engines can be concatenated to realize a larger PE array and internal storage (*i.e.*, intermediate FIFO). However, constructing the ability to concatenate Smith-Waterman engines in a non-PR implementation requires additional routing and logic resources to connect the engines. The additional routing and logic requires more complex circuits that may reduce the performance of the application; the cir-

cuits' propagation delay can force the application to operate at a slower clock speed to meet timing constraints. Alternatively, the application can use PR to swap Smith-Waterman engines for a larger engine (*i.e.*, the equivalent of concatenating engines without the extra logic/routing) or multiple large engines.

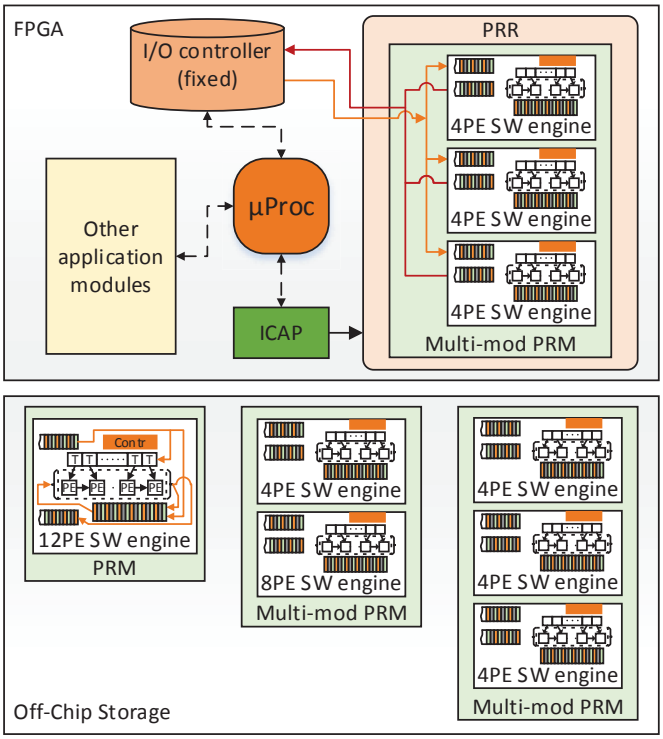

Figure 5.7

Smith-Waterman PR application prototype

Figure 5.7 shows a Smith-Waterman PR prototype application. The design is similar to the non-PR design in Figure 5.6 with the exception that a multi-module PRM is used to provide a set of independent Smith-Waterman engines. In this prototype example, the application assumes input S sequences that can fit in the intermediate FIFO of a 4-PE Smith-

Waterman engine. Furthermore, the prototype assumes larger inputs (*i.e.*, S sequences) that fit within 4-PE boundaries (*i.e.*, can fit in the intermediate FIFOs of Smith-Waterman engines that contain $4n$ PEs).

Like the non-PR design, the multi-module PRM can provide several independent $4n$-PE Smith-Waterman engines provided there are enough FPGA logic and routing resources to support each engine. In Figure 5.7, the application uses a multi-module PRM that instantiates three 4-PE Smith-Waterman engines. Each engine in the PRM will receive its own T and S character sequences to store in its shift register and intermediate FIFO, respectively. The microprocessor will control the data flow of the application and PR. The microprocessor will interface with the I/O controller to retrieve input data and export the processed results. It will also interface with the ICAP configuration interface to initiate PR. The micro processor will also handle how the T and S character sequences are handed to the $4n$-PE Smith-Waterman engines within the multi-module PRM. Lastly, the micro processor will monitor the S input size and determine which multi-module PRM to load via PR to support the incoming input. The multi-module PRMs with alternative $4n$-PE array instantiations are retrieved from an external storage (*e.g.*, flash memory or DDR RAM).

It should be noted that, although the Smith-Waterman engines' PE array sizes in the prototype are multiples of four, the Smith-Waterman engine options are not restricted to these boundaries. For example, it is possible for a multi-module PRM to instantiate a 4-PE and a 6-PE Smith-Waterman engine as long as each engine has been designed and implemented. It should also be noted that the prototype shows a single PR region (PRR). However, a complete Smith-Waterman PR application will contain multiple PRRs that can

house a PRM with a single Smith-Waterman engine or a multi-module PRM with several Smith-Waterman engines.

5.8 The Tandem Smith-Waterman Hardware Overview

The hardware is a two-stage pipeline similar to Figure 5.8. Process_1 is a static Tandem module, called the Tandem engine (Teng), that implements a protein identification procedure modeled after the X!Tandem algorithm. The protein identification process uses two databases, a database of unidentified/unknown spectra (Observed spectra) and a database of known spectra (Hypothetical spectra). Both databases are stored in read-only memory (ROM) in the FPGA fabric. Teng attempts to identify a peptide spectrum from a biological mixture of unknown Observed proteins by comparing them to known Hypothetical spectra. The details of the identification process, including the matching algorithm, was described in Sections 5.4 and 5.5. The architecture of Teng is discussed in Section 5.8.2 below.

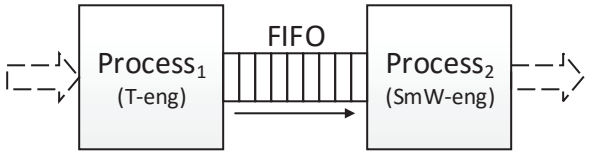

Figure 5.8

Two-stage pipeline

Process_2 is a Smith-Waterman module, called the Smith-Waterman engine (SmWeng), used to find similarities (in function) between the peptide of a protein identified by Teng, and the peptides derived from a database (the SmWeng database) of other known proteins.

The Smith-Waterman engine is the PRM of the pipeline. The Smith-Waterman engine is a non-PR completed stand-alone processing engine that was designed, implemented, and tested in a research effort prior to this dissertation research. The processing component of SmWeng is a pipeline of processing elements (PEs) design to received streamed input data (*i.e.*, one data element per clock cycle) at a speed of 100MHz (up to to 250MHz). The pipeline begins finding similarities immediately upon the receipt of the first data element. The search continues until all streamed data has been received and all PEs in the pipeline have finished. The algorithm and design details of SmWeng are provided in Chapter 5.7 and the supporting hardware designed to incorporate SmWeng into the pipeline (as a PRM) is discussed in Section 5.8.4.

The Smith-Waterman engine is the PRM of the pipeline primarily because it is the faster of the two pipeline stages. The Teng must first compare a single Observed peptide, using the matching algorithm, to each peptide in the Hypothetical database. Once the Hypothetical database has been searched, Teng scores its results and produces the peptide with the highest score rating as output. This output is the input to SmWeng and is used to retrieve and compare to a set of known peptides from SmWeng's peptide database. Since all required input is streamed in from the database and simultaneously processed, the majority of the Smith-Waterman process is complete when the final streamed input data is received. In addition, the characteristics of the identified peptide found by Teng are used to retrieve peptides with similar characteristics from the SmWeng database as input. Using the characteristic information reduces the known peptides streamed to SmWeng and, in effect, gives it a smaller space to search for similarities.

The FIFO between Teng and SmWeng, called the Teng result FIFO, buffers the Teng output when SmWeng is unable to retrieve the data; FIFO memories are commonly used as buffers between asynchronous producers and consumers. Since SmWeng is the PRM in the pipeline, when partial reconfiguration is initiated, the FPGA resources used by SmWeng will be reallocated and repurposed as a part of an on-the-fly instantiation of a second parallel Teng (see Figure 5.9). The end result is an incomplete pipeline because of the missing SmWeng. Therefore, the primary use of the FIFO is to buffer the values of the parallel Tandem engines until it is time to reconfigure the Teng PRM back into the SmWeng PRM. When the pipeline is complete, the FIFO buffers Teng output to minimize pipeline stalls.

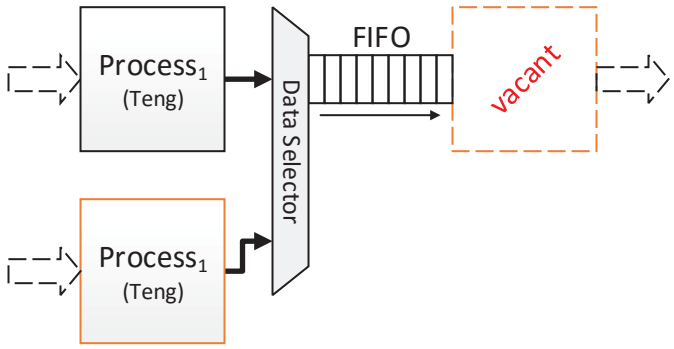

Figure 5.9

Pipeline after the PR of SmWeng

The hardware includes several static controller modules that aid in controlling the flow of data and assisting in partial reconfiguration. These controllers include three memory controllers that act as the interfaces between Teng and the Hypothetical and Observed databases (one each) and between SmWeng and its respective database; a "border con-

troller" used to monitor and control the connections to the I/O ports of the PRMs; and a PR controller that initiates preparations for the reconfiguration of a PRM and initiates the execution of PR. The controllers are discussed below in Sections 5.8.1 and 5.8.2

5.8.1 Input Data Memory Controllers

Input database retrieval for Teng and SmWeng is managed by a memory controller that acts as the interface to the ROM databases (one controller for each ROM). The memory controller is a state machine that implements the general memory read protocol, *i.e.*, send read request with address, wait for the assertion of a read acknowledgment signal along with the presentation of the read data; the read acknowledge signal acts as a "data ready" signal. The state machine was designed to interface with any storage structure that uses addressing for data storage and retrieval with minimal compatibility design modifications. The controller includes a read enable signal for Teng and SmWeng to control the frequency of read operations, and an address port that allows the PRMs to specify the memory read location.

The transmit state of the state machine supports single data element read operations, *i.e.*, a one clock cycle assertion and then deassertion of the read enable signal. The state also supports a continuous read operation (*i.e.*, streamed one element per cycle) if the read enable is asserted and held asserted for multiple cycles until deasserted. The memory controller has its own address counter and upon the assertion of the read enable, performs a memory prefetch read request for the next contiguous memory address (at single element

boundaries). The prefetch enables the streamed read capability of the state machine and also reduces the wait delay for single read requests.

The SmWeng database memory controller assumes a continuous read operation by default and therefore all input is streamed until SmWeng asserts a signal to throttle (*i.e.*, pause/stall) the read and presentation of input data. The Teng match process performs single element read operations controlled by the read enable signal. Although streaming is not used by Teng, streaming support was added to its memory controllers for a possible instance where an Observed peptide spectrum has been traversed but the read position (next element address) in the Hypothetical database remains at an element in the current Hypothetical peptide spectrum. In this instance, continuous read support allows Teng to bypass the remaining peptide data to reach the start of the next Hypothetical peptide to be matched with the Observed spectrum.

When processing input, Teng will not attempt, nor has a need for, random memory access. Once the first read request is sent, Teng will always expect the next contiguous element until the entire Hypothetical database has been read. Therefore, the read prefetch can be done on every read operation without concern for reading incorrect data. In addition, performing a continuous read to bypass elements in memory may seem counterintuitive and if a change in address is required, a reset signal along with a new starting address can be sent to the controller. However, the Hypothetical spectra are limited to lengths ranging from 8–25 amino acids whereas the Observed spectrum is not limited and is typically larger than the Hypothetical. It is more likely that a Hypothetical spectrum will be completely traversed before reaching the final element in the Observed spectrum. Therefore, for this

research, it was decided to not use additional memory resources on the FPGA to store addresses to the Hypothetical peptides in the database and design Teng to bypass spectrum elements when necessary.

5.8.2 The Border and PR Controllers

The border controller prepares the hardware for partial reconfiguration. Its task is to lock connections to the PRM I/O ports stopping all data transfers before the PRM is reconfigured. It does so by monitoring the execution status of the PRM. All I/O control and status signals between static hardware and the PRM are routed through the border controller and these signals are as follows:

- Memory/FIFO read enable,
- FIFO write enable and write acknowledge,
- FIFO full/empty,
- Database ready and reset (*i.e.*, restart from beginning or at new address),
- PRM result ready and result received,
- PRM idle (*i.e.*, all state machines are in the IDLE state), and
- Active PRM signal (a 2-bit signal that represents which PRM, Teng or SmWeng, is instantiated).

When signaled to prepare for PR, the border controller monitors the control and status signals and artificially asserts or deasserts them to cause the PRM to reach an IDLE state. It then signals the PR controller that it is safe to initiate PR. To prepare the Teng PRM for partial reconfiguration the border controller immediately deasserts the read enable signal and database ready signals of Teng that are connected to the Observed database. Once the Teng idle status signal is asserted, signaling all Teng functions have completed, the border

controller deasserts the connected Hypothetical database read enable and database ready signals. The controller also deasserts the outbound Teng result ready signal, deasserts the inbound result received signal, and then asserts the PR ready signal to the PR controller.

Immediately disconnecting Teng from the Observed database is a safe process because the PR controller does not signal the border controller to begin preparations until after Teng finishes reading in a Observed spectrum. In addition, Observed spectra are preloaded by Teng (loaded while another Observed spectrum is being processed) into a dual ported RAM. The RAM is needed because Teng contains two Match modules that share the same Observed spectrum (see Section 5.8.3) and using the RAM allows Teng to perform the protein identification process while disconnected from the Observed database. Waiting for an idle state to close off all remaining control signals ensures that all processing of the Teng PRM has been completed and performing PR will not corrupt the results.

Preparations for the partial reconfiguration of SmWeng is similar to that used by Teng. The border controller immediately disconnects SmWeng from the FIFO by asserting FIFO-empty signal connected to SmWeng and deasserts the FIFO read enable signal from SmWeng to the FIFO. The PR controller will only signal the preparation for PR when SmWeng is not reading a Teng result from the FIFO. To close the connection between SmWeng and the SmWeng database, the border controller waits for the assertion SmWeng idle status signal. It then deasserts the SmWeng database data ready signal, asserts the input stream throttle signal connected from SmWeng to the database, and then asserts the PR ready signal to the PR controller.

The PR controller monitors the flow of data between static and currently instantiated PRM (*i.e.*, Teng or SmWeng). It also monitors the data capacity of the FIFO where a "FIFO empty" status (perform SmWeng to Teng PR) and "FIFO full" status (perform Teng to SmWeng PR) are used by the PR controller to begin the preparation for and initiation of PR. The FIFO is configured with programmable full and empty status signals used as capacity threshold status signals for the PR controller. Once a threshold status is signaled, the PR controller monitors the Observed database read enable and outgoing data if Teng is active, or the FIFO output and read enable if SmWeng is active.

The Observed spectrum contains an "end-of-spectrum" flag that marks the last element in the spectrum. If the end flag is seen and FIFO full threshold is reached, the PR controller signals the border controller to begin PR preparations for the Teng PRM. If the empty threshold is reached and SmWeng is not reading from the FIFO, the PR controller signals the PR preparation of SmWeng.

Since only one PRM is instantiated at any given time, Teng and SmWeng PRMs are designed with a 2-bit "active" status signal that reflects which PRM (01 for Teng and 10 for SmWeng) is currently instantiated. An assertion, 1 value, of the active status signal does not mean that the state machines of the PRM are in a non-idle state, it only signals that the PRM is instantiated. When used in conjunction with the FIFO full/empty status, the active status signal aids the PR controller in signaling the appropriate PR process. For example, if the FIFO is full and the Teng PRM is active the PR controller will begin the process for the reconfiguration of Teng to SmWeng. In contrast, if the FIFO is empty and Teng is active, the PR controller will not begin the process for the reconfiguration of SmWeng

to Teng because Teng is already instantiated, making the process unnecessary. Lastly, to prevent the premature execution of the PRM during PR, the reset signal to the PRM is held asserted and the PR controller does not signal the border controller to release the lock on the I/O controllers until PR has completed (signaled by the assertion of the PRM active signal).

5.8.3 The Tandem Engine

To implement the Tandem engine (Teng), the Tandem prototype described in Section 5.5 was modified to be compatible with the two-stage Teng-SmWeng pipeline. One major change is that the Escore and Match wrapper stages are no longer PRMs. This change and other modifications to the Tandem prototype are provided in a list below. To make the Match and Escore partial reconfiguration modules, a wrapper module was required to ensure that their I/O port interfaces were identical even if many ports were left unconnected. Since the wrappers are no longer required for Match and Escore, they will from now on be referred to as the "Match stage" and "Escore stage", respectively. The changes implemented in Teng are as follows:

- The Escore and Match wrapper stages are no longer PRMs, Teng itself is a PRM that replaces SmWeng.
- Teng now has two parallel Match stages to match Hypothetical peptide spectra against an Observed spectrum.
- Teng has a Match stage controller that distributes Hypothetical peptide spectra, read from the database, to the input FIFOs of the parallel Match stages.
- Teng has a local RAM module, named *ScanStorage*, used to store an Observed peptide spectrum and preload a second spectrum while the parallel Match stages are processing the first.

- A second processing module, *Escore-stg2*, was added to the Escore stage of Teng to complete the computation of Equation 5.3.
- Teng produces an Escore stage result as a single precision floating-point value.

A visual of Teng is presented in Figure 5.10. A visual of the Tandem Smith-Waterman pipeline is presented in the Appendix. With respect to the Tandem prototype, the *MinMax*, *Match*, and *Hyperscore* modules are still in the Match stage of the Teng pipeline and the *Histogram* and *Escore* modules are still in the Escore stage of the pipeline. The second Match stage was added in Teng to speed up the production rate of the matching process—two Hypothetical peptides are now matched and scored concurrently.

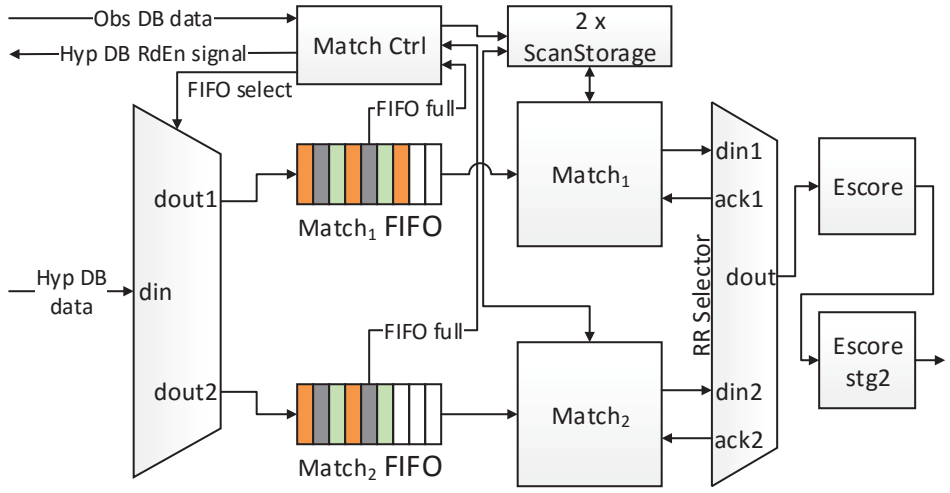

Figure 5.10

Architecture of the Tandem Engine

The Match stage controller distributes the Hypothetical spectra from the database by filling the input FIFOs of the Match stages in a round robin fashion. An entire Hypothetical

spectrum is written to the FIFO before the controller moves on to the next Match stage FIFO. Therefore, the controller bypasses an input FIFO if it is full but will not switch if the FIFO becomes full while a spectrum is being written to it. Observed spectra are also loaded into *ScanStorage* by the Match stage controller. *ScanStorage* is a memory module that contains two identical storage banks. The banks are mutually exclusive and only one bank can be written-to at any given time. An Observed spectrum written to a bank must be committed to memory first before the bank is made available for reading. Once committed, read access is enabled for the bank and write access is switched to the second bank.

Both parallel Match stages of Teng instantiate their own *ScanStorage* module to store a local copy of the Observed spectra. This is necessary because *ScanStorage* is not dual-ported and, therefore, simultaneous read access to the same bank is not supported. To compensate for this limitation, when writing an Observed spectrum the Match stage controller writes to both *ScanStorage* modules at once guaranteeing that the two *ScanStorage* modules store the same data. With the data in *ScanStorage* the same, the parallel Match stages concurrently match Hypothetical spectra against the same Observed spectrum. Once all the spectra in the Hypothetical database have been written to the FIFO, the Match stage controller does not switch the *ScanStorage* modules to the next bank until both Match stages have completed matching the remaining Hypothetical data in their FIFOs, and final processing has transitioned to the Escore stage.

All Teng modules that come from the Tandem prototype still use DSPs to perform integer operations on fixed-point binary numbers. The *Escore-stg2* module performs computations using single precision floating-point operations. The built-in DSPs of the FPGA

are not capable of performing division operations. Therefore, the *Escore* module only computes the summation operations $\sum x_i$, $\sum y_i$, $\sum x_i y_i$, and $\sum x_i^2$ of Equation 5.3 (using the DSPs) and accumulated n value—the number of peaks in the histogram that were used to calculate the summations. The remaining computations (b_0, b_1, and y) in the Tandem prototype are left to be performed by the recipient of the partial calculation (*e.g.*, host system CPU).

The *Escore-stg2* module was added to the Escore stage because a completed Equation 5.3 result is required as input to SmWeng. The module uses the single precision floating-point cores provided by the Xilinx core generation tool. The floating-point core support for the division operation was the primary reason for converting the escore fixed-point data representation to floating-point. In addition, the X! Tandem algorithm produces a floating-point escore value and therefore, Teng was designed to do the same with the addition of the *Escore-stg2* module.

The *Escore-stg2* module instantiates four fixed-point to floating-point converter cores, four floating-point multiplication cores, two adder-subtractor cores, and one division core. The computation of Equation 5.3 is controlled by a 10-state state machine that sets the order of arithmetic operations into the pipeline shown in Figure 5.11. The inputs to this pipeline are registered converted floating-point values of the summations and n values provided by the *Escore* module. The computed b_1 value is registered to be multiplied with the highest hyperscore value in the histogram, x_{max} (the x in Equation 5.3); x_{max} is also a registered floating-point value converted from fixed-point. The state machine reuses some floating-point cores to compute the value of b_0 and *Escore-stg2* produces the y value of

the Hypothetical peptide that resulted in the highest hyperscore ($y = b_0 + b_1 x_{max}$). This y value and corresponding matched peptide identification information is written to the Teng FIFO. *Escore-stg2* takes approximately 42 clock cycles to produce a y value.

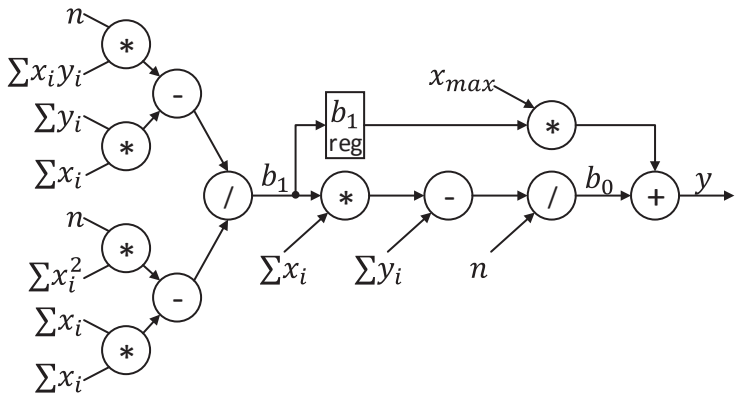

Figure 5.11

Escore-stg2 module computation pipeline

5.8.4 The Smith-Waterman Engine Supporting Hardware

Since SmWeng is a non-PR complete standalone processing engine, supporting hardware was implemented to incorporate SmWeng into the pipeline. The supporting hardware is a SmWeng data controller and the SmWeng memory controller described above, and a wrapper PRM that instantiates SmWeng (the SmWeng PRM). The SmWeng data controller is responsible for retrieving the input data from the FIFO; using this data to submit a read request to the SmWeng database (via SmWeng memory controller) and connecting the streamed data to SmWeng. If the FIFO is not empty, the SmWeng data controller waits for the assertion of the SmWeng result ready signal to begin streaming the next in-

put. When FIFO data is unavailable (FIFO empty or locked by the border controller) the SmWeng data controller returns to an IDLE state when result ready is asserted.

To perform resource sharing PR, two wrapper PRMs were implemented. One wrapper instantiated SmWeng (making the SmWeng PRM) and the other instantiated Teng (the Teng PRM). Both PRMs were named the same and had identical external ports (in name and width) but, the internal design supported the instantiated module. For the SmWeng PRM, all external ports unused by SmWeng were tied to ground (*i.e.*, a logic zero); the same is done in the Teng PRM for ports unused by Teng. When performing PR from SmWeng to Teng, the Teng PRM is loaded into the PRR and the SmWeng PRM is loaded when configuring from Teng to SmWeng.

5.9 The ChipScope Logic Analyzer and IP Core

Xilinx provides the ChipScope Logic Analyzer tool that can monitor the execution of hardware design loaded onto the FPGA. To use this tool, a ChipScope intellectual property core must be included in the hardware design; the ChipScope core is generated using the Xilinx CoreGen tool. The ChipScope core monitors the state of the hardware modules in the design. The state of these modules are then displayed in a waveform by the ChipScope tool. For experimentation, several ChipScope cores where instantiated in the Tandem Smith-Waterman design.

A ChipScope core records, in real time, the state of signals in hardware for a user defined window of execution—specified in units of time (*e.g.*, 4096ns). Recording, is triggered by an event in the hardware execution. The trigger is also user specified and

reflects a condition in the hardware where the user wishes to begin recording. The recorded data is stored, by the ChipScope core, in the BRAMs of the FPGA; signal status and changes are monitored but not stored until the trigger condition is met. The trigger signals and signals monitored by the ChipScope core are specified by the user in the hardware design. Therefore, after any addition or removal of trigger and monitored signals, the hardware must be reimplemented because of the change in the design architecture.

5CHANGED BY RIKK assertions/deassertions -¿ assertions (deassertions) The ChipScope tool lets the user define trigger conditions where, the user specifies which single trigger signal assertion (or deassertion), or combination of signal assertions (deassertions), or bus value will start recording the hardware execution. Once recording has completed, the tool displays the waveform of the recorded signals and bus values. The user can use this result to verify the correctness of the implemented hardware. The real-time execution results are similar to a software-based simulation of the hardware design. However, the advantage of using ChipScope is that the results provided are not simulated. The results recorded by ChipScope is the actual hardware execution and not an estimate of the execution based off of an analysis of the hardware behavioral design.

5.10 Experiments

For this research the Tandem Smith-Waterman hardware pipeline was realized. All controllers, static Teng module, and input databases were implemented on the FPGA. The Teng PRM and SmWeng PRM partial bitstreams are stored on a PC and uploaded to the FPGA using the JTAG configuration interface and the Xilinx iMPACT tool. Partial

reconfiguration was also executed using the JTAG configuration interface. To verify that partial reconfiguration was successful, the ChipScope tool was used to monitor the signals of the instantiated PRM and the flow of data to and from the PRM.

The Tandem Smith-Waterman design was executed using a single Observed database and a single SmWeng database. The spectra in the Observed database was matched against three different-sized Hypothetical databases using five different parts-per-million (PPM) values of 10, 5, 2, 0.5, and 0.0625. The hardware implementation process was run for each Hypothetical and PPM combination. Therefore, a total of 15 distinct Tandem Smith-Waterman configurations were implemented, tested, and timed for this research. To measure the execution time of Teng and SmWeng, timer modules were implemented to count the number of clock cycles the modules took to complete processing its input. The ChipScope tool was used to gather the times recorded by the timer modules. The reconfiguration time estimate formula (Chapter 4.5) was used to estimate the reconfiguration time of the ICAP configuration interface. This reconfiguration time, the recorded execution time of the Teng and SmWeng, and the FIFO threshold values were used to derive a generalized mathematical formula for determining the value of performing PR on similar, slow-to-fast, pipelines.

CHAPTER 6

EXPERIMENTAL RESULTS

The execution results of the Tandem Smith-Waterman design are presented in this chapter. The FPGA is located on a PCIe expansion slot card. The card can be placed in stand-alone mode, where the FPGA can be programed and used as its own system, external to a host machine (*e.g.*, PC or server machine). The card can also be used as a component of the host system, where the FPGA could be made into a hardware co-processor. For this research, the FPGA was used in stand-alone mode. All data was stored internal to the FPGA or uploaded to the FPGA via the JTAG configuration interface.

The chapter begins with sections discussing the FIFO thresholds and changes to the target hardware. It then follows with the implemented PR design configuration bitstream sizes and configuration times (Section 6.2). Section 6.3 presents the three Hypothetical databases, the Observed database, and the SmWeng database and their capacities. The chapter concludes with the Tandem Smith-Waterman pipeline timed execution.

6.1 FIFO Size and Thresholds

The Teng FIFO is 72-bits wide and a capacity of 1024 72-bit rows. The 72 bits contain the 32-bit single-precision floating-point escore value of the peptide with the maximum hyperscore value, the 10-bit ID of this peptide, and the 20-bit fixed-point hyperscore of

the peptide (total 62 bits). The 10-bit peptide ID is the start-of-frame address location in the SmWeng database. The streamed input will be used by SmWeng to find the functional similarities between the matched peptide and other known peptides in the SmWeng database.

Because the combined escore, ID, and hyperscore values sum to 62 total bits of data, the remainder of the 72-bit Teng FIFO width was left to support larger peptide ID and hyperscore values that can result from modifications performed in future work. The FPGA resource for implementing FIFOs is a 36-bit wide FIFO36 memory resource. To generate FIFOs, several FIFO36 resources are be combined to make a larger FIFO of any given capacity and width. Because the FIFO36 FPGA resource used for implementing FIFOs is 36 bits wide and the 62-bit combined escore, ID, and hyperscore value does not fall on the 36-bit boundary, two FIFO36 resources are used to support the 62-bit width. Therefore, instantiating a 72-bit wide Teng FIFO and reserving the remaining bits for future design modifications did not affect the number of FIFO36 FPGA resources used by the FPGA to generate the Teng FIFO.

The Teng FIFO is configured with user-defined programmable full and programmable empty status thresholds. The thresholds mark the capacity status in the Teng FIFO where PR can be initiated. The FIFO must be filled beyond 80% in capacity (about 819 elements) before the programmable full status is signaled by the FIFO. When this threshold is reached and the Teng PRM is active, the PR controller will reconfigure Teng to SmWeng. Instances where SmWeng is active and the full threshold is reached would be after the partial reconfiguration from Teng to SmWeng. In this case PR will not occur. For the pro-

grammable empty status to be signaled by the FIFO, the stored capacity must fall below 10% (about 10 elements). In this instance, if SmWeng is active, the PR controller will swap SmWeng for another Teng.

6.2 Target Hardware Modifications

The Tandem Smith-Waterman hardware design uses multiple memory structures that are implemented using the block RAM resources (BRAM) of the FPGA. In addition, five ChipScope cores are instantiated to monitor several modules in the design. The ChipScope cores record 2048ns or 4096ns of execution time and require 300-500 BRAM to store the recorded data. The three ROM databases, *ScanStorage* module, Teng FIFO, input FIFO to the Teng parallel Match stages, the FIFO between the Match stages and the Escore stage, the memory used by the *Histogram* module (to store the histogram), and the inclusion of the ChipScope modules exceeded the available BRAM resources of the Virtex-5 FPGA. As a result, the Virtex-6 FPGA became the target hardware for this research.

To support the new target hardware, all Xilinx CoreGen hardware cores were regenerated for the Virtex-6 and the user constraints file for the design was rewritten to reflect the hardware specifications of the Virtex-6. The DSP and BRAM primitive instantiations in the hardware design were rewritten to instantiate the equivalent primitives of the Virtex-6; several ports on the Virtex-5 primitives were not available with the Virtex-6 primitive and some were supported due to backwards compatibility. The digital clock manager (DCM) primitive on the Virtex-5 FPGA was removed in the Virtex-6 and replaced with a mixed-mode clock manager (MMCM). Therefore, the clock module that instantiated the DCM

and supporting I/O and global clock buffers was removed from the design. The clock module was replaced by an MMCM core generated using the Xilinx CoreGen tool. Modification made to Tandem Smith-Waterman design for software simulation testing for the Virtex-5 was removed because the design failed to pass the synthesis phase of the hardware implementation process for the Virtex-6. In addition, several errors in the Xilinx CoreGen generated, floating-point core, Verilog code were found to cause synthesis to fail. These errors were manually corrected and reported to Xilinx.

6.3 Bitstream Sizes and Configuration Time

The Tandem Smith-Waterman design has two architectural configurations. In the first configuration, a single Teng and a single SmWeng are instantiated and the pipeline is a complete two stage pipeline. The Teng FIFO buffers Teng results for SmWeng and the data within the FIFO is consumed by SmWeng once the FIFO signals a non-empty state. The second configuration is a parallel Teng module configuration. The pipeline in this configuration is incomplete because the SmWeng module is replaced by a second Teng. In this configuration the purpose of the Teng FIFO is to buffer Teng results until SmWeng is reinstantiated to replace the second Teng.

The Xilinx PlanAhead tool generates a "full" configuration bitstream for all architectural configurations of the hardware design in addition to the partial bitstreams for the PRMs. Like the Xilinx PR design flow, the tool assumes that each configuration is a complete standalone configuration that does not depend on the presence of a PRM to generate

a final result. The PRM is assumed to make changes to hardware function, not augment a pipeline to increase the production rate of a pipeline stage.

The bitstream sizes for the two architectural configurations are both 12.4MB. These configuration bitstreams include the digital logic representation of all static elements (*i.e.*, static modules, clocking resources, memory and FIFOs, and I/O buffers) and the digital logic of the PRM instantiated by the relative architecture. The partial bitstream sizes for Teng and SmWeng PRMs are also identical, which is 2.29MB. For a partial reconfiguration design, PR modules are placed inside a partial reconfiguration region (PRR). The PRR is a user constrained rectangular partition of FPGA resources reserved for the instantiated PRMs. The region must encompass all the FPGA resources needed by all the PRMs that is to be instantiated by the PRR. Partial bitstreams are generated for the resources in the PRR, not specifically the resources of the PRM. Therefore, the partial bitstream configures all the resources within the PRR by setting them to an active or inactive state relative to the PRM that is being instantiated. Thus, the Teng and SmWeng partial bitstreams are equally sized because they are instantiated by the same PRR.

6.3.1 JTAG and ICAP Configuration Times

Using the 66Mbps JTAG interface, the bitstream upload time (*i.e.*, configuration time) for a full configuration bitstream is 32 seconds. The upload time for the partial bitstreams is 5 seconds. The JTAG interface (pronounced *jay-tag*) is an external configuration interface, relative to the FPGA, that is built into the FPGA board. The interface is connected to a PC via a USB JTAG platform connector. To configure the FPGA, the bitstreams are read

from the PC hard drive and uploaded to the FPGA using the Xilinx ISE iMPACT tool. iMPACT provided the time taken by JTAG to upload bitstreams. The USB connection and reading the bitstreams from the PC hard drive adds a lot of overhead to the configuration process. Excluding the overhead, the JTAG upload time for the 2.29MB partial bitstream is approximately 278ms.

The iMPACT tool and JTAG interface was used to perform manual partial reconfiguration for experimentation. The FPGA was initialized by uploading the parallel Teng configuration to the FPGA. When the FIFO threshold was reached and the PR controller signals ready for PR, the SmWeng PRM partial bitstream was uploaded to the FPGA using iMPACT. Once the SmWeng PRM was uploaded, the Xilinx ChipScope tool was used to verify that the Teng result data in the Teng FIFO was read and processed by SmWeng. A similar task was done for the PR from SmWeng to Teng. The FPGA was initialized with the single Teng with SmWeng configuration and the SmWeng was reconfigured into another Teng by uploading the Teng partial bitstream. Once configured, ChipScope was used to verify that the second Teng received and Observed spectrum and began to match the spectrum to the Hypothetical database.

The ICAP interface is an internal configuration interface primitive to the FPGA fabric. It can receive a configuration bitstream from any internal (*e.g.*, BRAM) or external (*e.g.*, DDR RAM, Hard disk, PCIe, Ethernet) data source, as long as logic is provided to connect them. The ICAP interface supports the transmission of 32 bits at a frequency of 100MHz. Therefore, it is capable of uploading bitstreams to the configuration memory of the FPGA at a speed of 3.2Gbps. Using ICAP, the estimated configuration time for the full configu-

ration bitstream is approximately 31ms and 5.8ms for the partial bitstreams—3.1 million and 573 thousand clock cycles, respectively, in a 100MHz design. The ICAP interface is the superior configuration interface for executing PR and it enables autonomous partial reconfiguration. However in the current state of the Tandem Smith-Waterman design, using JTAG to upload partial bitstream, and ChipScope to verify the proper execution of the modules and obtaining module execution times was a sufficient alternative for experimentation purposes. Therefore, integrating ICAP into the Tandem Smith-Waterman design was reserved for future work.

6.4 Input Database Capacities

The design was tested with three Hypothetical-spectra databases, one Observed-spectra database, and a SmWeng database. The databases were generated as ROM memories using the BRAM primitives of the FPGA. The SmWeng database is a 12KB 36-bit wide ROM initialized with known peptide data and upon a read request, the data is streamed to the SmWeng PRM. The stream is framed with start-of-frame and end-of-frame flags that mark the beginning and ending, respectively, of the stream. Therefore, each 36-bit row read from the database is a fraction of the framed input data. The SmWeng database address to the input's start-of-frame is provided with the matched peptide information written to the Teng FIFO.

The dimensions of the three Hypothetical database ROM memories and corresponding storage capacities are listed below:

1. 42-bit by 16.4K deep (42×16.4K) ROM, 86KB storage
2. 42×32K ROM, 172KB storage

3. 42×64K ROM, 344.2KB storage

The three Hypothetical databases will be referred to as the 16K, the 32K, and the 64K database, respectively. The databases were populated with *in silico* trypsin-digested peptide spectra of known proteins found in FASTA files of human and animal proteins. The Hypothetical database generated from the FASTA files is 76MB, which is too large for the FPGA to store. Therefore, the three Hypothetical database ROM memories contain a subset of the larger Hypothetical database.

The 16K database contains 212 Hypothetical spectra, the 32K database contains 410 spectra, and the 64K database stores 700 spectra. The Hypothetical peptide spectra in the databases are not uniform in size (*i.e.*, number of peaks per spectrum) and therefore, doubling the database capacity does not necessarily double the number of spectra stored. A peak in a Hypothetical spectrum is a single 42-bit value read from the database. The 42-bit peak contains the 36-bit m/z of a point in the Hypothetical spectrum and a 6-bit flag that identifies the point's ion type; the six ion types are a, b, c, x, y, and z. For the matching algorithm, the intensity of the Hypothetical spectrum points are assumed to be one. The peak is a fixed-point value that is compared, by Teng, to a peak read from the Observed database.

The Observed database is a 274KB ROM of unknown peptide spectra. The database is 30K deep and 72 bits wide. The read width of the database is 72 bits that contain the fixed-point mass and intensity value of a peak in Observed spectrum; read width is the number of bits returned per memory read request. Both the mass and the intensity are 36-bit fixed-point values with 12-bits of fraction. This Observed data comes from "raw" data

files that contain the finalized results of a scanned biological mixture that has gone through the MS/MS process (see Section 5.2.1). Observed spectrum data is a collection of digested peptides extracted from a scanned biological mixture of proteins. The Observed spectrum does not contain peaks corresponding to the peptides of a single protein, the peaks are the peptides formed from the amino acids of multiple proteins.

6.5 Execution Performance

A timer module was implemented to record the time taken by Teng and SmWeng to generate an output. The timer module records the time, in number of clock cycles, between the engines' request for input and the assertion of the result ready signal when the input has been processed. The timer also stores the global minimum and global maximum recorded times over all processed inputs. Timing of SmWeng begins when SmWeng retrieves a single Teng result from the Teng FIFO. The timer stops when SmWeng signals that its computed result is available. The SmWeng timer is restarted when SmWeng reads from the Teng FIFO again.

Timing of Teng begins after the Observed spectrum is written to the ScanStorage memory module and when the write commit signal is asserted. The timing stops when the output of *Escore-stg2* is written to the Teng FIFO. The Teng Match stage and Escore stage, in Figure 5.10, are designed to operate concurrently and independently of each other. The Escore stage processes all hyperscore data items stored in the FIFO, regardless of the current state (processing or idle) of the Match stage. When finished matching and Observed

spectrum to the Hypothetical database, the Match stage begins matching the next available Observed spectrum. The stage does not wait for the Escore stage to produce a result.

Because of the potential for the Match stage to process a new Observed spectrum while the Escore stage is still processing the hyperscores of a previous Observed spectrum, two timers were used to record the execution time of the matching and escoring of a single Observed spectrum. One timer is used to time the Match stage of Teng and the other to time the Escore stage. The Match timer begins when the Observed spectrum is committed to ScanStroage and ends when the final hyperscore of the matched Observed spectrum is written to the input FIFO of the Escoring stage. The timer is then restarted when the next Observed spectrum is committed to the ScanStorage, even while the Escore stage is still processing. This approach prevents missing the start of matching the next Observed spectrum, as a result of waiting for the Escore stage to finish and stop the timer.

The timing of the Escore stage begins when a hyperscore data value (typically the first) is produced by the Match stage. The current time recorded by the Match stage timer is used as the starting time for the Escore stage timer. Passing the Match stage current time to the Escore timer continues the accumulation of clock cycles for the complete execution of the Teng process. The time returned by the Escore stage timer reflects the total time taken by the Match stage and the Escore stage; the number of clock cycles taken by Teng to process the Observed spectrum.

6.5.1 Execution Times of Teng and SmWeng

Table 6.1 shows the execution times of the Teng pipeline stage for the 32K and 64K Hypothetical databases. The table lists the minimum and maximum clocks cycles taken to match and score Observed spectra to the Hypothetical database. The Observed database was matched against all three Hypothetical databases with three different PPM (*i.e.*, parts per million) values. When matching the mass of an Observed spectrum peak to a Hypothetical spectrum peak, a match range is applied to the Hypothetical peak. The match range is an upper bound and lower bound set by a user-defined delta value called the PPM (Section 5.2.2). The mass of an Observed spectrum peak must fall within the Hypothetical peak match range to indicate that a match has been found.

Table 6.1

Min and max Teng pipeline stage execution times in clock cycles

Hypothetical DB			**Teng**		**Teng PRM**	
ROM	capacity	PPM	min	max	min	max
42×32K	172KB	10	52 775	92 651	34 506	96 439
42×64K	344.2KB	10	94 605	164 071	67 062	189 823
42×32K	172KB	5	52 798	89 808	34 799	88 871
42×64K	344.2KB	5	66 511	171 493	66 658	114 949
42×32K	172KB	2	53 404	89 884	35 315	93 642
42×64K	344.2KB	2	67 958	162 306	67 959	109 311
42×32K	172KB	0.5	52 564	91 974	34 630	93 894
42×64K	344.2KB	0.5	72 026	170 136	66 650	188 820
42×32K	172KB	0.0625	51 873	92 014	34 516	94 674
42×64K	344.2KB	0.0625	77 686	170 253	66 536	187 632

The times in Table 6.1 reflect the clock cycles taken by Teng and the Teng PRM to match its Observed spectrum to the Hypothetical database and generate a single escore output value. The global min/max times stayed the same when testing the execution of the single Teng configuration versus the parallel Teng configuration. The difference between the configurations is that the parallel Teng configuration matched two Observed spectra and generated two escore values within the same amount of time the single Teng configuration produced one escore value. The Teng process using the 16K Hypothetical database, not shown in Table 6.1, achieved timings within the range 17747–55900 clock cycles for the different PPM values. The Teng search times gathered for experimentation tend to increase by twice the number of clock cycles when the Hypothetical database doubled in size. This trend suggests that the time to search the Hypothetical database grows linearly with the size of the database.

The SmWeng PRM achieved a maximum process time of 106 clock cycles. Because of the limitation in BRAM resources, the SmWeng database was restricted to 12KB. This reduced the amount of known peptides searched by the SmWeng PRM. However, the time taken by Teng is much greater than SmWeng that, for the 64K database, if the SmWeng PRM search time was multiplied by one thousand, it will still complete processing data before Teng produces a single output. In a future work (Chapter 8), all databases will be stored in external memory/storage (*e.g.*, DDR RAM or hard disk). When using external memory, the communication time will increase due to a higher read latency compared to the internal FPGA ROM and database capacity will be larger (growing from KBs to MBs of data searched) because of the greater storage capacity.

CHAPTER 7

ANALYSIS

This chapter discusses the design choices made for the hardware-based Tandem Smith-Waterman pipeline implementation and design options for future work. It will begin with the timed performance of Teng. Fifteen Observed spectra were identified to find a trend in Teng execution relative to spectrum size. Section 7.2 explains why ROMs were used for the databases and the estimated overhead associated with moving the databases to DDR RAM. Section 7.3 lists the supporting hardware required for a resource sharing PR design. Lastly, this chapter concludes with the contribution of this research, which is a mathematical formula for determining the value of performing PR to improve the production of a bottleneck pipeline stage.

7.1 Algorithm Performance and Configuration Time

The execution times of the Teng algorithm using the 64K Hypothetical database (700 spectra, 344.2KB of data) ranged between 66.5 thousand to 188.8 thousand clock cycles. The SmWeng database contains 700 entries. Searching the entire SmWeng database will take approximately 74.2 thousand clock cycles (K cycles). For any given input value retrieved from the Teng FIFO, SmWeng will not search the entire database. The SmWeng

module will search a subset of the database, which are known peptides with characteristics similar to that of the matched Observed spectrum.

The number of database entries searched by SmWeng can vary per Teng result. However since this module was implemented in a previous research effort, the architecture of the design could not be viewed to determine the procedure for selecting items from the database. Thus, for this research, it was decided to use the worst-case performance of the algorithm for timing analysis and experimentation. For a worst-case performance, a search of the entire SmWeng database was assumed. Lastly, if this is effective in the worst case, then the real effectiveness of SmWeng will be greater.

The 66.5K cycle Teng time makes SmWeng the slower process in the pipeline. With Teng being the faster module in this case, there is no benefit in reconfiguring SmWeng into a parallel Teng module because SmWeng will cause the pipeline to stall once the Teng FIFO is filled to capacity. The 66.5K cycle Teng match time corresponds to an 85-peak Observed spectrum in the database. This spectrum is the smallest spectrum in the 30K Observed database. The Observed database is a subset of spectra taken from a raw data file of scanned biological mixtures. The raw file contains over 35 thousand scans, which include spectra with fewer than 85 peaks. Therefore with the 64K Hypothetical database, partial reconfiguration is not useful when matching Observed spectra of 85 peaks or fewer.

To derive the mathematical PR model for this research, timings were taken of several spectra in the Observed database to find a trend in the Teng execution time with respect to the size of the Observed spectrum. Figure 7.1 shows the execution times of 15 different spectra with peaks ranging from 85, the smallest spectrum, to the largest Observed spec-

trum of 524 peaks. These spectra represent the various spectrum sizes (number of peaks) in the Observed database. Other spectra in the database either have the same number of peaks as the selected spectra or are relatively close. The 64K Hypothetical database was used when gathering the Teng timings and the PPM used for matching was 10. Figure 7.1 shows a linear relationship between the size of the Observed spectrum and the Teng match time. As the Observed spectrum size increases, the Teng match time also increases.

Time cataegorization

peaks	time
85	66536
93	68224
123	77573
152	86349
178	94605
214	104949
248	114489
290	121650
300	128651
342	133961
388	150160
408	158145
431	164071
475	172270
524	189823

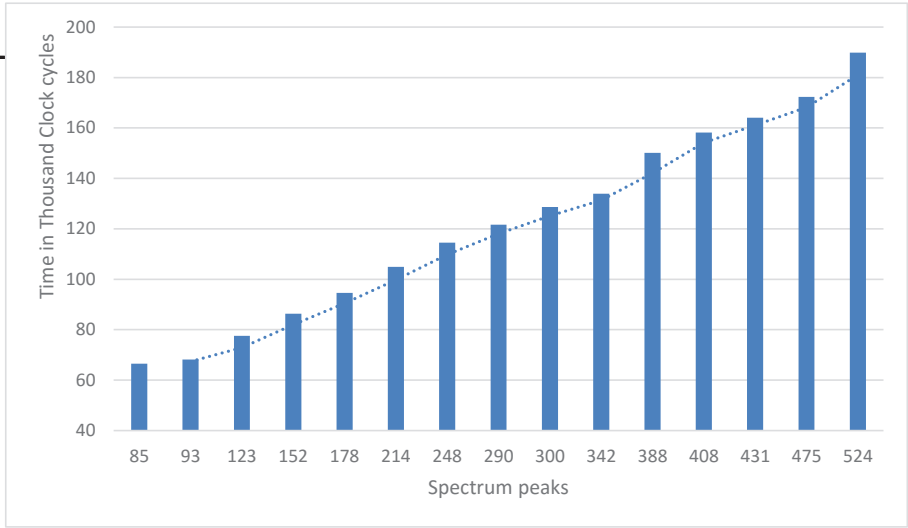

Figure 7.1

Graph of timing trends.

In a two stage partial reconfiguration design, if a module is reconfigured to increase the production of a slower stage, this module must be capable of consuming the buffered results of the slow stage fast enough to prevent pipeline stalls when returned back to its original configuration. In the Tandem Smith-Waterman design, with the 64K Hypothetical database, this point is when Teng performs its match process on Observed spectra that

consist of approximately 388 or more peaks. The time to match an Observed spectrum of 388 peaks is more than twice the 74.2K cycle time taken by SmWeng to search the SmWeng database. Therefore, when the Teng FIFO is full as a result of the parallel Teng configuration and after performing PR, the SmWeng can consume two results from the FIFO in the time taken by Teng to produce one result.

If smaller Observed spectra are being processed, the spectra with 248–290 peaks are matched by Teng in a little more than 1.5 times the K-cycles taken by SmWeng to search the SmWeng database. Processing spectra with 248–290 peak allow for at least three Teng output values to be consumed by SmWeng per two output values generated by Teng. From the point of 248 peaks, smaller Observed spectrum begin to converge toward one input consumed by SmWeng per one item produced by Teng. At this point, PR should not be performed.

7.1.1 Configuration Time for PR

Partial reconfiguration is best performed autonomously through ICAP by the FPGA, if it is a stand-alone device (or autonomous system), and by the host system that monitors the configuration and execution status of FPGA if the FPGA is a co-processor. To upload the 2.29MB partial bitstream to the FPGA, ICAP is estimated to take 537K cycles at 100MHz. Using the time illustrated in Figure 7.1, when matching Observed spectra that consists of 248–342 peaks, this reconfiguration time equates to approximately six Observed spectra matched upon the completion of PR. When processing spectra with 388+ peaks, four Observed spectra will have been matched by the time of completion. Therefore

when processing spectra of 248–342 peaks, performing a PR cycle (*i.e.*, PR from parallel Teng modules to SmWeng and later back to parallel Teng modules) yields approximately 12 matched Observed spectra (8 for 388 peaks).

In addition to reconfiguration time, PR depends upon the total spectra stored in the Observed database. A raw file can contain over ten thousand (35 thousand in the file used for this research) scans of biological data, and a typical Observed database will consist of several thousand spectra (or all spectra in the raw file). Hence, the number of Observed spectra in a database is far greater than the number of spectra matched during PR. A limitation would be if the majority of the scans in the raw file consists of Observed spectra with peaks (*e.g.*, 85–123 peaks in this case) that reduce the Teng module match time down to a time similar to that taken by SmWeng to search the SmWeng database. Otherwise, when using an Observed database that consists of large spectra (*e.g.*, 388+ peaks), there will be opportunities of a PR cycle to be performed throughout execution.

There is, however, an instance where only a single direction PR (Teng to SmWeng) should be performed. This instance is when the Observed spectra in the database (or those remaining to be processed) is fewer than the capacity of the Teng FIFO, where the number of spectra are less than the Teng FIFO full threshold. Given this instance, if the configuration of the Tandem Smith-Waterman design is in the parallel Teng configuration, PR should be done to configure the Teng PRM to SmWeng PRM upon processing the final Observed spectrum. If the hardware is already in the single Teng, single SmWeng configuration, PR of the SmWeng PRM to Teng PRM engine should not be done; unless the

remaining spectra can fill over half the Teng FIFO, there is no need to perform PR to increase matching the remaining spectra.

Lastly, the configuration time estimate of ICAP assumes the same communication interface as the databases, which is a memory interface handled by a controller that streams the partial bitstream from a ROM memory on the FPGA to the ICAP interface also on the FPGA. Both partial bitstreams must be stored to allow for swapping the Teng PRM and the SmWeng PRM. There are not enough memory resources on the FPGA to store the partial bitstream of the PRMs. The bitstreams must then be stored off-chip in a programmable ROM built into the FPGA board or in DDR RAM.

The use of external memory will increase the communication overhead, which will increase the configuration time. Nevertheless, in a production design, the Hypothetical, Observed, and SmWeng databases will also be stored in external memory (or disk) and with larger capacities. At the least, the communication overheads, between FPGA and external memory, for the databases and ICAP will be the same, yielding similar results (with larger overheads) as estimated above.

7.2 Database Storage

The Hypothetical, Observed and SmWeng databases are all stored as ROMs using BRAM memories of the FPGA. The Virtex-6 has 704 memory resources that can be instantiated as either BRAM or FIFO. Instantiating the database as ROMs on the FPGA for the Tandem Smith-Waterman hardware was a decision made to simplify the design. With the databases being on the FPGA, no additional logic was implemented to instantiate the

Xilinx memory interface core and manage the DDR memory address scheme (includes row, bank, and column bits) for all databases (each had a different read width). In addition, no synchronization logic was written (or FIFOs instantiated) to transition data received from the 533MHz memory interface core to the 100MHz clock domain of the Tandem Smith-Waterman hardware.

The BRAM-based ROMs memories had a simplified address scheme—each single digit increment of the address was a row of data. This simplified the address counter of the memory controller modules that were implemented as the interface between the ROMs and the Teng and SmWeng stages. The address counter of the controller was a 32-bit register whose stored address is incremented by one every time the read enable signal was asserted. The address counter does not need to account for which bank or column in memory to request data from. Every address received per read request yields data bit sizes in accordance with the read width (a row in this design) of the ROM. The ROMs also ran on the 100MHz clock, and they were configurable to be dual ported allowing concurrent reads, writes, or read-write operations to be performed on the memory. Lastly, the read delay for the ROM was one clock cycle.

The disadvantage of the ROM databases is that their capacity is limited to the available BRAM of the FPGA. Because of this limitation, to perform experiments for this research required the search databases (Hypothetical and SmWeng) and unidentified data (the Observed database) to be reduced down from 100+ megabytes of data to 630.2KB (64K Hypothetical + Observed + SmWeng database capacities). In addition, the databases competed with the ChipScope monitoring core for BRAM resources; the ChipScope cores

used 361 BRAM resources. Unfortunately the ChipScope cores were needed to monitor the signals of the Tandem Smith-Waterman hardware for debugging, experimentation, and data collection. Therefore, without losing visibility of the execution status of monitored hardware, the databases were fixed at a set maximum (274KB Observed, 12KB SmWeng). The exception is the 64K (344.2KB) Hypothetical database which requires the removal of one of the ChipScope cores at the expense of losing the monitoring of the static Teng module.

7.2.1 Databases in DDR SODIMM RAM

The target device for experimentation was the HiTech Global Virtex-6 FPGA development board [37]. The development board is equipped with a DDR3 SODIMM external RAM memory slot that supports up to 4GB of RAM storage and a data transfer rate (*i.e.*, bandwidth) up to 1875Mbps. To connect to the external memory, a Xilinx memory interface must be generated and instantiated by the hardware design. The memory interface core supports memory read widths of 64 bits and a maximum bandwidth of 1060Mbps. Ergo, 1060Mbps is the maximum achievable bandwidth between the FPGA and external memory.

The benefit of using external memory is the storage capacity. Modern SODIMM RAM memory can store gigabytes of data. With gigabytes of available storage, larger databases can be generated for the Tandem Smith-Waterman pipeline. However, to use RAM as source of storage introduces two overheads. The first overhead is the data transfer time.

With the Virtex-6 memory interconnect bandwidth and a read width of 64 bits, the data transfer time (tXFER) is 60.4ns per 64 bits of data sent ($64\mathrm{b}/1060\mathrm{Mbps}$).

The second overhead is the read latencies. Latencies associated with memory reads are the column address strobe (CAS or tCL) latency, the column address delay (tRCD), and the row percent time (tRP). These latencies are multiplied by the RAM clock cycle time and then summed to determine the delay (in nanoseconds) between sending an address to the RAM controller, and the receipt of the corresponding data. For example, if a row in RAM has not been selected, the latency is tRCD + tCL; if a row in memory has been selected and a read request is for a different row, the latency is tRP + tRCD + tCL; if a row has been selected and a read request is for the same row, the latency is tCL. For each read request, tXFER occurs twice, once for the read request sent to RAM and the other for the data returned from the read. Thus, the delay for a single read request is $2\times$tXFER$+$read latency.

With respect to the Tandem Smith-Waterman design, assume that the 64K Hypothetical database is stored in a DDR3 SODIMM RAM. The RAM is a 1 gigabyte Crucial CT102464BC160B DDR3 SODIMM RAM memory [17] equipped on the HiTech Global Virtex-6 development board. The RAM's clock cycle time is 1.25ns and its read width is 64 bits. The latencies tCL, tRCD, and tRP are all 11 clock cycles. This means the true latency is, $latency(ns) = RAM\ clock\ cycles\ time(ns) \times latency\ clock\ cycles$, which is $1.25 \times 11 = 13.75ns$. If a row in this memory has been selected and a read request is for a different row, it will take 41.25ns for the memory to present the data requested and 120.8ns of transfer delay. The 64K Hypothetical database has $65,555$ rows of Hy-

pothetical peptide data and during the match process, Teng reads the entire Hypothetical database. Every read request made to the Hypothetical database will include the 41.25ns read + 120.8ns transfer delay (162.05ns). This is a 17 clock cycle delay in the 100MHz Tandem Smith-Waterman pipeline. In contrast, the 64K Hypothetical ROM database has a one clock cycle (10 ns) delay before data is returned from the ROM. Seventeen clock cycles per read is the estimated delay for the Hypothetical and SmWeng databases. For the Observed database the delay is 38 clock cycles, because of the 72-bit combined mass and intensity values associated with a peak in an unknown peptide spectrum. The 72 bits will require two read requests to retrieve all the peak's data.

To avoid stalling the pipeline as a result of memory overheads, a FIFO can be used to buffer the input data. The buffer can reduce delays by allowing the memory controller to streamline memory reads (assuming reads are from contiguous rows in memory) and fill the buffer without having to wait for a read request from the pipeline. The Tandem Smith-Waterman design uses FIFOs to buffer Hypothetical data for the parallel match stages in Teng but it neither buffers the Observed nor the SmWeng database. For SODIMM RAM to be used, the Tandem Smith-Waterman pipeline will need to buffer all input data.

7.2.1.1 Partial Bitstreams in DDR SODIMM RAM

Storing the partial bitstreams is SODIMM RAM will not change the points for PR in the Tandem Smith-Waterman design if all input databases are also stored in RAM. Since the memory overhead are same for reading from the databases and reading the partial

bitstream, there will be little difference in configurations times other than the addition of the memory overhead.

7.3 Designing for Resource Sharing PR

During the implementation of this application this research discovered that the following must be included in a hardware application's design if support for resource sharing PR is to be achieved:

1. The application must include a controller that monitors the FIFO buffered data. The controller should respond by preparing for PR (*e.g.*, signal the border controller) when the empty and full FIFO thresholds have been reached.
2. The application must include a controller to distribute input data to the parallel bottleneck modules (*e.g.*, the parallel Teng modules).
3. The application must include a controller to lock all input data paths of a module that is to be reconfigured (a PRM), and then once all remaining data has been processed by that module, signal to begin the PR process. This controller must keep these inputs locked until the PRM has been reconfigured back.
4. The application must have collector logic to gather output results from the parallel bottleneck modules and then funnel them to the FIFO leading to the next pipeline stage.
5. Data traversing the pipeline should not have a sequence order, otherwise you run the risk of breaking the sequence when parallel bottleneck modules are instantiated. Alternatively, additional logic may be implemented to perform re-sequencing.
6. When performing PR, the reset signal to the module being instantiated should be asserted to keep the module from transitioning to an unpredictable state once PR is completed.
7. The application should reconfigure the extra bottleneck module back to the "fast" module if all input has been processed by the parallel bottleneck modules, even if the FIFO full threshold has not been reached.

7.4 Contribution of Research

For this specific design, observing trends in execution is sufficient for determining an appropriate time for PR but it does not apply to two-stage slow-to-fast pipelines in general. Therefore as stated in the hypothesis of this research, a mathematical formula (*i.e.*, mathematical model) was developed for determining the value of performing PR upon the fast module, to improve the production of the slow module. The model accounts for reconfiguration time, execution times, and FIFO threshold values and is as follows:

$$
\begin{aligned}
&\text{let } tRC \ = \ \text{time to perform partial reconfiguration in clock cycles} \\
&\text{let } tBN \ = \ \text{execution time of the bottleneck model in clock cycles} \\
&\text{let } tPRM \ = \ \text{the worstcase execution time of the fast PRM in clock cycles} \\
&\text{let } FIFO_{full} \ = \ \text{the threshold for a FIFO full status} \\
&\text{let } FIFO_{empty} \ = \ \text{the threshold for a FIFO empty status} \\
\\
&nProd = \left\lceil \tfrac{tRC}{tBN} \right\rceil \\
&nPRM = \left\lfloor \tfrac{tBN}{tPRM} \right\rfloor \\
&nFull = FIFO_{full} - (nProd - FIFO_{empty}) \\
&nFill = nFull/2 \\
&T1 = (nFill \times tBN) + (nFull \times tPRM) + (2 \times tRC) \\
&T2 = (nFull \times tBN) + (nFull \times tPRM) \\
&result = \big(T2 - T1, \quad nPRM - \tfrac{T2}{T1}, \quad nFull - nProd\big)
\end{aligned}
\tag{7.1}
$$

In Equation 7.1 the value $nProd$ is the number of items produced by the bottleneck modules during reconfiguration. The value $nPRM$ is the number data items consumed from the FIFO by the PRM in the time it takes for the bottleneck module to produce a

single result. Value $nFull$ is the amount of items needed to reach the $FIFO_{full}$ threshold and $nFill$ is the number of items produced by the two parallel bottleneck modules. The value $T1$ is sum of the time to fill the FIFO with parallel bottleneck modules, the time for the PRM to consume the same number of total number of elements, and the PR cycle time. The value $T2$ is the time for $nFull$ items to pass through both stages of the pipeline. The result of this equation is a tuple of three values. These values can be either a positive number, a negative number, or zero. If the value is positive, then at least a single PR cycle is worth performing. If the value is negative or zero, then PR should not be performed. In addition, the farther the value is from zero, the higher the value of performing PR is beneficial (if positive) or not beneficial (if negative).

The $T2-T1$ subtraction is a comparison of the times taken to process a number of items placed into the FIFO when using PR, and the time taken to process the same number of items without PR. If the subtraction results in a negative number, then the processing time using the parallel bottleneck modules plus reconfiguration overhead will yield no benefit over the pipeline that does not use PR.

Value $nPRM - \frac{T2}{T1}$ is used to find baseline execution times between the bottleneck module and PRM were the producer-consumer relationship between them is 1-to-2. For every one data item produced in time tBN, two items are consumed by the PRM and results produced. Times $T1$ and $T2$ that result in a negative $nPRM - \frac{T2}{T1}$ value represent a producer-consumer relationship that is, or is approaching a 1-to-1 relationship and therefore has no benefit from using PR. For example, the Observed spectrum in Figure 7.1 with a tBN time that results in a negative value is the spectrum with 342 peaks; tBN is used to

calculate both $T1$ and $T2$. Therefore of the measured times in Figure 7.1, the 388-peak Observed spectrum represents the baseline 1-to-2 relationship between the bottleneck module and the PRM and the baseline $nPRM - \frac{T2}{T1}$ value is 0.514.

The value $nPRM - \frac{T2}{T1}$ accounts for items read from the FIFO that has been fully processed by the PRM in time tBN. The $nPRM - \frac{T2}{T1}$ value ignores any item read from FIFO that the PRM has not finished processing. For example, assume time tBN is 14 clock cycles and $tPRM$ is 6. In time tBN, the PRM would have removed three items from the FIFO but only produced results for two of them. The value $nPRM - \frac{T2}{T1}$ only accounts for the two items produced and ignores the progress made on the third item. To account for partially processed items, the floor calculation from $nPRM$ can be removed. Removing the floor will produce a real-number result for $nPRM$ where the fractional portion of the number accounts for the fraction of the item partially processed at the end of tBN.

Removing the floor from $nPRM$ no longer restricts $nPRM - \frac{T2}{T1}$ to return a negative value when the times $T1$ and $T2$ passes the baseline 1-to-2 producer-consumer relationship and begins to approach a 1-to-1 relationship. Therefore, removing the floor value is intended for designers that want to "min-max" their design parameters (*i.e.*, execution times, FIFO capacity and thresholds) by considering partially processed items. The 1-to-2 baseline value of $nPRM - \frac{T2}{T1}$ should be recorded first before removing the floor from $nPRM$.

Values lower than the baseline approach a 1-to-1 relationship. A negative value for $nPRM - \frac{T2}{T1}$, with the floor removed from $nPRM$, shows that PR is not worth performing. It should be noted, however, that PR is not worth performing on any 1-to-1 relationship

regardless of the value of $nPRM - \frac{T2}{T1}$ if it is below the 1-to-2 baseline. The benefit of removing the floor calculation from $nPRM$ is that it allows designers to set a $nPRM - \frac{T2}{T1}$ value—that accounts for partially processed items—above the baseline as a target and find times $T1$ and $T2$ that will achieve a new baseline. The larger the $nPRM - \frac{T2}{T1}$ value is when compared to the 1-to-2 baseline, the better expected performance for the pipeline when used with PR.

The subtraction $nFull - nProd$ accounts for the elements written to the FIFO during PR. If the number of items placed into the FIFO during PR is greater than the number of items placed into the FIFO (to reach the $FIFO_{full}$ threshold) by the parallel bottleneck modules, then it is not worth performing PR because the pipeline will stall when reconfiguring the parallel bottleneck module back to the fast PRM. This subtraction assumes that $FIFO_{full}$ is greater than half capacity of the FIFO. The subtraction also assumes that the user defines a $FIFO_{full}$ threshold where the delta between $FIFO_{full}$ and the maximum capacity of the FIFO is greater than $nProd$. The model does not account for the delta and this extension is reserved for future work.

In Tandem Smith-Waterman pipeline, the configuration and execution parameters for the baseline relationship is as follows:

- 64K Hypothetical database
- PPM $= 10$
- $tRC = 573000$ clock cycles; estimated PR time using ICAP
- $tBN = 150160$ clock cycles; 388-peak Observed spectrum
- $tPRM = 74200$ clock cycles; worst-case performance of SmWeng
- $FIFO_{full} = 819$; $\lfloor 80\% \rfloor$ of 1024 FIFO capacity

- $FIFO_{empty} = 102$; $\lfloor 10\% \rfloor$ of 1024 FIFO capacity

The model resulted in the following:

- $T2 - T1 = 52$ million clock cycle difference in processing time
- $nPRM - \frac{T2}{T1} = .0514$
- $nFull - nProd = 709$

All values are positive meaning that at this baseline configuration of the Tandem Smith-Waterman pipeline, it is worth performing PR for spectra in the Observed database that contain 388 peaks or greater; the 342 peak spectrum shown in Figure 7.1 resulted in a negative $nPRM - \frac{T2}{T1}$ value.

7.4.1 Resource Cost Analysis

To analyze the cost of resource utilization, a non-PR implementation of the Tandem Smith-Waterman pipeline was compared to the PR implementation. The non-PR implementation has two parallel Teng modules and a SmWeng module. The logic of the design stayed relatively the same as the PR design. The non-PR design still used the memory controllers as interfaces to the ROM databases, the database sizes did not change, and the PR controller was still used to distribute the Observed spectrum data between the parallel Teng modules. The change to the PR controller for the non-PR design was the removal of the FIFO monitoring logic and logic that signaled for the preparation of PR and signal PR ready. The border controller was also removed in the non-PR design since there was no need to throttle the input and output of the Teng and SmWeng modules. In both the

non-PR and PR designs, all ChipScope IP cores and logic used for monitoring the hardware was removed. Both designs were implemented using the Xilinx tools and bitstreams generated.

The bitstream of the non-PR design was 12.4MB, which is the same size as the PR design (both the parallel Teng configuration and the single Teng, single SmWeng configuration). The similarity in bitstream sizes explains how Xilinx generates bitstreams for a PR design. In a PR design, the PRR contains the logic of the PRM. When uploading a bitstream, the resources in the PRR must be initialized regardless of whether or not a PRM is loaded into the PRR. When a Teng or SmWeng module is loaded to the PRR, the remaining unused resources must be given a configuration that makes them inactive. Therefore, the 12.4MB bitstream size for the PR design, includes the configuration data of the resources in the PRR that are unused; this was explained earlier in Chapter 6.3. In addition to unused resources in the PRR being configured, the bitstreams are restricted to FPGA reconfigurable frame boundaries. Therefore, bitstreams are padded if they do not meet reconfigurable frame boundaries. This also explains the similarity in bitstream size for the non-PR and PR implementations.

The bitstream is a high-level indication of FPGA resource utilization, *i.e.*, the more resources used, the larger the bitstream. However since the non-PR and PR design share the same bitstream size, a lower level resource comparison is used. Table 7.1 shows the resource utilization result from the place-and-route report generated by the Xilinx implementation tool. Place-and-route (PAR) is the process that determines a placement of hardware design components on the FPGA fabric and routes for the signals that connect them.

The PAR report provides the most accurate representation of the resource utilization for a hardware design.

Table 7.1

Resource utilization comparison between a non-PR and PR design of the Tandem Smith-Waterman pipeline

Resource	non-PR	PR	Difference
slice registers	9187	6425	~42%
slice LUTs	17192	13269	~29%
occupied slices	6660	6057	~9%
LUT Flip-Flop pairs used	19526	14913	~30%
bonded IO buffers	123	116	~6%
RAMB36/FIFO36	380	376	~1%
DSP48	64	64	0

In Table 7.1 the 376 RAMB36/FIFI36 resource utilization of the PR design includes nine RAMB36 memories in the PRR that were not used by the Teng and SmWeng PRMs. The PRR must contain the required resources for all PRMs to be instantiated within. When being placed on the FPGA it is typical that the PRR will include resources that are not required by the PRMs. These unused resources count toward utilization because all resources in the PRR are unavailable to modules outside of the PRR, which limits the resources available for static logic.

The table shows that there is not much of a difference in resource utilization between a non-PR Tandem Smith-Waterman design and the PR design, Therefore, the non-PR design may be the better option *if* the target FPGA logic can support all instantiated Teng and

SmWeng modules. However, in the case of a two-stage slow-to-fast non-PR pipeline where instantiating all modules consume more resources than what the target FPGA can support, then PR can be used to swap in modules and resource sharing PR used to improved the performance of the bottleneck module.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In a traditional hardware application, all functional components (*i.e.*, modules) are instantiated on the FPGA. The effect is that as more functions are added to the application, more FPGA logic resources are needed to realize these function. In addition, more routing resources are required to connect the functions resulting in complex circuits that affect the application execution frequency (due to propagation delay overhead). The Xilinx partial reconfiguration (PR) design flow introduces a basic partial reconfiguration design technique (basic PR) for hardware applications. The basic PR design objective is to reduce the application time overheads (frequency and propagation delays) by implementing the functional components of the application as interchangeable partial reconfigurable modules (PRMs) and then exchanging them, when a change in function is necessary, by loading the required function onto the FPGA via PR.

The basic PR design flow focuses on exchanging application functions to avoid the overhead of having all functions instantiated (a non-PR design). Basic PR does not consider using PR to improve the performance of the application. This research introduced a resource-sharing advanced partial reconfiguration design technique. Resource sharing PR extends the Xilinx basic PR design flow by using partial reconfiguration to improve

the performance of a hardware application. High performance FPGA-based applications are typically implemented as interacting parallel modules consisting of high-throughput pipelines [47]. These pipelines typically have a bottleneck module that dictates its throughput. Resource sharing PR improves the pipeline throughput by reallocating the resources used by an idle waiting PRM and repurposing those resources to instantiate an additional parallel bottleneck module (Chapter 4.2). The effect is a broken pipeline but an improved production of the bottleneck pipeline stage.

Resource sharing PR is performed upon a pipeline in two steps: 1) the reallocation of the idle waiting PRM resources to realize an additional parallel bottleneck module, and 2) once enough data has been processed, the reconfiguration of these resources back to the original PRM. This two step process is called a PR cycle. The target pipeline of this research is a two-stage "slow-to-fast" pipeline where the data flow through this pipeline transitions from a "slow" bottleneck module to a relatively "fast" module and the connector between the modules is a FIFO used as a buffer. In this pipeline the PR cycle will reconfigure the fast module into an additional bottleneck module and once a certain FIFO capacity threshold is reached (approximately 80%), the bottleneck module is reconfigured back to the fast module.

For experimentation, a resource sharing partial-reconfigurable hardware application was implemented. The architectural design of the application is a two-stage slow-to-fast pipeline with a FIFO buffer. The stages in the pipeline were hardware implementations of two known Bioinformatics search algorithms, the X! Tandem protein identification algorithm and the Smith-Waterman sequence matching algorithm (Chapter 5.8). These

algorithms were specifically chosen because of their performance in execution. Smith-Waterman is a simple algorithm and was selected as the fast algorithm. In comparison to Smith-Waterman, the X! Tandem algorithm is complex and slow in production.

The X! Tandem hardware engine, Teng, identifies proteins using the peptides scanned from a biological mixture. The *scan* is represented as a spectrum of peptide masses with corresponding charge intensity values. The spectrum size is the number of peaks (peptide mass-intensity pair) that exists in the spectrum. What protein the spectrum represents is unknown. The spectrum is identified by matching it against spectra in a database of known proteins. During this process, the entire known protein database is searched by Teng.

The Smith-Waterman hardware engine, SmWeng, attempts to find similarities in function between several known proteins and the Teng identified protein by using the characteristics of the known proteins and the characteristics of the Teng identified protein. In the Tandem Smith-Waterman pipeline, SmWeng was designated as the PRM. The worst-case performance of SmWeng was used in the analysis of the experimental data (Chapter 7). The SmWeng worst-case performance and the performance of Teng were used to derive a mathematical model for determining the value of performing PR—the hypothesis of this research. Resource sharing PR was used to reconfigure SmWeng for an additional Teng and then back once an 80% $FIFO_{full}$ threshold was met.

Partial reconfiguration was performed manually using the JTAG interface when the pipeline signaled ready for PR. The experimental results revealed a linear relationship between the known spectra database size and the search time for Teng. This showed that as the database increases, the Teng search time will also increase; the database, however, will

remain fixed in size once selected but it should be noted that its size does affect the search time.

Lastly in this research, a mathematical PR model was derived for pipelines where data traversing two interconnected modules transitions from a slow processing module to a relatively fast processing module (Chapter 7.4). The model measures the value of executing a PR cycle upon two-stage slow-to-fast pipelines; extending the model for longer pipelines that contain the same transition between two stages is reserved for future work. The model was created to aid designers in determining whether PR is worth performing. The model was derived by analyzing the trend in the execution times of Teng.

For a given fixed-size database of known protein spectra, the trend is that the Teng identification process times follow a linear relationship with the size of the unidentified spectrum input for Teng. Using this trend a 1-to-2 performance baseline was identified meaning for every one data item produced by Teng, two items are consumed by SmWeng and results produced. The hardware parameters (bottleneck and PRM execution times, estimated reconfiguration time, FIFO full/empty thresholds) at this baseline where used to derive a general PR model for two-stage slow-to-fast pipelines. The result of the model is three values that represent the time to process items placed in the FIFO, the producer-consumer relationship between the bottleneck and PRM, and a contrast between items placed in the FIFO during PR and items placed in the FIFO by the parallel bottleneck modules. If any one of these values are zero or less, then there is not a benefit in performing resource sharing PR.

8.1 Future work

Although this research shows cases where PR is valuable, there are some limitations to this work. Future research could include the modifications below:

- Extend the mathematical model to account for the return reconfiguration using the maximum capacity of the FIFO.
  - This extension considers the comparison $nProd \geq (FIFO_{capacity} - FIFO_{full})$. If true, then PR is not worth performing because on the the return of the PR cycle—where the parallel bottleneck module is reconfigured back to the fast module—the FIFO will fill to capacity and cause the pipeline to stall.
- Implement a fully autonomous design that uses the ICAP interface to perform partial reconfiguration.
  - The PR controller is designed to operate autonomously using the programmable full and programmable empty FIFO signals to determine when PR should be executed. The next step is to implement a module that will connect the ICAP interface to the storage location of the partial bitstreams.
- Move all databases to RAM and incorporate the Xilinx memory controller into the Tandem Smith-Waterman design.
  - The address counter of the current memory modules will need to be modified to support the DDR memory address scheme.
- To minimize read delays when accessing the databases in DDR memory, buffer (or store) future input data while the the pipeline is processing other data.
  - This is currently done for the Observed data using the two banks if the ScanStorage memory module. While one Observed spectrum is being processed, another is being loaded into the second bank. In a future revision, all incoming data (Hypothetical and SmWeng) should be stored this way.
- Extend the mathematical model to support longer pipelines
  - In its current state, the model should theoretically support longer pipelines where a two-stage slow-to-fast pipeline transition exists. Verify that the model supports this with little change.
  - Modify the model to support the option of using multiple fast modules to generate additional parallel bottleneck modules (more than just one).
- Enable the design to be used as a CPU co-processor.

- The current design targets an FPGA development board in stand-alone mode. A future change will be to extend the design to interact with a CPU and respond to instructions given by the CPU.

These modifications to the PR model and the Tandem Smith-Waterman pipeline are planned to be implemented in future journal and conference publications.

REFERENCES

[1] Altera Corp., "Stratix II GX Device Handbook (SIIGX5V1 v4.4),", http://www.altera.com/, June 2009.

[2] Altera Corp., "Stratix III Device Handbook (SIII5V1 v2.2),", http://www.altera.com/, March 2011.

[3] Altera Corp., "Stratix IV Device Handbook (SIV5V1 v4.6),", http://www.altera.com/, September 2012.

[4] Altera Corp., "Quartus II Handbook Version 13.0 Volume 1: Design and Synthesis (QII5V1-13.0.0),", http://www.altera.com/, May 2013.

[5] Altera Corp., "Stratix V Device Handbook (SV5V1 v2013.06.21),", http://www.altera.com/, June 2013.

[6] Altera Corporation, *Accelerating High-Performance Computing With FPGAs*, Tech. Rep. WP-01029-1.1, Altera Corporation, 2007.

[7] Altera Corporation, *Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform*, Tech. Rep. WP-01035-1.0, Altera Corporation, September 2007.

[8] Altera Corporation, "Altera Corporation Website,", http://www.altera.com/, 2013.

[9] E. K. Anderson, M. French, and D.-I. Kang, "System on a Programmable Chip Adaptation Through Active Partial Reconfiguration," *Proceedings of the 2008 International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA 2008, Las Vegas, Nevada, USA*, July 2008, pp. 104–110.

[10] R. D. Anderson and Y. S. Dandass, "A Protocol for Realtime Switched Communication in FPGA Clusters," *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, Las Vegas, Nevada, USA, July 2011.

[11] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration," *Proceedings of the 42nd annual Design Automation Conference*, New York, NY, USA, June 2005, DAC'05, pp. 335–340, ACM.

[12] Z. Baruch, O. Cret, and K. Pusztai, "An Efficient Sequence to Apply Slicing Lines in Fpga Placement,", `http://users.utcluj.ro/~baruch/papers/Slicing-Placement.pdf`, July 2013.

[13] L. Bossuet, G. Gogniat, and W. Burleson, "Dynamically Configurable Security for SRAM FPGA Bitstreams," *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004, p. 146.

[14] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A Massively Parallel FPGA-based Coprocessor for Support Vector Machines," *Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'09)*, April 2009, pp. 115–122.

[15] C. Choi and H. Lee, "An Reconfigurable FIR Filter Design on a Partial Reconfiguration Platform," *First International Conference on Communications and Electronics (ICCE'06)*, October 2006, pp. 352–355.

[16] C. Conger, A. Gordon-Ross, and A. D. George, "Design Framework for Partial Run-Time FPGA Reconfiguration," *Proceedings of the 2008 International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA 2008, Las Vegas, Nevada, USA*, July 2008, pp. 122–128.

[17] Crucial, "DDR3 SDRAM SODIMM,", `http://forum.crucial.com/crucial/attachments/crucial/dram/11012/1/DDR3\%20SDRAM\%20SODIMM.pdf`, June 2016.

[18] A. Dandalis and V. Prasanna, "Configuration compression for FPGA-based embedded systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 12, 2005, pp. 1394–1398.

[19] A. Dandalis and V. K. Prasanna, "Configuration Compression for FPGA-based Embedded Systems," *Proceedings of the 2001 ACM/SIGDA ninth International Symposium on Field Programmable Gate Arrays*, New York, NY, USA, 2001, FPGA '01, pp. 173–182, ACM.

[20] Y. S. Dandass, S. C. Burgess, M. Lawrence, and S. M. Bridges, "Accelerating String Set Matching in FPGA Hardware for Bioinformatics Research," *BMC Bioinformatics*, vol. 9, no. 1, 2008, p. 197.

[21] J. Delorme, J. Martin, A. Nafkha, C. Moy, F. Clermidy, P. Leray, and J. Palicot, "A FPGA Partial Reconfiguration Design Approach for Cognitive Radio Based on NoC Architecture," *2008 Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference (NEWCAS-TAISA'08)*, June 2008, pp. 355–358.

[22] J. L. Devore, *Probability and Statistics for Engineering and the Sciences*, first edition, Brooks/Cole Publishing Company, Monterey, California, 1982.

[23] I. Eidhammer, K. Flikka, L. Martens, and S. Mikalsen, *Computational Methods for Mass Spectrometry Proteomics*, first edition, John Wiley & Sons Ltd., England, 2007.

[24] W. M. El-Medany and M. R. Hussain, "FPGA-Based Advanced Real Traffic Light Controller System Design," *4th IEEE Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS'07)*, September 2007, pp. 100–105.

[25] G. Estrin, "Organization of Computer Systems–The Fixed Plus Variable Structure Computer," *1960 Proceedings of the Western Joint Computer Conference*, San Fransisco, California, USA, September 1960, `http://doi.ieeecomputersociety.org/10.1109/AFIPS.1960.28`, pp. 33–40.

[26] E. Eto, "Difference-Based Partial Reconfiguration (XAPP290 v2.0),", `http://www.xilinx.com/`, December 2007.

[27] F. Ferrandi, M. D. Santambrogio, and D. Sciuto, "A design methodology for dynamic reconfiguration: the Caronte architecture," *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005, pp. 4 pp.–.

[28] C. Fobel, G. Grewal, and A. Morton, "Using Hardware Acceleration to Reduce FPGA Placement Times," *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, 2007, pp. 647–650.

[29] M. Fons, F. Fons, and E. Cant, "Fingerprint Image Processing Acceleration Through Run-Time Reconfigurable Hardware," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 57, no. 12, December 2010, pp. 991–995.

[30] R. Garcia, A. Gordon-Ross, and A. D. George, "Exploiting Partially Reconfigurable FPGAs for Situation-Based Reconfiguration in Wireless Sensor Networks," *Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'09)*, April 2009, pp. 243–246.

[31] O. Gotoh, "An improved algorithm for matching biological sequences.," *Journal of Molecular Biology*, vol. 162, no. 3, 1982, pp. 705–708.

[32] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, second edition, Pearson Education Limited, Harlow, England, 2003.

[33] H. Gu and S. Chen, "Partial Reconfiguration Bitstream Compression for Virtex FPGAs," *Image and Signal Processing, 2008. CISP '08. Congress on*, 2008, vol. 5, pp. 183–185.

[34] V. Gudise and G. Venayagamoorthy, "FPGA placement and routing using particle swarm optimization," *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, 2004, pp. 307–308.

[35] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, New York, NY, USA, February 1998, FPGA'98, pp. 65–74, ACM.

[36] R. Hecht, S. Kubisch, A. Herrholtz, and D. Timmermann, "Dynamic reconfiguration with hardwired networks-on-chip on future FPGAs," *Field Programmable Logic and Applications, 2005. International Conference on*, 2005, pp. 527–530.

[37] HiTech Global, "HiTech Global Retailer Website,", `http://www.hitechglobal.com`, June 2016.

[38] Y. Hori, A. Satoh, H. Sakane, and K. Toda, "Bitstream Encryption and Authentication with AES-GCM in Dynamically Reconfigurable Systems," *International Conference on Field Programmable Logic and Applications (FPL'08)*, September 2008, pp. 23–28.

[39] J. S. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook, "Dynamic reconfiguration to support concurrent applications," *Computers, IEEE Transactions on*, vol. 48, no. 6, 1999, pp. 591–602.

[40] R. Jia, F. Wang, R. Chen, X.-G. Wang, and H.-G. Yang, "JTAG-based bitstream compression for FPGA configuration," *Solid-State and Integrated Circuit Technology (ICSICT), 2012 IEEE 11th International Conference on*, 2012, pp. 1–3.

[41] H. C. Leligou, L. Redondo, T. Zahariadis, and D. R. Retamosa, "Reconfiguration in Wireless Sensor Networks," *Developments in E-systems Engineering (DESE)*, September 2010, pp. 59–63.

[42] Z. Li and S. Hauck, "Configuration Compression for Virtex FPGAs," *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, 2001, pp. 147–159.

[43] D. Lim and M. Peattie, "Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations (XAPP290 v1.0),", `http://www.xilinx.com/`, May 2002.

[44] Microsemi, "Microsemi SoC Products Group,", `http://www.actel.com`, 2011.

[45] G. R. Morris, R. D. Anderson, and V. K. Prasanna, "An FPGA-Based Application-Specific Processor for Efficient Reduction of Multiple Variable-Length Floating-Point Data Sets," *Proceedings of the 17th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'06)*, Steamboat Springs, Colorado, USA, September 2006, http://dx.doi.org/10.1109/ASAP.2006.11, pp. 323–330.

[46] G. R. Morris, R. D. Anderson, and V. K. Prasanna, "A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer," *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Napa, California, USA, April 2006, http://dx.doi.org/10.1109/FCCM.2006.8, pp. 3–12.

[47] G. R. Morris and V. K. Prasanna, "A Hybrid Approach for Accelerating a Sparse Matrix Jacobi Solver using an FPGA-Augmented Reconfigurable Computer," *Proceedings of the 9th Military and Aerospace Programmable Logic Devices Conference (MAPLD'06)*, Washington DC, USA, September 2006, http://klabs.org/mapld06/sessions/b.html.

[48] S. Mühlbach and A. Koch, "A Dynamically Reconfigured Network Platform for High-Speed Malware Collection," *2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, December 2010, pp. 79–84.

[49] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, 2004, pp. 766–773.

[50] J. Rodríguez-Araújo, J. J. Rodríguez-Andina, J. F. na, and M. Chow, "Field-Programmable System-on-Chip for Localization of UGVs in an Indoor iSpace," *Industrial Informatics, IEEE Transactions on*, vol. 10, no. 2, May 2014, pp. 1033–1043.

[51] M. Rubio-Solar, M. Vega-Rodriguez, J. Perez, A. Gomez-Iglesias, and M. Cardenas-Montes, "A FPGA Optimization Tool Based on a Multi-island Genetic Algorithm Distributed over Grid Environments," *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, 2008, pp. 65–72.

[52] B. C. Searle, "X!Tandem Explained," *Rapid Communications in Mass Spectrometry*, vol. 17, 2003, pp. 2310–2316.

[53] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *Computers and Digital Techniques, IEE Proceedings* -, vol. 153, no. 3, 2006, pp. 157–164.

[54] B. Sellers, J. Heiner, M. Wirthlin, and J. Kalb, "Bitstream compression through frame removal and partial reconfiguration," *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009, pp. 476–480.

[55] J. E. Sim, W.-F. Wong, G. Walla, T. Ziermann, and J. Teich, "Interprocedural Placement-Aware Configuration Prefetching for FPGA-Based Systems," *Proceedings of 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'10)*, May 2010, pp. 179–182.

[56] T. Smith and M. Waterman, "Identification of common molecular subsequences.," *Journal of Molecular Biology*, vol. 147, no. 1, 1981, pp. 195–197.

[57] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, October 1982, pp. 928–951.

[58] P. Sundararajan, *High Performance Computing Using FPGAs*, Tech. Rep. WP375 (v1.1), Xilinx Inc., September 2010.

[59] The Global Proteome Machine Organization, "X! TANDEM Spectrum Modeler,", `http://www.thegpm.org/`, December 2013.

[60] W. Vanderbauwhede and K. Benkrid, eds., *High-Performance Computing Using FPGAs*, first edition, Springer, New York, 2013, FPGA and HPC: Used frontmatter made available for free in electric form online.

[61] T. Wollinger, J. Guajardo, and C. Paar, "Security on FPGAs: State-of-the-Art Implementations and Attacks," *ACM Transactions on Embedded Computing Systems*, vol. 3, August 2004, pp. 534–574.

[62] K. Wu and J. Madsen, "Run-time Dynamic Reconfiguration: a Reality Check Based on FPGA Architectures From Xilinx," *23rd NORCHIP Conference*, November 2005, pp. 192–195.

[63] Xilinx Inc., "Early Access Partial Reconfiguration User Guide, For ISE 8.1.01i (UG208 v1.1),", `http://www.xilinx.com/`, March 2006.

[64] Xilinx Inc., "Virtex-4 FPGA Configuration User Guide(UG071 v1.11),", `http://www.xilinx.com/`, June 2009.

[65] Xilinx Inc., "Virtex-5 FPGA User Guide (UG190 v5.2),", `http://www.xilinx.com/`, November 2009.

[66] Xilinx Inc., "Virtex-5 FPGA Configuration User Guide(UG191 v3.9.1),", `http://www.xilinx.com/`, August 2010.

[67] Xilinx Inc., "Virtex-5 FPGA XtremeDSP Design Considerations User Guide (UG193 v3.4),", `http://www.xilinx.com/`, June 2010.

[68] Xilinx Inc., "Virtex-6 FPGA Configuration User Guide(UG360 v3.0),", `http://www.xilinx.com/`, January 2010.

[69] Xilinx Inc., "Command Line Tools User Guide (UG628 v13.3),", http://www.xilinx.com/, October 2011.

[70] Xilinx Inc., "Xilinx University Program Partial Reconfiguration Flow Presentation Manual,", http://www.xilinx.com/university/workshops/partial-reconfiguration-flow, 2011.

[71] Xilinx Inc., "Xilinx University Program (XUP) Partial Reconfiguration Flow Workshop and Teaching Materials,", http://www.xilinx.com/university/workshops/partial-reconfiguration-flow, 2011.

[72] Xilinx Inc., "Partial Reconfiguration User Guide (UG702 v14.5),", http://www.xilinx.com/, April 2013.

[73] Xilinx Inc., "PlanAhead User Guide (UG632 v14.6),", http://www.xilinx.com/, June 2013.

[74] Xilinx Inc., "Xilinx Incorporated Website,", http://www.xilinx.com, 2013.

[75] Xilinx Inc., "XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices (UG627 v14.5),", http://www.xilinx.com/, March 2013.

[76] M. Yang, A. Almaini, and P. Wang, "FPGA placement optimization by two-step unified Genetic Algorithm and Simulated Annealing algorithm," *Journal of Electronics (China)*, vol. 23, no. 4, 2006, pp. 632–636.

[77] M. Yang, A. E. A. Almaini, L. Wang, and P. Wang, "FPGA placement using genetic algorithm with simulated annealing," *ASIC, 2005. ASICON 2005. 6th International Conference On*, 2005, vol. 2, pp. 806–810.

[78] D. Yin, D. Unnikrishnan, Y. Liao, L. Gao, and R. Tessier, "Customizing Virtual Networks with Partial FPGA Reconfiguration," *ACM SIGCOMM Computer Communication Review*, vol. 41, 2011, pp. 125–132.

[79] A. S. Zeineddini and K. Gaj, "Secure Partial Reconfiguration of FPGAs," *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, December 2005, pp. 155–162.

[80] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, 1977, pp. 337–343.

APPENDIX A

ILLUSTRATION OF THE TANDEM SMITH-WATERMAN PIPELINE

Figure A.1 is a complete pipeline of the Tandem Smith-Waterman pipeline. This image shows the Hypothetical, Observed and SmWeng databases and all the controllers involved in the preparation and invocation of dynamic partial reconfiguration.

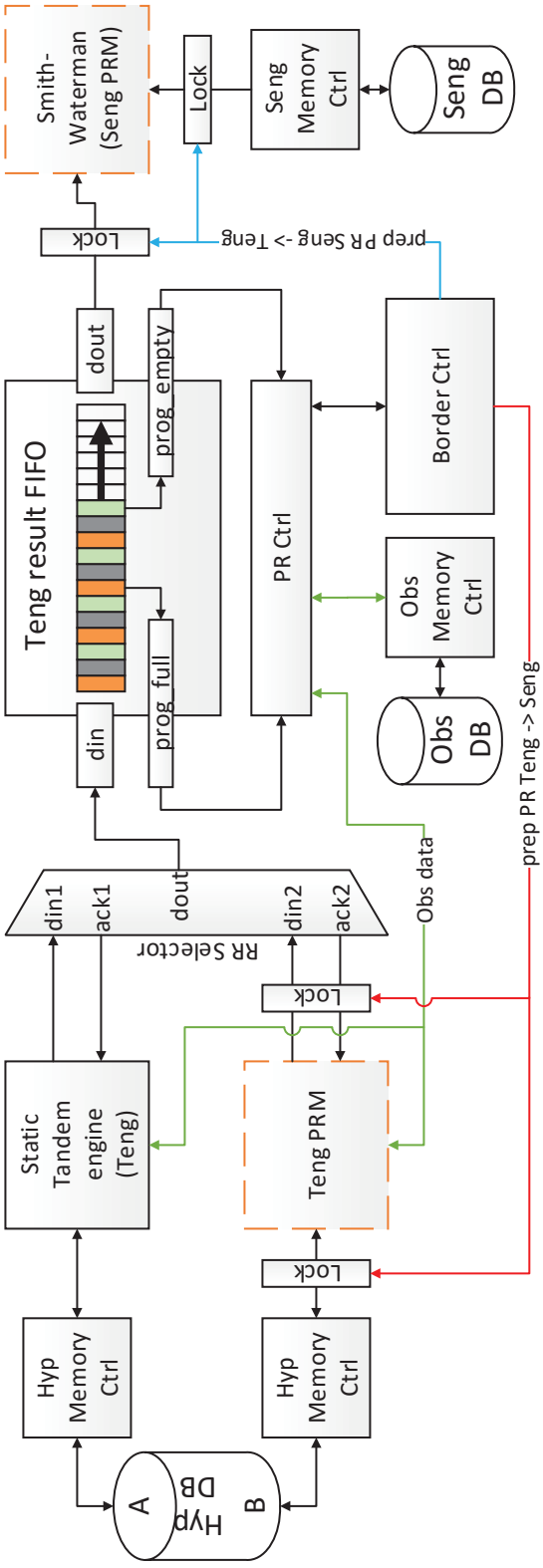

Figure A.1

Architecture of the Tandem Smith-Waterman pipeline