CoNFV: A Heterogeneous Platform for Scalable Network Function Virtualization

XUZHI ZHANG, XIAOZHE SHAO, GEORGE PROVELENGIOS, NAVEEN KUMAR DUMPALA, LIXIN GAO, and RUSSELL TESSIER, University of Massachusetts Amherst, USA

Network function virtualization (NFV) is a powerful networking approach that leverages computing resources to perform a time-varying set of network processing functions. Although microprocessors can be used for this purpose, their performance limitations and lack of specialization present implementation challenges. In this manuscript, we describe a new heterogeneous hardware-software NFV platform called CoNFV that provides scalability and programmability while supporting significant hardware-level parallelism and reconfiguration. Our computing platform takes advantage of both field-programmable gate arrays (FPGAs) and microprocessors to implement numerous virtual network functions (VNF) that can be dynamically customized to specific network flow needs. The most distinctive feature of our system 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 and scheduling algorithm dynamically assesses resource deployments as network flows and conditions are updated. We show that our deployment algorithm can successfully reallocate FPGA and microprocessor resources in a fraction of a second in response to changes in network flow capacity and network security threats including intrusion.

CCS Concepts: • **Networks** → **Middle boxes / network appliances**; **Network management**; *Network resources allocation*; • **Hardware** → **Reconfigurable logic applications**; *Hardware accelerators.*

Additional Key Words and Phrases: Network function virtualization, Coordinator, Heterogeneous middleboxes, FPGA, Performance-aware VNF deployment.

ACM Reference Format:
Xuzhi Zhang, Xiaozhe Shao, George Provelengios, Naveen Kumar Dumpala, Lixin Gao, and Russell Tessier. 2020. CoNFV: A Heterogeneous Platform for Scalable Network Function Virtualization. *ACM Trans. Reconfig. Technol. Syst.* 37, 4, Article 111 (August 2020), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As the Internet has grown, the need for increasingly complex operations in network infrastructure has led to the implementation of numerous embedded network functions. These functions serve an array of purposes ranging from general-purpose computation to security to improving network efficiency. For example, firewalls and intrusion detection functions provide security while network address translators (NATs), load balancers, packet classifiers, and proxy caches support efficient

This work was supported by the National Science Foundation under grant CNS-1525836. We thank Intel for the donation of the DE5 boards.
Authors' address: Xuzhi Zhang; Xiaozhe Shao; George Provelengios; Naveen Kumar Dumpala; Lixin Gao; Russell Tessier, University of Massachusetts Amherst, Department of Electrical and Computer Engineering, Amherst, MA, 01003, USA.

1936-7406/2020/8-ART111 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

network operation in enterpise, cloud and data center environments. Although application-specific integrated circuits (ASICs) were initially the network function implementation mechanism of choice [22], the need to dynamically update and swap functions has led to increased interest in network function virtualization (NFV). To support NFV, programmable compute platforms based on microprocessors and field-programmable gate arrays (FPGAs) respond to rapid in-network updates as function needs change.

The coordination of heterogeneous network function platforms in a distributed environment is a challenge. Microprocessors are straightforward to program and allow for fast network interfacing using toolsets such as the Data Plane Development Kit (DPDK)[1]. However, they are inherently sequential and specialized network functions often require significant run time. To augment these common NFV components, several researchers [13, 20] have examined implementing complex NFV functions in FPGAs and managing their functions over time through dynamic reconfiguration. Although effective, these systems generally take a localized view of network state information in individual middleboxes rather than a global perspective that simultaneously considers state from multiple middleboxes to enhance NFV deployment.

In this manuscript, we describe CoNFV, a network function platform based on FPGAs, microprocessors, and supporting software. CoNFV makes three significant extensions beyond existing NFV platforms:

(1) CoNVF uses a global *coordinator* to collect and distribute global state information across middleboxes. Unlike earlier systems, state information is used to dynamically allocate virtual network functions (VNFs) to both FPGA and microprocessor-based middleboxes.
(2) In our system, VNFs can be automatically migrated from microprocessors to FPGAs and vice versa. Migration to FPGAs is supported with partial FPGA reconfiguration.
(3) A customized allocation and scheduling algorithm has been developed to dynamically evaluate heterogeneous middlebox deployment based on global state information and middlebox usage.

To demonstrate our system, a library of FPGA-based and software modules has been implemented and tested for the following VNFs: specialized SQL injection (SQLi) attack detection, distributed denial-of-service (DDoS) detection, and network address translation (NAT). These function modules, implemented in either FPGA hardware or processor software, are swapped into middleboxes on demand. In some cases, multiple microprocessors are grouped together to achieve needed throughput, latency, and computer performance levels. Our allocation and scheduling tool periodically identifies changes in required VNF deployment, assembles the components from available libraries, and dynamically reconfigures the component FPGAs and virtual machines (VMs) that implement the network functions. Our prototype network function virtualization environment is assessed using Intel DE5 FPGA boards, microprocessor-based VirtualBox middleboxes, and a 10 Gigabit-per-second (Gbps) software defined networking (SDN) network switch. The system is shown to be scalable both in middlebox count and in state request rate to the coordinator.

The remainder of this manuscript is structured as follows. Section 2 presents NFV and the use of FPGAs in networking functions. In Section 3, we present our scalable hardware and software system. Implementation details for the NFV framework and support for dynamic reconfiguration are provided in Sections 4 and 5. Section 6 provides details on data plane traffic management. Our allocation and scheduling algorithms are outlined in Section 7 and the experimental methodology is detailed in Section 8. The benefits of our dynamic reconfiguration approach are quantified in Section 9. Section 10 concludes the paper and offers directions for future work.

[1] dpdk.org

2 RELATED WORK

To provide context for our new NFV system that manages global state information and uses it for VNF allocation and scheduling, we review the basics of NFV and contrast our approach with previous efforts.

2.1 Network Function Virtualization

NFV has been a focus of network builders for nearly ten years, as network hardware and programming environments have matured. The approach supports the virtualization of network appliance (middlebox) operations into programmable components that may be chained to provide a range of networking services. Although NFV is most often considered in the context of data centers [21], any extended networking environment is a candidate platform. Typical network functions include border controllers such as firewalls, load balancers, and wide-area network (WAN) accelerators that protect a network. With the advance of server virtualization technology, it became possible to decompose traditional network border controller functions into virtual machines running different software, and into reconfigurable FPGA components. When designing and developing the software and FPGA circuits that provide virtual network functions, it is possible to break operations into components and package those components into one or more functions. 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 WAN accelerator that is installed in the same or a different middlebox.

2.2 Reconfigurable Network Functions

Reconfigurable logic provides an ideal platform for network functions due to the parallelism, specialization, and adaptability offered by FPGA devices [23]. These characteristics match well with the multi-Gbps 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 [2] and data centers [17], their use in network and application processing will continue to grow. As an example of the efficiency of FPGAs versus microprocessors for NFV we contrast the performance of DPDK, which gives the CPU low-level access to network interface card drivers, to a Stratix V FPGA implementation of regular expression matching (REME)[2]. Figure 1 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.

2.2.1 Need for NFV Global State. A number of microprocessor and FPGA-based platforms have been deployed for network applications involving performance improvement, load balancing, and security. A key aspect of these systems is the need to maintain state to evaluate the performance and threat level of the system over time. A wide range of FPGA-based network intrusion detection systems have been implemented using CAMs [6], shift and compare circuits [1, 16], and Bloom filters [3]. 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 network intrusion detection system (NIDS) ruleset [16]. Although DDoS prevention using FPGAs has received less attention than packet classification and NIDS,

[2]More details on the microprocessor and FPGA test framework are provided in Section 4.1

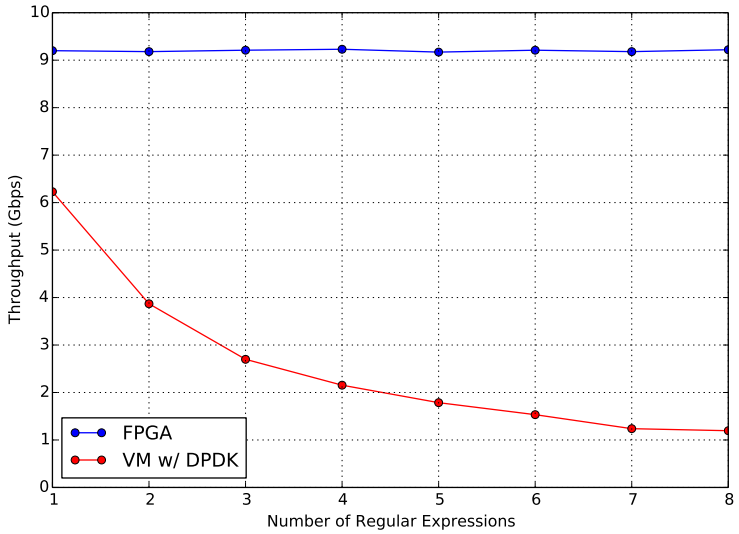

Fig. 1. Regular expression using DPDK: Processing throughput versus number of regular expressions

worm identification and matching circuits have been implemented in FPGAs that operate at line rates [10]. Ge et al. [4] use OpenStack with partial FPGA reconfiguration to support deep packet inspection and NAT. These NIDS, NAT, and DDoS applications locally buffer state information related to pattern matching hit rates and address translations. We show that the sharing of this information across middleboxes can improve overall system performance.

2.3 Network Function Virtualization Global State Management

Several previous works have explored the management of global VNF state information. Split/Merge [18], OpenNF [5] and E-State [15] manage global state in a distributed fashion. OpenNF delegates global state management to the control plane of the network function, an approach that is inappropriate for FPGAs. Split/Merge and E-State require each middlebox to maintain a partial supply of the global state. When a middlebox requests state, it must be fetched from another middlebox. Read global state operations incur considerable overhead (up to 20 ms [15]) due to peer-to-peer network communication versus the 0.2 ms per read operation required by our approach. Our centralized state approach also allows for larger state storage (1 million state entries) versus previous techniques (65,000 entries [14]). StatelessNF [8] isolates states from middleboxes and stores them in a separate repository connected via a high-speed network, in contrast to our approach that uses standard Ethernet links. The global coordinator in CoNFV also not only stores state information but uses the information to schedule and allocate NFV resources.

Several NFV management systems for FPGA-based VNFs have been developed in the past although none use global state collection and resource allocation based on global state. Kachris et al. [9] provide an analysis of the potential use of FPGA reconfiguration to dynamically support functions such as firewalls, packet parsing, IP lookup, deep packet inspection, and virus scanning. A comprehensive system [20] 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. Nobach et al. [13]

developed a system that can move functions between microprocessors and FPGAs on-demand. The Hyper system [19] uses a global mediator to hide middlebox resource heterogeneity when assigning VNFs to resources, although state information is not used. Our allocation algorithm operates about 5× faster than the one used in Hyper on a similarly-equipped workstation. Although these works allow for VNF migration, they do not directly address global state management and global state is not used to allocate NFV resources for the available functions.

2.4 Comparison to Previous Implementation

This paper substantially extends our previous conference publication that describes a heterogeneous network function virtualization system [26]. Our platform possesses the ability to share network state information and migrate network functions between heterogeneous middleboxes based on a changing network environment using partial FPGA reconfiguration and the deployment of VMs. New additions in this work include the deployment of a resource reassignment algorithm that considers resource constraints versus application needs. Additionally, an OpenFlow-based SDN switch has been integrated into our system to facilitate flow steering at much faster network speeds than our previous work. Finally, we show that the use of global state in a centralized coordinator facilitates allocation algorithm decision making and leads to rapid resource deployment in case of network intrusion and SQLi attacks.

3 SYSTEM DESIGN

3.1 System Overview

It is common for middleboxes positioned across a subnetwork to deploy distributed functions using commodity hardware, custom hardware, virtual machines (VM), 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 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 workload 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 2. Microprocessor- and FPGA-based middleboxes are distributed across the network. The middleboxes share state information through TCP connections to the CoNFV coordinator. As shown in Section 9, the coordinator is able to handle state for a scalable set of middleboxes, with minimal packet processing 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 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

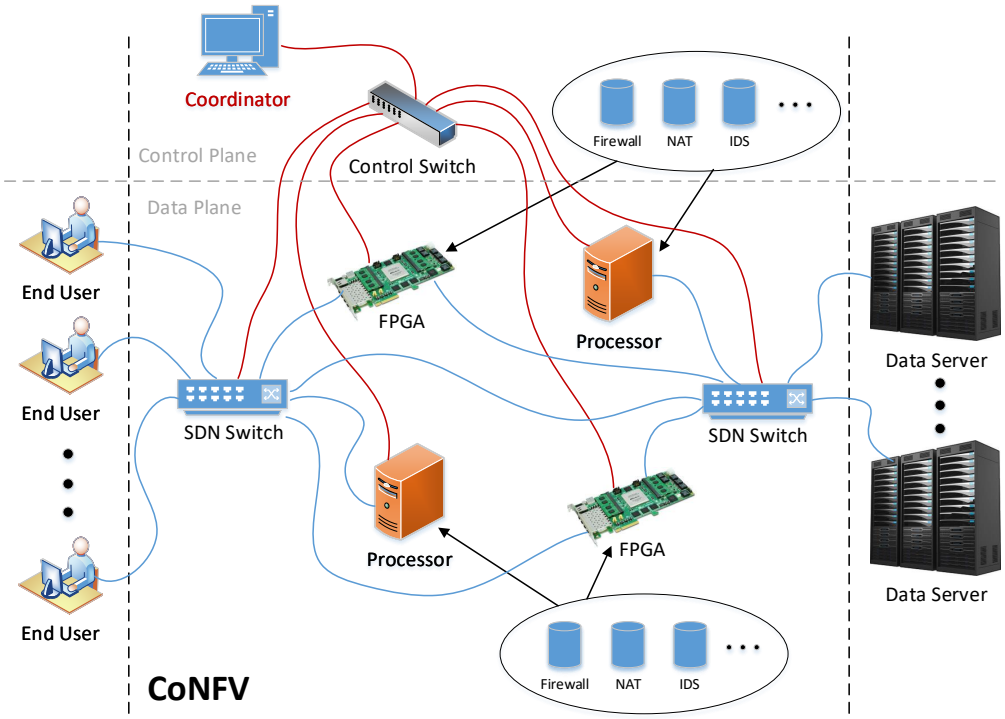

Fig. 2. Overview of the CoNFV configurable network function virtualization system using processor- and FPGA-based middleboxes.

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 evaluator. 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 reconfiguration for FPGA packet processors. A control *interface* allows for interaction with the coordinator. The specific functions of these modules for three applications is detailed in Section 8.

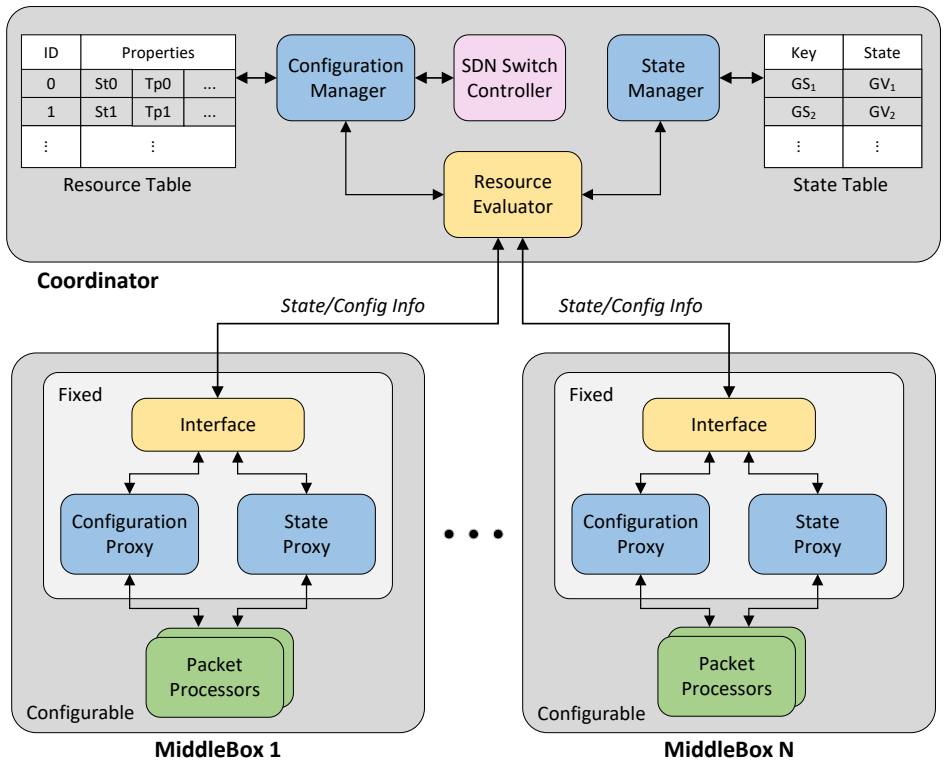

Fig. 3. Middlebox and global coordinator interaction. Middleboxes can be either processor- or FPGA-based.

3.2 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 8, we describe how CoNFV can be used to address 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 network address translation. When NAT receives the first packet of a flow it creates state which determines the translation from an external (IP, port) pair to an internal (IP, port) pair on the local subnetwork. This information must be shared across all middleboxes

performing NAT translation for the subnetwork to avoid (IP, 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.

3.3 Dynamic Resource Management

NFV resources must be managed using a global view of function deployment. In response to 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, virtual machine (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, 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 SDN switch. OpenFlow [12], a standard protocol for enabling SDN, is leveraged.

4 SYSTEM IMPLEMENTATION

4.1 Framework Overview

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, 4 GB). 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 Intel 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 FPGA- and processor-based middleboxes appears in Figure 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 (VMs) using VirtualBox[3] which allows full virtualization of a guest operating system. VirtualBox allows multiple isolated user spaces (virtual machines). Each virtual machine operates like a stand-alone server. Software middleboxes are effectively isolated from each other in separate VirtualBox containers 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]https://www.virtualbox.org/wiki/Downloads

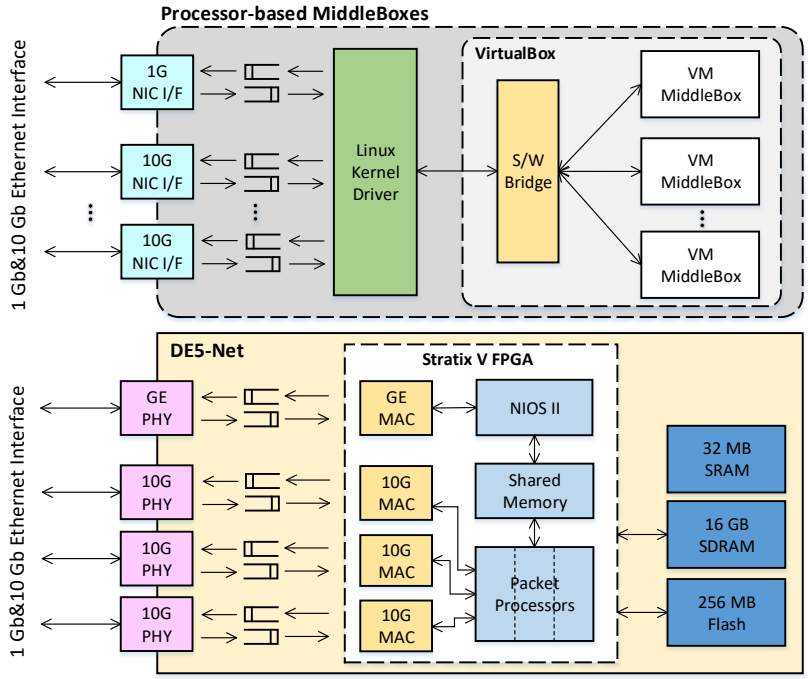

Fig. 4. High-level overview of processor- (top) and FPGA-based (bottom) middleboxes in CoNFV

4.2 Coordinator Implementation

4.2.1 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 middlebox's compute capabilities. Prior to use, a middlebox must be registered with the coordinator and the SDN switch must be configured to forward flows to the 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 source/destination processing capacities. Upon system startup, the configuration manager sets the source and destination ports 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.

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

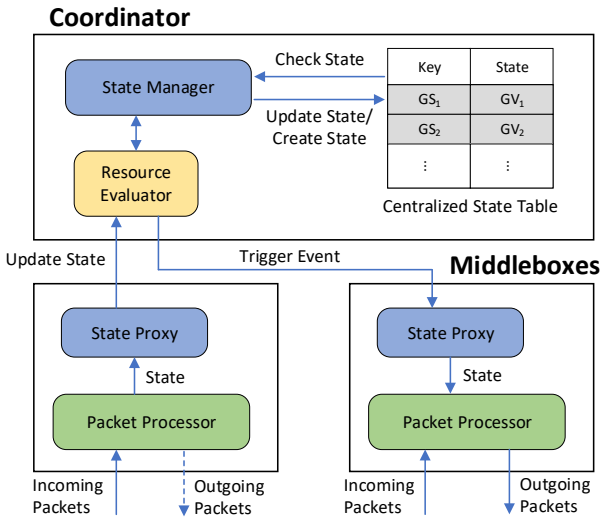

Fig. 5. Trigger state operations

used to record input and output port information and source and destination MAC addresses in the flow table. Port information is also forwarded to the SDN switch controller. OpenFlow protocol is used to communicate between the coordinator and the SDN switch.

4.2.2 Trigger States. 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 5 shows, when a packet comes into a middlebox, the packet processor inspects the packet and sends it out. According to the semantics of the network function, 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.

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.

4.2.3 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, port) pair to another (IP, 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 6 illustrates the procedure of

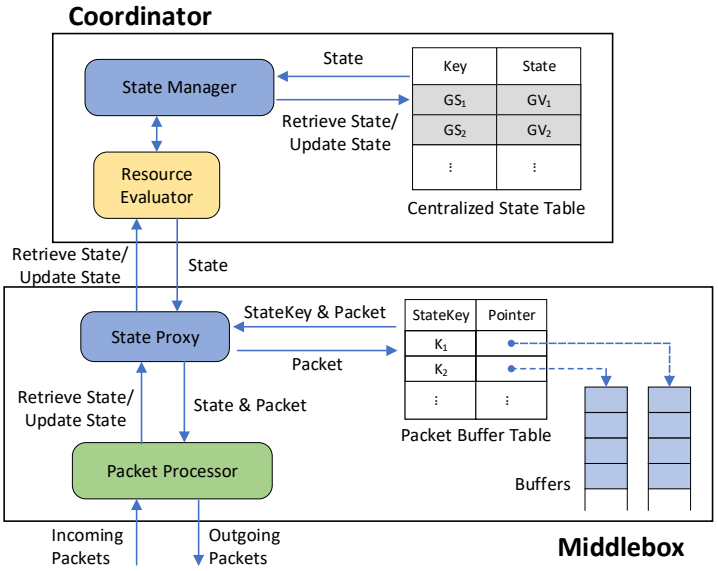

Fig. 6. State retrieval operations

asynchronous state operations. Each packet that incurs a 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. We did not encounter this scenario in our experimentation.

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.

4.2.4 Middlebox-Coordinator Configuration. Inside a middlebox, incoming packets are processed by a packet processor and state is manipulated by the state proxy. Both FPGA-based and processor-based state proxies provide defined interfaces for state update and retrieval. The state proxy provides APIs for updating or retrieving different kinds of states. Thus the packet processor is able to operate

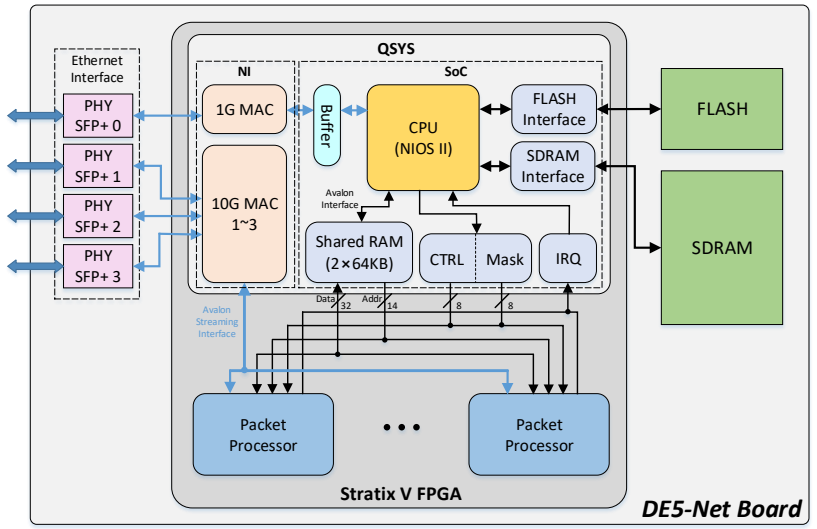

Fig. 7. Detailed FPGA implementation for multiple middlebox packet processors

on global states even if it does not know where the state is located. The state proxy interacts with the state manager in the coordinator using a collection of message types.

Messages from a middlebox to the coordinator include:

- **State Configure:** information about the format of state information.
- **Event Configure:** information about events that can be detected by the coordinator.
- **State Update:** information to update the global state stored in the coordinator
- **State Fetch:** a retrieval request for data from the coordinator.

Messages from the coordinator to a middlebox include:

- **Reply State:** retrieved state information from the coordinator.
- **Event Trigger:** a message which initiates a remediation effort at a middlebox (e.g. firewall configuration).

These messages are used to coordinate real-time middlebox response.

4.3 DE5 Middlebox and FPGA Module Library

A detailed view of the FPGA platform that can accommodate multiple packet processing middlebox functions is shown in Figure 7. A NIOS II 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. 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. In our FPGA platform, fixed FPGA circuitry includes two subsystems: the network interface (NI) subsystem and the Nios

II-based system on chip (SoC) subsystem (Figure 7). 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. 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 for boot code. Embedded NIOS II software was developed using MicroC/OS-II and the NicheStack TCP/IP Stack. 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 7 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, three middlebox functions for NAT, SQLi, and DDoS have been created with the interface, allowing for interoperability.

5 DYNAMIC RECONFIGURATION

As described in Section 7, 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 resources. 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. Based on the reconfiguration algorithm described in the next section, 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 initiate 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 new software VNF middlebox is started in an isolated VirtualBox. The creation of a new VNF function in an FPGA middlebox requires a series of steps. To support FPGA-based middlebox configuration, the FPGA is partially reconfigured. An effective approach for middlebox configuration is to swap one of the middlebox packet processor modules in Figure 7. Multiple configurations for the FPGA are available in on-board flash memory. Our partial reconfiguration approach requires the definition of a partial reconfiguration boundary that consists of the 207 interface signals on the module. These signals interface to lookup tables in the module which are

driven to a known value during reconfiguration. Partial reconfiguration is controlled by a state machine implemented in FPGA logic. During partial reconfiguration, the state machine calculates the memory address where the reconfiguration bitstream is stored and sends it to a dedicated FPGA circuit, which is triggered to retrieve new configuration information from flash and programs it into the FPGA configuration memory via a control block instantiated in the device.

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 switch 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.

As described in Section 7, traffic previously sent to a processor-based middlebox can be rerouted to an FPGA-based middlebox due to an increase in required VNF throughput. In our system, 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. A detailed example using middlebox functionality migration is described in Section 9.

6 DATA PLANE MANAGEMENT

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[4] with OpenFlow 1.3 protocol used to communicate with the NETGEAR ProSafe switch. By setting the flow table located in the SDN switch, the SDN controller configures the switch to steer the traffic flows. The table defines what actions should be applied to packets that enter the switch. The actions are saved as entries in the flow table.

Per-packet actions are specified via a series of match criteria in the table. A match criteria is defined over fields in the Ethernet header, IP header, TCP or UDP header, and potentially other header information. For example, criteria can match over source IP, destination IP, protocol, and input port fields. Actions can include packet dropping, sending a packet out of a particular port, sending the packet to the SDN controller, modifying packet header fields or performing other actions. If an incoming packet matches 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 a rule. The hard timeout field determines when a rule should be deleted after the rule is installed in 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 supports three different operations to manage input traffic flows, *forwarding*, *copying*, and *splitting*. By default, the SDN switch operates 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 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 this forwarding operation. Inspection functions (e.g. SQLi, DDoS) 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

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

of the input packets. The SDN controller modifies an 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 under guidance from the resource evaluator. Each output channel corresponds to an output port on the SDN switch which connects to a middlebox.

7 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.

7.1 Performance and Resource Management 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 unit (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.

7.2 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. Based on the results presented in Section 9, 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

Table 1. Notations used in VNF deployment algorithm

R	set of computation resources (CRs)
M	set of customer-defined VNF instances
λ_m	latency requirement of VNF instance $m \in M$
Λ_{rm}	processing latency of computation resource $r \in R$ working as VNF m
θ_m	throughput requirement of VNF instance $m \in M$
Θ_{rm}	expected throughput of computation resource $r \in R$ working as VNF m
ϕ_m	required processing capacity of VNF 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

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.

7.3 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 1 presents notation for VNF instance parameters and (1) represents the cost of using a CR r to implement VNF m.

$$c_{rm} = \frac{\lambda_m - \Lambda_{rm}}{\lambda_m} + \frac{\Theta_{rm} - \theta_m}{\theta_m} + \frac{\Phi_{rm} - \phi_m}{\phi_m} \tag{1}$$

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 (2). For the number of resources in our system, a full enumeration of possible assignments is possible to find the lowest cost match.

$$\min \sum_{r \in R, m \in M} c_{rm} \cdot g_{rm} \tag{2}$$

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.

```
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
```

Algorithm 1: High-level allocation algorithm

7.4 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 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 in 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 performance costs outlined in (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.

$$underprovisioned_{rm} \rightarrow (\Lambda_{rm} > \lambda_r)\ or\ (\Theta_{rm} < \theta_m)\ or\ (\Phi_{rm} < \phi_m) \tag{3}$$

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

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 (1)
    // In this case, the *move* is implementation in FPGA or adding another VM
    Move m into r
end
else 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 2: Algorithm to address underprovisioning.

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

Algorithm 3: Algorithm to address overprovisioning

monitors middlebox performance and global state to identify resource provisioning imbalances and global state triggers. As defined by (3), underprovisioning indicates that the current CR assignment for a VNF is insufficient in throughput, latency, or compute capacity.

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

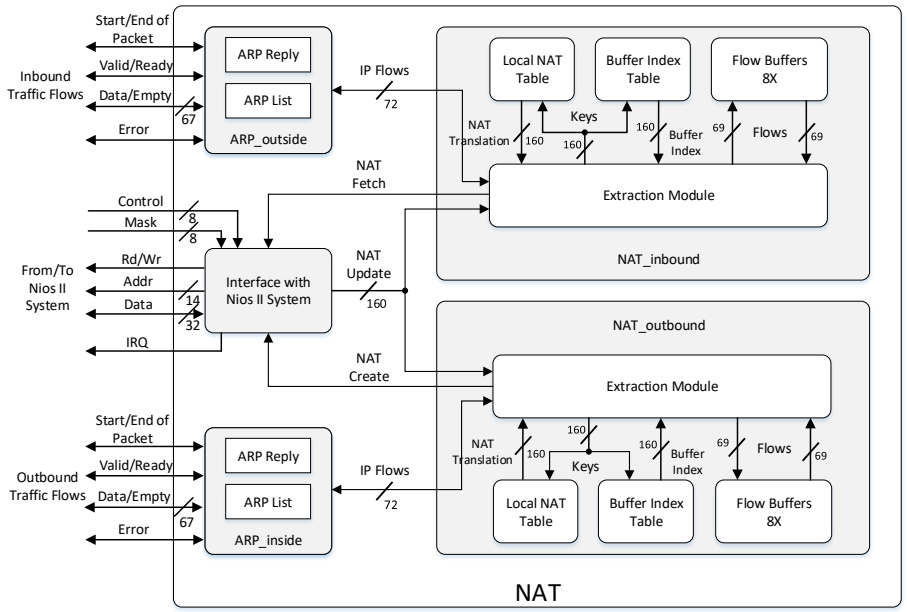

Fig. 8. FPGA middlebox implementation of NAT application

are n CRs, the calculation time complexity is $O(n)$ and allows the coordinator to quickly adapt to network dynamics.

8 MIDDLEBOX APPLICATIONS

For experimentation, three FPGA-based library modules which meet the requirements of the previous section were created and tested. The following discussion provides an overview of module operation and use.

8.1 NAT Implementation

As mentioned in Section 3.2, the NAT function converts an inside, local subnet (IP, port) pair to a public network (IP, port) pair. 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 8. The interface signals on the left of the figure match the packet processor interface signals shown in Figure 7. The extraction module extracts the source address, source port, destination address, destination port, and protocol information from the packet header to form a key. The address resolution protocol (ARP) module contains two ARP lists (caches) and reply modules. These blocks allow for the conversion of IP addresses to physical addresses. 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 × 69 bit 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. For each flow, the key is used as the input to the buffer index table and the local NAT translation table (implemented as a hash table) of depth 4,096 entries. If the translation is not found in the table, a NAT state fetch from the coordinator is initiated by the state proxy. The round trip time to fetch the translation from the coordinator is about 0.2 ms.

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. Inbound and outboard messages to/from the coordinator are 39 and 29 bytes in size, respectively. 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.

8.2 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[5]. 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 [25]. In the design, a REME can take at most 64 input characters. In our system, TCPreplay[6] is used to send packets ranging in size from 54 to 1514 bytes through SQLi detectors via 10 Gbps ports at varying speeds. Our regular expression matching engine has similarities to a previous implementation [25] except that a CAM is used to preprocess each character.

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.

8.3 DDoS Implementation

An additional module used to test our system was a distributed denial of service (DDoS) block, based on an earlier design [7, 11]. During a DDoS attack, the attacker floods a victim's network with SYN packets without sending the corresponding ACK packets. To detect a DDoS attack, the sequential change-point method described in [24] is used. 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 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.

[5]http://www.bro.org
[6]http://tcpreplay.synfin.net/

Table 2. 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 attack detector	25,708	15,172	2,755,072
DDoS attack detector	14,608	9,979	3,004,928
Firewall	10,571	12,609	3,490,560
NIOS II	2,034	1,806	1,137,408
Network interface	12,938	15,103	239,595
Memory interface	16,036	18,002	225,040
Shared memory	3	0	1,572,864
Available in FPGA	469,440	938,880	52,428,800

Table 3. 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
Firewall	-	9.35	-	1.20

8.4 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.

9 EXPERIMENTAL RESULTS

Experiments were performed in the lab using our PC and FPGA-board virtualization system described in detail in Section 4.1. Three Xeon processor-based workstations were sliced into four VirtualBox middleboxes each (unless otherwise noted), an Intel Duo processor-based machine was used as the coordinator, and two Stratix V based DE5 boards were used as FPGA processors. Each FPGA can support two computation regions. Experimental results for the performance, scalability, and reconfiguration of the system are described below.

Performance Test: The resource counts of the packet processor modules, the NIOS II and supporting components are shown in Table 2. Comparing the SQLi attack detector and the DDoS attack detector, the former requires more logic resources and defines the region size for partial reconfiguration. Any combination of two DDoS and SQLi cores or one NAT core can be included in

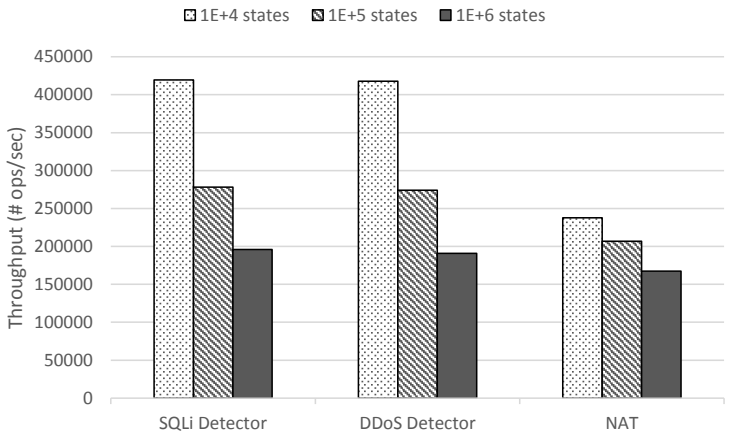

Fig. 9. Results of coordinator stress test. For each test, requests are made to the coordinator at the fastest rate supported by the network interface.

an FPGA at a time in the two partially reconfigurable regions. Each region includes 154,300 lookup tables (LUTs), 308,600 flip flops (FFs), and 17,315,000 block RAM (BRAM) bits. 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 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 8 were used. The test results are shown in Table 3. The reduced latency numbers and higher throughput for FPGA versus VM indicate the benefit of FPGA usage.

Scalability: Coordinator 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 interface to test its processing capabilities. Figure 9 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 figure, 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 9, 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.

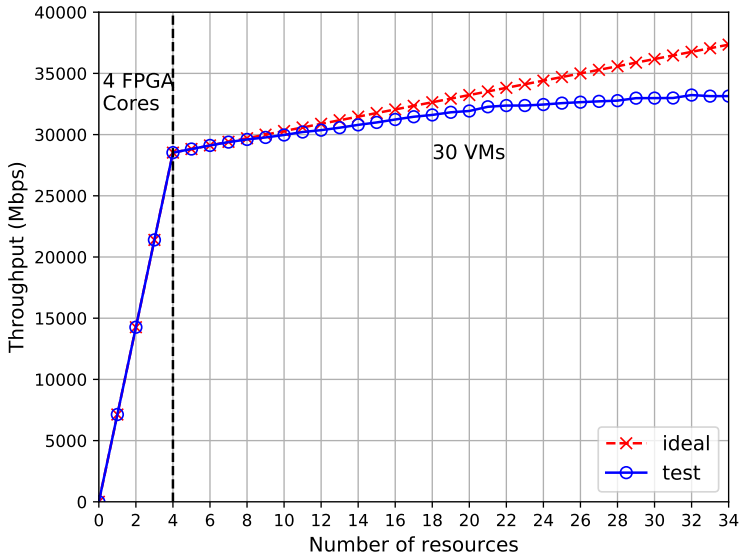

Fig. 10. Scalability of SQLi implemented with up to 2 FPGAs (2 CRs each) and 3 servers (30 virtual machines)

Scalability: Packet Processing 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 1,514 bytes. Figure 10 shows the scalability of our heterogeneous network system for between 1 and 34 middleboxes for the SQLi application. The first four middleboxes 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 cores), system performance versus the ideal case initially remains nearly identical indicating the 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 cores). 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.

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. A VNF instance can be either migrated between a VM and an FPGA packet processor, or deployed to multiple VMs to gain more computational power. The migration of a VNF instance needs three steps: reconfiguring the new hardware resource to support the VNF,

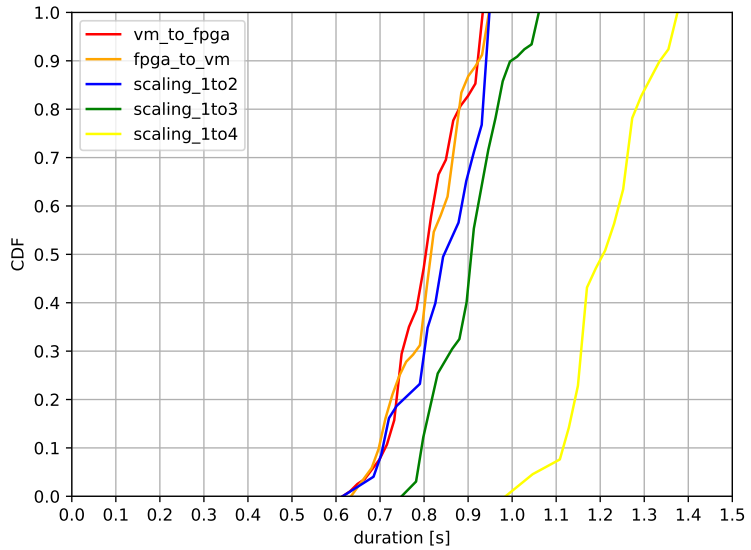

Fig. 11. Cumulative distribution function of configuration and migration times of VNFs in CoNFV. The term *scaling_1to4* indicates the amount of time needed to scale from 1 VM to 4 VMs

.

steering the network flows from an old hardware resource 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.

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 11 illustrates the cumulative distribution function (CDF) of the time required to perform several system configuration changes. As shown in Figure 11, migrating a VNF instance from a VM to an FPGA packet processor (red curve) takes less time than performing migration in the opposite 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 added 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.

It should be noted that the configuration proxy in the middlebox allows the currently-processed packet and any buffered packets associated with the VNF to complete processing before it is shut down or migrated. Packets destined for the VNF that arrive after the VNF has been removed are dropped. Since an SDN switch update is not performed until the new VNF is ready during migration, in practice, dropped packets for this case occur infrequently.

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 performed an experiment

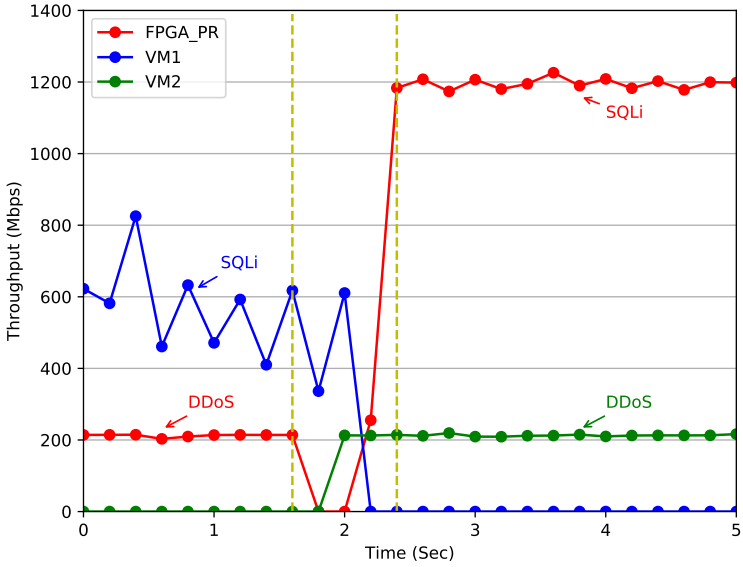

Fig. 12. Performance of system resources during partial FPGA reconfiguration. Resource migration is performed between the yellow lines in the figure.

with transient variations in the incoming workloads for DDoS and SQLi. Initially, 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 coordinator detects the imbalance of input and output traffic rates of the software SQLi attack detection and then determines that it can perform an FPGA middlebox update to support SQLi. The coordinator locates an overprovisioned FPGA NFV function, and swaps the current FPGA NFV function with an SQLi detection core during this period of high SQLi traffic.

In a multi-application experiment, a system with two VMs and one FPGA middlebox was used to demonstrate how quickly a packet processing function can be replaced within an FPGA by the configuration manager. The steps needed to perform the reconfiguration are described in Section 5. A software-based flow generator on the source host generates the packets that are checked for SQLi and DDoS attacks, 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 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. 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 0.41 Gbps (the VM throughput limit in Table 2), the configuration manager in the coordinator detects the imbalance of the input and output traffic rates of VM1. Since the DDoS detector throughput is less than 0.44 Gbps and can be handled in software, its function is migrated to VM2 and the FPGA middlebox is reconfigured to support SQLi detection.

Figure 12 show the delays associated with the redirection of the SQLi traffic from VM1 to the FPGA and FPGA reconfiguration using partial device (FPGA_PR) configuration. Results in the graph

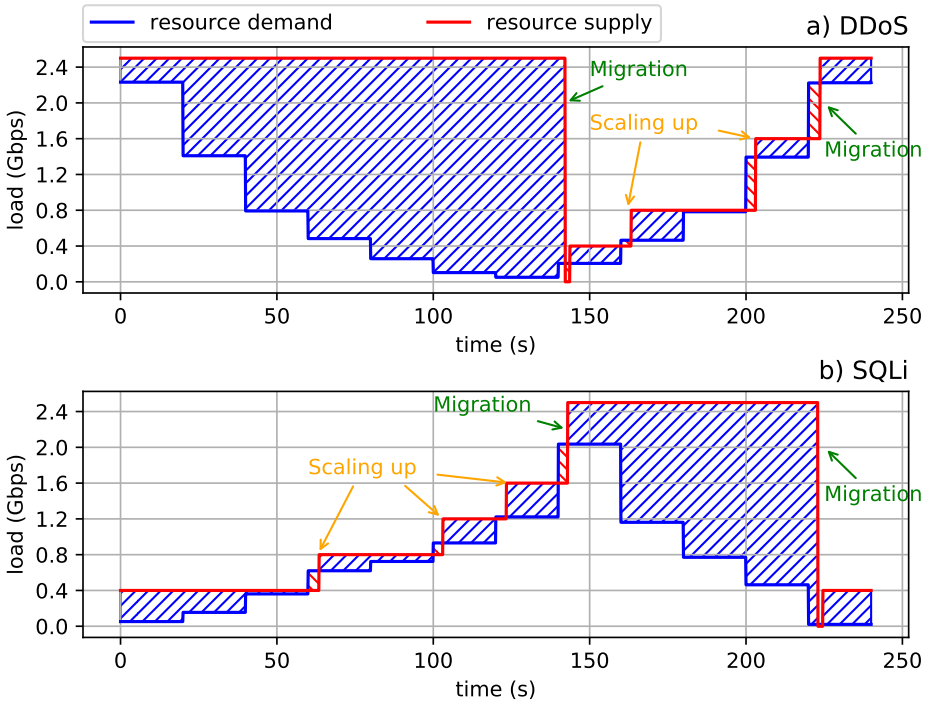

Fig. 13. Demand for SQLi and DDoS and resource supply using FPGAs and VMs. The processing demand and supply for DDoS is shown in (a). The corresponding values for SQLi are shown in (b). The resource demand curves are taken from prior work [13].

were generated from experimentation with FPGA and VM middleboxes in the lab. The partial FPGA reconfiguration process requires about 0.4 seconds which primarily consists of partial bitstream loading from flash by the FPGA circuit. The size of the entire bitstream is 31.3 MB, while the partial bitstreams for both SQLi and DDoS are 15.7 MB. The connection between the coordinator and the NIOS II on the FPGA stays active during the partial reconfiguration of the SQLi bitstream.

Algorithm Test: In this experiment, we evaluate the performance of the allocation algorithms described in Section 7 using realistic network workloads that have been previously used to assess NFV platforms [13]. Throughput demand and supply curves for one VNF of DDoS (a) and SQLi (b) appear in Figure 13. Results were collected from the system hardware. As shown in the Light+Peak scenario in Figure 13(b), the instantaneous surge in SQLi attacking flows results 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 14 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. Figure 15 provides a timeline of activities for

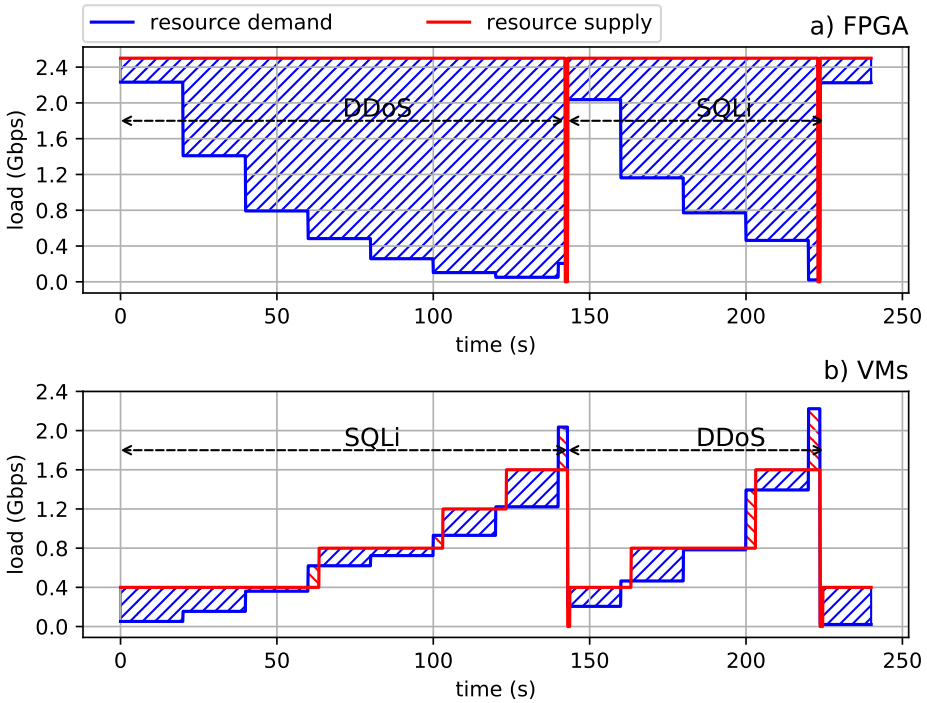

Fig. 14. Demand for SQLi and DDoS and resource supply using FPGAs and VMs

.

scaling from one VM to two VMs. In this case VNF traffic is split evenly by the switch for the two VMs.

10 CONCLUSION AND FUTURE WORK

In this manuscript, we have described a new heterogeneous hardware-software NFV platform that provides scalability and programmability while supporting significant hardware-level parallelism and reconfiguration (Figure 16). Field-programmable gate arrays and microprocessors are used to implement VNFs in a series of middleboxes. The assignment of VNFs to middleboxes is periodically assessed by a global coordinator that stores shared state. With the help of a 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. Our system was tested with multiple VNFs and standard traffic flow models. In the future we will explore using larger FPGAs that have numerous partially reconfigurable regions. New, faster reconfiguration approaches using internal FPGA reconfiguration ports will also be explored.

REFERENCES

[1] Zachary Baker and Viktor Prasanna. 2006. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. *IEEE Transactions on Secure and Dependable Computing* 3, 4 (Oct. 2006), 289–300.

[2] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *International Symposium on Field-Programmable Custom Computing Machines*. 109–116.

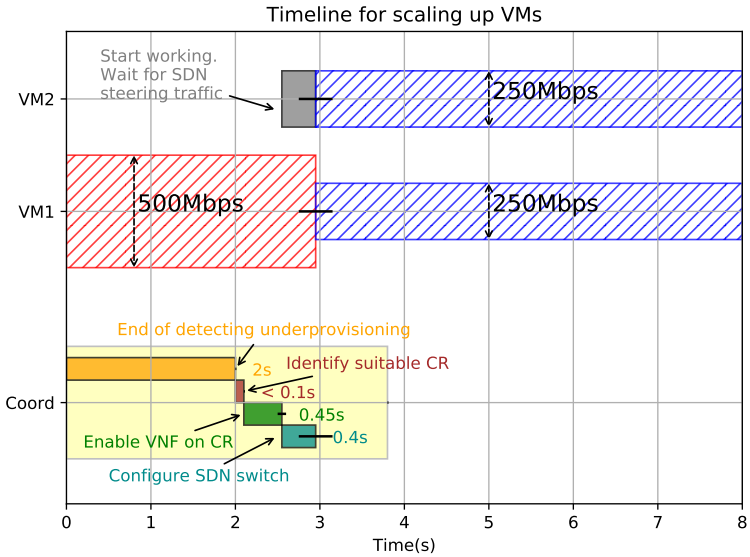

Fig. 15. System reconfiguration timeline of VM addition in response to underprovisioning.

Fig. 16. Photo of the CoNFV network function virtualization system. The two FPGA boards are shown on the left. The small desktop workstation is the coordinator.

[3] Sarang Dharmapurikar and John Lockwood. 2004. Deep packet inspection using parallel Bloom filters. *IEEE Micro* 24, 1 (2004), 52–61.

[4] Xiongzi Ge, Yi Liu, David H.C. Du, Liang Zhang, Hongguang Guan, Jian Chen, Yuping Zhao, and Xinyu Hu. 2014. OpenANFV: Accelerating Network Function Virtualization with a Consolidated Framework in OpenStack. In *ACM conference on SIGCOMM*. 353–354.

[5] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*. 163–174.

[6] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett. 2002. Granidt: Towards gigabit rate network intrusion detection technology. In *Proc. International Conference on Field Programmable Logic and Applications*. 404–413.

[7] Hamid Gholam Hosseini and Kang Li. 2012. Implementation of Transient Signal Detection Algorithms on FPGA. *International Journal of Computer Applications* 975 (2012), 8887.

[8] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoon, and Eric Keller. 2015. Stateless Network Functions. In *Proceedings: HotMiddleBox Conference*. 49–54.

[9] Christoforos Kachris, Georgios Sirakoulis, and Dimitrios Soudris. 2014. Network Function Virtualization based on FPGAs: A framework for all-programmable network devices. *CoRR* (June 2014), 1–5. https://arxiv.org/abs/1406.0309

[10] John W. Lockwood, James Moscola, Matthew Kulig, David Reddick, and Tim Brooks. 2003. Internet Worm and Virus Protection in Dynamically Reconfigurable Hardware. In *Proc. of the Military and Aerospace Programmable Logic Device Workshop*. 10.

[11] Kejie Lu, Dapeng Wu, Jieyan Fan, Sinisa Todorovic, and Antonio Nucci. 2007. Robust and efficient detection of DDoS attacks for large-scale Internet. *Computer Networks* 51, 18 (Dec. 2007), 5036–5056.

[12] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

[13] Leonhard Nobach, Benedikt Rudolph, and David Hausheer. 2017. Benefits of Conditional FPGA Provisioning for Virtualized Network Functions. In *International Conference on Networked Systems*. 1–6.

[14] Vladimir Olteanu, Felipe Huici, and Costin Raiciu. 2015. Lost in Network Address Translation: Lessons from Scaling the World's Simplest Middlebox. In *Proceedings: HotMiddleBox Conference*. 19–24.

[15] Manuel Peuster and Holger Karl. 2016. E-State: Distributed State Management in Elastic Network Function Deployments. In *IEEE NetSoft Conference and Workshops*. 6–10.

[16] Salvatore Pontarelli, Giuseppe Bianchi, and Simone Teofili. 2013. Traffic-Aware Design of a High-Speed FPGA Network Intrusion Detection System. *IEEE Trans. Comput.* 62, 11 (Nov. 2013), 2322–2334.

[17] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

[18] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *10th USENIX Symposium on Networked Systems Design and Implementation*. 227–240.

[19] Chen Sun, Jun Bi, Zhilong Zheng, and Hongxin Hu. 2017. HYPER: A Hybrid High-Performance Framework for Network Function Virtualization. *IEEE Journal on Selected Areas in Communications* 35, 11 (Nov. 2017), 2490–2500.

[20] Naif Tarafdar, Thomas Lin, Nariman Eskandari, David Lion, Alberto Leon-Garcia, and Paul Chow. 2017. Heterogeneous Virtualized Network Function Framework for the Data Center. In *International Conference on Field Programmable Logic and Applications*. 1–8.

[21] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2017. Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *International Symposium on Field-Programmable Gate Arrays*. 237–246.

[22] Jonathan S. Turner, Patrick Crowley, John DeHart, Amy Freestone, Brandon Heller, Fred Kuhns, Sailesh Kumar, John Lockwood, Jing Lu, Michael Wilson, Charles Wiseman, and David Zar. 2007. Supercharging PlanetLab: a high performance, multi-application, overlay network platform. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Kyoto, Japan, 85–96.

[23] Deepak Unnikrishnan, Ramakrishna Vadlamani, Yong Liao, Jérémie Crenne, Lixin Gao, and Russell Tessier. 2013. Reconfigurable Data Planes for Scalable Network Virtualization. *IEEE Trans. Comput.* 62, 12 (Dec. 2013), 2476–2488.

[24] Haining Wang, Danlu Zhang, and Kang G Shin. 2004. Change-point monitoring for the detection of DoS attacks. *IEEE Transactions on Dependable and Secure Computing* 1, 4 (2004), 193–208.

[25] Yi-Hua Edward Yang, Weirong Jiang, and Viktor K. Prasanna. 2008. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 30–39.

[26] Xuzhi Zhang, Xiaozhe Shao, George Provelengios, Naveen Kumar Dumpala, Lixin Gao, and Russell Tessier. 2017. Scalable Network Function Virtualization for Heterogeneous Middleboxes. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 219–226.