Architecture for Building Hybrid Kernel-User Space Virtual Network Functions

Nguyen Van Tu, Kyungchan Ko, and James Won-Ki Hong
Dept. of Computer Science and Engineering, POSTECH, Pohang, Korea
Email: {tunguyen, kkc90, jwkhong}@postech.ac.kr

***Abstract*—Network Function Virtualization (NFV) is one of the important aspects of modern network architecture. NFV decouples Network Functions (NFs) from hardware, therefore produces Virtual Network Functions (VNFs) that can run on standard, commodity servers, which in turn mostly run Linux kernel. In this paper, we propose a general architecture for building hybrid kernel-user space VNFs which leverages extended Berkeley Packet Filter (eBPF). eBPF is a framework in Linux kernel that enables network programmability inside kernel for optimal performance. However, the programmability of eBPF is limited due to safety and security of the kernel. Our proposed architecture applies hybrid approach: leave the simple work inside the kernel with eBPF and let complex work be processed in the user space. This architecture allows building complex VNFs to have both speed and flexibility. To demonstrate, we use the proposed architecture to build two VNFs: Dynamic Load Balancer and Deep Packet Inspection with Dynamic Sniffing. The evaluation results show that both VNFs significantly outperform the widely used solutions.**

***Keywords*—Network Function Virtualization, Virtual Network Functions, extended Berkeley Packet Filter**

I. Introduction

A common network infrastructure consists of many components called network functions (NFs), such as the router, firewall, encryption, deep packet inspection (DPI), load balancer, network address translation (NAT), domain name service (DNS), etc. In a legacy network, network service providers deploy these NFs as a physical network appliance per each function with tightly coupled, dedicated and proprietary hardware-software, which makes them depend on the hardware vendors. Also, the network infrastructure is becoming more complex, more on-demand network functions are going to be added and changed. This leads to high cost to buy, upgrade and maintain network infrastructure, as well as difficulties to quickly cope with sudden network changes.

Network Function Virtualization (NFV) [1] is the solution for these problems. NFV decouples NFs from their dedicated hardware into software Virtual Network Functions (VNFs) and enables VNFs to run on virtual machines (VMs) on commodity servers. By moving NFs into software that can run on any standard hardware, NFV can reduce the dependency on proprietary hardware for network service providers and operators. As a result, service providers can reduce both capital expenditures (CAPEX) and operating expenditures (OPEX). NFV can optimize space for deployment of the physical network equipment and reduce network power consumption. NFV enables flexibility in deployment and management, which allows network operators to get network upgrades easier and cope with changes in network service demands in a more agile manner.

In most cases, a VNF is a piece of software that runs on top of the Linux kernel, regardless of the deployment method (on a virtual machine or a container). Thus, the performance may be drastically degraded compared to hardware NF. To increase the performance, a VNF can be built as a kernel module. However, a kernel module costs much more effort to maintain, takes more time to merge into upstream kernel, and can affect the stability and security of the kernel. Recently, a kernel framework called extended Berkeley Packet Filter (eBPF) [2] has been developed, allowing in-kernel programming from user space. Nevertheless, to ensure the stability and security of the kernel, the flexibility of eBPF is strictly limited compared to the user space program. Therefore, building a VNF with both speed and flexibility is still a challenge.

To address that problem, we propose a hybrid method for building VNFs with eBPF. Our contributions are:

- We present a general architecture that applies the hybrid method for building VNFs: partition the work between kernel space and user space. The architecture leverages both the performance of the eBPF in-kernel program and the flexibility of the user space program, thus allows developing complex VNFs with high performance. We will discuss in details various aspects of the architecture in this paper.
- To prove the feasibility of our architecture, we apply our method to develop two widely used VNFs: a Dynamic Load Balancer (LB) and Traffic Classification DPI (TC-DPI) with Dynamic Packet Sniffing. The evaluation results in both cases show that our approach significantly outperforms the currently used solutions.

The remainder of this paper is organized as follows. In Section II, we present background and related work. A general architecture of our hybrid method for building VNFs with eBPF is described in Section III. In Section IV and V, we present the Dynamic LB and the Dynamic Sniffing DPI as examples which can be implemented by using our method, respectively. Experiment and evaluation are given in Section VI. Finally, Section VII concludes this paper and discusses our future work.

978-3-901882-98-2 © 2017 IFIP

II. Background and Related Work

A. Software network data plane

Extended Berkeley Packet Filter (eBPF), eXpress Data Path (XDP) [3], and Data Plane Development Kit (DPDK) [4] are three software frameworks that enable high performance network data plane. eBPF is designed to be safe and secure for the kernel, but also to provide enough capability to do various kernel work such as kernel tracing and networking. By allowing the program to execute directly inside the kernel, eBPF can avoid the penalty of kernel-user space switching cost. Also, in networking, eBPF allows packet processing at low-level of kernel networking stack (kernel Tun-Tap, Traffic Control, or Socket layer), thus improves the performance even more. XDP can be considered as an eBPF's extension, which provides access to lower network stack (Driver layer) but has more limited programmability than eBPF. Our architecture can be applied to XDP without any modification. DPDK can achieve high performance by bypassing the kernel network stack and processing packets entirely in the user space. However, compared to DPDK, eBPF and XDP have their advantages [3]: they do not require large pages or dedicated CPUs, they can leverage the Linux kernel security model, etc.

IOVisor [5] is an open source project that leverages eBPF and XDP to enable virtualized in-kernel IO services for kernel tracing, monitoring, security, and networking functions. The most important sub-project of IOVisor - BPF Compiler Collection (BCC) [6] - is a framework to develop and deploy eBPF program quickly. In our work, we have used BCC to develop our demonstration use cases.

B. eBPF limitation

Although eBPF enables in-kernel network programmability, its flexibility is limited to ensure the stability and security of the kernel, and still much less than a fully featured user space program. The total number of instructions in an eBPF program is limited, the looping instruction is limited, and the language is only a small subset of C language. More important, an eBPF program is very restricted from using kernel service. Only a small number of eBPF helper functions is provided, and no user space or third-party service can be used.

C. InKeV and BPFabric

InKeV [7] and BPFabric [8] are two recent works on kernel programmable network with eBPF. InKeV presents an in-kernel data plane architecture with eBPF. It mentions that NFs can be implemented in kernel space to avoid the cost of sending packets to user space programs. Although the evaluation shows that this approach can achieve high throughput and flow creation rates, it does not discuss in detail the architecture of VNF with eBPF and does not consider the limitation of eBPF. Thus, this approach is not suitable for complex VNFs (such as Dynamic LB, Deep Packet Inspection, etc.).

BPFabric proposes an in-kernel programmable network data plane with eBPF that targets to replace traditional data plane 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, like InKeV, BPFabric does not consider the eBPF limitation and does not explain how to build more complex VNFs.

III. General Architecture

To tackle the eBPF constraints and build VNFs with both speed and flexibility, we propose an architecture for VNF based on eBPF using the hybrid approach, which is shown in Fig. 1. In basic form, a hybrid VNF is divided into two parts: the kernel space part - which is the eBPF program, and the user space part. The kernel eBPF program is attached to a networking event such as packet send-receive and will be executed every time an event happens. When a packet comes in, it is first processed by the eBPF kernel program. If there is no indication of how to process the packet in the kernel space, the packet is sent to the user space for further processing. The user space program then processes the packet and installs an indication of processing next packets to the kernel space so that the next packets can be processed directly in the kernel, thus avoid the kernel - user space switching cost.

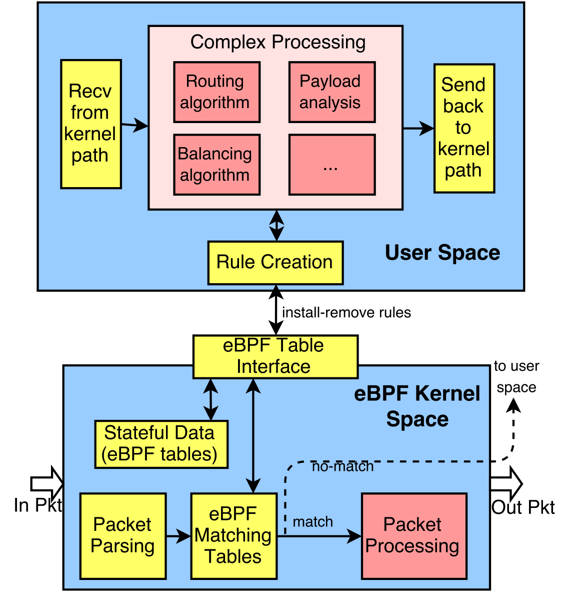

Fig. 1. General architecture of hybrid VNF with eBPF

A. eBPF kernel space processing flow

1) Packet Parsing: Firstly, a packet passes through a network interface is filtered by the eBPF program, at the *Packet Parsing* block. This block has two missions: parsing and filtering. *Packet Parsing* filters which packets need to be passed to next processing step, and which packets do not need to be processed. If a packet passes the filter, the packet structure and other data will be used for matching in the BPF tables.

The parsing sequence goes sequentially from the lowest layer (Ethernet frame) to the highest layer (application layer). *Packet Parsing* should only parse a packet up to the required layer and should terminate the eBPF program immediately if the packet does not match the filter to achieve optimal performance.

2) eBPF Matching Tables: 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 *eBPF Matching Tables*. eBPF matching tables are eBPF tables which store *key-value* entries.

If there is a matched entry, the packet will be processed at *Packet Processing* with the custom parameters provided by the *value* field. If there is no match, the packet will be sent to the user space program for further processing. There are three ways to send a packet to the user space:

- Do nothing to the packet. The packet will pass over the kernel networking stack as a default action. The user space program can then receive and process the packet.
- Redirect the packet to a listener virtual network interface (e.g., tap interface). The user space program listens on this interface to get the packet. This method may increase the performance compared to the first one (since the user space program will not receive packets sent from other eBPF programs).
- Clone the packet and redirect the cloned packet to the listener virtual network interface. This method is needed in some cases when the original packet flow needs not to be modified, such as when the VNF acts as a sniffing instance (e.g., Deep Packet Inspection on a virtual link).

3) Packet Processing: This block does all the packet processing in the eBPF kernel program. In the hybrid architecture, this block should do the simple work that needs to be applied for all packets that match the *eBPF Matching Tables*, including header modifications, packet cloning/forwarding/dropping, packet statistic, checksum calculations, etc. *Packet Processing* can also utilize eBPF tables to store stateful data. Those data can be used in both kernel space and user space program.

B. eBPF user space processing

The user space program listens and processes the packet sent from the related eBPF 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. Then, the packet can be sent back to the kernel space for eBPF processing if required (usually, if the packet is sniffed, then it does not need to be sent back to kernel space).

After the user space program processes the packet, the processing indication is created by *Rules Creation* in the form of *key-value* filter rules and installed into *eBPF Matching Tables* in the eBPF kernel program. Next packets that have similarity with this packet (by matching a *key-value* entry) can be processed directly in the kernel space.

C. Kernel-User space Interface

eBPF provides an interface for the user space program to communicate with the eBPF kernel program: the BPF table interface. Both kernel and user space program can asynchronously read and write to eBPF table. In our architecture, *eBPF table interface* has two missions:

- *Match/action table interface*: Install (or remove) rules which indicate how to process next packets into (or from) eBPF kernel space.
- *Generic data interface*: Query the stateful data of the eBPF kernel program. The stateful data table is optional. The generic data interface may be needed if the user space program needs extra information such as packet statistic.

D. Split work between Kernel-User space

One of the main problems of the hybrid architecture is how to partition work between kernel and user space programs. There are three cases which can happen:

- If the VNF is simple that all work can be done in the kernel (e.g., simple load balancing, monitoring, etc.), then this option is optimal for speed. The user space program is optional and may only be used for 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 (no work can be done in the kernel space), then we fall back to full user space VNF (e.g., Deep Content Inspection).

The first method, which does all work in the kernel, may provide highest performance but may not be suitable for complex VNFs. Also, the boundary between these methods is not always clear, (e.g., a LB can be done in both fully in-kernel or hybrid method). It depends on the requirement to choose the suitable methods (e.g., if the LB only does round robin algorithm, then full in-kernel may be better, but if the LB does more sophisticated algorithm such as dynamic weighted round robin, then the hybrid method is more suitable).

E. Comparison with OpenFlow

Our approach is very similar to how OpenFlow [9] network operates: using match/action tables. A packet is processed inside the eBPF kernel program (equivalent to the switch in OpenFlow) if there is a matched entry or sent to user space program (equivalent to the controller in OpenFlow) 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 eBPF is much less than the communication delay between OpenFlow controller and OpenFlow switch.

- OpenFlow has fixed functions that are indicated in its specification (no programmability), while eBPF provides programmable in-kernel network data plane.

Those two differences enable eBPF to do a variety of VNFs with high performance.

IV. Dynamic Load Balancer

This section presents how a dynamic load balancer can be developed using our architecture. In a modern computing server, it is common to have many application servers deployed on virtual machines or containers. There may be many application servers that serve only one service. A load balancer (LB) distributes requests to these servers, using a scheduling algorithm. Nowadays, with the increasing number of data centers and cloud services, LB is one of the most important NFs.

There are two types of LB: hardware-based LB and software-based LB. Hardware-based LBs have the advantage of processing speed, but they are expensive and not flexible, especially in the current network architecture based on virtualization, where software LBs take advantage. Software-based LBs, in turn, can be divided into user space LBs and kernel space LBs, depending on how they are implemented. The user space LBs, such as HAproxy [10], are flexible but slow compared to kernel space solutions like IP Virtual Server (IPVS) [11] or eBPF solutions like Cilium [12]. IPVS is a static kernel framework, which is too generic (thus slower than a function-specific and light weight module) and hard to extend. Cilium focuses on the container, and because its LB function is fully implemented in the kernel with eBPF, the algorithm flexibility is limited.

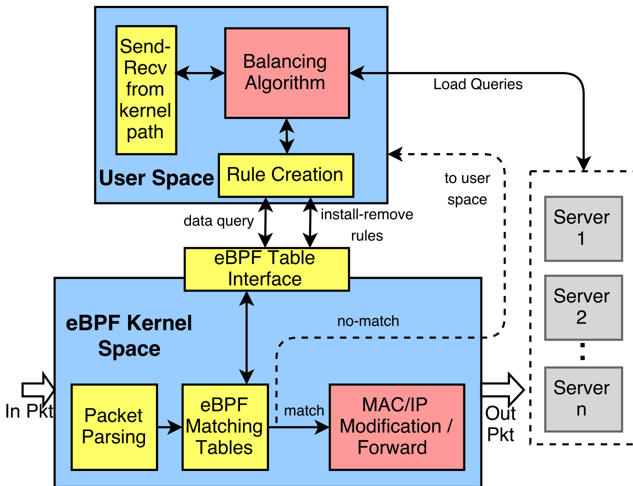

Fig. 2. Load Balancer Architecture with Hybrid eBPF

Fully eBPF in-kernel LBs like Cilium is fast. However, they cannot apply the complex dynamic balancing algorithm (currently Cilium only provide static source-hashing algorithm), which can be a problem when the network and servers condition continuously change. To address this issue, we apply the hybrid approach to our LB.

A. Architecture

The architecture of Dynamic LB is illustrated in Fig. 2. Same as the general architecture, our LB is divided into user space part and eBPF kernel space part. The user space LB manages the server pool, executes dynamic balancing algorithm to choose the next server and creates rules for eBPF matching tables.

In the kernel space, eBPF program process packets at the kernel traffic control layer. LB parses incoming packets in the order of Ethernet, IP and TCP protocols. Then packets are passed to eBPF matching tables. To support bi-directional flow, we create one table for the client-server direction and another for the server-client direction. In the table for the client-server direction, the *key* consists of *Source IP, Source Port, Destination Port* and the *value* consists of *Source MAC, Destination MAC, Destination IP*. In the table for the server-client direction direction, the *key* consists of only *Destination IP* and the *value* consists of *Source MAC, Destination MAC, Source IP*. If there is a matched entry, LB replaces MACs and IPs according to the matched result, and also LB forwards flows to the designated interface for transmitting packets to the server or client. We use the Destination Network Address Translation (DNAT) method for redirecting the packets (which means the LB only modifies the destination header fields). If there is no matched entry, the packet is sent to the user space program to create related rules through balancing algorithm.

B. Dynamic Weighted Round Robin

By putting the balancing algorithm in user space, we can implement flexible and complex balancing algorithms. In this work, we use dynamic weighted round robin. Basically, the weighted round robin algorithm distributes requests from senders based on the weight of each server. For example, when the weights of the servers are $1, 1, 2$ for servers $s1, s2, s3$, respectively; requests from client will be distributed in the order of $s1, s2, s3, s3$.

In Dynamic LB, the Dynamic means that the each server weight is periodically updated in the runtime with regarding the status of each server. In this implementation, we set the weight to the average server free (which equals to the total computation power minus the average server load). For the LB to get the information about the servers, we deploy a Simple Network Management Protocol (SNMP) daemon for each server to report average server load to the LB user space program. LB uses this information to select the next server. Although we use weighted round robin, other weighted algorithms can be easily implemented without significant change.

V. DPI with Dynamic Sniffing

This section presents how a DPI with dynamic sniffing can be developed using our architecture. Deep Packet Inspection (DPI) is another important VNF in current network architecture. It does deep analysis about the traffic and can provide

this information to other VNFs, such as bandwidth manager, firewall, etc.

Traditionally, a DPI instance could be attached to a switch port, and a port-mirroring setup mirrors all packets from one or more switch interfaces to the DPI for analysis. However, for some DPIs such as Traffic Classification (DPI-TC), there is no need to do the analysis for the packet belonging to a flow that is already classified. Even when the DPI-TC does early dropping for classified packets, there is still cost of kernel networking stack processing and kernel - user space switching. An SDN approach could be applied to do some traffic filtering and send only unclassified flow to the DPI-TC [13]. However, with the fixed functions of OpenFlow, the filtering capability and granularity are limited.

Because of the eBPF limitations, we cannot develop DPI engine inside the kernel with eBPF. However, by applying the hybrid approach, we can use eBPF to improve the performance of the DPI-TC by only sending the required packets to the DPI instance (dynamic packet sniffing).

A. Architecture

We have developed our Dynamic Sniffing DPI based on nDPI [14], which is a popular open source DPI for traffic classification. nDPI is claimed to be fast and accurate. The architecture of Dynamic Sniffing nDPI (DS-nDPI) is shown in Fig. 3.

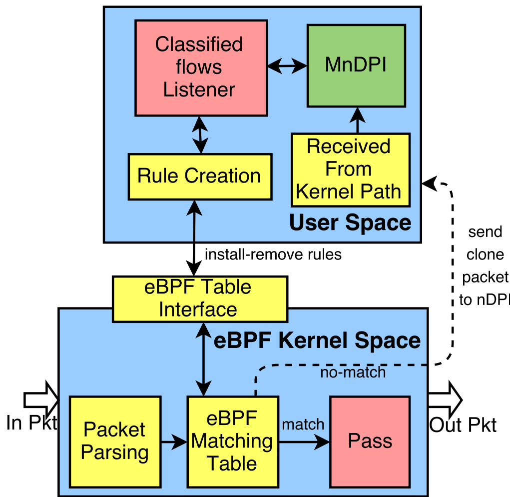

Fig. 3. DS-nDPI architecture with hybrid eBPF

At the kernel traffic control layer, the eBPF program dynamically sniffs the packets that have not been classified and sends these packets to the DPI engine. Firstly, the packet is parsed by *Packet Parsing* up to TCP/UDP layer. Then a combination of *Source IP, Source Port, Destination IP, Destination Port* is created as a *key* to be matched at the BPF table. If there is a matched entry, the packet belongs to a classified flow and no further action is required. If not, eBPF clones the packet and sends it to the listener interface for classification.

For fast prototyping, we keep the DPI instance as a separated C program which is a modified version of nDPI - MnDPI. MnDPI classifies traffic on the listener interface. After the classification succeeds, the classified flow identification (combination of *Source IP, Source port, Destination IP, Destination Port*) is sent to the *Classified flows Listener* over AF_UNIX Inter Process Communication (IPC) socket. Then a rule to indicate the classified flow is created and installed into the eBPF kernel program. When the MnDPI does the idle flows cleaning, removal flow messages are sent to the *Classified flows Listener* to remove the idle flows from the eBPF table.

VI. Evaluation

A. Dynamic Load Balancer

To evaluate the Dynamic LB performance, we deployed a web server-client topology and measured the average completion time per client request. We created a topology which contains three web servers and one LB, as shown in Fig. 4. The LB and web servers, each is deployed inside a VirtualBox VM running Ubuntu with 512 MB RAM and one core CPU. The host machine runs Ubuntu 14.04 with CPU Intel Core i5 3570 and 8 GB RAM.

We deployed a simple python web server in each server VM, and used Apache Bench [15] to send requests to the servers. The load balancer VM runs HAproxy, IPVS with round robin algorithm, and eBPF hybrid LB with dynamic weighted round robin. We also developed a full in-kernel eBPF load balancer with round robin algorithm for comparison. The result is given in Fig. 4.

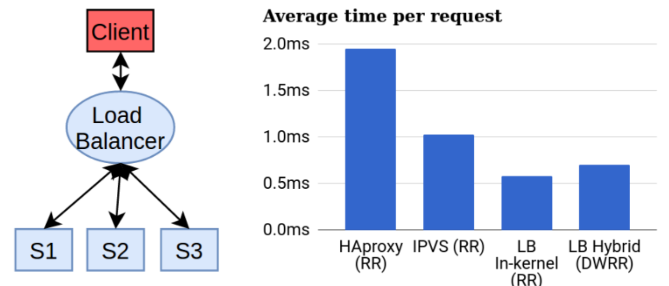

Fig. 4. Hybrid dynamic LB evaluation topology (left) and result (right)

The evaluation result shows that the average time per request of the hybrid LB, although is 1.2 times higher than the full in-kernel eBPF LB, still outperforms the others: 1.5 times shorter than IPVS and 2.8 times shorter than HAproxy. In the case we need speed with a simple scheduling algorithm, full in-kernel eBPF may be a good option, but if we need more complex scheduling algorithm, then we can use hybrid LB. Also, hybrid LB is faster than kernel IPVS because IPVS is a generic framework with many processing steps, while hybrid LB is more lightweight by focusing on its specific function.

B. nDPI with Dynamic Sniffing

To evaluate the DS-nDPI performance, we deployed a virtual link and measured the throughput affection of DS-

nDPI on traffic in this link. We created a virtual ethernet link with Linux *iptool* [16] and used *tcpreplay* [17] to replay real Internet traffic in this link, as shown in Fig. 5. The traffic includes over 1200 unique flows and spreads over 28 application protocols. DS-nDPI classified all flows on this link, and the performance was compared with the standard nDPI [18]. We deployed the test bench inside a VirtualBox VM running Ubuntu with one core CPU and 1 GB RAM.

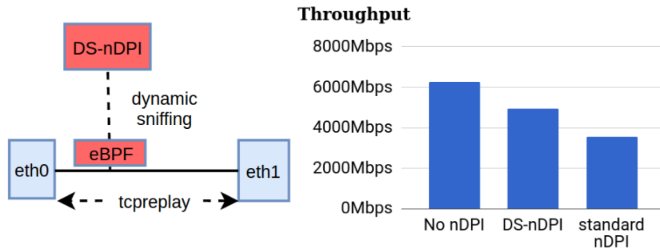

Fig. 5. DS-nDPI evaluation topology (left) and result (right)

The evaluation results in Fig. 5 show that dynamic sniffing causes 21% maximum throughput reduction of the tcpreplay, compared to 43% maximum throughput reduction of standard nDPI. The maximum throughput on the link with DS-nDPI is 1.38 times higher than the standard nDPI. The classification results of the standard nDPI and DS-nDPI are same because we did not modify the classification engine.

To sum up, in both cases, the hybrid eBPF architecture greatly improves the performance of VNFs compared to user space VNFs, and even the kernel space VNFs. We use BCC with Python user space program for quickly building the two example VNFs. However, eBPF does not limit the user space programming language. Thus, programmers can use other languages (such as C) for the user space program to get even better performance.

VII. Conclusion

In this paper, we proposed a hybrid method for building VNFs that leverages the speed advantages of eBPF in-kernel program and flexibility of the user space program, enabling building complex VNFs with high performance. We presented a general architecture and discussed various aspects of this architecture, including the match/action approach, packet processing in kernel and user space, how to split the work between kernel and user space program, and so on. To prove the feasibility of our proposed architecture, we have developed two VNFs with the hybrid method: load balancer with dynamic weighted round robin algorithm, and nDPI with dynamic packet sniffing. The evaluation results show that the hybrid method improves performance significantly compared to the widely used solutions. Since this paper focuses on standard-alone VNFs, our next step is to integrate our work with a VNF Manager (VNFM) such as OpenBox [19].

Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00195, Development of Core Technologies for Programmable Switch in Multi-Service Networks).

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," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, First quarter 2016.

[2] S. Alexei, "BPF - in-kernel virtual machine," *Linux Collaboration Summit*, 2015.

[3] eXpress Data Path (XDP), `https://www.iovisor.org/technology/xdp`.

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

[5] IO Visor Project, `https://www.iovisor.org`.

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

[7] A. Zaafar, M. H. Alizai, and A. A. Syed, "InKeV: In-Kernel Distributed Network Virtualization for DCN," *ACM SIGCOMM Computer Communication Review*, July, 2016.

[8] S. Jouet and D. P. Pezaros, "BPFabric: Data Plane Programmability for Software Defined Networks," in *Symposium on Architectures for Networking and Communications Systems (ANCS) '17*, 2017, pp. 38–48.

[9] 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.

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

[11] IP Virtual Server Project (IPVS), `http://www.linuxvirtualserver.org/software/ipvs`.

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

[13] Z. A. Qazi, C. C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *ACM SIGCOMM '13 Conference*, 2013, pp. 27–38.

[14] 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.

[15] Apache Bench, `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[16] Linux IP tool, `http://lartc.org/howto/lartc.iproute2.html`.

[17] Tcpreplay, `http://tcpreplay.appneta.com`.

[18] nDPI, `https://github.com/ntop/nDPI`.

[19] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *ACM SIGCOMM '16 Conference*, 2016, pp. 511–524.