Building Hybrid Virtual Network Functions with eXpress Data Path

Nguyen Van Tu*, Jae-Hyoung Yoo†, and James Won-Ki Hong*
*Dept. of Computer Science and Engineering, POSTECH, Pohang, Korea
†Graduate School of Information Technology, POSTECH, Pohang, Korea
Email: {tunguyen, jhyoo78, jwkhong}@postech.ac.kr

***Abstract*—Network Function Virtualization (NFV) decouples network functions from dedicated, proprietary hardware into software Virtual Network Functions (VNFs) that can run on standard, commodity servers. One challenge of NFV is to provide high-throughput and low-latency network services. In this paper, we propose eVNF - a hybrid architecture to build and accelerate VNFs with eXpress Data Path (XDP). XDP is a Linux kernel framework that enables high-performance and programmable network processing. However, the programmability of XDP is limited to ensure kernel safety, thus causing difficulties in applying XDP to NFV. eVNF solves this problem by taking a hybrid approach: leave the simple but critical tasks inside the kernel with XDP, and let complex tasks be processed outside XDP, e.g., in user-space. With the hybrid architecture, eVNF allows building fast and flexible VNFs. We used eVNF to build three prototype VNFs: Firewall (eFW), Deep Packet Inspection (eDPI), and Load Balancer (eLB). We evaluated these VNFs in two service function chains using OpenStack. Our experiments showed that eVNF can significantly improve service throughput as well as reduce latency and CPU usage.**

***Keywords*—Network Function Virtualization, eBPF, eXpress Data Path, OpenStack, Firewall, Deep Packet Inspection, Load Balancer**

I. Introduction

A common network infrastructure consists of many components called network functions (NFs), such as Routers, Firewalls, Deep Packet Inspection (DPI), Intrusion Detection Systems (IDS), Load Balancers, Network Address Translation (NAT), etc. In legacy networks, network service providers deploy these NFs as physical network appliance per each function with tightly coupled, dedicated and proprietary hardware-software. This leads to several problems: a high cost to buy, upgrade and maintain network infrastructure; dependency on the hardware vendors; and difficulties to quickly cope with fast changes on network services nowadays. The consequences are high capital expenditures (CAPEX), high operational expenditures (OPEX), and profit reduction for the network service providers.

Network Function Virtualization (NFV) [1] has been proposed as a solution to solve these problems. NFV decouples NFs from their dedicated hardware into software Virtual Network Functions (VNFs) and enables VNFs to run on commodity servers. By moving NFs into software that can run on any standard hardware, NFV brings several advantages: reducing the dependency on proprietary hardware for network service providers and operators; optimizing space for deployment of the physical network equipment and reduce network power consumption; and enabling flexibility in deployment and management, which allows network operators to get network upgrades and cope with changes in network service demands easier. As a result, NFV helps service providers to reduce both CAPEX and OPEX.

In many cases, a VNF is a piece of software that runs on top of the Linux kernel and kernel networking stack. In the context of this paper, we call this VNF a traditional (or legacy) VNF. The performance of legacy VNFs may be degraded compared to hardware NFs, considering that the generic Linux kernel networking stack does not target very-high performance tasks. In recent years, a kernel framework called extended Berkeley Packet Filter (eBPF) [2] has been developed, which allows injecting a small program into the kernel and process packets at the low level of the kernel networking stack, thus allowing high-throughput and low-latency packet processing. eXpress Data Path (XDP) [3] is a special type of eBPF program that processes packets at the lowest level of the networking stack and provides even better efficiency. Nevertheless, to ensure the stability and security of the kernel, the flexibility of eBPF/XDP is strictly limited compared to generic software programs. Consequently, building general VNFs with eBPF/XDP is still a challenge.

To address the above problem, we propose eVNF - a hybrid architecture to build and accelerate VNFs with XDP. Our contributions are:

- We propose eVNF - a hybrid architecture to build high-throughput and low-latency VNFs with XDP. eVNF partitions the task of a VNF into two parts: simple but critical tasks are processed in an XDP program, while the other complex tasks are processed outside of the XDP program (in a kernel module or a user-space program). We will discuss in detail various aspects of the architecture in this paper.
- To prove the feasibility of eVNF, we use it to implement three widely-used VNFs: Firewall (eFW), Deep Packet Inspection (eDPI), and Load Balancer (eLB). We chained these VNFs in two SFCs using OpenStack and evaluated them with respect to three aspects: throughput, latency, CPU usage. Our evaluation results showed that eVNF can significantly and simultaneously increase throughput, reduce latency and CPU usage of VNFs. We also evaluated

978-3-903176-24-9

the effect of using Data Plane Development Kit (DPDK) for inter-VNF networking and combining multiple VNFs in one VM.

This paper extends our recent work [4] in several aspects. Firstly, while [4] only considers one VNF in each VM, this work considers how to apply eVNF in the case of using multiple VNFs in one VM, which can provide better throughput and latency as well as resource usage efficiency. Secondly, we compare the performance of eVNF when using and when not using acceleration method for inter-VMs networking, such as DPDK. In both cases, eVNF significantly improve the throughput and latency. Thirdly, we discuss the scalability of eVNF. Finally, we compare our approach with OpenFlow [5]; they look similar that the first glance, but have fundamental differences.

The remainder of this paper is organized as follows. In Section II, we present background and related work. eVNF architecture is described in detail in Section III. In Section IV, we present eFW, eDPI, and eLB as prototype VNFs using eVNF architecture. Experiment and evaluation are given in Section V. Finally, Section VI concludes this paper and discusses our future work.

II. Background and Related Work

A. Acceleration methods for NFV

There are several methods to improve NFV throughput and latency. The first method is using highly-optimized system-wide NFV frameworks, such as E2 [6] and NetBricks [7]. With the observation that a VNF usually needs to parse the packet structure first when a packet enters the VNF, E2 uses a dedicated block to parse the packet structure only once, then passes the parsed information as metadata to the next VNFs. NetBricks accelerates NFV by removing the cost of VM virtualization and uses compile-time and run-time checking to ensure VNF isolation instead. Both E2 and NetBricks use zero-copy to further accelerate packet I/O. They also have their own management systems to manage and optimize the VNF chaining process. E2 and NetBricks are complete NFV MANO systems with their own APIs to build VNFs, hence they are not compatible with other NFV MANOs such as OpenStack. In contrast, VNFs built using eVNF architecture should be able to work with any generic VM-based MANO systems.

The second method is to offload VNFs or parts of VNFs to dedicated hardware. NBA [8] and GPUNFV [9] can use GPU to accelerate packet processing. ClickNP [10] and HYPER [11] can offload packet processing into Field Programmable Gate Arrays (FPGAs). UNO [12] can offload packet processing into smart NICs. While our work does not exploit and hardware offloading feature yet, it has been shown that eBPF/XDP programs can be offloaded into smart NICs for throughput improvement [13]. The third method is using a framework for high-performance packet processing, which is covered in the next subsection.

B. Software frameworks for packet processing

Generally, a VNF is built as software that runs on top of the Linux kernel with generic networking stack (Fig. 1). The VNF then can be put in a physical host, a Virtual Machine (VM), or a container. The Linux kernel network stack is programmed for general usage, hence it is complex and has many processing layers. Also, if the VNF runs in user-space, there is a high cost to pass packets from the kernel space to the VNF (kernel-user space context switching). As a result, for the high-rate packet processing tasks, the cost to bring the packet from the Network Interface Card (NIC) to the VNF becomes significant. To solve this problem, there are two methods: putting the VNF at the low-level of networking stack inside the kernel, or using kernel-bypass to bring packets directly from NIC to the VNF.

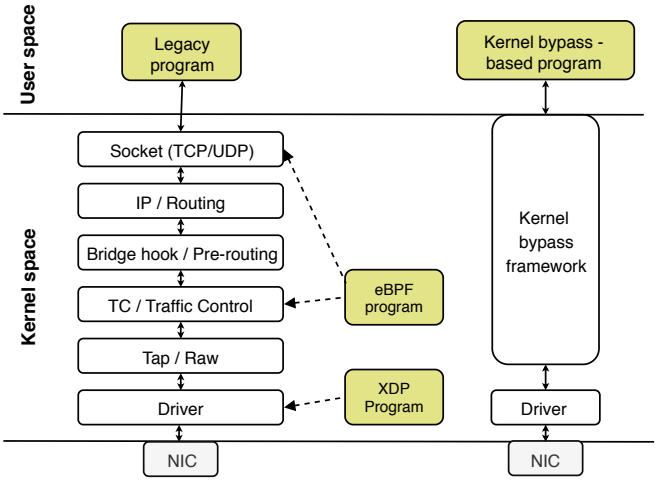

Fig. 1. From left to right: kernel networking stack, eBPF/XDP in-kernel, and kernel-bypass

eBPF/XDP are in-kernel packet processing software frameworks (Fig. 1). eBPF allows injecting a custom program into the kernel as a small in-kernel virtual machine. This program will run when a specific kernel event is activated (such as packet sent or received). In networking, eBPF allows packet processing at low-level of the kernel networking stack (kernel Tun-Tap, Traffic Control, or Socket layer), avoiding the penalty of kernel networking stack traversal and kernel-user space switching, thus can accelerate packet processing. XDP is a special type of eBPF program that provides access to the lowest layer of the kernel network stack (at NIC ring in the driver layer), which allows even higher performance. eBPF is integrated into Linux kernel from v3.15, XDP is integrated into the kernel from v4.8.

DPDK [14] is a popular kernel-bypass packet processing framework (Fig. 1). Currently, DPDK performs better than XDP in packet processing [3]. However, when it comes to NFV, DPDK has limitations. DPDK requires a dedicated CPU core for polling packets. This could be beneficial when the polling CPU serves a networking infrastructure with multiple VMs, but could be overkill when the polling CPU only serves one VNF VM. Each VM using DPDK needs at least one vCPU

for DPDK packet polling, so when the number of VNF VMs increases, the number of DPDK-dedicated vCPUs for packet polling is increased. DPDK also requires dedicated huge-page memory for packet polling, thus the amount of DPDK-dedicated RAM also becomes large when the number of VNF VMs increases. XDP does not have these two limitations. Also, DPDK is a kernel-bypass framework, hence it comes with a high cost to re-inject packets to the kernel when necessary. XDP can pass packets to the kernel networking stack with a nearly-zero performance hit instead, hence can easily leverage traditional VNFs built on the generic networking stack.

C. eBPF/XDP limitations

Although eBPF/XDP enables in-kernel network programmability, its flexibility and capability are limited to ensure the stability and security of the kernel [15]. The size of an eBPF/XDP program is limited, the allowed programming language is only a small subset of C. More importantly, an eBPF/XDP program is very restricted from using kernel services. Only a small number of helper functions is provided, and no user space or third-party service can be used.

D. Related work on eBPF/XDP

In networking, eBPF/XDP can be used to accelerate network infrastructure [16, 17, 18]. BPFabric [16] proposes an in-kernel programmable network data plane with eBPF that targets to replace traditional data planes such as Open vSwitch. As a side work, it also demonstrates that some light-weight VNFs can be programmed with eBPF (such as network telemetry and lightweight anomaly detection). However, BPFabric does not consider the eBPF limitation and does not explain how to build more complex VNFs. IOVisor-OVN [18] is another work that tries to replace the OvS back-end with eBPF/XDP. Cilium [17] uses eBPF/XDP to implement the networking infrastructure for containers. While also using eBPF/XDP, IOVisor-OVN and Cilium are orthogonal to our work. IOVisor-OVN and Clilium are for VM-VM and container-container networking, respectively; eVNF is for building high-performance VNFs.

Recently, there are several efforts that take advantage of XDP to implement specific VNFs such as Firewall [19, 20], DDoS Protection [21] and Load Balancer [22] with greater performance compared to traditional solutions. However, these works focus on specific VNFs and do not provide a general architecture for using XDP in various VNFs. They also do not evaluate their XDP-based VNFs in SFCs.

III. eVNF Architecture

To tackle the XDP constraints and build fast and flexible VNFs, we propose an architecture for VNF based on XDP using a hybrid approach, called eVNF (Fig. 2). In its basic form, a hybrid VNF is divided into two parts: the low-layer part, which is an XDP program in kernel space; and the upper-layer part, which can be a kernel module or a user space program. We call the low-layer part the fast path, and the upper-layer part the slow path. The fast path can process packets with high performance, but it has the limitations of an XDP program. The slow path offers lower performance than the fast path but does not have XDP's limitations. The fast path XDP program is attached to a network interface that receives packets and is be executed when packets are received. When a packet comes in, it is first processed by the fast path. If there is no indication (rule) of how to process the packet in the fast path, the packet is passed to the slow path for further processing. The slow path program then processes the packet and installs a rule to the fast path so that the next packets can be processed directly in the fast path to avoid the cost of kernel-user space switching and kernel networking stack traversal.

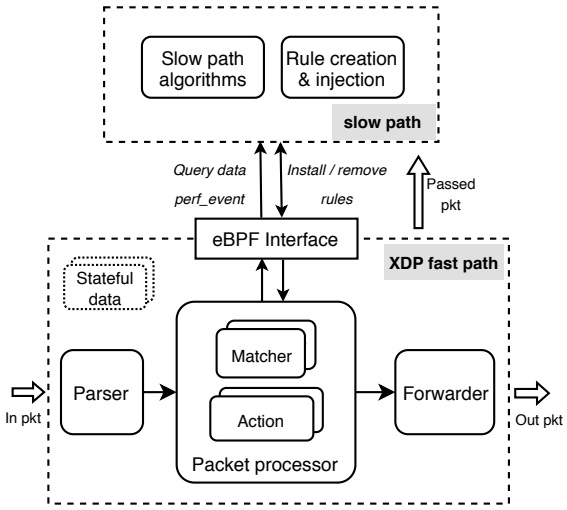

Fig. 2. eVNF Architecture

A. Fast path processing flow

Packet Parser: Firstly, packets which pass through a network interface are filtered in the *Parser* block. This block has two missions: parsing and filtering. *Parser* filters packets to be sent and not to be sent to the next processing step. If a packet passes the filter, the packet structure and other data can be used in the next step.

The parsing sequence goes sequentially from the lowest layer (Ethernet frame) to the highest layer that is required by the VNF (IP, TCP/UDP or application layer). To achieve optimal efficiency, *Parser* should only parse a packet up to the required layer and should terminate the XDP program immediately if the packet does not match the filter.

Packet Processor: This block does the main packet processing in the XDP fast path program. It contains (possibly multiple) *Matcher(s)* and *Action(s)*.

With *Matcher*, the processing decision is decided by the match/action method: the packet metadata (extracted from parsed data and other stateful data) is used as a *key* to find the equivalent entry in the *Matcher*. A *Matcher* is an eBPF table which stores *key-value* entries.

If there is a matched entry, the packet will be processed by the *Action* with the custom parameters provided by the *value* field. If there is no match, the packet will be sent to the slow path. There are two ways to send a packet to the slow path:

- Pass the packet to the kernel stack. The slow path program can then receive and process the packet.
- Redirect the packet to an exclusive virtual network interface (e.g., TAP interface). The slow path listens on this interface to get the packet. This method can be used if the slow path requires a specific interface and does not want to receive packets from other programs.

Also, in many cases, we may need to send custom information to the slow path to aid the processing. In these cases, eBPF provides a special mechanism to push data to the slow path, called *perf_event*. The slow path should continuously poll events from the fast path.

The *Action* should do the simple but critical tasks that should be applied to all packets that match a rule in the *Matcher*. What tasks need to be done here depend on each specific VNF and how the fast path and slow path are defined. For example, in our eLB, *Packet Processor* uses the lookup result from the *Matcher* to select the next server, modifies destination IP and recalculates checksums. *Packet Processor* can also utilize eBPF tables to store stateful data. Those data can be accessed from both the slow path and the fast path.

Forwarder: If the packet passes the *Packet Processor*, *Forwarder* modifies the MAC source and destination, then forward the packet to the next node via a designated interface. *Forwarder* might have an ARP-like table (which is an eBPF table) to look up the destination MAC. This table is maintained by the slow path.

B. Slow path processing flow

The slow path program listens and processes packets sent from the related XDP program. The process can be a routing algorithm, load balancing algorithm or payload analysis, etc., depending on the VNF that the programmer wants to build.

After the slow path processes packets, the rule which indicates how to process incoming similar packets is created in the form of *key-value* and installed into the equivalent *Matcher* in the fast path. Next packets that have similarity with this packet (by matching a *key-value* entry) can be processed directly in the fast path.

C. Fast-slow path communication

Because eBPF tables can be accessed by both fast and slow path, we use it as an interface for bidirectional communication between the fast path and the slow path. Both fast and slow path can asynchronously read and write to eBPF tables. In our architecture, we utilize eBPF tables for two types of interface:

- **Match/action interface**: Install (or remove) rules into (or from) the fast path which indicate how to process next packets.
- **Generic data interface**: Query the stateful data of the fast path program. The stateful data is optional. The generic data interface may be needed if the slow path program needs extra information such as the current state of the XDP program or packet statistics.

D. Split work between fast and slow path

One of the main problems of the eVNF architecture is how to partition work between the fast and slow path. There are three cases which can happen:

- If all work can be done in XDP (e.g., simple load balancing, per-flow statistic, etc.), then this option is optimal for throughput and latency. The slow path program is optional and may only be used for initial configuration.
- If the VNF is complicated but can be divided into fast-path and slow-path (usually, when a group of packets can be processed in the same way as a flow), then the hybrid method can be applied (e.g., traffic classification, load balancer, firewall, dynamic monitoring, etc.)
- If the VNF requires complex processing that is different for all packets or requires access to the payload of all packets (which means little or no work can be done in XDP), then we fall back to legacy VNF (e.g., Deep Content Inspection).

The boundary between these methods is not always clear (e.g., a load balancer can be done either fully in-kernel or hybrid). It depends on the VNF function requirement to choose the suitable method (e.g., if the load balancer only uses a simple static scheduling algorithm, then we can implement every task in XDP, but if the load balancer uses a more sophisticated algorithm, then the hybrid method is the way to go).

E. Working mode

There are three working modes depending on how the processing rules are installed into the fast path.

- **Proactive**: In this mode, rules are installed into the fast path without the need of sending packets from the fast path to the slow path. This working mode provides the best efficiency.
- **Reactive**: In this mode, rules are installed into the fast path after there is a new packet flow sent to the slow path (which means the fast path does not have an equivalent rule for this packet yet). This working mode provides better flexibility in implementing VNF algorithms.
- **Offloading**: eVNF architecture can offload some part of packet processing from a legacy VNF to XDP with little or no modification in the legacy VNF. In this mode, if the packet is not suitable to be processed in XDP, it is passed to the legacy VNF, and no rule will be installed into the fast path. For example, we can offload simple firewall rules into XDP, while complicated rules (e.g., with connection tracking) can be put in iptables.

F. Performance bound and scalability

Because we do not have prior information on how many packets will be processed by the fast path and by the slow path, we need to consider the performance hit in the case all packets go through the slow path. Because XDP is designed to work in conjunction with the kernel networking stack, passing packets from XDP to upper kernel layers has negligible cost. The XDP program size is also limited to ensure that an XDP program can process packets quickly enough. As a result, the lower bound of eVNF is nearly the same as the performance of legacy VNFs (will be shown later in our evaluation). This also ensures the performance of the offloading mode of eVNF.

Regarding the scalability, XDP does scale well with the number of CPU cores [3]. XDP works well with Linux scalability mechanisms such as Receive Side Scaling (RSS), which distributes packets over CPU cores. Hence, the scalability of eVNF depends on how programmers design the slow path.

G. eBPF vs. XDP

Our architecture can be applied to both eBPF and XDP. Choosing whether to use eBPF or XDP to implement the fast path depends on the VNF requirements. XDP offers higher throughput and lower latency, but it only works in ingress direction and does not work for egress direction. XDP also lacks several capabilities compared to eBPF, such as packet cloning.

H. Service function chain

One important aspect of NFV is the ability to chain VNFs together to provide network services. In this work, we consider the case that VNFs are deployed in VMs. When deployed with the ratio of 1 VNF - 1 VM, eVNF-based VNFs are compatible with NFV MANOs that use VM-based chaining, such as OpenStack [23].

In some cases, users may want to put several VNFs that usually come together into the same VM. This is supported by the Linux networking stack, but can be a problem with XDP: there should be only one XDP program that directly listens to a network interface at a time. The solution is using eBPF/XDP *tail-call*, a special mechanism that allows multiple XDP programs to be chained together. However, in eVNF, packets can travel either through the fast path or the slow path, hence we use *tail-call* to chain the fast path programs and let the kernel network stack processes the slow path of VNFs. The packet path thus can be complicated and the chaining should be handled carefully. Otherwise, some packets may not pass through the expected path. The maximum number of XDP programs in a chain with *tail-call* is 32, which is more than enough for a typical SFC.

I. Comparison with OpenFlow

Our approach may look similar to how OpenFlow network operates: using fast path and slow path, using match/action tables. A packet is processed inside the eBPF kernel program (equivalent to the OpenFlow switch) if there is a matched entry or sent to user space program (equivalent to the OpenFlow controller) if there is no match. However, our approach is applied at a smaller scale: in one host machine instead of the whole network. There are two main differences compared with OpenFlow:

- The switching time between user space and kernel space in XDP is much lower than the communication delay between OpenFlow controller and OpenFlow switch.
- OpenFlow has fixed functions that are indicated in its specification (no programmability), while XDP is programmable. Users can program any custom actions they want within the capability of XDP.

These two differences enable building high performance VNFs with eVNF architecture using eBPF/XDP.

IV. VNF Implementation with eVNF

In this section, we briefly present the overview, design, and implementation of three VNFs using eVNF architecture: Firewall (eFW), Deep Packet Inspection for traffic classification (eDPI), and Load Balancer (eLB). For prototype builds, we use BPF Compiler Collection[1], a toolkit for building eBPF/XDP-based applications. We also show how to integrate eVNF with OpenStack. The source code is available on Github [2].

A. eFW

Iptables is widely used for configuring the Linux kernel firewall. With iptables, users can create flexible filtering rules to filter traffic based on flow characteristics and connection state. One problem with iptables is that it is based on netfilter, so the packets still need to traverse several kernel networking layers before being processed. The other problem is that iptables uses sequence lookup (it scans every rule until finding a matching entry), this makes it slow if the matched rule is not in the head of the list. There are several works that tried to solve these two problems by replacing iptables with eBPF/XDP [19, 20], but not yet succeeded in fully replacing iptables. Iptables is a very complex system that is hard to be fully implemented in XDP, due to its programmability limitation.

eFW tries to solve the first problem. With the observation that a firewall might have some common simple rules [20] (e.g., open port 80/443 to public access, open others for internal network only), eFW offloads these common simple rules to XDP, and let the other packets still be processed by iptables rules. We implemented eFW with a *Matcher* to lookup for port, and another *Matcher* to lookup for subnet. Any packets that are not matched will be passed to iptables. In our design, eFW works in offloading mode and netfilter is considered as the slow path. Note that rules in eFW fast path have higher priority than rules in iptables.

[1]BPF Compiler Collection, https://github.com/iovisor/bcc

[2]eVNF, https://github.com/dpnm-ni/ni-evnf.git

B. eDPI

Deep Packet Inspection (DPI) for Traffic Classification (DPI-TC) is used to classify traffic in the network (e.g., email, VoIP, video, etc.) and is important for monitoring and accounting.

For traffic classification, there is no need to do the analysis for the packet belonging to a flow that is already classified. Open-sourced DPI-TC such as nDPI [24] only uses several first packets to detect the application protocols. Packets that come to nDPI and belong to classified flows are discarded. Although nDPI does not process classified packets, there is still a cost of kernel networking stack processing and kernel-user space switching.

We propose eDPI to solve the above problem. Because of the XDP limitations, we cannot develop an DPI engine inside the kernel with XDP. However, by applying eVNF, we can use XDP to improve DPI-TC by only sending the unclassified packets to the DPI instance.

We implemented eDPI based on nDPI, using eVNF architecture in reactive mode. In the fast path, we create a *Matcher* that matches 5-tuple flows to filter the classified and unclassified traffic. Only unclassified traffic is passed to the slow path (which is the nDPI instance), while classified traffic is directly forwarded to the next target. In the slow path, nDPI detects the traffic, creates a 5-tuple rule for the classified flow, and installs it into the fast path, thus next packets that match the 5-tuple will not be passed to nDPI. Not all flows but only 5-tuples of long elephant flows (such as video, multimedia, VoIP) should be injected into the fast path.

C. eLB

It is common to have multiple application servers that serve one service. A load balancer distributes requests to these servers using a scheduling algorithm. There are several popular software load balancers such as HAproxy [25]. These load balancers are implemented in user space, they are flexible but slow compared to eBPF/XDP based solutions like Katran [22].

By applying eVNF, we can use XDP to implement eLB with more flexible load balancing algorithms, such as dynamic feedback scheduling [26]. Dynamic feedback scheduling takes real-time server load into account. It periodically queries the server loads and re-calculates the traffic distribution to each server. Details of the algorithm are presented in [26].

In eLB, the fast path calculates the hash of source IP and port, matches them in a server selection *Matcher* to get the next server IP address. However, to ensure the consistency of connections, eLB has a connection tracking table, so that a packet that belongs to an already established connection will be forwarded to the correct server. The slow path periodically queries servers to get real-time statistics, and changes the server selection *Matcher*. The server selection *Matcher* is an array with each entry corresponding to one server. By changing the number of entries for each server, we can change the percentage of requests sent to each server. eLB can also work in offloading mode with HAproxy, we do not use it in our testbed though.

eLB uses Simple Network Management Protocol (SNMP) to get the average server load information. We only consider CPU load for our prototype. We use the Destination Network Address Translation (DNAT) method for redirecting the packets (which means the eLB only modifies the destination header fields of incoming packets and source header fields of outgoing packets).

D. OpenStack integration

Although XDP has a generic mode that can work with any NIC driver, XDP requires supported NIC drivers to achieve optimal performance. For eVNF to work optimally with OpenStack, users should use Kernel Virtual Machine (KVM) as the hypervisor and use *virtio-net* as the virtual network interface for guest VMs. *Virtio-net* driver is designed specially to accelerate VM networking and is supported by XDP from kernel v4.10.

In OpenStack, DPDK can be used to accelerate the network infrastructure. While eBPF/XDP based solutions such as IOVisor-OVN are available, the performance still lags behind DPDK [3]. Users can take the best of both worlds: using DPDK for inter-VM networking and using XDP to accelerate VNF packet processing.

V. Evaluation

In this section, we evaluate three eVNF-based VNFs in two SFCs. We deploy them with OpenStack [23], an open-source software platform for cloud computing and Network Function Virtualization. We evaluate our eVNF-based VNFs regarding three aspects: latency, throughput, and CPU usage; and compare them with equivalent legacy VNFs. In each test, throughput, latency and CPU usage are measured simultaneously. We also evaluate the effect of DPDK and the effect of combining multiple VNFs in one VM.

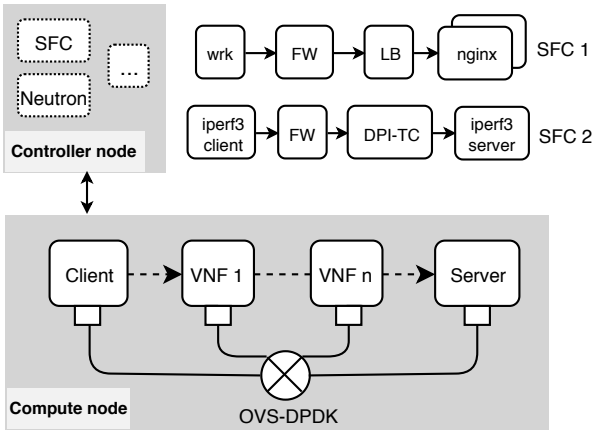

Fig. 3. Testbed setup

In our testbed, we use OpenStack Rocky version with one controller node and one compute node (Fig. 3). All controller

services are installed on a separate node. All VMs and VNFs are deployed on one OpenStack compute node, which does not represent the real environment, but does not affect the purpose of our experiment. Each VNF is deployed in one separate VM. Open vSwitch (OvS)[3] is used for inter-VM and VNF networking. All VMs and VNFs run Ubuntu server 16.04 with kernel v4.14. Each VNF is allocated 1 vCPU and 1 GB of RAM. The OpenStack compute node is a Dell R610 server with 2 Intel Xeon X5650 CPUs and 24GB RAM, distributed in 2 NUMA nodes. Hyper-threading is enabled.

To create SFCs, we first used OpenStack *sfc classifier* to filter the traffic that needs to travel the SFCs, then used OpenStack *sfc port chain* to chain VNFs together.

A. Web service SFC: firewall - load balancer

It is common to deploy a firewall and load balancer at the front face of a web service. We created an SFC with a firewall and a load balancer between clients and servers (Fig. 3 SFC 1). Each client/server VM has 4 vCPUs. We used two servers that run nginx[4]. The client used wrk[5] to send HTTP GET requests to servers, the servers then send HTTP responses with no accompanying data. The number of parallel connections was changed during the test. We used DPDK to accelerate OvS. In the first test, we used iptables firewall and HAproxy load balancer (running in layer 4 mode). In the second test, we used our eFW and eLB. We also measured the average CPU usage in the firewall and load balancer instances during the test.

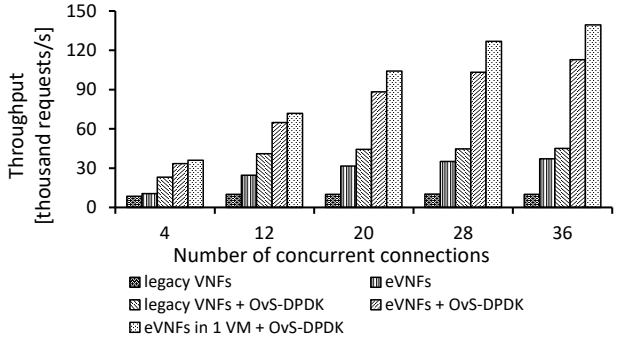

Fig. 4. Throughput measured in SFC 1

In general, eVNF simultaneously increases the throughput (Fig. 4) and reduces end-to-end latency (Fig. 5) in all cases. eVNF improves the throughput (number of completed requests per second) by 1.4x times when the number of concurrent connections is 4, and by up to 2.5x times when the number of connections is 36. When eVNF is not used and the number of connections exceeds 12, the throughput is not increased any more because the CPUs of HAproxy are saturated (Fig. 6). This does not happen in the case of using eVNF. Regarding the end-to-end latency, eVNF also reduces the end-to-end latency by 32% when the number of connections is 4, and by up to about 51% when the number of connections is 36. Note that the end-to-end latency is contributed by not only the VNFs, but also the inter-VM networking and client/server processing.

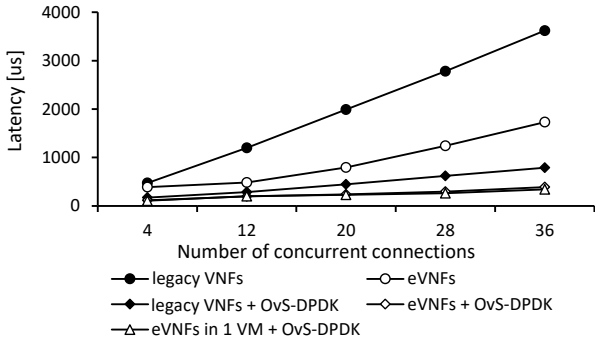

Fig. 5. End-to-end latency measured in SFC 1

eVNF drastically reduces CPU usage, especially in the case of eLB vs. HAproxy (Fig. 6). In the case of eFW and iptables firewall, CPU usage is always below 2%. Although the CPU usage of eFW is always lower than the CPU usage of iptables firewall, the measurement noise might be high, so we are not confident to conclude anything. However, in the case of the load balancer: HAproxy CPU usage is always higher than 90% and is saturated when the number of connections is larger than 20; while eLB CPU usage is always smaller than 2% in all cases. HAproxy layer 4 load balancer works at user-space level, thus when there are a high number of requests, there is a huge cost for kernel networking stack traversal and kernel-user space switching.

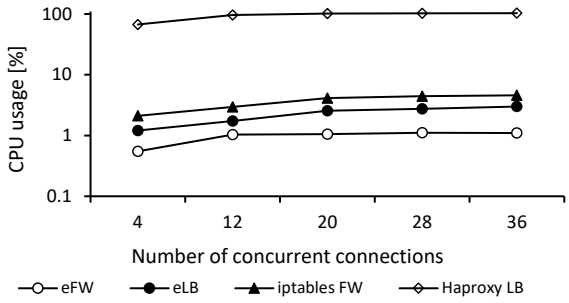

Fig. 6. CPU usage of VNFs in SFC 1

B. Effect of DPDK

In this test, we used the same setup with the web service SFC 1, but switched between OvS (without DPDK) and OvS-DPDK for inter-VNF networking. The cost for switching packets between VNFs is high; using DPDK brings a huge improvement in both throughput (Fig. 4) and latency (Fig. 5). With eVNF-based VNFs, using DPDK improved throughput by about 2.6x to 3.2x times and reduced latency by about 70% to 78%. With traditional VNFs, using DPDK improved throughput by about 2.7x to 4.5x times and reduced latency by about 64% to 78%.

[3]OvS, https://www.openvswitch.org
[4]nginx, https://nginx.org/en
[5]wrk, https://github.com/wg/wrk

Although DPDK has a huge benefit, there are some OpenStack features that do not support DPDK (such as Distributed Virtual Routing). This can lead to performance degradation. If DPDK is not used, eVNF can still be used to accelerate VNFs. When not using DPDK, eVNF increased throughput up to 3.7x times and reduced latency up to 52%. eBPF/XDP-based solutions for inter-VM networking such as IOVisor-OVN can be used, we did not test this in our work though. If DPDK is available to use, the combination of DPDK and eVNF provides the best performance.

C. Multiple VNFs in one VM

Firewall and load balancer usually come together in an SFC for web services, hence it might be beneficial to put them in the same VM. In this test, we used the same setup with the web service SFC 1, but put eFW and eLB in the same VM and chained them using XDP *tail-call*. Compared to separating eFW and eLB into two VMs, putting them in one VM improved the throughput up to 1.2x times (Fig. 4) and reduced the latency by up to 12% (Fig. 5), while reducing the total number of vCPUs used from 2 to 1.

D. Network provider SFC: Firewall - Traffic Classification

Network providers might need to use DPI-TC for traffic accounting. We created a SFC with a firewall and a DPI-TC (Fig. 3 SFC 2). OvS-DPDK is used. Both client and server used iperf3[6]. Each client/server VM has 4 vCPUs. The traffic contains 4 iperf3 flows with equal throughput. In the first run, we used eFW and eDPI. To emulate the different percentage of elephant flows in the traffic, we modify the eDPI protocol signature to detect each iperf flow as elephant or non-elephant traffic flow. The percentage of elephant traffic is switched between 0%, 25%, 50%, 75%, and 100%, which are equivalent to 0, 1, 2, 3, and 4 elephant flows, respectively. We used eFW and nDPI for the second run. For the final run, we used iptables firewall and nDPI.

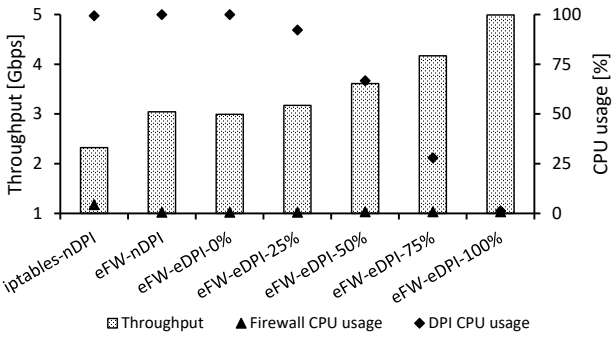

Fig. 7. Throughput & CPU usage in SFC 2

With eVNF, throughput and CPU usage are improved when the percentage of elephant flows is increased. When there is no elephant flow offloaded to XDP, maximum throughput is 3.0 Gbps and the CPU usage of eDPI instance is saturated. When the percentage of elephant flows is 50%, the throughput increases to 3.6 Gbps and CPU usage of eDPI decreases to 66%. In the ideal case when all traffic flows are elephant flows, the throughput increases to 5.0 Gbps and the CPU usage of eDPI reduces to about 1%.

When we keep eFW but switch the DPI between nDPI and eDPI, the throughput in the nDPI case (3.04 Gbps) is nearly the same as the eDPI case without any elephant flows (3.0 Gbps). This shows that XDP has a negligible performance impact on eDPI. Most of the CPU usage goes to kernel networking stack and DPI engine.

When we keep nDPI but switch from eFW to iptables firewall, the CPU usage of firewall is very small in both cases, but the throughput is reduced (from 3.04 Gbps to 2.3 Gbps). In the Iptables firewall, packets need to travel several layers in the kernel networking stack (including skb buffer allocation), which has negligible cost in terms of CPU, but might increase latency and have a significant impact in the context of high-throughput traffic.

VI. Conclusion

In this paper, we proposed eVNF - a hybrid method for building VNFs that leverages the speed of XDP in-kernel packet processing and the flexibility of generic software programs, enabling building complex VNFs with high performance. We discussed various aspects of eVNF architecture. To prove the feasibility of eVNF, we have developed three prototype VNFs with the hybrid method: eFW, eDPI, and eLB. The evaluation results showed significant improvement in throughput, latency and CPU usage compared to traditional solutions.

In this work, we only consider VM-based VNFs. Nowadays, container-based NFV, which has its own advantages over VM-based NFV, is becoming popular. While eBPF/XDP can be used for container networking [17], whether eVNF architecture can be applied for containers is still an open question. Also, we have not yet done the evaluation of eVNF with SR-IOV, which could further improve the performance. We plan to solve these issues in our future work.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (2018-0-00749, Development of Virtual Network Management Technology based on Artificial Intelligence).

References

[1] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," in *IEEE Communications Surveys Tutorials*, vol. 18, 2016, pp. 236–262.

[2] A. Starovoitov, "BPF – in-kernel virtual machine," *Linux Kernel Developers' Netconf*, 2015.

[3] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Pro-*

[6]iperf3, https://github.com/esnet/iperf

ceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT), 2018, pp. 54–66.

[4] N. Van Tu, J. Yoo, and J. W. Hong, "eVNF - Hybrid Virtual Network Functions with Linux eXpress Data Path," in *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2019.

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

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

[7] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, 2016, pp. 203–216.

[8] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, 2015.

[9] X. Yi, J. Duan, and C. Wu, "GPUNFV: A GPU-Accelerated NFV System," in *Proceedings of the First Asia-Pacific Workshop on Networking (APNet)*, 2017, pp. 85–91.

[10] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 1–14.

[11] C. Sun, J. Bi, Z. Zheng, and H. Hu, "HYPER: A Hybrid High-Performance Framework for Network Function Virtualization," in *IEEE Journal on Selected Areas in Communications*, vol. 35, Nov 2017, pp. 2490–2500.

[12] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. V. Lakshman, "UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017, pp. 506–519.

[13] K. Jakub and B. David, "eBPF/XDP," in *ACM SIGCOMM '18 Tutorial*, 2018.

[14] Data Plane Development Kit (DPDK), `http://dpdk.org`.

[15] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network service with eBPF: Experience and lessons learned," *High Performance Switching and Routing (HPSR)*, 2018.

[16] S. Jouet and D. P. Pezaros, "BPFabric: Data Plane Programmability for Software Defined Networks," in *Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2017, pp. 38–48.

[17] Cilium, `https://github.com/cilium/cilium`.

[18] A. Zaafar, M. H. Alizai, and A. A. Syed, "Coupling the Flexibility of OVN With the Efficiency of IOVisor: Architecture and Demo," *OvS 2016 Fall Conference*, 2016.

[19] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating Linux Security with eBPF Iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, New York, NY, USA, 2018, pp. 108–110.

[20] D. Anant, G. Shankaran, H. Richard, and M. Puneet, "eBPF / XDP based firewall and packet filtering ," in *Linux Plumbers Conference*, 2018.

[21] G. Bertin, "XDP in practice: integrating XDP in our DDoS mitigation pipeline," in *NetDev 2.1 - The Technical Conference on Linux Networking*, 2017.

[22] Katran, `https://github.com/facebookincubator/katran`.

[23] Openstack, `https://www.openstack.org`.

[24] L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano, "nDPI: Open-source high-speed deep packet inspection," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, Aug 2014, pp. 617–622.

[25] HAproxy, `http://www.haproxy.org`.

[26] Dynamic Feedback Load Balancing Scheduling, `http://kb.linuxvirtualserver.org/wiki/Dynamic_Feedback_Load_Balancing_Scheduling`.