12-18-2020

System Design and Implementation for Hybrid Network Function Virtualization

Xuzhi Zhang

SYSTEM DESIGN AND IMPLEMENTATION FOR HYBRID NETWORK FUNCTION VIRTUALIZATION

A Dissertation Presented

by

XUZHI ZHANG

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 2020

Electrical and Computer Engineering

SYSTEM DESIGN AND IMPLEMENTATION FOR HYBRID NETWORK FUNCTION VIRTUALIZATION

A Dissertation Presented

by

XUZHI ZHANG

Approved as to style and content by:

Russell Tessier, Chair

Lixin Gao, Member

Michael Zink, Member

Jie Xiong, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor Professor Russell Tessier for his many years of thoughtful, patient guidance and support. I sincerely appreciate the opportunity he gave me to pursue my research under his supervision, as well as the constant encouragement and constructive criticism to help me improve my writing and presentation skills. I'm so grateful for his efforts in my research work and so many valuable suggestions, which make this work possible. He is the best advisor and teacher I could have wished for. I especially want to thank him for his care and support to me during the most difficult time in my family.

I also would like to thank Professor Lixin Gao for her advices on my research. Her valuable ideas and comments on my work are greatly appreciated. I want to thank Professor Michael Zink and Professor Jie Xiong for serving as my PhD committee members, reading the dissertation, critiquing my work and providing constructive suggestions to make this dissertation better.

I am very grateful to my colleagues and good friends Xiaozhe Shao and Georgios Provelengios. Thanks to them for not hesitating to share their knowledge and experience, and many useful discussions on my research. It was an unforgettable experience to study, do research and have fun in Reconfigurable Computing Group for the past five years. Thanks to all my former and current RCG labmates Shrikant Vyas, Naveen Kumar Dumpala, Shivukumar B. Patil, Chethan Ramesh, Naren Prabhu, Aiden Gula, and Lijuan Xia. Many of them have made valuable contributions towards the work in this thesis.

Finally, this thesis is dedicated to my wife Rusi, and our lovely babies. Whenever I feel depressed and lose my spirit, their love and smile always encourage me. My wife

has been a constant source of love, encouragement and support all these years. She was doing so well to take care of everyone in our family. Without her I could never get my thesis done. I thank her from my heart for the love, patience and understanding.

ABSTRACT

SYSTEM DESIGN AND IMPLEMENTATION FOR HYBRID NETWORK FUNCTION VIRTUALIZATION

SEPTEMBER 2020

XUZHI ZHANG

B.Sc., HARBIN ENGINEERING UNIVERSITY, HARBIN, CHINA

M.Sc., HARBIN INSTITUTE OF TECHNOLOGY, HARBIN, CHINA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier

With the application of virtualization technology in computer networks, many new research areas and techniques have been explored, such as *network function virtualization (NFV)*. A significant benefit of virtualization is that it reduces the cost of a network system and increases its flexibility. Due to the increasing complexity of the network environment and constantly improving network scale and bandwidth, it is imperative to aim for higher performance, extensibility, and flexibility in the future network systems. In this dissertation, hybrid NFV platforms applying virtualization technology are proposed. We further explore the techniques used to improve the performance, scalability and resilience of these systems.

In the first part of this dissertation, we describe a new heterogeneous hardware-software NFV platform that provides scalability and programmability while supporting significant hardware-level parallelism and reconfiguration. Unlike a traditional

NFV system which virtualizes dedicated hardware appliances into software-based network functions running on general-purpose microprocessors, our computing platform takes advantage of both field-programmable gate arrays (FPGAs) and microprocessors to implement numerous virtual network functions (VNFs) that can be dynamically customized to specific network flow needs. A distinctive feature of our system for enhancing scalability is the use of global network state to coordinate NFV operations. Traffic management and hardware reconfiguration functions are performed by a global coordinator which allows for the rapid sharing of network function states and continuous evaluation of network function needs. With the help of state sharing mechanism offered by the coordinator, customer-defined VNF instances can be easily migrated between heterogeneous middleboxes as the network environment changes. A resource allocation algorithm dynamically assesses resource deployments as network flows and conditions are updated.

In the second part of this thesis document, we explore a new session-level approach for NFV that implements distributed agents in both FPGA- and processor-based middleboxes to steer packets belonging to different sessions through session-specific service chains. With our session-level approach, we support inter-domain service chaining with both FPGA- and processor-based middleboxes, dynamic reconfiguration of service chains for ongoing sessions, and the application of session-level approaches for UDP-based protocols. To demonstrate our approach, we establish inter-domain service chains for QUIC sessions, and reconfigure the service chains across a range of FPGA- and processor-based middleboxes. We show that our session-level approach can successfully reconfigure service chains for individual QUIC sessions. Compared with software implementations, the distributed agents implemented on FPGAs show better performance in various test scenarios.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Virtualization has revolutionized the computing and information technology (IT) world. This technology abstracts applications, guest operating systems, networks or data storage away from the true underlying hardware or software. It brings cost and space savings, faster provisioning, easier backup and update, better scalability, and flexibility. Virtualization has been applied in a diverse set of computing technologies. For example, memory virtualization is a popular technique used in computer architecture to share distributed memory resources among processes. Hardware virtualization is the virtualization of computers as complete hardware platforms which allow multiple systems or users to share one host. Application virtualization is a software technology that encapsulates computer programs from the underlying operating system on which it is executed. With the popularization of virtualization technology in various fields of IT, computer networking researchers have turned their attention to it. New research areas include network function virtualization (NFV) [52].

NFV brings virtualization to the network by decoupling the network functions from proprietary hardware appliances and virtualizing them into software or logic blocks which can operate on general purpose microprocessors or reconfigurable hardware. The potential benefits of NFV is significant. Virtualization of network functions deployed on commodity hardware helps to reduce capital investment and energy consumption, decreasing the time to market of a new service or product. NFV enables operators to enforce high-level policies expressed by enterprise or service networks by

directing flows through appropriate network function instances, and further enables isolation among high-level policies performed for different customers.

1.1 Trends and Challenges in NFV

As the Internet has evolved, increasingly diverse network functions, or middleboxes, have been deployed to provide services for business and social needs. Typical network functions, such as firewalls, network address translations (NATs), load balancers, packet classification, and proxy caches, process packets in sophisticated ways to ensure reliability and improve performance in enterprise, service provider, and cloud provider networks. These services traditionally were provisioned by telecommunications service providers (TSPs) through the deployment of proprietary devices and equipment for each service function. As customer requirements for more diverse and higher data rate services increase, TSPs must purchase, store and operate new physical equipment. In addition, to bring new services into the networks, TSPs must acquire a variety of middleboxes and hire skilled professionals to integrate and maintain their services. These issues lead to high capital expenditure (CAPEX) and operational expense (OPEX) for TSPs [52]. As a result, TSPs must find ways to build more dynamic and service-aware networks that have reduced operating and capital expenses and improved service flexibility.

Network function virtualization leverages virtualization technology to replace dedicated application specific integrated circuit (ASIC) based appliances with software-based network functions running on generic commodity hardware. In this way, a given service is broken down into a set of virtual network functions (VNFs) that can be implemented in software running on one or more industry standard servers. TSPs can easily instantiate and relocate these VNFs at different network locations without purchasing and installing new hardware. NFV enables TSPs to rapidly test and deploy newly targeted and tailored services based on customer needs.

Several design considerations must be examined before implementing an NFV. The network should allow users to access the applications they need, when they need them. Therefore, TSPs need to consider the following key factors before deploying an NFV.

- **Performance:** Performance is a factor network users consider when they choose a service provider. As network functions are decoupled from proprietary hardware appliances and virtualized into software applications, throughput and latency may be affected. Wang and Ng [84] illustrated that virtualization can cause significant throughput instability and abnormal delay variations in a data center network. To keep performance degradation as small as possible after moving a given VNF from dedicated hardware to a NFV platform, possible solutions include leveraging modern software technologies, such as Intel's Data Plane Development Kit (DPDK)[1], or using a form of hardware acceleration [9].

- **Management:** The management of a virtualized environment within an NFV framework is a big challenge. The NFV infrastructure needs to be able to instantiate VNFs in the right location at the right time. The cost and processing capacity of resources may vary significantly between network points, which increases complexity in decision making for VNF deployment. The NFV framework should also be flexible enough to dynamically allocate and scale hardware resources to meet new customer needs or changes in the network environment. The underprovisioning of network services may cause service disruption. Thus network carriers often overprovision their services [25] to guarantee stability. To improve resource utilization, NFV can exploit the elasticity feature of the infrastructure to effectively adjust resource allocation without affecting the provision of service. This manage-

[1]https://www.dpdk.org/

ment functionality is helpful, especially when hardware resources are limited in the network.

- **Reliability:** Service reliability may be affected by two aspects. As mentioned above, the elasticity of service provisioning may require the scaling and migration of VNFs based on traffic load and user demand. These operations are necessary but increase the uncertainty of the system. Reliability requires consideration of how scheduling impacts system stability. For example, a service outage caused by VNF migration can be reduced by temporarily running the target VNF on a third-party resource. An unexpected traffic increase or service element failure can also influence system reliability. To increase reliability, fault tolerance capabilities or redundancy can be added to NFV systems.

- **Security:** When TSPs migrate telecommunications infrastructure to NFV, security is an important consideration. Security impacts system resiliency [77] and the overall quality of offered services [76]. NFV can be combined with other cloud technologies. Multiple service subscribers may share the same cloud hardware infrastructure which raises security concerns. Security risks include: (1) the NFV infrastructure should be protected from subscriber services; (2) functions and services from different subscribers should be protected and isolated from each other.

- **Scalability:** TSPs provide services to millions of subscribers. This level of support requires NFV to be scalable and responsive to vast numbers of VNF requests from service users. NFV scalability also requires the ability to scale up a VNF across computation resources to meet performance demand. If a VNF is deployed across multiple middleboxes, global state must be maintained.

Virtual machines (VMs) operating on general-purpose microprocessors have been used to accommodate VNF instances. Although microprocessors are straightforward to program and allow for fast network interfacing using toolsets such as the DPDK,

they are inherently sequential and specialized network functions often require significant run time. Field programmable gate arrays (FPGAs) offer a hardware acceleration solution that provides parallelism, specialization, and programmability. FPGA logic blocks can be partially reconfigured which allows a part of the FPGA functionality to be dynamically changed while the device is still in operation. This feature guarantees NFV system flexibility and reliability.

Network functions maintain statistics about packets and flows during packet processing. Per-flow states are maintained locally by a single middlebox. For multi-flow states, if a single middlebox meets the requirements of a given VNF, the states still can be maintained locally. However, if a VNF is deployed on multiple middleboxes and each middlebox only processes a subset of flows, a state sharing mechanism is necessary to share multi-flow states among distributed middleboxes. A feasible solution involves building a global state table in the NFV orchestrator to manage the state sharing across middleboxes.

1.2 Thesis Overview

Previous research has demonstrated the benefits of introducing FPGAs into NFV systems [36] [79] [56]. Although comprehensive, these works do not directly discuss global state management or how to use global state to allocate NFV resources for available functions.

In the first part of this thesis, we describe CoNFV, a network function virtualization platform based on FPGAs, microprocessors, and supporting software running on commodity hardware. CoNFV is a distributed NFV environment that efficiently supports state sharing for a scalable collection of middleboxes. As the functional needs of the network change based on link capacity or state-based triggers, a global *coordinator* rebalances the allocation of VNFs via the creation of virtual machine threads and the dynamic reconfiguration of FPGA modules. To demonstrate our system, a library

of FPGA-based and software modules has been implemented and tested for the following VNFs: specialized SQL attack detection, distributed denial-of-service (DDoS) detection, packet firewall, and network address translation (NAT). These function modules, implemented in either FPGA hardware or processor software, are swapped into middleboxes on demand. Our system is aided by a scheduling and allocation algorithm that automatically considers the performance capabilities of the target NFV resource versus the requested function. In some cases, multiple microprocessors are grouped together to achieve needed throughput, latency, and computer performance levels.

A differentiating feature of our new system versus previous ones is the use of globally-shared state in determining function allocation and scheduling. Our allocation tool periodically identifies changes in required VNF deployment, assembles the components from available libraries, and dynamically reconfigures the component FPGAs and VMs that implement the network functions. Our prototype network function virtualization environment is assessed using Intel DE5 FPGA boards, microprocessor-based VirtualBox[2] middleboxes, and a 10 Gigabit-per-second (Gbps) SDN network switch. The system is shown to be scalable both in middlebox count and quantity of shared state.

In the second part of this thesis, we describe a new session-level approach for dynamic service function chaining in heterogeneous NFV systems. FPGAs provide an ideal platform for network functions implemented in middleboxes due to their parallelism, specialization, and adaptability. However, integrating both processor and FPGA-based middleboxes into a network can be a challenge. Network traffic must be steered through these heterogeneous middleboxes in a distributed fashion at the right time. We explore a new session-level approach to solve this challenge. Our

[2]https://www.virtualbox.org/

new approach steers traffic along middlebox service chains by deploying distributed agents on middleboxes, which avoids reliance on a central controller. We implement agents on both FPGA- and processor-based middleboxes. Agents steer packets of individual sessions through corresponding service chains without any alterations to end-host applications, middlebox applications, or IP routing. Agents can reconfigure a service chain by inserting/removing middleboxes in the chain. The FPGA-based agent cooperates with a partial reconfiguration IP core to manage the dynamic reconfiguration of middlebox functions during service chain reconfiguration. We verify our new approach with QUIC protocol sessions and show the benefits of implementing our agents with FPGA circuits versus software-based implementations. Our session-level approach successfully reconfigures service chains for individual QUIC sessions.

1.3 Thesis Outline

The remainder of this thesis document is organized as follows: In Chapter 2, we introduce the motivation and necessary background material for this thesis. Background on network function virtualization, software-defined networking, FPGA architecture, the application of FPGAs in NFV systems, and service function chaining are described. Finally, background on hardware virtualization and Docker platform are provided.

In Chapter 3, we introduce the first major contribution of this dissertation - a new heterogeneous hardware-software NFV platform called CoNFV that provides scalability and programmability while supporting significant hardware-level parallelism and reconfiguration. We first describe the framework design and implementation of our system, highlight the novel state sharing mechanism across middleboxes, and present the methods and procedures to dynamically reconfigure NFV resources in response to the network environment changes. In the next section, we detail the implementation of four high-performance FPGA-based VNF modules. Next, this chapter presents

the implementation details of the global state table which is used for cross-middlebox state sharing. Our evaluation of the system, presented in the last section, shows that FPGA-based VNF modules have significant performance advantages over corresponding software implementations. Our system is shown to be scalable for collections of network functions exceeding one million shared states. A function migration performed with FPGA full reconfiguration and VM thread creation takes about 12 seconds in our NFV platform.

In Chapter 4, we introduce FPGA partial reconfiguration to the NFV platform. Multiple programmable logic regions on the same FPGA chip can be individually reconfigured to different VNF instances while the other parts of the FPGA circuit are still functional during the partial reconfiguration process. We describe a performance and resource model for computation resources (CRs) in an NFV system and present our performance-aware VNF deployment algorithm which includes offline initialization and online reallocation of VNF instances. Our evaluation shows that partial reconfiguration can accelerate the migration of FPGA-based VNF modules by a factor of 15. Our algorithm can effectively adjust the deployment of VNF instances on available NFV resources in response to the network traffic load changes.

In Chapter 5, we introduce a new session-level approach that can be used to establish inter-domain service chains for QUIC protocol sessions, and reconfigure the service chains across a range of FPGA- and processor-based middleboxes. We first present the architecture of our session-level approach. Then, we explain how to use our approach to dynamically reconfigure a service chain. In the next section, we describe the details of implementing this approach on an FPGA. Finally, we evaluate our method with three experiments using both FPGA- and processor-based middleboxes.

In Chapter 6, we summarize this thesis work and provide directions for future work.

CHAPTER 2

BACKGROUND

2.1 Network Function Virtualization

The term network function virtualization (NFV) was conceived in 2012 by a specification group, part of the European Telecommunications Standards Institute (ETSI) [14]. The concept of NFV originated from service providers who wanted to make adding new network functions and applications easier to use and faster. NFV provides a way to create, distribute, and operate networking services. It virtualizes an entire class of network node (or middleware) functions into building blocks that may be connected, or chained, together to provide a range of networking services. Typical network functions include border controllers (such as firewalls, load balancers, and wide-area network (WAN) accelerators) that protect a network.

The NFV framework consists of three main components: virtualized network functions (VNFs), network function virtualization infrastructure (NFVI), and network function virtualization management and orchestration architecture framework (NFV-MANO Architecture Framework) [15]. Figure 2.1 shows the high-level NFV framework. The VNFs are typically software implementations of network functions that can be deployed on a NFVI. The NFVI is the totality of all hardware and software components that build the environment where VNFs are deployed. The NFV-MANO is the collection of all functional blocks, data repositories used by these blocks, and reference points and interfaces through which these functional blocks exchange information for the purpose of managing and orchestrating NFVI and VNFs.

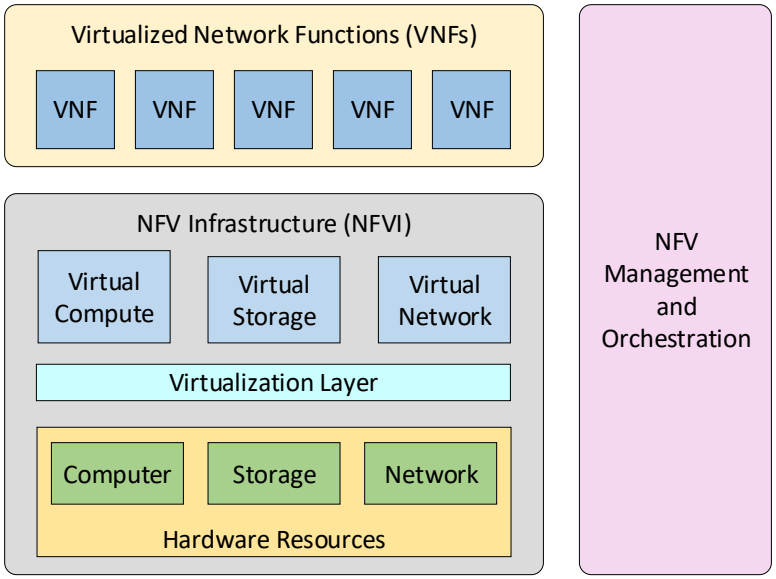

Figure 2.1: High-level NFV framework

Prior to NFV, a network border controller typically consisted of a collection of custom hardware appliances, each of which was designed for a specific network function. With the advance of hardware virtualization technology, it became possible to decompose traditional network border controller functions into virtual machines (VMs) running different software, and eventually into reconfigurable FPGA components. When designing and developing the software and FPGA circuits that provide VNFs, it is possible to break operations into components and package those components into one or more functions. To provide isolation among network functions customized for each customer, it is important to install each component into a resource that can be fairly shared. Programmable middleboxes are often hosted in one or more physical nodes consisting of commodity hardware. They are connected by tunnels to satisfy the requirements of a customer. For example, each customer might provide a policy rule set for its firewall and install those rule sets in its own middlebox. In addition to

the firewall, the customer might install a wide area network (WAN) accelerator that is installed in the same or a different middlebox.

Software-based network function virtualization has mainly focused on the control and management of middlebox functions. Qazi et al. [62] employed software-defined networking (SDN) principles to enforce policies for traffic steering. Sherry et al. [70] proposed to use cloud services to perform network functions. Gember et al. [21] aimed to provide mechanisms for tenants to specify their middlebox needs, and automatically deploy and scale middleboxes that maximize performance. A number of studies [22, 48] have focused on designing software-based programmable middleboxes in a virtualized environment. Although software-based middleboxes are flexible, the parallelism and specialization of hardware may be needed for high throughput functions.

2.2 Software-Defined Networking

Software-defined networking (SDN) enables fast response to network changes by managing switch-based traffic from a centralized control console. An SDN instance consists of three major parts: application, control plane, and data plane (Figure 2.2). The application label indicates diverse network functions to satisfy specific demands, such as a security mechanism [71] or a network measurement solution [89]. Applications communicate with a controller at the control plane via the northbound interface of the control plane. The controller uses the southbound interface of the SDN-enabled switch to connect to the data plane. The data plane is the part that supports a shared protocol (e.g., OpenFlow [49]) with the controller and handles packets based on the configurations that are manipulated by the controller.

SDN principles can be traced back to 2004 when the Internet Engineering Task Force (IETF) began considering various ways to decouple the control plane of a network from the data plane that forwards network traffic [87]. Early interest led to

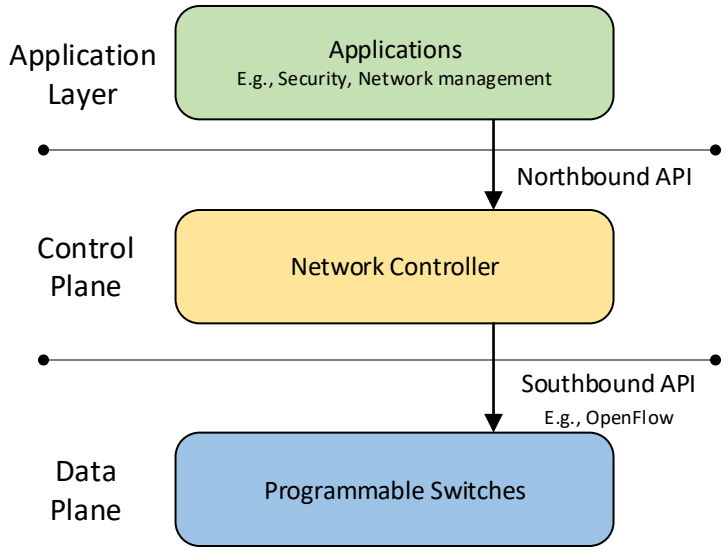

Figure 2.2: SDN architecture concept

the creation of OpenFlow from the Ethane project [10]. Many researchers point to OpenFlow as being synonymous with SDN. However, OpenFlow is an open standard for a communications protocol that enables a controller to interact with switches, as shown in Figure 2.2. OpenFlow is not the only protocol available or in development for SDN.

NFV is complementary to SDN but not dependent on it (or vice-versa). NFV can be implemented without an SDN being required, although the two concepts and solutions can be combined, and potentially greater value accrued [14]. An SDN is a critical step on the path to a modernized network, but many services, such as routing, wide area network (WAN) optimization and security are still tied to the underlying hardware. NFV addresses this problem by decoupling the network function from the hardware and virtualizing it, allowing it to be run in a virtual machine or as a reconfigurable circuit on FPGA. For NFV, SDN can help to enhance performance, simplify compatibility with existing deployments, and facilitate operation and maintenance procedures.

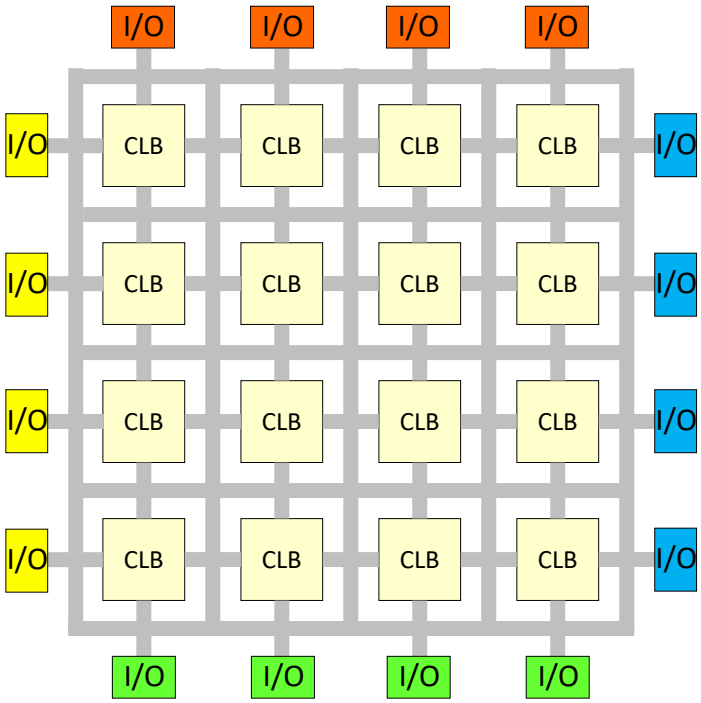

Figure 2.3: FPGA architecture

2.3 Overview of FPGA Technology

An FPGA is an integrated circuit that can be electrically programmed in the field to become almost any kind of digital circuit or system [40]. In contrast to an ASIC where the circuit behavior is permanently fabricated into the silicon, the behavior of an FPGA is reconfigurable after device fabrication. This flexibility is attributed to internal programmable logic blocks and routing circuitry [40]. Compared to ASICs, FPGAs require less time and money to achieve an initial working design, getting designs to market more quickly. However, the enhanced flexibility of FPGAs makes them larger, slower, and more power hungry than their ASIC counterparts [39].

The most common SRAM-based FPGA architectures consist of configurable logic blocks (CLBs) which implement logic functions, programmable routing to interconnect these functions and I/O blocks to make off-chip connections (Figure 2.3) [40]. A

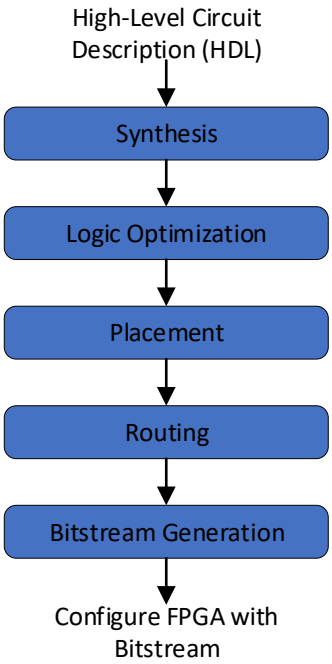

Figure 2.4: FPGA application development flow

CLB is the basic logic resource in an FPGA. CLBs are generally composed of three basic components: lookup tables (LUTs), flip-flops, and multiplexers [2]. FPGA CLBs are typically interconnected with programmable routing circuitry that runs horizontally and vertically across the device. State-of-the-art FPGAs also integrate blocks of internal static RAM (SRAM), digital signal processing blocks (DSPs), digital clock managers (DCMs), high speed I/O interfaces and hardened networking protocol blocks such as Gigabit Ethernet and PCI Express cores [80].

By using these pre-built CLBs and programmable routing channels, developers can implement custom hardware functionality on FPGAs as needed. The FPGAs are programmed and configured using a hardware description language (HDL) such as Verilog and VHDL. After finishing the HDL design, the developer uses a computer-aided design (CAD) tool (e.g., Quartus Prime [30] for Intel FPGAs and Vivado [86] for Xilinx FPGAs) to translate the hardware description to an optimized technology-

mapped netlist. The netlist is placed and routed for the FPGA device architecture under constraints of area, clock period and power. Finally, a bitstream is generated for target FPGA device programming. A typical FPGA application development flow is illustrated in Figure 2.4.

2.4 FPGAs in NFV

Reconfigurable logic provides an ideal platform for network functions due to the parallelism, specialization, and adaptability offered by FPGA devices [82]. These characteristics match well with the multi Gigabit-per-second throughput constraints frequently imposed on networking infrastructure and the need for frequent updates required by changing packet analysis and filtering metrics. As FPGAs continue to be integrated into cloud computing environments [9] and data centers [61], their use in network and application processing will continue to grow.

A number of FPGA-based platforms have been deployed for network applications involving performance improvement, load balancing, and reliability. Song and Lockwood [73] demonstrated the effectiveness of using a ternary content-addressable memory (TCAM) for data lookups in packet classification. This FPGA-based TCAM structure was used to exactly match a series of predefined prefixes and port numbers. A packet classifier [35] was used in a decision-tree-based, 2-D multi-pipeline architecture in a Virtex 5 device to obtain up to 80 Gbps throughput. A wide range of FPGA-based network intrusion detection systems have been implemented using CAMs [24], shift and compare circuits [7, 60], and Bloom filters [12]. FPGA logic allows for the implementation of a massive number of parallel matching circuits and Bloom filter hash functions that can be customized to a changing set of matching rules, including the entire SNORT NIDS ruleset [60]. Hardware-based functions for ruleset matching can easily be synthesized from a high-level language, such as C. Although DDoS prevention using FPGAs has received less attention than packet classification and

NIDS, worm identification and matching circuits have been implemented in FPGAs that operate at line rates [46].

Over the past five years, increasingly more complex and flexible NFV systems that support VMs and FPGAs have been developed. Kachris et al. [36] provide an analysis of the potential use of FPGA reconfiguration to dynamically support functions such as firewalls, packet parsing, IP address lookup, deep packet inspection, and virus scanning. A comprehensive system [78] takes advantage of the flexibility and on-the-fly reconfigurability of FPGA and CPU resources within a cloud data center. This approach builds upon OpenStack resource management functions to dynamically allocate both types of resources. The system supports application programming interfaces (APIs) [45] and implements a collection of string-matching functions. Ge et al. [19] also use OpenStack with partial FPGA reconfiguration to support deep packet inspection and network address translation (NAT). Nobach et al. [57] developed a system that can move functions between microprocessors and FPGAs on demand. State migration is an important aspect of the work [55]. The Hyper system [75] uses a global mediator to hide middlebox resource heterogeneity when assigning VNFs to resources. Although these works are comprehensive and build on OpenStack resources, global state management is not directly addressed or used to allocate NFV resources for the available functions. State and configuration management for subnetwork FPGAs can be limited by a lack of global state coordination support and the inability to swap functions using network-wide information.

2.5 Service Function Chaining

Service function chaining (SFC) [27] is a method to steer traffic through a set of intermediate services provided by diverse middleboxes to compose complex network services. Traditionally SFC deployments are static and rely on network configurations to stitch middleboxes together. As NFV and software-defined networking (SDN)

technologies mature, dynamically-composed service chains and fine-grained packet forwarding become possible.

Previous research efforts have implemented service function chaining. A number of solutions steer network traffic by controlling network elements (e.g. switches). Stratos [20] and E2 [59] use fine-grained forwarding rules to build static service chains within clouds. OpenNF [22] and Split-Merge [65] provide dynamic service chaining by updating packet forwarding rules in SDN switches. These solutions rely on logically centralized controllers to install and update forwarding rules, inheriting the shortcomings of using a central controller. It is difficult to outsource middleboxes to the cloud since one controller cannot control the entire path. Reliance on a central controller brings the risk that a controller failure may lead to errors in the distributed network. When a controller updates forwarding rules due to changes in policy, topology, or load, the paths of ongoing sessions may be changed, so that packets in the same sessions will not be able to traverse the same middleboxes. In addition, since all forwarding rules are determined by the central controller, middleboxes do not have the ability to directly select a specific service chain for their outgoing packets.

The weakness of the central controller approach can be overcome by using a session-level solution. Key session-level functions are performed by the middleboxes, providing scalability and control decentralization. Existing protocols for session-level service chaining include Network Service Header (NSH) [63], Segment Routing Header (SRH) [17], and Dysco [90]. NSH supports service chaining encapsulation without the use of forwarding rules. This protocol has received widespread attention from both academia and industry [38, 64, 85]. Despite the popularity of the NSH protocol, it is an intra-domain format and does not support dynamic reconfiguration.

SRH [1] encodes a list of IPv6 addresses of virtual network functions (VNFs) in packet headers to perform SFC. This approach requires service functions to support the header and does not support dynamic reconfiguration. Dysco is a session-level

protocol that steers the traffic of a TCP session through an SFC. Like our approach, it can dynamically reconfigure the SFC without requiring changes to IP routing, end-host applications, or middlebox applications. Dysco does not support connectionless protocols or UDP-based session protocols, such as QUIC and has not been implemented with FPGAs.

2.6 Hardware Virtualization and Docker Containers

In a traditional physical computing environment, software such as an OS or enterprise application has direct access to the underlying computer hardware and components, including the processor, memory, storage, certain chipsets and OS driver versions. This approach makes it difficult to move or reinstall software on different platforms. Hardware virtualization [8] helps decouple software from computer hardware. It provides a logical view of a physical resource to entities that require shared access to the resource.

Three types of hardware virtualization are full virtualization, para-virtualization, and OS-level virtualization [43]. In full virtualization (Figure 2.5(a)), the virtual machine simulates hardware to allow unmodified guest code targeted to the same instruction set to be run in isolation. An example of a full virtualization platform is VirtualBox [11]. In para-virtualization (Figure 2.5(b)), the virtual machine does not necessarily simulate hardware, but instead offers a special API that can be used by modifying the guest OS [41]. OS-level virtualization (Figure 2.5(c)) runs applications and replicas of the same operating system on the same server. The guest OS environments share the same instance of the OS as the host system. Thus, the same OS kernel is also used to implement the guest environments, and the applications running in a given guest environment view it as a stand-alone system [37].

Docker [51] is a computer program that performs OS-level virtualization. It is designed to make it easy to create, deploy, and run applications by using containers.

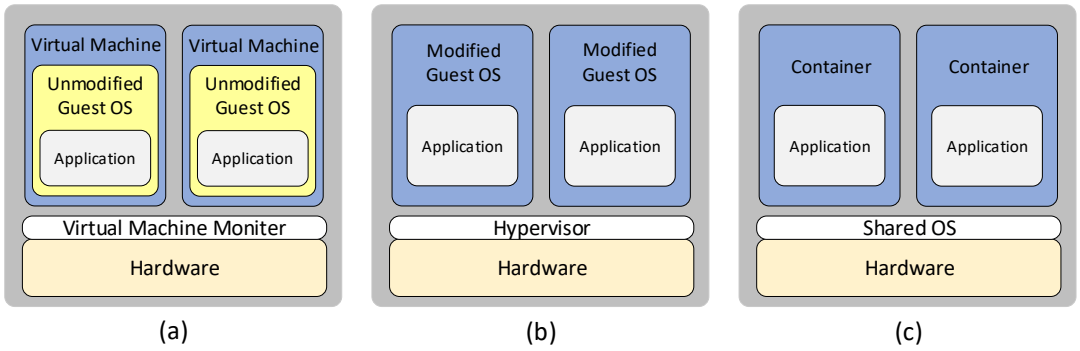

Figure 2.5: (a) Full virtualization (b) Para-virtualization (c) OS-level virtualization

Containers are isolated from each other and bundle their own application, tools, libraries and configuration files. They can communicate with each other through well-defined channels. All containers are run by a single operating-system kernel and are thus more lightweight than virtual machines.

CHAPTER 3

SCALABLE NETWORK FUNCTION VIRTUALIZATION FOR HETEROGENEOUS MIDDLEBOXES

Over the past decade, a wide-ranging collection of network functions in middleboxes has been used to accommodate the needs of network users. Although the use of general-purpose processors has been shown to be feasible for this purpose, the serial nature of microprocessors limits NFV performance. In this chapter, we describe a new heterogeneous hardware-software approach to NFV construction that provides scalability and programmability, while supporting significant hardware-level parallelism and reconfiguration. Our computing platform uses both FPGAs and microprocessors to implement numerous NFV operations that can be dynamically customized to specific network flow needs. As the number of required functions and their characteristics change, the hardware in the FPGA is automatically reconfigured to support the updated requirements. Traffic management and hardware reconfiguration functions are performed by a global coordinator which allows for the rapid sharing of middlebox state and continuous evaluation of network function needs.

The remainder of this chapter is organized as follows. Section 3.1 presents our scalable hardware and software system. In Section 3.2, we present the implementation details of the framework. Section 3.4 describes the details of four FPGA-based middlebox applications. We discuss the methodologies of state sharing and traffic management respectively in Section 3.3 and Section 3.5. Finally, in Section 3.6, we provide an evaluation of our heterogeneous hardware-software NFV platform.

3.1 System Design

3.1.1 System Overview

It is common for middleboxes positioned across a subnetwork to deploy distributed functions using commodity hardware, custom hardware, virtual machines, or reconfigurable hardware. Information from multiple packet flows must often be utilized for these stateful, distributed functions. Information is collected locally during packet processing from flows that pass through the middlebox. For a variety of applications, such as NAT and SQL injection (SQLi) attack detection, a distributed approach allows for parallel analysis of multiple flows, each collecting correlated information. The scalable CoNFV system collects global state information and shares this information among distributed FPGA and microprocessor packet processors. The CoNFV coordinator gives each middlebox access to global state information using programmable interfaces. Subsets of this information are cached in the middleboxes for some applications.

Middlebox and coordinator functionality can be quickly updated as network function needs change. For example, many NFV operations can initially be assigned to software for low and moderate traffic loads. As network traffic and computational workloads increase for a function, instances can be migrated to FPGA-based hardware. A traffic and workload decrease for a specific function can have the opposite effect. The allocation of functions to middleboxes is dynamically assessed and orchestrated by the coordinator as state-based network conditions are processed. The coordinator automatically reallocates resources as needed.

An overview of our global state-sharing system for heterogeneous middleboxes is shown in Figure 3.1. Microprocessor- and FPGA-based (DE5) middleboxes are distributed across the network. The middleboxes share state information through TCP connections to the CoNFV coordinator. As shown in Section 3.6, the coordinator is able to handle state for a scalable set of middleboxes, with minimal packet processing

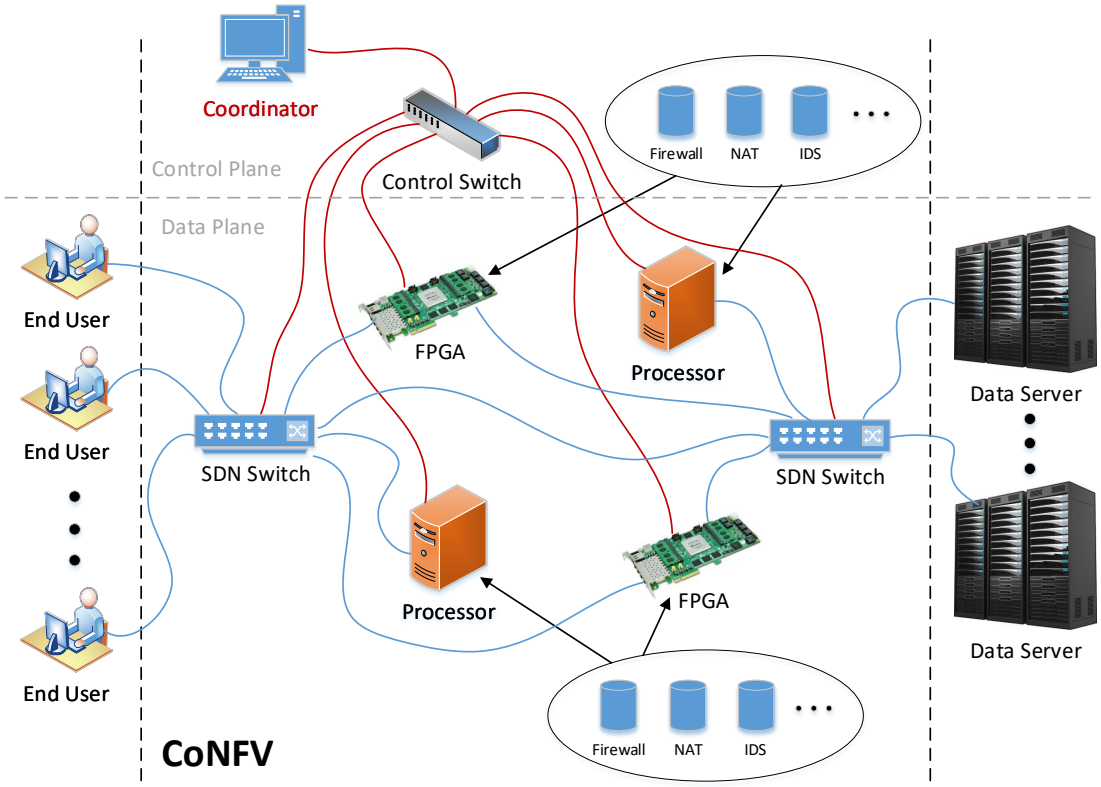

Figure 3.1: Overview of the CoNFV configurable NFV system using processor- and FPGA-based (DE5) middleboxes

slowdown. SDN switches are used to control middlebox access and provide support for chaining. The network setup represents a number of interconnect configurations, including those found in data centers.

Figure 3.2 shows the framework of the system. The coordinator stores global state values in a table as a set of key-value pairs. Each middlebox can access global state (GV) using a key. The *state manager*, a software module which can be configured for each application, can both retrieve and update state. The *resource evaluator* assesses the current utilization of middlebox resources in response to messages and state variables and can choose to perform middlebox resource rebalancing. The *configuration manager* creates an entry for each enrolled middlebox in the resource table and updates their properties according to the information collected by the resource eval-

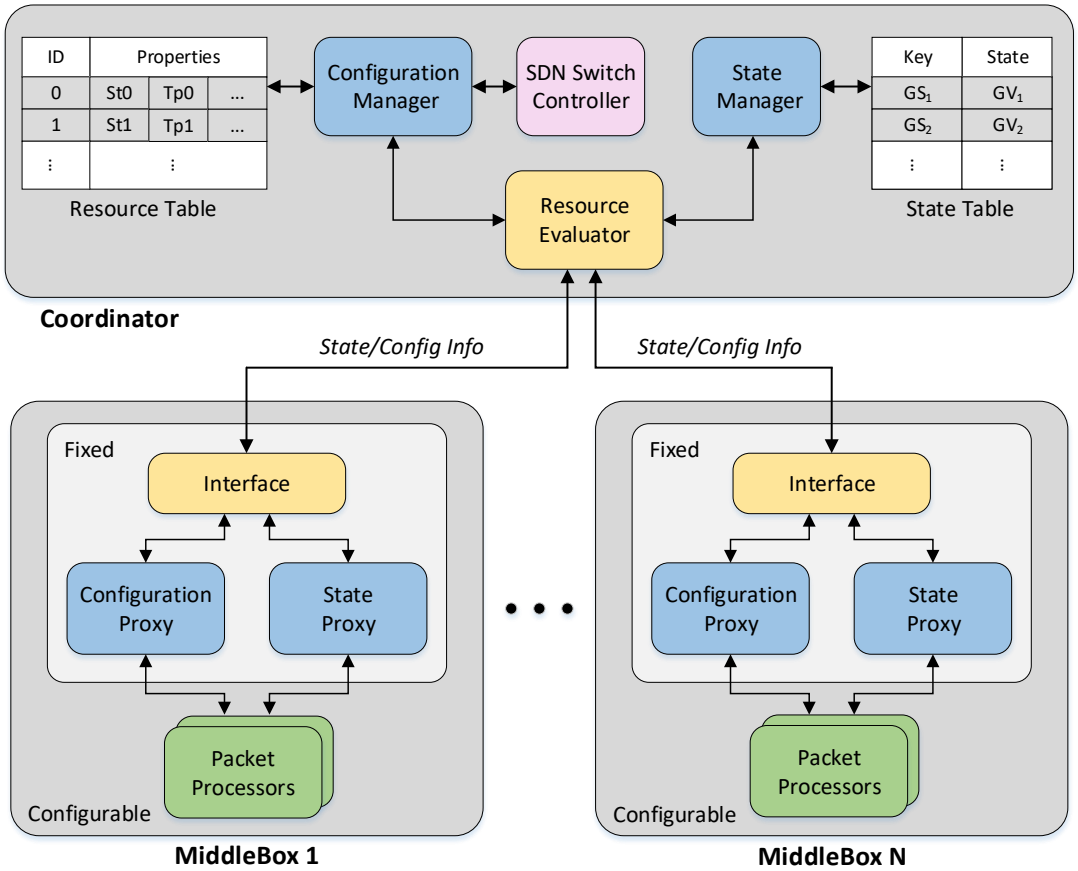

Figure 3.2: Middleboxes and global coordinator interaction. Middleboxes can be either processor- or FPGA-based

uator. These properties include the resource state (St idle/active) and throughput (Tp). The configuration manager coordinates the resource assignment based on the global middlebox information recorded in the resource table and triggers an in-line *SDN switch controller* to steer traffic flows running through the SDN switch.

Each middlebox contains one or more *packet processors* and an associated *state proxy* module. After a state request originates in the packet processor, the state proxy module generates and sends state requests to the coordinator and receives state updates from the coordinator. The *configuration proxy* module coordinates either software thread activation/deactivation for packet processors or hardware re-

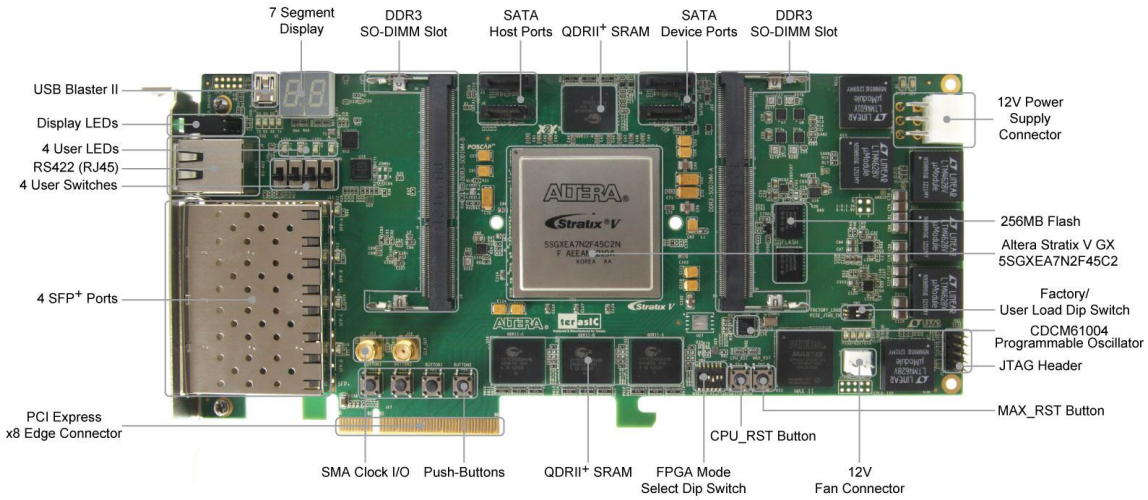

Figure 3.3: DE5-Net FPGA board (top)

configuration for FPGA packet processors. A control *interface* allows for interaction with the coordinator. The specific functions of these modules are detailed in Section 3.2.

3.1.2 DE5-Net FPGA Development Kit

The FPGA-based middleboxes in our system were built on the Terasic DE5-Net FPGA board [80]. Figure 3.3 shows the top view of the board. The DE5 is empowered with the top-of-the-line Intel 28-nm Stratix V GX FPGA. The Stratix V GX FPGA features integrated transceivers that transfer at a maximum of 12.5 Gbps, allowing the DE5 to be fully compliant with version 3.0 of the PCI Express standard, as well as allowing an ultra low-latency, straight connections to four external 10G SFP+ modules. Considering the user demands of high capacity and high speed for memory and storage, the DE5 also delivers with two independent banks of DDR3 SO-DIMM RAM, four independent banks of QDRII+ SRAM, high-speed parallel flash memory, and four SATA ports.

The specific version of the Stratix V FPGA we used in our project is 5SGXEA7N. This FPGA includes 234,720 adaptive logic modules (ALMs), 2,560 20-Kbit (M20K)

embedded memory blocks, 256 variable precision DSP blocks, and 28 fractional phase-locked loops (PLLs). The ALM is the basic building block of Stratix V devices. An ALM has an 8-Input fracturable look-up table (LUT), two dedicated embedded adders, and four dedicated registers. The hardware resources provided by the Stratix V GX FPGA are totally sufficient for all of our designs.

3.1.3 Cross-Middlebox State Sharing

Our system relies on state sharing for two types of actions: function triggering and state retrieval. Inspection functions evaluate network traffic and examine packets for monitoring, intrusion detection, and identification of other invasive attacks. Manipulation functions examine and modify flows by dropping, updating or creating new packets. State sharing for these two types of flows proceeds as follows:

Trigger state: For inspection functions, data packets are passively inspected as they enter a middlebox for specific characteristics of attacks such as DDoS or SQLi. If an event is observed that requires a global state update, state information both in the middlebox and in the coordinator are updated. As the state is updated in the centralized state table on the coordinator, it is checked by the resource evaluator to determine if remediation elsewhere in the network is needed. In Section 3.4, we describe how CoNFV can be used to address distributed DDoS and SQLi attacks. A firewall or packet filter can be enabled at one or more points in the network in response.

Retrieval state: For manipulation functions, global states are updated during packet processing. Middleboxes that require retrieved state generally manipulate packets. In the case of state retrieval, individual packet processors request state information if it is not available locally. The coordinator provides a global repository for state information and can update state as needed. A common use of state retrieval is for NAT. When NAT receives the first packet of a flow it creates state which

determines the translation from an external (IP address, port) pair to an internal (IP address, port) pair on the local subnetwork. This information must be shared across all middleboxes performing NAT translation for the subnetwork to avoid (IP address, port) assignment overlap. In CoNFV, translation information (global state) is stored in the coordinator. If a middlebox receives a packet and its translation information is not stored locally, the information can be obtained from the centralized repository.

3.1.4 Dynamic Resource Management

NFV resources must be managed using a global view of function deployment. To manage all NFV resources in an organized manner, the coordinator creates a resource table that records the current working state of all accessible resources in the network. The working state is different from the sharing state we discussed in the last subsection. It includes the operating status (i.e. idle or occupied) and the demand versus supply of computing power for NFV resources at a particular moment. One form of computing power is throughput.

In response to computing power mismatching or changing threats or monitoring goals, resources are reallocated under the control of the configuration manager in the coordinator. This unit coordinates the migration, creation, and destruction of functions in real-time to meet functional needs. For processor-based middleboxes, VM threads are created or destroyed in response to stimuli from the coordinator. For FPGA-based systems, portions of the FPGA circuitry are swapped to change functionality. As shown in Figure 3.2, FPGA resources are split into fixed resources that manage function interfaces and packet processing resources that can be dynamically reconfigured. For example, in response to the configuration proxy, portions of the FPGAs can be swapped.

After the rebalancing of the resource assignment, traffic flows need to be steered to corresponding middleboxes. The configuration manager coordinates the flow steering

in collaboration with the SDN switch controller by rewriting the flow table located in the under-controlled SDN switch. OpenFlow, a standard protocol for enabling SDN, is leveraged.

3.2 Framework Implementation

Our coordinator and middlebox framework includes commodity processor-based components, FPGA boards and an SDN switch. The coordinator is implemented using a processor-based Intel Duo server (2.66 GHz, 4GB). Processor-based middleboxes are implemented using a twelve-core Intel Xeon workstation (2.4 GHz, 32 GB SDRAM, two 10 Gbps NICs, and four 1 Gbps NICs). FPGA-based middleboxes are implemented using Terasic DE5 boards that include Stratix V FPGAs. TCP sockets are used to enable middlebox/coordinator interactions. The communication between the coordinator and the middleboxes is sufficiently frequent that the coordinator maintains a live connection for each middlebox since it is costly to initialize a new connection for each state operation. The SDN switch is a Netgear ProSafe M4300-8X8F 10 Gbps switch with 16 data ports and a 1 Gbps control port. The coordinator and SDN switch are interconnected via a 1 Gbps link.

A high-level view of processor- and FPGA-based middleboxes appears in Figure 3.4. In this configuration, network functions with the highest throughput and lowest latency are assigned to the FPGA on the DE5 board. The DE5 contains 16 GB SDRAM, 256 MB flash, a Stratix V 5SGXEA7N FPGA and four 10 Gbps Ethernet ports. Three 10 Gbps ports are used for data input and output and the fourth is used for 1 Gbps communication with the coordinator.

When the number of needed middleboxes exceeds available FPGA hardware, additional middleboxes can be spawned in software on the PC servers. A PC server is sliced into virtual machines using VirtualBox which allows full virtualization of a guest operating system. VirtualBox allows multiple isolated user spaces (virtual machines).

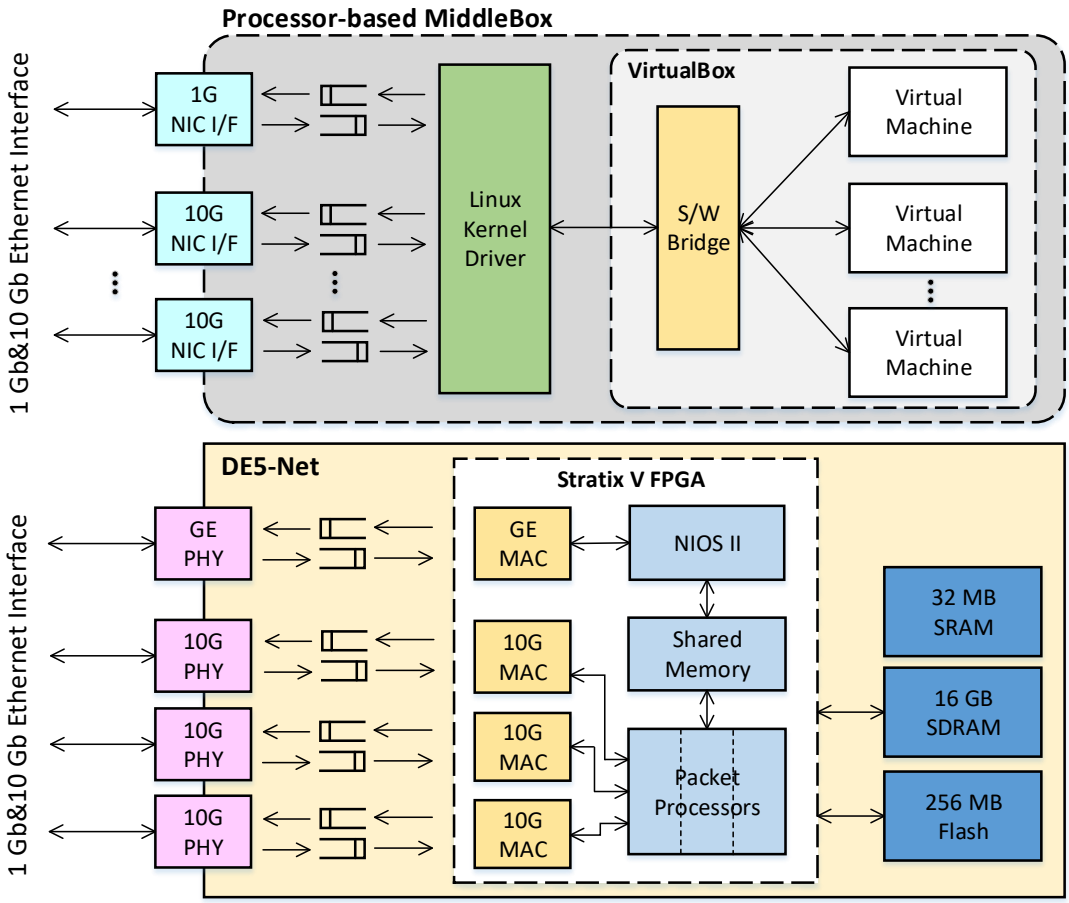

Figure 3.4: High-level overview of processor- (top) and FPGA-based (bottom) middleboxes in CoNFV

Each virtual machine operates like a standalone server. Software middleboxes are effectively isolated from each other in separate VirtualBox VMs that guarantee a fair share of CPU cycles and physical memory to each middlebox. Hardware and software middlebox functions can be customized based on the designer's specifications.

3.2.1 FPGA-based Middlebox Platform

A detailed view of the FPGA platform that can accommodate multiple packet processing middlebox functions is shown in Figure 3.5. We build an on-chip system by using the Qsys [32] system integration tool and instantiate necessary intellectual

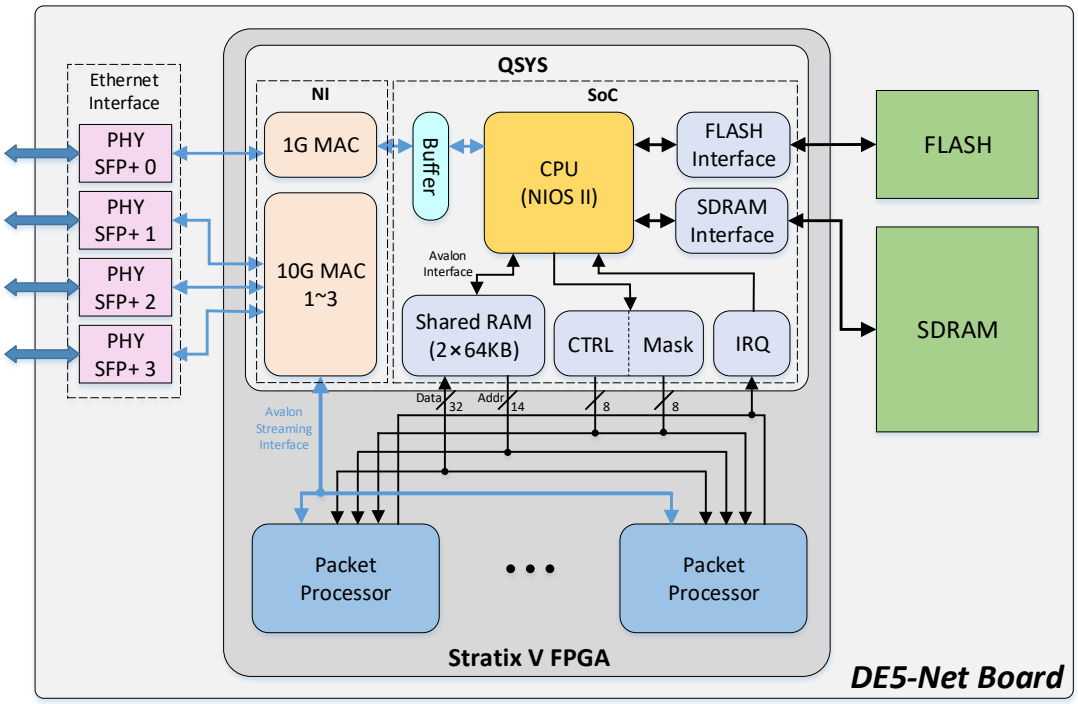

Figure 3.5: Detailed FPGA implementation for multiple middlebox packet processors

property (IP) functions in it. A Nios II [31] soft microprocessor is used as the interface, state proxy, and configuration proxy. This resource can communicate with the coordinator via a TCP connection implemented on a 1 Gbps link through a switch. The interface between the Nios II and one or more middlebox packet processors takes place via shared memory, a control register and an interrupt request (IRQ) controller accessed with the Avalon interface [33]. The packet processors implement functions in conjunction with a network interface (NI) that includes media access controller (MAC) IP cores, data queues and port controllers. Incoming data from the PHY are placed in the input queues. Processed packets are sent to the output queues from which they are forwarded to the physical Ethernet interfaces.

Qsys is the next-generation system integration tool in Intel Quartus Prime software. It saves significant time and effort in the FPGA design process by automatically generating interconnect logic to connect IP functions and subsystems. In our

FPGA platform, we use Qsys to build the main body of the fixed FPGA circuitry which includes two subsystems: the network interface (NI) subsystem and the Nios II-based system on chip (SoC) subsystem (Figure 3.5). The NI subsystem contains one 1G MAC IP core, three 10G MAC IP cores, and a direct memory access (DMA) controller used to manage the data switching between the 1G MAC and the Nios II microprocessor connected through a data buffer. Three 10G MAC cores are connected individually to different packet processors via the Avalon streaming interface [33].

In the SoC subsystem, we utilize the Nios II soft microprocessor working as the CPU of the system. The Nios II is a 32-bit RISC embedded-processor architecture. To support its working, necessary hardware peripherals are required and connected to the Nios II via the Avalon interface. Two off-chip memories (SDRAM, flash memory) are accessed by the Nios II via memory interface controllers. SDRAM is used for data and instructions, and flash memory is used to save the boot code and other arbitrary data we want to keep after power off. We developd embedded software working on Nios II microprocessor by using the MicroC/OS-II [3] and NicheStack TCP/IP Stack [4]. The software operates as a TCP client, a state proxy and a configuration proxy for all packet processors in one middlebox.

The implementation shown in Figure 3.5 illustrates the signal interfaces associated with the middlebox packet processors. These interfaces include data, address, and control connections to the shared memory and the network interface. These interfaces represent an effective *boundary* for partial FPGA reconfiguration of middlebox functionality. For this project, four middlebox functions for NAT, SQLi detection, DDoS detection, and packet firewall have been created with the interface, allowing for interoperability.

3.2.2 Coordinator Implementation

3.2.2.1 Coordinator and SDN Switch Initialization

During system operation, middleboxes can either be active or idle. To indicate availability, a middlebox informs the coordinator via an enroll message that includes information about the middleboxs compute capabilities. Prior to use, a middlebox must be registered with the coordinator and the SDN switch must be configured to forward flows to required destination middlebox. The coordinator's middlebox registration function, which is implemented in the configuration manager and SDN switch controller blocks, performs this function.

The configuration manager maintains a resource table for each middlebox. The table contains the following per-middlebox information: device ID, device status, device type, available NFV functions, assigned switch ports, and supply/demand processing capacities of the current middlebox. Upon system startup, the configuration manager sets the device status and other information in the resource table as *enroll* messages from middleboxes are received. When a middlebox completes operation in response to a message from the coordinator or an unplanned service interruption, an *exit* message is sent to the coordinator.

OpenFlow protocol is used to communicate between the coordinator and the SDN switch. The SDN switch controller in the coordinator oversees setting ports in the SDN switch via a series of flow modification (flow-mod) messages. These messages configure the switch by writing values into the *flow table* in the switch. Entries in the table are used to route incoming packets based on header information. During system startup, the switch sends its operational parameters to the SDN switch controller. If a packet arrives with a header that does not match an entry in the flow table, a default rule will broadcast the packet to the switch output ports. A reply message is used to record input and output port information and source and destination MAC

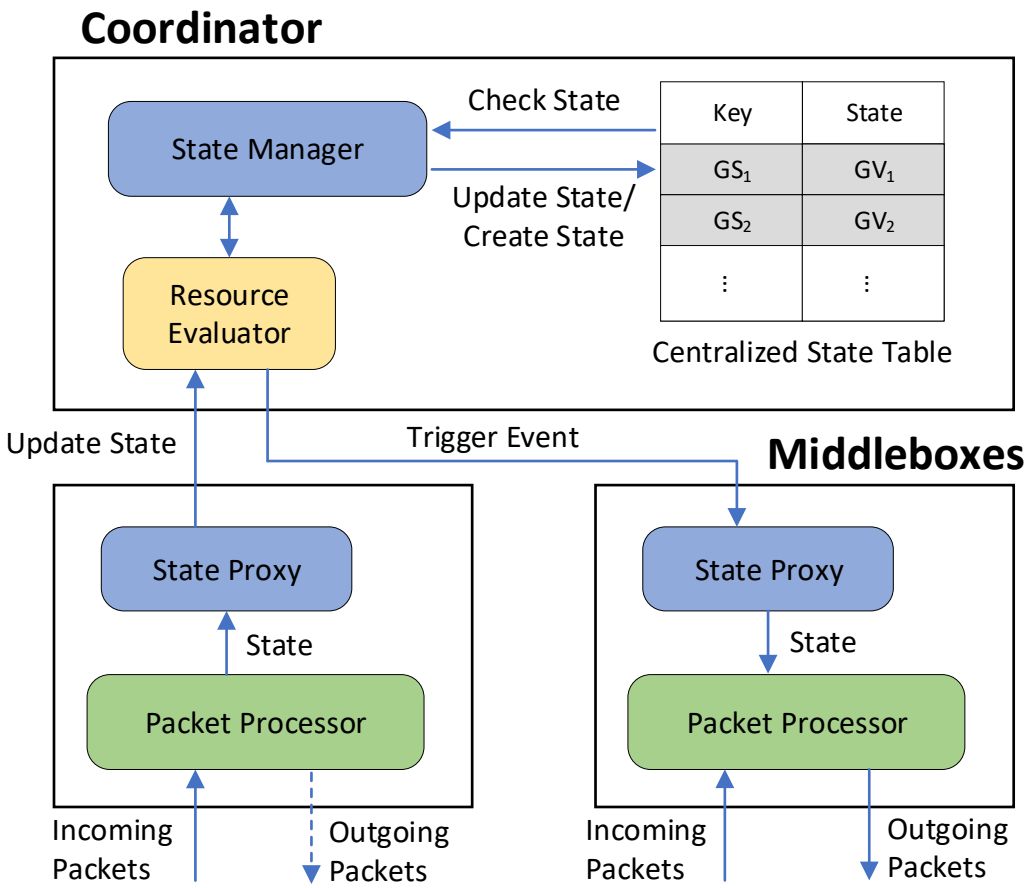

Figure 3.6: Trigger state operations

addresses in the flow table. Port information is also forwarded to the SDN switch controller.

3.2.2.2 Trigger State

A state table of trigger states is located in the coordinator. Middleboxes update trigger states through the state proxy during packet processing. Inside the coordinator, the state manager updates or creates trigger states according to the received state from middleboxes. As Figure 3.6 shows, when a packet comes into a middlebox, the packet processor inspects the packet. According to the semantics of the network

function, the middlebox may send the packet out and the inspection result might lead to a state update. Whenever the state manager updates or creates a trigger state, a state checker in the *resource evaluator* is triggered to detect malicious activities based on the new state. If a malicious activity is detected, the associated reactions, such as logging or notification, are engaged. Other middleboxes might be triggered by the resource evaluator to take appropriate protective measures, such as filtering the incoming packets.

Trigger states do not directly affect the packet processing. They are maintained to detect malicious activities. The semantics of detections are determined by the network function designer and provided to the coordinator for use by the resource evaluator.

3.2.2.3 State Retrieval

Asynchronous state operations used in our system allow a packet processor to process other packets without blocking while state is retrieved from the coordinator. However, asynchronous state operations might put packets out of order. For example, if the processing of a packet does not need a state operation, the packet can be processed immediately without waiting for the state return. Network functions that satisfy this condition are not uncommon. For example, for NAT, every packet in a flow requires the same mapping from one (IP address, port) pair to another (IP address, port) pair. Packets with known translations can proceed while others wait for translation information. During asynchronous state operation, the middlebox is able to process, for instance, the next incoming packet first. When the state is returned from the coordinator, the middlebox continues the processing of the previous packet. Asynchronous state operations buffer packets that require coordinator lookups using a *packet buffer table*. Figure 3.7 illustrates the procedure of asynchronous state operations. The state proxy maintains a packet buffer table. Each packet that incurs a

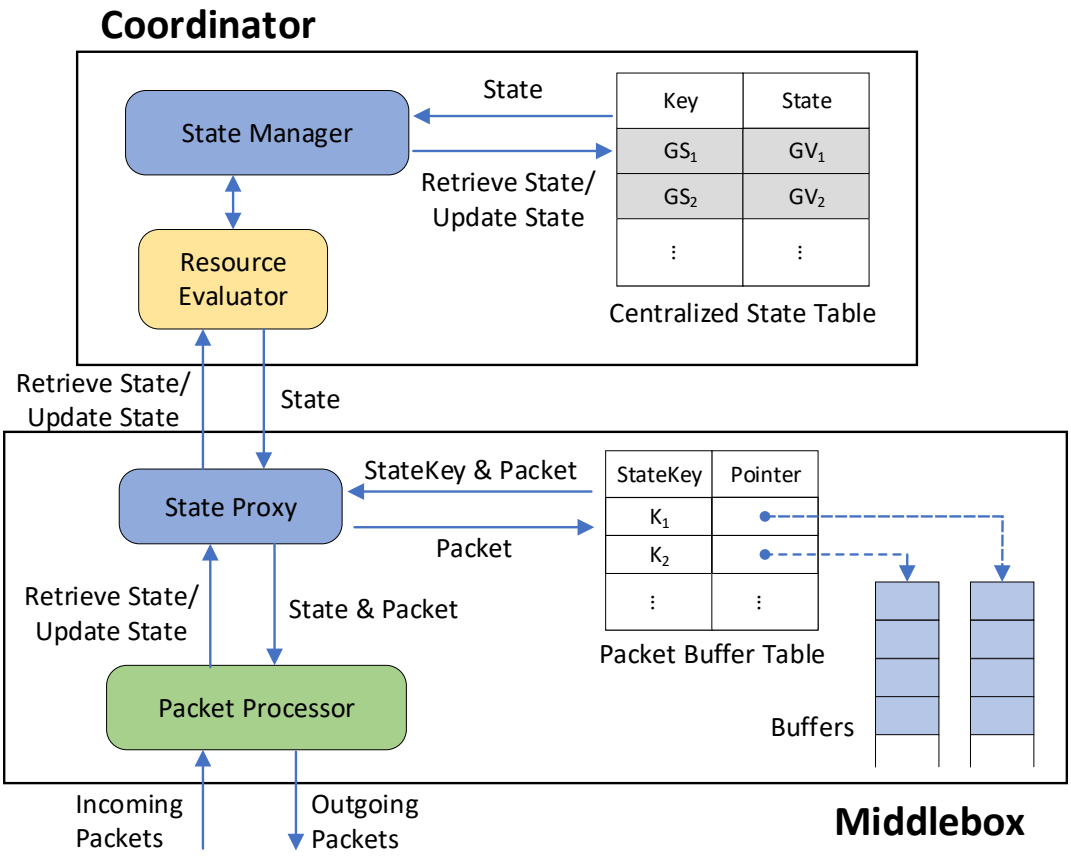

Figure 3.7: State retrieval operations

state operation is buffered in the table. Packets in the table are indexed by the keys of global states. Then, when the state is returned, the associated packet is retrieved from the table. If the table becomes full, an incoming packet may be dropped if it requires a coordinator lookup.

During packet processing, state retrievals can be much more frequent than state updates. In this case, it is beneficial to cache global states at middleboxes to reduce remote retrieval delay. To cache states, the state proxy in each middlebox maintains a *cache table* that stores the key-value pairs of states. When the packet processor retrieves a state, the state proxy checks the cache table first. If it misses, the state proxy retrieves the state from the coordinator. When the state returns, it is added

into the cache table. To record the locations of all local copies of state, the coordinator maintains a set of locators for each state value in a *duplication table*.

To keep copies consistent, the coordinator collects two kinds of information: where the copies of state are located and when the state is updated. When a state value needs to be updated in the coordinator, an invalidation message is first sent to affected middleboxes to clear stale values. Following acknowledgment messages from the middleboxes, the new state value is written in the centralized state table. As a result, local cache tables always either have a valid translation or must request an up-to-date one from the global coordinator. For the NAT application, state table invalidations only occur when the table is full.

3.2.3 Dynamic Reconfiguration

The assignment of VNFs to middleboxes is a dynamic process based on two factors, the throughput requirements of data streams and threats, as assessed by the resource evaluator in the coordinator. The DE5 provides a high-performance platform to implement middleboxes. However, the choice of an FPGA platform for virtualization does create scalability concerns. Not all middleboxes may contain an FPGA or there may be insufficient resources to implement all needed middlebox functions in FPGAs. As a result, our system allows for the seamless use of both hardware and software middleboxes in the same system with the same coordinator interfaces and support for VNF implementation of the same functions at different performance levels. Both types of resources are considered for dynamic VNF allocation.

Although minor updates to the hardware middlebox through configuration registers can enable parallelism and provide flexibility, it may not be sufficient for substantial changes in threats which require new hardware modules. As a result, techniques are needed to trade off computation between hardware and software to best use re-

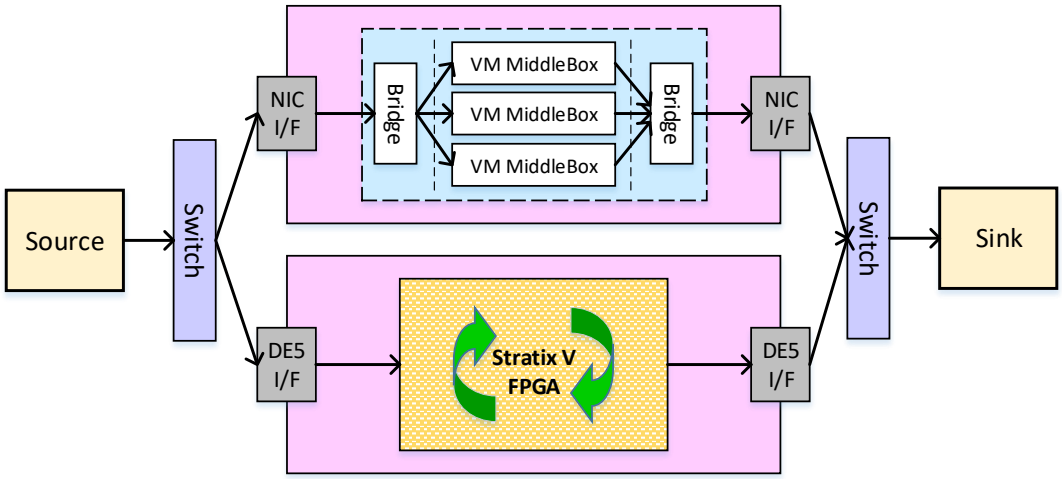

Figure 3.8: Multi-receiver setup for scalable NFV including dynamic FPGA reconfiguration

sources. This evaluation takes place under control of the configuration manager based on feedback from the middleboxes.

Resource assessment - To assess the throughput performance of currently executing VNFs, middleboxes send update messages to the coordinator with input rate and output throughput statistics every 10 ms. The resource evaluator dynamically determines the need for spawning, elimination, or migration of VNFs across middlebox resources. The configuration manager in the coordinator may also receive a trigger from the resource evaluator to consider middlebox resource allocation. The resource table in the configuration manager is used to compare current resource deployment, required middlebox computation, and the ability to accommodate triggers. A request for VNF migration, spawning, or removal is added to a task queue in the coordinator that is checked once every 0.1s.

Configuration update - To configure VNF functionality on a processor-based or FPGA-based middlebox, a message is sent from the coordinator to the host system with the required action and VNF specified. A software VNF middlebox is started in

an isolated VirtualBox. To support FPGA-based middlebox configuration, the FPGA can be either partially or completely reconfigured. Both approaches are supported in our system. Whole-chip FPGA programming on the DE5 is initiated by a trigger signal sent from the FPGA to the MAX II CPLD used for configuration loading. Multiple configurations for the FPGA are available in on-board flash memory. The start address of the configuration image is specified in flash and used by the CPLD to initiate configuration image loading into the FPGA. Before reconfiguration starts, the Nios II can overwrite this start address so that the next FPGA image can be changed. Once the new FPGA image has been loaded, the TCP connection between the coordinator and the interface implemented in the Nios II is reinitialized. A more effective approach for middlebox configuration is to swap one of the middlebox packet processor modules in Figure 3.5, which is known as FPGA partial reconfiguration. We will discuss this approach in detail in Chapter 4.

SDN switch configuration - Once the processor-based or FPGA-based middlebox has been properly configured, the coordinator is notified and the configuration manager and SDN switch controller work together to send flow modification messages to the SDN to modify source and destination port entries in the switch flow table. These updates allow packet traffic to (or away from) the newly configured (or stopped) middleboxes. Level 2 routing is used for packet transfer in this case.

In our system, traffic previously sent to a processor-based middlebox can be rerouted to an FPGA-based middlebox due to an increase in required VNF throughput. This action includes the activation of the FPGA VNF, reprogramming of the SDN switch followed by processor-based VNF deactivation. The configuration manager sends messages to the middleboxes to replace their current functions with alternative configurations and to the SDN switch to reroute affected traffic (Figure 3.8). A detailed example using middlebox functionality migration is described in Section 3.6.

3.3 Scalability Considerations - Global State Table

3.3.1 Background

In a typical NFV system, various VNFs are deployed in networks for a variety of purposes, e.g. monitoring, security, or performance optimization. These network functions record the statistics or information about packets or flows during packet processing. For the case that a network function is deployed at a single commodity or specialized hardware, the states are maintained by the middlebox locally. However, in a distributed middlebox environment, multiple middleboxes combine together to work for the same network function. If the network function only requires per-flow states for packet processing, the distributed middleboxes are able to independently process their own flows and do not impact each other. But if the network function requires multi-flow states that are related to the packet processing of multiple flows, it is essential to share the multi-flow states among the distributed middleboxes.

The multi-flow states are global states. In our system, all global states are stored in the global state table maintained by the state manager in coordinator. All middleboxes are able to access the state table through the APIs that are provided by the state proxy in the middleboxes. The design and implementation of the global state table in the coordinator are based on the work of Shao et al. [69]. Two types of state operations (i.e. trigger and retrieval) are supported which have been introduced in detail in Section 3.2.2. With the help of the global state table, our NFV system can be scaled up easily. As a new hardware resource is added to the system, it is able to use the APIs to fetch multi-flow states from the state table as it needs them. In addition, our system also benefits from the global state table to make itself more robust in the face of network environment changes. After resource reallocation in response to changing threats or monitoring goals, network functions can retrieve the flow states from the state table and quickly restore their functionality.

3.3.2 Global State Implementation

The global states are configured in the coordinator in response to the *state configuration* messages sent by state proxies. The state configuration message has two arguments: state name and state type. Different middleboxes can use the same state name to access the same state table. The state type value determines what kind of state operations can be performed. Three primitive states are supported: integer, string and set.

- StateConfig (stName, stType)

The state table is abstracted as a data structure that supports the following three state operations. Where "Key" is the state key while "State" is the update state that is generated during the packet processing. "Op" indicates the operation identifier.

- void Check (Key, Op)
- state Update (Key, Op, State)
- state Retrieve (Key, Op)

After setting up the global state table, middleboxes can register events and set the event trigger conditions in the coordinator via the *event configuration* message. Each event has a unique name. The state name in the message indicates which state table this event is associated with. The trigger conditions, i.e. threshold and period, determine when and how often the event would happen. If the event is triggered, a corresponding operation will be executed.

- EventConfig (evName, stName, Threshold, Period, Op)

3.3.3 Interactions with Global State Table

In our system, to enable the interactions between middleboxes and the coordinator, we adopt TCP sockets for reliable communication. As the communication

between the coordinator and middleboxes is so frequent, a persistent connection with each middlebox is maintained by the coordinator.

During packet processing, four messages are used for interactions as follows.

- StateUpdate (stName, Key, Op, State)
- StateRetrieve (stName, Key, Op)
- ReplyState (stName, Key, State)
- EventTrigger (evName, stName, Key, State)

The first two messages are sent by the state proxies to the coordinator. These two messages use a key to update the state value of a specific entry in the global state table or fetch the state value of the entry from the table. The type of state value depends on the semantics of network functions. It is configured by the state configuration message. According to the type of state value, different operations can be executed on it. For example, if the state is an integer, the value of it can be increased or decreased by the *state update* message. The operation in a *state retrieve* message can have different meanings. In a NAT application case, the state proxy can send a retrieve message to require the coordinator to create a new translation, save it and return it, or search in the state table for an existing translation and send it back.

The *reply state* message is sent by the coordinator in response to the state-retrieve message. It includes the state value that a state proxy is querying. The *event trigger* message is also sent by the coordinator. When the trigger condition of the event is met, the event-trigger message is sent to middleboxes that use the same state name to share a common global state table. Middleboxes can get to know the current state value after receiving the event-trigger message and will take corresponding actions.

3.4 FPGA-based Middlebox Applications

For experimentation, four FPGA-based library modules which meet the requirements of the previous section were created and tested.

3.4.1 NAT Implementation

Network address translation (NAT) is a method of remapping one IP address space (IP address, port) into another by modifying network address information in the IP header of packets while they are in transit across a traffic routing device. The original usage of this technique was as a shortcut to avoid the need to readdress every host when a network was moved. In the face of IPv4 address exhaustion, NAT has become a popular and essential tool in conserving global address space. It has also been used as a mechanism for the transition between IPv4 and IPv6 addresses [74].

In our implementation, all translations are determined at the coordinator and stored in the coordinator's global state memory. Translation information is returned to a requesting middlebox via a *reply state* message following a *state fetch* message. The middlebox packet processor was implemented in FPGA logic while the state proxy was implemented using a Nios II processor.

The blocks used in the FPGA-based NAT application are shown in Figure 3.9. The interface signals on the left of the figure match the packet processor interface signals shown in Figure 3.5. The NAT module allows other packets to be forwarded while the middlebox waits for the NAT translation to arrive from the coordinator. As a result, packet buffering is needed. In our implementation, eight 8K entry $\times$ 69 bits buffers are used for packet sizes ranging from 64 to 1,500 bytes. A buffer index table, implemented as a hash table, is used to store the index of the buffers for specific flows. The extraction module extracts the source address, source port, destination address, destination port, and protocol information from the packet header to form a key. For each flow, the key is used as the input to the buffer index table and the local

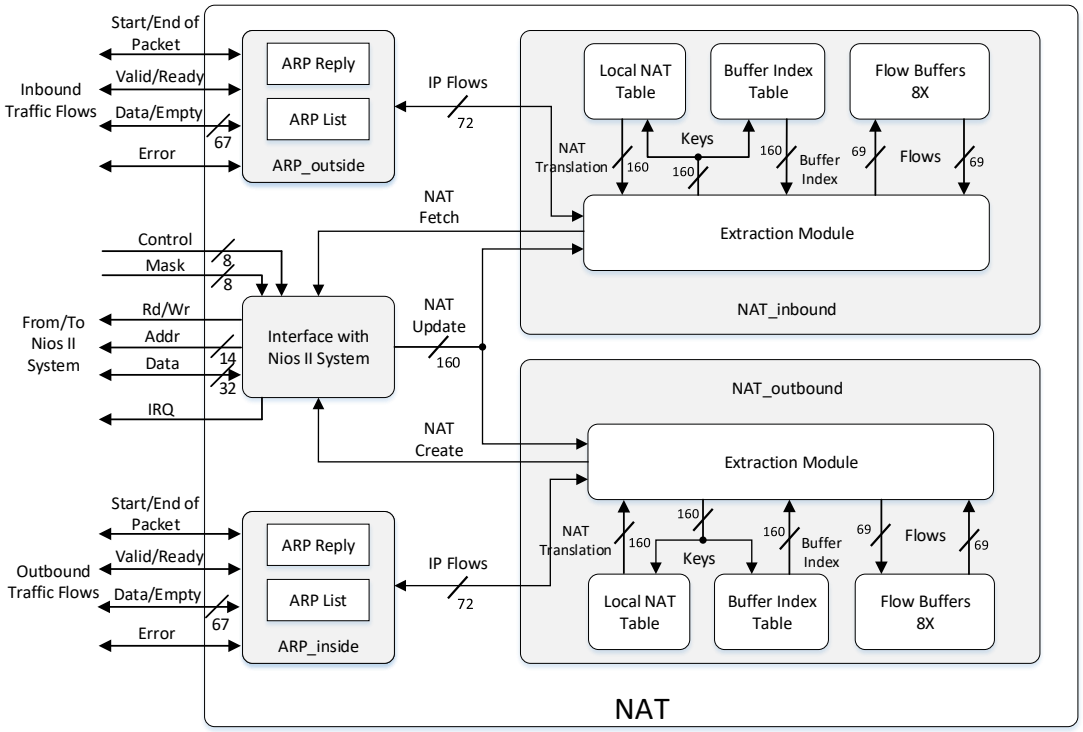

Figure 3.9: FPGA middlebox implementation of NAT application

NAT translation table (implemented as a hash table) of depth 4,096 entries. If the translation is not found in the local NAT table, a NAT state fetch for the coordinator is initiated by the state proxy. The round-trip time to fetch the translation from the coordinator is about 0.2 million seconds.

A NAT update message provides the translation which is stored in the local NAT table. Separate translation units are provided in the middlebox for inbound and outbound subnet traffic. There is an ARP module designed for each unit. The ARP module contains an ARP list (cache) and a reply module. The ARP list block allows for the conversion of IP addresses to physical addresses and updates the physical addresses for the output packets. The ARP reply block responses the ARP request by sending a ARP reply packet which contains the link layer address mapping.

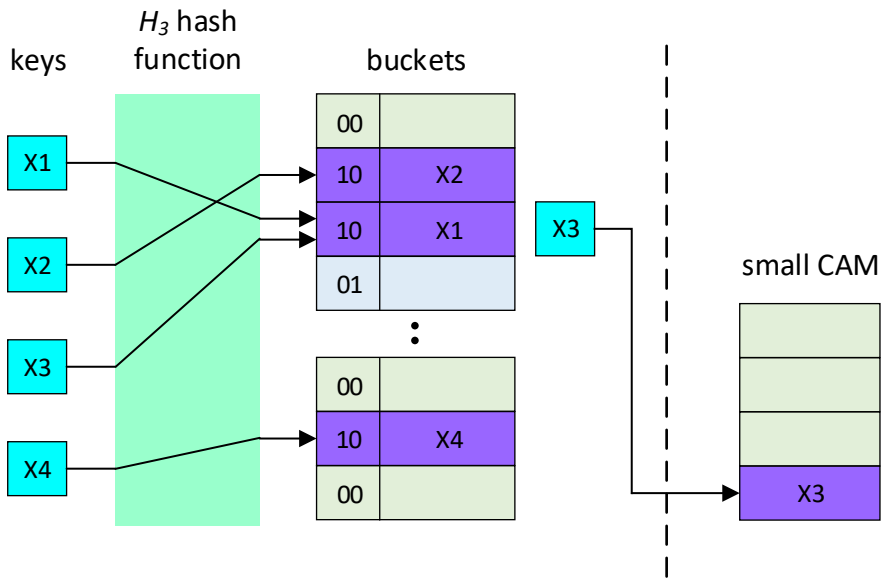

Figure 3.10: Improved hash table implementation using CAM

In order to get the local NAT translation and the flow buffer index quickly, we implemented these two tables as hash tables with a *H3* hash function [66]. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. The pros and cons of a hash table are obvious, its search speed is very fast ($O(1)$) but there may be hash collisions, which slow down the search and introduce uncertainty in search time. To alleviate this problem, we added a content-addressable memory (CAM) into the hash table module. As shown in Figure 3.10, whenever a collision happens, the new entry will be added into the CAM until the CAM is full, then the new entry will be added into the next empty slot of the hash table. We add two extra bits to the head of each entry and save them in the hash table. The first bit shows if the current slot is occupied and the second bit shows if the entry in the current slot has been deleted. Since the search speed of CAM is also $O(1)$, combining the hash table and CAM does not reduce the

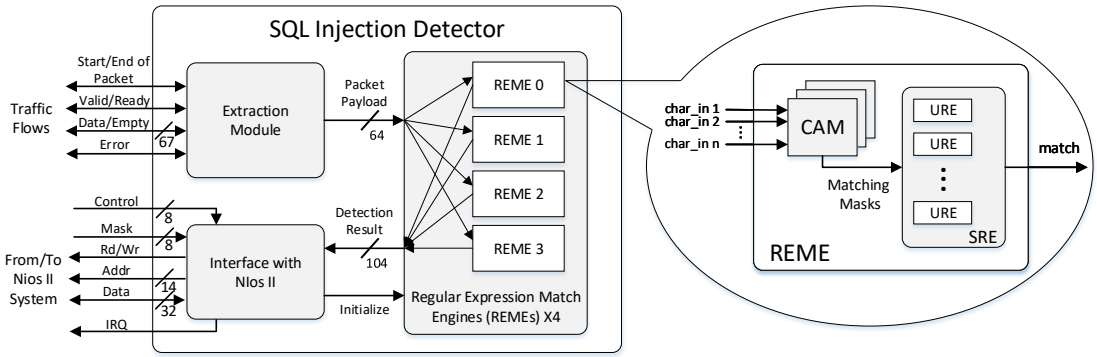

Figure 3.11: FPGA middlebox implementation of SQLi detection block

search speed, while significantly improving the collision problem at the expense of small additional memory resources.

The software version of the NAT middlebox implemented on a PC performs the same functions and uses the same message sizes. The state proxy is implemented as a separate VirtualBox module programmed with APIs.

3.4.2 SQL Injection Detection

The second function used to test our system was an SQLi detection block. Both FPGA and processor-based implementations of this application are supported. Processor implementations are based on Bro[1]. SQLi detection attempts to identify possible web-based attacks by examining packet payloads for known attack data. The SQLi implementation uses a regular expression matching engine (REME) to find keywords in the GET and POST request lines of an HTTP packet [88]. Figure 3.11 shows the block implementation of our SQLi dectection module on FPGA.

Regular expression matching (REM) is an important mechanism used in many popular intrusion detection software such as Bro, to perform deep packet inspection against potential threats. One regular expression is constructed with operators and

[1]https://www.zeek.org/

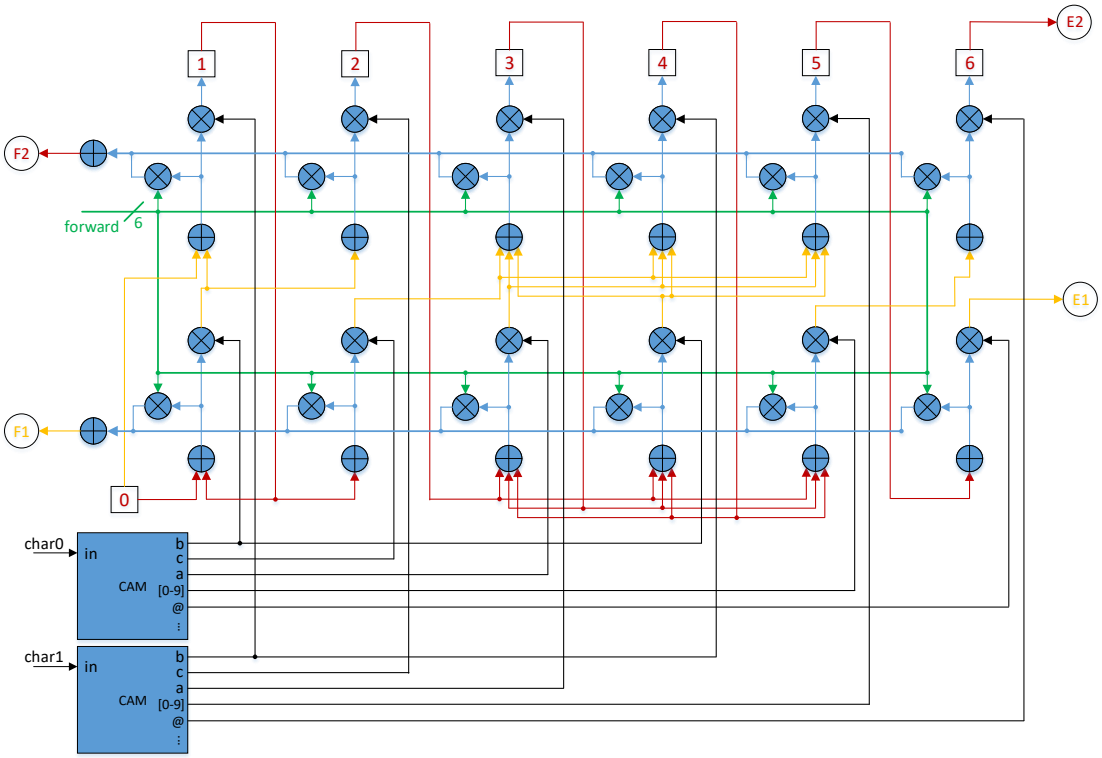

Figure 3.12: An example of the URE circuit design for the regular expression $b + c(a|b)*[0–9]@$

characters. There are three basic operators used in the regular expression: *concatenation* $(\cdot)$, *union* $(|)$, and *Kleene closure* $(*)$ [72]. Other common operators can be constructed by proper arrangements of the three basic operators. In our design, we implemented the REME on FPGA as a non-deterministic finite automaton (NFA) [18] based on a modified McNaughton-Yamada construction [88]. One REME can take at most 64 input characters. These characters are combined and arranged with the three basic operators to create a specific regular expression. The REME uses this initialized regular expression to inspect all input data.

As shown in Figure 3.11, a REME contains a CAM array and a list of unit regular expression (URE) blocks. All CAMs in the array store the same contents which are the characters used to build the regular expression. The CAM array helps to improve

the work efficiency of the REME by parallelly matching multiple input bytes with saved characters. In a URE block, a matching circuit is created for a limited length regular expression. Multiple URE blocks can be sequentially connected together in the REME module to accommodate longer regular expression matching. However, there should be no return path between connected two URE blocks. Figure 3.12 shows an example of the matching circuit for a regular expression $b + c(a|b)*[0–9]@$ in a URE block. It contains six stages each of which is used for a specific matching operation. This circuit is able to match two input bytes at the same time. A control signal named as *forward* is used to extract the matching result from any stage of this circuit.

In our system, TCPreplay[2] is used to send packets ranging in size from 54 to 1514 bytes through SQLi detectors via 10 Gbps ports at varying speeds. When a detection occurs, a 41-byte set of information is sent to the coordinator as a message. This information includes the packet source and destination. The coordinator then sends a 51-byte signature to a firewall on another middlebox which is either implemented in an FPGA or a VirtualBox VM. The firewall is located between the client and the switch input to the subnets. After activation by the coordinator, the firewall identifies packet headers with offending source and destination addresses and ports and drops them.

3.4.3 DDoS Implementation

An addition module used to test our system was a distributed denial of service (DDoS) block, based on an earlier design [23, 47]. During a DDoS attack, the attacker floods a victim's network with SYN packets without sending the corresponding ACK packets. Incoming packets which arrive at the middlebox are sampled and a counter (SYN_ACK_CNT) is used to keep track of unmatched SYN packets for up to 1,000

[2]https://tcpreplay.appneta.com/

destination addresses. The values of the *SYN_ACK_CNT* counters are periodically evaluated to identify deviations from expected values as determined by the mean and standard deviation of the counters. If the values vary beyond a variable threshold for a destination address, a possible DDoS attack is identified. This result triggers a message for the coordinator. The coordinator can identify messages from a number of middleboxes to identify if a pattern exists for a specific destination address. After activation by the coordinator, the software rate limiter identifies packets with offending SYN messages and limits their transmission.

3.4.4 Firewall Implementation

The final module used to test our system was a packet-based firewall blocker. A hardware hash table was implemented in the module to save packet blocking information. The firewall tracks in-transit packets and filters them by source and destination network addresses, protocol, and source and destination port numbers. When a packet matches a set of filtering rules stored in the firewall (the packet exists in the blocking list), it is dropped by the firewall. Otherwise it is allowed to pass. The coordinator sends messages to add new entries to the blocking list as needed.

3.5 Data Plane Traffic Management

The data plane traffic management is necessary to support middlebox functionality migration. For our system, the SDN controller in the coordinator was implemented using the RYU SDN framework[3] with OpenFlow 1.3 protocol used to communicate with the NETGEAR ProSafe M4300-8X8F 10 Gbps switch. By properly setting the flow table located in the SDN switch, the SDN controller can configure the switch to steer the traffic flows as needed.

[3]https://osrg.github.io/ryu/

The flow table in the SDN switch is used to define what *actions* should be applied to packets that enter this particular switch. These actions are saved as different entries in the flow table. To determine which packets a particular action should be applied to, the SDN controller needs to specify a series of *match criteria* for each action in the table. These match criteria can be defined over fields in the Ethernet header, IP header, TCP or UDP header, and potentially other header information. For example, we can define match criteria over source IP, destination IP, protocol, and input port fields. Actions can include dropping a packet, sending a packet out of a particular port, sending the packet to the SDN controller, modifying packet header fields or performing some other actions. The action and match criteria are the two main fields used to form a rule added into the flow table. In addition, a flow rule normally includes three other fields which are the *priority*, *hard timeout*, and *idle timeout* (Figure 3.13). If the incoming packets match multiple rules in the flow table, the action in the rule with the highest priority is executed. Two timeout fields are associated with how the SDN switch automatically removes the rules. The hard timeout field determines when the rule should be deleted after the rule is installed into the flow table. The idle timeout field indicates that if no packet matches a rule for a given amount of time, the rule should be removed.

The SDN controller we created supports three different operations to manage input traffic flows, *forwarding*, *copying*, and *splitting*. By default, the SDN switch works as a standard layer-2 switch which utilizes a MAC address to determine the port used to forward a frame. For manipulation functions (e.g. NAT), the SDN switch forwards the network traffic to middleboxes that perform specific functions. The packet header is updated in the middlebox and then forwarded to its original destination. The SDN controller adds a new rule with specific match criteria (i.e. input port, source IP, destination IP, protocol) and action (i.e. output port) fields to the flow table to achieve the forwarding operation. This rule has higher priority than

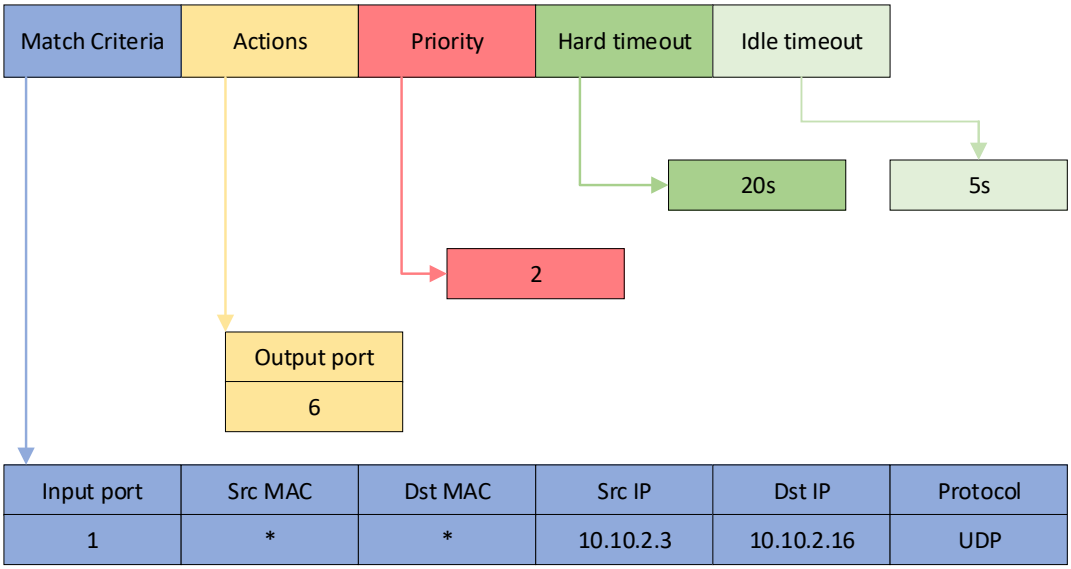

Figure 3.13: An example of a rule entry in the flow table

the default rule, so the incoming packets will follow this rule to be forwarded to the manipulation function instead of being sent directly to the output port. For inspection functions (e.g. SQLi, DDoS), the packets usually do not need to be sent out after they are examined by the functions, and multiple inspection functions may operate in parallel in different middleboxes to monitor the same network flows. Therefore, input packets to the SDN switch can be copied prior to transmission to inspection functions. Copying typically does not influence the original flow direction of the input packets. The SDN controller modifies the existing rule by adding a new output port to the action field to make a copy of an input flow.

When the traffic volume moves beyond the processing capability of a single middlebox, the network function may be spread across multiple middleboxes. The traffic is split by the switch and each middlebox processes a subset of flows. A weighted round-robin load balancer was implemented in the SDN controller to guide traffic flow splitting in the switch. As the number of middleboxes for the same network function

increases, the SDN controller adds new output channels to the load balancer. Each output channel corresponds to an output port on the SDN switch which connects to a middlebox. The round-robin load balancer forwards a network flow to the output channel in turn. The weight of an output channel is determined by the process capability of connected middlebox minus current traffic volume in the channel. The higher the weight, the larger the proportion of flows received by the channel. The SDN controller creates a rule for each flow and adds it to the flow table. The output channel is used as the action field for each rule.

3.6 Evaluation

We evaluate the performance of our system by comparing the throughput and latency of the middlebox applications implemented both on FPGAs and VMs. We also measure the resource usage of each function module on the Stratix V FPGA. In addition, we analyze the scalability of the system and use a stress test to show the ability of the coordinator to manage millions of global states. In an initial experiment, we demonstrated the middlebox functionality migration in our system by leveraging the full FPGA reconfiguration technology. The following sections describe some of the techniques used to obtain experimental results.

3.6.1 Performance Test

The resource counts of the packet processor modules and the Qsys system are shown in Table 3.1. Comparing the SQLi attack detector, the DDoS attack detector, and the firewall module, the former requires more logic resources and defines the region size for a packet processor on the FPGA chip. All circuits operate at 156.25 MHz, a clock speed that is derived from the data transfer rate of 10 GHz / 64. Network and memory interface circuits consume a non-trivial amount of logic resources since data flow control is needed. The performance benefits of using the FPGA circuits versus

Table 3.1: Resource usage for NFV library cores targeted to a Stratix V 5SGXEA7N

	LUTs	FFs	Block Mem bits
NAT	40,309	54,779	24,017,920
SQLi detector	25,708	15,172	2,755,072
DDoS detector	14,608	9,979	3,004,928
Firewall	10,571	12,609	3,490,560
Qsys system	31,008	34,911	3,174,907
↪ Nios II CPU	2,034	1,806	1,137,408
↪ Shared memory	3	0	1,572,864
↪ Memory interface	16,036	18,002	225,040
↪ Network interface	12,938	15,103	239,595
Available in FPGA	469,440	938,880	52,428,800

Table 3.2: Throughput and latency comparison of VM and FPGA module implementations without using DPDK

	Throughput (Gbps)		Latency (us)	
	VM	FPGA	VM	FPGA
NAT	0.52	8.48	1,009.00	1.34
SQLi	0.41	9.20	10.60	1.20
DDoS	0.44	9.28	5.04	1.20

VM implementations in a 10G network were assessed by measuring the throughput and latency of the same VNFs working on different infrastructures. The three network functions introduced in Section 3.4 were used. The test results are shown in Table 3.2. The dramatically reduced latency numbers and higher throughput for FPGA versus VM indicate the benefit of FPGA usage.

To consider accelerated CPU computation we use the Intel Data Plane Development Kit (DPDK), which gives the CPU low-level access to network interface card drivers, bypassing the traditional network stack. We implement the software version

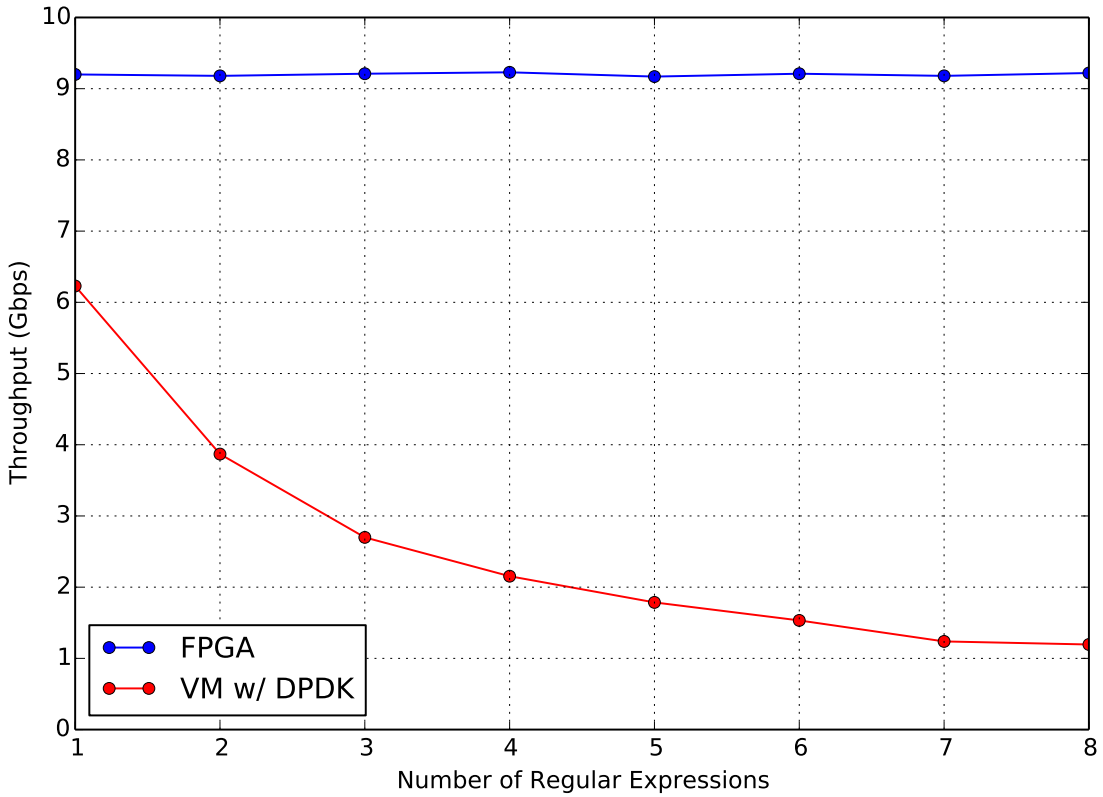

Figure 3.14: REME using DPDK: processing throughput versus number of regular expressions

of regular expression matching engine (REME) on Bro with DPDK, and compare it with the corresponding FPGA implementation. Figure 3.14 shows that by using DPDK, the throughput of software implementation is moderately less than the FPGA version for one REME. As the number of REMEs scales, the software throughput is attenuated by microprocessor performance limitations while the parallelism of the FPGA allows for relatively constant throughput.

3.6.2 Stress Test

For a distributed system, the state manager in the coordinator may manage millions of global states for a VNF instance. In a second experiment, the state manager was flooded with state requests at the maximum rate of the coordinator network

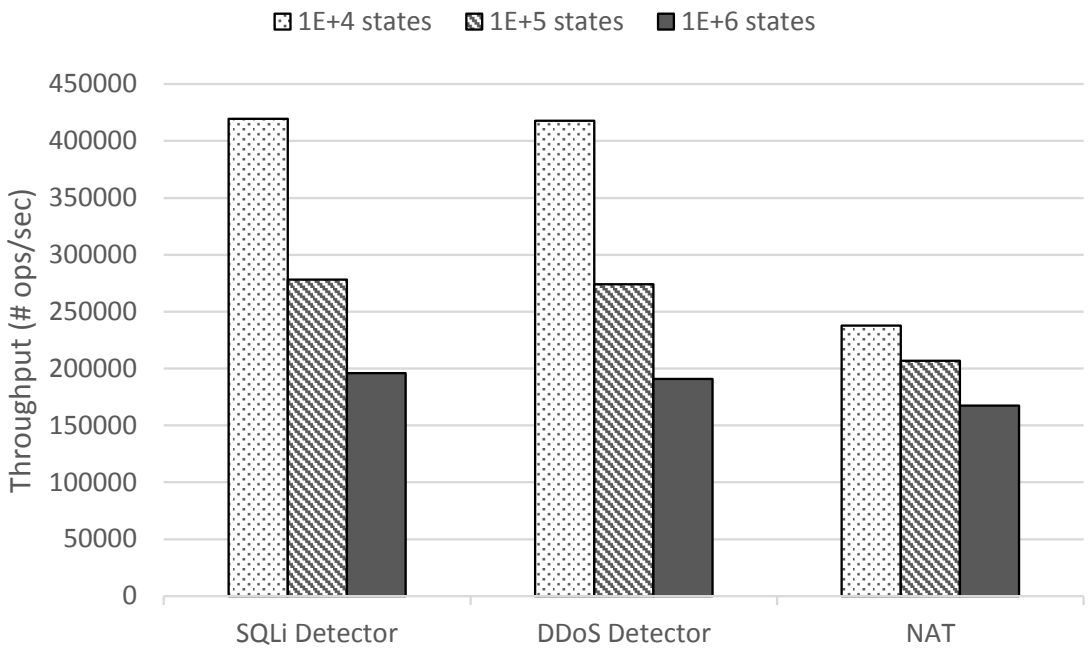

Figure 3.15: Results of coordinator stress test. For each test, requests are made to the coordinator at the fastest rate supported by the network interface

interface to test its processing capabilities. Figure 3.15 shows the throughput of the state manager portion of the coordinator for the three VNFs with the number of global states growing from ten thousand to one million. As the figure shows, the coordinator keeps a high processing speed of more than 100,000 operations per second for the three functions.

From the result, it is apparent that the coordinator supports similar throughput for SQLi states (matched patterns) and DDoS states (destination address count mismatches), which are both inspection functions. In both cases, the global state table is supported with hash tables in the coordinator. As the number of states increases, more hash collisions occur in the global state table during state insertion or update which affects performance. For the NAT application, the coordinator generates new translations when misses occur in the global state table, decreasing coordinator throughput.

Based on the results in Figure 3.15, the operation processing rate of the coordinator scales to support tens of high-throughput FPGA middleboxes. If a middlebox has a 10 Gbps input rate, at most 23,148,148 and 825,627 packets per second would be processsed for 54 and 1,514 byte packets, respectively. However, since SQLi, DDoS, and NAT state requests are only generated at most once per thousands of packets, on average, state processing by the coordinator is scalable.

3.6.3 Scalability Test

In a third experiment, the ability of the FPGA circuits and virtual machine-based middleboxes to process packets for a scaled set of middleboxes was tested. To get a comparable test result and exhibit the scalability clearly, this experiment was performed using the 10G network. To evaluate scalability, system throughput was measured using an increasingly large set of hardware and software middleboxes and examining overall processing throughput using the SQLi application. Software versions of SQLi were implemented using Bro software. Three workstations sliced into between one and ten VirtualBox middleboxes each were used to implement software SQLi. Two DE5 boards implemented FPGA versions (two SQLi cores per FPGA). All middleboxes were connected to the coordinator via TCP connections. A separate PC was used to generate packets for the subnetwork using TCPreplay and to retrieve packets. An SDN switch under coordinator control was used to steer generated packets to middleboxes. Packets used for testing range in size from 54 to 1514 bytes. Figure 3.16 shows the scalability of our heterogeneous network system for between 1 and 34 middleboxes for the SQLi application. The first four CRs used in the system are FPGA-based (two cores each), hence the higher slope of throughput on the left side of the graph.

As middleboxes are scaled up to a total of 16 (12 VM and 4 FPGA), system performance versus the ideal case initially remains nearly identical indicating the

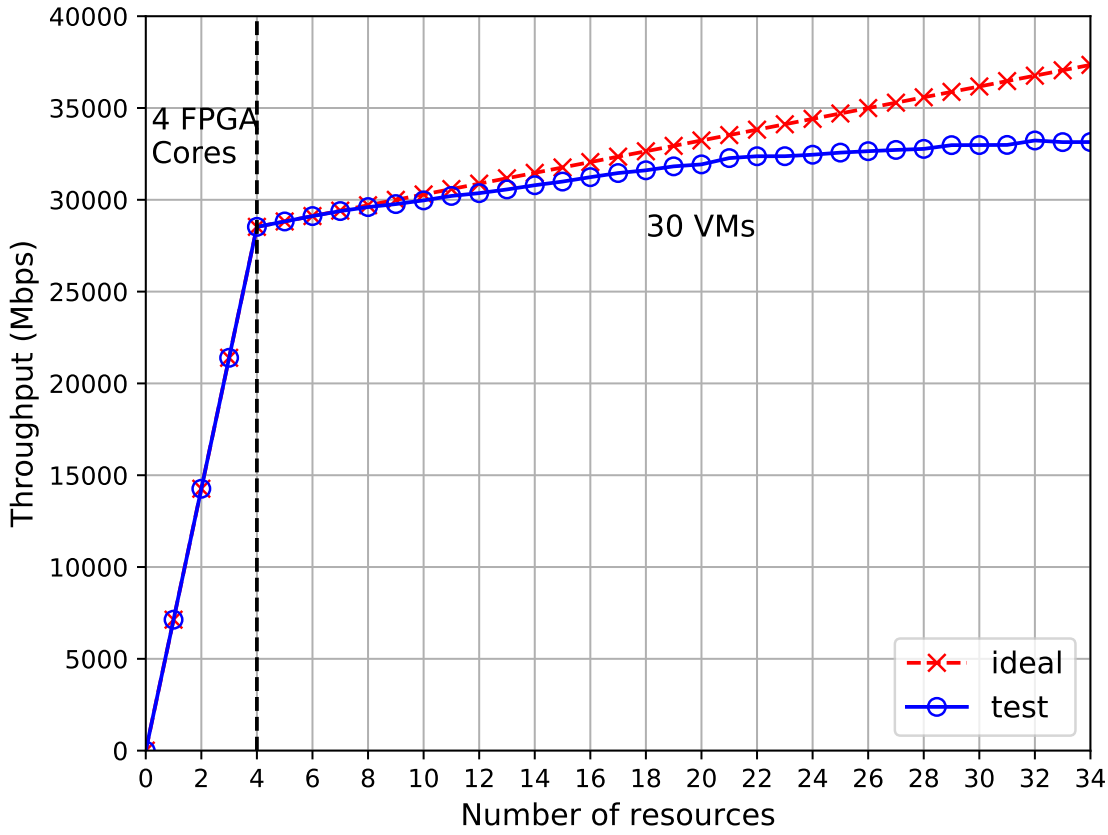

Figure 3.16: Scalability of SQLi implemented with up to 2 FPGAs (2 cores each) and 3 servers (30 virtual machines)

capability of the state manager in the coordinator to keep up with simultaneous state requests from both FPGA and VM middleboxes. A flattening of the curve is observed at 22 middleboxes (18 VM and 4 FPGA). At this point, all cores in each of the three workstations hosting the VM middleboxes are assigned dedicated processes. The addition of middleboxes causes processing limitations beyond this point, leading to reduced throughput scaling.

3.6.4 Reconfiguration Test

The use of NFV requires the ability to dynamically reconfigure middleboxes in response to changing networking needs. For example, it may be necessary to periodically change middlebox functionality between DDoS and SQLi operations. We per-

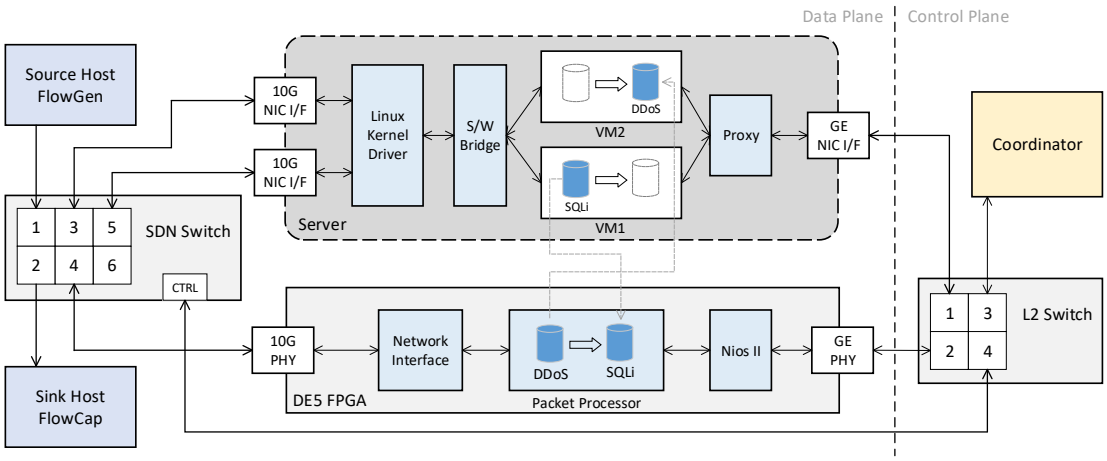

Figure 3.17: Reconfiguration test environment with two VM and one FPGA (single core) middleboxes

formed an experiment with transient variations in the incoming workloads for DDoS and SQLi. Initially, FPGA hardware is used to detect DDoS attacks and software is used to detect SQLi attacks. Although a traffic increase targeted to the SQLi middlebox does not necessarily imply an attack, a microprocessor cannot perform SQLi detection effectively due to throughput limitations. In this case, the microprocessor sends a message to the coordinator indicating the desire for an FPGA middlebox update to support SQLi. The coordinator can decide to swap FPGA NFV functions from DDoS to SQLi attack detection during this period of high SQLi traffic if DDoS processing is limited at the moment.

In a final experiment we determined how quickly a packet processing function can be replaced within an FPGA by the configuration manager in a system with two VM and one FPGA middleboxes. The test environment is shown in Figure 3.17. The software-based flow generator on source host generates the SQLi and DDoS attacking flows, and sends them to the sink host via the SDN switch. The SDN switch is controlled by the coordinator to copy and forward flows to processor- and FPGA-based middleboxes. The copy flows will not be sent back to the network after being pro-

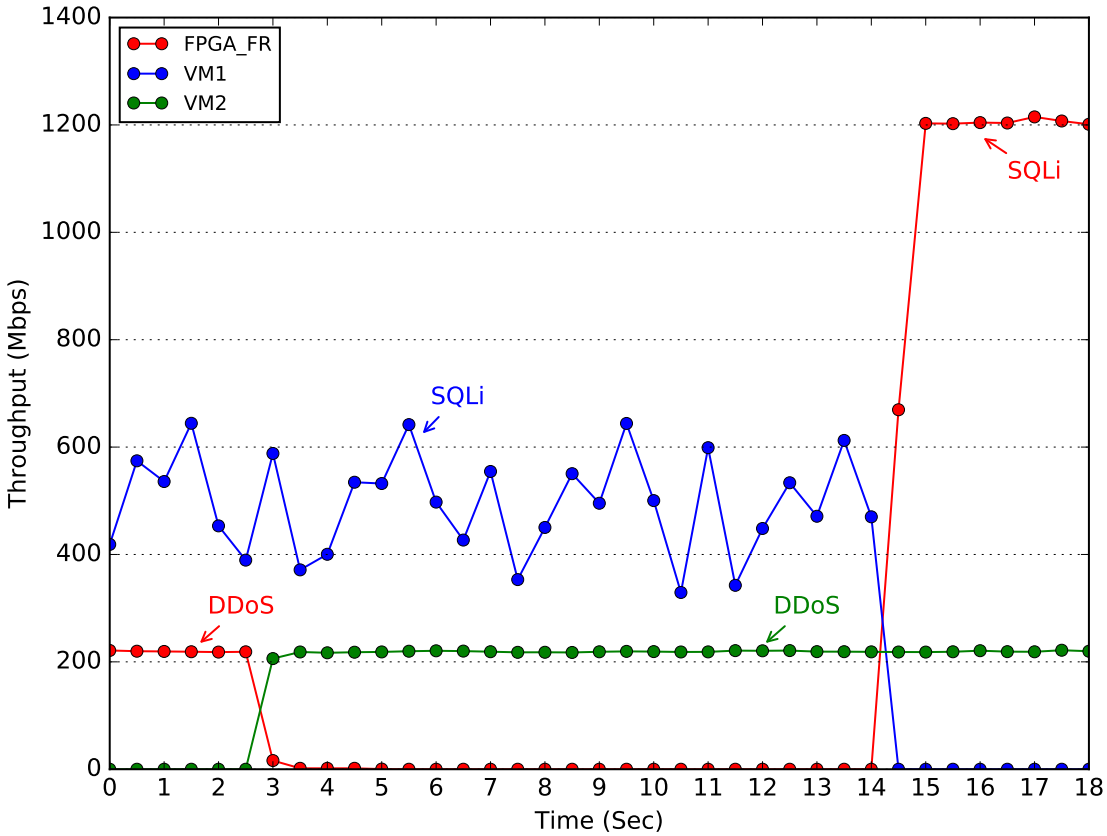

Figure 3.18: Performance of system resources during full FPGA reconfiguration

cessed by middleboxes. The coordinator connects with the proxy on the server and the Nios II microprocessor via a general L2 switch. It monitors the traffic volume changes on both FPGA- and VM-based packet processors and dynamically triggers the reconfiguration of the packet processors in response to the situation. The steps needed to perform the reconfiguration are described in Section 3.2.3. As seen in Figure 3.18, initially a DDoS detector is implemented in the FPGA and an SQLi detector is implemented in VM1. When input traffic rate into VM1 consistently exceeds 410 Mbps (the VM throughput limit in Table 3.2), VM1 notifies the configuration manager in the coordinator. Since the DDoS detector throughput is less than 440 Mbps and can be handled in software, its function is migrated to VM2 and the FPGA middlebox is reconfigured to support SQLi detection.

Figures 3.18 shows the delay associated with the redirection of the SQLi traffic from VM1 to the FPGA and FPGA reconfiguration using full device configuration (FPGA_FR). Results in the graph was generated from experimentation with FPGA and VM middleboxes in the lab. The full FPGA reconfiguration process requires about 12 seconds. This delay includes the time needed to remap traffic using the SDN switch, reconfigure the FPGA, reboot the Nios II, and reinitiate the connection between the Nios II and the coordinator. The size of the entire bitstream is 31.3 MB for both SQLi and DDoS.

3.7 Conclusion

In this chapter, a new heterogeneous hardware-software approach to NFV construction is demonstrated that provides scalability and programmability. The platform leverages both FPGAs and microprocessors to support a range of user defined network functions with a common interface. As the number of required functions and their characteristics change, FPGA logic is automatically reconfigured under system-wide control. To evaluate our approach, a series of software tools and NFV modules have been implemented. The scalability and hardware reconfigurability of the hybrid system is demonstrated for known network attacks.

CHAPTER 4

PERFORMANCE-AWARE VNF DEPLOYMENT WITH PARTIAL RECONFIGURATION

4.1 Introduction

FPGA-based middleboxes introduced in this dissertation support strong resource isolation. For example, the architecture shown in Figure 3.5 reserves separate logic elements for each packet processor. The functionality of the packet processors should be dynamically reconfigurable based on customer requirements and changes in the network environment. In Chapter 3, we described how to update the network function of an FPGA middlebox by reconfiguring entire FPGA circuits (full reconfiguration). However, if multiple packet processors are implemented on the same FPGA, all network functions running on these packet processors will be disrupted during the process of FPGA full reconfiguration, which means that these packet processors do not have logical isolation. Full reconfiguration has an additional drawback that the soft microprocessor on the FPGA needs to be resynchronized with the *coordinator* after full reconfiguration which significantly increases the time spent on the function migration process.

The first half of this chapter exploits partial reconfiguration to address the isolation and reconfiguration time overhead issues associated with full reconfiguration. Partial reconfiguration allows selective regions of the FPGA to be reconfigured while the other parts of the device are still in operation. For example, one packet processor in an FPGA middlebox (Figure 3.5) can be partially reconfigured while the other packet processors, as well as the Nios II microprocessor, are still functional. Partial

reconfiguration makes the allocation of resources in our system more flexible since multiple packet processors on a single FPGA middlebox can be used for different network functions and they can be reconfigured independently.

Our CoNFV system is enhanced by a scheduling and allocation algorithm that automatically considers the performance capabilities of the target NFV resource versus the requested function. As the functional needs of the network change based on link capacity or state-based triggers, a global coordinator takes advantage of the algorithm to rebalance the allocation of VNFs via the creation of VM threads and the dynamic reconfiguration of FPGA modules. In the second half of this chapter, we introduce our scheduling and allocation algorithm aimed at two scenarios, including relatively slow offline initialization where could happen only occasionally, and fast online VNF deployment, which could be triggered frequently. To evaluate the algorithm, we designed and performed an experiment in the lab to show how the coordinator reallocates the NFV resources for two network functions (i.e. SQLi and DDoS detectors) when network traffic volumes change over time.

The remainder of this chapter is structured as follows. Section 4.2 presents the details of the application of FPGA partial reconfiguration in our CoNFV system. Next, the Section 4.3 outlines our allocation and scheduling algorithms. The experimental methodology is detailed in Section 4.4 and experimental results are discussed in Section 4.5.

4.2 The Application of Partial Reconfiguration in CoNFV

In Chapter 3, we have demonstrated that our CoNFV system supports dynamic reconfiguration of VNFs on heterogeneous middleboxes by leveraging either software thread activation/deactivation on VM or hardware reconfiguration for entire FPGA circuits. A significant research contribution of this chapter is to take advantage of the FPGA partial reconfiguration technique to speed up the dynamic reconfiguration

of FPGA packet processors. To achieve this, we build upon our network function virtualization platform (CoNFV) presented in Chapter 3.

The FPGA-based middleboxes in the system are implemented on a Intel Stratix V FPGA which is partially reconfigurable. As shown in Figure 3.5, multiple packet processors are able to be implemented on one FPGA-based middlebox. These packet processors can work as the hardware carriers for the same or different network functions. To support packet processor isolation and facilitate partial reconfiguration, the FPGA is divided into static and partial reconfiguration (PR) regions. The static region holds the modules that are shared across multiple packet processors. The main module is an on-chip system built with the Intel Qsys tool. It includes the network interface (NI) modules and a Nios II-based SoC. The program running on Nios II makes it working as the configuration proxy and state proxy on an FPGA-based middlebox. In addition to the Qsys on-chip system, the static region also includes other modules, like a digital clock manager (DCM) module, a system reset module, and the PR control and CRC blocks.

Isolated features of packet processors are implemented in PR regions. Specific functions in these regions are determined by the network functions the designer decides to implement. For example, A SQLi detector includes an packet header extraction module, an interface module, and multiple regular expression matching engines (REMEs) (Figure 3.11). The fixed interfaces shown in Figure 3.5 guarantee an effective boundary for partial FPGA reconfiguration of packet processor functionality. For experiment purpose, the SQLi detection function, DDoS detection function, and packet firewall were chosen to be implemented in the PR regions. As shown in Table 3.1, the SQLi attack detector requires more logic resources compared to the DDoS attack detector and firewall, and defines the region size for partial reconfiguration. To configure the PR process on Stratix V FPGA, we used the Altera PR IP core which automatically instantiates the Stratix V PR control block and the Stratix V CRC

block in the design. Specific details of PR process and partial bitstream generation are described in the following two subsections.

4.2.1 Partial Reconfiguration Process

Partial reconfiguration is based on the *revision* feature in the Intel Quartus Prime software [5]. The initial design is the base revision, where the designer defines the boundaries of the static region and partial reconfiguration (PR) regions on the FPGA. From the base revision, the designer creates multiple revisions, which contain the static region and describe the differences in the configurable regions. In order to create the PR regions, the designer must organize the design into logical and physical partitions for synthesis and fitting. A PR region can have multiple implementations. Each implementation is called a *persona*. Partial reconfiguration uses personas to pass the logic, which implements a specific set of functions to reconfigure a PR region of the FPGA, from one revision to another. In contrast to a PR region, a static region has a single implementation or persona.

Our FPGA-based middlebox supports the creation of multiple packet processors on a single FPGA and the instantiation of network functions on them. Each packet processor works as a PR region which allows its logic to be dynamically reconfigured to different network function personas. In our current design, we create two packet processors on the FPGA. Each packet processor has sufficient hardware resources to support the instantiation of a SQLi attack detector, a DDoS attack detector or a firewall. Our partial reconfiguration approach requires the definition of a partial reconfiguration boundary that consists of the 207 interface signals which could be found on all function modules. All the nets between the static and reconfigurable regions with the exception of clock and reset signals are connected through the boundary interface. The clock to the PR regions is fed from global clock buffers in the static region, while the reset signal is generated from the system reset module which is also

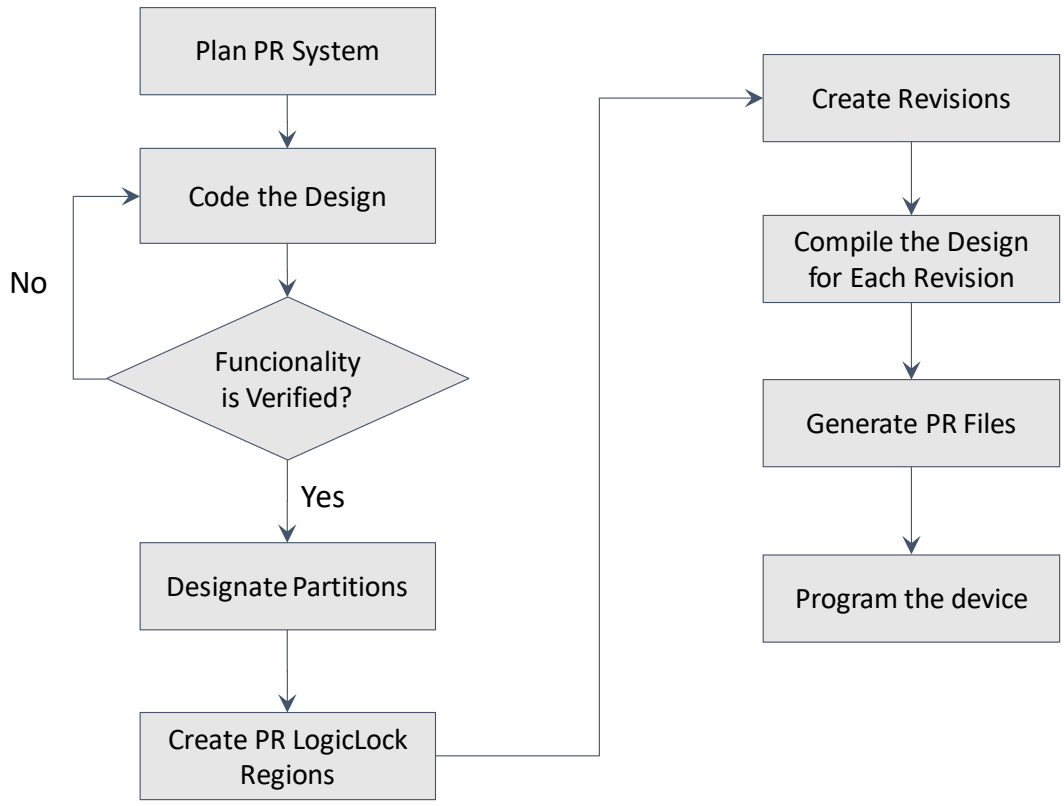

Figure 4.1: Partial Reconfiguration Design Flow

implemented in the static region. All the boundary interface signals are driven to a known value during partial reconfiguration.

We choose the design revision which has two SQLi personas implemented on both packet processors as the base revision. A typical partial reconfiguration design flow is shown in Figure 4.1. After finishing coding the base revision design using HDL, we need to set up the design logic for partitioning, and determine placement assignments to create a floorplan. We set up two LogicLock regions [5] on the Stratix V FPGA layout using the Chip Planner tool integrated with the Intel Quartus Prime and assign packet processor partitions to these physical placement constrained regions. Then we need to create two other revisions. We implement two DDoS personas in PR regions in one revision and place two firewalls at PR regions in the other. In order to get the

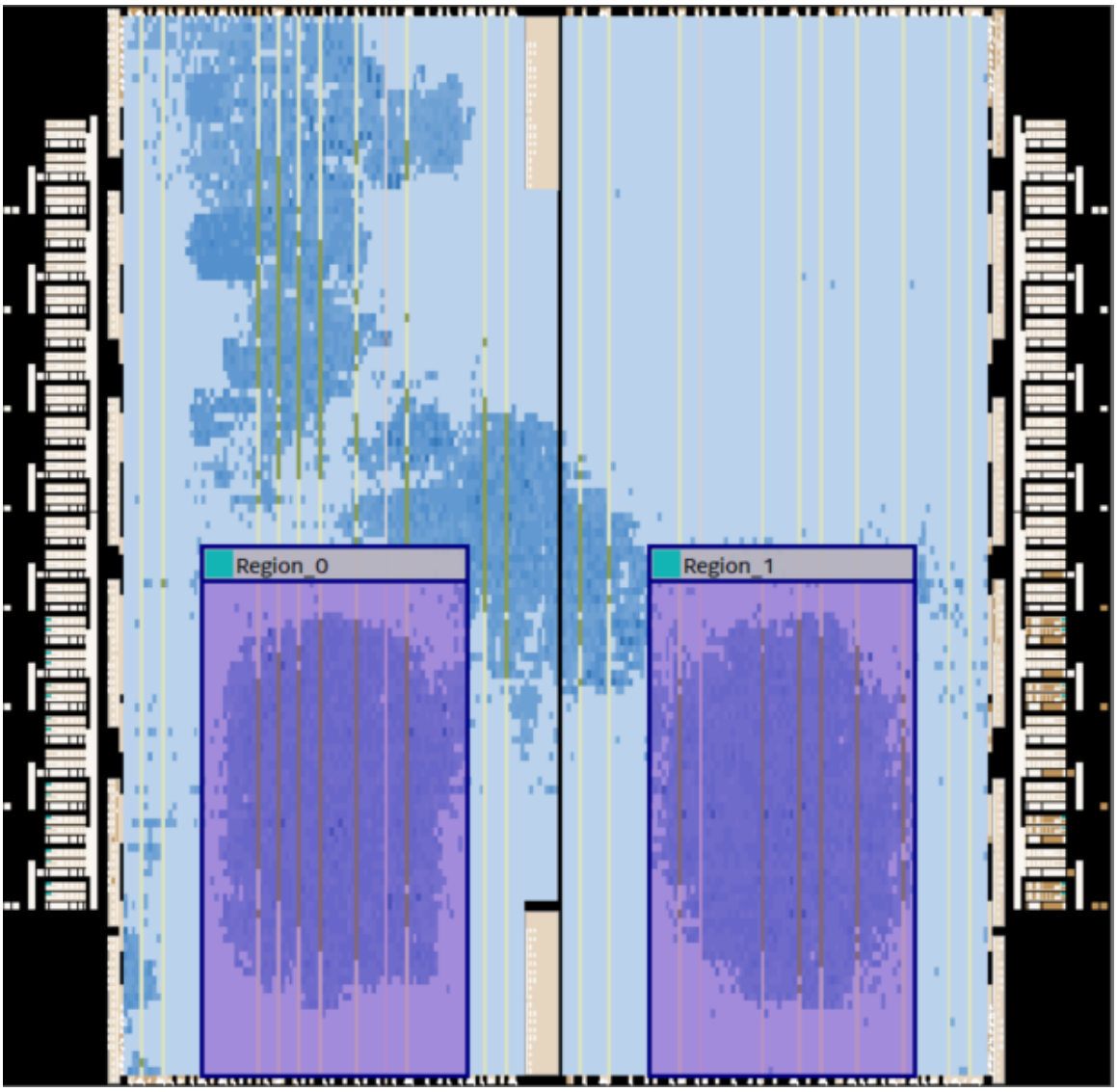

Figure 4.2: Layout of static and partial reconfiguration regions for FPGA-based middlebox on Stratix V

reconfiguration files for all personas, we need to compile the design for each revision. After synthesis, place and route steps, we get the layout of a Stratix V device with two packet processors located on the underside of the static region which is illustrated in Figure 4.2. Finally, the static and partially-reconfiguration designs are assembled and the respective bitstreams are generated.

The partial reconfiguration for an Intel FPGA device can be used in the SCRUB mode or the AND/OR mode [5]. When a designer implements a design on an Intel FPGA device, the design implementation is controlled by the bits stored in configuration RAM (CRAM) inside the FPGA. The SCRUB mode overwrites the CRAM bits corresponding to a PR region with new data regardless of what was previously contained in the region. The AND/OR mode uses a two-pass method to reconfigure a PR region. In the first pass, all the CRAM bits corresponding to the PR region are ANDed with '0's while those outside the PR region are ANDed with '1's. In the second pass, new data is ORed with the current value of 0 inside the PR region, and in the static region, the bits are ORed with '0's so they remain unchanged. Due to this working mechanism, the designer can have two PR regions that have a vertically overlapping column in the device by using AND/OR mode. However, the programming file size of a PR region using the AND/OR mode could be twice the programming file size of the same PR region using SCRUB mode, which may lead to the doubling of the reconfiguration time. In our design, we use the SCRUB mode to partially reconfigure the functionality of the packet processors using comparatively small size partial bitstream files.

4.2.2 Partial Bitstream Generation

Partial FPGA reconfiguration requires a priori generation of partial bitstreams for all personas. In our design, the partial bitstreams of six personas are generated, which are used to configure the logic of two PR regions. Each PR region supports the implementation of three different network functions (i.e. SQLi, DDoS and firewall). We use three revisions to manage our partially reconfigurable design. When these individual revisions are compiled in the Quartus Prime software, the assembler produces an SRAM Object File (.sof) and two Masked SRAM Object Files (.msf) for each revision. The .sof file has the information on how to configure the static region

as well as the corresponding PR regions. The .msf files are generated specifically for partial reconfiguration, one for each PR persona. The .msf file is used to mask out the static region so that the bitstream can be computed for the PR region.

After getting the .sof files and .msf files for all revisions, the next step is to convert them to the Partial-Masked SRAM Object Files (.pmsf) by using the Quartus Convert Programming File tool. The .msf file helps determine the PR region from the associated .sof file during the PR bitstream computation. Once all the .pmsf files are created, the PR bitstreams are processed using the Convert Programming File tool to produce the raw binary .rbf files for reconfiguration, one for each persona. The size of final partial bitstreams (.rbf files) for SQLi, DDoS, and firewall is 12.5 MB each. Compared to the size of the entire bitstream (.sof file) which is 31.3 MB, the file size of partial bitstream is less than half the size.

Partial bitstreams are converted to S-record files, which are suitable for use by the flash programmer, by running Quartus bin2flash command. We store the S-record files in different sections in the on-board flash memory. During partial reconfiguration, the Nios II processor reads the required configuration from a section of the flash memory and sends it to the PR IP core, which then assigns the configuration to the associated packet processor region to dynamically reconfigure its functionality. The configurations we generated for our experiments are shown in Table 4.1. Any configuration can fit within a single PR region.

4.2.3 Accelerating Partial Reconfiguration

Fetching a partial bitstream from flash memory using a Nios II microprocessor and triggering the reconfiguration of a PR region takes nearly 2 seconds. There are two reasons for this delay. First, the data transfer rate between the Nios II and flash interface controller is limited due to the low processing speed of the microprocessor. Second, the serial nature of Nios II forces the partial bitstream to be temporarily

Table 4.1: Experimental configurations

Configurations	Description	ALMs	M20Ks
I	SQLi detector for PR 0	18,969/36,480	202/384
II	SQLi detector for PR 1	19,612/36,480	202/384
III	DDoS detector for PR 0	11,795/36,480	217/384
IV	DDoS detector for PR 1	11,908/36,480	217/384
V	Firewall for PR 0	10,409/36,480	262/384
VI	Firewall for PR 1	10,187/36,480	262/384

stored in the SDRAM connected to the processor after it is read from flash memory. The Nios II then forwards this bitstream to the PR IP core.

In order to accelerate the partial reconfiguration, we created an FPGA module in Verilog. This module takes the place of the Nios II microprocessor in passing the partial bitstream from the flash memory to the PR IP core. After receiving the start address of the target partial bitstream stored in the flash memory, a finite state machine (FSM) in the module is triggered to read the data from the flash and send the bitstream data to PR core. Partial reconfiguration is accelerated by 5× versus the NIOS II approach.

4.3 Performance-Aware VNF Deployment

Due to state sharing, CoNFV is capable of scaling capacity or migrating VNF instances as performance requirements change within the hybrid network middlebox infrastructure. To support this feature, a performance-aware VNF deployment algorithm has been designed into CoNFV to satisfy both functional and performance requirements from customers and dynamically-changing online traffic volumes. Since a VNF can be performed by multiple heterogeneous compute resources (e.g. FPGA or microprocessor) in the network, an algorithmic approach is needed to assign VNFs to computation resources (CRs). In our system, these regions can be either virtual

machines executed by general-purpose processors or FPGA-based packet processors. The VNF instantiation must satisfy performance constraints in terms of latency, throughput, and compute capacity requirements, ideally at minimal cost.

4.3.1 Performance and Resource Model for CRs

To support our VNF deployment algorithm, a resource model for a selected VNF and a CR operating as a specific VNF has been created. The model is based on performance and capacity parameters. The model *capacity* indicates the available computational capability in terms of processing functions. For example, for SQLi injection detection, a regular expression matching engine (REME), represents a unit of computation. For hardware CRs, the number of parallel REMEs represents the compute capacity of the CR. *Latency* and *throughput* represent the processing latency and throughput required by a VNF or available from a VNF implementation. Throughput directly affects the achievable input and output data rate for a VNF implementation. The requested input rate and achieved output rate for an implementation are measured in real time and periodically collected by the coordinator to guide the online resource deployment.

4.3.2 Performance-Aware VNF Allocation

For VNF allocation, the resource allocator considers VNF performance requirements such as latency, throughput and compute capacity. Although some VNFs may be constrained to a software-only implementation, we consider here that three different implementations of the same VNF (one VM, multiple VMs, and FPGA) are available. Given the breadth of possible resource choices and VNF implementations, the initial and dynamic assignment of VNFs to CRs is a significant issue. In the following subsections, VNF allocation algorithms that can assign VNFs to both hardware and software CRs are described. A goal of the algorithm is to assign VNFs to resources that best match the required latency, throughput, and compute capacity constraints.

Table 4.2: Notations used in VNF deployment algorithm

R	set of computation resources (CRs)
M	set of customer-defined VNF instances
λ_m	latency requirement of instance $m \in M$
Λ_{rm}	processing latency of computation resource $r \in R$ working as VNF m
θ_m	throughput requirement of instance $m \in M$
Θ_{rm}	expected throughput of computation resource $r \in R$ working as VNF m
ϕ_m	required processing capacity of instance $m \in M$
Φ_{rm}	processing capacity of computation resource $r \in R$ working as VNF m
c_{rm}	overall cost of computation resource $r \in R$ working as VNF m
g_{rm}	Binary value to designate computation resource r was used for VNF m

Based on the results presented in Section 4.5, VM (software-based) VNF implementations, although more plentiful, generally provide inferior performance compared to FPGA (hardware-based) implementations.

Upon network system reset, an initial, offline assignment of VNFs to CRs is performed using a resource-based cost function. During system operation, CRs are deployed or redeployed based on changes in network traffic and triggers activated by global state within the coordinator. VNF migration between CR resources can be performed if the cost associated with system down time is considered in concert with the cost benefit of more balanced resource deployment. This second, on-line algorithm is activated repetitively in the deployed system.

4.3.3 Offline Initialization

During network reset, customer-required VNF instances must be allocated to available hardware and software CRs. The purpose of the offline deployment is to support the performance and resource requirements of all VNF instances while minimizing performance costs. The latency, throughput and capacity consumptions of VNF instances using models determined via simulation and test execution are used to choose

software or hardware instances and place them at available locations in the platform. The SDN switch is used for inter-CR interconnection.

A bin packing solution is used to minimize the overall cost while meeting VNF resource constraints and latency and throughput requirements. Table 4.2 presents notation for VNF instance parameters and (4.1) represents the cost of using a CR r to implement VNF m.

$$c_{rm} = \frac{\lambda_m - \Lambda_{rm}}{\lambda_m} + \frac{\Theta_{rm} - \theta_m}{\theta_m} + \frac{\Phi_{rm} - \phi_m}{\phi_m} \tag{4.1}$$

In general, the cost of assigning a VNF to a CR is minimized when the latency, throughput, and computational capacity of the CR is best matched to the VNF. In the context of VMs, computational capacity indicates the number of operations (e.g. expression matchers) required by the VNF. Our bin packing formulation assigns a VNF m to a CR r as indicated by the binary variable g_{rm}. Iterative improvement progresses through a series of cost reducing swaps until further cost reductions are not possible (4.2). For the number of resources in our system, a full enumeration of possible assignments is possible to find the lowest cost match.

$$\min \sum_{r \in R, m \in M} c_{rm} \cdot g_{rm} \tag{4.2}$$

It should be noted that multiple resources from the set R can be grouped to implement a VNF M. Depending on performance requirements, between one and four VMs can be instantiated to implement a single VNF. The offline initialization is performed infrequently. A full evaluation of minimum cost VNF implementation takes less than one second.

4.3.4 Online VNF Instance Deployment

As network traffic volumes vary, some VNFs implemented in CRs may provide insufficient processing capabilities. Our networking platform has the ability to assess

Algorithm 1: High-level allocation algorithm

```
Data: Num. VNFs: M, Num. CRs: R
Result: Assignment of VNFs to CRs
while network system operational do
    if at least one underprovisioned_rm, m ∈ M then
        Identify m with largest resource gap
        Call underprovision(m)
    end
    if new VNF creation due to trigger then
        Create VNF M + 1
        Call underprovision(M + 1)
    end
    if at least one overprovisioned_rm, m ∈ M then
        Identify m with largest resource gap
        Call overprovision(decrease_resource, m)
    end
    if trigger condition for VNF m no longer valid then
        Call overprovision(remove_VNF, m)
    end
end
```

new traffic patterns and flow rates and update deployments via a fast online approach, while still optimizing resource costs. The input and output traffic rates at each CR are collected by the coordinator every 0.1s to detect network performance imbalances. If a VNF m is underprovisioned or overprovisioned on a CR r, online redeployment is triggered. Online redeployment can also take place if the resource evaluator determines a condition has been triggered based on an evaluation of global state (e.g. a firewall deployment as a result of SQLi attack detection) or if the condition no longer exists.

To handle increased traffic volumes, two strategies can be adopted: scaling up the current deployment of VNF m by adding more of the same type of CR resources (e.g. VMs) or migrating VNF m to a CR with additional resources and performance. To perform scaling, the coordinator must choose available resources which satisfy the *performance and resource constraints*. To support on-line deployment, new dynamic allocation algorithms were created. The algorithms consider both the resource and

Algorithm 2: Algorithm to address underprovisioning

```
Data: VNF num. m
Result: Assignment of VNFs to CRs
Identify CRs (multiple VMs, FPGA) that meet performance and resource requirements
if at least one suitable CR is free then
    Identify CR r with lowest cost c_rm in (4.1)
    // In this case, the move is implementation in FPGA or adding another VM
    Move m into r
end
if all suitable CRs are busy then
    Identify suitable CRs with VNFs that are overprovisioned, set OP
    Identify VNFs in OP that can be implemented with a free CR
    // Determine migration cost for VNFs in OP to free resources
    Identify CR r in OP for lowest-cost implementation of m and lowest-cost
      implementation of VNF in r to a free CR
    Perform swap
end
```

Algorithm 3: Algorithm to address overprovisioning

```
Data: Parameter: decrease_resource or remove_VNF, VNF num. m
Result: Assignment of VNFs to CRs
if decrease_resource then
    if CR r contains multiple VMs then
        Remove one or more VMs implementing m
    end
    if CR r is a single VM or an FPGA then
        Make no change
    end
end
if remove_VNF then
    Remove m and deallocate associated CR r
end
```

performance costs outlined in (4.1). If a new VNF is needed or an existing one is underprovisioned, (re)deployment can straightforwardly be performed if a suitable free resource is available in the system. However, if a free resource is not present, it may also be appropriate to migrate a currently deployed and overprovisioned VNF to a lower provisioned resource to make room in the CR for the underprovisioned or new VNF.

To support dynamic on-line VNF allocation, algorithms for addressing both resource underprovisioning and overprovisioning have been developed. VNF deployment is continually assessed in a loop by the resource manager in the coordinator as illustrated in Algorithm 1. The coordinator monitors middlebox performance and global state to identify resource provisioning imbalances and global state triggers. As defined by (4.3), underprovisioning indicates that the current CR assignment for a VNF is insufficient in throughput, latency, or compute capacity.

$$underprovision_{rm} \rightarrow (\lambda_m > \Lambda rm) \ or \ (\Theta_{rm} > \theta_m) \ or \ (\Phi_{rm} > \phi_m) \tag{4.3}$$

Overprovisioned ($overprovisioned_{rm}$) indicates that performance and resource needs are met by the current CR but they could also be met by another, more resource efficient CR. The coordinator examines all computation resources to select appropriate resources, then calculates the overall deployment costs and picks the lowest cost one to achieve the online deployment update. In the case of underprovisioning due to a resource mismatch or a trigger, an attempt is made to commence VNF operation in a free CR. If a free resource is unavailable, a swap with an overprovisioned resource is performed, as shown in Algorithm 2. If a VNF is overprovisioned and contains multiple VMs, a VM is deallocated, as shown in Algorithm 3. If an FPGA-bound VNF that could be supported by VMs is overprovisioned, it is left intact until an underprovisioned VNF requests its use. Overall, if there are n CRs, the calculation time complexity is O(n) and allows the coordinator to quickly adapt to network dynamics.

4.4 Experimental Approach

4.4.1 Comparison with Previous Approach

In the first experiment, we make use of the same test environment (Figure 3.17) which was used in the full reconfiguration test. But this time we make the packet processor on FPGA partially reconfigurable. To justify the benefits of applying the

partial reconfiguration approach in our system, we compare this approach against the full reconfiguration approach described in Chapter 3 by measuring the total time cost for the function migration between different hardware resources.

4.4.2 Testbed Setup for Resource Scheduling and Allocation

The evaluation testbed comprises a NetGear 10G SDN switch, a general L2 switch, a Terasic DE5 FPGA board, two 12-core Intel Xeon workstations (2.4 GHz, 32 GB SDRAM, two 10 Gbps and four 1 Gbps NICs), and three Intel Duo servers (2.66 GHz, 4 GB) (Figure 4.3). We implemented a software-based flow generator tool on the source host and used it to generate bandwidth-adjustable multi-flow network traffic. The network traffic is able to be steered by the SDN switch to any output port. The SDN switch can also make copies of the network traffic and forward them to the VM- or FPGA-based packet processors. The traffic is finally captured by the sink host. The coordinator is implemented using a separate Intel Duo server. It connects to the control port on SDN switch, the proxy application working on the server and the Nios II microprocessor via a general L2 switch. The coordinator monitors the network traffic changes and accordingly manages the scheduling and reallocation of the compute resources within the network.

The testbed totally has six heterogeneous CRs, two of which are hardware CRs located on the FPGA middlebox, and four others are located on two servers. Two VMs as software CRs are installed on each server. Both hardware CRs and software CRs support the deployment of the same network function. Packet processing on a separate CR is physically isolated due to the connection of each CR to a single physical network interface. The processing ability of the VM- and FPGA-based CRs are pre-measured, as shown in Table 3.2.

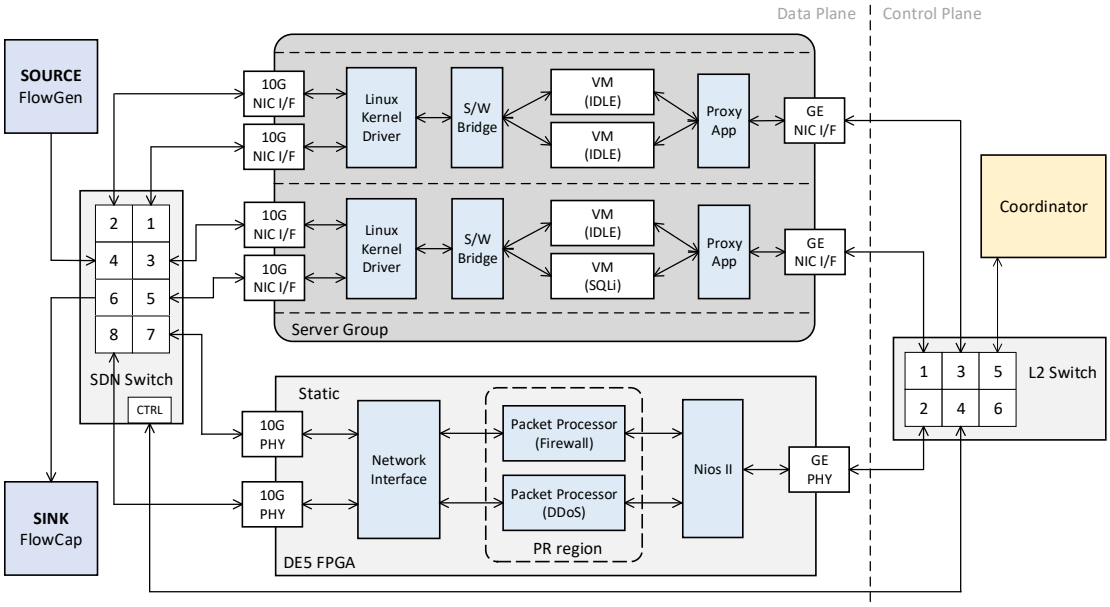

Figure 4.3: The experimental testbed. Available computer resources include 4 VM and 2 FPGA-based packet processors

4.4.3 Algorithm Evaluation

We evaluate our resource scheduling and allocation algorithm in an NFV environment with heterogeneous resources. There are three VNF instances (i.e. an SQLi attacking detector, a DDoS attacking detector and a firewall) that need to be deployed in the test environment. The coordinator finds suitable CRs for these VNF instances according to their performance and capacity requirements, and initializes the VNF deployment as shown in Figure 4.3. The DDoS detector and firewall were placed at two packet processor cores within a single FPGA, while the SQLi detector was allocated to a VM.

The flow generator on the source host generates two types of network attacking flows: SQLi attacking flows and DDoS attacking flows. The coordinator controls the SDN switch to forward a copy of the SQLi attacking flows to the SQLi detector and forward a copy of DDoS attacking flows to the DDoS detector. All copy flows will

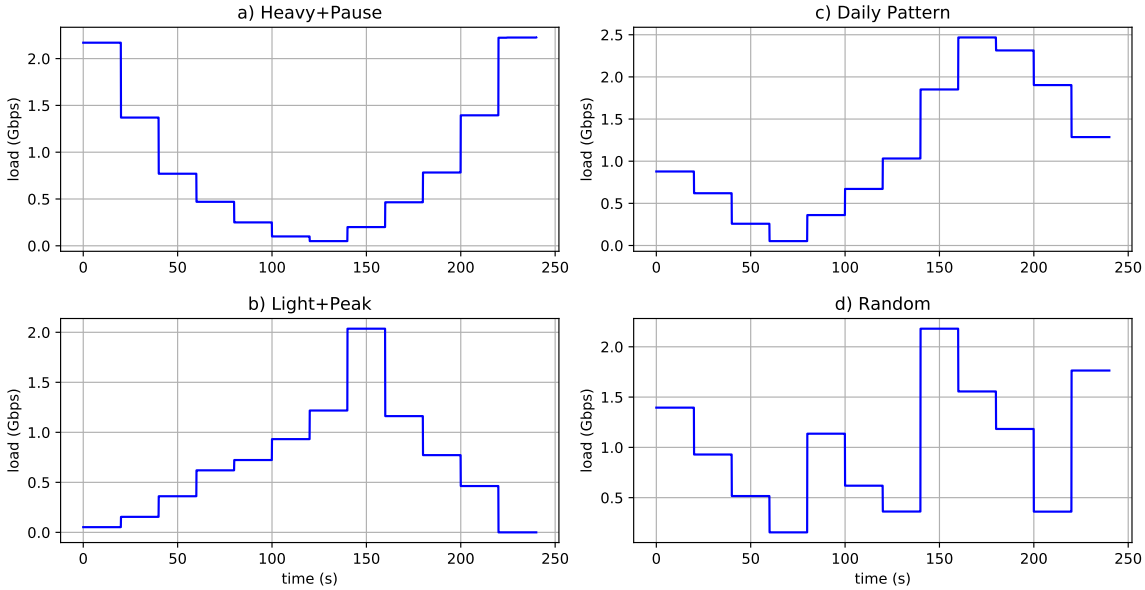

Figure 4.4: Traffic load patterns used in our evaluation model

be consumed at the CRs and the original network flows will finally go through the firewall to the sink host.

In our system model, the coordinator decides when to scale up/down a set of software CRs to satisfy the requirement of a VNF instance, and when a processor-based VNF instance must be migrated to a FPGA and vice versa. The coordinator dynamically adjusts resource allocation to accommodate two unbalanced resource provisioning situations, namely underprovisioning and overprovisioning. To evaluate the online VNF instance deployment algorithm introduced in Section 4.3, we use realistic network workloads that have been previously used to assess NFV platforms [57] (Figure 4.4). Pattern 4.4a is starting with a high traffic workload with a pause to go down and then recover. Pattern 4.4b shows the network traffic augmentation and diminution process due to the transient surge in workload. The daily pattern 4.4c resembles a typical backbone link [28], which we have scaled to a link capacity of 2.5 Gbps. The random scheme was constructed by taking random numbers in the range of 0 and 2500. The total evaluation time is 250 seconds.

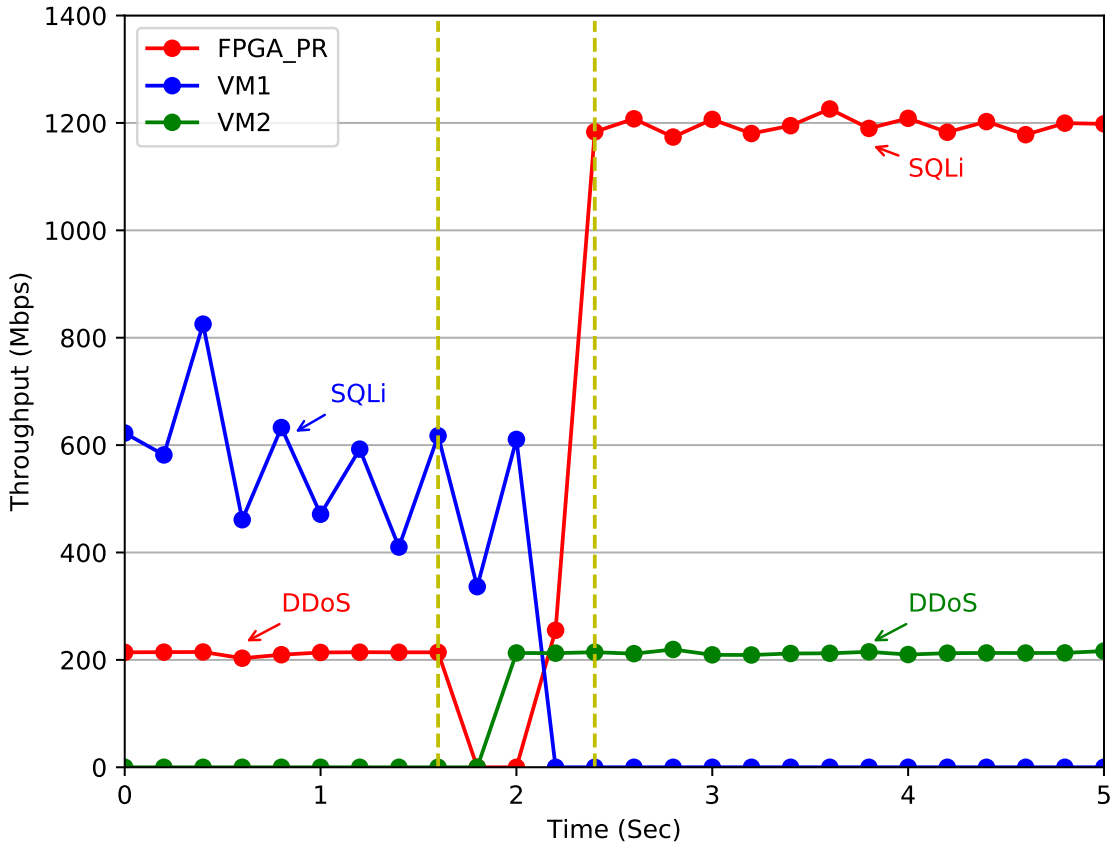

Figure 4.5: Performance of system resources during partial FPGA reconfiguration. Resource migration is performed between the yellow lines in the figure.

4.5 Experimental Results

4.5.1 Speedup by Partial Reconfiguration

In order to get a performance comparison between the FPGA partial reconfiguration approach and the full reconfiguration approach, we follow the same experimental steps described in Subsection 3.6.4 to perform another reconfiguration test. Figure 4.5 shows the delays associated with the redirection of the SQLi traffic from VM1 to the FPGA and FPGA reconfiguration using partial device (FPGA_PR) configuration.

Compare to full reconfiguration, the partial FPGA reconfiguration process requires about 0.8 second which primarily consists of partial bitstream loading from flash. The size of the partial bitstreams for both SQLi and DDoS are 15.7 MB, which is half of the size of the entire bitstream. The FPGA reconfiguration time is dramatically

reduced for partial versus full reconfiguration because, first, the partial bitstream is smaller; second, the Nios II does not need to be resynchronized with the coordinator; third, a dedicated hardware module implemented on FPGA takes place of Nios II to load bitstream from flash faster. Since partial reconfiguration is much faster, we will employ this technique in the algorithm evaluation experiment.

4.5.2 Time Cost for Resource Allocation

In preparation for examining the performance of VNF migration, the time costs for reallocating hardware resources to deal with underprovisioning or overprovisioning are evaluated. We have introduced two approaches in Section 4.3.4 that can be used to handle the mismatch between performance demand and supply for a given VNF. A VNF instance can be either migrated between a VM and an FPGA packet processor, or deployed to multiple VMs to gain more computation power. A series of 100 tests were conducted in the laboratory using four VMs and a DE5 board to assess the duration of various system configuration changes for VNF deployment. Figure 4.6 illustrates the CDF of the time required to perform several system configuration changes.

The migration of a VNF instance needs three steps: reconfiguring the new hardware resource to support the VNF, steering the network flows from an old hardware to a new one, and releasing the old resource. Similarly, scaling up the VNF deployment by adding more VM resources also has two steps. The first step is to run the same VNF in newly added VMs. The second step is to rebalance the distribution of workloads across the group of VM resources. The configuration manager in the coordinator performs these processes and interacts with the SDN switch controller to steer and balance the network flows.

As shown in Figure 4.6, migrating a VNF instance from a VM and an FPGA packet processor (red curve) takes less time than performing migration in the opposite

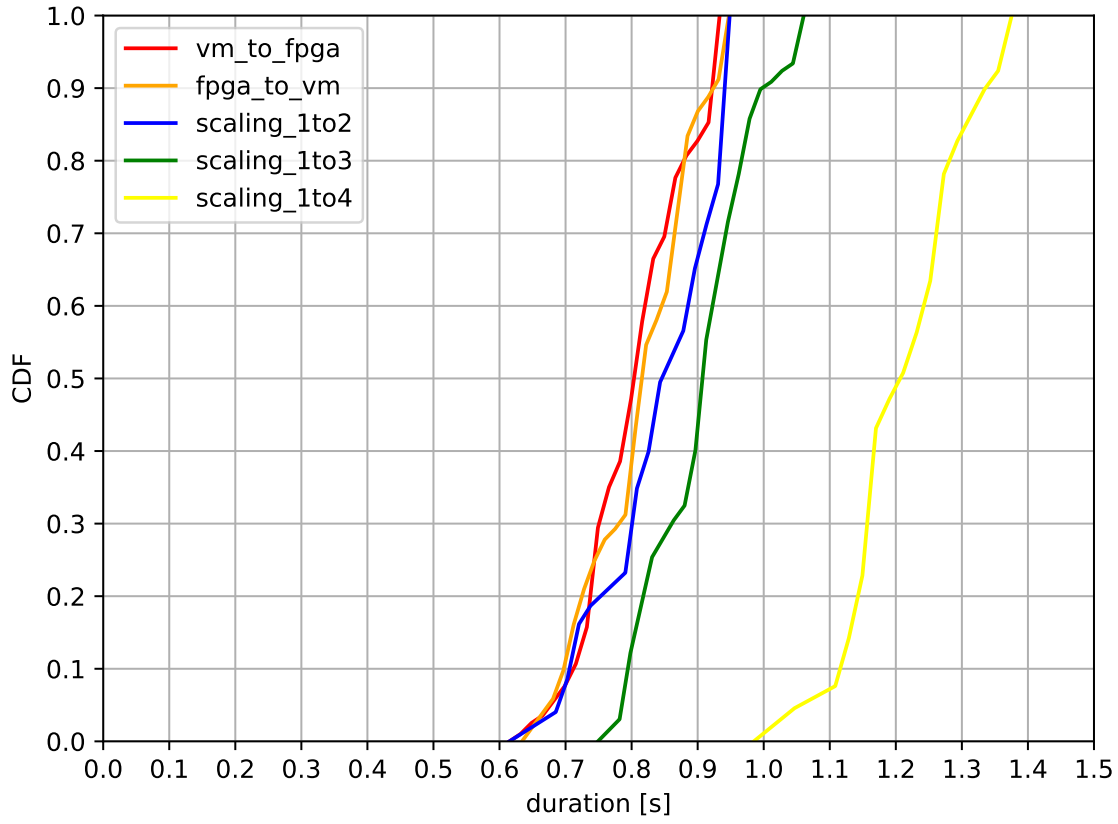

Figure 4.6: Cumulative distribution function of configuration and migration times of VNFs in CoNFV. The term *scaling_1to4* indicates the amount of time needed to scale from 1 VM to 4 VMs

direction (orange curve). Thus, reconfiguring the FPGA packet processor to support a given VNF instance is faster than launching the same VNF software on the VM due to overheads associated with the operating system. The blue, green and yellow curves indicate that the duration of scaling up VMs increases with the number of VM resources added. As more VMs are added to support the same VNF instance, the time it takes to launch the same VNF software on multiple VMs and rebalance the network traffic load across the VM group increases accordingly. The yellow curve shows that scaling up the VNF deployment from one VM to four VMs is significantly slower than the other procedures. It is because two of the newly added VMs are installed on the same workstation. Starting software VNF simultaneously in these

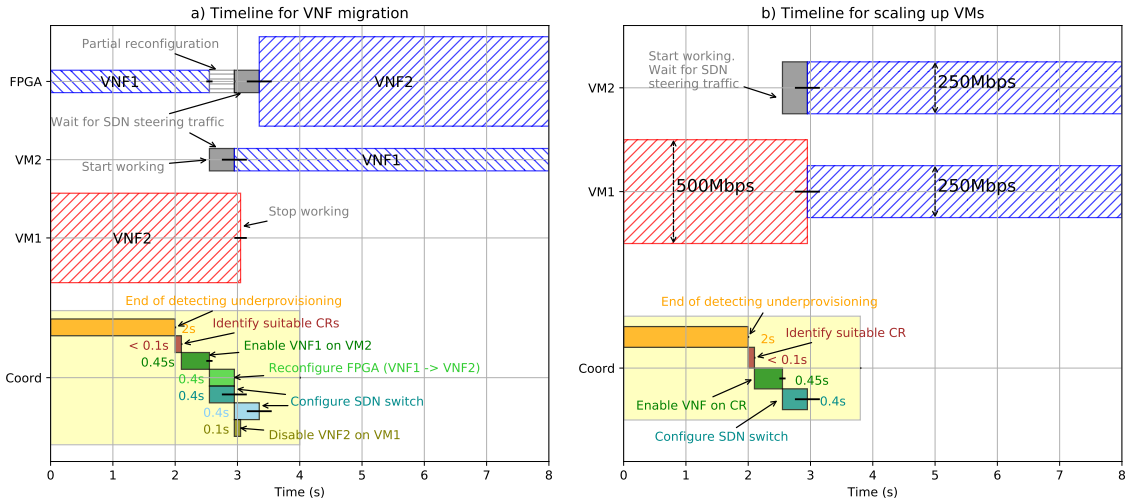

Figure 4.7: System reconfiguration timelines of VNF migration (a) and VM addition (b) in response to underprovisioning.

two VMs increases the duration for reallocating the VNF instance. The Figure 4.6 also indicates that the time cost of the migration approach is not worse than the time cost of deploying VNF on multiple VMs, which provides guidance for the coordinator to perform online VNF instance deployment.

Based on the result shown in Figure 4.6, we provide the timeline for swapping the VNF between FPGA and VM (Figure 4.7a), as well as the timeline for scaling up the VNF deployment within two VMs (Figure 4.7b). The steps of the coordinator performing these two operations are shown in detail in Figure 4.7. In order to decrease the probability of false positive, the coordinator spends 2 seconds to collect enough information from the running CRs to detect the underprovisioning. Then the coordinator identifies the proper strategy to relieve the underprovisioned state of a given VNF instance, either migrates the given VNF to a higher performance CR (e.g. FPGA) or extends its deployment by adding more CRs with similar performance.

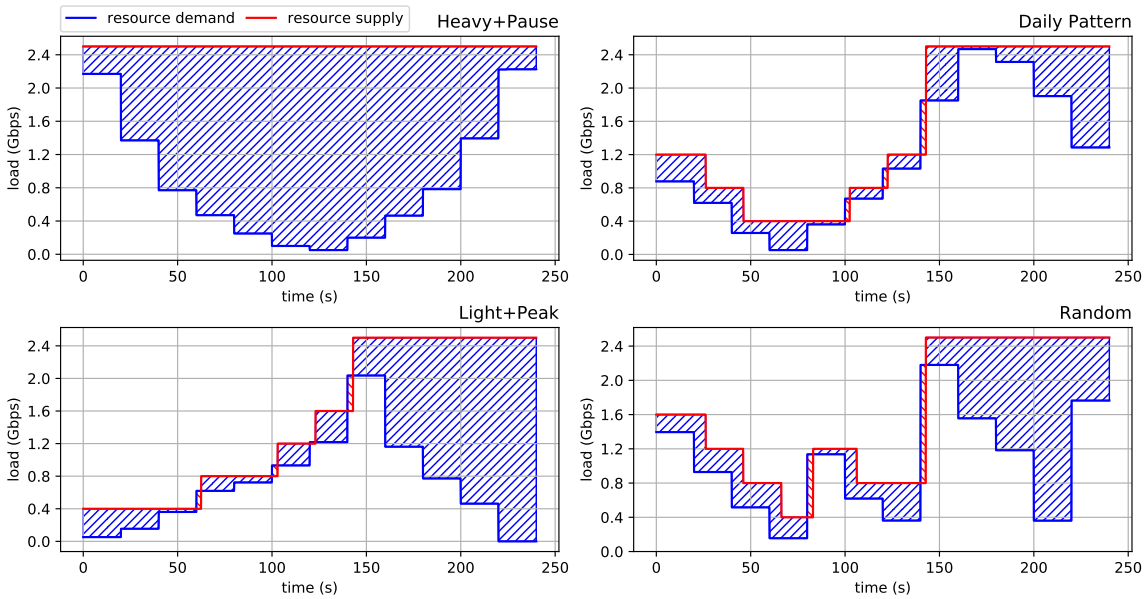

Figure 4.8: Resource supply vs. resource demand. Single SQLi is instantiated in the testbed and tested respectively with four traffic load patterns.

4.5.3 Algorithm Evaluation Results

We evaluate our VNF deployment algorithm with the network traffic patterns shown in Figure 4.4. Consider two scenarios for adjusting VNF deployment as the load changes across the network. First, there are idle CRs available for upgrading the performance or computer capacity of an underprovisioned VNF instance. Second, all suitable CRs are busy when the resource underprovisioning occurs. We conducted separate experiments on these two situations.

In the first experiment, we only deploy one VNF instance (an SQLi detector) in our testbed. Figure 4.8 shows the change in the resource supply versus the resource demand as the network load fluctuates. The red (blue) curve denotes the supply (demand), the red and blue shaded areas depict under and overprovisioning respectively. In the Heavy+Pause scenario, the traffic load is very high at the beginning, so the coordinator initially deploys the SQLi detector on an FPGA packet processor. In the Light+Peak scenario, as the traffic load increases gradually, the coordinator finds

that scaling up the VNF deployment on VM groups costs less than migrating the VNF from VM to FPGA. Therefore the coordinator adds more VM resources for the deployment of the SQLi detector. When the traffic load goes up to a point that the SQLi detector has been rather underprovisioned on all available VM resources, the coordinator migrates the VNF to an FPGA resource which has a higher performance advantage. The VNF migration from VM to FPGA also happens in Daily Pattern and Random cases when available VM resources cannot provide sufficient computation power. Before that, the coordinator narrowed the VNF deployment on multiple VMs as the network traffic load decreased according to Algorithm 3. Due to the fact that generally the overprovisioning is less urgent than underprovisioning, the coordinator is not eager to reduce the allocation of compute resources for an overprovisioned VNF instance. That's why we can find in Figure 4.8 the coordinator spends more time to identify overprovisioning situation.

In the second experiment, three VNF instances were initially deployed in the testbed as shown in Figure 4.3. Two types of attacking flows with different load patterns were synchronously forwarded to the SQLi detector and the DDoS detector. Throughput demand and supply curves for one VNF of DDoS (a) and SQLi (b) appear in Figure 4.9. Results were collected from the system hardware. As shown in the Light+Peak scenario in Figure 4.9(b), the instantaneous surge in SQLi attacking flows result in an increased demand for the computation power of CRs, ultimately exceeding the processing power provided by four VMs in the testbed at around the 140 second mark. At that point, the coordinator identifies that all the FPGA packet processors are occupied by other VNFs, but the DDoS detection function on one FPGA packet processor is overprovisioned. Migration of SQLi to the FPGA and DDoS to one VM takes place at this point. As DDoS traffic increases, more VMs are allocated to the flow processing until at 235 seconds the resources are swapped back.

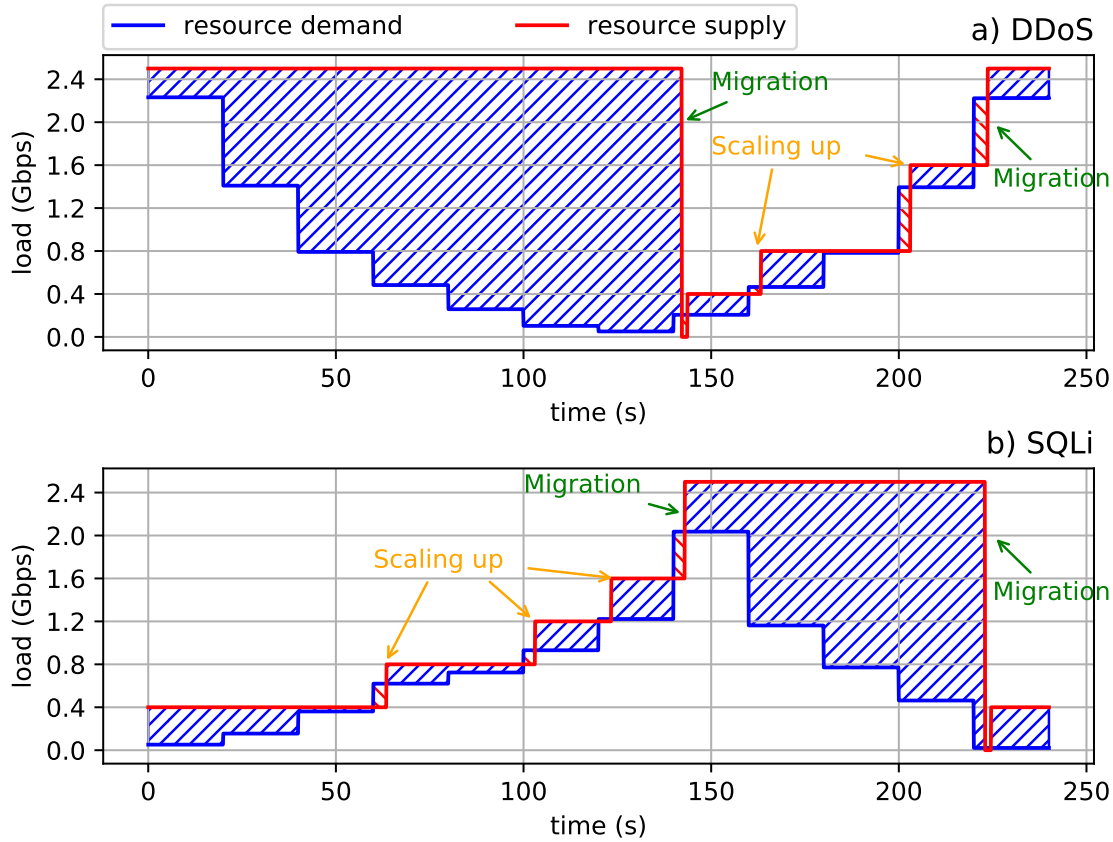

Figure 4.9: Demand for SQLi and DDoS and resource supply using FPGAs and VMs. The processing demand and supply for DDoS are shown in (a). The corresponding values for SQLi are shown in (b). The resource demand curves are taken from prior work [57]

Figure 4.10 shows the traffic load variation for the same experiment from the perspective of the FPGA packet processor and the VM group. The FPGA packet processor provides sufficient processing power to the accommodated VNF instances. To avoid underprovisioning, the number of VMs scales upwards based on traffic load demand. During the VNF migration, both FPGA and VM-based CRs experienced a brief turnoff time due to the fact that no CRs available at the time for temporarily placing the VNF instance to be migrated.

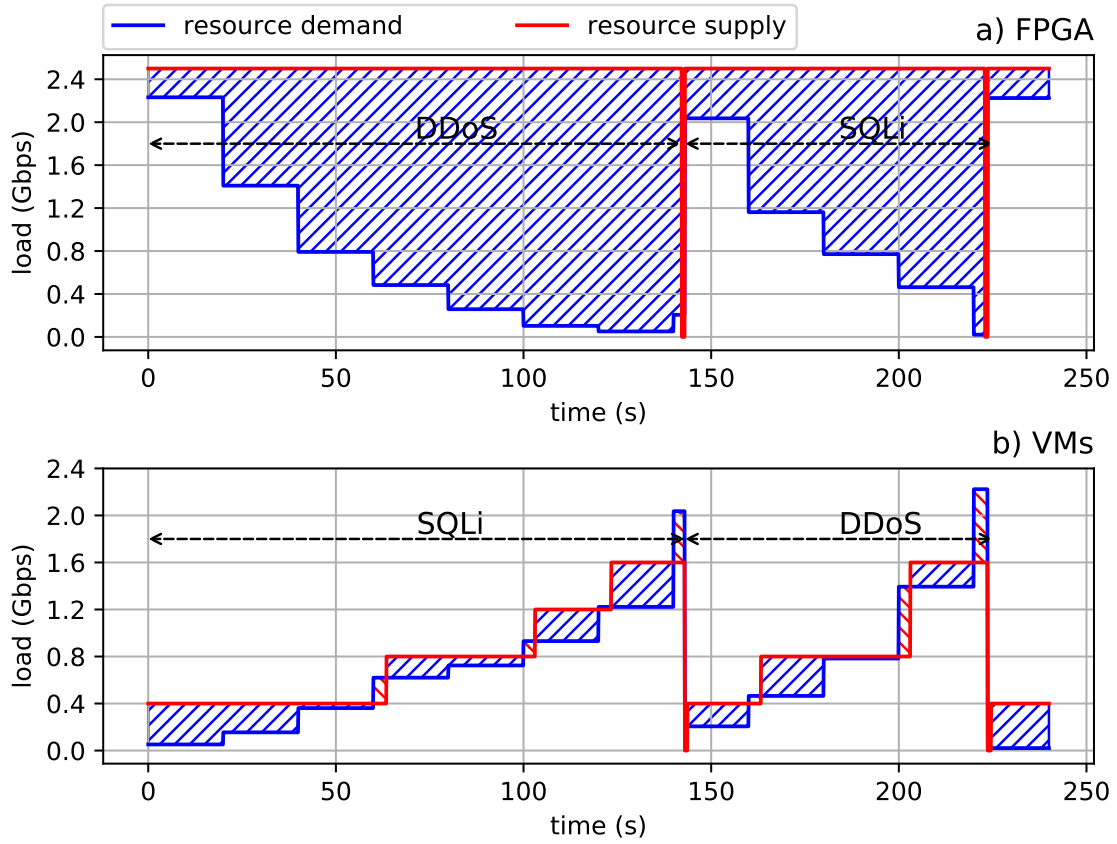

Figure 4.10: Demand for SQLi and DDoS and resource supply using FPGAs and VMs

4.6 Conclusion

In this chapter, we demonstrated the benefits of using partial FPGA reconfiguration to improve the logical isolation and speed up the dynamic reconfiguration of an FPGA packet processor in our system. Partial FPGA reconfiguration is shown to accelerate the migration of FPGA-based VNFs by a factor of 15. We also illustrated a new performance-aware VNF deployment algorithm which autonomously adjusts the usage of heterogeneous compute resources in an NFV system. Several experiments were performed in the lab and delineated that our algorithm can effectively use available resources to alleviate underprovisioning situation and reasonably release resources for overprovisioned VNF instances.

CHAPTER 5

DYNAMIC SERVICE CHAINING FOR HETEROGENEOUS MIDDLEBOXES

5.1 Introduction

Most NFV systems, including systems with FPGAs, use chains of functions. For large-scale NFV systems, traffic must be steered through functions in a sequence. Although early systems used a centralized controller for sequence orchestration [6, 16, 20, 62], the approach was burdened by the risk of central point failure, difficulties in synchronizing routing table rules, and steering traffic across network domain boundaries. In contrast, the use of *distributed agents* in session-based traffic coordination can effectively overcome these shortcomings [90]. A session is a series of interactions between two communication endpoints that occur during the span of a single connection. Packets belonging to the same session have the same source and destination addresses. A session-level approach for service chaining steers packets belonging to the same session through middleboxes deployed between the session endpoints. To date, NFV based on session-level approaches has only been applied to microprocessor-based systems due to the difficulty in dynamic traffic steering among heterogeneous middleboxes and managing partial FPGA reconfiguration.

In this chapter, we describe a new distributed-agent NFV system that supports the dynamic service chaining of FPGAs and microprocessors. Our new approach deploys distributed agents in end-hosts and both FPGA- and processor-based middleboxes and steers packets of individual sessions through corresponding service chains without any alterations to end-host applications, middlebox applications, or IP routing.

We implement FPGA-based agents to support high-performance packet processing. The agent cooperates with a partial reconfiguration IP core to manage the dynamic reconfiguration of middlebox functions on FPGAs. We verify our new approach with QUIC sessions and show the benefits of implementing our agents with FPGA circuits to steer QUIC sessions compared with software-based implementations. To support inter-domain routing when end hosts of the service chain are on different networks, similar to data center setups, we implement distributed agents to steer traffic based on the session information saved in the packet header. Agents update the packet header at the IP and transport layers to steer packets belonging to different sessions through corresponding service chains.

The remainder of this chapter is structured as follows. In Section 5.2, we present the architecture of our session-level approach. In Section 5.3, we describe how to use our approach to dynamically reconfigure a service chain. Details of the implementation are provided in Section 5.4. In Section 5.5, we evaluate our method by a series of experiments using both FPGA- and processor-based middleboxes. Section 5.6 concludes the chapter.

5.2 Architecture

A service chain is a series of connected network functions, which provides a pathway for network traffic that includes network services. Network functions in the chain, such as a firewall, intrusion detection system (IDS), or content cache, can be implemented on commodity hardware, custom hardware, virtual machines (VM), or reconfigurable hardware deployed across multiple subnetworks. In Figure 5.1, the client and server at either end of the service chain are located in different subnetworks. FPGA and processor resources in the local area network (LAN) and cloud are used to deploy network functions to process in-transit data packets.

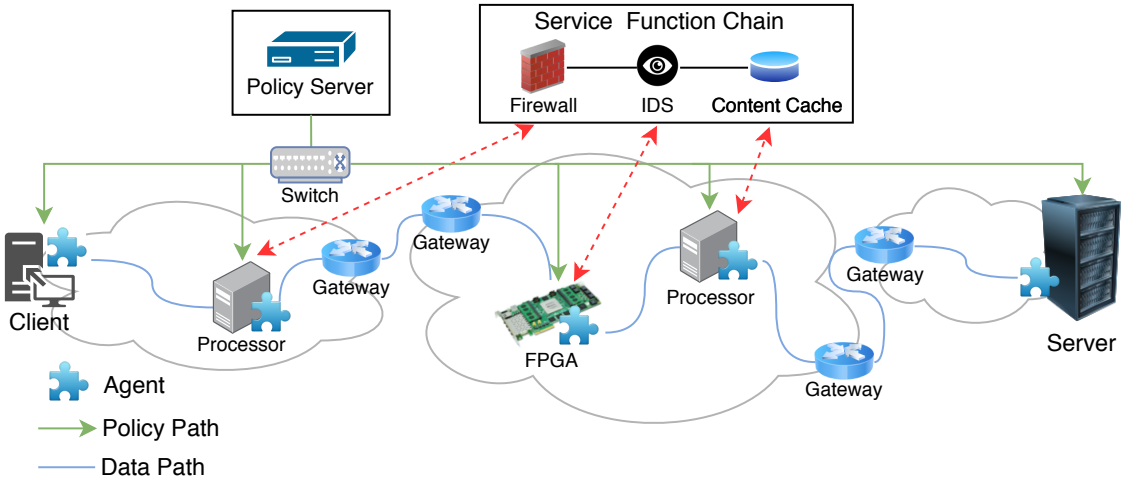

Figure 5.1: An example of a inter-domain service chain established by using our agents and the policy server

A session is a series of interactions between two communication endpoints. In Figure 5.1, end-hosts (i.e. server and client) communicate with each other by establishing sessions. Our approach runs agents on end-hosts, FPGA- and processor-based middleboxes, builds a service chain while creating a session, and directs session packets through network functions in the service chain. Agents rely on basic IP routing and high-level policies, which can be obtained from a policy server, to steer packets between end-hosts and middleboxes located in different subnetworks.

5.2.1 Components and interfaces

A service chain for a session includes a chain of middleboxes and subsessions. Each subsession connects an end-host and a middlebox, or two middleboxes. **Agents** set up individual subsessions in the service chain. The establishment and teardown of subsessions are synchronized with the setup and teardown of the session. An agent can maintain multiple subsessions at the same time. Each subsession is identified by a five-tuple (i.e. source/destination IP address, source/destination port number, and protocol), that is used for a specific service chain. The agent on an end-host or a middlebox in a service chain forwards packets with the original header of the session

to the end-host application or the middlebox application; as such, our approach works with existing application-layer protocols. Before transmitting session packets to the next middlebox or end-host, the agent rewrites packet headers using the subsession five-tuple. In this way, agents steer packets through the service chain.

The first agent in a service chain (e.g., the agent on client end-host in Figure 5.1) starts service chain creation according to a chaining policy received from a **policy server**. The chaining policy specifies an ordered list of middleboxes and end-hosts located in the service chain. Each middlebox or end-host in the policy is identified by its IP address. For example, the policy for establishing the service chain in Figure 5.1 includes an ordered list of IP addresses for the client end-host, the first processor-based middlebox, the FPGA-based middlebox, the second processor-based middlebox, and the server end-host. During service chain creation, the chaining policy is passed forward by the agents along the service chain from the first agent in the chain to the last one. Each agent sets up a subsession to connect to the next middlebox or end-host indicated by the policy.

Agents in a service chain can be triggered by the policy server to reconfigure the service chain for an ongoing session. Each middlebox or end-host communicates with the policy server via a TCP connection. In other use cases, agents can initiate the reconfiguration of the service chain based on the current working status of middlebox applications without the involvement of the policy server. For example, an IDS middlebox may detect a certain type of attack and start service chain reconfiguration to add a packet interception function (e.g. firewall) to the service chain.

In our approach, agents in a service chain can receive high-level policies from the policy server to determine how to establish or reconfigure the service chain, but the policy server cannot enforce policies by installing forwarding rules in network devices. Thus, our policy server is unlike the SDN controller that centrally manages network traffic. The establishment and reconfiguration of a service chain are carried

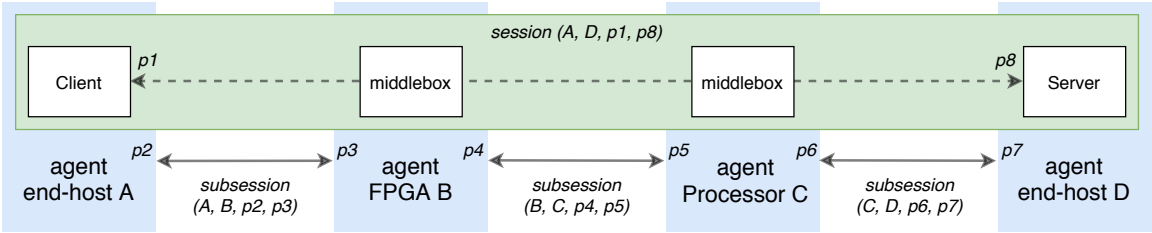

Figure 5.2: A session composed of a chain of FPGA- and processor-based middleboxes and subsessions

out exclusively by the distributed agents running on end-hosts and middleboxes in one or multiple subnetworks.

5.2.2 Service chaining of heterogeneous middleboxes

A service chain can be set up during session creation. One service chain can span multiple sessions, and a session can exist for multiple service chains. As an example of how to establish a service chain during a session, we present the case where the service chain and the session have the same end-hosts, as shown in Figure 5.2.

In this example, the client and server run as applications on end-host A and end-host D, respectively. A session is set up between a client and server. The service chain consisting of end-host A, FPGA-based middlebox B, processor-based middlebox C, and end-host D is to be established for the session. Before agents set up the service chain for the session, session packets are transmitted between the client and server through basic IP routing by network devices according to the source and destination addresses in a session five-tuple. During this setup phase, the network paths used to transfer session packets are undefined.

To ensure session packets proceed through a sequence of middleboxes in the service chain, agents distributed on end-hosts and middleboxes must be organized to steer session packets along the chain's path. To achieve this goal, agents must establish subsessions along the service chain to connect the nodes (i.e. end-hosts and middleboxes) in the chain. Agents at connected nodes rewrite session packet headers

with the subsession five-tuples as the packet proceeds through the chain. The session created between the client and server has the original five-tuple represented as the IP addresses for end-host A and end-host D, source port $p1$ and destination port $p8$, and the session protocol. Two end-hosts, two middleboxes, and three subsessions form a completed service chain between the client and server.

Establishment of the service chain: Session creation is achieved through a handshake process. Different protocols require unique handshake methods. For example, a TCP session uses a three-way handshake (i.e. SYN, SYN-ACK, ACK) to make a connection between the client and server. A QUIC session starts a connection through a 1-RTT (round-trip time) handshake [34]. The handshake process is usually started by the client. Establishment of the service chain in Figure 5.2 begins when the first packet for a handshake from the client arrives at the agent at end-host A. The agent intercepts the packet and extracts the five-tuple from the packet header. The source and destination IP addresses in the five-tuple are used to find a matching policy that indicates an address list [A, B, C, D] for establishing the service chain. The agent at end-host A then creates a subsession to connect end-host A and middlebox B. The five-tuple of the new subsession includes the address of end-host A as the source IP address, the address of middlebox B as the destination IP address, the new allocated ports (i.e. $p2$ and $p3$) as the source and destination ports, and a protocol that is the same as the original session protocol. The agent rewrites the packet header with the new subsession five-tuple to ensure that the packet will be sent to middlebox B.

The agent at end-host A sends all packets from the client that belong to the same session to middlebox B via the newly created subsession. Subsession packets from middlebox B traveling in the opposite direction are sent to the client. The agent at end-host A needs to restore packet headers using the original session five-tuple before sending packets to the client. The agent creates two dictionary entries for mapping between the original session five-tuple and the subsession five-tuple. Mapping entries

in the dictionary are saved locally. The agent rewrites the header of packets in the same session according to these mapping entries. The address list and the original session five-tuple are added to the payload of the handshake packet and sent to middlebox B by the agent at end-host A. The agent on middlebox B then uses the address list to create the subsession with the next middlebox in the chain. The original session five-tuple (from end-host A) stored in the packet payload is used by agents in the service chain to generate locally saved mapping entries.

When the agent on middlebox B receives the handshake packet from end-host A, it checks to see if the payload carries an address list. If it does, the agent removes the address list from the payload and saves the list locally, then rewrites the packet header with the original session five-tuple also stored in the payload and delivers the packet to the middlebox application. The agent also creates dictionary entries to map the subsession to the session and vice-versa. When the handshake packet emerges from the middlebox, the agent retrieves the saved address list [B, C, D] and removes its own address to get [C, D]. It then follows the procedure above to create a new subsession from B to C, rewrites the packet header, appends the original session five-tuple and the address list to the packet payload, and transmits the modified handshake packet. This process continues along the service chain until the handshake packet reaches end-host D where it is delivered to the endpoint of the service chain. The agent at end-host D restores the handshake packet using the original session five-tuple saved in the packet payload and delivers the packet to the server application running on end-host D.

When the server at end-host D replies to the handshake packet, an acknowledge packet travels back along the chain of subsessions and middleboxes to continue the handshake. The forward and reverse paths for session packets must go through the same middleboxes. Between middleboxes, however, the forward and reverse network paths traversed by subsession packets need not be the same. When the acknowledge

packet arrives at the client at end-host A, the service chain for the session has been established. After the establishment of the service chain, all session packets exchanged between the client and server will travel through the service chain. For these packets, agents do not need to append the original session five-tuple and the address list to the packet payload. Agents steer packets along the service chain by modifying packet headers according to the mapping entries saved during service chain creation.

Teardown of the service chain: An established service chain allows for session packet transmission in two directions independently. For some protocols like TCP, one end of a session can send a FIN packet to indicate that it will send nothing more. Agents along the service chain tear down subsessions and the service chain as they receive the FIN, so when the TCP session is torn down normally, the chain is torn down along with it. Other protocols may not send FIN-like packets to indicate the end of the session. The termination of the session is achieved by the agents timing out the subsessions along the service chain. One agent maintains a subsession by saving the five-tuple translation between the original session and the subsession. If the agent can no longer receive packets from the session, it deletes the translation from its local mapping table, tearing down the subsession. With the teardown of all subsessions along the service chain, the chain is also torn down. If necessary, agents can use heartbeat signals to keep subsessions alive.

5.2.3 Service chain setup for QUIC sessions

QUIC [34] is a multiplexed and encrypted-by-default transport layer network protocol. It improves the performance of connection-oriented web applications by establishing a number of multiplexed connections between two endpoints over User Datagram Protocol (UDP) rather than Transmission Control Protocol (TCP) [50]. QUIC uses UDP as its basis, which reduces connection and transport latency compared to

that of TCP. A QUIC session starts with a handshake phase, during which client and server establish a shared secret using the cryptographic handshake protocol [81].

The secure handshake phase is completed by exchanging *Initial* and *Handshake* packets between the client and the server [34]. To adopt our service chain setup approach for a QUIC session, we append the original session five-tuple and the address list to the *Initial* packet payload without encryption. The service chain is established after the *Initial* packet is exchanged between the QUIC client and server. The termination of the QUIC session is achieved by the agents timing out the subsessions along the service chain.

5.3 Dynamic Reconfiguration

A service chain can benefit from dynamic reconfiguration to improve network resource utilization, save resource consumption, and adjust network resources to adapt to changes in the network environment. In our approach, reconfiguration of a service chain can be triggered by the policy server or a middlebox. As shown in Figure 5.3, the agents at the two ends of a segment of a service chain are the left anchor and the right anchor. Here, we define the left anchor as the agent close to the client, and the right anchor as the agent close to the server. Reconfiguration is initiated by the agent acting as the left anchor.

The policy server can trigger the service chain reconfiguration by sending a policy including a new address list to the left anchor. The new address list includes the IP addresses of the left anchor, middleboxes, and right anchor of the new path that will replace the old path. For example, the left anchor B in Figure 5.3 receives the address list [B, G, E] in which B is the left anchor and E is the right anchor. The address list [B, G, E] indicates the new path. Service chain reconfiguration is achieved by transmitting a series of control packets between the left and right anchors to create a new path and then switching the old path to the new path through two anchors.

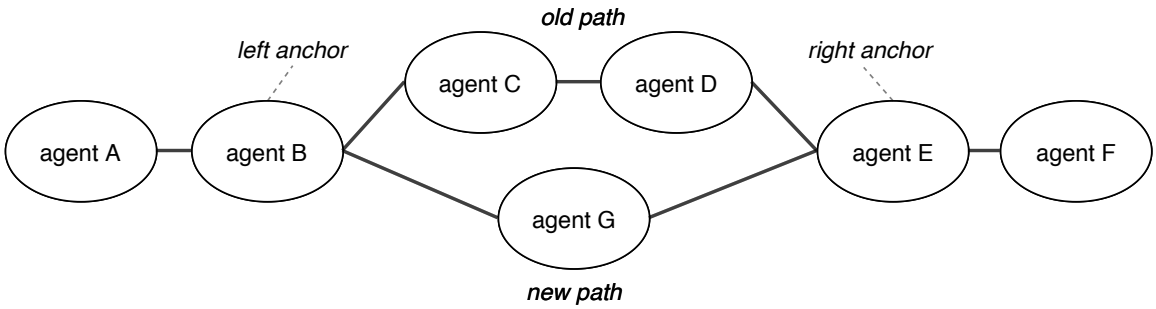

Figure 5.3: Agents reconfigure a segment of a service chain, replacing an old path with two middleboxes by a new path with one

Control packets are used to resolve contention if multiple portions of a service chain try to change at the same time, set up a new path for the service chain, and cancel reconfiguration if a new path cannot be created. The protocol detail for service chain reconfiguration follows the approach described by Zave et al. [90].

5.3.1 Reconfiguration protocol

We use a series of control messages to reconfigure the service chain segment between the left and right anchors. The control messages include *lock_request/lock_cancel* messages for locking/unlocking states of agents on the old service chain segment and a set of 3-way handshake messages (i.e. *Initial*, *Initial-ACK*, *ACK*) for establishing a new service chain segment. To simplify the implementation and reduce additional delays brought by processing control information, all control messages are implemented as UDP packets.

Just as the agent for end-host A in Figure 5.2 needs the address list [B, C, D] to set up the original service chain, the left anchor B in Figure 5.3 needs an address list [G, E] to specify the new path that will replace the old one. Figure 5.4 shows the exchange of control messages between the left and right anchors during the service chain reconfiguration. The red packets travel on the old path to configure agents' lock states, so they are forwarded through the agents of current middleboxes C and

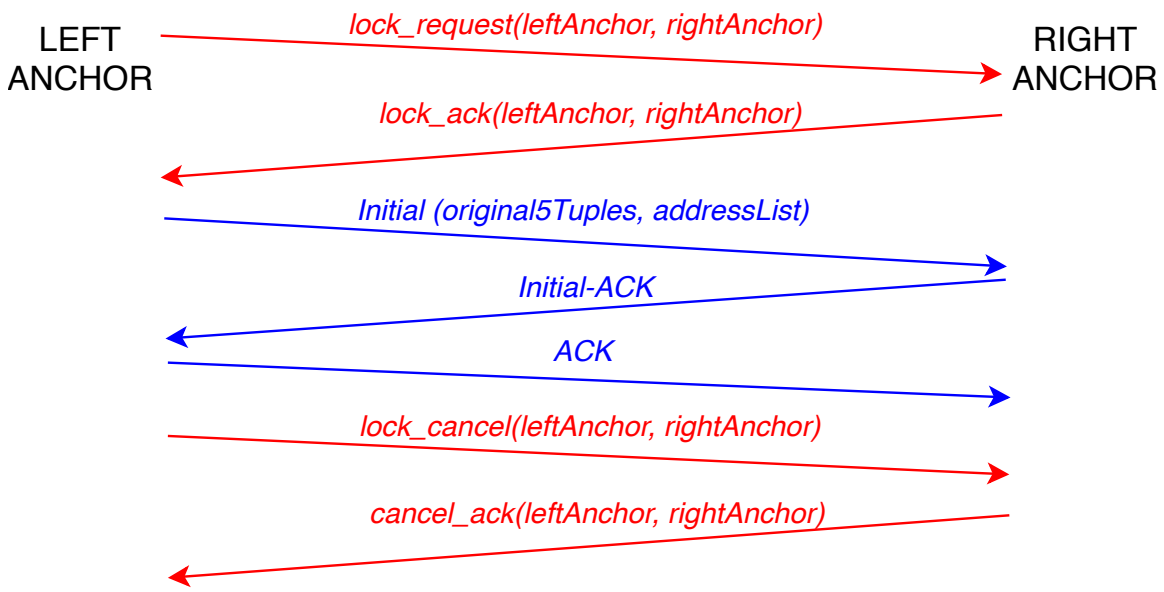

Figure 5.4: Control messages exchanged for reconfiguration. Red packets travel on the old path, blue on the new path

D. The blue 3-way handshake sets up the new path within the service chain. The *Initial* packet carries an address list so that the agents can include all the addressed middleboxes before the right anchor.

We use a previously-developed mechanism [90] to prevent conflicts caused by simultaneous reconfiguration of overlapping service chain segments. Each agent maintains three types of states: *unlocked*, *lockPending*, or *locked*. If the state is *lockPending* or *locked*, the agent uses a variable *requestor* to hold the left anchor of the request for which it is pending or locked. Reconfiguration starts with a *lock_request* sent by the left anchor, after which the left anchor changes its state from *unlocked* to *lockPending*. If an agent on the old path receives a *lock_request* from the left, the agent is not the right anchor, and its state is *unlocked* or *lockpending* but the *lock_request* is from the same left anchor, then it forwards the packet to the right, while setting the state to *lockPending* and *requestor* to the left anchor. When the right anchor receives the *lock_request* and its state is *unlocked*, it replies with a *lock_ack*, which travels on the old path to the left anchor, sets the state of every agent on the path to *locked*.

When an agent whose state is *locked* receives a new *lock_request*, if the *lock_request* comes from the same left anchor, the agent replies with a *lock_ack* directly, otherwise, it replies with a *lock_nack* to indicate that the agent is currently busy. A *lockPending* agent would send back a *lock_nack* if it receives a *lock_request* from a different left anchor. Any agent receiving a *lock_nack* from its right will change its state to *unlocked*.

The 3-way handshake is used to establish a new service chain segment between the left and right anchors. Agents on the new path use the method described in Section 5.2.2 to set up subsessions between middleboxes for a new service chain segment. During reconfiguration, the left anchor sends data on the old path. The left anchor only switches the path after it receives the *Initial-ACK* packet from the right anchor. The right anchor then forwards data received from the server on the new path after it receives the *ACK* packet. After swapping the path successfully, the left anchor sends a *lock_cancel* packet on the old path to restore agents' states to *unlocked*.

5.3.2 Partial reconfiguration with agents

An FPGA provides a high-performance platform to implement virtual network functions (VNFs) with performance that is often better than processor-based middleboxes [91]. When the throughput of the data stream or the requirements for specific VNFs change, the implementation of VNFs on different platforms can be dynamically adjusted. Since one FPGA platform may have multiple VNFs deployed, our approach allows an agent on an FPGA-based middlebox to partially reconfigure a VNF implementation during service chain reconfiguration without interrupting the operation of other VNFs on the same hardware. Agents running on processor-based middleboxes can adjust (reconfigure) middlebox functionality by starting/terminating VNF processes.

The need for a specific VNF can be included in the control packet during service chain reconfiguration. The agent triggers the partial FPGA reconfiguration of middlebox functionality before forwarding the control packet to the next middlebox. The session on the service chain continues to operate during reconfiguration, as session data is still transmitted on the old path during the FPGA partial reconfiguration.

5.3.3 State migration after reconfiguration

The reconfiguration of a service chain is often accompanied by replacing one middlebox with another and migrating states between swapped middleboxes. A state manager module [91] can be included in the policy server to handle state migration after chain reconfiguration. The server gives each middlebox in the service chain access to global state information via programmable interfaces. After the service chain reconfiguration, the newly inserted middlebox in the service chain retrieve states from the policy server via network interfaces.

5.4 Implementation

5.4.1 Framework overview

Figure 5.5 shows an overview of a service chain framework that is established using distributed agents and the policy server. We implement our agents on both FPGA- and processor-based middleboxes, and end-hosts. These agents establish and dynamically reconfigure service chains of QUIC sessions across heterogeneous middleboxes. FPGA-based middleboxes are implemented using two Terasic DE5 boards that include Intel Stratix V FPGAs. Processor-based middleboxes are implemented as Docker containers working on three twelve-core Intel Xeon workstations (2.4 GHz, 32 GB SDRAM, two 10 Gbps NICs, and four 1 Gbps NICs). QUIC clients and servers run on a 28-core Intel Xeon workstation (2.6 GHz, 128 GB SDRAM, two

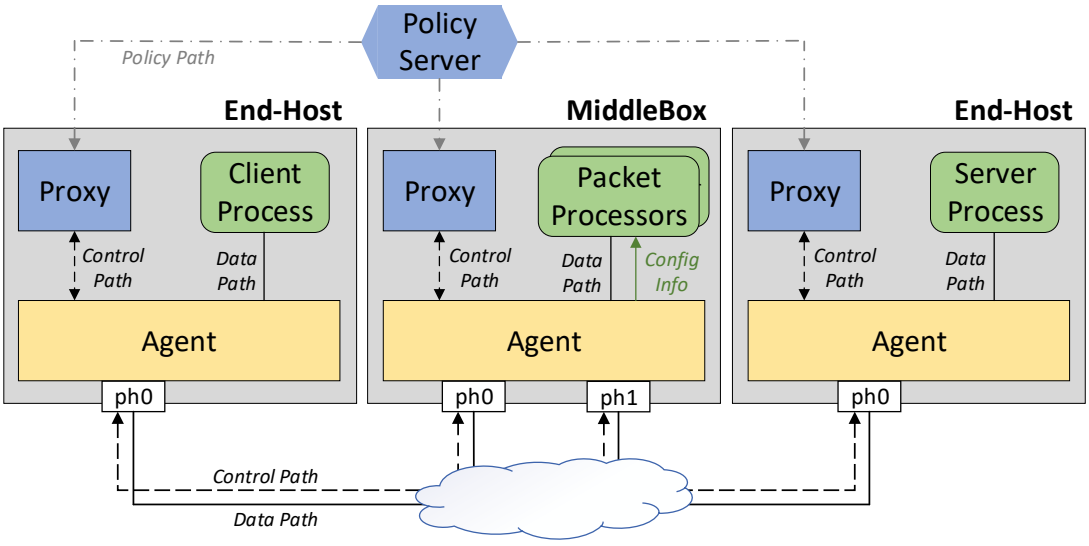

Figure 5.5: Overview of a service chain established by using our agents and the policy server

10 Gbps NICs, and two 1 Gbps NICs). The policy server is implemented using a processor-based Intel Duo server (2.66 GHz, 4 GB).

Our agents support FPGA- and processor-based VNFs by intercepting packets going to/from the network. The agent on general-purpose commodity hardware (i.e. end-hosts, processor-based middleboxes) is implemented as software running in user space. It utilizes the host network stack to communicate with the applications running on the host (e.g. QUIC client and server) and receive/send session packets from/to NICs. The agent on the FPGA platform is implemented as a dedicated hardware circuit that interacts with packet processors and network interfaces via the Avalon streaming bus. One FPGA platform can support up to three packet processors that operate as different network functions. The agent connects each packet processor to a physical network port. Both FPGA and processor packet processors are dynamically reconfigurable. The policy for reconfiguring local packet processors is obtained from the policy server.

In order to communicate with the policy server, we implemented a *proxy* module in each middlebox and end-host. The proxy connects to the policy server via a TCP connection. It forwards the service-chaining polices to the agent. For processor-based middleboxes, the proxy was implemented as a software program. On the FPGA platform it was implemented using a NIOS II processor. Compared to data packet transmission, communication between the policy server and proxies is infrequent.

5.4.2 Agent implementation on FPGA

The blocks used in a single agent unit are shown in Figure 5.6. The module is pipelined and divided into five submodules: the unwrapping module, the payload parser module, the reconfiguration manager module, the agent core, and the wrapping module. For experimentation, we implemented our agent design on a Stratix V 5SGXEA7N FPGA. One FPGA platform has three packet processors deployed (top half of Figure 5.6). The agent implemented on an FPGA contains three agent units, each of which connects a single packet processor with an individual 10G MAC core through the Avalon streaming bus.

The data packets from the network are fed to the unwrapping module. This module extracts the five-tuple information from the packet header and sends it and the payload to the payload parser module. The payload parser module uses the five-tuple as a key to search the hash table to obtain the address of the next middlebox and the original five-tuple of the session (before subsession modification). The module extracts the data packet payload and the control information used for service chain establishment and reconfiguration. When needed, the payload parser module updates the control information according to the policy received from the proxy module and adds an address list for constructing a service chain to the control frame. The payload parser module stores the received data packet in a small buffer.

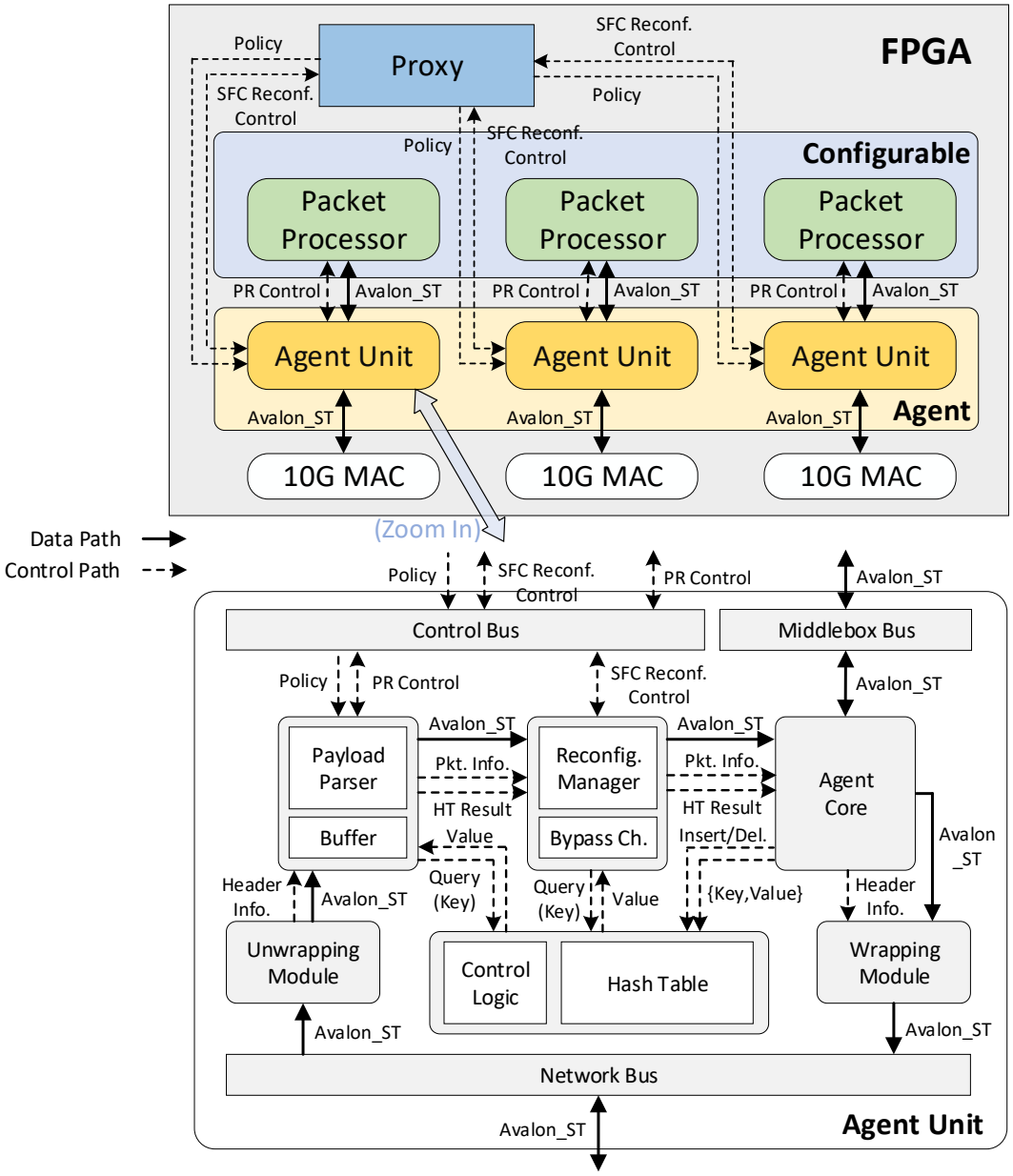

Figure 5.6: Implementation of an agent on an FPGA. The agent unit in the top subfigure is expanded in the bottom subfigure.

The reconfiguration manager module responds to the SFC reconfiguration control command received from the proxy module to generate reconfiguration control packets and start the reconfiguration of a service chain segment. This module uses a state

machine to manage the various stages of the service chain reconfiguration process. It obtains the address of the next middlebox on the service chain segment to be reconfigured from the hash table. If not used for reconfiguration, the reconfiguration manager module allows data packets to be passed directly to the agent core module through a bypass channel.

The agent core module performs hash table operations, rewrites the packet headers for data packets processed by the packet processor, forwards packets emerging from the packet processor, and provides header information about the next subsession to the wrapping module. The agent core module (1) inserts new entries in the hash table to record the mapping relationship between the original session and the new subsession during service chain establishment; (2) deletes old entries when a service chain is torn down and; (3) updates entries to reconfigure a service chain segment. The packet payload and header information are integrated into a complete data packet in the wrapping module, which is sent to the 10G MAC through the network bus.

5.4.3 Dynamically reconfigurable VNFs

For experimentation, a packet-based firewall blocker was created and tested. The firewall module was implemented in a partially reconfigurable region (PR region) on the FPGA device. The firewall module contains a hardware hash table to save packet blocking information. It tracks in-transit packets and filters them by source and destination network addresses, protocol, and source and destination port numbers. When a packet matches a set of filtering rules stored in the firewall (the packet exists in the blocking list), it is dropped by the firewall. Otherwise it is allowed to pass.

Table 5.1: Resource usage for SFC implementation cores targeted to a Stratix V 5SGXEA7N

	LUTs	FFs	Block Mem bits
Agent unit	20,976	21,948	891,770
Proxy	14,598	19,957	2,934,968
PR region	11,440	5,720	839,680
Firewall	2,822	5,283	504,960
Network interface	12,807	18,082	239,657
Available in FPGA	469,440	938,880	52,428,800

5.5 Evaluation

Three separate experiments were performed in the lab using our PC and FPGA-board virtualization system. An open-source tool ngtcp2[1] was used to generate QUIC sessions between clients and servers. We measured the latencies for session initiation to quantify the overhead introduced by our agents in establishing a service chain. Then, we measured the throughput of QUIC sessions in a service chain to verify the scalability of chains with agents. Finally, we assessed the ability of the system to reconfigure a service chain across multiple subnetworks. Results from these experiments are described in subsequent subsections. The resource counts of the agent, the proxy, the PR region for a single packet processor, the firewall module, and the network interface are shown in Table 5.1. The size of the partial bitstream for the PR region is 5.8 MB, while the entire bitstream for the FPGA is 31.4 MB.

5.5.1 Session initiation

In this experiment, service chains including two end-hosts and up to six middleboxes were created. Each chain was created for a QUIC session in a single subnetwork. Two approaches were used: 1) (standard centralized) IP routing rules were inserted

[1]https://github.com/ngtcp2/ngtcp2/tree/draft-23

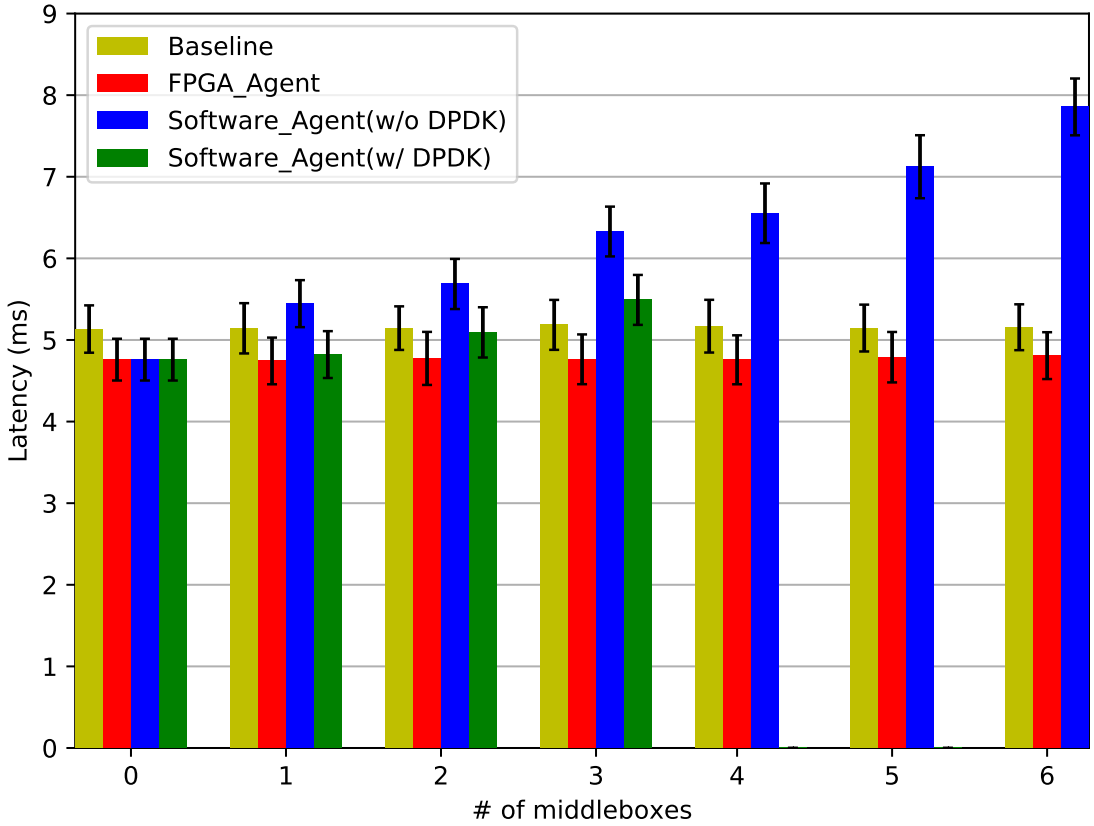

Figure 5.7: Latency for session initiation

in an SDN switch (Netgear ProSafe M4300-8X8F); 2) our agents were used in end-hosts and middleboxes. For both approaches, we measured the round-trip time of the handshake between the QUIC client and server during session initiation. We use the service chain set up time from the first approach as a baseline to measure the latency overhead introduced by our agents. During the experiment, only the overhead of agents, not the VNFs, was measured.

In the baseline test case, all service chain middleboxes are FPGA-based. Each of the two DE5 boards implements three packet processors per FPGA. No agents are deployed in the service chain. Session packets are forwarded by the SDN switch according to the routing rules inserted into the switch in advance.

The agent-based approach can establish a service chain using three formats: (1) a hardware circuit implemented the agent and the FPGA packet processor, (2) the agent and packet processor are implemented in software in a container that accesses the network stack, and (3) a software agent and packet processor based on the Intel Data Plane Development Kit (DPDK)) that directly accesses the network interface card are implemented in a processor-based middlebox. Three workstations sliced into two container middleboxes each are used to implement case (2). There a total of six non-DPDK software agents and three DPDK-based software agents were tested in the experiment. Agents running at end-hosts are software agents implemented using DPDK. Figure 5.7 shows the latency overhead caused by different agent implementations compared to the baseline.

The bars at the left in Figure 5.7 show that the agents deployed on end-hosts speed up the handshake process between the QUIC client and server. The agent skips the OS kernel through the DPDK and directly obtains data packets from the network interface card. The main source of latency overhead introduced by the agent is the time it takes to obtain packets from network interface, query the 5-tuple mapping table, and set up the subsession. Due to hardware parallelism, FPGA agents have obvious advantages over software agents in processing these tasks. The latency overhead introduced by FPGA agents (red bars) can be ignored compared with the total duration of the handshake during session initiation. In contrast, as more software agents are deployed in the service chain, the latency overhead caused by agents (blue bars) will increase. By using DPDK, a software agent can get data packets from the network interface card faster, thereby reducing the latency overhead caused by the software agent. However, the serial nature of the processor limits the performance of the software agent in establishing subsessions.

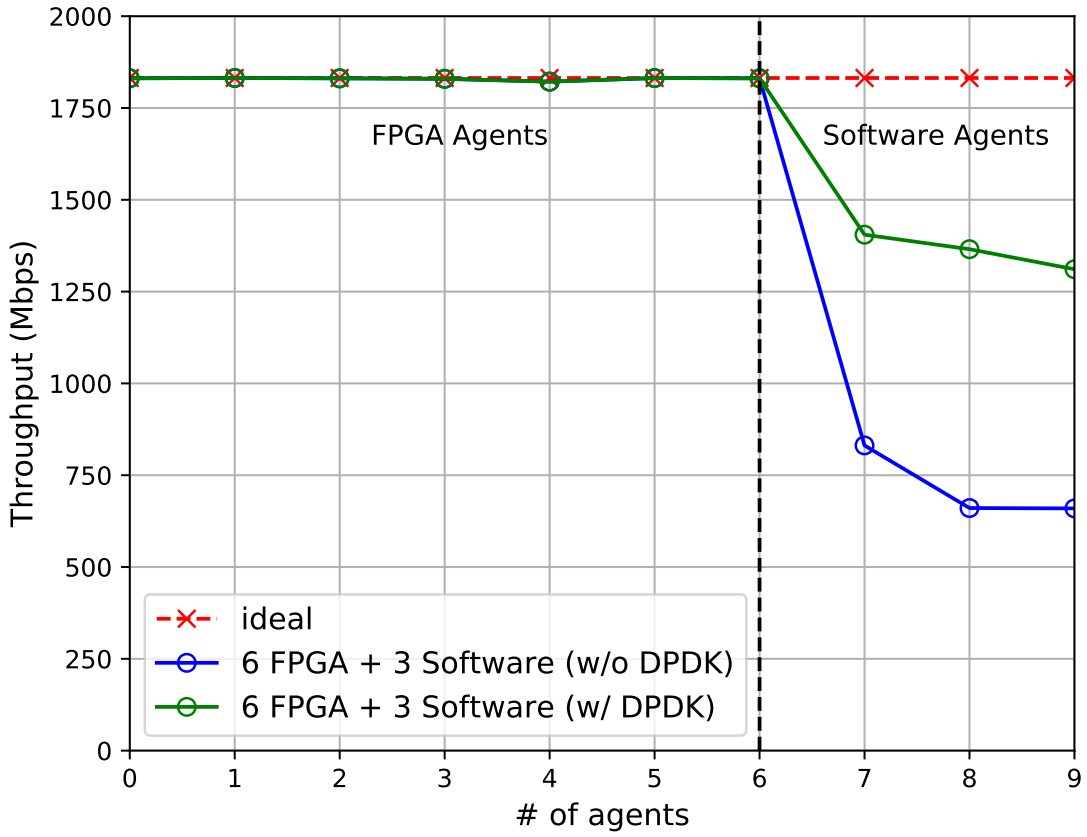

Figure 5.8: Scalability of agents implemented with up to 2 FPGAs (3 agents each) and 3 workstations (3 non-DPDK agents or 3 DPDK-based agents). Six sessions were used for this experiment.

5.5.2 Scalability test

The ability of software agents implemented with/without DPDK, and FPGA agents to forward packets for a scaled set of middleboxes in a service chain was tested. Six QUIC sessions were run simultaneously on the same service chain. To evaluate scalability, overall session throughput on the service chain was measured using an increasingly large set of hardware and software middleboxes. Like the session initiation test, two versions of software agents were implemented. Software agents implemented without using DPDK were run in containers installed in three workstations (one container per workstation). DPDK-based software agents were run directly on the host OS for three workstations. Two DE5 boards were used to implement FPGA

versions (three agents per FPGA). Figure 5.8 shows the scalability of our agent-based approach working on heterogeneous middleboxes with up to nine agents deployed in the service chain. The first six middleboxes used in the chain are FPGA-based, which shows a higher throughput value on the left side of the graph.

As agents are scaled up to a count of six FPGA agents, shown as the green line on the left side in Figure 5.8, system performance versus the ideal case remains nearly identical indicating the good scalability of our agents implemented as FPGA circuits. The pipelined structure of the FPGA agent implementation ensures the high throughput of FPGA agents in this test. The throughput of the FPGA agent is constrained by the overall session throughput on the service chain. The green and blue lines on the right side of the graph show that software agents introduce throughput degradation to the service chain. The serial nature of the processor limits the performance of software agents during the service chain operation. For software agents implemented without DPDK, the various levels of the network stack cause further system performance slowdown.

5.5.3 Dynamic reconfiguration

A significant weakness of the central controller approach for service chaining is the inability of the controller to manage the chain across network boundaries. Our agent-based approach implements inter-domain service chaining by deploying agents in multiple subnetworks and connecting the middleboxes and end-hosts distributed in different subnetworks by establishing subsessions across network boundaries. In a final experiment, QUIC clients and servers, located in different subnetworks, and FPGA and processor-based middleboxes were connected through a level-3 router (Figure 5.9). Each subnetwork has its own IP address domain, as shown in the figure.

QUIC is an encrypted-by-default protocol. Since the QUIC protocol implements congestion control algorithms, loss recovery, and the packet encryption/decryption

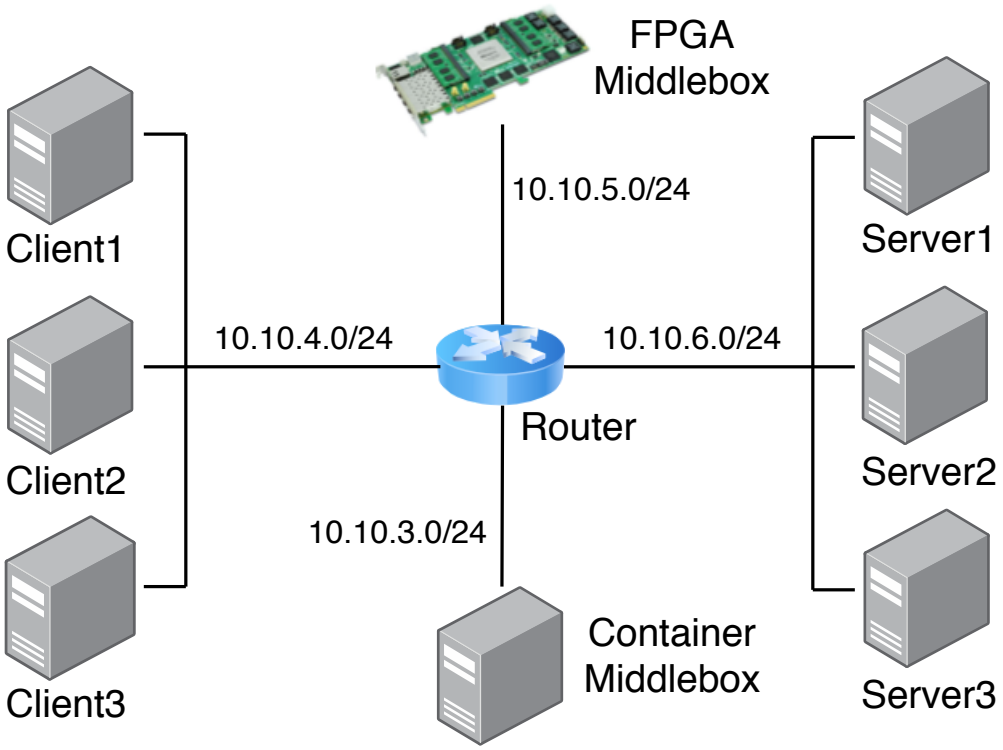

Figure 5.9: Testbed topolgy for the evaluation of the reconfiguration experiment

Table 5.2: Throughput and latency comparison of software and FPGA firewall implementations under the traffic of a total three QUIC sessions

	Software (w/o DPDK)	Software (w/ DPDK)	FPGA
Throughput (Mbps)	796	1157	1270
Latency (us)	115.6	85.8	3.7

function in the user space at end-hosts, rather than the kernel space, the bandwidth of the QUIC session is limited. Previous studies have tested QUIC performance under 120 Mbps bandwidth [42]. In this experiment, we use *ngtcp2* to generate QUIC sessions. In order to obtain the maximum QUIC session bandwidth for testing, three QUIC clients continuously send data packets to three QUIC servers with the maximum packet size (1400 bytes per packet). The bandwidth for three QUIC sessions is 1270 Mbps.

This experiment assesses an application scenario in which clients are located on a local network and servers are in the cloud. FPGA and processor-based middleboxes are located in the client-server path. Initially, a service chain including three QUIC client-middlebox-server paths are established by agents. A processor-based container is established by the agent and used as a firewall middlebox for three QUIC sessions. As shown in Table 5.2, the FPGA-based firewall has significantly better throughput and latency than software versions.

During service chain operation, agents are used to modify the service chain three times, each time migrating one QUIC session from the processor-based middlebox to the FPGA-based middlebox. The agents on the client and server act as left and right anchors, respectively. For a QUIC session between one client-server pair, the session traffic is redirected to pass through the router and the FPGA middlebox rather than the processor-based middlebox.

Figure 5.10 shows the QUIC throughput of the container-based and FPGA-based middleboxes over time, including three session migrations to the FPGA. The time series represents throughput measures at one-hundred-millisecond intervals. Three QUIC sessions initially pass through the same container middlebox with a total traffic rate less than 1000 Mbps (the software firewall throughput limitation shown in Table 5.2). Service chain modification occurs at 5, 10, and 15 seconds after the start of the experiment. The left anchor exchanges control messages with the right anchor at each modification time point to sequentially redirect a QUIC session from the container middlebox to the FPGA middlebox. When the first QUIC session is redirected, the control message sent by the left anchor triggers the agent on the FPGA middlebox to configure FPGA firewall circuit as a packet processor in a PR region. Partial FPGA reconfiguration requires about 196 ms and requires the loading of a firewall partial bitstream from flash. The agent stays active during the partial reconfiguration of the firewall bitstream. The time interval between the two yellow vertical dashed

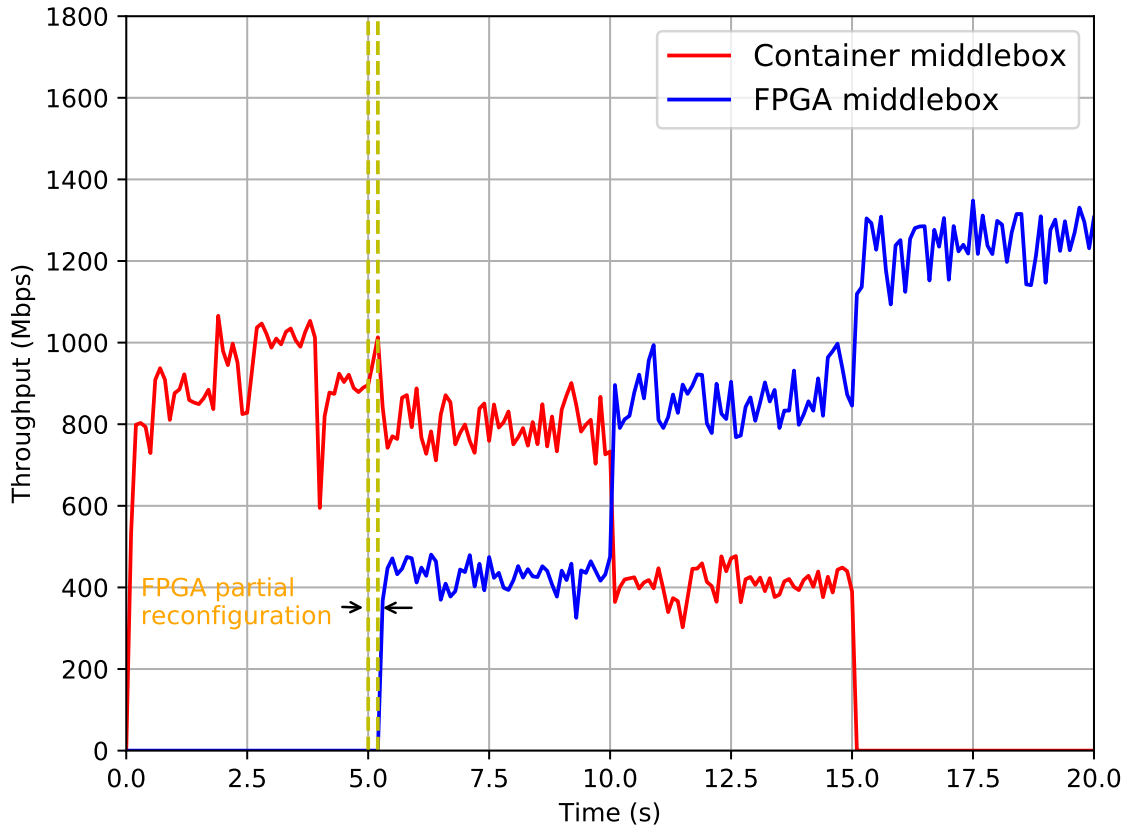

Figure 5.10: Throughput of three QUIC sessions on processor and FPGA middleboxes. Initially, all three sessions are implemented on processors (left). Service chain modification is performed every 5 seconds to migrate a QUIC session from a processor (container) middlebox to the FPGA middlebox.

lines in Figure 5.10 indicates the partial reconfiguration process of the firewall on the FPGA middlebox. After all three QUIC sessions have been redirected to pass through the FPGA middlebox, the overall throughput (blue line on the right) is significantly higher than when all sessions passed through the container-based middleboxes (red line on the left).

5.6 Conclusion

In this chapter, we have described a new session-level approach for inter-domain service chaining of heterogeneous middleboxes. Distributed agents implemented on

field-programmable gate arrays and microprocessors are used to steer packets of QUIC sessions through designated service chains. We implement a policy server to initialize service chaining policies and trigger service chain reconfiguration. Our results show that both the software and hardware agent implementations have good scalability, but the hardware agent exhibits better throughput during packet processing and lower latency for session initiation. The migration of service chains for three sessions across multiple subnetworks is demonstrated using agents running on end-hosts and FPGA- and processor-based middleboxes.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The work in this thesis examined solutions in the research area of network function virtualization (NFV). In the first part of the document, we described a new heterogeneous hardware-software approach to NFV construction, and addressed challenges arising from the adoption of FPGAs in NFV, such as dynamic reconfiguration, state sharing and resource management. In the second part, we explored a new session-level approach for inter-domain service chaining of FPGA- and processor-based middleboxes, and dynamic reconfiguration of service chains for ongoing sessions.

6.1 Summary of Contributions

For our first contribution, we demonstrated techniques that integrate FPGAs in *heterogeneous* network function virtualization platforms. Our system addresses state sharing issues in previous FPGA-based NFV systems with the aid of a global *coordinator* to collect and distribute global state information across both FPGA- and microprocessor-based middleboxes. With the help of the state sharing mechanism offered by the coordinator, customer-defined VNF instances can be easily migrated from microprocessors to FPGAs and vice versa as the network environment changes. Migration to FPGAs is supported with partial FPGA reconfiguration. A customized allocation and scheduling algorithm has been developed to dynamically evaluate heterogeneous middlebox deployment based on global state information and middlebox usage. Our evaluation demonstrates the scalability and hardware reconfigurability of the hybrid system. We show that our deployment algorithm can successfully real-

locate FPGA and microprocessor resources in a fraction of a second in response to changes in network flow capacity and network security threats including intrusion.

Second, we developed a new session-level approach that implements NFV distributed agents to establish and dynamically reconfigure service chains across network boundaries without relying on a centralized controller. Agents support the deployment on both FPGA- and processor-based middleboxes. During service chain reconfiguration, the agent can trigger the reconfiguration of the middlebox function on the service chain by either partially reconfiguring a VNF implementation on the FPGA-based middlebox or starting/terminating a VNF process on the processor-based middlebox. We evaluated our approach by establishing service chains for QUIC sessions. Our evaluation demonstrated that our session-level approach can successfully establish and reconfigure inter-domain service chains for individual QUIC sessions. The distributed agents implemented on FPGAs show better performance in terms of throughput and latency overhead compared with software implementations.

6.2 Future Work

The research presented in this dissertation provides guidelines for future work in heterogeneous network function virtualization and distributed service function chaining.

Larger and more diverse VNFs: To evaluate our CoNFV system, we implemented four types of VNFs on FPGAs, including an SQLi detector, a DDoS detector, NAT, and a packet firewall. Larger and more complex network functions can also benefit from FPGA implementation, such as packet classification functions [13], web data compression/decompression functions [58], etc. FPGA support to improve software VNF performance can also be explored. Compared with complete VNF implementations on an FPGA, an FPGA could be used to accelerate performance bottlenecks in

the VNF software implementation [44]. This approach could reduce the difficulty of hardware implementation and make more effective use of FPGA resources.

Partially reconfigurable regions on a larger FPGA: With the increase in density, decrease in power consumption, and improvement in speed, today's FPGAs fill a completely different set of design needs from those of the past. One FPGA has enough resources to implement different network functions. When multiple network functions are implemented on the same FPGA, partitioning an FPGA into partially reconfigurable regions for function placement becomes a problem to be solved [83]. Security issues, such as side-channel attacks [67], that come with resource sharing are also worth studying.

Advanced NFV resource scheduling and allocation algorithms: Resource scheduling and allocation are important topics in hybrid network function virtualization. NFV resource scheduling and allocation are NP-hard optimization problems [29]. Current solutions to solve these problems include linear programming algorithms [26, 54], recursive greedy algorithms [68], and metaheuristic-based solutions [53], etc. With the development of machine learning technology, deep reinforcement learning provides a possibly more effective solution. The use of machine learning for NFV is a research area worth exploring.

Applicability of the session-level approach to other network protocols: In this thesis, we evaluated the use of our session-level approach to establish and dynamically reconfigure a service function chain of heterogeneous middleboxes for QUIC sessions. The session-level approach can also be applied to sessions established by other network protocols, such as TCP-based protocols. A general agent implementation suitable for both TCP- and UDP-based protocol sessions could be developed in the future.

BIBLIOGRAPHY

[1] A. AbdelSalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, and L. Veltri, "Implementation of Virtual Network Function Chaining through Segment Routing in a Linux-Based NFV Infrastructure," in *Proceedings of the 2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–5.

[2] Altera, *FPGA Architecture White Paper*, July 2006.

[3] ——, *Using MicroC/OS-II RTOS with the Nios II Processor*, May 2011.

[4] Altera, *Using the NicheStack TCP/IP Stack*, June 2011.

[5] Altera, *Design Planning for Partial Reconfiguration*, November 2013.

[6] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming Slick Network Functions," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2015, pp. 1–13.

[7] Z. K. Baker and V. K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs," in *International Conference on Field Programmable Logic and Applications*. Springer, 2004, pp. 311–321.

[8] D. Barrett and G. Kipper, *Virtualization and Forensics: A Digital Forensic Investigators Guide to Virtual Environments*. Syngress, 2010.

[9] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 109–116.

[10] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1–12, 2007.

[11] P. Dash, *Getting Started with Oracle VM VirtualBox*. Packt Publishing Ltd, 2013.

[12] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," in *Proceedings of the 11th Symposium on High Performance Interconnects*. IEEE, 2003, pp. 44–51.

[13] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast Packet Classification Using Bloom Filters," in *2006 Symposium on Architecture For Networking And Communications Systems*. IEEE, 2006, pp. 61–70.

[14] ETSI, *Network Functions Virtualisation - Introductory White Paper*, October 2012.

[15] ETSI, *Network Function Virtualisation (NFV); Architectural Framework*, October 2013.

[16] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 543–546.

[17] C. Filsfils, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, "IPv6 Segment Routing Header (SRH)," in *RFC 8754*. IETF, 2018.

[18] R. W. Floyd and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," in *21st Annual Symposium on Foundations of Computer Science*. IEEE, 1980, pp. 260–269.

[19] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu, "Openanfv: Accelerating network function virtualization with a consolidated framework in openstack," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 353–354, 2014.

[20] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella, "Stratos: A Network-Aware Orchestration Layer for Virtual Middleboxes in Clouds," *arXiv preprint arXiv:1305.0209*, 2013.

[21] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward Software-Defined Middlebox Networking," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, 2012, pp. 7–12.

[22] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2014.

[23] H. GholamHosseini and K. Li, "Implementation of Transient Signal Detection Algorithms on FPGA," *International Journal of Computer Applications*, vol. 975, p. 8887, 2012.

[24] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," in *International Conference on Field Programmable Logic and Applications*. Springer, 2002, pp. 404–413.

[25] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 68–73, 2008.

[26] A. Gupta, M. F. Habib, P. Chowdhury, M. Tornatore, and B. Mukherjee, "On Service Chaining using Virtual Network Functions in Network-enabled Cloud Systems," in *2015 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS)*. IEEE, 2015, pp. 1–3.

[27] J. Halpern, C. Pignataro *et al.*, "Service Function Chaining (SFC) Architecture," in *RFC 7665*. IETF, 2015.

[28] A. Hassidim, D. Raz, M. Segalov, and A. Shaqed, "Network Utilization: the Flow View," in *2013 Proceedings IEEE INFOCOM*. IEEE, 2013, pp. 1429–1437.

[29] J. G. Herrera and J. F. Botero, "Resource Allocation in NFV: A Comprehensive Survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[30] Intel, "FPGA Design Software - Intel Quartus Prime." [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

[31] Intel, "Nios II Processors." [Online]. Available: https://www.intel.com/content/\www/us/en/products/programmable/processor/nios-ii.html

[32] Intel, "Platform Designer - Intel's System Integration Tool." [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/features/qts-platform-designer.html

[33] Intel, *Avalon Interface Specifications*, September 2018.

[34] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport," *Internet Engineering Task Force*, 2019.

[35] W. Jiang and V. K. Prasanna, "Scalable Packet Classification on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 9, pp. 1668–1680, 2011.

[36] C. Kachris, G. Sirakoulis, and D. Soudris, "Network Function Virtualization based on FPGAs: A Framework for all-Programmable network devices," *arXiv preprint arXiv:1406.0309*, 2014.

[37] K. Kolyshkin, "Virtualization in linux," *White paper, OpenVZ*, vol. 3, p. 39, 2006.

[38] S. Kulkarni, M. Arumaithurai, K. Ramakrishnan, and X. Fu, "Neo-NSH: Towards Scalable and Efficient Dynamic Service Function Chaining of Elastic Network Functions," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE, 2017, pp. 308–312.

[39] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[40] I. Kuon, R. Tessier, and J. Rose, *FPGA Architecture: Survey and Challenges*. Now Publishers Inc, 2008.

[41] J. A. Landis, T. V. Powderly, R. Subrahmanian, A. Puthiyaparambil, and J. R. Hunter Jr, "Computer System Para-Virtualization using a Hypervisor that is Implemented in a Partition of the Host System," July 2011, uS Patent 7,984,108.

[42] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 183–196.

[43] H. Lee, "Virtualization Basics: Understanding Techniques and Fundamentals," *School of Informatics and Computing*, 2014.

[44] X. Li, X. Wang, F. Liu, and H. Xu, "DHL: Enabling Flexible Software Network Functions with FPGA Acceleration," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1–11.

[45] T. Lin, N. Tarafdar, B. Park, P. Chow, and A. Leon-Garcia, "Enabling Network Function Virtualization over Heterogeneous Resources," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2017, pp. 58–63.

[46] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, "Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware," in *Proceedings of the Military and Aerospace Programmable Logic Device Conference*, 2003.

[47] K. Lu, D. Wu, J. Fan, S. Todorovic, and A. Nucci, "Robust and Efficient Detection of DDoS Attacks for Large-Scale Internet," *Computer Networks*, vol. 51, no. 18, pp. 5036–5056, 2007.

[48] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 459–473.

[49] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[50] P. Megyesi, Z. Krämer, and S. Molnár, "How Quick is QUIC?" in *2016 IEEE International Conference on Communications (ICC)*. IEEE, 2016, pp. 1–6.

[51] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[52] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-art and Research Challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[53] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and Evaluation of Algorithms for Mapping and Scheduling of Virtual Network Functions," in *Proceedings of the 2015 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–9.

[54] H. Moens and F. De Turck, "VNF-P: A Model for Efficient Placement of Virtualized Network Functions," in *10th International Conference on Network and Service Management (CNSM) and Workshop*. IEEE, 2014, pp. 418–423.

[55] L. Nobach, "Seamless Flexibility in High-Performance Network Functions Virtualization," Ph.D. dissertation, Technische Universität, 2018.

[56] L. Nobach and D. Hausheer, "Open, Elastic Provisioning of Hardware Acceleration in NFV Environments," in *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE, 2015, pp. 1–5.

[57] L. Nobach, B. Rudolph, and D. Hausheer, "Benefits of Conditional FPGA Provisioning for Virtualized Network Functions," in *2017 International Conference on Networked Systems (NetSys)*. IEEE, 2017, pp. 1–6.

[58] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "FPGA Implementation of GZIP Compression and Decompression for IDC Services," in *2010 International Conference on Field-Programmable Technology*. IEEE, 2010, pp. 265–268.

[59] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A Framework for NFV Applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 121–136.

[60] S. Pontarelli, G. Bianchi, and S. Teofili, "Traffic-Aware Design of a High-Speed FPGA Network Intrusion Detection System," *IEEE Transactions on Computers*, vol. 62, no. 11, pp. 2322–2334, 2012.

[61] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 13–24.

[62] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013, pp. 27–38.

[63] P. Quinn, U. Elzur, and C. Pignataro, "Network Service Header (NSH)," in *RFC 8300*. IETF, 2018.

[64] P. Quinn and J. Guichard, "Service Function Chaining: Creating a Service Plane via Network Service Headers," *Computer*, vol. 47, no. 11, pp. 38–44, 2014.

[65] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 227–240.

[66] M. Ramakrishna, E. Fu, and E. Bahcekapili, "A Performance Study of Hashing Functions for Hardware Applications," in *Proceedings of the International Conference on Computing and Information*, 1994, pp. 1621–1636.

[67] C. Ramesh, S. B. Patil, S. N. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier, "FPGA Side Channel Attacks without Physical Access," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 45–52.

[68] R. Riggio, T. Rasheed, and R. Narayanan, "Virtual Network Functions Orchestration in Enterprise WLANs," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 1220–1225.

[69] X. Shao, L. Gao, and H. Zhang, "CoGS: Enabling Distributed Network Functions with Global States," in *Proceedings of the 2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–9.

[70] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[71] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, M. Tyson *et al.*, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *20th Annual Network & Distributed System Security Symposium*. NDSS, 2013.

[72] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2001, pp. 227–238.

[73] H. Song and J. W. Lockwood, "Efficient Packet Classification for Network Intrusion Detection using FPGA," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, 2005, pp. 238–245.

[74] P. Srisuresh, "Network address translation - protocol translation (nat-pt)," *RFC 2766*, 2000.

[75] C. Sun, J. Bi, Z. Zheng, and H. Hu, "HYPER: A Hybrid High-Performance Framework for Network Function Virtualization," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2490–2500, 2017.

[76] T. Taleb and Y. Hadjadj-Aoul, "QoS2: a Framework for Integrating Quality of Security with Quality of Service," *Security and Communication Networks*, vol. 5, no. 12, pp. 1462–1470, 2012.

[77] T. Taleb, A. Ksentini, and B. Sericola, "On Service Resilience in Cloud-Native 5G Mobile Systems," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 483–496, 2016.

[78] N. Tarafdar, T. Lin, N. Eskandari, D. Lion, A. Leon-Garcia, and P. Chow, "Heterogeneous Virtualized Network Function Framework for the Data Center," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–8.

[79] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 237–246.

[80] Terasic, *DE5-Net FPGA Development Kit User Manual*, June 2018.

[81] M. Thomson and S. Turner, "Using transport layer security (tls) to secure quic," *Internet Engineering Task Force*, 2017.

[82] D. Unnikrishnan, R. Vadlamani, Y. Liao, J. Crenne, L. Gao, and R. Tessier, "Reconfigurable Data Planes for Scalable Network Virtualization," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2476–2488, 2012.

[83] K. Vipin and S. A. Fahmy, "FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–39, 2018.

[84] G. Wang and T. E. Ng, "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," in *2010 Proceedings IEEE INFOCOM*. IEEE, 2010, pp. 1–9.

[85] A. Wion, M. Bouet, L. Iannone, and V. Conan, "Distributed Function Chaining with Anycast Routing," in *Proceedings of the 2019 ACM Symposium on SDN Research*, 2019, pp. 91–97.

[86] Xilinx, "Vivado Design Suite." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[87] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework," RFC 3746, April, Tech. Rep., 2004.

[88] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact Architecture for High-Throughput Regular Expression Matching on FPGA," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 30–39.

[89] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 29–42.

[90] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic Service Chaining with Dysco," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 57–70.

[91] X. Zhang, X. Shao, G. Provelengios, N. K. Dumpala, L. Gao, and R. Tessier, "Scalable Network Function Virtualization for Heterogeneous Middleboxes," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 219–226.