Reconfigurable Technologies for Next Generation Internet and Cluster Computing

Deepak C. Unnikrishnan

University of Massachusetts - Amherst, deepak.cu@gmail.com

Follow this and additional works at: http://scholarworks.umass.edu/open_access_dissertations

Recommended Citation
Unnikrishnan, Deepak C., "Reconfigurable Technologies for Next Generation Internet and Cluster Computing" (2013). Dissertations.
Paper 823.

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.
RECONFIGURABLE TECHNOLOGIES FOR NEXT GENERATION INTERNET AND CLUSTER COMPUTING

A Dissertation Presented

by

DEEPAK UNNIKRISHNAN

Submitted to the Graduate School of the University of Massachusetts Amherst in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2013

Electrical and Computer Engineering
RECONFIGURABLE TECHNOLOGIES FOR
NEXT GENERATION INTERNET AND CLUSTER
COMPUTING

A Dissertation Presented
by
DEEPAK UNNIKRISHNAN

Approved as to style and content by:

________________________________________
Russell G. Tessier, Chair

________________________________________
Lixin Gao, Member

________________________________________
Eric Polizzi, Member

________________________________________
Arun Venkataramani, Member

________________________________________
C. V. Hollot, Department Chair
Electrical and Computer Engineering
ACKNOWLEDGMENTS

First and foremost, I owe my deepest gratitude to my adviser Prof. Russell Tessier. Prof. Tessier was kind enough to accept me as a fresh first-year graduate student into his research group. Over the last six years, he has not only instilled in me, a strong appreciation for computer systems research, but his constant encouragement and constructive criticism has immensely helped improve my writing and presentation skills. I sincerely thank him for the opportunities and confidence he has given me all these years.

It is an honor and privilege for me to be able to work with an accomplished researcher as Prof. Lixin Gao. Prof. Gao has served the role of an informal adviser in my Ph.D and her ideas have gone a long way in shaping the contents of this thesis. I thank her for the valuable guidance. I sincerely thank committee members, Prof. Eric Polizzi and Prof. Arun Venkataramani for critiquing my work and providing constructive guidelines to make this dissertation better.

I am grateful to the National Science Foundation for funding my research. I thank Xilinx and Altera for equipment donations. Acknowledgments are due to the Department of Electrical and Computer Engineering for providing me with a graduate assistantship award to pursue higher studies in USA.

I thank Dr. Andrew Leaver, Thiagaraja Gopalsamy, Gurvinder Tiwana and Paul Leventis for hosting me at Altera Corporation for two internships. I have enjoyed the company of several truly amazing people during my stay.

The members of the Reconfigurable Computing Group will always remain dear to my heart. Special thanks to Sailaja Madduri and Ramakrishna Vadlamani for
introducing me to the group. They have been great friends all this time. I thank fellow graduate students in the group - Justin Lu, Kekai Hu, Sandesh Virupaksha, Xiaobin Liu and Cory Gorman for their company. I thank ex-members of the group - Jia Zhao, Vishwas Vijayendra, Gayatri Prabhu, Akilesh Krishnamurthy, Salma Mirza, Murtaza Merchant, Harikrishnan, Emmanuel Seguin and Ben Bovee, without whom all these six years would not have been more enjoyable. Many of them have made valuable contributions towards the work in this thesis.

Doing a Ph.D. would have been difficult without the support of my parents and my brother. I thank them for their love and support. I thank our family friend, Kuberan, for providing me valuable guidance during the graduate application process.

My wife Lekshmi has been a constant source of love, encouragement and support all these years. Despite being a graduate student herself, she has managed our personal lives so well. Her selfless effort has contributed towards the making of this thesis in the form of numerous brainstorming sessions, assistance with experiments and help with the writing. I thank her from my heart for the love, hard work and patience.

I consider myself very fortunate to have spent six wonderful years of my life in the beautiful college town of Amherst. Amherst will be greatly missed in my life.
ABSTRACT

RECONFIGURABLE TECHNOLOGIES FOR NEXT GENERATION INTERNET AND CLUSTER COMPUTING

SEPTEMBER 2013

DEEPAK UNNIKRISHNAN
M.S., UNIVERSITY OF MASSACHUSETTS, AMHERST
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell G. Tessier

Modern web applications are marked by distinct networking and computing characteristics. As applications evolve, they continue to operate over a large monolithic framework of networking and computing equipment built from general-purpose microprocessors and Application Specific Integrated Circuits (ASICs) that offers few architectural choices. This dissertation presents techniques to diversify the next-generation Internet infrastructure by integrating Field-programmable Gate Arrays (FPGAs), a class of reconfigurable integrated circuits, with general-purpose microprocessor-based techniques. Specifically, our solutions are demonstrated in the context of two applications - network virtualization and distributed cluster computing.

Network virtualization enables the physical network infrastructure to be shared among several logical networks to run diverse protocols and differentiated services.
The design of a good network virtualization platform is challenging because the physical networking substrate must scale to support several isolated virtual networks with high packet forwarding rates and offer sufficient flexibility to customize networking features. The first major contribution of this dissertation is a novel high-performance heterogeneous network virtualization system that integrates FPGAs and general-purpose CPUs. Salient features of this architecture include the ability to scale the number of virtual networks in an FPGA using existing software-based network virtualization techniques, the ability to map virtual networks to a combination of hardware and software resources on demand, and the ability to use off-chip memory resources to scale virtual router features. Partial-reconfiguration has been exploited to dynamically customize virtual networking parameters. An open software framework to describe virtual networking features using a hardware-agnostic language has been developed. Evaluation of our system using a NetFPGA card demonstrates one to two orders of improved throughput over state-of-the-art network virtualization techniques.

The demand for greater computing capacity grows as web applications scale. In state-of-the-art systems, an application is scaled by parallelizing the computation on a pool of commodity hardware machines using distributed computing frameworks. Although this technique is useful, it is inefficient because the sequential nature of execution in general-purpose processors does not suit all workloads equally well. Iterative algorithms form a pervasive class of web and data mining algorithms that are poorly executed on general purpose processors due to the presence of strict synchronization barriers in distributed cluster frameworks. This dissertation presents Maestro, a heterogeneous distributed computing framework that demonstrates how FPGAs can break down such synchronization barriers using asynchronous accumulative updates. These updates allow for the accumulation of intermediate results for numerous data points without the need for iteration-based barriers. The benefits of
a heterogeneous cluster are illustrated by executing a general-class of iterative algorithms on a cluster of commodity CPUs and FPGAs. Computation is dynamically prioritized to accelerate algorithm convergence. We implement a general-class of three iterative algorithms on a cluster of four FPGAs. A speedup of 7× is achieved over an implementation of asynchronous accumulative updates on a general-purpose CPU. The system offers 154× speedup versus a standard Hadoop-based CPU-workstation cluster. Improved performance is achieved by clusters of FPGAs.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiv</td>
</tr>
</tbody>
</table>

## CHAPTER

1. INTRODUCTION .................................................. 1
   1.1 Trends and Challenges in Future Internet Systems ............. 1
   1.2 Thesis Statement ............................................. 4
   1.3 Thesis Overview ............................................. 4
   1.4 Applications Considered in the Thesis ........................ 6
   1.5 Thesis Outline and Preview of Results ........................ 8

2. BACKGROUND .................................................. 12
   2.1 Overview of FPGA Technology ................................ 12
   2.2 FP GAS in Networking and Computing Infrastructure ........... 14
   2.3 Network Virtualization ....................................... 15
   2.4 Cluster Computing - Architecture and Programming Model ....... 24

3. NETWORK VIRTUALIZATION USING FPGAS ....................... 29
   3.1 Review of FPGA-based Virtualization Platforms ............... 30
   3.2 System Design ............................................... 31
      3.2.1 Design Goals ........................................... 31
      3.2.2 NetFPGA ............................................... 33
      3.2.3 Architecture Overview .................................. 34
      3.2.4 Packet Forwarding ..................................... 36
      3.2.5 Hardware Data Planes .................................. 36
3.2.6 Software Data Planes .............................................. 38
3.3 Data Plane Scaling and Virtual Network Migration ................. 39
  3.3.1 Single-receiver Approach ........................................ 40
  3.3.2 Multi-receiver Approach .......................................... 41
  3.3.3 Scheduling Virtual Network Migration ............................ 43
3.4 Case Study - Data Planes ............................................. 44
  3.4.1 IPv4 .................................................................... 44
  3.4.2 Routing on Flat Labels .......................................... 45
3.5 Scalability Considerations - Virtual Forwarding Tables .......... 46
  3.5.1 Related Work ...................................................... 46
  3.5.2 Design Challenges .................................................. 47
  3.5.3 Shared Virtual Forwarding Table Design ......................... 49
  3.5.4 Handling Virtual Prefix Conflicts .................................. 50
3.6 Evaluation ................................................................. 54
  3.6.1 Throughput ........................................................... 55
  3.6.2 Latency ............................................................... 56
  3.6.3 Network Scalability ................................................. 58
  3.6.4 Overhead of Dynamic Reconfiguration ......................... 62
  3.6.5 Frequency of Dynamic Reconfiguration ......................... 62
  3.6.6 Cost Analysis ....................................................... 63
  3.6.7 Resource Usage ..................................................... 64
  3.6.8 Size of Conflict CAM ............................................. 65
3.7 Conclusion ................................................................. 66
4. CUSTOMIZING VIRTUAL NETWORKS WITH PARTIAL
  RECONFIGURATION ...................................................... 68
  4.1 Introduction ................................................................... 68
  4.2 Background on Partial Reconfiguration ............................ 70
  4.3 A Partially Reconfigurable Network Virtualization Platform ...... 71
    4.3.1 Partial Bitstream Generation ...................................... 73
    4.3.2 Host Virtualization ............................................... 75
    4.3.3 Dynamic Virtual Network Allocation ............................ 75
  4.4 Experimental Approach ................................................ 76
    4.4.1 Testbed Setting .................................................... 76
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Dataplane latency for IPv4 and ROFL</td>
<td>58</td>
</tr>
<tr>
<td>3.2 Cost and throughput for virtual networking systems</td>
<td>64</td>
</tr>
<tr>
<td>3.3 Resource utilization of IPv4 and ROFL data planes</td>
<td>65</td>
</tr>
<tr>
<td>3.4 Percentage of prefixes which overlap</td>
<td>66</td>
</tr>
<tr>
<td>4.1 Experimental configurations</td>
<td>75</td>
</tr>
<tr>
<td>4.2 Dynamic power consumption in a Virtex II device</td>
<td>84</td>
</tr>
<tr>
<td>5.1 Feature comparison of FPGA programming models</td>
<td>89</td>
</tr>
<tr>
<td>5.2 ReClick primitives</td>
<td>97</td>
</tr>
<tr>
<td>5.3 Resource Utilization and Latency of ReClick components</td>
<td>103</td>
</tr>
<tr>
<td>5.4 Resource utilization of ReClick IPv4 and onion router dataplanes</td>
<td>106</td>
</tr>
<tr>
<td>6.1 List of iterative algorithms</td>
<td>130</td>
</tr>
<tr>
<td>6.2 Speedup of Maiter versus Hadoop for 1, 2, and 4 workers</td>
<td>131</td>
</tr>
<tr>
<td>6.3 Maestro execution time for varying problem and cluster size</td>
<td>140</td>
</tr>
<tr>
<td>6.4 Network traffic volume in a 2 worker cluster</td>
<td>140</td>
</tr>
<tr>
<td>6.5 Resource utilization on a Stratix IV FPGA</td>
<td>142</td>
</tr>
<tr>
<td>6.6 Energy/cost estimates for a 4 worker cluster executing PageRank</td>
<td>142</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Layered networking model in the Internet</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A virtualized network</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>FPGA architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>FPGA application development flow</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>A virtualized physical network</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Virtual router architecture</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Summary of host virtualization techniques</td>
<td>21</td>
</tr>
<tr>
<td>2.6</td>
<td>Cluster organization in datacenters</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>Overview of MapReduce framework</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>NetFPGA 1G</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>NetFPGA reference router</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Architecture overview</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Detailed system architecture</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>Packet format for layer-3 virtualization</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Multi-receiver setup</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>Architecture of SRAM-based external forwarding tables</td>
<td>48</td>
</tr>
<tr>
<td>3.8</td>
<td>Receiver throughput versus packet size for a single virtual router</td>
<td>56</td>
</tr>
<tr>
<td>3.9</td>
<td>Experimental setup for measuring latency</td>
<td>57</td>
</tr>
</tbody>
</table>
3.10 Average latency for scaling virtual data planes ................. 60
3.11 Average throughput for scaling virtual data planes .......... 61
4.1 A partially-reconfigurable network virtualization platform ... 72
4.2 Layout of static and partially reconfigurable regions in Virtex II 74
4.3 The experimental testbed ........................................ 77
4.4 Throughput comparison ........................................... 78
4.5 Instantaneous forwarding performance - static reconfiguration 79
4.6 Instantaneous forwarding performance - partial reconfiguration 80
4.7 Average throughput for varying reconfiguration frequencies .. 81
4.8 Average throughput of scaling virtual networks ................. 82
4.9 Virtual network migration ........................................ 83
5.1 ReClick virtualization platform architecture .................... 91
5.2 Component and configuration .................................... 93
5.3 Packet format of NetFPGA aligned to 64 bit words ............. 95
5.4 Conditional inserts and removals ................................ 98
5.5 The generic architecture of a ReClick component ............... 99
5.6 Compiler Framework ............................................ 100
5.7 A ReClick IPv4 and Onion router ............................... 102
5.8 Topology for experiments ....................................... 104
5.9 Throughput comparison of ReClick and NetFPGA routers ....... 105
6.1 Illustration of iterative execution of PageRank ................. 110
6.2 Illustration of accumulative updates ........................... 115
6.3 Visualizing asynchronous accumulative updates ............... 116
6.4 Cluster setup for a four node Maestro system .......................... 118
6.5 Altera DE-4 .................................................. 119
6.6 Implementation of asynchronous accumulative updates on FPGA ....... 120
6.7 Threshold selection circuit ........................................... 123
6.8 Update data path ................................................ 124
6.9 Maestro prototype in lab ............................................. 128
6.10 Speedup of Maestro (1 FPGA) versus Maiter (1 microprocessor) ...... 132
6.11 Speedup of Maestro (1 FPGA) versus Hadoop (1 microprocessor) ..... 132
6.12 Speedup of Maestro (2 FPGAs) versus Maiter (2 processors) .......... 134
6.13 Speedup of Maestro (2 FPGAs) versus Hadoop (2 processors) ........ 134
6.14 Real-time network trace of Maestro for PageRank ....................... 136
6.15 Real-time network trace of Maiter for PageRank ......................... 136
6.16 Speedup of Maestro (4 FPGAs) versus Maiter (4 processors) ........... 139
6.17 Speedup of Maestro (4 FPGAs) versus Hadoop (4 processors) ........... 139
6.18 Best case speedup of Maestro versus Maiter ............................ 141
6.19 Best case speedup of Maestro versus Hadoop ............................ 141
6.20 Network trace for a partitioned 1.2 million node graph ................. 144
1.1 Trends and Challenges in Future Internet Systems

From its humble beginnings as a research initiative intended to interconnect simple networks of computers, the Internet has evolved into an essential infrastructure for a broad spectrum of services that range from simple electronic mail to sophisticated services such as e-commerce, content sharing and online social networks. Modern web applications are marked by distinct networking and computing characteristics. For example, while video streaming sites are sensitive to network throughput, real time stock trading applications are sensitive to network latency. E-commerce services demand a high level of network security. Search and business logic require large data processing capabilities for information retrieval.

Web applications operate over a large framework of networking and computing equipment formed from general-purpose processors and Application Specific Integrated Circuits (ASICs). For example, the networking infrastructure in the Internet is built on a vast array of routers and switches, where a layered hierarchy of networking protocols running in general-purpose and network processors provide services such as guarantees of packet delivery, security and performance to end applications. As the Internet has evolved, numerous protocols have been proposed and deployed in the upper layers of the networking stack (Figure 1.1). Classic examples include the File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP) in the application layer. These and several other protocols provide a plethora of useful architectural choices to application designers. The di-
versity of protocols in the application layer is largely attributed to its programmable nature.

In contrast, the middle and lower layers of the networking stack have remained virtually unchanged for decades [17]. For example, the transport layer in the Internet has been dominated by protocols such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). The network layer uses the Internet Protocol (IP). Although the lack of diversity and the choice of fewer protocols in these layers may be attributed to stability concerns, the vulnerabilities and performance issues of these protocols are well-known [22]. In recent years, numerous alternate protocols and architectural styles have been proposed in literature to overcome these issues [65] [33]. However, few have made inroads into mainstream networks.

While the need to introduce new networking technologies is fairly clear, several challenges exist. First, aggressive architectural changes in the network core are non-trivial and require wide agreement among network infrastructure providers [29]. Network operators are not only apprehensive of the consequences of deploying experimental protocols on their stable equipments, but they are also concerned about the economic incentives of such measures. Commercial vendors do not expose flexible features in the networking equipment for experimentation and design space exploration.
Recent efforts such as OpenFlow [58] call for better programmability for existing networking devices.

Technology choices also influence the diversity of networking layers. Today, these choices are almost always fine-tuned to the unique needs of each layer. For example, the upper layers of the networking stack are engineered on general-purpose microprocessors to aid programmability and ease of use while the lower (physical and data link) layers heavily rely on ASICs and proprietary network processors to maximize packet forwarding performance.

The Internet was originally envisaged to serve as a communication infrastructure. Early web applications only required minimal data processing capabilities that few isolated web servers could provide. However, with the proliferation of modern web-based search, data mining and scientific computing applications, the demand for raw computing horsepower has dramatically increased in recent years. While processor vendors have been able to accommodate this demand for some time with higher levels of transistor integration according to Moore’s law, this model no longer looks scalable as microprocessors have already hit the frequency and power wall [36].

Alternately, the computation can be parallelized on a cluster of homogeneous commodity hardware machines in datacenters. Datacenters allow an application to scale computing capacity by simply adding more machines. Several applications may share available resources on a need basis. While the datacenter computing model has been proven to be scalable [32] for a large class of applications, there are limitations. The limited memory bandwidth and data level parallelism restricts the ability of clusters to efficiently scale to data-parallel workloads that form the backend of search engines, scientific computing and data mining. Burgeoning infrastructure and energy costs [66] further limit application scalability. In summary, existing computing clusters offer a fairly generalized solution to a large class of problems in web applications.
As web applications grow in complexity, future Internet systems will need to support a greater level of diversity in computing and networking technologies to adequately reflect these needs. By diversity, we mean mechanisms by which a multitude of architectural styles and design policies can co-exist and evolve with existing systems. While in the short term, infrastructure diversity is necessary to meet the immediate needs of applications, diversity plays an important role in the long-term evolution of the Internet itself. For example, the availability of technology choices allow users and applications to test novel design techniques, make decisions and weed out inefficient approaches.

1.2 Thesis Statement

The goal of this thesis is to architect heterogeneous solutions for the diversity issues in networking and computing infrastructure by integrating Field Programmable Gate Arrays, a specialized class of reprogrammable integrated circuits with general-purpose microprocessor techniques. We believe that data-parallel architecture, reconfigurable nature and fast design cycles uniquely position FPGAs to address these issues. The integration of FPGAs with microprocessors provide a roadmap to adopt reconfigurable computing technology in mainstream networking and computing infrastructure. In support of our thesis, we demonstrate the benefits of integrating FPGA technology with general-purpose processors in two emerging Internet applications - network virtualization and distributed cluster computing.

1.3 Thesis Overview

Despite their merits, several challenges exist before FPGAs can be deployed in real systems. First, like all integrated circuits, FPGAs are fundamentally logic and memory constrained. In networked systems, this necessitates efficient, yet scalable implementations of forwarding structures including data planes and routing tables.
The second challenge lies in closing the gap between hardware designers and software developers. Traditionally, networking protocols have been developed using software-based Application Programming Interfaces (APIs) in general-purpose or network processors. In rare cases, designers go to the extreme extent of building custom hardware (e.g. ASICs) to meet strict performance constraints. While FPGAs offer considerable design flexibility in comparison to ASICs, they still expose unfamiliar programming interfaces (Hardware Description Languages, behavioral/dataflow modeling and EDA) to most network developers.

Introducing FPGAs in existing distributed cluster computing frameworks [32] [6] is challenging because none of the popular distributed cluster computing frameworks include support for specialized hardware nodes. Further, the computing model includes several inefficiencies that originate from the assumption of homogeneous commodity hardware machines. For example, the presence of strict synchronization barriers between computations in popular cluster computing frameworks such as MapReduce [32] and Hadoop [6] limit application performance.

This thesis makes the following specific contributions to address the aforementioned issues:

1. We demonstrate an architecture to implement novel networking techniques on a shared FPGA. Our architecture allows applications to scale beyond the logic and memory limitations of the FPGA with the aid of software techniques. Applications migrate between hardware and software resources based on their specialized needs. Scalable forwarding structures are supported using off-chip resources.

2. We present a programming model to describe reusable networking components on FPGAs. This model provides application developers with a design entry point higher than many hardware description languages. In our model, networking features are specified as an interconnected graph of small modules (compo-
Figure 1.2. The physical network (shown in the bottom) is virtualized into two virtual networks - red and blue (shown in the top).

3. We illustrate a hardware architecture and a computing model that will facilitate the integration of FPGAs in general-purpose clusters. Our model uses asynchronous accumulative updates to eliminate the need for strict synchronization barriers in existing cluster computing software frameworks. We implement this framework using a cluster of four FPGAs and show that our model works well for iterative algorithms, a popular class of distributed algorithms in modern web applications.

1.4 Applications Considered in the Thesis

The techniques presented in this thesis are demonstrated in the context of two applications - network virtualization and distributed cluster computing.

Network virtualization marks an important step in introducing new networking technologies to production networks by explicitly sharing the routing resources
between multiple virtual networks. Figure 1.2 illustrates this concept, where each virtual network represents a logical slice of the physical network. The routing components of the virtual network run routing policies independent of the policies of the physical substrate and offer distinct end-to-end services with unique Quality-of-Service (QoS) parameters. Routing resources such as CPU cycles, memory and I/O bandwidth can be shared between virtual networks. Selective parts of the physical network stack may be exposed to virtual networks to facilitate programmability. For a more detailed motivation and background on network virtualization see section 2.3.

Virtual networks need access to useful programming interfaces to customize aspects of the networking stack. High packet forwarding performance is desirable to test novel networking protocols at realistic traffic capacities. Further, the shared operation of distinct networking technologies, some of which are experimental in nature, requires good traffic isolation policies between the virtual networks. In realistic systems, the physical networking infrastructure will need to scale to support hundreds of virtual networks.

Unfortunately, existing network virtualization techniques that are based on commodity general-purpose microprocessors and ASICs do not possess all of these features. For example, network virtualization techniques that use virtualized microprocessors offer flexible interfaces to customize networking features, but suffer from poor packet forwarding performance [21] [23] [34]. On the other end of the spectrum, ASICs expose limited programmable interfaces to customize all parts of the networking stack [77]. This motivates the need for a programmable hardware solution such as FPGAs that possess a unique combination of reconfiguration and fine-grained data parallelism.

Distributed cluster computing provides a suitable approach to scale modern web applications by parallelizing the computation on general-purpose machines. Iterative algorithms form an important workload for distributed computing. These
algorithms are generally structured to progress in a series of iterations where the results of the current iteration are derived from the results of the previous iteration using a fixed set of operations. Although simple, Conways Game of Life, where the state of a grid cell in a given iteration is based on the states of its neighbors from the previous iteration, provides a familiar example of an iterative algorithm. Many contemporary search and data mining applications use iterative algorithms to refine and process large volumes of data. For example, PageRank [27] is used to refine the rank values of web pages in the World Wide Web. K-means clustering [71] is an iterative algorithm used to classify data in computational biology.

MapReduce [32] is a popular cluster computing model that relies on commodity hardware machines for iterative distributed computing. To execute iterative computations using the MapReduce model, the computation is specified as a sequence of tasks. These tasks work on key-value pairs stored in a distributed file system. Iterations are synchronized at the end of each task by writing key value pairs into the distributed file system.

The MapReduce model provides a limited approach to execute iterative algorithms [35] [91] since the sequential execution nature in general-purpose processors is not well suited to the data-parallel nature of the workload. While specialized hardware could be introduced into existing clusters to solve this issue, this is not straightforward. Distributed software frameworks assume that all cluster nodes are homogeneous in nature requiring synchronization barriers between iterations. These barriers prevent specialized hardware nodes, for example, from rapidly making progress in the computation.

1.5 Thesis Outline and Preview of Results

This thesis is organized into seven chapters.
Chapter 2 introduces the motivation and background material necessary to navigate the rest of this thesis. In this chapter, we introduce the architecture of FPGAs and provide examples of real-world applications that use FPGAs. In the next section, we provide background on network virtualization and survey state-of-the-art network virtualization techniques. In the next section, we provide an overview of distributed cluster computing frameworks and enumerate their limitations.

Chapter 3 introduces the first major contribution of this dissertation - a heterogeneous network virtualization platform that overcomes previous scalability issues in FPGA-based virtual networking systems. In this platform, several high-throughput virtual networks are implemented on an FPGA while additional networks are spawned on a PC server using host virtualization techniques. We introduce virtual network migration, a technique that allows virtual networks with dynamically varying throughput requirements to be mapped onto a heterogeneous set of high-throughput and low-throughput virtual networking resources. We use virtual network migration to customize the properties of virtual networks in the FPGA using static FPGA reconfiguration. Next, this chapter presents techniques to implement hardware virtual routing tables in a shared fashion using inexpensive off-chip memories. We present two case studies - an IPv4 virtual router and a Routing on Flat Label (ROFL) virtual router to validate our techniques. Our evaluation of the system, presented in the last section, show that FPGA-based virtual data planes can forward packets with one to two orders of better throughput than state-of-the-art software-based network virtualization systems. A virtual network operating in the FPGA can be reconfigured within 12 seconds.

Chapter 4 addresses the isolation issues associated with static reconfiguration in FPGA-based virtual networks. The customization of routing characteristics in a virtual network is challenging while the FPGA is being shared by multiple virtual networks because customization requires static reconfiguration and full shutdown of
the device during the reconfiguration interval. Device shutdown adversely affects the traffic in shared virtual networks. To address this issue, the FPGA-based network virtualization platform presented in Chapter 3 is extended by introducing partially-reconfigurable virtual networks. Partial reconfiguration is a special property of an FPGA that allows selective regions of the silicon to be reconfigured on the fly while the rest of the device is operating. We illustrate the utility of this technique in enhancing the isolation of shared virtual networks and experimentally evaluate the network downtime in partially-reconfigurable virtual networks. Our evaluation shows that partial reconfiguration can accelerate the frequency of virtual network reconfiguration by a factor of $20\times$ with no impact on the traffic in shared virtual networks.

Chapter 5 introduces ReClick, a programming model to simplify the specification of networking features of FPGA-based virtual data planes. ReClick describes data plane features as sequential packet manipulating operations. Modules developed in a hardware-agnostic language can be reused and stitched together to create complex data plane structures. ReClick features architectural optimizations to implement shared virtual data planes in an area-efficient manner on the reconfigurable hardware. Two data planes - IPv4 and an onion router have been developed to illustrate the capabilities of the programming model.

Chapter 6 presents Maestro, a heterogeneous cluster computing system that integrates FPGAs and general-purpose CPUs. We demonstrate that asynchronous accumulative updates [91] can be used to break the synchronization barriers in existing cluster environments that rely solely on general-purpose CPUs. Our system is evaluated experimentally by executing a general-class of iterative algorithms on a heterogeneous cluster of commodity CPUs and FPGAs. Both CPU and Altera DE4 FPGA-based compute elements prioritize computations to accelerate algorithm convergence in our scalable system. A speedup of $7\times$ is achieved over an implementation of asynchronous accumulative updates on a CPU. The system offers $154\times$ speedup
versus a standard Hadoop-based CPU workstation. Additional speedup is obtained by parallelizing the computation on multiple FPGA boards.

**Chapter 7** concludes the dissertation and provides directions for future work.

The results from the research have been published in the following conference proceedings and journal articles:


5. “Reconfigurable Data Planes for Scalable Network Virtualization”, *Accepted/To Appear on IEEE Transactions on Computers* [79].

CHAPTER 2
BACKGROUND

2.1 Overview of FPGA Technology

FPGAs are integrated circuits that can be reprogrammed to perform any digital logic function. Unlike ASICs, where the circuit behavior is permanently fabricated into the silicon, the behavior of FPGAs can be altered after device fabrication. This flexibility is attributed to an electrically programmable logic and routing circuitry. The enhanced flexibility, however, comes at a price of slightly higher area, delay and power costs [49].

FPGA circuit structure is shown in Figure 2.1. The architecture is organized as a regular two dimensional array of logic blocks called lookup tables (LUT), each of which can perform a custom boolean logic function. All logic blocks are interconnected with a programmable routing circuitry that runs horizontally and vertically between the logic blocks. The programmable logic and routing circuitry is customized using Electronic Design Automation (EDA) software. In addition to millions of logic blocks, state-of-the-art FPGAs integrate specialized memory/DSP blocks, high speed I/O interfaces and hardened networking protocol implementations such as Gigabit Ethernet and PCI Express.

As the device density grows in integrated circuits according to Moore’s law, the fabrication of ASICs incurs high design engineering and mask costs. A large fraction of these costs originate from the extensive verification efforts required to make sure that the silicon works as expected post device fabrication. ASICs leave little room to commit design errors, and when they are made, the costs of re-spins are prohibitive.
In order to amortize the long design cycle ranging anywhere from six months to several years, ASICs rely on the large market volume.

FPGAs are low-cost alternatives to ASICs. Designing an application using FPGAs involves five steps as illustrated in Figure 2.2. The application is described using a hardware description language such as Verilog/VHDL. Next, the hardware description is translated by Computer Aided Design (CAD) tools into an optimized netlist. The netlist is packed into lookup tables. The packed netlist is placed and routed for the FPGA device architecture under constraints of area, clock period and power. In the final step, a bitstream is generated for programming the target device.

In recent years, advances in CAD technology have greatly simplified the process of FPGA application development. The application designer’s role is often limited to providing a high level description of the hardware behavior. The FPGA compilation process is fast requiring only few minutes to couple of hours. Design errors are largely tolerated by virtue of the reprogrammable nature of the logic and routing fabric. These features make FPGAs particularly attractive for rapid prototyping applications.
2.2 FPGAs in Networking and Computing Infrastructure

FPGAs are used in a variety of wireless and wireline back haul equipment [1] [4], primarily as glue-logic to interconnect network processors and to reduce the cost of hardware upgrades. Since FPGAs typically lead ASICs in process technology, they are used extensively to prototype new networking technologies before fabricating custom ASICs. The reprogrammable nature enables network equipment vendors to tolerate in-field operational failures. ASIC-style IPs along with the logic fabric in FPGAs enable customized network processors.

In recent years, the need for better programmability inside the network equipment has opened up new opportunities for FPGAs. This need is partly driven by the emergence of software defined networks (SDN). Software defined networking is a new paradigm that allows more programmability into existing networking equipment by allowing data and control flow modifications using open protocols. OpenFlow [58] is an open SDN initiative from academia, that allows control planes of proprietary switches and routers to be remotely controlled using open protocols. Better programmability into existing networking equipment allows novel networking protocols
to be tested and deployed under realistic traffic conditions without service disruption. FPGAs are uniquely positioned to address the programmability needs of software defined networks.

FPGAs have been used as coprocessors for accelerating high-performance computing applications such as financial analysis, biological sequence matching, medical imaging and scientific computing. Application speedups ranging from 20× to 300× [30] have been reported. FPGAs are particularly attractive in places where the cost of designing a custom ASIC-based coprocessor to suit the needs of the application are prohibitive. A detailed survey describing the opportunities and challenges of reconfigurable computing in the cloud can be found in [57].

2.3 Network Virtualization

This section provides the necessary motivation and background on network virtualization technology that forms the basis of our work in Chapters 3-5. State-of-the-art virtualization techniques are surveyed here. Virtualization is a well-known technique that has been applied in a diverse set of computing technologies in the past. In general, virtualization provides a logical view of a physical resource to entities that require shared access to that resource. Operating system virtualization, for example, allows multiple guest operating systems to be run on top of a single kernel and access shared hardware resources. Memory virtualization is a popular technique used in computer architecture to share distributed memory resources among processes. More recently, datacenters use server virtualization to share distributed hardware and software resources between multiple applications. Similarly, storage virtualization is used to present a unified logical view of distributed disk resources to the system administrator, simplifying management tasks. In general, any virtualization technique allows the separation of policies from the mechanisms that implement them.
The goal of network virtualization is to share the physical resources in the Internet such as routers and switches between multiple virtual networks. By slicing the physical network, differentiated services and routing policies can be deployed in virtual networks. It is natural to already think of the Internet as a shared network for exchanging information between computers. The shared operation eliminates the need for unique point-to-point connections between all the communicating parties. However, it is worthwhile to note that the unit of sharing in the Internet is a packet. Multiple applications interleave packets into the shared network. The packets are carried to their final destinations by intermediate routers and switches. The use of the packet as a unit of sharing is fairly fine grained because packets simply do not facilitate control over all aspects of the networking infrastructure that include the end hosts, links and routing devices.

The central idea behind network virtualization is to raise the unit of abstraction from that of a packet to a complete network slice consisting of routers, switches and links. By raising this abstraction, network users will be able to gain control over all aspects of the network - including infrastructure, links and end hosts. In a virtualized network, multiple virtual network slices share the physical network. The user of a virtual network has complete control over all aspects of the virtual networking slice, including routing policies, data plane characteristics and topology specifications. Virtual network users can use the improved control to devise efficient data movement mechanisms and deploy them rapidly on legacy network infrastructure. Further, the improved control can be used for service differentiation, experimentation and diversification.

Figure 2.3(a) shows a physical network, which is virtualized to support a red virtual network (Figure 2.3(b)) and a blue virtual network (Figure 2.3(c)). A virtual network is a slice of the physical network formed from virtual routers. Each virtual router represents a logical routing entity that executes in an isolated environment.
Figure 2.3. (a) Physical network, (b) Red virtual network, (c) Blue virtual network created using node virtualization. Node virtualization slices the routing resources of the physical router such as CPU cycles and memory resources between virtual routers. The virtual routers are interconnected with virtual links, some of which may span multiple physical hops.

Each virtual network might run different routing processes (with the same or different routing protocols) and therefore might have different views of the network topology. For example, Figure 2.3(b) shows a topology which differs from the one in Figure 2.3(c) although both virtual networks run on the same physical network. Since virtual networks run isolated from one another, virtual network owners can deploy their experimental protocols without affecting the actual network. By exposing
selective portion of the physical router architecture through the virtualization layer to network developers, physical network owners can promote controlled experimentation. By allowing multiple virtual networks to co-exist, virtualization promotes diversity. Additionally, physical network owners have the potential opportunity to transform the diversity into revenue.

Network virtualization presents several interesting sub-problems to the network researcher. For example, when several virtual networks have conflicting bandwidth/latency specifications, how can an effective mapping be performed to the physical resources? or how can infrastructure providers provision resources to maximize an objective such as network utilization or revenue? Although these problems are certainly interesting, they are beyond the scope of this thesis.

This thesis focuses on node virtualization. Specifically, we focus on issues which need to be addressed to carve a good network virtualization platform from existing physical routing resources. Figure 2.4 shows the structure of a physical routing node, which is partitioned into multiple virtual routing nodes (virtual routers). A virtual router consists of two major parts: a control plane, where the routing processes
exchange and maintain routing information; and a data plane, where the forwarding information base (FIB) stores the forwarding route entries and performs packet forwarding. The virtual routers are isolated from each other and can run different routing, addressing and forwarding schemes. Any virtual router joining the virtual network is marked with a color (e.g. red or blue) and data packets are colored in a similar fashion. The physical router provides DEMUX and MUX circuitry for the hosted virtual routers. After exiting a physical link, a colored packet will be delivered by the DEMUX to the virtual router with the same color. When packets are emitted from a virtual router, they are colored with the router’s color at the MUX before entering the physical link. Because of this packet-level separation, a virtual router can only communicate with virtual routers of the same color.

There are many aspects to consider in the design of a virtualization substrate. We enumerate some of them below:

- **Flexibility**: Customization of the networking stack is a fundamental design objective for virtualization. Deploying new routing policies such as ROFL [28] requires modifications to several aspects of the network core such as the routing protocol and the address lookup algorithm. Other examples include QoS schemes that require certain queuing and scheduling approaches and security mechanisms such as network anonymity or onion routing [33] [92]. Existing overlay virtual network testbeds such as PlanetLab [51] support only customization of layers IP and above, limiting the scope of network customization. It is desirable for the virtualization substrate to support distinct, yet co-existing data-plane and control-plane policies.

Gaining more programmability into existing networking devices requires simplified, yet powerful interfaces that can manipulate both data plane and control plane features of the physical routing substrate. Existing overlay networks such
as VINI [21] and PlanetLab [51] expose APIs to customize upper layers of the networking stack.

- **Performance:** Superior data plane performance is desirable in order to evaluate new networking techniques under realistic traffic conditions. Further, when these techniques are proven to be viable, virtual networks will need to offer capacities similar to the physical routing platform to attract applications to migrate to virtual networks. Unfortunately, the packet forwarding capacity of existing network virtualization approaches that rely on host/container virtualization techniques cannot match the data plane bandwidth of commercial routers [21] [24].

- **Scalability and Resource Provisioning:** Since the routing resources of the physical node are shared between virtual networks, the contention for available resources such as CPU cycles, memory bandwidth increases as additional networks are created. The virtualization node will therefore need to incorporate mechanisms to share the available resources in an efficient manner, minimizing any degradation in performance as virtual networks scale. Further, since experimental networks will need to operate at different traffic capacities [51], the available routing resources should be efficiently allocated among all virtual networks. The allocation must be performed in a dynamic and transparent fashion as virtual network requirements change.

- **Isolation:** In experimental testbeds, traffic interference between virtual networks affects the quality of network measurements. Malicious users can exploit interference to introduce security threats. When network parameters are reconfigured in a virtual network, other shared networks should not be affected.

Other goals that are relevant, but not highlighted here, include security, ease of management and backward compatibility with legacy Internet architectures.
Host virtualization is a popular technique used to share routing nodes in overlay network testbeds such as PlanetLab [51]. In host virtualization, virtual routing instances are created by first splicing the router’s\(^1\) host software into isolated environments and then executing routing processes within these environments. Host virtualization is achieved using full-virtualization, container-based (operating system) virtualization or para-virtualization [82].

Full virtualization illustrated in Figure 2.5(a) allows several virtual machines to execute on top of the hardware. Each virtual machine emulates the underlying hardware and hosts an unmodified guest operating system (OS) by providing the required binary translation, memory and I/O management mechanisms. Virtual routing instances run as application processes in the guest operating systems without being aware of the underlying virtualization layer. Examples of full-virtualization include VMware [82] and Hyper-V [7].

Container virtualization or OS virtualization allows virtualization at the operating system level [72]. In this form of virtualization illustrated in Figure 2.5(b), multiple guest OS instances run as application processes on top of the host OS. Virtual routing instances execute as application processes within each guest OS. Since all guest

\(^1\)The router in this context is a microprocessor system running a general-purpose operating system
operating systems share the host operating system kernel, container virtualization obviates the need for binary translation and I/O management mechanisms, making it easier for users to deploy and manage routing instances. Examples of container virtualization include OpenVZ [13] and Linux VServer [8].

Para-virtualization illustrated in Figure 2.5(c) runs OS instances on a virtualization layer called the hypervisor. Unlike full-virtualization techniques, paravirtualization requires modification of the guest operating system, making it less portable. The most popular paravirtualization platform available today is Xen [34].

Host virtualization techniques offer a good combination of flexibility and cost-effectiveness for network virtualization purposes. The CPU cycles and physical memory in the physical routing platform can be fairly shared between different virtual networks using software schedulers. The network stack of systems implemented using host virtualization techniques can be programmed using software APIs. Several overlay virtual networks and testbeds implement host virtualization technology [23] [60].

Bhatia, et al. [23] developed a virtualization platform which can be scaled to sixty independent virtual networks. This system allows for individual network customization and the use of a commodity operating system which can support a variety of services, including tunneling [15]. Packet forwarding is performed in the kernel under application control. Keller and Green [47] proposed a system which allows for customized packet handling for each data plane in a virtualized network. This system uses an unvirtualized Linux kernel to host multiple concurrent data planes implemented in Click [5]. Packet handling is specified as an interconnected graph of networking functions. Liao et al. [52] proposed the parallel operation of a cluster of commodity hardware machines to accelerate virtual machine packet forwarding performance. The throughput of virtual machines can be improved by applying efficient packet handling techniques that use optimized system calls and packet copying oper-
ations in memory [53]. A comprehensive survey of virtual network implementations using software techniques can be found in [29].

Although the substantial progress of host virtualization techniques is important [23] [34], the serial nature of general-purpose microprocessors and the overhead of virtualization layer limits the achievable performance of software-based virtual network devices. It has been observed that software-based data plane implementations exhibit statistical variations in network parameters due to jitter and resource contention. For example, in container-based virtualization and full virtualization techniques [21], each virtual network resource must contend for hardware and operating system resources such as CPU cycles, bandwidth and physical memory. An analysis of overlay testbeds such as PlanetLab [51] show that virtual networks also show variations in bandwidth. Host virtualization techniques allow limited dynamic provisioning mechanisms such as rate limiting and bandwidth reservations within the available bandwidth capacity. However, as the number of virtual networks scale, opportunities for bandwidth revisions are severely limited due to the overall bandwidth limitations.

Many commercial networking systems employ ASICs [4] in the form of network processors. ASICs are specifically tuned for low-power high performance applications. In recent years, several vendors have added virtualization support to existing ASIC-based networking systems. For example, Cisco Nexus 7000 series [4] integrates switch virtualization support. Cavium [3] Octeon series features virtual SoCs with separate virtual memory and I/O interfaces.

Although ASICs are fine-tuned for high performance, they do not provide the necessary flexibility to customize several aspects of the networking stack. For example, the Supercharging PlanetLab [77] is an ASIC-based virtualization platform that only provides a customizable forwarding table interface. This makes it hard to implement diverse data plane policies such as network anonymity, onion routing and
QoS schemes. ASICs also incur long design cycles and high mask costs, making them prohibitively expensive for prototyping new network architectures.

2.4 Cluster Computing - Architecture and Programming Model

This section provides the necessary background on distributed cluster computing models that form the basis of our work in Chapter 6.

A cluster is a collection of commodity hardware machines interconnected by local area networks that run distributed software frameworks. The origin of computing clusters dates back to the nascent years of the Internet when interconnected machines were used to solve scientific problems. In recent years, clusters have become much more affordable due to the availability of low-cost microprocessor technology, high-speed interconnection networks and distributed computing software.

Modern web applications such as search engines and content sharing sites rely on clusters for heavy duty data processing. For example, the popular search engine, Google, uses clusters housed in datacenters to process web pages from the World Wide Web [32]. Amazon [2] and Microsoft [16] lease parallel machines for utility computing. While general-purpose processors impose hard limits on application performance due
Figure 2.7. Overview of MapReduce framework

to limits on instruction-level, data-level, and thread-level parallelism, a cluster allows applications to scale by simply parallelizing the problem on more machines.

The high-level architecture of a cluster is shown in Figure 2.6. Several servers are interconnected using a high-speed interconnection network. The individual servers may be virtualized to run multiple virtual machines using host virtualization technology. An application may be executed on one or more virtual machines. Within each server, management functions such as allocating the correct CPU cycles and memory resources to applications are performed by a software-based management layer called hypervisor. The network fabric that interconnects the servers is organized as a tree of edge switches, aggregation switches and core switches, with servers placed at the leaves of the tree. The bandwidth capacity of the links increases progressively towards the root.

A distributed software framework handles data management functions. This framework is responsible for distributing the workload, collecting the final outcomes, load-balancing, fault-tolerance and providing programming interfaces for application
designers to specify the application behavior. MapReduce [27] is a widely popular distributed cluster computing model popularized by Google. MapReduce was designed with scalability, simplified cluster management and robust fault-tolerance as the primary design goals.

We illustrate the MapReduce computing model in Figure 2.7. The input dataset is specified in the form of key value pairs (KV pair) and stored in a distributed file system (data store). A computing node, known as the master, breaks the input dataset into smaller chunks. These chunks are assigned to multiple machines, called workers, over the local network. Workers process the data in two phases - Map and Reduce. The map phase is performed in parallel by all the workers. This phase transforms the individual KV pairs into intermediate KV pairs. Next, all workers shuffle the intermediate KV pairs over the network to aggregate values with the same key together. Finally, in the reduce phase, intermediate values with the same output key are combined to produce the final solution. The reduce phase may be executed in one or more machines. Fault tolerance is supported through implicit data-replication mechanisms.

Hadoop [6] provides an open-source Java-based implementation of MapReduce which is widely used by Yahoo! and Facebook. Several applications that use the MapReduce programming model including PageRank [27], a well-know algorithm used in web search, link prediction [54] algorithms and recommendation systems [20] in video streaming and social network analysis.

Iterative algorithms form a large class of algorithms that are parallelized in datacenters using the MapReduce framework. In such algorithms, the input data is successively refined by repetitively performing the same set of operations in iterations. For example, PageRank [27] is a well-known iterative algorithm that is used to calculate the relative importance of the vertices in a graph. PageRank has practical utility in web search, link prediction and recommendation systems. The general
PageRank algorithm is described as follows: Consider a web linkage graph \( G(V,E) \), where \( V \) represents the webpages (vertices of the graph), and \( E \), the set of hyperlinks between webpages (edges of the graph). An edge exists between nodes \( i \) and \( j \) if a hyperlink exists from node \( i \) to node \( j \). To calculate the relative importance of webpages, each node \( v \) in the graph is initially assigned PageRank score \( R(v) = \frac{1-d}{|V|} \). The PageRank of each node is successively refined from the current values. The refined PageRank score of a node \( v \) in the \((i+1)\)th iteration \( R^{(i+1)}(v) \) is computed as:

\[
R^{(i+1)}(v) = \frac{1-d}{|V|} + \sum_{u \in N^-(v)} \frac{d.R^{(i)}(u)}{|N^+(u)|}
\]  

(2.1)

where \( N^- \) denotes the set of nodes which have directed edge connections towards node \( v \), \( N^+ \) denotes the set of nodes that have outgoing edges from node \( v \), \( d \) denotes a constant dampening factor. The iterative computation runs until the difference in the PageRank values between two consecutive iterations has a value less than \( \varepsilon \). In the PageRank example, the final PageRank scores of all webpages can only by determined by iterating a number of times over the web linkage graph.

To parallelize the PageRank example using MapReduce, the web linkage graph is partitioned and distributed across all the workers. Next, each map task operates on a node \( v \). The map operation calculates \( \frac{R^{(i)}(v)}{|N^+(v)|} \) for all outgoing links from \( v \). This partial ranking score is shuffled to outgoing nodes. In the reduce phase, each node sums the partial ranking scores received from its incoming edges and adds \( \frac{1-d}{|V|} \) to compute its PageRank for the iteration. The operation repeats until the algorithm converges according to Equation 6.2.

Although MapReduce provides a scalable approach to execute iterative algorithms, it is quite inefficient. For example, the process of scheduling each iteration as a separate MapReduce task wastes CPU and I/O cycles. Since repeated reads and writes must be performed from the file system between iterations, the I/O overhead is significant.
Since each new MapReduce iteration can only start after the completion of the previous iteration, the reduce phase starts only after receiving all the intermediate KV pairs from other map tasks. These requirements impose strict synchronization barriers between iterations degrading application performance. Such synchronization barriers also cause bursty traffic patterns leading to network congestion.

The sequential nature of general-purpose processors make MapReduce not optimally suited to execute data-parallel workloads. MapReduce assumes that computing nodes are fairly homogeneous in nature - i.e. all the machines in the cluster make roughly equal progress at any given time during the computation. This assumption makes it difficult to introduce data-parallel architectures that may better suit the nature of the computation.
CHAPTER 3

NETWORK VIRTUALIZATION USING FPGAS

While the reconfigurable nature and data-parallel architecture make FPGAs suitable for virtual networking applications, several practical challenges exist in designing a realistic network virtualization substrate. For example, the constraints in logic and memory resources in FPGAs limit the number of virtual networks that can simultaneously share the device. In contrast, host virtualization techniques scale well to support hundreds of virtual networks by simply sharing the CPU and memory resources. The limited silicon real-estate in FPGAs also necessitates efficient hardware forwarding structures such as routing logic and routing tables.

This chapter addresses the scalability issue by designing a novel heterogeneous and scalable network virtualization platform that integrates FPGAs and existing host virtualization techniques. Section 3.1 surveys existing network virtualization approaches that use FPGAs and enumerates several limitations. Section 3.2 introduces the design goals and architecture of the heterogeneous virtualization platform. Section 3.3 describes virtual network migration as a technique to scale data planes in the FPGA. In section 3.4, we present two case studies that demonstrate the capabilities of the system. Section 3.5 demonstrates a technique to implement shared virtual forwarding tables using inexpensive external memories. Finally, in section 3.6, we provide an evaluation of the FPGA-based network virtualization platform.
3.1 Review of FPGA-based Virtualization Platforms

The evaluation of network virtualization platforms built from FPGAs is much more limited than previous software efforts. Anwer et al. [18] [19] demonstrate the implementation of up to 8 virtual data planes in a single Virtex II Pro on a NetFPGA board. Physical links in this platform are virtualized by associating each NetFPGA network port with one or more virtual ports in hardware. The control planes are implemented in OpenVZ [13] containers running in host software. Although this architecture has been shown to provide twice as much throughput as a software kernel router, a number of limitations exist. The logic resources of the FPGA impose a hard cap on the number of supported virtual networks, limiting scalability. The hardware data planes use non-scalable structures such as FPGA on-chip memories to implement key networking features such as forwarding tables.

CAFE [56] implements a similar platform that supports distinct virtual data planes on the NetFPGA. A salient feature of the CAFE architecture is the presence of user configuration registers that allow real-time updates to virtual routing table protocols. However, like previous approaches, CAFE presents scalability issues and offers limited ways to customize the properties of the virtual data planes.

Although these initial FPGA-based approaches provide useful initial insight into the applicability of FPGAs in virtual networking, none of them provide a complete platform that addresses the scalability issue. Further, these previous efforts do not demonstrate mechanisms to customize data plane characteristics other than forwarding tables. The limited scalability of FPGAs and low packet forwarding performance in software-only network virtualization approaches motivate us to consider a heterogeneous and adaptive approach to assigning virtual networks to hardware and software resources.

Our system makes three specific contributions to existing network virtualization platforms:
1. **Heterogeneous data planes:** We present a heterogeneous virtualization platform that combines fast hardware data planes implemented in FPGAs with slower software data planes implemented using host virtualization techniques. The heterogeneity in virtualization resources is used to scale the number of data planes beyond the logic capacity of pure FPGA-based virtualization platforms. We validate this system using both IP and non-IP based data planes.

2. **Dynamic Virtual Network Migration:** The system adapts to cater to the changing virtual network service requirements by dynamically migrating active virtual networks between hardware and software data planes. FPGA reconfiguration is used to aid data plane migration. During FPGA reconfiguration, unmodified hardware data planes can be temporarily migrated to software so that they can continue to transmit traffic.

3. **Scalable Virtual Forwarding Tables:** To promote scalability, the system implements an optimized hardware data plane architecture that stores forwarding tables from multiple virtual routers in a shared fashion using inexpensive off-chip SRAM memories. The architecture obviates the need for heavy pipelining in hardware.

In the following sections, we present the major design goals, an overview of the architecture, details of the hardware and software data planes and strategies to scale the data planes.

### 3.2 System Design

#### 3.2.1 Design Goals

Our design decisions are driven by two design goals. The primary design goal of the system is to improve the scalability of existing homogeneous FPGA-based network virtualization platforms. The scalability restrictions in existing FPGA-based
platforms originate from two factors. First, the limited logic resources (slices, flip flops etc.) constrain the number of simultaneous data planes that can operate on the device. Simply increasing the FPGA size to scale the number of data planes is cost-inefficient since FPGA cost generally does not scale linearly with device capacity. Second, individual hardware data planes use separate on-chip memory resources (BRAMs, TCAMs) to store forwarding tables. Such an implementation does not scale well with larger forwarding tables or a greater number of data planes. It is therefore important to scale both the number of data planes and the size of forwarding tables to build a practical network virtualization platform.

The secondary design goal of the architecture is to improve the design flexibility of hardware data planes through FPGA reconfiguration. Although FPGAs offer high data plane design flexibility by virtue of their reconfiguration properties, customization of individual hardware data planes in the same FPGA through static reconfiguration additionally requires that traffic in active virtual networks, other than the one being customized, be stopped during the reconfiguration procedure. It is therefore necessary for the architecture to support hardware data plane customization with minimal traffic disruption in shared hardware data planes.

Our architecture includes hardware and software techniques to address these design goals. Specifically, we implement additional virtual data planes in host software using container virtualization techniques to scale the number of data planes beyond the logic capacity of the FPGA (Section 3.3). We address the limitations in memory scalability by implementing forwarding tables from multiple hardware data planes in a shared fashion using inexpensive external SRAM memories located outside the FPGA (Section 3.5). The architecture enables customization of hardware data planes using virtual network migration between hardware and software when virtual networking requirements change.
3.2.2 NetFPGA

The high-level architecture of our system built on the NetFPGA [84] platform. The NetFPGA [62] is an open FPGA-based development platform for teaching and research from Stanford University. The NetFPGA platform shown in Figure 3.1 includes an FPGA board, open-source gateware and software. The board features a Xilinx Virtex II Pro FPGA integrated with four 1 Gbps Ethernet interfaces, a 33 MHz PCI interface, 64 MB of DDR2 DRAM and two 32 MB SRAMs. The board is attached to a PC and programmed via the PCI interface. Many applications have been developed using this platform including an open IPv4 router, a programmable network interface card, a line rate packet generator and an FPGA-based Software Defined Radio (SDR) platform [9].

The NetFPGA reference router [62] is a modular IPv4 router implemented in FPGA logic. The hardware datapath of the NetFPGA reference router, shown in Figure 3.2, is implemented as a pipeline of fully customizable modules. Each module includes a register file for control and statistic collection. The registers in the register file are memory-mapped and can be programmed from host software through the PCI interface. The hardware data path of the base router consists of input queues, an input arbiter, an output port lookup module, and output queues. Incoming packets
Figure 3.2. NetFPGA reference router

from PHY Ethernet interfaces are placed into input queues. The input arbiter module services each queue in a round robin fashion. The output port lookup module consists of ternary CAM (TCAM)-based forwarding tables that support IP lookup and ARP lookup mechanisms. Processed packets are sent to the output queues from where they are forwarded to the physical interface. The forwarding tables of the reference router are software-programmable via the PCI-register interface.

The control plane for the base router is implemented in host software running the Linux operating system. The control plane currently supports a modified OSPF (PW-OSPF) routing protocol. More information on the NetFPGA reference router is available from [10].

3.2.3 Architecture Overview

The high level architecture of the network virtualization platform built on top of NetFPGA infrastructure is shown in Figure 3.3. In this system, virtual data planes that require the highest throughout and lowest latency are implemented on a Virtex II-Pro 50 FPGA on the NetFPGA while additional software virtual data planes are implemented in OpenVZ [13] containers running on the PC. The forwarding tables of the hardware virtual data planes can either be implemented using BRAM and SRL16E blocks within the FPGA or using the 36 Mbit SRAM located external to the FPGA. In either case, forwarding tables can be updated from software through the
PCI interface. The PCI interface facilitates flexible control plane implementations in software.

In addition to the NetFPGA board, our system includes a PC server to host the software virtual data planes. The PC server is sliced into virtual machines using OpenVZ [13]. The OpenVZ framework is a lightweight virtualization approach used in several network virtualization systems [52] [83] and it is included in major Linux distributions. The OpenVZ kernel allows multiple isolated user-space instances (hereafter referred to as containers). Data planes can be spawned in host software when an FPGA can no longer accommodate new data planes. Since software virtual data planes must be effectively isolated from each other, they are hosted in isolated OpenVZ containers. The OpenVZ virtual environment guarantees that the each container gets a fair share of CPU cycles and physical memory. Each instance of the OpenVZ container executes a user mode Click modular router [5] to process the packets. The forwarding functions of Click can be customized according to the virtual network creator’s preferences.
3.2.4 Packet Forwarding

Packet forwarding operates as follows. When a packet arrives at an Ethernet interface (PHY), the destination address in the packet header is used to determine the location of its data plane. If the packet is associated with a virtual network hosted in the FPGA, it is processed by the corresponding hardware data plane. Otherwise, it is transmitted to the host software via the PCI bus. A software bridge provides a mux/demux interface between the PCI bus and multiple OpenVZ-based data planes. Periodically, the virtual network administrator can reconfigure virtual networks in the FPGA to take changes in bandwidth demands and routing characteristics into account. While the FPGA is being reconfigured, all traffic is routed by the host software.

Next, we describe the detailed architecture of FPGA-based and software-based data planes.

3.2.5 Hardware Data Planes

The hardware data planes of our virtualization platform are constructed by customizing NetFPGA’s modular datapath [10], as shown in Figure 3.4. We retain the basic components of the datapath including input queues, input arbiter and output queues. Besides these standard components, the system includes two additional hardware modules. The dynamic design design select module provides the demux interface in hardware for packets arriving at the physical network interfaces to virtual data planes. The CPU Transceiver module facilitates transmission of packets to virtual data planes in host software.

When packets enter the system, they are automatically classified by the dynamic design design select module based on virtual destination addresses in the packet header. Packets belonging to virtual networks can be classified based on virtual IP addresses or virtual MAC addresses in the packet header. The mapping from virtual
networks to virtual data planes can be programmed into the *dynamic design select table* using NetFPGA’s register interfaces by a person administering virtual networks (hereafter referred to as the operator). The *CPU Transceiver* module provides an interface to transmit and receive packets from virtual data planes in host software. More details on the operation CPU transceiver module are described in section 3.3. Packets processed by hardware data planes are sent to the output queues and subsequently forwarded through one of NetFPGA’s physical interfaces.

We implement the forwarding logic of hardware data planes by customizing instances of the output port lookup module [10], which encapsulates the forwarding logic of the NetFPGA reference router. Each virtual data plane has its own unique set of forwarding table control registers. This architecture offers two advantages. First, it ensures close to line rate data plane throughput for each virtual data plane. Second, independent hardware resources facilitate strong resource isolation between the virtual networks. By providing unique forwarding engines to each virtual data plane, the system allows network users to customize their data planes independently.

Forwarding tables of individual data planes are implemented using TCAM or BRAM resources within the FPGA or using SRAM memories located outside the...
FPGA. When forwarding tables are implemented using on-chip memory, the forwarding logic integrates TCAMs that support IP lookup and ARP lookup mechanisms. Section 3.5 describes the implementation of forwarding tables using external SRAMs. When forwarding tables are stored in external SRAMs, input and output queues must be implemented using the DDR2 DRAM memory. We implement the control planes for the virtual networks hosted in the FPGA in host software using the Linux operating system. The control planes currently support a modified OSPF (PW-OSPF) routing protocol.

Figure 3.4 shows the architecture of virtualization platform which supports four hardware virtual data planes and an interface to additional software data planes. The hardware data planes in this example support both IP and non-IP based forwarding techniques. The IP-based data planes support source-based, destination-based and source-and-destination-based routing approaches. The non-IP data plane forwards packets based on ROFL [28], a flat label lookup. The implementation of these data planes are described in section 3.4.

3.2.6 Software Data Planes

Software data planes provide low throughput extensions to the data planes implemented in the FPGA. Additionally, they usefully enhance the isolation properties of the virtualization platform by forwarding packets that would ordinarily be forwarded from hardware data planes during FPGA downtime. We use container virtualization techniques to implement the software data planes. Container virtualization techniques are popular because of their strong isolation properties and ease of deployment.
We virtualize the Linux server attached to the NetFPGA card using OpenVZ. OpenVZ virtualizes a physical server at the operating system level. Each virtual machine performs and executes like a stand-alone server. The OpenVZ kernel provides the resource management mechanisms needed to allocate resources such as CPU cycles and disk storage space to the virtual machines. Compared with other virtualization approaches, such as full virtualization and paravirtualization [82], OS-level virtualization provides the best performance and scalability. The performance difference between a virtual machine in OpenVZ and a standalone server is almost negligible [72].

The OpenVZ containers run Click as a user-mode program to execute virtual data planes. Click allows data plane features to be easily customized. Each OpenVZ container has a set of virtual Ethernet interfaces. A software bridge on the PC performs the mapping between the virtual Ethernet interfaces and the physical Ethernet interfaces located in the PC. A penalty of running user mode Click inside the OpenVZ container is slow forwarding speed.

3.3 Data Plane Scaling and Virtual Network Migration

We consider two separate approaches to scale the number of data planes beyond the logic capacity of the FPGA. In the first approach, all packets initially enter the NetFPGA card. The CPU Transceiver module within the FPGA forwards packets targeted for virtual networks implemented in software to the host PC via the PCI bus. Click routers running in OpenVZ containers process the packets and return them back to the NetFPGA card. Processed packets are transmitted through NetFPGA’s physical interfaces. We subsequently refer this approach as the single receiver approach. In the second multi-receiver approach, the NetFPGA card only receives packets targeted for hardware data planes. A separate PC network interface card
Figure 3.6) receives and transmits packets destined for software virtual data planes. We describe the details of each approach below.

### 3.3.1 Single-receiver Approach

If an incoming packet does not have a match for a hardware virtual data plane in the dynamic design select table on the FPGA, the packet is sent to the CPU transceiver module shown in Figure 3.4. The CPU transceiver examines the source of the packet and places the packet in one of the CPU DMA queues (CPU TX Q) interfaced to the host system through the PCI interface. The system exposes CPU DMA queues as virtual Ethernet interfaces to the host OS. The CPU transceiver modules modifies the layer 2 address of the packet to match the address of the virtual Ethernet interfaces of the target software data plane. The kernel software bridge forwards the Ethernet packet to its respective OpenVZ container based on its destination layer 2 address (DST MAC for IPv4 in Figure 3.5). The Click modular router within the OpenVZ container processes the packet by modifying the same three packet fields as the hardware router (DST VIP, SRC IP, and DST IP for the IPv4 data plane). The software bridge then sends the packet to a CPU RX Q on the NetFPGA board via the PCI bus. After input arbitration, the dynamic design select module sends the processed packet to the CPU transceiver. The CPU transceiver module extracts the source and exit queue information from the processed packet and places it in the output MAC queue interface (MAC TX Q) for transmission.

The software interface enables on-the-fly migration of virtual networks from software to hardware and vice versa. The virtual network operator can dynamically migrate a virtual network from hardware to software in three steps. In the first step, the operator initiates an OpenVZ virtual environment that runs the Click router inside the host operating system. Next, the operator copies all the hardware forwarding table entries to the forwarding table of the host virtual environment. In the final step,
the operator writes an entry into the dynamic design select table indicating the association of the virtual IP with a software data plane. Our current implementation imposes certain restrictions on virtual network migration from software to hardware. If the software virtual data plane has a forwarding mechanism that is unavailable in any of the hardware virtual data planes, network migration to hardware requires reconfiguration of the FPGA.

3.3.2 Multi-receiver Approach

In this approach, the NetFPGA card receives packets destined for all FPGA-based data planes while a separate NIC attached to the host PC receives all traffic destined for software data planes. This approach relies on network switches to forward packets to software or hardware data planes, as shown in Figure 3.6. We use layer 2 addressing to direct each packet to the appropriate destination (NetFPGA card or PC NIC). When deployed in the Internet, we assume that the sender is capable of classifying each packet as targeted to either the NetFPGA card or PC NIC based on the virtual layer 3 address. This approach requires the use of external hardware (switches) but simplifies the FPGA hardware design since all packets arriving at the NetFPGA card are processed locally on the card and CPU RX Q and CPU TX Q ports are unused.

Although virtual networks may be statically assigned to either software or hardware data planes during network allocation, several practical reasons require networks to be dynamically migrated between the two platforms during operation. First, from a service provider’s standpoint, the initial virtual network allocation may not be sufficient to support the dynamic QoS requirements of virtual networks during operation. Second, from an infrastructure provider’s standpoint, shifting lower-throughput networks to software and higher-throughput networks to hardware can improve the overall utilization of the virtualization platform. Additionally, network migration can
reduce the impact of data plane customization on virtual networks in shared hardware. For example, the virtual network operator can migrate unmodified virtual networks in an FPGA to software data planes, reconfigure the FPGA with data plane changes and migrate the networks back to the FPGA to resume operation at full throughput. All unmodified virtual networks can continue their operation at lower throughput using software data planes during FPGA reconfiguration.

We illustrate data plane migration by considering an example where the FPGA is shared by multiple IPv4-based virtual networks. The data plane characteristics of any FPGA-based data plane in this case can be modified using the following steps:

1. Before migration, the operator creates Click instances of all active hardware virtual data planes using the OpenVZ virtual environment.

2. The Linux kernel sends messages to all nodes attached to the network interface requesting a remap of layer-3 addresses targeted at the NetFPGA board to layer-2 addresses of the PC NIC. Each virtual network includes a mechanism to map between layer-2 and layer-3 addresses. When a virtual network uses IP, the Address Resolution Protocol (ARP) is used to do the mapping between layer-2 and layer-3 addresses. In our prototype, where IP is used in the data
plane, the ARPFaker element [5] implemented in Click is used to generate ARP reply messages to change the mapping between layer-2 and layer-3 addresses.

3. Once addresses are remapped, all network traffic redirects to the PC for forwarding with software virtual data planes.

4. The operator now reprograms the FPGA with a new bitstream that incorporates changes in network characteristics. We used a collection of previously-compiled FPGA bitstreams in our implementation.

5. Following FPGA reconfiguration, the operator writes routing tables back to the hardware.

6. In a final step, the Linux kernel sends messages to all nodes attached to the network interface requesting a remap of layer-3 addresses back to the NetFPGA interface. The virtual network then resumes operation in the hardware data plane for the instantiated hardware routers. We quantify the overhead of this dynamic reconfiguration approach in Section 3.6.

All virtual networks remain fully active in software during the reconfiguration. The traffic to virtual networks in software is forwarded through the PC NIC (Figure 3.6). We use ARP as a mechanism to map virtual IP addresses to virtual MAC addresses. Non-IP data planes can use a similar scheme by incorporating a mechanism to map the non-IP virtual addresses (such as flat labels) to the physical (MAC) addresses. Custom elements written using Click can be used to perform such mapping.

3.3.3 Scheduling Virtual Network Migration

Network service requirements and the availability of virtualization resources are also subject to realtime variations. It is therefore important to cleanly separate service requirements from virtualization resources. This separation can be achieved using a scheduling interface that maps service requirements to virtualization resources while
maximizing the overall utilization (bandwidth, latency etc.) of the virtualization platform. Our system implements a simple greedy scheduling technique to assign virtual networks to hardware or software data planes so that the overall bandwidth of the virtualization platform is maximized while aggregate bandwidth and capacity limitations in both platforms are respected. The scheduler attempts to greedily pack low-throughput virtual networks into OpenVZ containers. If a network cannot be executed in a software plane due to bandwidth limitations, it is assigned to a hardware plane. The scheduler recomputes virtual network assignments whenever a virtual network is removed from the platform or when service requirements change during operation. The output of the scheduler can be used by the operator to perform virtual network migrations.

3.4 Case Study - Data Planes

We illustrate the capabilities of the FPGA-based network virtualization platform by implementing two realistic data planes - a virtual data plane that uses conventional IP forwarding and a non-IP data plane that uses flat label lookup.

3.4.1 IPv4

The IPv4 data plane design example uses layer 3 virtualization based on IPIP tunneling [15]. Tunneling transforms data packets into formats that enable them to be transmitted on networks that have incompatible address spaces and protocols. In this tunneling approach, the network operator assigns a virtual IP address from a private address space to each node in a virtual network. To transmit a packet to another virtual node in the private address space, the source node encapsulates the packet data in a virtual IPv4 wrapper and tunnels it through intermediate routers. When the packet reaches a virtual node, the data plane uses an inner virtual IP address to identify the next virtual hop. The packet is then tunneled to its final
destination. Tunnel-based layer 3 virtualization is a popular virtualization strategy that has been deployed in many software virtualization systems such as VINI [21].

The dynamic design select module uses the destination virtual IP address (DST VIP in Figure 3.5) as an index into the design select table to determine the associated data plane. If a match to a virtual network in the FPGA is found, the dynamic design select module sends the packet to the hardware plane. The forwarding engine maps the virtual destination IP address to the next hop virtual destination IP address and rewrites the source and destination IP addresses (SRC IP and DST IP in Figure 3.5) of the packet before forwarding the processed packet through output queues.

### 3.4.2 Routing on Flat Labels

Routing On Flat Labels (ROFL) [28] uses direct host identifiers instead of hierarchical prefixes to route packets. Routing uses a greedy source-based policy. ROFL assumes that each router in the network has a unique ID assigned from a global circular namespace. The routers maintain pointers to successors and predecessors in this circular namespace and hold IDs of hosts that are registered with them (resident IDs). Additionally, each router caches source routes of previously routed packets. When a packet is received, its destination ID is compared with IDs of nodes that are available in the forwarding table. The closest ID in the namespace is then selected. The router also checks for an entry from cached source routes. The packet is forwarded to the closest of the two entries.

In our system, the ROFL data plane stores host identifiers (ID) in sorted order within a TCAM-based forwarding table. We modified the TCAM lookup algorithm to return the shortest ID match instead of the longest prefix match as in IPv4. A second TCAM implemented within the FPGA is used as a routing cache. When packets arrive at the data plane, the forwarding logic extracts the destination host ID from the packet header. The ID is then used for simultaneous searches in the
forwarding table and the routing cache. The data plane uses the lowest ID among the search results to forward the packet. The control plane of ROFL supports the OSPF protocol.

3.5 Scalability Considerations - Virtual Forwarding Tables

The architecture of forwarding tables is an important design consideration for FPGA-based virtual data planes. Typical forwarding tables need to store hundreds of thousands of entries and consume significant memory resources within the FPGA. Although the design of efficient forwarding tables for general-purpose (e.g. non-virtualized) IP-based routers has been well researched in the past [39] [68], recent advances in network virtualization have inspired researchers to revisit this problem in the context of network virtualization.

3.5.1 Related Work

Two recent research efforts investigate techniques to share memory efficiently between virtual routers. Fu and Rexford [37] present a shared data structure that exploits the overlap between forwarding table entries. The forwarding tables are initially represented in a binary tree based data structure called trie. The nodes in the trie store the next hop information while the edges represent successive bits of the forwarding table address. Each node in the trie additionally stores a bitmap that associates a virtual router with a specific forwarding table entry. The forwarding information in non-leaf nodes of the trie are successively pushed to the leaf nodes by applying a graph transformation technique called leaf pushing. The leaf nodes that store the same next hop information for all virtual routers are subsequently combined, reducing the overall memory requirement. The authors claim that up to 10 medium sized forwarding tables can be stored using a 120 Mb SRAM. However, the memory
requirements are likely to increase when forwarding table entries from virtual routers are widely dissimilar. A hardware evaluation of the algorithm has not been reported.

Song et al. [73] propose *trie braiding* to compact multiple forwarding tables into a single trie-based data structure. The forwarding tables are initially represented as independent tries. An objective of trie merging is to maximize the overlap of nodes between different tries by increasing the tries’ structural similarity. Each node in the trie stores a braiding bit that indicates the direction of traversal in the trie. The braiding bit can be used to swap the left and right sub tries. A dynamic programming based heuristic performs a series of such swaps to maximize the similarity between the tries. Similar tries can be overlapped, yielding a compact data structure. The authors claim that up to 16 separate routing tables with a total table size of 290K entries can be stored using a 36 Mbit SRAM. However, this approach suffers from slow forwarding table insertions since all braiding bits need to be recomputed for each insertion. Like the previous approach, no quantitative evaluation of the packet forwarding performance on hardware has been reported for this approach.

Although trie-based approaches are attractive, practical implementations require heavy pipelining in hardware to achieve high throughput. The hardware cost of trie-based techniques exponentially grows with longer prefix lengths. This motivates us to look at alternate approaches that store forwarding table entries from multiple virtual routers in a shared fashion while require less pipelining in hardware.

### 3.5.2 Design Challenges

The design of SRAM-based IPv4 forwarding tables for virtual routers is challenging for two reasons.

First, IP based packet forwarding uses longest prefix matching, wherein, the longest matching entry is selected to forward a packet if the destination address matches multiple forwarding table entries. Longest prefix matching is typically re-
Figure 3.7. Architecture of SRAM-based external forwarding tables

alized using single-cycle lookup ternary content addressable memories (TCAM) [10]. Since SRAMs lack parallel search mechanisms, practical lookup algorithms that are feasible in hardware and which do not rely on parallel search techniques are necessary to implement high throughput forwarding tables for virtual routers. Second, the shared use of external SRAM between multiple virtual routers can potentially lead to virtual prefix overlaps. We illustrate this issue by introducing the notion of virtual prefixes.

In layer 3 virtualization, each node of the virtual network is assigned a unique 32-bit virtual IPv4 address. Forwarding table entries consist of the prefix followed by the next hop information represented as next hop address and output port. A virtual prefix covers the address space of nodes whose most significant address bits match the prefix. Prefix overlaps are possible when virtual network operators sharing the same virtualization platform choose their prefixes independent of each other. Overlapped prefixes may map to similar locations in the SRAM leading to prefix conflicts.
3.5.3 Shared Virtual Forwarding Table Design

The high level architecture of the system that implements shared external SRAM-based forwarding tables for virtual routers is shown in Figure 3.7. The architecture extends the popular DIR-24-8-BASIC technique [39] used for high speed SRAM-based prefix lookups. The DIR-24-8-BASIC technique exploits the bias towards certain prefix lengths in typical backbone routers. For example 99.93% of IPv4 prefixes have length 24 bits or less [39]. By expanding all prefixes of length 24 bits or less and relocating these prefixes to SRAM locations that can be accessed with single memory access, the average prefix lookup time can be minimized.

In our system, each virtual forwarding table in hardware is identified by a unique identifier (VID). The 36Mbit SRAM located external to the FPGA is organized as two 18 Mbit memory banks. Each memory bank consists of $2^{19}$ (512K) entries where an entry is 36 bit wide. The first bank (L1 in Figure 3.7) stores all prefixes whose lengths are less than 19 bits. The second bank (L2 in Figure 3.7) is divided into multiple sets with each set consisting of $2^{13}$ entries. The second bank stores prefixes whose lengths are greater than 19 bits. When a virtual prefix of length $l \leq 19$ bits needs to be stored, $2^{19-l}$ entries are written into L1. Each entry is 36 bit wide and consists of a 1 bit flag, 3 bit output port and 32 bit next hop address. The flag bit is set to 0 for prefixes of length $l \leq 19$ bits. For prefixes of length $l > 19$ bits, an entry indexed by the most significant 19 bits of the prefix is written. The flag bit of this entry is set and the remaining bits point to an index location in L2. L2 reserves a set of $2^{13}$ entries for each prefix of length $l > 19$ bits. Each entry in the set corresponds to one of the longer $2^{13}$ prefixes indexed by the shared entry in L1. The entries in L2 store the 3 bit output port information followed by the 32-bit next hop entry. This approach could be scaled to cover 99% of all IPv4 prefixes with a 72 MByte SRAM.
3.5.4 Handling Virtual Prefix Conflicts

The SRAM can be conveniently shared between multiple virtual routers when prefixes do not conflict with each other. However, when virtual prefixes from two or more virtual routers conflict, they index to one or more exactly similar locations in SRAM. The prefix conflict can be resolved by relocating the overlapped prefix to an unoccupied location available in L1 or L2.

The relocation is performed in three steps.

1. The software control plane calculates an indirect index to relocate the prefix in SRAM. The indirect index is determined on a first-fit basis from the available pool of SRAM locations.

2. The virtual router id (VID), original prefix and the indirect address are then written to a Conflict CAM.

3. The next hop and output port information are written to the indirectly indexed locations in SRAM.

The Conflict CAM, implemented as a TCAM within the FPGA, maps an overlapped prefix to an indirectly indexed location in either L1 or L2. Each entry in the Conflict CAM consists of the virtual prefix placed with the virtual router id (VID) of the overlapped prefix. During prefix lookups, Conflict CAM can be used to detect prefix overlaps with a single cycle overhead. We discuss the design considerations regarding the size of Conflict CAM in section 3.6.

Routing Table Updates: Routing table updates to SRAM-based virtual routing tables are primarily handled as control plane operations in host software. Algorithms 1 and 2 describe the prefix update mechanism. Before a prefix can be written, the software must detect and resolve prefix collisions. The software maintains an array of status bits that reflect the availability of SRAM locations. The unavailability of an SRAM location is indicated by setting the corresponding status bit. Before a
Algorithm 1: UpdatePrefix

Input: Prefix/length p/l, ⟨port, next hop⟩

1 if p/l does not overlap then
2 WriteEntry (p/l, ⟨port, next hop⟩)
3 else
4 index/l ← p/l
5 Conflict CAM ← ⟨vid, p/l⟩, index/l
6 WriteEntry (index/l, ⟨port, next hop⟩)
7 end

Algorithm 2: WriteEntry

Input: Prefix/length p/l, ⟨port, next hop⟩

1 if l ≤ 19 then
2 Select L1
3 for $2^{19-l}$ entries do
4 \begin{align*}
5 &L1 [p] \leftarrow 0, \langle \text{port, next hop} \rangle \\
6 &\text{statusbit}[p] \leftarrow 1 \\
7 &p \leftarrow p + 1
8 \end{align*}
9 end
10 else
11 Select L1
12 $L1[p] \leftarrow \{1, index_{L2}\}$
13 Select L2
14 for $2^{32-l}$ entries do
15 \begin{align*}
16 &L2[\text{index}_{L2}] \leftarrow \langle \text{port, next hop} \rangle \\
17 &\text{statusbit}[\text{index}_{L2}] \leftarrow 1 \\
18 &\text{index}_{L2} \leftarrow \text{index}_{L2} + 1
19 \end{align*}
20 end
21 end

virtual prefix is written to L1 or L2, the software checks all status bits corresponding to the SRAM locations of the prefix. The unavailability of at least one SRAM location indicates a prefix collision. If no collisions are detected, the prefix is directly used as an index to the SRAM. For prefixes of length $l\leq19$ bits, $2^{19-l}$ entries are written into L1 with their flag bit set to 0. For prefixes of length $l>19$ bits, an entry indexed by the most significant 19 bits of the prefix is written with the flag bit set to 1 and the remaining bits set to an index location in L2. Finally, $2^{32-l}$ locations in L2 are
Algorithm 3: LookupPrefix

\textbf{Input}: DstAddr addr, VirtualRouterId vid
\textbf{Output}: NextHop next hop, OutputPort port

1. Select Conflict CAM
2. \texttt{lookup-addr} –\langle vid, addr \rangle
3. \textbf{if} Conflict CAM entry exists \textbf{then}
   4. \texttt{p} – index
   5. \langle port, next hop \rangle – ReadEntry (p)
4. \textbf{else}
6. \langle port, next hop \rangle – ReadEntry (lookup-addr)
7. \textbf{end}
8. \textbf{return} \langle port, next hop \rangle

Algorithm 4: ReadEntry

\textbf{Input}: DestAddr addr
\textbf{Output}: NextHop next hop, EgressPort port

1. Select L1
2. \texttt{entry} – L1 [addr]
3. \textbf{if} MSB of entry equals 0 \textbf{then}
   4. \langle port, next hop \rangle – entry
5. \textbf{else}
6. \texttt{index}_{L2} – entry
7. \texttt{offset} – (13 LSBs of addr)
8. \texttt{addr} – (\texttt{index}_{L2} + \texttt{offset})
9. Select L2
10. \texttt{entry} – L2 [addr]
11. \langle port, next hop \rangle – entry
12. \textbf{end}
13. \textbf{return} \langle port, next hop \rangle

updated with the next hop information. In either case, the status bits in software are set following the prefix update.

If a prefix collision is detected, the software calculates an indirect index from the pool of available SRAM locations. The virtual prefix in conjunction with the virtual router ID and the generated indirect index is written into the Conflict CAM. The overlapped prefix is updated at the SRAM location indirectly indexed by the Conflict CAM entry.
**Routing Table Lookups:** The address lookup procedure is described in Algorithms 3 and 4. When a packet is received, its destination virtual IP address is extracted from the packet header. The virtual IP is added with the virtual router id information from the dynamic design select module to construct a *lookup address*.

Next, a search is performed for the lookup address in the Conflict CAM. If a match is found, the indirect index obtained from the Conflict CAM is used to index L1. Otherwise, the virtual IP address is directly used as an index into the L1 table. The most significant bit (MSB) of the L1 table entry is examined to see if an additional memory access is required or not. If the MSB is 0, no additional memory access is required and the next hop information can be directly obtained from L1 table. Otherwise, L1 entry is combined with least significant 13 bits of the prefix to obtain an index into L2. Subsequently, the next hop and output port information are retrieved from L2. Thus, short prefixes ($l \leq 19$) require only a single memory access while longer prefixes ($l > 19$) require an additional memory access. An experimental evaluation of the packet forwarding performance of the architecture is presented in Section 3.6.

Each ROFL virtual data plane uses a forwarding table to store ordered resident host IDs and a pointer cache to cache recent source routes. We implement the forwarding table in external SRAM since it is likely to use more memory resources than the pointer cache. The pointer cache is implemented using the TCAM memory within the FPGA. The control plane maps the circular namespace of each virtual router onto a continuous block of SRAM locations. This mapping is achieved by using a hash of the virtual router ID (VID) and the namespace base address. Several virtual routing tables can share the SRAM by partitioning the SRAM into multiple namespaces, each belonging to a virtual router. Each SRAM location corresponds to a label in the namespace. The forwarding table stores only a limited set of labels (valid labels). The SRAM locations corresponding to these labels store the egress port information.
for these labels. All other labels (invalid labels) store the egress port information of the closest label in the namespace.

**Updates:** To store a new label within a namespace, the control plane software updates the corresponding location in SRAM with the egress port information of the new label. Additionally, the egress port information of all previous invalid labels are set to the new egress port information.

**Lookups:** The data plane hashes the virtual router ID and destination ID extracted from the packet header into an SRAM location corresponding to the namespace label. Simultaneously, the FPGA forwarding logic searches for the label in the data plane’s local pointer cache. The egress port information from the SRAM namespace is compared with the results from the pointer cache and the lowest of the two entries is used to forward the packet.

### 3.6 Evaluation

We evaluate the performance of our system by measuring the throughput, latency and resource usage of data planes. In addition, we analyze the scalability of the system and report the overhead of virtual network migration during FPGA reconfiguration. The following sections describe some of the techniques used to obtain experimental results.

In an initial experiment, we compared the baseline performance of a single hardware virtual data plane running in the NetFPGA hardware and a Click software virtual data plane running in the OpenVZ container. Figure 3.6 shows the testbed network used in our experiments. We used the NetFPGA packet generator/capture tools to generate traffic of different packet sizes and rates. We loaded the packet generator with PCAP files [31] whose packet sizes ranged from 64 to 1024 bytes. These packets were subsequently transmitted to the system at the line rate of 1 Gbps.

We consider four specific system configurations:
1. Hardware data plane with external SRAM routing tables - The NetFPGA board receives and transmits all packets. The forwarding tables are stored in a 4.5 Mbyte SRAM located external to the FPGA.

2. Hardware data plane with TCAM routing tables - The NetFPGA board receives and transmits all packets. The forwarding tables are stored in a 32 entry TCAM located within the FPGA.

3. Click from NIC - The PC NIC (Figure 3.6) interfaces receive network traffic and use Click data planes executing in OpenVZ containers to forward packets.

4. Click from NetFPGA - The NetFPGA network interfaces receive the traffic. Click data planes forward the packets in OpenVZ containers. The PCI bus transfers packets between the NetFPGA hardware and the OpenVZ container.

**3.6.1 Throughput**

The throughput of the four approaches for differing packet sizes is shown in Figure 3.8. These values show the maximum achievable throughput by each implementation for a packet drop rate of no more than 0.1% of transmitted packets. We measured the receiver throughput using hardware counters in the NetFPGA PktCap capture tool.

The throughput of shorter packets drops considerably in the software-based implementations. In contrast, the single hardware virtual data plane consistently sustains throughput close to line rates for all packet sizes. The hardware provides one to two orders of magnitude better throughput than the OpenVZ Click router implementations due to inherent inefficiencies in the software implementation. The OpenVZ running in user space trades off throughput for flexibility and isolation.

**Discussion:** The performance degradation in software implementations results from frequent operating system interrupts and system calls during packet transfers.
Figure 3.8. Receiver throughput versus packet size for a single virtual router between user space and kernel space. For smaller sized packets, the frequency of packet arrivals at the forwarding interface increase during any given time interval. The forwarding overhead increases at higher transmission rates, and eventually packets are dropped at rates higher than a threshold. Effectively, this translates to an increase in packet forwarding overhead and lower forwarding rates.

The 10-100× improvement in hardware datapath results from the data-parallel nature of packet forwarding path. In NetFPGA, packets that arrive at the input queues are processed using a 64-bit wide pipelined datapath composed of multiple stages. A 64-bit packet word can be transferred from stage to stage during every clock cycle. When the clock runs at 62.5MHz, the datapath offers a peak throughput equal to 62.5M×64 = 4Gbps. Unlike software forwarding approaches, the throughput does not drop for smaller-sized packets due to the pipelined nature of the design.

3.6.2 Latency

We use the experimental setup shown in Figure 3.9 to measure the latency of all four configurations mentioned above. Unlike our previous work that used the ping
utility for latency measurements [80], the latency experiments described here use the hardware-based NetFPGA packet generator to accurately generate and capture network traffic. While standard software utilities can only measure network latencies on the order of several milliseconds, the NetFPGA packet generator operating at 125 MHz can report latencies with an accuracy of ±8 ns. In our test setup, we configured ports 0 and 1 of the packet generator in the loopback configuration to provide a baseline measurement while ports 2 and 3 were attached to the experimental virtual router. We simultaneously transmitted two packets of size 64 bytes through ports 0 and 2 and later captured the forwarded packets from ports 1 and 3. The difference in the arrival timestamp values of the two packets indicate the latency of the experimental data plane. We averaged the observed latencies across ten repeats of the experiment.

Table 3.1 shows the latency of a single data plane for all four configurations. For SRAM hardware data planes, we separately evaluated the performance of short (length \( \leq 19 \) bits) and long prefixes (length \( > 19 \) bits) to examine the overhead of two-level memory access required for long prefix lookups in external SRAM. In general, the hardware data planes incur one to two orders of magnitude less latency than software data plane implementations. Although the external SRAM-based forwarding table requires 5 additional cycles for each short prefix lookup than its TCAM counterpart, the observed network latency increases by only 0.1 msec. The moderate increase is justifiable given the large number of prefixes that can be stored in the external SRAM.
Table 3.1. Dataplane latency for IPv4 and ROFL - Both long and short prefixes are used

<table>
<thead>
<tr>
<th>Data plane</th>
<th>Configuration</th>
<th>Prefix Type</th>
<th>Cycles / Freq (Mhz)</th>
<th>Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4</td>
<td>Hardware data plane (TCAM)</td>
<td>Short/Long</td>
<td>1/62.5</td>
<td>3.01</td>
</tr>
<tr>
<td></td>
<td>Hardware dataplane (SRAM)</td>
<td>Short</td>
<td>6/62.5</td>
<td>3.02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Long</td>
<td>21/62.5</td>
<td>3.17</td>
</tr>
<tr>
<td></td>
<td>Click from NIC</td>
<td>Short/Long</td>
<td>-</td>
<td>262.30</td>
</tr>
<tr>
<td></td>
<td>Click from NetFPGA</td>
<td>Short/Long</td>
<td>-</td>
<td>408.20</td>
</tr>
<tr>
<td>ROFL</td>
<td>Hardware data plane (TCAM)</td>
<td>-</td>
<td>1/62.5</td>
<td>2.45</td>
</tr>
<tr>
<td></td>
<td>Hardware dataplane (SRAM)</td>
<td>-</td>
<td>4/62.5</td>
<td>2.40</td>
</tr>
</tbody>
</table>

Longer prefixes incur an additional 15 cycles due to two memory accesses, resulting in a 5% increase in the observed latency. The ROFL data plane uses 4 cycles for each lookup.

The additional cycles consumed for SRAM-based IP lookup and ROFL lookup does not necessarily limit the packet forwarding performance. In fact, the impact of higher latency on the overall throughput of the virtualization platform can be hidden by exploiting the pipelined nature of the design. We determined that a 32x32 FIFO buffer inserted between the forwarding logic and the dynamic design select module is sufficient to sustain the line throughput (1 Gbps). The resultant increase in the FPGA logic requirement was less than 1%.

3.6.3 Network Scalability

Network scalability can be measured in terms of both throughput and latency. For these experiments, we configured the test topology as shown in Figure 3.6. Six specific system configurations were considered for systems that consisted of 1 to 15 virtual networks. The software-only Click from NIC and Click from NetFPGA cases are the same as defined in Section 3.6.

Additional cases which combine NetFPGA and software data planes include:
1. Hardware+Click from NIC (SRAM) - The PC NIC receives and transmits all network traffic targeted to OpenVZ-based virtual networks. The NetFPGA physical interfaces receive and transmit all network traffic targeted to FPGA-based virtual networks. This case represents the multiple receiver approach described in Section 3.3. The hardware virtual data planes use external SRAM-based forwarding tables.

2. Hardware+Click from NIC (TCAM) - This approach similar to case 1 except that hardware virtual data planes use on-chip-TCAM based forwarding tables.

3. Hardware+Click from NetFPGA (SRAM) - The NetFPGA network interfaces receive and transmit all network traffic. Hardware virtual data planes perform some of the forwarding operations while the rest are handled using Click data planes in OpenVZ containers. For the latter cases, packets are transferred between the NetFPGA hardware and OpenVZ over the PCI bus. This case represents the single receiver approach described in Section 3.3. The hardware virtual routers use external SRAM-based forwarding tables.

4. Hardware+Click from NIC (TCAM) - This approach is similar to case 3 except that hardware virtual data planes use on-chip TCAM based forwarding tables.

For cases 2 and 4, we implemented up to four virtual data planes in the FPGA and the rest (up to 11) as Click processes executing within OpenVZ containers. For cases 1 and 3, we deployed up to three virtual data planes in the FPGA and remaining networks (up to 12) in software. The setup to measure transmission latency for the four cases is shown in Figure 3.9. As shown in Figure 3.10, the average network latency of the Click OpenVZ virtual router is approximately an order of magnitude greater than that of the hardware implementation. The latency of OpenVZ increases by approximately 15% from one to fifteen virtual data planes. This effect is due to context switching overhead and resource contention in the operating system. Packets
Figure 3.10. Average latency for an increasing number of IPv4-based virtual data planes

routed through OpenVZ via the NetFPGA/PCI interface incur about 50% additional latency overhead than when they are routed through the NIC interfaces. The average latency of hardware data planes remains constant for up to four data planes. After this, every additional software router increases the average latency by 2%.

To measure aggregate throughput when different numbers of virtual data planes are hosted in our system, we transmitted 64 byte packets with an equal bandwidth allocated for all networks. Next, we incrementally increased the bandwidth share of each virtual network until the networks began to drop more than 0.1% of the assigned traffic. A single OpenVZ software virtual data plane can route packets through the PC NIC interface at a bandwidth up to 11 Mbps. The throughput dropped by 27% when fourteen additional software data planes were added. The software virtual data plane implementation which routes packets from the NetFPGA card to the OpenVZ containers can sustain only low throughput (approximately 800 Kbps) with 64 byte packets and 5 Mbps with 1500 byte packets due to inefficiencies in the NetFPGA PCI interface and driver. The FPGA sustains close to line rate aggregate bandwidths for
Figure 3.11. Average throughput for an increasing number of IPv4-based virtual data planes

up to four data planes. The average aggregate bandwidth dropped when software data planes are used in addition to FPGA-based data planes.

The top two plots (HW+Click from NIC and HW+ Click from NetFPGA), which overlap in Figure 3.11, show the average aggregate throughput when software data planes are used in conjunction with hardware data planes. Since the hardware throughput dominates the average throughput for these two software data plane implementations, minor differences in bandwidth are hidden. Further, the use of a log scale hides minor differences in throughput between the two software implementations.

Systems which contain more than the four virtual data planes implemented in hardware exhibit an average throughput reduction and latency increase as software data planes are added. For systems that host a range of virtual networks with varying latency and throughput requirements, the highest performance networks could be allocated to the FPGA while lower performing networks are implemented in software.
3.6.4 Overhead of Dynamic Reconfiguration

To evaluate the cost and overhead of dynamic reconfiguration, we initially programmed the target FPGA with a bitstream that consisted of a single virtual data plane. Next, we sent ping packets to the system at various rates which were then forwarded using the NetFPGA hardware plane. Next, we periodically migrated the hardware plane to an OpenVZ container in host software using the procedure described in Section 3.3. After FPGA reconfiguration, we moved the data plane back to the NetFPGA card. We determined that it takes approximately 12 seconds to migrate a hardware data plane to a Click router implemented in OpenVZ. The FPGA reconfiguration, including bitstream transfer over the PCI bus, required about 5 seconds. Transferring the virtual router from software back to hardware took around 3 seconds. The relatively high hardware-to-software migration latency was caused by the initialization of the virtual environment and the address remapping via ARP messages. The software to hardware transfer only requires writes to forwarding table entries over the PCI interface. Our experiments show that if a source generates packets at the maximum sustainable throughput of OpenVZ-based data planes, our system can gracefully migrate the virtual router between hardware and software without any packet loss.

3.6.5 Frequency of Dynamic Reconfiguration

To examine the impact of frequent dynamic reconfiguration on a data plane implemented in an FPGA, we performed an analysis based on experimentally-determined parameters. Consider a situation where a hardware data plane is unchanged for an extended period of time, but must be occasionally migrated from hardware to software when a different hardware data plane is updated or replaced. The overall bandwidth of the unchanged data plane can be represented as:
\[ B_{\text{avg}} = \frac{B_{\text{sw}} \ast t_{\text{reconfig}} + B_{\text{hw}} \ast (T - t_{\text{reconfig}})}{T} \]  

(3.1)

where \( B_{\text{hw}} \) represents the aggregate bandwidth of FPGA data planes, \( B_{\text{sw}} \) represents the aggregate bandwidth of software data planes, \( t_{\text{reconfig}} \) represents the time required to update the FPGA including FPGA reconfiguration time, and \( T \) represents the period of time between FPGA reconfigurations. For our analysis, we assume that four FPGA-based data planes with an individual throughput of 1 Gbps (\( B_{\text{hw}} = 1000 \) Mbps) are reconfigured every 12 seconds (\( t_{\text{reconfig}} = 12 \) s), based on our experimentally-collected results. During reconfiguration, all active virtual networks are migrated to host software using the procedure described in Section 3.3 and software data planes offer an aggregate throughput of 11 Mbps with 64 byte packets (\( B_{\text{sw}} = 11 \) Mbps). Based on (3.1), if reconfiguration is performed every 15 seconds, the average throughput (\( B_{\text{avg}} \)) of unchanged hardware datapath drops from 1 Gbps to 200 Mbps. However, if reconfiguration takes place once every 2 minutes, the average throughput only drops 10% to about 900 Mbps.

### 3.6.6 Cost Analysis

Table 3.2 provides a cost/benefit analysis for different virtual networking systems. We assume that a PC can support 60-100 virtual networks on the basis of different host virtualization strategies (full/container virtualization) [23] and offer packet forwarding rates between 10Mbps-40Mbps [23]. The NetFPGA 1G board costs $1300 and can support up to 5 virtual networks. Since a pure FPGA virtual networking system such as the one described in [18] can only accommodate up to 5 virtual networks, a higher cost per virtual network (approx. $260 per virtual network) will be incurred although each virtual network can operate at two orders of better throughput rates when compared to a standard PC. In contrast, a heterogeneous system like ours can offer different cost/throughput choices to virtual networks. For example, up to 5 virtual networks can operate at two orders of better throughput and a higher number of
Table 3.2. Cost and throughput for virtual networking systems

<table>
<thead>
<tr>
<th>System</th>
<th>Max. Virtual Networks</th>
<th>Cost</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>60-100</td>
<td>$600</td>
<td>10Mbps - 40Mbps</td>
</tr>
<tr>
<td>NetFPGA 1G</td>
<td>5</td>
<td>$1300</td>
<td>1 Gbps</td>
</tr>
<tr>
<td>NetFPGA 1G+PC</td>
<td>65-105</td>
<td>$1900</td>
<td>1.01 Gbps-1.04 Gbps</td>
</tr>
</tbody>
</table>

virtual networks (approx 60-100) can operate at lower forwarding rates (10 Mbps). The system will require $\times 3$ increase in the overall system cost in comparison to a standard PC. It is likely that the system cost per virtual network may amortize when higher volumes of heterogeneous virtual networking platforms are deployed.

3.6.7 Resource Usage

When internal forwarding tables are used, the Virtex II Pro FPGA can accommodate a maximum of five virtual data planes, each with a 32-entry TCAM-based forwarding table. When the CPU transceiver module is included, the FPGA can accommodate a maximum of four virtual data planes. Each virtual data plane occupies approximately 2000 slice registers and 3000 slice LUTs. A fully-populated design uses approximately 90% of the slices and 40% of the BRAM. Table 3.3 shows the resource utilization of up to five IPv4 virtual data planes and a single ROFL data plane. All designs operate at 62.5 MHz. Synthesis results for the virtual router design implemented on the largest Virtex 5 (5vlx330tff1738) show that a much larger FPGA could support up to 32 IPv4 virtual data planes.

When external SRAM based forwarding tables are used, the FPGA can only store up to 3 virtual data planes. We attribute the reduction in the number of data planes to the additional overhead of DRAM arbitration logic used for implementing the input and output queues. The DRAM arbitration logic alone consumes about 15% of the overall FPGA resources. A hardware virtual data plane that incorporates the DRAM and SRAM arbitration controllers with a 32-entry Conflict CAM consumes 66% of the total slices and 47% of the total registers. However, we do not expect the logic
Table 3.3. Resource utilization of IPv4 and ROFL data planes

<table>
<thead>
<tr>
<th></th>
<th>TCAM Lookup</th>
<th>SRAM Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ROFL</td>
<td>IPv4</td>
</tr>
<tr>
<td></td>
<td>#Planes</td>
<td>Slices</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10321</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10068</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12882</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>15696</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>18509</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>21322</td>
</tr>
</tbody>
</table>

cost of the arbitration logic to scale with the number of virtual data planes. Larger FPGAs such as Virtex 5 will be able to amortize the additional cost with additional data planes.

3.6.8 Size of Conflict CAM

The size of the Conflict CAM is an important design consideration for hardware IPv4 data planes since it uses internal FPGA memory resources to store overlapped prefixes. The size of the Conflict CAM depends heavily on the amount of prefix overlaps between different virtual data planes. Unfortunately, for experimental purposes it is difficult to estimate the amount of prefix overlaps due to the lack of availability of realistic virtual router forwarding tables.

The RIS [14] project provides snapshots of Border Gateway Protocol (BGP) routing tables collected from Internet backbone routers. Although these sample routing tables contain large numbers of prefixes, they do not necessarily represent realistic forwarding tables since the prefixes generally tend to be highly similar across tables.
Table 3.4. Percentage of prefixes which overlap

<table>
<thead>
<tr>
<th>BGP Table</th>
<th>Total Prefixes</th>
<th>Prefix Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc12</td>
<td>339K</td>
<td>13.0%</td>
</tr>
<tr>
<td>rrc13</td>
<td>346K</td>
<td>12.6%</td>
</tr>
<tr>
<td>rrc15</td>
<td>339K</td>
<td>15.0%</td>
</tr>
<tr>
<td>rrc16</td>
<td>345K</td>
<td>13.8%</td>
</tr>
</tbody>
</table>

Song et al. [73] observed that virtual routers in the future Internet are unlikely to have similar prefixes. Existing VPN services, for instance, largely use dissimilar prefixes with different prefix aggregation schemes. Hence, for our analysis, we construct synthetic forwarding tables by partitioning four existing publicly available BGP routing tables, as shown in Table 3.4.

We uniformly distribute a set of 100K prefixes chosen randomly from each BGP table between four virtual forwarding tables. Next, we calculate prefix overlaps for each virtual data plane and then average across all four virtual routers. In general, each forwarding table exhibits 12-15% prefix overlap with prefixes found in other tables. Each overlapped prefix in a system with \( n \) virtual data planes needs \( \log(n) \) bits for the virtual ID, 32 bits for the virtual prefix and 19 bits for the indirect index. A system with 4 FPGA-based virtual data planes that stores 100K prefixes with 13% prefix overlap will need approximately 663 Kbits of on-chip memory for the Conflict CAM. The on-chip resources of modern FPGAs such as Virtex-5 are sufficient to address this memory requirement.

3.7 Conclusion

This chapter described a heterogeneous network virtualization environment that uses host virtualization techniques to scale existing FPGA-based virtualization platforms. An important contribution of this work is the development of a scalable virtual networking environment that includes both hardware and software data plane imple-
mentations. A full suite of architectural techniques are used to support this scalable environment including dynamic FPGA reconfiguration and a forwarding table for the FPGA routers which is optimized for virtual routing.
CHAPTER 4

CUSTOMIZING VIRTUAL NETWORKS WITH PARTIAL RECONFIGURATION

4.1 Introduction

The co-existence of virtual networks on shared resources necessitates effective isolation of virtual routing instances from each other. Isolation is an important characteristic from a traffic management, autonomy and security perspective. For example, network operators require effective traffic isolation policies to enforce Quality-of-Service (QoS) guarantees to virtual networks. Isolated routing instances are also essential to independently implement, customize and manage diverse data plane/control plane mechanisms in the network core. Finally, without effective isolation, virtualization opens up opportunities for a malicious routing instance to interfere and attack other virtual routing instances.

An ideal virtualization platform must support strong resource and logical isolation. Host network virtualization techniques (e.g. VINI [21] and PlanetLab [51]) implement logical isolation of virtual routing instances by splicing physical resources such as CPU cycles, physical memory, network bandwidth among virtual containers. Virtual network administrators can use CPU reservations and rate limiting policies in the hypervisor to implement customized isolation policies. By running independent network stacks in the virtual containers, host virtualization techniques also provide the ability to independently customize most aspects of the network stack.

FPGA-based network virtualization platforms introduced in previous research and in this dissertation supports strong resource isolation. For example, the architecture
presented in [18] and in Figure 3.4 reserves separate logic-elements for each data plane. However, in this approach, customization of individual data planes requires reconfiguring the entire FPGA (static reconfiguration). Virtual networks, other than the one being modified, will need to be stopped during the reconfiguration period, causing traffic disruption and loss of logical isolation. Static reconfiguration, therefore, limits the logic isolation between the shared virtual networks. Static reconfiguration has the additional drawback that the overhead of reconfiguration grows linearly grows with the number of virtual networks sharing the FPGA substrate. The overhead results from the need to migrate all the shared virtual networks into software before the reconfiguration can be performed.

This chapter presents an architecture that exploits partial reconfiguration to address the isolation and reconfiguration overhead issues associated with static reconfiguration. Partial reconfiguration allows selective regions of the FPGA to be reconfigured while the device is in operation. To evaluate this architecture, we compare and contrast partial reconfiguration and static reconfiguration approaches presented in chapter 3. For both approaches, we compare (i) the reconfiguration interval, (ii) the impact of traffic on shared virtual networks during the period of reconfiguration and (iii) the impact of the two reconfiguration strategies on the average bandwidth of virtual networks in the substrate. We also evaluate these techniques when throughput of virtualized networks change over time.

The remainder of this chapter is organized as follows. Section 4.2 provides a background on partial reconfiguration. We survey previous work that uses partial reconfiguration in networking systems. Next, section 4.3 presents the details of the dynamically reconfigurable network virtualization platforms. The experimental methodology used to evaluate the system is described in Section 4.4 and experimental results are discussed in Section 4.5.
4.2 Background on Partial Reconfiguration

Partial reconfiguration allows a selective region of the FPGA to be reconfigured while the rest of the device is still operating. Partial reconfiguration greatly enhances the flexibility of FPGA implementations by allowing parts of the application to be implemented as independent modules that may be dynamically swapped in and out of the reconfigurable hardware. Examples of such applications include communication systems which require dynamic selection of encoding/decoding algorithms based on channel noise or security applications which require varying standards of encryption based on the confidentiality level. When compared to static reconfiguration, partial reconfiguration is often fast because only a small region of the silicon is frequently reprogrammed. Selective reconfiguration also facilitates only active parts of the application to be incorporated into the bitstream, saving precious FPGA area.

Partial reconfiguration requires a design flow that is slightly different from conventional FPGA application development. In the partial reconfiguration flow, designers must partition the application into separate static and dynamic regions. The static region of the application remains unchanged during the application lifetime. The dynamic region of the application may be selected at runtime from one of the many available configurations for that region. For example, in the communication application example, the implementation of encoding/decoding algorithms can be pre-compiled into multiple configurations, while the rest of the design can remain as part of the static region. The configurations for both static and dynamic regions are independently synthesized into individual bitstreams. A specialized software allows designers to define the layout of static and dynamic regions in the FPGA. The application is composed by integrating the static and dynamic regions. The configuration for the dynamic region may be dynamically swapped into the FPGA from a pool of recompiled bitstream configurations. Specific details on the partial reconfiguration flow are described in section 4.3.
Partial reconfiguration has been used in a variety of networking systems. The Field Programmable Port Extender (FPX) system [55] uses a partially-reconfigurable Xilinx FPGA to implement a high-speed switch. The FPX system allows packet processing functions to be implemented as reconfigurable modules. Simplified reconfiguration interfaces in the form of standardized APIs are used to adapt the modules [74]. A reconfigurable accelerator for packet processing functions in network processors [59] allows customization of common networking tasks such as tree lookup and pattern matching through partial reconfiguration. The feasibility of this approach has been demonstrated using a network intrusion detection application. A dynamically-reconfigurable network processor [44] allows specific parts of a network processor to be reconfigured to meet the specific workload characteristics. The approach was validated using IP forwarding, encryption and media processing flows on Virtex II and Virtex 4 devices. Although steps in a similar direction, these approaches are not directly applicable for multiple virtual routers used by virtual networks.

4.3 A Partially Reconfigurable Network Virtualization Platform

A significant research contribution of this chapter is a network virtualization system that offers the ability to independently customize hardware data planes without the need to full reconfigure of the entire FPGA. To achieve this, we build upon our network virtualization platform presented in chapter 3.

The detailed architecture of the system is shown Figure 4.1. The hardware virtual routers in the system are implemented on a Xilinx Virtex II Pro device which is partially reconfigurable. The Virtex II Pro device is interfaced to four 1 Gbps Ethernet interfaces and SDRAMs on the NetFPGA board. The board is connected to a PC via the PCI interface. Software virtual routers are implemented using container virtualization on the host workstation.
Figure 4.1. Detailed system implementation of a partially-reconfigurable network virtualization platform on a NetFPGA board and workstation.

To support virtual router isolation and facilitate partial reconfiguration, the FPGA is divided into static and partially-reconfigurable regions (PRR). This approach contrasts with previous approaches to FPGA-based network virtualization [18] [80] that do not isolate hardware virtual routers in specific FPGA regions. The static region holds the modules that are shared across multiple virtual routers. These modules include the input arbiter, packet classifier and the output queues. The MAC RX/TX queues interface to the physical MAC and the input arbiter, while the CPU RX/TX queues interface to the host workstation via the PCI bus and the input arbiter. The static region also holds a CPU transceiver module to facilitate the implementation of additional virtual routers in the host software.

Isolated features of virtual routers are implemented in partially-reconfigurable regions. Specific functions in these regions include header verification, checksum verification, IP lookup, ARP lookup and time to live (TTL) updates. These functions are grouped into the Fwd Logic block in Figure 4.1. A forwarding table for each reconfigurable virtual router is stored in block RAMs (BRAMs). The tables can be updated via the PCI bus by control planes running in host software. The PR regions can be configured by downloading partial bitstreams over a JTAG interface.
Specific details of partial bitstream generation are described in Section 4.4. The packet interface between static and partially-reconfigurable regions consists of FPGA bus macros.

4.3.1 Partial Bitstream Generation

Partial FPGA reconfiguration requires a priori generation of partial bitstreams for all virtual routers. For our design, virtual routers with column-based FPGA resources are generated in advance of system execution via synthesis and placement constraints and stored in a library. Virtual routers are swapped into the FPGA at run time as needed. In our implementation, slice-based, synchronous bus macros with 8-bit data widths are used as interfaces between the reconfigurable virtual routers and the static logic. All the nets between the static and reconfigurable regions with the exception of global and clock signals are connected through bus macros. The clock to the partially-reconfigurable region is fed from global clock buffers in the static region. The early-access partial reconfiguration (EAPR) [85] design methodology from Xilinx is used to create partial bitstreams. The EAPR methodology requires the designer to follow the following series of steps for generating partial bitstreams.

The static and dynamically-reconfigurable portions are described using distinct sets of Verilog files. A top-level file is created which describes both static and partially-reconfigurable regions and bus macros used for inter-region interfacing. Each portion is synthesized to logic blocks and memory components under timing constraints. Resource counts are evaluated to ensure dynamically-reconfigurable portions are appropriately sized to fit in FPGA columns. Constrained placement is performed for the two design portions using the Xilinx ISE Constraints Editor. The FPGA regions for the static and partially-reconfigurable sections are manually identified using the PlanAhead Layout Editor. The partially-reconfigurable sections can be used for any of the synthesized dynamically-reconfigurable planes. Following placement, timing
analysis using timing constraints is performed with the ISE Timing Analyzer. Finally, the static and partially-reconfiguration designs are assembled and the respective bitstreams are generated.

Figure 4.2 illustrates the layout of a Virtex II Pro device with one reconfigurable virtual router located on each side of the static region. In the Virtex II Pro device, an entire column in a partially reconfigurable region must be reprogrammed at once using a partial bitstream [85]. Multiple reconfiguration regions cannot be placed within the same column. The operation of the device continues unaffected while one or more columns are reconfigured.

Bitstreams generated using the EAPR flow are downloaded using the Xilinx Impact tool via the JTAG interface running at 12 MHz. Given the small size of the target Virtex II device, a maximum of two virtual routers can be implemented in the FPGA. Each virtual router can be dynamically assigned a configuration from Table 4.1 through partial reconfiguration. The first configuration (Configuration I) follows forwarding based on destination IP addresses. The second configuration (Configuration II) forwards packets using flow information. In this case, packets are forwarded
Table 4.1. Experimental configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
<th>Slices/LUTs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Dest based IP routing</td>
<td>1443/1861</td>
<td>8</td>
</tr>
<tr>
<td>II</td>
<td>Flow based routing</td>
<td>1864/2348</td>
<td>8</td>
</tr>
</tbody>
</table>

by performing prefix lookups based on source and destination addresses in the packet header. Both configurations fit within a single FPGA column.

### 4.3.2 Host Virtualization

The partially-reconfigurable network virtualization platform integrates OpenVZ-based virtual routing instances similar to those described in chapter 1. For this system, all packets are received by the NetFPGA card. The destination virtual IP address is used to associate packets with hardware or software virtual routers. A programmable CAM table (Design Select Table in Figure 4.1) stores the virtual IP to virtual router mappings. Packets associated with a hardware virtual router are sent to the corresponding PRR via bus macros. Processed packets are placed into the output queues for further transmission. Packets associated with software virtual routers are sent to the CPU transceiver module which are subsequently forwarded by Click routers running in OpenVZ containers.

### 4.3.3 Dynamic Virtual Network Allocation

The partially-reconfigurable network virtualization system also allows a virtual network operator to migrate a virtual network between hardware and software virtual routers by modifying entries in the Design Select Table and reconfiguring the routers. Our system includes a virtual network allocator that services virtual network service requests.

Virtual network service requests fall into three categories: (1) a new virtual network is added to the system, (2) a virtual network is removed from the system, or (3) the bandwidth of an existing virtual network is modified. To support changes,
an allocation algorithm which supports the following system updates has been implemented:

**Virtual network removal**: If a removal request is made, the hardware or software virtual network is removed. All other virtual networks are left in place. A hardware-based virtual router can be removed by programming a blank partial bit-stream into the selected reconfiguration region. A software-based virtual router can be removed by destroying the OpenVZ container.

**Virtual network addition**: If sufficient bandwidth is available, a new software virtual network is created upon request. If not, the network is allocated in hardware. If neither allocation is feasible, the request is rejected.

**Virtual network bandwidth adjustment**: A request for a bandwidth reduction is applied to the affected virtual network in the system. Other networks are unaffected. If the bandwidth of an existing virtual network is increased, the allocation of all virtual networks in hardware and software is rebalanced. In some cases networks are migrated from software to hardware and vice versa. A greedy approach is currently used to rebalance the virtual networks. For example, if needed, the lowest bandwidth hardware virtual network is migrated from hardware to software or the highest bandwidth software virtual network is migrated from software to hardware to make room in the target resource.

### 4.4 Experimental Approach

#### 4.4.1 Testbed Setting

The source-router-sink topology shown in Figure 4.3 is used to measure the performance of the system. Network traffic is generated and captured with the NetFPGA packet generator tool [31] located on a separate workstation. The hardware-based packet generator can accurately generate and capture traffic at line rate (1 Gbps). The hardware-based packet generator only reports the average throughput during
4.4.2 Comparison with Previous Implementation

To justify the benefits of the partially-reconfigurable network virtualization platform, we compare this approach against the static reconfiguration approach described in chapter 3.

4.4.3 Virtex 5 Implementation

Although no in-system experiments were performed, the virtual router architecture shown in Figure 4.1 was also implemented on a Virtex 5 (VLX330T) device. Virtex 5 offers enhanced placement flexibility by allowing reconfiguration regions of arbitrary rectangular shapes to be placed within the same column. This placement flexibility combined with the availability of additional logic resources allows designers...
to implement up to 20 virtual routers in partially reconfigurable regions. Total resource usage of the system including both static and partially reconfigurable regions is approximately 68% of the entire Virtex 5 device. Each partially reconfigurable region is isolated in a rectangular shape which can be configured with a partial bitstream.

4.5 Experimental Results

The key performance parameters used in the evaluation of the system are the observed throughput of the virtual routers, traffic isolation between the virtual networks and the overhead of reconfiguration.

4.5.1 Single Virtual Router Throughput

In an initial experiment, the baseline performance of a partially reconfigurable virtual router is compared against the performance of one virtual router using the statically-reconfigurable approach, described in Section 4.4.2, and the NetFPGA ref-
Figure 4.5. Instantaneous forwarding performance for two virtual networks on a Virtex II using static reconfiguration

ference router [10]. The NetFPGA packet generator tool [31] is used to generate and capture packets at line rate (1 Gbps). All three designs operate at 62.5 MHz. Figure 4.4 compares the throughput at the receiver for different packet sizes in all three cases. The performance of the partially-reconfigurable virtual router matches the performance of the reference router and the previous statically-reconfigurable virtual router for all packet sizes. Although not shown in Figure 4.4, experiments with two partially reconfigurable virtual routers show that the combined aggregate throughput of the virtual networks for 64 byte packets is 1,953,125 packets per second (1 Gbps).

4.5.2 Instantaneous Throughput

In the next experiment, the impact of reconfiguration on forwarding performance of shared hardware virtual routers is evaluated. Consider a scenario where two virtual routers $A$ and $B$ with identical configurations (Configuration I in Table 4.1) are implemented in a FPGA. At $t=3s$, virtual router $B$ is replaced by virtual router $B'$ which implements Configuration II. Figure 4.5 shows the instantaneous throughput of each of the three virtual routers sampled every 0.5 seconds if a static reconfiguration
Figure 4.6. Instantaneous forwarding performance for two virtual networks on a Virtex II using partial reconfiguration approach is used. At the start of reconfiguration at $t=4.5s$, $B$’s throughput drops to 0, while $A$’s throughput drops by more than an order of magnitude since it has been migrated to software. Virtual router $B'$ starts forwarding packets 12 seconds later when the FPGA has completed full reconfiguration. Figure 4.6 shows the instantaneous throughput for the partially-reconfigurable case. Although $B$’s throughput drops to 0 at the start of partial reconfiguration, $A$’s throughput shows no change. After partial reconfiguration completes, full throughput of virtual network $B'$ is restored.

4.5.3 Average Throughput

Figure 4.7 indicates the benefit of using partial reconfiguration of virtual routers versus the static reconfiguration approach for the Virtex 5 device for cases when all virtual networks are located in FPGA hardware. In this experiment, it is assumed that virtual networks in the FPGA either remain static or must be configured either every 30 seconds or 180 seconds. Two cases are considered; either one or four 1 Gbps ports on the NetFPGA card are used for an overall potential throughput of 1 Gbps or 4 Gbps. The graph shows the per-virtual network throughput as the number of
FPGA-based virtual routers increases. The throughput of the partially-reconfigurable (PR) virtual routers is unaffected since all routers except the one being configured remain active during reconfiguration. However, for the static reconfiguration (SR) cases, an FPGA shutdown for 12 seconds [80] causes increased throughput loss as the number of virtual routers and network ports increases.

The frequency of reconfiguration plays an important part in the benefit of partial reconfiguration. If virtual router reconfiguration never occurs or occurs infrequently, the statically reconfigurable approach can achieve higher throughput. For example, Figure 4.8 shows the average throughput of the heterogeneous system which includes both hardware and software virtual routers if reconfiguration is never performed. The use of rigid placement regions for the partially reconfigurable virtual routers limits the number of virtual networks versus the statically-reconfigurable case. For example, a total of 2 partially reconfigurable virtual routers can be placed in a Virtex II while 4 statically reconfigurable virtual routers can be supported. For the Virtex 5, the virtual...
router count is 20 and 32, respectively. Since fewer high-speed virtual routers can be implemented in hardware, the overall throughput of the dynamically reconfigurable system drops off a bit earlier. However, since periodic virtual network reconfiguration is expected for future systems, the results shown in Figure 4.7 represent a more realistic scenario.

The run time overhead of partial reconfiguration depends on the size of the partial bitstream and the frequency of the JTAG interface. Experimental results indicate that a 680 KB bitstream can be reconfigured over a 12 MHz JTAG interface in 0.6 seconds. This number is in contrast to the 12 seconds required for full (static) reconfiguration of the same FPGA through the PCI interface, including bitstream download time.

**4.5.4 Dynamic Virtual Network Allocation**

The effects of virtual network allocation described in Section 4.3.3 were quantitatively evaluated using 1000 virtual networks whose bandwidths are distributed according to a sample bandwidth distribution measured from PlanetLab nodes [51].
Software-based virtual routers, implemented as OpenVZ containers, offer an aggregate bandwidth of 100 Mbps. FPGA-based virtual routers offer up to 1 Gbps aggregate throughput. It is assumed that virtual networks addition and removal requests arrive according to a Poisson distribution with a mean arrival period of 2 hours. The mean lifetime of a virtual network is a Poisson distribution with a mean of 64 hours. Additionally, it is assumed that the bandwidth of each active virtual network changes every hour by an amount which ranges from 0% up to a maximum variance. The change in bandwidth for each specific network is uniformly distributed up to the maximum percentage variance. A high variance value indicates large fluctuations in real-time bandwidth requirements (both increases and decreases). If a bandwidth variation increase cannot be met, the current bandwidth is maintained.

Figure 4.9 shows the percentage of successful bandwidth revisions for different variance values for cases when virtual network migration is performed and when it is not performed. A larger number of bandwidth revisions are granted when virtual networks have small fluctuations from their initial bandwidth assignments. Reallocation and virtual network migration are not needed in most of these cases. However,
Table 4.2. Dynamic power consumption in a Virtex II device

<table>
<thead>
<tr>
<th>Region</th>
<th>Dynamic Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR-Region1</td>
<td>173.38</td>
</tr>
<tr>
<td>PR-Region2</td>
<td>165.68</td>
</tr>
<tr>
<td>Static-Region</td>
<td>593.56</td>
</tr>
</tbody>
</table>

when virtual networks show large fluctuations from their current bandwidth assignments, reallocation and virtual network migration play important roles in satisfying 10-15% more bandwidth revision requests. Virtual network additions and removals were included in generating these results.

4.5.5 Power Consumption

Table 4.2 shows the dynamic power consumption of the Virtex II system running at 62.5 MHz with two IP routing data planes. The dynamic power consumption of the virtual routers is dependent on their internal structure. Total static power consumption of the Virtex II device is 158.75 mW. When a virtual router is unused, the corresponding reconfigurable region can be shut down by downloading a blank configuration bitstream, saving approximately 16% of total device power consumption.

4.6 Conclusion

This chapter demonstrated partial reconfiguration as a technique to improve the logical isolation of virtual networks sharing the FPGA-based network virtualization platform. By selectively reconfiguring parts of the chip, partial reconfiguration brings about 20x reduction in virtual network reconfiguration time. The reduction in reconfiguration time is useful in scenarios where the virtual networking platform must adapt frequently to cater to the dynamic service requirements of virtual networks.
 CHAPTER 5
RECLICK - A MODULAR DESIGN FRAMEWORK FOR FPGA DATA PLANES

In the previous chapters, we demonstrated techniques to use FPGAs as flexible high-performance network virtualization substrates. In reality, FPGAs remain inaccessible to the wider networking research community due to the lack of sequential programming models and limited opportunities for design reuse.

The traditional approach to designing a networking application with FPGAs involves several steps that include describing the application behavior in a hardware description language such as Verilog and VHDL, synthesizing the design to hardware, and optimizing the design to meet timing and area constraints. Unlike software techniques, modeling the network application behavior in behavioral/dataflow-style programming languages like Verilog/VHDL represents a paradigm shift for application designers who are accustomed to writing in sequential programming languages such as C/C++. Synthesizing and optimizing the design in hardware further necessitates detailed understanding of the FPGA architecture and timing parameters. Hardware debugging tools do not expose familiar interfaces to software developers.

Software-based tools for routing protocol specification (e.g. Click [48]) provide the ability to hierarchically compose data plane features from reusable packet processing components. Design reuse is useful since many routing protocols share similar packet processing features such as packet length calculation and checksum updates. Reusing the common packet processing blocks across multiple data planes reduces the overall design cycle time and debugging effort. Unfortunately, existing FPGA-based data
plane design frameworks for networking applications do not offer sufficient opportunities for design reuse. In many cases, the networking functionality is modeled as a monolithic behavioral block. While it may be possible to reuse blocks by carefully partitioning functionality as separate Verilog/VHDL modules, the lack of standardized interfaces and flow control mechanisms makes this process difficult. Design reuse in hardware also assumes significance in the context of limited silicon real-estate available in FPGAs.

In this chapter, we introduce ReClick, a software framework to design, deploy and reuse data plane features in FPGA-based network virtualization platforms. ReClick abstracts the intricacies of reconfigurable hardware design by providing the data plane designer a network-specific language sufficient to express many common packet processing operations. ReClick exposes an interface which is similar to Click [48], the widely-used data plane design framework for software virtual routers. Using this interface, designers can compose complex packet processing blocks from simpler ones. Further, ReClick exploits design reuse as a mechanism to optimize the resource utilization of virtual data planes within the FPGA. Data plane designs constructed using ReClick maximize packet forwarding performance through pipelining.

Designs are automatically compiled to FPGA hardware without extensive user intervention. A validation flow based on register transfer level (RTL) simulation is also in place for debugging and assessment prior to hardware deployment. A collection of pluggable modules which can be used with the framework have been developed and made available to the research community. The effectiveness of the framework is demonstrated with two data plane design examples - an IPv4 router and an IP router enhanced with onion routing capabilities. These data planes have been verified on a Virtex II FPGA available on the NetFPGA platform.

The rest of this chapter is organized as follows: Section 5.1 surveys previous programming models for FPGA-based packet processing. Section 5.2 introduces the
ReClick programming model and describes the changes introduced to the original network virtualization platform described in chapter 3 to support design modularity. Section 5.3 describes the data plane design flow. Section 5.4 illustrates the flow by providing two data plane design examples - an IPv4 router and an IPv4 router with enhanced onion routing capabilities. Finally, section 5.5 compares the packet forwarding performance and resource efficiency of the generated designs after synthesizing them onto the FPGA.

5.1 Programming Models for FPGA-based Packet Processing Systems

Several recent research attempts try to close the design gap between application development using FPGAs and software. Horta et al. [42] provide a first attempt to introduce programmability in FPGA-based packet processing systems. A module-based approach to implement reconfigurable high speed packet processing circuits is presented. Dynamic hardware plugins are assembled in hardware for single data planes using a restrictive set of directives.

NetThreads [50] uses multiprocessors constructed from the FPGA fabric (soft multiprocessors) to implement packet processing features. The soft microprocessors are embedded within the packet processing data path of a NetFPGA card. Packet processing features are described using C programs that execute on the multiprocessor system. Although writing C-style programs simplify the task of the application designer, multiple cycles required to execute packet processing tasks limit the packet forwarding performance of this approach to 5,000 packets per second.

Click [48] is a widely popular framework for building software routers. Click allows users to write configurations that describe packet processing functions as a graph of interconnected modules called elements. While configurations are written in a custom Click language, the behavior of individual elements can be described in
C++. The elements are interconnected through ports that either actively forward (push) or passively receive (pull) data. Click has been widely adopted in network research by virtue of its simple design and the availability of a diverse collection of reusable open source modules.

Nikander et al. [64] propose a tool chain that compiles C++-based Click elements to Verilog descriptions. In this approach, Click elements described in C++ are first transformed into an intermediate representation (LLVM), A set of optimizations are applied to improve the hardware synthesis characteristics. The optimized code is converted back into C code and then compiled using 3rd party C-to-Verilog synthesis tools such as AHIR [69] to generate hardware descriptions. This approach has limitations because Click C++ descriptions use virtual functions and polymorphism, that do not provide efficient hardware translation.

Brebner et al. [26] propose a system that can compile finite state machines described using high level XML descriptions to FPGA bitstreams. The packet processing system is composed of threads and hooks. Threads represent a unit of concurrency in the programmable logic while hooks provide wrappers around unconventional packet processing blocks to be interfaced to the system. The programming model, however, constrains designers to use finite state machine models, a rather nonintuitive way to describe packet processing blocks.

The G [25] [63] framework represents a first attempt to convert packet processing descriptions in a high-level language to Verilog descriptions. G uses a design philosophy that is similar to the one used by Click. Packet processing is specified as a pipeline of interconnected modules. A module can perform simple operations on the packet such as “set a field in the packet”, “insert a field after an offset in the packet” or “push a packet through a specific port”. The G language infrastructure includes a simulator and debugger for functional verification of designs. Complex packet processing operations such as packet switching and scheduling are not yet supported.
Table 5.1. Feature comparison of programming models for FPGA-based packet processing systems

<table>
<thead>
<tr>
<th>Framework</th>
<th>Frontend</th>
<th>Virtualization support</th>
<th>Module selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>[26]</td>
<td>XML</td>
<td>No</td>
<td>Static</td>
</tr>
<tr>
<td>NetThreads</td>
<td>C</td>
<td>No</td>
<td>NA</td>
</tr>
<tr>
<td>G</td>
<td>G, Click</td>
<td>No</td>
<td>Static</td>
</tr>
<tr>
<td>Chimpp</td>
<td>Verilog HDL, Click</td>
<td>No</td>
<td>Static</td>
</tr>
<tr>
<td>SwitchBlade</td>
<td>Verilog HDL</td>
<td>Yes</td>
<td>Dynamic</td>
</tr>
<tr>
<td>ReClick</td>
<td>ReClick, Verilog HDL, Click</td>
<td>Yes</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

Additionally, the proprietary nature of the framework, the lack of availability of a library of modules and the use of Xilinx-specific interconnect technology are likely to affect the popularity of the framework.

Chimpp [67] is a framework similar to G for writing Click-style packet processing descriptions on the NetFPGA platform. Modules can be parameterized using XML descriptions. Unlike G, Chimpp allows configurations to be composed of a combination of hardware and software elements. However, the behavior of hardware-specific elements must be described using Verilog/VHDL, limiting access to typical network programmers.

SwitchBlade [19] takes an alternative approach by providing a model that allows packet processing modules to be swapped in and out of the reconfigurable hardware without the need to resynthesize the hardware. Frequently-used hardware blocks are presynthesized to the FPGA in advance. Users select a subset of modules that are required to process the packet through register interfaces. The selection is later encoded in a bitmap header which is appended to incoming packets. Each module in the datapath examines the bitmap and decides whether or not to process the packet. Presynthesized elements as well as new modules need to be written in Verilog which may be a challenge for networking researchers who are not familiar with hardware design.
Table 5.1 summarizes the features supported in previously discussed frameworks. In general, these efforts are either proprietary or require designers to be familiar with hardware design knowledge. Except SwitchBlade, none of the frameworks provide a straightforward approach to virtualize the hardware. ReClick provides a modular design environment similar to Click that allows existing Click configurations to be migrated to reconfigurable hardware. New modules, designed in a hardware-agnostic language, can be dynamically reused between multiple data planes. The generated designs can be readily deployed on open hardware platforms like NetFPGA.

5.2 ReClick - Architecture and Programming Model

The work presented in this chapter makes the following specific contributions to enhance the programmability of FPGA-based network virtualization platforms:

1. An architecture for FPGA-based network virtualization featuring extensible modular data plane components. The system supports component reuse between multiple active virtual data planes in the FPGA. Pipelining is used within components to achieve the highest packet forwarding rates. The operations on packets are scheduled to minimize packet forwarding latency.

2. A software framework that describes common packet processing features of virtual data planes as a permutation of simple operations on packets, hiding hardware implementation details. A compilation framework that can translate these descriptions to area-efficient hardware descriptions.

3. A Click-like interface to compose and deploy virtual data planes from reusable data plane components.
5.2.1 Architecture of the Virtualization Platform

The ReClick programming model and architecture is explained in the context of an existing FPGA-based network virtualization platform described in chapter 3. Figure 5.1 shows the architecture of the network virtualization platform used with ReClick. The architecture implements two specific extensions from the basic system presented in chapter 3 to support extensible and modular virtual data planes.

First, the forwarding logic resources previously implemented using output port lookup modules [80] are organized as a hierarchical pipeline of smaller packet processing units. Each unit represents an independent packet processing entity with several streaming interfaces. The framework facilitates the integration of two types of packet processing units namely ReClick components and custom RTL blocks (see Figure 5.1). The fundamental difference between these two types of units lies in the way they describe packet processing behavior. ReClick components (hereafter referred to as components) are specified in the domain specific language discussed in Section 5.2.2 as a permutation of simple packet processing primitives.

The decomposition of virtual data planes into independent packet processing units provides opportunities for design reuse within the shared network virtualization plat-
form. Consider, for example, a virtual data plane that describes a new protocol, such as path splicing [61]. Such a data plane performs several conventional IP processing tasks such as time-to-live (TTL) and checksum updates. In many cases, the similarity between the virtual data planes can be exploited to reduce the area overhead of implementing virtual data plane features separately in the FPGA-based network virtualization platform. For example, a new virtual data plane can be deployed by adding a few components to an existing virtual data plane configuration or by reusing a subset of the existing data plane components.

To facilitate resource sharing, the dynamic design select table (in Figure 5.1) has been modified to associate a 32-bit bitvector tag (Vector in Figure 5.1) with each incoming packet. The bitvector tag, programmed from software through a user register, is used to select those virtual data plane components that are required to process the packet. Each bit in the bitvector corresponds to a component in the virtual data plane. For simplicity, we reserve the lower order bits in the bitvector for those components of the virtual data plane that process incoming packets first. A bit corresponding to a component is set if that particular component is used to process the packet. Each component in the virtual data plane checks its bit position in the bitvector tag associated with the packet. If the bit is set for the incoming packet, it is processed by the component. Otherwise, the packet is simply forwarded to the next module.

As an example, consider three virtual networks - black, white and grey as shown in Figure 5.1. The black virtual network does not share components with any other virtual network and hence, has its own dedicated routing resources. The white and gray virtual networks, however, share routing components (except component 3). In this case, a single data plane configuration (C), is sufficient to address the requirements of both the virtual networks. The bit vector configuration for all the networks are indicated in Figure 5.1.
5.2.2 Programming Primitives

Our framework exposes two types of programming interfaces to application developers. The first interface facilitates the development of independent packet processing components by combining a set of simple primitives. The second interface, which is similar to the software router development framework, Click, allows virtual data planes to be composed by stitching together multiple components.

Figure 5.2(a) shows a ReClick component. The component interfaces include a set of input/output ports which may include optional configuration parameters. The input ports of each component are actively driven by packet outputs from previous components. ReClick implements this push style dataflow in a manner similar to the Click modular router framework [48]. Several such components may be interconnected to form realistic virtual data plane configurations. For example, Figure 5.2(b) shows a simple virtual data plane configuration that accepts packets from the NetFPGA pipeline (e.g. from dynamic design select in Figure 5.1) via the FromDevice(NetFPGA) component, filters non-IP packets (CheckIPHeader), decrements the TTL field in the packet (DecIPTTL), modifies the packet header to be forwarded through a specific NetFPGA physical interface (DispatchToPort) and forwards the packets to the rest of the NetFPGA pipeline (e.g. output queue) via the ToDevice(NetFPGA) component. Configurations can be formulated using Click style de-
/* Instantiate components */
src :: FromDevice ( NetFPGA ) ;
checkip :: CheckIPHeader ( ) ;
ttl :: DecIPTTL ( ) ;
dispatch :: DispatchToPort ( ) ;
sink ::ToDevice ( NetFPGA ) ;
/* Interconnect component instances */

Program 1: Click description of the virtual data plane configuration in Figure 5.2

Descriptions. An example of the Click formulation of the virtual data plane in Figure 5.2(b) is shown in Program 1.

The behavior of individual components can be described in the domain specific language, ReClick, or, if preferred, by the data plane designer, using conventional RTL descriptions. Like other domain specific languages [25], the packet is the central operational entity in a ReClick component. Packets vary in size and packet sizes can exceed the datapath width of the hardware pipeline. Packet operations are therefore conducted as a sequence of operations on packet words. The packet word represents the largest quantum of packet data that can be accommodated using the hardware datapath in a single clock cycle. In a fully pipelined design, each packet word can be operated upon in a single clock cycle.

Figure 5.3 shows the first few words of an IPv4 packet processed by the NetFPGA reference router [10]. The NetFPGA reference router uses a 64-bit wide datapath. The packet word consists of one or more fields, whose contents represent meaningful information. For example, the most significant 48 bits of word 2 indicates the destination MAC address, while the lower order bits 8 to 15 of word 4 indicate the TTL information. ReClick provides a set of primitives that can characterize frequent packet processing operations (Table 5.2). These primitives can be combined with our software infrastructure to form a virtual data plane.
We illustrate the capabilities of ReClick by considering a simple design example DecIPTTL. DecIPTTL is a frequently-used packet processing component which is used to filter packets whose TTL values have expired (indicated by a value of zero in the TTL field). Program 2 describes the operation of a DecIPTTL component using the set of primitives presented in Table 5.2. The component interfaces include an input port (in0) and two output ports (out0, out1). Valid packets are forwarded via out0 to the next component while expired packets are dropped via out1. ReClick features two special datatypes - Packet and Field, in addition to standard datatypes. The Packet type is used to describe a packet, which is operated upon by the component as it transits from inputs to outputs.

The Field type is used to define packet fields within words. ReClick represents a field as a tuple of two parameters - the index of the word relative to the start of the packet and the subset of meaningful bits within that word. Standard data type variable declarations are associated with integer values that characterize the storage width. These values provide useful information for the ReClick compiler while inferring hardware components. All primitives, except `assign`, operate on packet
program 2: reclick description of a decipttl component

words. the get and set primitives are used to modify packet field information. they are described in more detail in section 5.2.3. standard expressions can be used to modify variable data or field information. the insert and remove primitives (not shown in the example) allow custom user fields to be inserted or removed from specific bit positions within the packet word. assign statements are used to associate packets arriving at the input ports of the component with packet variables.

if-else style conditional statements are supported for a subset of primitives as indicated in table 5.2. conditional statements enhance the expressiveness of the packet processing descriptions by adding flexibility to operate on packets based on static (compile-time) or run-time decisions. for example, wrapping set statements
Table 5.2. Reclik primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Arguments</th>
<th>Description</th>
<th>Conditional Execution?</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>{field, packet}</td>
<td>Extracts the field from the packet word. Assigns to user variable</td>
<td>No</td>
</tr>
<tr>
<td>set</td>
<td>{field, packet, value}</td>
<td>Set field of packet to desired value-variable</td>
<td>Yes</td>
</tr>
<tr>
<td>insert</td>
<td>{value, position, packet}</td>
<td>Insert a user defined field at the given position in the packet</td>
<td>No</td>
</tr>
<tr>
<td>remove</td>
<td>{field, packet}</td>
<td>Removes a user defined field from the packet</td>
<td>No</td>
</tr>
<tr>
<td>assign</td>
<td>{packet, port}</td>
<td>Assign inputs to packets or packets to output ports</td>
<td>Yes</td>
</tr>
</tbody>
</table>

within conditional statements enables packet values to be conditionally modified. However, the programming model supports conditional inserts and removals in an indirect fashion.

Consider a scenario as shown in Figure 5.4 where two distinct fields need to be inserted at a specific position in the packet based on the falsity or trueness of a user-defined expression. The semantics of this feature can be correctly implemented with two ReClick components as shown in Figure 5.4. The conditional forward component checks the user-defined condition and pushes the packet through one of the two available ports. The ports are attached to two distinct insert modules that perform the insert operation.

ReClick allows special variables called handlers to be defined. The handler variables are modeled as simple memory elements that store configuration parameters or packet flow statistics within components. For example, a handler variable whose value is incremented on the receipt of a first packet word can be used to keep track of the number of packets handled by the particular component. ReClick models handler variables as user registers in hardware.
Figure 5.4. Conditional inserts/removals can be implemented in an indirect fashion using Click configurations. In this example, a conditional insertion is implemented as two separate ReClick components.

5.2.3 Hardware Model

Packet forwarding performance is critical to FPGA-based virtual data planes. As a result, a ReClick component is modeled as a hardware pipeline as shown in Figure 5.5. The ReClick compiler generates the elements of the pipeline according to the packet processing behavior specified by the user. Not all pipeline elements shown in the figure are required by all component descriptions. The pipeline consists of a collection of the following set of modules:

1. **get** - The *get* module implements a table that stores the words and fields of interest in the packet. Each incoming packet word is checked against this table to extract fields of interest. The contents of the table are sequenced by the ReClick compiler.

2. **set** - The *set* module is similar to *get* except that it is used for packet modification operations. The set module includes a table that stores fields and words that need to be modified. The module identifies fields of interest in the packet.
Figure 5.5. The generic architecture of a ReClick component

word and modifies them as they are clocked out of the component. The contents of the set table are sequenced by the ReClick compiler.

3. **insert** - The *insert* module inserts fields at specific positions within the packet word and adjusts the packet length. Additional words are inserted whenever necessary.

4. **remove** - The *remove* module removes fields of interest from specific positions in the packet word and adjusts the packet length.

5. **schedule** - The *schedule* module is responsible for inter-component flow control. Additionally, it provides the ability to conditionally forward packets between multiple ports.

Packet forwarding at high throughput requires that each component is free from pipeline stalls. However, this condition is seldom the case. A write operation on a packet word whose value depends on information from words that are yet to be received by the pipeline causes a pipeline to stall. For example, a set operation on the DSTPORT (destination port) of word 1 in Figure 5.3 depends on the DSTPHI
field from word 5 and the DSTIPLO field from word 6 (destination IP address). This dependency causes the pipeline to stall at least for 6 cycles.

To address such write after read hazards, we introduce a shift buffer between the input and output ports. The size of the shift buffer is statically computed at compile time as the index of the farthest word from the first word of the packet, whose field values affect packet modification or scheduling decisions. For example, in the previous example, a shift register of 6 words is used. When packets arrive at the component’s input ports, they are successively shifted through the shift module during every cycle. The shift buffer ensures that field information from all dependent words is available before packet modification or scheduling decisions are performed.

5.3 Design Flow

The phases of the ReClick framework are illustrated in Figure 5.6. ReClick behavioral descriptions are parsed and typechecked for errors by the frontend. The scheduler examines the description to detect operations on fields that can be scheduled in the
same cycle. Specifically, fields belonging to the same word can be scheduled in the same cycle. A wider hardware datapath allows longer packet words, and hence, more field operations to be sequenced in the same cycle. However, this advantage comes at the expense of a higher hardware cost. In general, the hardware datapath width represents an important area-tradeoff parameter for the virtual data plane designer. For simplicity, we choose a 64-bit wide datapath which is similar to that used in the NetFPGA reference router architecture.

Operations that are dependent on field values from multiple packet words are scheduled according to the \textit{as soon as possible (ASAP)} schedule. Such operations are immediately scheduled when all dependent information is available from the hardware pipeline. The backend uses the schedule information to generate register transfer level descriptions in Verilog HDL. Except for the shift buffer, all component features are generated on an as needed basis. The backend generates table entries for \texttt{get} and \texttt{set} modules within the component pipeline according to the schedule determined in the previous step. Parameterizable insert and remove modules are instantiated according to the component description. Finally, the compiler generates hardware structures, such as wires and registers, to stitch together the component pipeline.

To supplement user-defined components, automatically generated RTL descriptions are added to a library for use in subsequent designs. The library supports the inclusion of additional custom RTL blocks wrapped in standard streaming interfaces that conform to the NetFPGA reference datapath. The ReClick compiler generates an RTL description for each component. We have developed a collection of library components as shown in Table 5.3. Multiple such components can be instantiated using the ReClick frontend to produce a virtual data plane description which is readily pluggable into the NetFPGA datapath.
Figure 5.7. An IPv4 router. Subfigure (a) represents a standard router. Subfigure (b) includes onion router capabilities

5.4 Example ReClick Configurations

We illustrate two design examples to demonstrate the capabilities of ReClick.

5.4.1 IPv4 Router

Figure 5.7 illustrates an IPv4 router example designed from simple ReClick components. The first two modules (CheckIPHeader and DropBroadcast) are used to filter out non-IP and broadcast packets. The lookup module is a custom RTL block which is described in Verilog HDL. The module is available for designers from a library. The lookup module extracts the destination virtual IP address in the packet and looks it up in a ternary CAM-based forwarding table within the FPGA. It also
Table 5.3. Resource Utilization and Latency of ReClick components on Virtex II

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Slices</th>
<th>FFs</th>
<th>LUTs</th>
<th>Lines of Code</th>
<th>Latency (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckIPHeader</td>
<td>Checks IP header and drops non-IP packets</td>
<td>192</td>
<td>324</td>
<td>160</td>
<td>223</td>
<td>5</td>
</tr>
<tr>
<td>DecIPTTL</td>
<td>Decrements TTL and drops expired packets</td>
<td>30</td>
<td>210</td>
<td>339</td>
<td>227</td>
<td>3</td>
</tr>
<tr>
<td>DecryptOnion</td>
<td>Decrypt packet data</td>
<td>1037</td>
<td>676</td>
<td>1155</td>
<td>291</td>
<td>6</td>
</tr>
<tr>
<td>Discard</td>
<td>Discard the packet</td>
<td>12</td>
<td>0</td>
<td>3</td>
<td>165</td>
<td>1</td>
</tr>
<tr>
<td>DispatchToPort</td>
<td>Forward packet through a specific port</td>
<td>666</td>
<td>324</td>
<td>167</td>
<td>180</td>
<td>1</td>
</tr>
<tr>
<td>DropBroadcast</td>
<td>Filter broadcast packets out</td>
<td>196</td>
<td>324</td>
<td>312</td>
<td>217</td>
<td>2</td>
</tr>
<tr>
<td>EtherMirror</td>
<td>Swap ethernet source and destination addresses</td>
<td>388</td>
<td>356</td>
<td>329</td>
<td>197</td>
<td>3</td>
</tr>
<tr>
<td>FromDevice</td>
<td>Interface to NetFPGA input datapath</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>0</td>
</tr>
<tr>
<td>IPMirror</td>
<td>Swap destination and source IP addresses</td>
<td>427</td>
<td>388</td>
<td>298</td>
<td>197</td>
<td>6</td>
</tr>
<tr>
<td>ToDevice</td>
<td>Interface to NetFPGA output datapath</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>54</td>
<td>0</td>
</tr>
</tbody>
</table>

features an ARP table to obtain the next-hop MAC information. The DecIPTTL module recalculates the time to live (TTL) values and filters out expired packets. Register interfaces for writing forwarding table entries and reading bookkeeping information are automatically inserted by the compiler. All components except Lookup are ReClick components. Lookup is a custom RTL module.

5.4.2 Onion router

Onion routing is a widely popular technique to implement secure and anonymous communication over public networks. The sender node chooses a set of onion routers to anonymously route a packet to the destination node. A path is constructed from this node set. The sender then wraps the packet using successive layers of encryption to create an onion packet. The onion is passed to successive onion routers, each of which removes a layer of encryption before forwarding the packet to the next intermediate router. The destination node removes the final layer of encryption to recover the packet data.
We implement an onion router in ReClick by extending the IPv4 router presented in the previous subsection. A decryption component is attached to the front of the data processing pipeline. While real onion routers use public-key cryptography to encrypt packets, we use a symmetric decryption algorithm for simplicity. The onion router shares all components except *DecryptOnion* with the standard IPv4 router. A single configuration, as illustrated in Figure 5.7(b), can be used for both data planes.

5.5 Evaluation

We evaluate ReClick by comparing the packet forwarding performance and resource consumption of an IPv4 data plane which is automatically generated by our framework against a hand-coded IPv4 reference router implementation which is available from the NetFPGA project. Additionally, we compare the ReClick IPv4 data plane with equivalent data planes generated using Chimpp [67] and Switchblade [19] frameworks using similar metrics.

5.5.1 Packet Forwarding Performance

For performance evaluation, we compare the throughput of a single IPv4 virtual data plane generated from ReClick against the NetFPGA reference router. The Virtex II FPGA can accommodate up to four IPv4 virtual data planes. Each virtual data
plane operates at a clock frequency of 62.5 MHz. Figure 5.8 shows the experimental setup for measuring packet throughput. The NetFPGA packet generator [31] is used to accurately generate traffic at line rate (1 Gbps). Packets of sizes varying from 64 bytes to 1024 bytes are used to flood the physical Ethernet interfaces of the target NetFPGA card.

Figure 5.9 compares the throughput of the ReClick modular router against the throughput of the NetFPGA reference router for varying workloads. The ReClick IPv4 router consistently handles line rate traffic for all packet sizes (1 Gbps) demonstrating that modular organization of the virtual data plane does not impose any forwarding performance loss on the network virtualization platform. However, the individual components do introduce additional latency into the packet forwarding pipeline. These latencies are characterized in Table 5.3. The shift buffers between input and output ports prevent the increased latency from affecting packet throughput.

**Figure 5.9.** Packet forwarding throughput of ReClick IPv4 router and NetFPGA reference IPv4 router for varying packet sizes
Table 5.4. Resource Utilization of ReClick IPv4 and onion routers on a Virtex II

<table>
<thead>
<tr>
<th>Resource</th>
<th>NetFPGA IPv4 router</th>
<th>ReClick IPv4 router</th>
<th>ReClick Onion router</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>14640</td>
<td>14562</td>
<td>15599</td>
</tr>
<tr>
<td>Slice FF</td>
<td>15801</td>
<td>16439</td>
<td>17115</td>
</tr>
<tr>
<td>LUTs</td>
<td>23669</td>
<td>23470</td>
<td>24625</td>
</tr>
<tr>
<td>IO</td>
<td>356</td>
<td>356</td>
<td>356</td>
</tr>
<tr>
<td>BRAMS</td>
<td>25</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

5.5.2 Resource Consumption

Table 5.4 presents the logic resources consumed by the ReClick IPv4 router, an extended ReClick IPv4 router that supports packet encryption for onion routing and the NetFPGA reference router implementation. The resource utilization statistics were derived from Xilinx ISE 10.1 synthesis reports generated after the logic map step of the compilation process. All designs were subsequently mapped to silicon through ISE physical design (e.g. place, route, and bitstream generation). The ReClick IPv4 router consumes approximately 49.7% of the available 4 input lookup tables (LUTs) and 62% of the available slices. The logic utilization is thus comparable to that of a hand-coded reference design available from the NetFPGA development platform. However, the presence of shift buffers, which are realized using block RAM memories (BRAMs) and registers within the FPGA, increase the utilization of BRAM resources by 25% and registers by 1%. We believe that a highly fine-grained virtual data plane composition approach is likely to increase the consumption of BRAM and register resources. Alternately, designers can choose to embed more features within each component, allowing for tradeoffs between modularity and logic resources. The onion router consumes an additional 5% slices, 3% LUTs and 2% registers beyond the consumption of the IPv4 design example. Table 5.3 summarizes the detailed logic resource usage and code size for each ReClick component.
5.5.3 Comparison of ReClick with Other Frameworks

To provide a fair evaluation, we compare the throughput and resource consumption of a ReClick-generated IPv4 data plane with throughout and resource consumption of IPv4 data planes described in SwitchBlade [19] and Chimpp [67]. All evaluated data planes were implemented in a Virtex II FPGA available on the NetFPGA 1G platform. The resource utilization of SwitchBlade and Chimpp IPv4 data planes were obtained from previously published research data [19] [67]. The IPv4 router described in Chimpp uses 4% more slices than the reference hand-coded design. In contrast, the logic utilization of the ReClick router is comparable to the hand-coded implementation. The base SwitchBlade platform features data plane components supporting preprocessor blocks for OpenFlow, IPv6, variable bit extraction and path splicing supporting up to four IPv4 data planes. This configuration uses approximately 79% of available 4-input LUTs, 89% of available slices and 42% of slice flip flops. The base ReClick IPv4 router features only preprocessing blocks for IPv4 routing and hence consumes 27% fewer slices and 7% fewer registers when compared to the SwitchBlade platform. Since ReClick supports component sharing between data planes, we expect the resource usage to grow nonlinearly with the number of data planes hosted in the virtualization platform. All the data planes support line rate forwarding (1 Gbps).

5.6 Conclusion

This chapter introduced ReClick, a modular data plane design framework for FPGA-based network virtualization platforms. ReClick proposes abstraction and reuse and key design philosophies. Using this framework, we have demonstrated efficient implementation of several packet processing components and larger data plane configurations.
Iterative algorithms represent a pervasive class of data mining, web search and scientific computing applications. In iterative algorithms, a final result is derived by performing repetitive computations on an input data set. Existing techniques to parallelize iterative algorithms use software cluster computing frameworks such as MapReduce [32] and Hadoop [6] to distribute data for an iteration across available resources and collect per-iteration results. These platforms are marked by the need to synchronize data computations at iteration boundaries, impeding system performance.

In this chapter, we demonstrate that FPGAs in distributed heterogeneous computing systems can serve a vital role in breaking this synchronization barrier. Our Maestro system uses asynchronous accumulative updates to execute a general-class of iterative algorithms on a heterogeneous cluster of commodity CPUs and FPGAs. These updates allow for the accumulation of intermediate results for numerous data points without the need for iteration-based barriers. Both CPU and Altera DE4 FPGA-based compute elements prioritize computations to accelerate algorithm convergence in our scalable system. Computation is dynamically prioritized to accelerate algorithm convergence.

A general-class of iterative algorithms have been implemented on a cluster of four FPGAs. A speedup of $7 \times$ is achieved over an implementation of asynchronous accumulative updates on a general-purpose CPU. The system offers up to $154 \times$ speedup.
versus a standard Hadoop-based CPU workstation cluster. Improved performance is achieved by clusters of FPGAs.

6.1 Iterative Algorithms

In general, iterative algorithms arrive at the final outcome by repetitively performing the same set of operations over the input data. The intermediate values of an iteration depend on the intermediate values of the previous iteration. The iterative computing model can be represented as

\[ v^k = F(v^{k-1}) \]  

(6.1)

where \( v^k \) is an n-dimensional data vector \( \{v^k_1, v^k_2, ..., v^k_n\} \) denoting all the values of the \( k^{th} \) iteration and \( F \) represents the update function. The intermediate results of the current iteration are reused as inputs in the subsequent iteration, until a termination criterion is met. Since each element of the data vector \( v^k \) can be computed separately, iterative algorithms are highly data-parallel in nature. The data-parallelism can be exploited to accelerate the convergence of iterative computations using general-purpose cluster computing frameworks such as MapReduce and Hadoop.

Many search and data mining applications in the cloud use iterative algorithms to refine and process web data. For example, PageRank is an iterative algorithm which is used to refine rank values of webpages in the World Wide Web. Link prediction [54] and recommendation systems [20] use iterative algorithms such as Adsorption and Expected Hitting Time. K-means [71] clustering is an iterative algorithm used to classify data in computational biology.

Example: PageRank is an iterative algorithm that is used to calculate the relative importance of the vertices (webpages) in a graph. The general PageRank algorithm iterates over a web address linkage graph \( G(V, E) \), where \( V \) represents the webpages (vertices/nodes of the graph), and \( E \), the set of hyperlinks between webpages (edges
of the graph). An edge exists between nodes $i$ and $j$ if a hyperlink exists from node $i$ to node $j$. Assume that there are $N$ webpages in the web graph. To calculate the relative importance of webpages, each node $u$ in the graph is initially assigned an initial PageRank score $R(u) = \frac{1-d}{N}$, where $d$ is a constant dampening factor. The page rank of a node in the iteration $i+1$ is successively refined from its previous value in the $i^{th}$ iteration as:

$$R^{(i+1)}(u) = \frac{1 - d}{N} + \sum_{u \in B_u} d \times \frac{R^{(i)}(u)}{L(u)} \quad (6.2)$$

In this equation, $B_u$ represents the set of pages that have hyperlinks to $u$ and $L(u)$ represents the number of hyperlinks from webpage $u$.

Consider the iterative computation of PageRank in MapReduce as shown in Figure 6.1. The nodes in the web graph are stored in a distributed file system which is partitioned over multiple workstations. The rank of each webpage is initialized to a value $\frac{1-d}{N}$. During the start of an iteration, the graph nodes are copied into the RAM. The computation executes as a sequence of “Map” and “Reduce” task. In the map phase, the rank of a webpage $A$ is dampened by $d \times \frac{PR(A)}{L(A)}$ where $L(A)$ is the number of pages that have directed edges (hyperlinks) from $A$. The results from the map phase are shuffled to $A$’s neighbors (i.e. webpages that have hyperlinks from $A$). In the
reduce phase, each page collects all partial rank scores sent from its neighbors and updates its current rank value by applying a reduction operation (+ for PageRank). The results of the reduce phase are finally dumped into the distributed file system. These results are subsequently reused as inputs by the next iteration. The execution of the algorithm terminates when the rank scores of webpages remain largely unchanged between subsequent iterations.

Iterative algorithms suffer from several inefficiencies when executed in general-purpose cluster frameworks. We summarize the major limitations below:

1. **Synchronization barriers**: In MapReduce, assuming that there are \( n \) values in the input data set and \( n \) computing nodes, the value of the \( j^{th} \) node in the \( k^{th} \) iteration is updated as

\[
v_j^k = F(v_1^{k-1}, v_2^{k-1}, ..., v_n^{k-1})
\]

The function to compute \( v_j^k \), \( F \), is applied only when all \( n \) values from the previous iteration \( (v_1^{k-1}, v_2^{k-1}, ..., v_n^{k-1}) \) are received from all other nodes (e.g., in PageRank, the summation is applied only after collecting all partial scores from neighboring pages as shown in Figure 6.1). Although intermediate values from a previous iteration may arrive at different time intervals, each worker must wait for other workers to finish the previous iteration. This requirement imposes strict synchronization barriers.

2. **Intermediate result storage**: As shown in Figure 6.1, MapReduce relies on distributed file systems to store intermediate results of iterations. Repeated reads and writes to the file system between successive iterations wastes CPU cycles and I/O bandwidth.

3. **Stragglers and Fast Nodes**: MapReduce also assumes that the computing nodes are fairly homogeneous in nature - i.e., all machines in the cluster make
roughly equal progress at any given time during the computation. Support for slower nodes (also called as stragglers) is provided through speculative task execution as follows: Workers request new jobs from the master node to fill their empty task slots. The master schedules a new job for an empty slot in the following order: If there are failed jobs in other workers, they are scheduled first. Otherwise, the master schedules non-running tasks for the new task slots. If none are available, the master speculatively executes a job from the slowest worker in the new task slot. The objective of the MapReduce scheduling algorithm, is to minimize the job response time [88]. The slowest tasks in the MapReduce model have the least priority for speculative execution. In the presence of computing nodes which vary greatly in computing capacity, the speculative MapReduce execution model has the additional overhead of migrating the tasks between the worker nodes. Any benefit obtained from introducing faster worker nodes is reduced due to this overly conservative scheduling model.

4. Support for Heterogeneous Nodes: MapReduce is designed to execute only on general-purpose processors, although, it is well known that processors are not well suited to execute data-parallel workloads. The framework offers no support for specialized data-parallel architectures such as FPGAs or GPGPUs.

6.2 Improvements to MapReduce Model

Several improvements of the original MapReduce framework have been proposed to accelerate iterative algorithms [89] [90] [91]. iMapReduce [89] transforms the map and reduce tasks into persistent tasks that stores intermediate results from iterations in memory, eliminating the need for unnecessary reads/writes to the distributed file system. Each worker schedules the reduce phase as soon as the intermediate map results for that worker are available, obviating the need for strict synchronization barriers. Further, workers do not shuffle data such as web linkage information that is
invariant across iterations. Priter [90] identifies a subset of the input dataset that can lead to faster convergence towards the final outcome and performs iterations only on that subset. Maiter [91] proposes a completely asynchronous approach by allowing workers to independently update their partitions of the input dataset and propagate these values through asynchronous updates.

6.3 MapReduce on Special-purpose Hardware

A number of deployments of MapReduce and other implementations of iterative algorithms on distributed hardware have been demonstrated. MapReduce was introduced as a compute model for FPGAs and GPUs in 2008 [43]. A set of libraries for a heterogeneous system containing a single component of each device was demonstrated. FPMR [70] demonstrated an implementation of iterative algorithms using an FPGA and external DDR memory. During the start of an iteration, data is buffered into local FPGA memories and it is retained in memory during the computation. FPMR addresses synchronization issues for a single-FPGA system by allowing computation to start as soon as intermediate values are available for a specific element up until an iteration boundary. If the computation requires multiple iterations, data values must be written back to the global memory. Axel [75] is a heterogeneous cluster consisting of FPGAs, GPUs and CPUs which are interconnected using Ethernet links. This paper specifically mentions the challenge of balancing computation across heterogeneous resources to avoid waiting on barriers (Section 6.6, paragraph 2). Mars [40] implements iterative algorithms on GPGPUs. The individual map and reduce tasks, specified using APIs, are assigned to GPU threads. Although these frameworks mark important steps towards integrating special-purpose hardware with existing PC clusters, they inherit the synchronization challenge of the distributed iterative compute model.
6.4 Asynchronous Accumulative Updates

In this section, we introduce the idea of asynchronous accumulative updates [91] to eliminate the need for synchronization barriers MapReduce model. In the synchronous compute model, the update function $F$ is applied only after a node collects the intermediate results of the previous iteration from all compute machines. Assuming that $n$ values are distributed equally among the compute machines, the synchronous approach requires each machine to possess $O(n)$ storage.

The asynchronous accumulative compute model (AAU) eliminates the need for strictly synchronous iterations. The key idea in the asynchronous accumulative compute model [91] is that each node propagates only the “change” in its value rather than the value of the node. Changes are accumulated and updates are propagated to other nodes in a completely asynchronous fashion.

Let $\Delta v$ represent the change in value of a node between two updates\(^1\). In the AAU model, the change in the rank value of a node (e.g. $\Delta v$) is propagated as a message to its neighbors. As a node receives changes from its neighbors, it accumulates these changes into a single memory location. To calculate its new value, a node applies the changes received from all its neighbors to its current value and resets the memory location where accumulated changes from other nodes are stored. Finally, the node propagates its own changes in the form of messages to all its neighbors.

As an example, consider the iterative computation of PageRank using asynchronous accumulative updates as shown in Figure 6.4. Let $\Delta PR(A)$ and $\Delta PR(B)$ represent the changes in the rank values of nodes A and B between subsequent updates. Both A and B independently propagate their changes to node C. Node C accumulates the changes propagated from other nodes to itself into a variable $\Delta PR(C)$ (the change in the rank value of C). The new rank score of C page is derived is cal-

\(^1\)An update occurs when a new value is calculated for a node
Figure 6.2. Illustration of accumulative updates - In (a), the change in rank of webpage A is accumulated into the change in rank of webpage C (\(\Delta PR(C)\)), following which node C is updated. In (b), the change in rank of webpage B is accumulated into change in rank of C (\(\Delta PR(C)\)) and node C is updated.

culated by applying the accumulated change (\(\Delta PR(C)\)) to its current value (PR(C)) in the update step. The change in C’s rank score is dampened by the factor \(dL(C)\) and then propagated to its neighbors. In the final update step, the change in the rank score is reset (\(\Delta PR(C)=0\)). The update and accumulate operations are asynchronously performed by each node.

For generality, consider that the value of a node \(u\) at a given point in time is \(v\). If a new input value arrives at this node, it does not need to be added to \(v\) immediately. Rather, it can be *accumulated* into a partial sum \(\Delta v\) which can later be added to \(v\). In this asynchronous accumulate model, each compute node performs two operations:

**Accumulate:** When a compute node receives a message \(m\) from any other worker, it is accumulated into a storage location \(\Delta v\). The accumulation is specified using an abstract operator \(\Theta\). Incoming values are accumulated in any order. There is one \(\Delta v\) for each data value \(v\).

\[
\Delta v \leftarrow \Delta v \Theta m \tag{6.4}
\]

In the PageRank example, \(\Theta\) is an addition operation.
Updates

Sync barrier

Synchronous computation

Asynchronous computation

Figure 6.3. Visualizing asynchronous accumulative updates

**Update:** \( \Delta v \) is added to \( v \), updating its value and messages are generated for other values which depend on \( v \) as an input. The messages are sent to the compute nodes which contain those values. This update operation is performed according to a scheduling policy in three steps: (1) The node adds the accumulated value \( \Delta v \) into its current value \( v \), (2) an update function \( g() \) is applied to the change in its current value, \( \Delta v \), and (3) the node propagates \( m=g(\Delta v) \) to all neighboring nodes and resets \( \Delta v \).

\[
v \leftarrow v \bigoplus \Delta v, \quad (6.5a)
\]
\[
\text{if}(\Delta v \neq 0) \text{ send } m = g(\Delta v) \quad (6.5b)
\]
\[
\Delta v \leftarrow 0 \quad (6.5c)
\]

For accumulative updates to guarantee correctness, the \( \bigoplus \) operator must possess commutative, associative and identity property over \( \bigoplus \) and \( g() \) must possess distributive properties.

Figure 6.3 provides a visual comparison of the propagation of updates in the synchronous (e.g. MapReduce) and the AAU model. In the synchronous model,
although updates arrive at different time intervals at a node, the next iteration in the node can only commence at specific synchronization intervals. In contrast, the AAU model allows updates to be propagated seamlessly in a streaming fashion.

**Scheduling Updates** - A worker node that owns a partition of the input data set performs updates according to a user-defined scheduling policy. In a round robin scheduling policy, a worker iterates through its data partition updating each value in order one by one. Although simple, the round robin strategy is quite inefficient. For example, if all data receive equal priority, updates may be performed on many values that are insignificant to the overall progress of the computation. In many applications, it is possible to reduce the time to convergence (e.g. fewer iterations/operations) by prioritizing updates for the subset of data with higher importance.

### 6.5 Maestro Cluster Design

The major contribution of this work is the scalable implementation of asynchronous accumulative updates (AAU) in a compute cluster consisting of FPGAs which contain the parallelism and specialization necessary to accelerate the customized computation versus a CPU-based cluster. The distinguishing features of this system that separate it from previous implementations of iterative algorithms (e.g. MapReduce and other implementations) include:

**Asynchronous updates:** Each computing node propagates results from its updates to other nodes as soon as they are generated without waiting for updates from other nodes. Updates received from other nodes are accumulated at the recipient node. Some updates may be used locally on the node which produces them.

**Scalable FPGA implementation:** An FPGA-based hardware architecture which implements the accumulative-update computing model has been developed and tested. The architecture allows users to scale the performance of individual FPGA
Figure 6.4. Cluster setup for a four node Maestro system including CPUs and FPGAs. Compute nodes include DRAM and disk storage (hdd).

nodes as well as the capacity of the cluster by attaching additional FPGA boards to the cluster network.

Prioritized updates in the hardware implementation: The intermediate results in our system are stored in DRAM during the computation, eliminating the need for frequent disk accesses. The use of accumulations limits the need to store numerous intermediate values. Effectively, intermediate results are combined using $\bigoplus$ operations (e.g. addition in the PageRank example). Updates are prioritized based on the size of $\Delta v$, where $v$ values with large $\Delta v$ are updated first. Prioritization is performed using a lightweight circuit within the programmable logic.

Our Maestro asynchronous accumulative update model is implemented on a compute cluster consisting of FPGA worker nodes as shown in Fig. 6.4. The cluster consists of a single master (CPU 0) workstation and several slave FPGAs interconnected in a LAN configuration. The master is a CPU node responsible for coordinating the tasks running in other slaves and checking for termination conditions. Slaves are built from Altera DE-4 development boards (Figure 6.5) with a Stratix IV EPS230GX device. Slaves run in parallel to execute the computation as tasks and communicate
via Gigabit Ethernet links attached to a NetFPGA router in star topology. The distributed file system (DFS) forms a logical storage that stores the input data used for iterative processing. DFS is implemented as a logical collection of hard drives located at separate workstations.

In order to simplify the process of accessing the distributed file system interface from the FPGA slave, in this prototype implementation each FPGA is attached to a CPU workstation (FPGA Assistant in Fig. 6.4) which manages all distributed file accesses on behalf of the FPGA. Specifically, the FPGA assistant is responsible for tasks such as loading the data from the distributed file system into the FPGA, checking for termination conditions and writing the computed results back into the DFS. FPGA assistants exchange information such as termination check information with the master using standardized message passing interfaces (MPI) based on Open-MPI [12]. In future implementations, these functions could be performed by a soft or hard processor implemented on the FPGA. Each FPGA slave node implements a hardware architecture for performing accumulative updates and a network interface for communicating with other worker nodes.
Each data element in the input set (e.g., each webpage in the PageRank example) is identified by a unique global key. A hash function of the key is used to make the node assignment. In the current implementation, a simple modulo (MOD) hash function is used, although more efficient functions could be considered in the future. Input data are organized as key-value pairs (KV pairs) and transferred to the appropriate node by the master at the beginning of the computation. A worker stores its partition of input data in state tables. The FPGA worker node stores state tables in a 1 Gbit DDR2 DRAM attached to the DE4 board. Messages communicated between nodes during the computation also use a key-value pair structure.

### 6.6 FPGA Architecture

The compute architecture in the FPGA slave provides dramatic performance advantages over microprocessor implementation due to customization of both the computation and the communication interface, optimizations that are not possible in a microprocessor or a GPU. The FPGA slave performs update and accumulate opera-
tions on a subset of key value pairs assigned by the master node. The architecture is shown in Fig. 6.6. Two hardware modules, Packet parser and Packet composer, handle communication with other slaves and the FPGA assistant. The packet parser, built by customizing the receive datapath of a NetFPGA reference router [62], parses incoming Ethernet packets and initiates appropriate actions (e.g. load KV pairs into the FPGA, start the computation, etc.). The packet composer, built by modifying the transmit datapath of the NetFPGA reference router, constructs Ethernet packets from outgoing messages. Update/accumulate operations on KV pairs are performed in parallel by several processors. Processors access KV pairs from the state table using a shared 32-bit Avalon interconnect. Each processor owns an equal share of KV pairs assigned to the FPGA slave and is responsible for all operations on these KV pairs. Users can vary the number of processors to suit the needs of the specific application. During every iteration, the processor selectively refines KV pairs to prioritize the ones that are more relevant to the overall computation. This is achieved by comparing the priority of each KV pair with a threshold set by the threshold selection module. To prevent any memory inconsistencies caused by one or more processors performing update/accumulate operations on the same KV pair, processors negotiate exclusive access to a KV pair using the coherency controller.

Next, we discuss each component in greater detail.

6.6.1 State Table

KV pairs are stored using a state table within the DRAM. Since many scientific and web data mining applications involve processing on sparse graphs, the state table is designed to store KV pairs in a memory efficient fashion. A state table entry is indexed by a hash of the key, and consists of five fields as shown in Fig. 6.6: the key, its current value ($v$), the accumulated change in the value between two consecutive update operations ($\Delta v$), priority field and the linkage information. The
Algorithm 5: Prioritized KV Pair Selection in FPGA

Input: StateTable table, StateTable size \( N \), circuit cells \( K \), sample size \( S \)

Output: set of prioritized KV pairs for update operation

1. \( \text{samples} \leftarrow \text{randomly select } S \text{ records from } N \text{ entries in table} \)
2. \( \text{K Cells} \leftarrow \text{samples} \)
3. \( \text{thresh} \leftarrow \text{Cells}[K].\text{priority} \)
4. \( \text{foreach record } r \text{ in table do} \)
5. \( \quad \text{if } r.\text{priority} \geq \text{thresh then} \)
6. \( \quad \quad \text{Select } \langle r.\text{nodeid} \rangle \text{ for update} \)
7. \( \quad \text{end} \)
8. \( \text{end} \)

linkage information is a pointer to a linked list of keys whose results depend on the current key (e.g. in PageRank, other webpages which are referenced by the current page). The state of the key including \( v \) and \( \Delta v \) fields is constantly modified by the update and accumulate operations.

6.6.2 Threshold Selection

Prioritizing the updates to KV pairs during an iteration is critical to accelerating algorithm convergence. A naive approach to select the \( K \) most relevant KV pairs is to simply sort all KV pairs by their priority values and then choose the top \( K \) KV pairs for update operations. While this approach is quite simple, it is quite inefficient since all keys must be sorted during each iteration.

Instead, Maestro uses a threshold-based heuristic as shown in Algorithm 5. The intuition of this heuristic is that the distribution of priority values in a statistical sample of the KV pairs provides a good approximation of the priority values in the state table [90]. To refine the top \( K \) pairs, a small subset of KV pairs (\( S \)) is randomly sampled. The sample is ordered by the value of priority fields using a threshold selection circuit consisting of a chain of \( K \) shift registers (cells). The threshold is set as the priority value of the \( K^{th} \) highest KV pair in the sorted sample. The threshold is then used by the processor during every iteration to measure a KV pair’s relative
importance to the computation. A KV pair is only chosen for update operations if its priority field has a value larger than the threshold.

In the customized FPGA implementation, a modified maximal-sequence linear feedback shift register (LFSR) circuit of length \( n \) bits (\( n = \lceil \log_2(N) \rceil \), \( N \) = number of keys in state table) is used to randomly select \( S \) samples from DRAM. As KV pairs are fetched, they are prioritized by a threshold selection circuit, as shown in Figure 6.7, by the value in the priority field. The circuit works on the principle of parallel insertion sort. A shift register chain of \( K \) cells hold the KV pairs. Each cell stores a KV pair fetched from the DRAM.

When a KV pair is read from the DRAM, a floating point comparator in the cell compares the priority field of the incoming key entry with the priority field of the key entry in the register. The \( low_{\text{out}} \) signal indicates whether the stored key’s priority is lower than the priority of the incoming key. Additionally, each cell observes the comparison outcome of its left neighbor through the \( low_{\text{in}} \) port. Based on the two comparisons, the cell makes a decision as follows: (1) If the left neighbor’s priority field and the cell’s own priority are lower than that of the incoming key entry (\( low_{\text{in}} = 1 \) and \( low_{\text{out}} = 1 \)), the cell shifts in the key entry from its left neighbor, (2) If the left neighbor’s priority is higher than the incoming key entry’s priority and the cell’s
priority is lower than that of the incoming key entry’s priority ($low_{in} = 0$ and $low_{out} = 1$) the cell replaces its current key entry with the incoming key entry, and (3) Otherwise, the cell simply retains its current key entry. After $S$ state table key entries are streamed in through the $data_{in}$ port, the top $K$ state table entries are available in the cells ordered by their priority values with the key entry with the highest priority appearing in the cell farthest to the left.

The circuit facilitates the extraction of the top $K$ entries from $S$ samples in $O(S)$ time complexity and $O(K)$ space complexity. The threshold value is set as the priority field of the KV pair in the cell that appears farthest to the right.

### 6.6.3 Processor

The processor performs update and accumulate operations on a subset of KV pairs assigned to the slave. Each processor can be configured in two modes - transmitter (TX) or receiver (RX). A processor in TX mode performs both update and accumulate operations while a processor in RX mode only performs accumulate operations.
The operation mode can be dynamically configured by the user through software configurable registers. Update/accumulate operations on KV pairs are sequenced using a five-stage pipelined datapath as shown in Figure 6.8 in order to maintain high throughput. The coherence controller ensures memory consistency for each key accessed by the processor during update/accumulate operations. The \texttt{Tcheck} module computes the progress of computation as measured by the sum of \( v \) fields of KV pairs owned by the particular processor. Each processor uses three memory interfaces to access the state table in DRAM. During an iteration, a processor configured in TX mode performs update operations on all KV pairs it owns in six steps:

1. The \textbf{Choose Key} module generates a KV pair address from the subset of KV pairs owned by the processor.

2. The \textbf{Lock Key} module ensures that the KV pair is not being operated upon by any other processor at the same time by atomically locking the KV pair.

3. The KV pair entry is read by the \textbf{Record Fetch & Filter} module from the DRAM state table. Next, the priority field of the KV pair is compared with the threshold set by the threshold selection circuit. If the priority value is higher than the threshold, the KV pair is marked for update operations.

4. In the \textbf{Update/Accumulate} stage, the marked KV pair is updated according to Eqs. (6.5a), (6.5b) and (6.5c). The message \( m = g(\Delta v) \) and a pointer to the links associated with the KV pair are forwarded to the \textbf{Link Access} stage.

5. In the \textbf{Write Key} stage, the updated \( v, \Delta v \) fields of the KV pair are written back into the DDR state table. The lock on the KV pair is released.

6. The \textbf{Link Access} stage forwards the message (msg) to all links associated with the KV pair. If the link is a KV pair located within the slave FPGA (local accumulation), the message is placed in LINK FIFO. Otherwise, it is placed...
in EXT FIFO (external accumulation). Messages placed in EXT FIFO are subsequently collected by the **Packet composer** module and dispatched to other FPGA slaves.

Accumulation messages generated locally or from other workers follow the pipelined datapath except that an Update/Accumulate operation only performs an accumulate on the KV pair. A processor configured in RX mode accepts messages for accumulation from other FPGA slaves via the RX FIFO. A transmitter processor (TX) prioritizes messages for local accumulation over updating new KV pairs.

### 6.6.4 Termination Check

Each slave FPGA measures and reports the progress of the local computation to the master node. Progress is defined as the sum of $v$ fields for all keys in the state table. Since update and accumulate operations are cumulative over the $v$ field of the KV pair entry, the rate of progress monotonically increases or decreases over time. Within the slave FPGA, **TCheck** modules attached to each processor compute local progress during every iteration. The results are aggregated and made available to the packet composer, which when requested, sends the estimated progress to the master node.

### 6.7 Ensuring Memory Consistency during Updates

When multiple processors operate on KV pairs resident in a shared global memory, memory inconsistencies can occur due to one or more processors writing to the same KV pair entry. For example, consider two processors, each performing an accumulate and update operation on the same KV pair. While the update operation resets the $\Delta v$ field in the state table entry, the accumulate operation accumulates the incoming message $m$ into the $\Delta v$ field according to $(\Delta v \leftarrow \Delta v \bigoplus m)$. Similarly, an inconsistency can also happen from two processors trying to perform identical oper-
ations (update/accumulate) on the same KV pair. To avoid memory inconsistency, all operations on KV pairs must be strictly atomic.

To address this issue, Maestro implements a **snoopy coherency protocol** that borrows principles from cache coherency protocols in symmetric multiprocessor systems. The protocol is implemented within the coherence controller block attached to each processor. The snoopy coherency protocol guarantees that simultaneous accesses to the same KV pair are serialized, enforcing strict memory consistency on each KV pair. If accesses do not conflict, update/accumulate operations proceed in parallel in all processors. To access a KV pair, each processor performs the following steps:

1. A request with the key is submitted by the processor to the coherence controller module.

2. The coherence controller requests access to a shared bus (snoopy bus in Fig. 6.6) from a bus arbiter. If multiple processors simultaneously submit requests, they are resolved in a round-robin fashion by the arbiter.

3. Once the bus is won, the coherence controller places the requested key on the shared bus and raises a check request.

4. All processors that share the bus respond to the check request with a response confirming the possession of the key. If no processors hold the key, the requested processor locks the key. Subsequently, the snoopy bus is released.

5. After an update/accumulate operation is performed on the key, the lock on the key is released.

### 6.8 System Scalability

The computing capacity can be scaled by adjusting the number of TX/RX processors within each FPGA or by attaching several FPGAs in a multi-node cluster
configuration. In multi-node configurations, at least one processor must be configured as a receiver processor to process update messages from other slaves. The number of transmitter and receiver processors can be dynamically varied by the user to suit the requirements of the application through software configurable registers. Section 6.11 describes the effect of varying the transmitter to receiver processor ratio for different applications in multi-node cluster configurations.

6.9 Cluster Configuration and Operation

Figure 6.9 illustrates a laboratory prototype of the Maestro cluster built on four Altera DE-4 boards. To parallelize an iterative algorithm using Maestro, a user must specify three components: a data partitioner, an iteration kernel, and a termination checker. These interfaces are sufficiently general to describe any algorithm which meets the asynchronous accumulative update criteria described in Section 6.4. The partitioner specifies the criterion to assign the keys to workers (e.g. the MOD operation in the PageRank example). The partitioner reads input key-value pairs
from a file and assigns them to the individual worker nodes. The partitioner is implemented as a C++ API. The iteration kernel specifies the accumulate and update operations and the initial values for the keys. These operations are described as Verilog templates. The termination checker component is used to describe the criterion which must be satisfied to terminate the iterative computation.

The user specifies the cluster configuration in a file as a list of hostnames/IP addresses of all the CPU nodes and FPGA assistants. In this file, the master (e.g. PC 0 in Fig. 6.4) is listed first followed by other nodes in the cluster. In addition, each machine locally stores a type file that identifies the type of the PC worker (CPU/FPGA assistant). If the machine is an FPGA assistant, the file also describes the network interface configuration for the FPGA-PC interface. The FPGA is programmed using a USB JTAG interface. After bitstream download, FPGA-PC Ethernet interfaces are brought up using TCL-based configuration scripts.

The CPU node designated as the master (e.g. CPU 0 in Fig. 6.4) runs the partition function to distribute the input data according to the hash function specified in the partitioner. The computation executes in every worker in three steps. The master instructs all workers to load the data partitions from the local file system into the DRAM-based state tables. The FPGA assistant converts partition data into packets and sends them over to the FPGA. Slaves start the iterative computation process and exchange messages via Gigabit Ethernet links attached to 1G NetFPGA reference router. To amortize the communication cost of sending a KV pair outside a slave, messages are aggregated until there are enough to fill the maximum transmission capacity of an Ethernet frame (150 key-value pairs). The total progress in slaves is checked periodically (e.g. every 4 seconds) by the master. Once terminated, the results of the computation are retrieved by FPGA assistants from the slave nodes.
Table 6.1. List of iterative algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Init&lt;sub&gt;j&lt;/sub&gt;</th>
<th>g&lt;sub&gt;j&lt;/sub&gt;(x)</th>
<th>⊕</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected</td>
<td>j</td>
<td>x · Δ&lt;sub&gt;j&lt;/sub&gt; · a&lt;sub&gt;ji&lt;/sub&gt;</td>
<td>max</td>
</tr>
<tr>
<td>PageRank</td>
<td>1 − d</td>
<td>d&lt;sub&gt;i&lt;/sub&gt; · Δ&lt;sub&gt;j&lt;/sub&gt; · a&lt;sub&gt;ji&lt;/sub&gt;</td>
<td>+</td>
</tr>
<tr>
<td>Katz metric [46]</td>
<td>1 (j = s) or 0 (j ≠ s)</td>
<td>β · x · Δ&lt;sub&gt;j&lt;/sub&gt; · a&lt;sub&gt;ji&lt;/sub&gt;</td>
<td>+</td>
</tr>
</tbody>
</table>

6.10 Experimental Approach

Setup: To evaluate Maestro, we implemented a compute cluster with four CPUs and four FPGA nodes. Each CPU node has an Intel Core2 Quad processors running at 2.44 GHz with 4 GB RAM. Machines have attached 1 Gbit/s network interface cards (NICs) to interface with the LAN setup. For Maestro cluster experiments, we fix the sampling size (S) as 1024, threshold selection circuit size (K) as 128. A termination check is performed by the master node every 4 seconds. The FPGA operates at a frequency of 125 MHz.

Algorithms: We consider three iterative algorithms shown in Table 6.1 for our experiments. For each algorithm, Table 6.1 specifies the initial value for the j<sup>th</sup> key (Init<sub>j</sub>), update function for the j<sup>th</sup> key (g<sub>j</sub>(x)) and accumulate (⊕) operators. The objective of the connected components (Connected) algorithm is to find all the connected nodes in a graph. In the iterative formulation of this algorithm, the j<sup>th</sup> key is initialized to a unique ID (Init<sub>j</sub> = j). Next, the j<sup>th</sup> key propagates its ID to all its i neighbors in the adjacency matrix a<sub>ji</sub> if the change in its value (Δ<sub>j</sub>) is non-zero (g<sub>j</sub>(x) = x · Δ<sub>j</sub> · a<sub>ji</sub>). When a key receives an ID, it compares its ID with the incoming ID and chooses the maximum of the two (⊕ = max). The algorithm converges when the IDs of all nodes do not change between subsequent iterations. Katz metric [46] finds the proximity measure of two nodes in a graph. It is computed as the sum over all the paths between two nodes exponentially dampened by the path length. In the iterative formulation of Katz, a key chosen as the source node (s). The source node is assigned an initial value of 1. All other nodes are initialized to 0. During
Table 6.2. Speedup of Maiter versus Hadoop for 1, 2, and 4 workers

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Cluster</th>
<th>Graph 1</th>
<th>Graph 2</th>
<th>Graph 3</th>
<th>Execution time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Hadoop</td>
<td>Maiter</td>
<td></td>
<td>Hadoop</td>
<td>Maiter</td>
</tr>
<tr>
<td>PageRank</td>
<td>1</td>
<td>1.3M</td>
<td>2505</td>
<td>114</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.6M</td>
<td>3639</td>
<td>467</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.2M</td>
<td>6673</td>
<td>717</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Katz</td>
<td>1</td>
<td>1.3M</td>
<td>4200</td>
<td>137</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.6M</td>
<td>4707</td>
<td>412</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.2M</td>
<td>10741</td>
<td>563</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Connected</td>
<td>1</td>
<td>1.3M</td>
<td>500</td>
<td>29.2</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.6M</td>
<td>1115</td>
<td>66</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.2M</td>
<td>1695</td>
<td>121</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

an iteration, every key node multiplies its current value by a constant dampening factor $\beta$ and propagates the result to other nodes. When a key receives a message, it accumulates the message ($\oplus = +$).

To evaluate Maestro, we also implement the three algorithms in Table 6.1 using Hadoop, an open source implementation of MapReduce [32], and Maiter frameworks, in addition to our heterogeneous system. The Hadoop implementation requires the use of strictly synchronous barriers and disk writes between successive iterations while Maiter provides an implementation of the asynchronous accumulative update-based computing model only using general-purpose CPUs. Evaluation is performed using graphs where in-degrees follow a log-normal distribution with parameters $(\sigma = 0.5, \mu = 2.3)$. Graphs are sized to nearly fill the capacity of 1 Gbit DRAM memory on the Altera DE4 board.

6.11 Evaluation

6.11.1 Execution Time

In an initial experiment, we compare the execution time of the asynchronous accumulative update based computing model implemented on a single FPGA versus a Maiter implementation on a general-purpose CPU. To illustrate the state-of-the-
Figure 6.10. Speedup of Maestro (1 FPGA) versus Maiter (1 microprocessor). Graph size=1.3 million nodes

Figure 6.11. Speedup of Maestro (1 FPGA) versus Hadoop (1 microprocessor). Graph size=1.3 million nodes
art nature of Maier, the speedup of Maier on a microprocessor versus a standard Hadoop implementation on a microprocessor is shown in Table 6.2. Maier executes 22×, 31× and 17× faster than the Hadoop version for PageRank, Katz and Connected benchmarks. The speedup results from the removal of synchronous barriers and disk writes between iterations. Our FPGA implementation makes further dramatic improvements on this Maier speedup by using FPGA parallelism and specialization.

Fig. 6.10 shows the speedup of executing the three benchmarks on one Maestro FPGA node normalized against the execution time on Maier on a single microprocessor. For the same setting, Figure 6.11 shows Maestro speedup normalized against the execution time on Hadoop. The input dataset is a 1.3 million node graph (900MB). In the experiment, the number of transmitter processors in the Maestro FPGA (P_{tx}) is varied from 1 to 8. With one transmitter processor (P_{tx}=1), Maestro is 77% faster than Maier (39× faster than Hadoop) for the PageRank benchmark. Speedup linearly scales as more processors are added to the FPGA. With eight processors in the FPGA, Maestro executes approximately 7× faster than Maier (154× faster than Hadoop) in PageRank. The Katz benchmark executes approximately 6× faster than Maier on eight processors (186× faster than Hadoop). Connected components is a relatively low compute intensive application (g_j(x) = x \cdot \Delta_j \cdot a_{ji}) which yields only a modest speedup of 2.2× versus Maier (38× vs Hadoop) with eight processors in the FPGA. In general, the performance gap between CPU and the FPGA implementation grows with the complexity of accumulate and update operations.

6.11.2 Processor Configuration

In this section, Maestro is evaluated in a multi-worker cluster environment. To understand the effect of processor configuration on the overall speedup of the application, we perform experiments in two and four worker configurations.
Figure 6.12. Speedup of Maestro versus Maiter on two workers for different transmitter to receiver processor configurations. Graph size=2.6 million nodes

Figure 6.13. Speedup of Maestro versus Hadoop on two workers for different transmitter to receiver processor configurations. Graph size=2.6 million nodes
Two workers: A two-FPGA cluster is setup according to the topology in Fig. 6.4. Each FPGA in the cluster includes eight processors. The ratio of transmitter to receiver (P_{tx}:P_{rx}) processors in the design is dynamically varied during the experiment. For each application, the problem size is doubled from that of the one worker experiment (2.6 million nodes/1.8 GB). The workload is evenly divided between all slaves using the MOD partition function. For comparison, Maister and Hadoop are executed on two CPU workstations interconnected in a LAN configuration.

Figure 6.12 compares the speedup of a two-worker Maestro cluster against a two-worker Maister cluster. Figure 6.13 compares the speedup of the Maestro cluster against the two-worker Hadoop implementation. Maister executes 8× faster than Hadoop for PageRank. With one transmitter and seven receiver processors (P_{tx}:P_{rx} = 1:7), Maestro executes 10× faster than Maister for PageRank. Two factors contribute to the speedup; first, the cost to send a KV pair outside the slave FPGA worker is relatively insignificant when compared to Maister since packet handling is performed exclusively in the programmable logic. The communication cost in Maister can be attributed to the latency involved in building packets and the transmitting them through the CPU's networking stack. Second, KV pairs that arrive at a slave in Maestro are asynchronously accumulated by seven receiver processors in parallel, allowing fresh updates from other workers to be quickly incorporated into the slave's state table.

Next, the update rate of KV pairs in each slave is increased by increasing the transmitter to receiver ratio P_{tx}:P_{rx}. As observed from Figure 6.12, the application speedup improves when more transmitter processors are added. Adding transmitter processors allows the FPGA slave to perform parallel updates on the state table and better utilize the network bandwidth by transmitting more KV pairs per second to other slaves. Further, since receiver processors outnumber transmitter processors, KV pairs are accumulated at higher rates.
In order to corroborate this hypothesis, we measure real-time network traffic at one of the ports of the NetFPGA router. Figure 6.14 illustrates the network trace characteristics for the PageRank benchmark when parallelized on two Maestro FPGA workers. For the transmitter to receiver ratio of 1:7, the network is utilized approximately at a rate of 10,000 packets per second (14.3 MBps). When an additional transmitter is added, the network utilization improves to approximately 19,000 packets per second (27.1 MBps) allowing the computation to finish early. Adding an additional transmit processor ($P_{tx}:P_{rx}=3:5$) further improves the network utilization.
to 30,000 packets per second (42.9MBps). For completeness, we also provide network traces for the same computation when executed in a 2-worker Maiter cluster in Figure 6.15. The average network utilization in Maiter is only 5000 packets per second (7.1MBps).

From Figure 6.12, we find that a balanced ratio of transmitter to receiver processors (4:4) yields the highest speedup in all benchmarks (26×, 16× and 4.1× for PageRank, Katz and Connected versus Maiter, or a speedup of 208×, 176× and 69.7× versus Hadoop).

When the $P_{tx}:P_{rx}$ ratio is increased further, higher update rates and lower accumulation rates cause RX FIFOs in Fig. 6.6 to overflow. Many KV pairs are lost, leading to incorrect convergence of the algorithm. To compensate for the higher update rates, we manually reduce the rate at which packets are transmitted from each FPGA by introducing a programmable delay between subsequent packet transmissions. The delay gives receiver processors sufficient time to process accumulations between subsequent packets without losing KV pairs. However, since the delay effectively lowers packet transmission rate, a drop in application performance is observed particularly for higher transmitter to receiver processor ratios. For example, a system with seven transmitter processors yields only a speedup of 17.9× versus Maiter.

The Katz benchmark has a lower speedup in comparison to the PageRank for the following two reasons: First, in the iterative formulation of Katz, a key chosen as the source node (s) is assigned an initial value of 1. All other nodes are initialized to 0. During an iteration, every key node multiplies its current value by a factor $\beta$ and propagates the result to other nodes. Values slowly trickle along the graph during the computation. In contrast, the computation in PageRank is more uniform across the entire graph. i.e. all nodes start to send and receive updates once the computation is initiated.
The second reason relates to the implementation of Katz in the FPGA. In the current implementation, an FPGA sends a packet when it has accumulated 150 “non-zero” KV pairs. i.e. the value of the KV pair must not be zero. This is a general optimization for all algorithms including PageRank and Connected to avoid sending zero values (which do not contribute any value to overall computation) to other FPGAs. But, in Katz, we will need to send KV pairs even if the value field is zero because, during the start of the computation there are not enough KV pairs to form a packet from non-zero KV pairs. As the computation progresses, the zero valued KV pairs choke the transmit path causing a drop in the overall speedup.

Like PageRank, Katz yields the highest speedup (16× versus Maiter, 176× versus Hadoop) when the total number of transmitters matches the number of receivers. However, no significant loss in speedup is observed even when an additional transmitter is added (Ptx:Prx=5:3) implying that the additional delay introduced at the transmitting side to match the drop in receiver rate does not lead to an observable increase in overall application execution time.

**Four workers:** For the four worker cluster, the problem size is doubled from that of the two worker experiment (5.2 million nodes/3.6 GB). The workload is evenly distributed between all slaves using a MOD partition function. Maiter runs on four CPU workstations interconnected in a LAN configuration. Figure 6.16 shows the speedup of Maestro for different processor configurations versus Maiter. Figure 6.17 shows speedup of Maestro for different processor configurations versus Maiter. For the PageRank benchmark, Maestro is 18× faster than equivalent Maiter implementation when each FPGA is configured with one transmitter and seven receiver processors. As more processors are converted to transmitters, the speedup improves. With four transmitter and receiver pairs in every FPGA, the four-worker Maestro executes 40× faster than the Maiter implementation and 360× faster than Hadoop. Katz demon-
Figure 6.16. Speedup of Maestro versus Maiter on 4 workers for different transmitter to receiver processor configurations. Graph size=5.2 million nodes.

Figure 6.17. Speedup of Maestro versus Hadoop on 4 workers for different transmitter to receiver processor configurations. Graph size=5.2 million nodes.
### Table 6.3. Maestro execution time for varying problem and cluster size

<table>
<thead>
<tr>
<th>Problem size (N)</th>
<th>Iterations (I)</th>
<th>Tp (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p=1</td>
<td>p=2</td>
</tr>
<tr>
<td>200k</td>
<td>239</td>
<td>291</td>
</tr>
<tr>
<td>400k</td>
<td>155</td>
<td>206</td>
</tr>
<tr>
<td>600k</td>
<td>131</td>
<td>181</td>
</tr>
<tr>
<td>800k</td>
<td>115</td>
<td>162</td>
</tr>
<tr>
<td>1 million</td>
<td>105</td>
<td>120</td>
</tr>
<tr>
<td>1.2 million</td>
<td>97</td>
<td>119</td>
</tr>
</tbody>
</table>

### Table 6.4. Network traffic volume for PageRank, 1.2 million nodes

<table>
<thead>
<tr>
<th>Maestro Configuration</th>
<th>Packets sent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cluster size=2</td>
</tr>
<tr>
<td>P&lt;sub&gt;tx&lt;/sub&gt;:P&lt;sub&gt;rx&lt;/sub&gt;=1:7</td>
<td>199,165</td>
</tr>
<tr>
<td>P&lt;sub&gt;tx&lt;/sub&gt;:P&lt;sub&gt;rx&lt;/sub&gt;=2:6</td>
<td>199,064</td>
</tr>
<tr>
<td>P&lt;sub&gt;tx&lt;/sub&gt;:P&lt;sub&gt;rx&lt;/sub&gt;=3:5</td>
<td>197,793</td>
</tr>
<tr>
<td>P&lt;sub&gt;tx&lt;/sub&gt;:P&lt;sub&gt;rx&lt;/sub&gt;=4:4</td>
<td>197,569</td>
</tr>
</tbody>
</table>

strates a speedup of 18.7× versus Maiter (356× versus Hadoop). Speedup drops when transmitters exceed receivers.

#### 6.11.3 Scalability - Varying Problem Size

Figure 6.18 summarizes the best case speedup with Maestro in a scaling problem size/cluster configuration. The FPGA in the one worker Maestro cluster includes eight transmitter processors. For two and four worker Maestro configurations, each FPGA was programmed with four transmitter and four receiver processors. In general, for all benchmarks Maestro demonstrates better speedup with larger problem sizes and cluster configurations. In the four worker configuration, Maestro offers 40×, 18.7× and 7.5× speedup versus Maiter for PageRank, Katz and Connected benchmarks. Figure 6.19 provides the best case speedup with Hadoop in a scaling problem size/cluster configuration. Maestro executes 360×, 356× and 105× faster than Hadoop respectively for PageRank, Katz and Connected benchmarks.
Figure 6.18. Best case speedup of Maestro versus Maiter for scaling problem and cluster sizes

Figure 6.19. Best case speedup of Maestro versus Hadoop for scaling problem and cluster sizes
Table 6.5. Resource utilization on a Stratix IV FPGA

<table>
<thead>
<tr>
<th>Resource</th>
<th>System Usage</th>
<th>Processor Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational ALUTs</td>
<td>64,178 (35%)</td>
<td>3,256 (1.7%)</td>
</tr>
<tr>
<td>Registers</td>
<td>70,299 (39%)</td>
<td>3,375 (1.8%)</td>
</tr>
<tr>
<td>Memory bits</td>
<td>1,621,781 (20%)</td>
<td>4,110 (0.03%)</td>
</tr>
</tbody>
</table>

Table 6.6. Energy/cost estimates for a 4 worker cluster executing PageRank

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Energy (KWh)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>Hadoop</td>
<td>$2,000</td>
</tr>
<tr>
<td>PageRank</td>
<td>Maiter</td>
<td>$2,000</td>
</tr>
<tr>
<td>PageRank</td>
<td>Maestro</td>
<td>$12,000</td>
</tr>
<tr>
<td>Katz</td>
<td>Hadoop</td>
<td>$2,000</td>
</tr>
<tr>
<td>Katz</td>
<td>Maiter</td>
<td>$2,000</td>
</tr>
<tr>
<td>Katz</td>
<td>Maestro</td>
<td>$12,000</td>
</tr>
<tr>
<td>Connected</td>
<td>Hadoop</td>
<td>$2,000</td>
</tr>
<tr>
<td>Connected</td>
<td>Maiter</td>
<td>$2,000</td>
</tr>
<tr>
<td>Connected</td>
<td>Maestro</td>
<td>$12,000</td>
</tr>
</tbody>
</table>

6.11.4 Scalability - Fixed Problem Size

Table 6.3 provides the total number of iterations and exection times $T_p$ for different problem sizes on a $p = 1, 2, 4$ FPGA Maestro configuration for Pagerank. Each FPGA implementation has one transmitter and seven receivers. A linear speedup is observed when additional FPGAs ($p = 1-4$) are used to solve a fixed size problem (e.g. problem size = 1200k). Each FPGA holds fewer state table entries as the problem is parallelized, resulting in a lower threshold for KV pair selection in larger cluster configurations. The drop in the threshold causes an overall increase in the number of iterations required to finish the computation.

Table 6.4 provides the total volume of packets transmitted by each Maestro node in a two-worker and four-worker cluster for the 1.2 million node PageRank problem. An FPGA in a two-worker cluster, transmits 199,165 packets with one transmitter processor. The total volume of traffic required to complete the computation does not significantly change when more transmitters are added within each FPGA. However, when the same problem is parallelized on four workers, the total volume of traffic sent by each FPGA drops by 23%. The drop in traffic can be attributed to the presence of fewer graph nodes within each FPGA.
6.11.5 Resource Usage

The logic and memory utilization of a 8 transmitter processor Maestro system on a Stratix IV EP4SGX230 device is shown in Table 6.5. Each processor in our system requires 3,256 ALUTs and 3,375 registers. The FPGA operates at a frequency of 125 MHz.

6.11.6 Energy/Cost Estimates

Table 6.6 compares the energy consumption and cost of executing three benchmarks in a four-worker cluster using Hadoop, Maiter and Maestro frameworks. For these comparisons, we assume that a CPU workstation costs $500 and consumes about 120W. Each Altera DE4 board costs $3000 and consumes approximately 10W power when attached to a x1 PCIe slot. Maestro consumes $238-342\times$ less energy in comparison to Hadoop for PageRank and Katz for a $7\times$ increase in the total system cost. Energy savings of approximately $35\times$ and $13\times$ are observed for these applications versus Maiter. As mentioned in Section 6.5, the FPGA Assistant CPUs are provided in this experimentation for prototyping. These processor-based assistants could be replaced by FPGA-based soft processors. Hence, they have been omitted from the energy and cost analysis.

6.11.7 Modeling Scalability

In this section, we model the scalability of the cluster beyond four FPGA workers. In an ideal scenario, when a problem with $N$ KV pairs is partitioned over $p$ FPGAs, each FPGA only needs to process $\frac{N}{p}$ state table entries. As a direct consequence, the computation should finish $n$ times faster than the time required on 1 FPGA. However, when the problem is parallelized, a larger share of the total links are now located outside each worker. Each slave not only needs to send more updates outside, but it has to also accommodate for the larger number of messages sent from other slaves to itself.
From real-time network traces observed at the ports of the NetFPGA router, we observe this higher communication cost reflects as an overall increase in the network utilization when a fixed size problem is parallelized over additional FPGAs. For example, the network utilization for a 1.2 million node problem parallelized on 2 FPGAs and 4 FPGAs is shown in Figure 6.20. When the problem is parallelized over two FPGA workers, the network link is utilized at a rate of 10,000 packets per second (the rate includes both transmitted and received packets at an FPGA). When the same problem is parallelized on four FPGAs, the link traffic increases to 20,000 packets per second. Based on this observation, we identify two factors that may limit the overall scalability of the system - the maximum capacity of the link and the maximum capacity of a receiver processor to accumulate the KV pairs in the received traffic.

The maximum capacity of the link places an absolute limit on the network utilization that can be achieved by each FPGA. Since each FPGA requires at least one receiver to process incoming traffic, the rate at which the receiver processor processes
KV pairs also influences the scalability of the system. We separately model these two factors.

**Link capacity limitation:** Assume that there are \( n \) FPGA workers in the cluster with each FPGA having 1 transmit and 1 receiver processor. Based on the data from network traces, the network utilization, \( B_{util} \) in packets per second can be formulated as a function of the number of FPGA workers, \( n \) as

\[
p_{util} = n \cdot b \text{ pps}; n \geq 2, b = 5000 \text{ pps} \tag{6.6}
\]

Since a transmitted packet includes \( K \) KV pairs where the size of a KV pair is 8 bytes, the total size of a packet is \((K+6)\cdot64\) bits including the 6-byte packet header. The maximum number of packets that can be transmitted or received per second over a link of bandwidth \( B_{max} \) bits per second is

\[
p_{max} = \frac{B_{max}}{((K + 6) \cdot 64)} \text{ pps} \tag{6.7}
\]

For a 1Gbps link, approximately 100,000 packets, each with 150 KV pairs can be transmitted or received per second. The condition for maximum link utilization is

\[
p_{util} = p_{max} \tag{6.8}
\]

Therefore, the maximum number of FPGAs that can operate over this link without exceeding the link bandwidth is

\[
n_{max,link} = \frac{p_{max}}{b} = \frac{100,000 \text{ pps}}{5000 \text{ pps}} = 20 \tag{6.9}
\]

Based on this model, we expect that the problem will scale linearly for 20 workers on a 1Gbps link. If the link capacity scales to 10Gbps, upto 200 workers can be supported.
**Receiver capacity limitation:** When a given problem is parallelized on \( n \) workers, assume that \( \frac{1}{n} \) of the total traffic in the link represents traffic directed to other FPGAs while the rest \( (\frac{n-1}{n}) \) of the traffic is received at each worker. The traffic arriving at a receiver processor \( (r_{rx/in}) \) may be computed as:

\[
r_{rx/in} = \left( \frac{n-1}{n} \right) \cdot p_{util} = \left( \frac{n-1}{n} \right) \cdot (n \cdot b \text{ pps}) \tag{6.10}
\]

Assume that a receiver processor can process a KV pair in \( c \) cycles. The processor operates at a frequency of \( f \) MHz and a packet contains \( K \) keys. The maximum rate at which a receiver processor can process packets is

\[
r_{rx/out} = \frac{f}{(c \cdot K)} \text{ pps} \tag{6.11}
\]

The condition for a receiver not to drop any packet is

\[
r_{rx/out} \geq r_{rx/in} \tag{6.12}
\]

\[
\frac{f}{c \cdot K} \geq \left( \frac{n-1}{n} \right) \cdot (n \cdot b) \tag{6.13}
\]

From Eq.6.13, the maximum number of FPGAs that can be supported without exceeding the receiver capacity will be

\[
n_{max,rx} \geq \left( \frac{f}{(c*K)} \right) \cdot b + 1 \tag{6.14}
\]

From simulations, we determined that \( c=12 \) clock cycles. The FPGA operates at \( f=125 \text{MHz} \) and keys per packet, \( K=150 \). The maximum rate at which each receiver processor processes packets \( (r_{rx/out}) \) is approximately 69,000 packets per second. From Eq. 6.14, approximately 15 FPGAs can be supported to scale the system with 1
receiver processor. In order to scale the system further, additional receiver processors may be added within each FPGA. For example, an additional receiver can scale the system to 30 FPGA workers.

In general, the scalability of the system is limited by the lower of the two bounds determined from Eq. 6.9 and 6.14.

\[ n_{\text{max}} = \text{Min}(n_{\text{max,link}}, n_{\text{max,rx}}) \]  

(6.15)

### 6.11.8 Comparison to Previous Work

FPMR reports a speedup of 33.5× versus a CPU implementation of MapReduce for RankBoost, a machine learning application to rank web documents. In contrast, our implementation of PageRank, a similar machine-learning application, demonstrates a speedup of 154× versus Hadoop on 1 FPGA. Further, Maestro can be scaled to yield higher speedups in larger configurations (up to 360× speedup for PageRank on four-worker system). Mars [40] implements MapReduce on graphics processors (GPGPUs) using Page View Rank that calculates the number of distinct page views from web logs to display the top 10 URLs that are frequently accessed. Mars demonstrates a speedup of 5× on the NVIDIA G80 GPGPU versus a MapReduce implementation on the Intel Quad-core processor. Our work improves the speedup and scalability from these previous implementations by applying asynchronous accumulative updates and prioritized data refinement.

### 6.12 Conclusion

In this chapter, we have presented Maestro, an FPGA-based distributed system that utilizes asynchronous accumulative updates (AAU) to execute iterative algorithms. This approach addresses the synchronization issue often found in distributed systems. Our work maps this approach to FPGA-based distributed systems, simpli-
fying system scalability and demonstrating significant speedups due to FPGA parallelism and specialization. Prioritized computations accelerate algorithm convergence through dynamic data refinement. We plan to make our FPGA code and software freely available.
CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Future Internet infrastructure will require networking and computing systems that offer performance and design flexibility to evolving web applications. While microprocessors offer fairly generalized solutions to a large class of problems in existing Internet systems, ASICs provide overly fine-tuned techniques at a higher cost. Our thesis is that heterogeneous systems that feature FPGAs and general-purpose processors are uniquely positioned to close the growing gap in application performance and design flexibility in next-generation Internet applications. In support of our thesis, we have demonstrated systems that integrate FPGAs with general-purpose processors in network virtualization and distributed cluster computing applications. In implementing these systems, we have exploited the reconfigurability, specialization and data-parallel architecture of FPGAs.

7.1 Summary of Contributions

As a first contribution of this dissertation, we have demonstrated techniques that integrate FPGAs to enable heterogeneous network virtualization platforms. Our system addresses scalability issues in previous FPGA-based virtual networking techniques with the aid of container virtualization technology and virtual network migration. Virtual networks hosted in an FPGA offer one to two orders of better throughput and lower latency in comparison to state-of-the-art network virtualization techniques that use container virtualization technology. A heterogeneous virtual networking system capable of supporting 15 virtual networks has been demonstrated.
We showed that FPGA partial reconfiguration can be exploited to dynamically reconfigure virtual networking parameters without affecting other shared networks in hardware. Our evaluation demonstrates that reconfiguring selective regions of the FPGA chip via partial reconfiguration allows virtual networks to be customized $20\times$ faster than the static reconfiguration approach which requires a full-shutdown of the device.

We proposed ReClick as a new programming model for prototyping networking protocols in FPGA hardware. ReClick aims to simplify the design effort required to develop virtual networking applications in programmable hardware by providing an application design entry point which is higher than most hardware description languages. Optimizations built into the ReClick framework allow limited FPGA resources to be effectively shared between several virtual networks. Our evaluation shows that an IPv4 router built from ReClick components can forward packets at 1Gbps.

In the final part of the dissertation, we have demonstrated techniques to enable heterogeneous computing clusters by integrating FPGAs with microprocessor-based workstations. A salient feature of our Maestro system is the use of asynchronous accumulative updates to break the synchronization barriers in general-purpose cluster computing frameworks like MapReduce and Hadoop. We demonstrated a scalable hardware architecture to implement this model on an FPGA. Our evaluation of Maestro with three iterative algorithms show that a four FPGA cluster offers upto $360\times$ speedup in comparison to an equivalent CPU-based Hadoop cluster. Further scalability can be achieved by spatially parallelizing the computation (e.g. by adding more processors within the FPGA) or with additional FPGA boards.
7.2 Future Work

The research presented in this dissertation provides guidelines for future work in heterogeneous network virtualization and distributed cluster computing.

**Heterogeneous virtual networking:** The virtual networking techniques demonstrated in this thesis were evaluated in a laboratory environment. In the future, these techniques may be applied in a broader setting such as overlay networks (e.g. PlanetLab [51]) or virtual networking testbeds (e.g. GENI [76]). For example, the virtual networking testbed, GENI, already supports several NetFPGA nodes that may help achieve this goal. In such a setting, a software-based management layer or “virtual network hypervisor” can simplify the process of deploying virtual networks on heterogeneous hardware and software resources. The hypervisor layer may be extended to obviate the need for significant user intervention during the virtual network migration process.

**Virtualized Forwarding Tables in Programmable Hardware:** Network virtualization requires careful sharing of memory resources (e.g. SRAM/DRAM) to implement shared forwarding structures. While the implementation of forwarding tables in general-purpose routers has been well-studied, the implementation of virtual forwarding tables poses additional challenges such as the need for strong isolation between different tables and the need for efficient memory utilization. In recent years, some initial work has been performed on software-based virtualized forwarding tables (e.g. virtual forwarding tables using tries [37] [73]). The feasibility of such techniques may be evaluated in programmable hardware. For example, Multiroot [38] demonstrates a trie-based approach for implementing virtualized forwarding tables in hardware.

**Partial reconfiguration for virtual networking:** Partial reconfiguration is a promising technique that can be exploited to avoid service disruptions in networking hardware. In this research, we demonstrated the utility of partial reconfiguration
in customizing virtual networking parameters on a Virtex II Pro FPGA. Advanced FPGAs like Xilinx Virtex 7 and Altera Stratix V provide more flexible partial reconfiguration interfaces that eliminate the need for column placed placement of reconfigurable regions, as needed for Virtex II. These techniques may be applied to increase the number of virtual networks that share the FPGA.

**Programming models for FPGA-based virtual data planes:** We demonstrated a hierarchical design methodology for developing virtual data planes from simple *components* without compromising packet forwarding throughput. In the future, the library of components may be expanded further to feature a rich set of common networking features. Yet another possible direction for research is the use of C-based high-level programming interfaces such as Open Computing Language (OpenCL) to develop virtual networking components. OpenCL [11], for example, provides a unified approach to develop code for a variety of heterogeneous platforms like CPUs, GPGPUs and FPGAs. An interface for OpenCL is already supported by Altera for high-end FPGAs.

**Evaluation of Maestro using better clustering algorithms:** Our evaluation of Maestro uses a simple partitioning function (MOD) to distribute the workload between workers in a cluster. The MOD partitioning function, however, does not minimize the communication cost (e.g. the number of edge cuts) between different partitions. In some cases, the number of edge cuts between two graph partitions may be as high as 50% of the total edges in the graph. Better clustering algorithms may be used in Maestro to localize the computation in the cluster. Chaco [41] and Metis [45] are two useful graph partitioning tools that integrate popular clustering algorithms such as Kernighan-Lin, spectral and inertial partitioning. These tools may be used to build a graph partitioning front-end for Maestro.

**Load balancing strategies in heterogeneous clusters:** Since FPGA and CPU workers offer varying degree of parallelism, the workload may be assigned in an
asymmetric fashion in a heterogeneous cluster. For example, an FPGA with more transmitter processors can be assigned a larger share of the workload so that the overall computation time may be minimized.

**Applicability of the AAU model in GPGPUs and multi-cores:** Our work provided initial insights into the feasibility of using asynchronous accumulative updates in clusters that use FPGAs and CPUs. In theory, the AAU model may be applied in the context of other heterogeneous systems that integrate multi-cores and general-purpose graphics processing units (GPGPUs). We expect that techniques developed as part of this work will provide useful implementation guidelines for such systems.
BIBLIOGRAPHY


[80] Unnikrishnan, Deepak, Vadlamani, Ramakrishna, Liao, Yong, Dwaraki, Abhishek, Crenne, Jeremie, Gao, Lixin, and Tessier, Russell. Scalable Network


