Heterogeneous Virtualized Network Function Framework for the Data Center

Naif Tarafdar, Thomas Lin, Nariman Eskandari, David Lion, Alberto Leon-Garcia, Paul Chow
University of Toronto
Toronto, Ontario, Canada
Email: {naif.tarafdar, t.lin, nariman.eskandari, david.lion}@mail.utoronto.ca,
alberto.leongarcia@utoronto.ca, pc@eecg.toronto.edu

***Abstract*—We present a framework for creating heterogeneous virtualized network function (VNF) service chains from cloud data center resources. Traditionally, these functions are packaged in software images within a catalog of networking applications that can be loaded onto a virtual machine CPU, and can be offered to users as a service. Our framework combines the best of both software and hardware by allowing users to chain traditional software-based VNFs with hardware-based VNFs that the user provides as an IP to generate a bitstream or a pre-generated VNF as part of a library. To accomplish this, our framework first creates the hardware bitstreams and programs the FPGA VNFs, loads any software VNFs requested, and programs the network to daisy chain the VNFs together. Furthermore, this enables an incremental design flow where the user can start by implementing a chain of VNFs in software and incrementally substitute software VNFs for their hardware counterparts. Our paper investigates two case studies to show the ability to switch between hardware and software VNFs in our framework and to demonstrate the benefit of using hardware VNFs. The first study is signature matching at fixed offsets, similar to matching packet headers. In this case study, the CPU can keep up at line-rate using specialized networking drivers. The second case study involves string matching within a packet, which requires scanning through the entire frame. In this case, the CPU performance drops to approximately 20 percent of the input rate, whereas the FPGA can continue to keep up at line-rate.**

I. Introduction

A cloud infrastructure network can be seen as a large collection of shared resources, which are provisioned to users by multiplexing them in time and space. This provisioning is known as Infrastructure as a Service (IaaS), where users can build large computing infrastructures without the capital investment of purchasing physical clusters of computing, networking, and storage devices. The sharing of the computational resources is done through virtualization, in which a layer of abstraction hides the physical details of the shared resources from the user. These physical details include the actual physical location of the resource, its physical specifications, as well as the presence of other users sharing the same resource, thus giving the illusion of a complete physical resource to the user. The privacy, performance, and other complications that arise due to the sharing of devices are popular areas of research. Current commercial services such as Amazon EC2 [1] have only scratched the surface when it comes to provisioning heterogeneous devices, as much work remains such as the work addressed by this paper for abstraction layers to integrate heterogeneous devices in large clusters within the cloud.

Field-Programmable Gate Arrays (FPGAs) have recently been shown to be able to address the power and performance issues being faced by data centers. The best example with published details is the Microsoft Catapult project where FPGAs were deployed in the Bing search engine [2] as part of its ranking mechanism. With only a 10% increase in power and 30% increase in cost, a 95% increase in performance was achieved. These performance and power savings multiply significantly at the scale of a data center.

The challenge of using FPGAs in a cloud is that there has been little infrastructure developed to provision FPGA resources in a way that allows many users to create and interact with their own virtual FPGA compute cluster. In contrast, this problem is much better understood for software-based virtual machines (VMs). What is needed is an implementation of a mechanism for provisioning FPGA clusters within a fully heterogeneous environment, where the clusters can communicate with any other network device be it a CPU, another FPGA cluster, or Internet-of-Things (IoT) device.

One area in which FPGAs are heavily used is the area of networking. This area requires low-latency processing, which might be hindered by packets traversing through the network stack of a standard operating system. FPGAs allow users to create custom hardware to process packets as they arrive at line rates. The recent work on Network Function Virtualization (NFV) [3] aims to virtualize packet processing functions, enabling users to instantiate software-based Virtualized Network Functions (VNFs) from the cloud as they would any other virtual machine. As networking bandwidth continues to grow in the coming years, we foresee that CPUs will be unable to satisfy network functions that require low-latency processing at multi-Gigabit line rates, thus creating an opportunity for FPGA-based VNFs.

In this work, our goal is to provide a joint interface for allowing users to create both software and hardware VNFs, and chain them together on the network. The user would specify the VNF descriptions, which are used to instantiate a cluster of virtual machines. These VNF descriptions, in a traditional sense, are CPU VM software images, but we have also adapted them to include FPGA bitstreams. This introduces a novel heterogeneous design flow, where the user

can create a VNF software chain and incrementally move network functions from software to hardware. This paper continues as follows. Section II introduces background information on the adoption of FPGAs in data centers, networking functions and service chaining. In Section IV we delve into our generated infrastructure and the deployment flow of our framework. In Section V we explain our results by quantifying the overhead of our framework, deployment infrastructure and a few case studies. Section VI introduces related work in FPGA networking functions and infrastructure for service chained network functions. Lastly in Section VII we explain future work in terms of infrastructure and applications and Section VIII concludes the paper.

II. Background

This section will review the background relating to Network Function Virtualization, Software-Defined Networking, and Service Function Chaining.

A. Software-Defined Networking and OpenFlow

Software-Defined Networking (SDN) is a concept that enables programmatic control of entire networks via an underlying software abstraction. This is achieved by the separation of the network control plane from the data plane as shown in Figure 1. SDN opens the door for users to test custom network protocols and routing algorithms, and furthermore, it allows the creation, deletion, and configuration of network connections to be dynamic. The current de facto standard protocol for enabling SDN is OpenFlow [4].

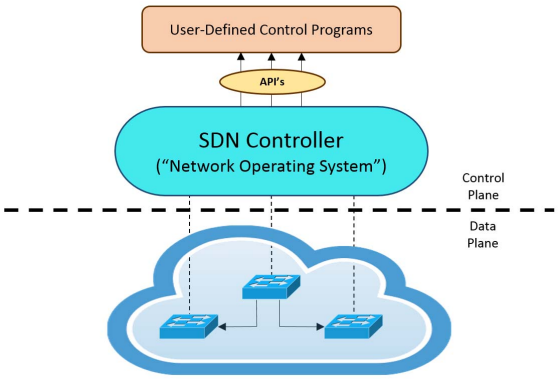


Fig. 1: System diagram of an SDN, where user-defined control programs manage network switches.

In OpenFlow, the control plane is managed by a user program running on a CPU that leverages APIs exposed by an SDN Controller. The SDN controller, often referred to as the "network operating system", abstracts away network details from the user programs. The controller manages the data plane and creates configurations in the form of flows. These flows describe the overall behaviour of the network, and can be used to specify custom paths through the network based on packet headers, or even specify operations on the packets themselves (e.g. drop packets, modify headers, etc.).

B. Network Function Virtualization

Network Function Virtualization (NFV) [3] is a concept for virtualizing network functions that have traditionally been provided by proprietary vendors as closed-box appliances. A network function is defined as a device that provides a well-defined network service, ranging from simple services such as firewalls, content caches, and load balancers, to more sophisticated ones such as intrusion detection & prevention. With recent gains in CPU performance, NFV aims to virtualize these services and create Virtualized Network Functions (VNFs) in software, running on commodity devices.

C. Service Chaining of VNFs

The activities surrounding SDN complements the recent work on NFVs. Both concepts aim to enable more flexible provisioning and management of infrastructure resources. When NFV is deployed in conjunction with SDN, it enables network operators the ability to create complex network services on demand by steering traffic through multiple VNFs realized using programmable resources. This practice is often called "service function chaining" (SFC), or simply "service chaining". In addition, since NFV allows VNFs to be created or migrated closer to where they are needed, SDN can be leveraged to automatically re-route traffic to wherever the VNFs are placed.

D. Data Center Setup

We use the SAVI testbed, a Canada-wide, multi-tier heterogeneous cloud [5], as our data center for this work. One of the main goals of the SAVI testbed is to explore the thesis that all components of the physical infrastructure can be virtualized. This testbed contains various heterogeneous resources such as FPGAs, GPUs, Network Processors, IoT sensors and a large number of conventional CPUs, all interconnected via a fully SDN-enabled network fabric. As part of the ongoing research into how to jointly manage and virtualize these resources, our work investigates the virtualization of FPGA platforms.

Previous virtualization work conducted on this testbed includes the work by Byma et al. [6] which provided partial FPGA regions as an extended OpenStack (see IV-B) resource, and the work by Tarafdar et al. [7], which creates large multi-FPGA clusters out of network-connected FPGAs. The FPGAs in this testbed are connected to 10 Gigabit Ethernet switches via SFP+ transceivers. This allows us to create chains containing a mix of both FPGAs and CPUs, as they are both connected together to the same network switches. Our proposed framework is able to programmatically install network flows to connect VNFs together, forwarding select traffic between VNFs to stitch together the various networked devices.

E. OpenStack

OpenStack [8] is an open-source cloud computing platform used to manage the SAVI testbed. It is divided into several services dedicated to managing different aspects of the infrastructure. The two main OpenStack services that we employ in

our platform are Nova and Neutron, both of which offer a set of APIs that enable clients to make queries and requests.

Nova is primarily responsible for the virtualization of compute resources, which involves the generation of virtual machines (virtualized CPUs) on physical servers. When a client requests a virtual machine, two fields are specified: a software *image*, and a *flavor*. The software image refers to all the software that is to be loaded onto the virtual machine, and contains both the operating system and any other applications that we want pre-installed in our virtual machine. These images are typically kept in an image repository and can be updated by users of the testbed. The flavor refers to the physical specifications of the virtual machine, such as number of CPU cores, RAM, and hard disk space.

Neutron is responsible for the state keeping of network resources. Its design leverages a pluggable backend to effect actual changes to the network. In the SAVI testbed, Neutron uses an SDN controller to control and virtualize the network. Thus, by liaising with Neutron, clients are able to create and register new virtual network ports, which assigns MAC addresses and IP addresses that will be valid for use within the network. When creating virtual machines, these ports are created implicitly, but we can explicitly create additional ports for non-virtual devices or for non-CPU devices.

III. Justification of Design

This design presents another hardware middleware layer for multi-FPGA environments. We define a middleware layer as a communication layer for multiple FPGAs to communicate. Our previous work in [7] has implemented low-level infrastructure and a packet-switched middleware layer for multi-FPGA applications. That work divides a large cluster of FPGA IP blocks connected by a logical switch where any IP block can communicate with any other IP block through the use of packet-headers. The work presented in this paper is a middleware layer model of a circuit-switched network, where we no longer need a header to arbitrate routing decisions as a path is fixed until we change it with our infrastructure. This is an appropriate use-case for network functions that intercept traffic without modifying traffic headers. Furthermore, since packet headers are not modified this allows us to swap portions of a VNF chain with multiple implementations, such as swapping a CPU implementation for an FPGA implementation. Our infrastructure allows a user to divert traffic by providing the IP address of their device. Since we provide both FPGAs and CPUs with IP addresses during device allocation within our cloud we can divert traffic to either device.

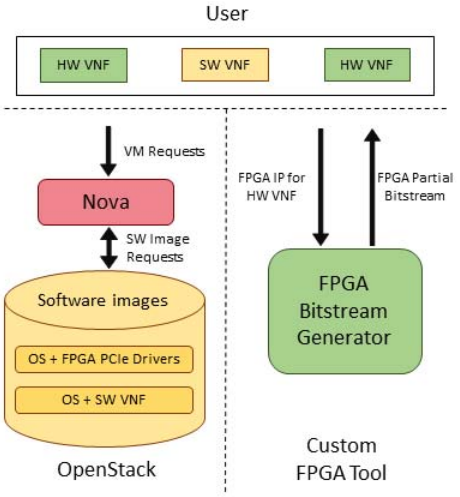
IV. Infrastructure

This section presents the overall infrastructure of our tool's flow. We start by discussing the service chain scheduler and the environment that we use to create the heterogeneous service chains.

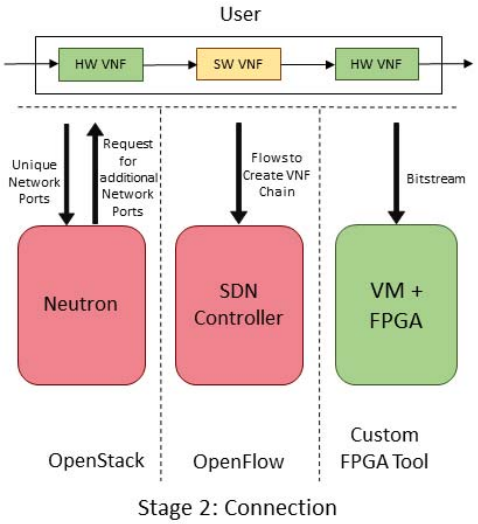
A. Service Chain Scheduler

The Service Chain Scheduler's main role can be described in two parts: it is first responsible for acquiring the user's requested resources from a pool of available resources, then it is tasked with making the necessary network configuration changes to steer traffic through the provisioned VNFs. Our Service Chain Scheduler leverages OpenStack to manage the infrastructure's compute resources, and OpenFlow to manage the networking between the various provisioned devices within our data center. Figure 2 shows the high-level view of the Service Chain Scheduler, where the two stages of its operation are illustrated.

In the first stage shown in Figure 2(a), it is responsible for the allocation of the CPU and FPGA resources. In our example, the user requests a chain of both Hardware and Software Virtualized Network Functions (denoted by HW VNF and SW VNF, respectively). Our framework issues the nec-



(a) The allocation stage of the Service Chain Scheduler.



(b) The connection stage of the Service Chain Scheduler.

Fig. 2: The Service Chain scheduler divided into two stages, an allocation stage and a connection stage.

essary directives to allocate virtual machines with a directly-connected FPGA via PCIe for the HW VNF, and a standard virtual machine with the appropriate software image for the SW VNF. Our framework also interfaces with another virtual machine dedicated to generating the appropriate bitstreams for each HW VNF, where it automatically packages the user IP provided in RTL with the appropriate ports and generates a partial bitstream that will be programmed into the FPGA. Alternatively, the user can also specify a hardware VNF from a library of pre-generated bitstreams. Section IV-C will explain the hardware infrastructure used to interface with the user-provided VNF.

The second stage, shown in Figure 2(b) is responsible for steering the network traffic and chaining the heterogeneous VNFs together. By default, each virtual machine is allocated a unique network port (within the data center). However, for the FPGA ports, our scheduler needs to issue further requests to Neutron and the SDN controller to acquire and register the FPGA's network ports. Finally, the Service Chain Scheduler generates the required network flows to interconnect the HW VNFs (by specifying their newly allocated ports) and SW VNFs together. The Service Chain Scheduler first calculates the network traffic without the VNF chain between a network source and sink. Then for each network hop, the traffic matching the flow is re-routed to the newly inserted VNF of the VNF chain. This is then repeated for each VNF in the chain. This process ensures that the network headers match the original source and sink but are re-routed through our VNF chain without header modification.

B. OpenStack and OpenFlow

We set up our OpenStack environment for allocation as follows. For a SW VNF, our framework simply issues a request for a virtual machine using an image containing the standard Ubuntu operating system and the desired SW VNF. For a HW VNF, the framework requests a virtual machine with a specific flavor that contains a PCIe-connected FPGA. The resulting virtual machine will have the standard Ubuntu operating system and have direct access to an FPGA (described in Section IV-C). In this setup, OpenStack uses *PCIe passthrough* to give a virtual machine full access to the underlying server's PCIe device [9].

For each virtual machine with an FPGA, we can infer the physical port the FPGA is connected to by looking at the ID of the virtual machine's underlying physical server, as we are aware of the FPGA switch port connections given a physical host. We use Neutron to allocate and register extra virtual network ports per FPGA port, associating each physical port to a corresponding virtual network port recognized by the SDN controller. At this stage, we can request for the SDN controller to install network flows to direct traffic between different virtual network ports within our data center. All of these steps are done by our Service Chain Scheduler as described in Section IV-A.

C. FPGA Hypervisor

We provide an FPGA Hypervisor that is an abstraction for certain I/O such as the network, PCIe, and off-chip DRAM on the FPGA. The FPGA hypervisor is shown in Figure 3. Along with our hypervisor, we provide a driver called the Hardware Abstraction Layer (HAL) that can communicate to the FPGA via PCIe. The HAL provides an API for the user to communicate with various components on the FPGA through a memory-mapped interface. The components include configuring a soft-processor on the FPGA, sending control data to an application, and writing data to off-chip memory. With these connections, our hypervisor allows users to create a data-path for their application using the network and the configuration of control signals via the HAL through PCIe.

The memory-mapped interface we use for the HAL to control the various components is the AXI protocol. This is a protocol for memory-mapped interfaces used between Xilinx memory-mapped IP cores as well as ARM cores [10]. For the data path of our application we use the AXI stream interface that is connected to the 10 Gigabit/s networking interface on the FPGA, which streams packets as 8-byte words at a clock rate of 156.25 MHz.

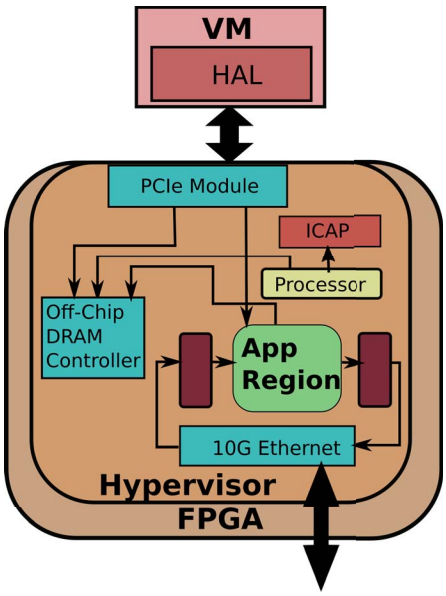


Fig. 3: System diagram of our FPGA Hypervisor.

D. Partial Reconfiguration Flow

Our framework issues requests to a dedicated bitstream generator. Our bitstream generator tool is another virtual machine in our data center with the appropriate software tools installed. To create partial bitstreams, we first need a design checkpoint that has our hypervisor synthesized with the partial region as a place holder. For a given user IP, our framework loads the IP into a Vivado project, wraps the project with the appropriate ports, and then transports the packaged IP into a project with the hypervisor IP. Our framework then creates a netlist for the application by synthesizing only the application

in the context of the entire hypervisor. This netlist is then loaded into our hypervisor checkpoint, and a partial bitstream is generated. Once these bitstreams are generated, the Service Chain Scheduler programs them onto the respective FPGAs using the PCIe interface provided by the hypervisor.

The programming and management of our application region is done through the soft processor in the FPGA. The processor manages the decoupling gates (denoted by the red boxes in Figure 3), and the ICAP within the FPGA. The ICAP is an IP within the FPGA Hypervisor that programs the FPGA with a partial bitstream. To ensure safe programming, the decoupling gates disconnect the application region from the Ethernet module during partial reconfiguration. Gating ensures that there are no transactions in flight via the data path (AXI stream from the network) or the control path (AXI from the PCIe) while we reprogram the application region. Once reprogrammed, the processor un-gates the application region, thus reconnecting it to the data and control paths again. The gating, programming and un-gating of the application region through the processor is initiated through an API call within the CPU through PCIe. We took the approach of using a processor to control the programming of the application region because this is not on the critical path (as opposed to directly controlling these components via PCIe). Furthermore we would like to extend our programming capabilities via the network so that we can easily communicate to the processor through network packets.

E. Design Flow for Hardware NFV

Our infrastructure in the cloud also allows designers of large multi-FPGA network designs to approach their design with an incremental design flow. An example of this design flow is as follows:

1) Implement all parts of the VNF design. Each function is an OpenStack image that contains a software application listening to the network port, performing a function, and outputting to the port.
2) Implement and test each individual VNF as an FPGA-offloaded design.
3) Swap the software-based VNF with the tested FPGA-based VNF.
4) If the service chain remains functionally correct, then repeat Steps 2 and 3 for the next VNF in the chain. Repeat until the whole chain is implemented using FPGAs.

V. Results

In this section we explain the resource utilization of our Hypervisor, the overhead to program the FPGAs and explore a service chain we created within our infrastructure, one with a SW VNF and one with a HW VNF.

A. Resource and Programming Overhead

The hypervisor includes a 10G SFP Ethernet module, Microblaze soft-processor for controlling the partial region and an off-chip memory controller. Table I summarizes the resource overhead of our FPGA Hypervisor.

TABLE I: Resource Overhead of our Framework on the Alpha Data 7V3 FPGA Board.

	LUTS	Flip-Flops	BRAM
FPGA Hypervisor	62344 (14.4 %)	76124 (8.79 %)	228 (15.5 %)
Total Available	433200	866400	1470

We also investigate the required time to acquire a virtual CPU within our data center, which is on the order of minutes (ranging from 1 to 3 minutes depending on the network load, the size of the software image, and whether or not the image has been cached). This time is the same for a VM with an FPGA and a VM without an FPGA.

Lastly, we investigate the required time to create and program our partial bitstreams. We also allow users to bypass the stage of creating bitstreams by allowing users to supply their own bitstreams instead of RTL. This is summarized in Table II

TABLE II: Time Required to Program and Create Application Bitstreams and Hypervisor Bitstreams.

	Time to Program FPGA (s)	Time to Create Bitstream (min)
FPGA Application Using Partial Reconfig	20 to 40	40 to 60
FPGA Hypervisor	60 to 90	90 to 120

B. Signature Matching and String Search Case Studies

We create the VNF chain shown in Figure 4. The Service Chain Scheduler redirects all packets coming from VNF A into VNF B and then into VNF C. VNF A is a traffic generator in software, VNF B is a Signature Matching Engine, and VNF C is a destination for all traffic. We created three versions of VNF B, an unoptimized software version, an optimized software version and a hardware version. We attempt to process these packets arriving at line rate, which is close to 10 Gb/s. Our first case study is the matching of signatures within an Ethernet packet. A signature is defined as a pattern at a specific offset. For traditional network packets (for example, layer 2 or layer 3 packets) a signature can be defined as a header. This notion can be extended to matching in the payload of packets which can be used for deep packet inspection. This can also be used to create custom headers for future internet protocols or custom networks within a data center. We explore a hardware implementation and software implementation of this application.

Fig. 4: System diagram of our FPGA Hypervisor.

1) Fixed Offset Signature Matching Hardware: First we explore the VNF chain in Figure 4 with VNF B implemented in hardware. This was done by providing a Software Image of VNF A, and VNF C and hardware IP implementing VNF B to

the Service Chain Scheduler that allocates the resources and networks the VNFs together. The hardware implementation of the signature match can be broken down into two blocks. The first being a matching engine, and the second is a gate. The matching engine is parameterized with the offset to look for, the pattern to match on and a mask to specify the bits of the signature we are interested in. The signature gate streams the packet out with the correct checksum if all the signatures match, or with an incorrect checksum (this causes the network interface to drop the packet) if not all signatures match. The signature gate is implemented to always stream out the packet as the signature gate might not know if all the signatures match until the last flit of the packet, and at which point the signature gate either outputs the correct checksum or an incorrect checksum, that results in the entire packet to be dropped by the network interface. Each offset is described in terms of the position of the word the offset is located in (these are being streamed in 8-byte words). These parameters are configured using a control path that comes from the virtual machine CPU. Each matching engine asserts a match flag if the packet matches the configured signature, and this signal stays asserted for the period the packet is being streamed. For multiple signatures these matching engines are pipelined and the match signals generated are propagated through an AND gate to determine if all signals have matched. To align the match signals generated by different matching engines at different stages of the pipeline, we insert shift-registers of varying lengths to make sure that the match signals from each matching engine lines up when the last matching engine observes the last flit of the packet. This engine for matching two signatures is shown in Figure 5.

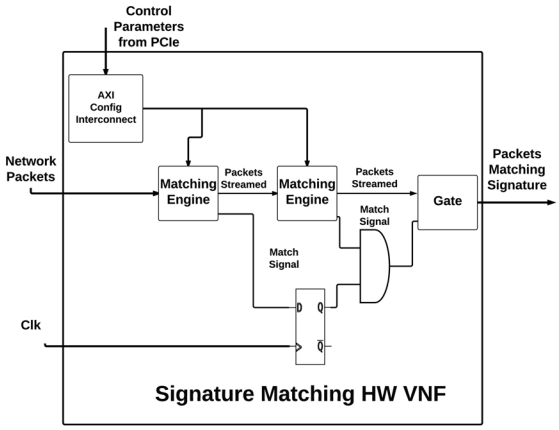


Fig. 5: System diagram of the FPGA implementation that matches two signatures.

The resource utilization of the signature matching engines and the signature gate is shown in Table III. The number of signature matching engines is dependent on the number of signatures we wish to match, whereas the number of signature gates is always one. For our case study we implemented up to 140 signatures on the FPGA but from our resource estimation we can fit close to 1100 signature matching engines on the FPGA.

TABLE III: Resource Overhead of one Signature Matching Engine and Signature Gate.

	LUTS	Flip-Flops	BRAM
Signature Matching Engine	301 (0.07%)	277 (0.0032 %)	0 (0 %)
Signature Gate	80 (0.02 %)	4 (0.0005 %)	0 (0 %)

To evaluate the performance of this design we used the tool tcpreplay [11] to send a stream of UDP packets to the FPGA. The rate was limited by the network connection at 10 Gb/s where we saw performance peak at close to 9.7 Gb/s. The slight drop is due to the extra overhead of framing and control bits that are not displayed with most network monitoring tools. The FPGA was able to keep up to line rate due to the pipelined nature of the solution as we observed the same throughput exiting the FPGA with no packet drop.

2) Fixed Offset Signature Matching Software: With our Service Chain Scheduler we can redirect network traffic from VNF A to our second implementation of VNF B, which is a software implementation of VNF B. Our initial software implementation used the C Berkeley sockets library. The application issues a system call using the sockets API to read a packet from the operating system's network stack. Each packet is then buffered and for each signature, if it is found at its prescribed offset, a flag is set. If all the flags are set, then the buffered packet is written to the outgoing socket via another system call. Figure 6 shows the performance of the packets being sent and and returned in software. Note that the packets are sent at approximately close to 9.7 Gb/s (Rx Rate is the rate of input to VNF B) but as we increase the number of signatures the software implementation cannot keep up to line rate (indicated by the Tx Rate, the rate of output from VNF B).

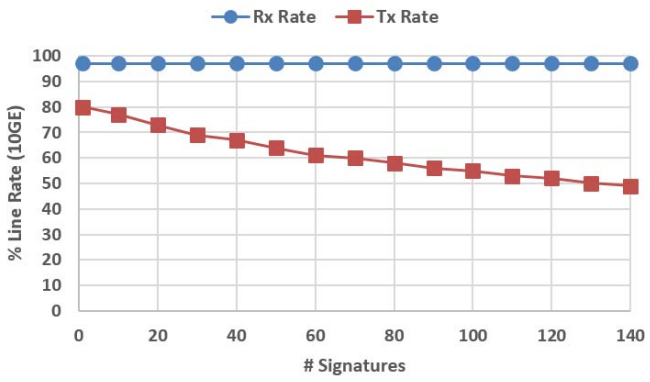


Fig. 6: Fixed offset signature matching using Berkeley sockets. Throughput of packets sent and returned versus the number of signatures.

The observed throughput drop can be attributed to either the overhead of the system calls to the network stack, or the actual CPU computation involved in finding the signatures.

To isolate the CPU computation we use the Intel Data Plane Development Kit (DPDK) [12], which gives the CPU low-level access to network interface card drivers, bypassing the traditional network stack. After re-implementing the software version of the signature matching with DPDK on the network interface we observed that the CPU can perform at line rate. We observe the transmission throughput at 9.7 Gb/s, equivalent to the rate packets are received, with no packet drop. This shows that in this particular use case the CPU computation can keep up with the FPGA implementation at line rate.

3) String Matching in Hardware: Now we look at implementing a second VNF function for VNF B, which is string matching. This is similar to the signature matching use case introduced earlier with the exception that we no longer configure our engine with an offset, as now the string can occur at any point in the packet. The VNF is configured with the search string through the HAL and PCIe interface. We currently fix our string length to 10 bytes. The overall architecture of the HW VNF remains the same with a matching engine per signature and a gate at the end of the stream. The implementation of the matching engine is modified to be a shift register, which stores the last 10 bytes that the matching engine has seen, and then compares this to the configured pattern we are checking for. Similar to the previous use case this is also a pipelined application that can operate at line-rate. The resource utilization of this VNF is summarized in Table IV. We also synthesized up to 140 signatures and we can fit up to 1200 signatures into our current application region. We use our Service Chain Scheduler to redirect traffic from VNF A to this implementation of VNF B.

TABLE IV: Resource overhead for one string matching and signature gate.

	LUTS	Flip-Flops	BRAM
String Matching Engine	272 (0.063%)	209 (0.0024 %)	0 (0 %)
Signature Gate	80 (0.02 %)	4 (0.0005 %)	0 (0 %)

Similar to the first use case this is also a pipelined implementation and the HW VNF can process the packets at line-rate. For a received packet transmission rate of 9.7 Gb/s we see a transmission rate of 9.7 Gb/s with 0 packet drop.

4) String Searching in Software: We implemented this application using the DPDK libraries to limit the overhead of the network interface. This application requires scanning through the entire packet for each signature the user wants to search for. The performance for this version of the SW VNF is shown in Figure 7. This application does show a drop in throughput after about 30 signatures, which is when the CPU cannot keep up to line rate and packets start to get dropped.

These use cases have shown that there exists cases where a VNF can process packets at line rate in software and in hardware, thus showing the potential of large-scale heterogeneous chaining of VNFs. This also shows how with our Service Chain Scheduler we can easily create VNFs, modify large VNF chains and substitute VNFs within a chain easily by redirecting flows.

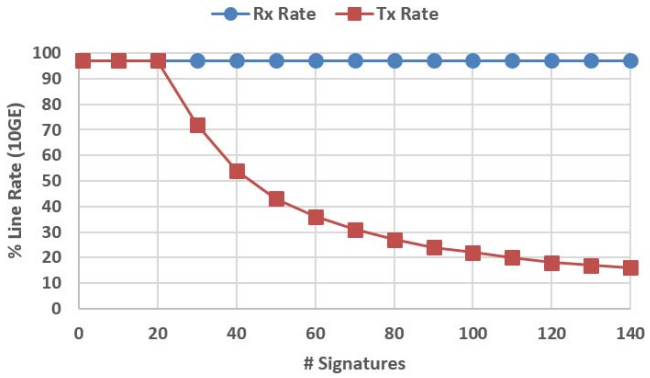


Fig. 7: String searching using DPDK: Throughput of packets sent and returned versus the number of string signatures.

VI. Related Work

This section presents related work in implementing VNFs on FPGAs and managing FPGAs in cloud platforms.

A. Hardware VNF Implementations

The use of FPGAs for network processing has been explored since the early 2000s. Lockwood et al. [13] demonstrated through the FPx platform that FPGAs can be useful when appended to the data path of network processing through several applications such as packet routing, data queuing and data content modification; these applications can be implemented through the use of VNFs. Other applications such as packet classification [14], and deep packet inspection [15], can also be implemented as VNFs and thus automatically chained together with our infrastructure. Similar to software applications in SDN we can similarly implement OpenFlow extensions in hardware such as [16]. All of these works show applications that can be modelled as hardware VNFs that now can be automated and combined with software VNFs in our platform.

B. FPGA Management in Cloud Platforms

Our Service Chain Scheduler allocates and connects resources, including FPGA resources from a pool of FPGA resources. In this section we will look at other related works that allocate FPGAs within a data center, or connect FPGA clusters in a data center.

The work proposed by Byma et al. [6] uses OpenStack to manage Virtual FPGA Resources (VFR). This infrastructure created an FPGA hypervisor with a 1 Gb/s network port and four application regions. Each application region is a VFR, which is programmed via partial reconfiguration. This infrastructure allows multiple users to share one FPGA.

The work proposed by Chen et al. [17] also uses OpenStack to manage non-network connected FPGAs. This was implemented by modifying the Linux hypervisor KVM. This creates the hypervisor in software to manage the FPGA.

IBM's SuperVessel project [18] uses OpenStack to provision a single non-network connected FPGA as part of an accelerator with a virtual machine CPU. SuperVessel allows users to create accelerator engines out of user provided IP, and returns to the user a programmed FPGA with a tightly coupled virtual machine CPU. IBM continued a proof of concept with network connected FPGAs in their Hyperscale project [19]. This proof of concept proposes modifying OpenStack to accept bitstreams and returns a network connected FPGA by returning an IP address of the network connected device.

Microsoft has also published their revised version of Catapult that has FPGAs connected directly to network switches [20]. The output of the host NIC connects to the other FPGA port. Thus the CPU network connection is through the FPGA, whereas the FPGA is directly connected to the network switch ensuring that FPGA applications have low-latency.

VII. Future Work

Currently our work considers a VNF as an entire FPGA bitstream. This may be wasteful in this situation as a VNF could be a small function. One area of future work can be to pack multiple VNFs as kernels within one FPGA and package one bitstream containing multiple VNFs. This is similar to the approach taken in Tarafdar et al. [7], where multiple FPGA kernels can be mapped onto one FPGA (as part of an FPGA cluster). We can also take the approach proposed by Byma et al. [6] where the one FPGA is divided into multiple partial regions. This would allow us to have the same granularity of one VNF per bitstream but share the FPGA. This approach would also allow us to share one FPGA across multiple users in the data center.

We would like to move towards a model that supports stand-alone FPGAs as well as FPGAs with PCIe CPU controllers. Currently the programming is managed via the PCIe-connected CPU. However if we configure our processor controlling the application region to be programmed via the network we can support stand-alone network-connected FPGAs in our framework. This will require us to remodel how these accelerators are allocated via OpenStack as we cannot manage them with their CPU anymore. We can continue to use OpenFlow to program the network flows between network-connected FPGAs.

We would also like to implement a large-scale library of VNFs. A library of VNFs where some VNFs are implemented in software and others in hardware can help us investigate VNF chains on a large-scale. It will also be interesting to study which applications to implement in software or in hardware. Existing packet processing libraries such as Snort, which is a library of security related can be implemented in a series of hardware and software VNFs within our framework [21].

VIII. Conclusion

Our infrastructure automates the creation of VNFs from a pool of resources available in a data center. Our use cases and related work has shown that there are situations for which a hardware VNF can be useful for high performance line-rate processing network applications. Our framework will allow the user to create a large chain of heterogeneous VNFs using software and hardware functions. This will allow users to specialize applications in modules that are appropriate for FPGA or CPU and allow an incremental deployment flow for FPGA VNF chains.

Acknowledgment

The authors would like to thank the SAVI testbed, NSERC, Xilinx, CMC and Huawei Technologies Canada for providing the infrastructure and funding this research.

References

[1] Amazon Web Services Inc. Amazon Web Services (AWS). http://aws.amazon.com, 2014.

[2] Andrew Putnam et. al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[3] ETSI. Network Functions Virtualisation (NFV); Architectural Framework. https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2013.

[4] Nick McKeown et. al. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[5] Thomas Lin, Byungchul Park, Hadi Bannazadeh, and Alberto Leon-Garcia. SAVI Testbed Architecture and Federation. In *International Conference on Future Access Enablers of Ubiquitous and Intelligent Infrastructures*, 2015.

[6] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116. IEEE, 2014.

[7] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul 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*, pages 237–246. ACM, 2017.

[8] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. OpenStack: Toward an Open-Source Solution for Cloud Computing. In *International Journal of Computer Applications*, 2012.

[9] Redhat. PCI Passthrough. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Virtualization/chap-Virtualization-PCI_passthrough.html, 2017.

[10] AMBA ARM. AXI Protocol Specification, 2003.

[11] Aaron Turner and M Bing. Tcpreplay: Pcap editing and replay tools for* nix. http://tcpreplay.appneta.com, 2005.

[12] DPDK Intel. Data plane development kit, 2014.

[13] John W Lockwood. An open platform for development of network processing modules in reprogrammable hardware. *IEC DesignCon01*, 2001.

[14] Jeffrey Fong et al. ParaSplit: A Scalable Architecture on FPGA for Terabit Packet Classification. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, pages 1–8, Aug 2012.

[15] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep Packet Inspection Using Parallel Bloom Filters. In *High Performance Interconnects*, pages 44–51, Aug 2003.

[16] Stuart Byma, Naif Tarafdar, Talia Xu, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Expanding OpenFlow Capabilities with Virtualized Reconfigurable Hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 94–97, 2015.

[17] Fei Chen et. al. Enabling FPGAS in the Cloud. In *Computing Frontiers*, 2014.

[18] IBM Research. OpenPOWER Cloud: Accelerating Cloud Computing. https://www.research.ibm.com/labs/china/supervessel.html, 2016.

[19] Jagath Weerasinghe et. al. Enabling FPGAs in Hyperscale Data Centers. In *UIC-ATC-ScalCom, 2015 IEEE 12th Intl Conf on*, pages 1078–1086. IEEE, 2015.

[20] Adrian M Caulfield et al. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, October 2016.

[21] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.