Real-time Monitoring of Packet Processing Time for Virtual Network Functions

Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong

Dept. of Computer Science and Engineering, POSTECH, Pohang, Korea
Email: {tunguyen, jhyoo78, jwkhong}@postech.ac.kr

***Abstract*—By enabling the deployment of softwarelized network functions on commodity servers, Network Function Virtualization (NFV) brings many benefits such as rapid development and deployment, simplicity and flexibility in network operations and management. Monitoring the performance characteristics of Virtual Network Functions (VNFs), such as packet processing time, is crucial to achieving maximum benefit from NFV. In this paper, we present Packer Processing Time Monitoring (PPTMon) - a solution for real-time and lightweight VNF packet processing time monitoring. PPTMon embeds timestamp information directly into the packets. PPTMon is implemented using extended Berkeley Packet Filter (eBPF) - a new Linux framework that allows high-speed packet processing. Our experiments showed that PPTMon can monitor VNFs with high accuracy and low performance overhead.**

***Keywords*—Real-time Network Monitoring, Network Function Virtualization, Virtual Network Function, VNF Monitoring, eBPF, Packet Processing Time**

I. Introduction

For many years, telcos have been using proprietary middleboxes from vendors. These middleboxes implement various network functions such as switches, routers, firewalls, network address translations (NATs), intrusion detection/prevention systems (IDSs/IPSs), traffic classifiers, web accelerators, and load balancers. Because they are vendor-specific and proprietary, these middleboxes are very costly to deploy, operate, maintain and upgrade. The process of deployment and upgrading is also slow and complex. In contrast, the rapid change in the network services requirements nowadays leads to a short life-cycle of middleboxes, thus the middleboxes need to be upgraded more frequently. Eventually, telcos need to pay high operating expenses (OPEX) and capital expenses (CAPEX).

Network Function Virtualization (NFV) [1] is an effort to tackle these problems. NFV decouples the hardware and software parts of the middleboxes, turns the network functions into plain software that can run on any industry-standard, commodity servers, thus called Virtual Network Functions (VNFs) [1]. NFV has a set of standards so that VNFs are vendor-neutral and can be easily chained together to provide useful services. Telcos have many benefits from applying NFV: lower deployment and maintenance cost, faster upgrade process, and more flexible in operation and management. NFV also has positive effects on the vendor side. New vendors have a chance to join the market, old vendors also need to join the NFV development if they do not want to fall behind and lose their customers.

Monitoring is crucial to the deployment, operation and management of any networking systems, including the NFV system. One of the most important key performance indicators of VNFs is the packet processing time, i.e., from the time when a packet goes in the VNF to the time when the packet goes out of the VNF. Monitoring packet processing time helps to ensure that VNFs are working correctly with the expected performance. The monitoring data can also be used as inputs for VNF modeling or machine learning-based NFV management systems. However, while NFV has been one of the main research topics in recent years, monitoring VNF packet processing time is still a challenge.

In this paper, we present PPTMon - a solution for real-time and light-weight packet processing time monitoring. PPTMon's algorithm embeds a custom timestamp header into the packet header. The header is inserted at the ingress point and removed at the egress point. Because adding a custom header to a packet can potentially affect how the VNF processes the packet, the PPTMon header format is carefully considered so that PPTMon is transparent to the legacy PPTMon-unaware VNFs. We implemented the algorithm in PPTMon using extended Berkeley Packer Filter (eBPF) [2] - a new Linux kernel framework that allows high-performance packet processing. We evaluated PPTMon in real NFV environment using OpenStack [3]. The evaluation results showed that PPTMon has an average of 3.6% performance reduction when doing stress test with various VNFs.

The remainder of the paper is organized as follows. In Section II, we present the background and related work. In Section III, we present the detailed algorithm, header format, and implementation of PPTMon. Section IV shows the evaluation results of PPTMon and discusses PPTMon's limitations and future improvements. Finally, Section V concludes this paper.

II. Related Work

In this section, we cover related work about monitoring packet processing time and VNF performance in general. We also cover eBPF and discuss its performance advantage.

Copyright 2020 KICS

A. Packet processing time monitoring

There are several methods and research efforts to monitor packet processing time in VNFs. A very naive method is to capture ingress and egress packets with timestamps using tools such as tcpdump[1], then subtract the timestamp values. However, capturing every packet inside a VNF incurs very high overhead, thus greatly reduces the throughput and increases the latency of the VNF. Hence, this method is impractical in production systems and only suitable for debugging and offline testing purposes.

NFVPerf [4] moves the packet capture functions out of the VNF by mirroring all VM-to-VM traffic to a central processing node. NFVPerf uses deep packet inspection to analyze the traffic metric such as delay and CPU usage. Using this approach, NFVPerf can minimize the negative effect on the VNF, at the cost of extra network bandwidth and an extra dedicated node for NFVPerf processing.

KOMon [5] eliminates the packet capture overhead by using a different approach. KOMon uses a kernel module to inspect traffic at the ingress point of a VNF, saves the payload hash and timestamp into a queue. At the VNF egress, KOMon tries to match the payload hash value of the packet with the ingress payload hash. If the values are the same, then KOMon substracts the timestamp to get the packet processing time. KOMon thus can provide real-time data with low overhead. However, KOMon has some limitations. KOMon only works with VNFs that process packets in FIFO model and does not change the packet payload, KOMon is implemented as a custom kernel module, thus it has potential security and stability issues. Also, the payload hash method can potentially cause incorrect measurements in the case of packets with the same payloads (e.g., re-transmission, packet dropped).

SymPerf [6] uses code analysis to predict the performance of a VNF during run-time. Therefore, SymPerf does not have any performance impact on the VNFs at all. However, SymPerf requires access to the VNF source code, which is not always available, e.g., black-box VNFs, or VNFs from third-party vendors. Also, the unpredictable events, such as anomalies that increase packet processing time, can not be captured during the run-time.

In [7], authors use ICMP echo request and reply packets to measure the packet processing time of a physical host. The method can monitor delay without any software injection on the host. However, because the method requires a special packet-capture card with timestamp function, it is only available for physical host, not VM. Also, the processing time of ping packets does not necessarily represent the processing time of other packets.

B. extended Berkeley Packet Filter

extended Berkeley Packet Filter (eBPF) [2] is a Linux kernel framework that allows attaching user-supplied programs to a kernel event type. An eBPF program lives inside the Linux kernel as a light-weight virtual machine and is called when the event happens. An eBPF program is written in a restricted subset of C language and is compiled to eBPF instruction set which is mapped closely to the hardware instruction set. Also, eBPF compiler supports just-in-time compilation. Therefore, eBPF provides performance closes to the native C code. Although running inside the kernel, unlike a custom kernel module, eBPF programs are verified using kernel eBPF verifier during the compile-time to ensure the safety and security of the kernel. Also, the exposed interface to write eBPF program is stable, thus programmers do not need to worry about maintaining the compatibility with the new kernel versions like a custom kernel module.

eBPF can be used for high-performance packet processing [8]. In networking, an eBPF program can be attached to several layers in the kernel networking stack, such as traffic control or socket layer, and is called when a packet event happens, such as a packet received or sent. Because eBPF is well integrated with the kernel networking stack, the post-processed packet can be passed to the kernel stack for processing as normal. Compared to the user-space equivalent program, an eBPF program does not have the overhead of kernel-user space context switching. Also, the kernel creates a copy of the packet when sending it to the user-space program, while the eBPF program processes the packet in-stack and does not require packet copy.

III. Design and Implementation

In this section, we show the detailed algorithm and workflow of PPTMon. We present the PPTMon header format and discuss why we chose such a design. Then we present the implementation detail of PPTMon with eBPF.

A. PPTMon's algorithm

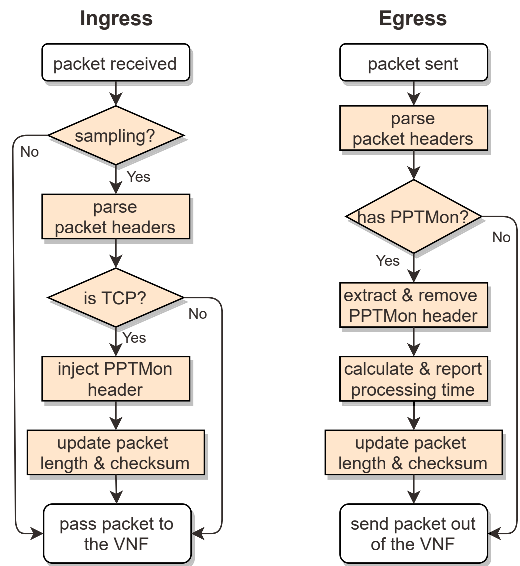

Fig. 1: PPTMon algorithm

[1]tcpdump, https://www.tcpdump.org

The idea of PPTMon is to attach a timestamp field into the packet header when the packet arrives at the VNF. Then when the packet is sent out, PPTMon calculates the packet processing time and removes the timestamp field. Fig. 1 shows the algorithm of PPTMon. At the early stage when the packet arrivals at a VNF, PPTMon inspects the packet and checks if the condition to inject the PPTMon header is passed. PPTMon then adds the PPTMon header with the current timestamp to the packet, updates the equivalent packet length and checksum, then passes the packet to the kernel networking stack. After the packet is processed by the VNF function, the packet is inspected by PPTMon before sent out. If the packet has a PPTMon header, the processing time is calculated as the difference between the current time and the stored timestamp. The accuracy of the PPTMon is thus guaranteed by the algorithm. The PPTMon header is then removed, packet length and checksum is restored, then the packet is sent out of the VNF.

Because running PPTMon for every packet can cause a high overhead and unnecessary huge amount of data, especially for high-throughput VNFs with hundreds of thousands of packets per second, we use sampling for PPTMon and leave the user to choose the sampling rate. When a new packet arrives, PPTMon simply checks the time passed from the last sampling, and if it exceeds the sampling period, a new PPTMon header is inserted into the packet.

B. PPTMon's header format

PPTMon header needs to be carefully designed because it is exposed to VNF and can potentially change the way VNF processes the packet. In PPTMon, PPTMon header is added to the packet in a TCP option field. The format of the PPTMon header as a TCP option is shown in Fig. 2, following the standard TCP option format [9]. PPT_H_KIND is set to 254, which is defined as an experimental option and should be ignored by the PPTMon-unaware VNFs, thus making PPTMon transparent to these VNFs. PPT_H_SIZE is the total length of the PPTMon header in bytes and is always set to 12 bytes. PPT_H_TSTAMP is a 64 bits value that stores the intermediate host time in nanoseconds.

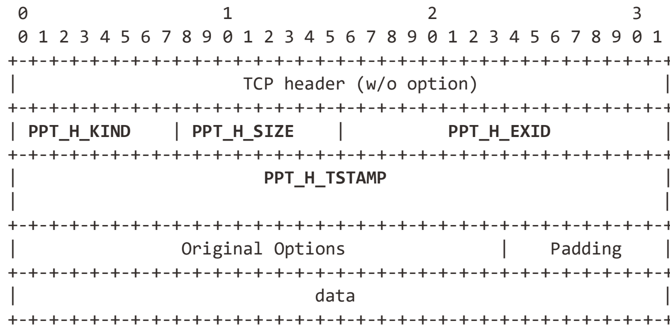

Fig. 2: PPTMon header as TCP option

Although applying a new TCP option kind requires proper registration [10], we design PPTMon so that it can be used in production without registering a new TCP option kind. Firstly, PPTMon header only exists when the packet is inside a VNF, thus PPTMon does not expose to the rest of the network and the Internet. Secondly, we follow the rule to share the usage of TCP experimental option [11] in a controller environment by setting a dedicated PPT_H_EXID for PPTMon.

While PPTMon header is currently only applied to TCP packets, it is also possible to insert PPTMon header as a UDP option [12]. In the UDP packet, both IP length field and UDP length field contains the header and data length. The redundant information can be used to indicate extra UDP option at the end of the UDP packet payload and it is permissible [12]. Moreover, while it is possible to put PPTMon as an IP option, several VNFs, such as iptables NAT, modify the IP header and may drop the IP option, thus losing the PPTMon header during the process.

C. Implementation

PPTMon is implemented in eBPF using BCC[2] - a framework for compiling and running eBPF programs. The eBPF code is attached to the Traffic Control (TC) *clsact* [13] at both ingress and egress, as shown in Fig. 3. Note that the VNF can also live in kernel space, such as iptables firewall[3]. The userspace module of PPTMon handles the processing of attaching and detaching eBPF programs to the kernel. It also handles the packet processing time reported by the PPTMon eBPF egress.

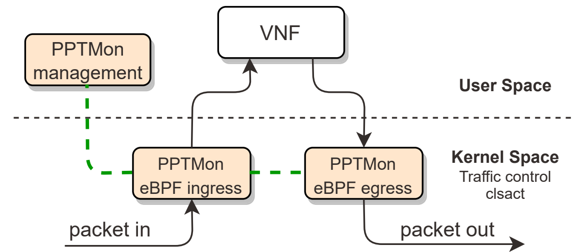

Fig. 3: PPTMon with traffic control clsact

In a precise definition, PPTMon measures the time from the TC *clsact* ingress, when Linux packet buffer *skb* is just created in the kernel; to the TC *clsact* egress, when the packet is already sent back to the kernel and is about to be sent out of the VNF. This measured time is the sum of the plain VNF processing time and the time Linux kernel handles the packet. The time Linux kernel handles the packet is negligible compared to the VNF processing time in normal cases, but it can be significant at high-speed VNF (e.g., iptables firewall), or when there are anomalies. Hence, PPTMon measured packet processing time actually provides a better metric to evaluate the state of the VNF than the plain VNF processing time.

In this work, PPTMon is implemented using eBPF. However, the same approach and algorithm can be applied for other

[2]BPF Compiler Collection, https://github.com/iovisor/bcc
[3]iptables, https://www.netfilter.org/projects/iptables/index.html

network stacks, such as DPDK[4]. Besides, in the case of using old Linux kernels that do not support eBPF, PPTMon can be implemented as a kernel module like KOMon [5]. We plan to publish PPTMon's source code when the UDP support is added in the near future.

IV. Evaluation

In this section, we present evaluation of three aspects of PPTMon: the average reported processing time, the effect of sampling rate to the performance, and the real-world performance when using PPTMon with the popular VNFs, such as firewall, NAT, IDS, DPI, and load balancer. We also discuss the limitations of PPTMon and future work for improvements.

A. Testbed setup

To evaluate PPTMon and observe how PPTMon performs in real use cases, we deployed our test scenarios using OpenStack [3], an open standard platform for cloud computing and NFV. The testbed deployment is shown in Fig. 4. We used a separate controller node to host OpenStack controller services so that their CPU usage does not affect the evaluation result. All VMs and VNFs were deployed on one OpenStack compute node. Although there are usually multiple compute nodes in a real operation environment, using only one compute node in this testbed does not affect the purpose of our experiments. We used Open vSwitch (OvS)[5] for inter-VM and VNF networking. All VMs and VNFs ran Ubuntu server 19.10 with kernel v5.3. Each VNF was allocated 2 vCPU and 2 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, with hyper-threading is enabled.

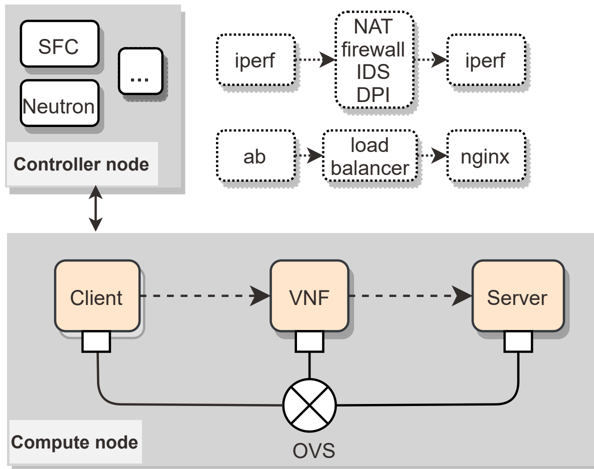

Fig. 4: PPTMon testbed with OpenStack

In all the tests, three VMs are used: two VMs ran as client and server, one VM was used for VNF deployment. In the case of transparent VNFs (firewall, IDS, DPI), OpenStack networking-sfc[6] was used to create the network function chains to forward packets via the VNF.

[4]DPDK, https://www.dpdk.org

[5]OvS, https://www.openvswitch.org

[6]networking-sfc, https://opendev.org/openstack/networking-sfc

B. Average processing time

In this scenario, we compared the values measured by PPTMon and tcpdump. Although PPTMon accuracy is ensured by the algorithm, the value only represents the delay time of the sampled packets, not all packets. Thus, we used tcpdump to capture all packets with timestamps to see whether PPTMon could represent the average packet processing time of the VNF.

In the test, both client and server ran iperf3[7] and exchanged TCP traffic. The VNF ran both PPTMon and tcpdump. Because of the tcpdump overhead, we ran iperf3 at low throughput (500 Kbps) so that the overhead of tcpdump would not affect the accuracy of the measurement. The VNF is iptables firewall with an increasing number of rules. We used dummy rules that did not match the iperf3 traffic, i.e., all packets need to be matched again all rules before going to the default rule, which forwards packets to the server. For each VNF setup, we ran the iperf3 traffic in 20 seconds and repeated 5 times.

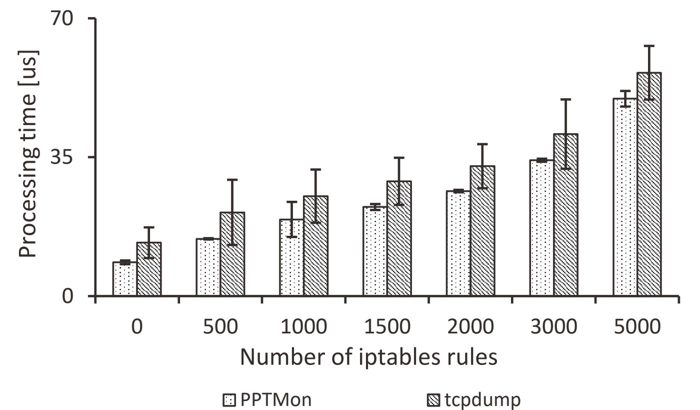

Fig. 5: Average packet processing time reported by PPTMon and tcpdump

The results are shown in Fig. 5. Intuitively, the processing time measured by both PPTMon and tcpdump increases when the number of flow rules increases. In all the cases, the average packet processing time reported by PPTMon was lower than the one from tcpdump. The average of the differences was 6.5 μs, with a standard deviation of 0.9 μs, which was quite stable. While both PPTMon and tcpdump work at the low level of the kernel stack, the differences are caused by the overhead of tcpdump. Because the value reported by tcpdump is the average of all packets, we conclude that PPTMon reported measurements can represent the average packet processing time.

This experiment also showed the baseline minimum latency of the kernel. In the case there is no firewall rule, the VNF just forwards the packet to the server right inside the kernel. The average delay measured by PPTMon was 8.5 μs with the standard deviation of 0.5 μs. The result was similar when the tcpdump is disabled.

[7]iperf3, https://iperf.fr

C. Effect of sampling rate

In this test, we measured how the sampling rate of PPTMon affects the performance. The VNF ran iptables firewall with no rules. The client and server ran iperf3. We recorded the average TCP throughput reported by iperf3 when the sampling rate is changed. Each run lasted 20 seconds and was repeated 5 times. The results are shown in Fig. 6.

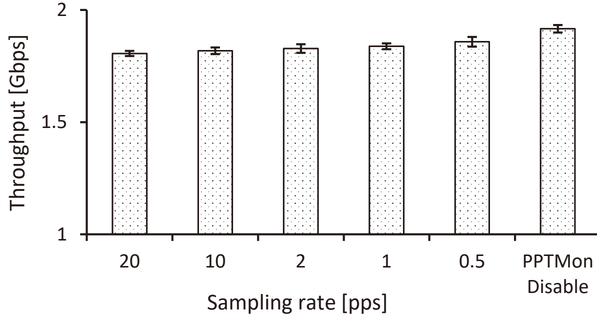

Fig. 6: Average throughput when changing the PPTMon sampling rate

The average throughput was slightly reduced when the sampling rate increased, with 5.7% reduction at 20 samples per second and 3% at 0.5 samples per second. We consider 5.7% is a small number, especially when the VNF is very lightweight: it just forwards the packets to the server directly in the kernel space. With more esource-intensive VNFs such as IPS, the throughput reduction caused by PPTMon is negligible, as will be shown in Section IV-D.

This experiment suggests two ways of using PPTMon. If users want to get the exact value of the sample packets every N sec (e.g., 1 sec), then the sampling rate can be set to $1/N$ packet per second (pps) (e.g., 1 pps). If the users want to get the average processing time of last N sec, then use can sample packets at 10x sampling rate $10/N$ pps (e.g., 10 pps for $N = 1$ sec), and then report the average of last 10 measurement values. The throughput difference between the sampling rate of 1 pps and 10 pps is only 1%. Thus, there is no real performance disadvantage.

D. Real-world performance

TABLE I: VNF configurations

VNF type	Software	Configuration
firewall	iptables netfilter	20 non-matching rules
DPI	nDPI	nDPI reader v3.2 stable with iperf3 protocol detection
IPS	Suricata	Suricata v4.1 in IPS mode with default rule set
NAT	iptables netfilter	full NAT mode
load balancer	IPVS	IPVS load balancer in destination NAT mode

In this test, we measured the maximum throughput of various VNFs when PPTMon is enabled and disabled. The VNF collection includes iptable firewall, Suricata[8] IPS, nDPI[9] DPI, iptables NAT, and IPVS[10] load balancer. The detailed configuration of each VNF is shown in Table I. When enabled, PPTMon's sampling period is set to 1 sec. Both client and server ran iperf3 and we recorded the average throughput. Each run lasted 20 seconds and was repeated 5 times. The result is shown in Fig. 7.

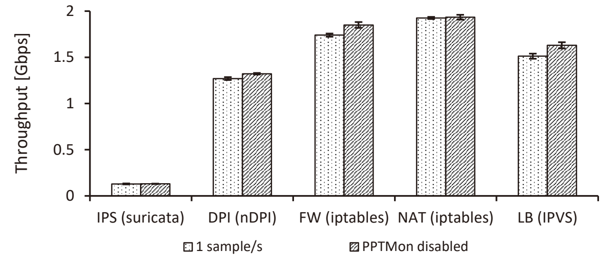

Fig. 7: Throughput when enable/disable PPTMon with various VNFs

In general, the throughput is slightly reduced when PPTMon is enabled. The percentages of throughput reduction were 0.6%, 3.9%, 5.9%, 0.4% and 7.2% for IPS, DPI, FW, NAT and LB, respectively. The effect of PPTMon overhead on throughput was reduced when the VNF is more resource-intensive and was increased when the VNF is more lightweight, except the case of NAT. In the case of iptables NAT, which is a lightweight VNF, the throughput reduction of PPTMon is only 0.4%. The average throughput reduction of all VNFs was 3.6%, and we consider it a small overhead.

E. Limitations and improvements

1) Limitations: Storing timestamp in the TCP option field implies one limitation of PPTMon: it does not work for VNFs which drop the TCP option header. However, many of the VNFs do not modify the packets (e.g., firewall, DPI, IDS and IPS), hence PPTMon works well with these VNFs. For the VNFs which modify the packets (e.g., NAT), PPTMon will work as long as the TCP option field is not removed.

In the case of load balancers, some load balancers, such as HAProxy[11], modify the TCP layer and drop the PPTMon header; while other load balancers, such as IPVS, keep the TCP header. Thus, PPTMon does not work with HAProxy, but it works with IPVS. Finally, if developers create new VNFs, then they just need to keep the PPTMon header to make PPTMon and the VNFs work together.

2) Improvements: There are three directions to improve the functionality of PPTMon. The first direction is to add UDP monitoring function to PPTMon, as discussed in Section III-B. The second direction is to deploy PPTMon in the physical host where VNF VMs are located instead of running directly

[8]suricata, https://suricata-ids.org
[9]nDPI, https://github.com/ntop/nDPI
[10]IPVS, http://www.linuxvirtualserver.org/software/ipvs.html
[11]HAProxy, http://www.haproxy.org

inside the VMs. PPTMon can attach its eBPF ingress and egress program to the virtual NIC of the VNF. There are two advantages: PPTMon can run more efficiently in a physical host than inside a VM, and the performance of the VNF is not affected by PPTMon.

The third direction is to monitor the latency of the whole service function chain (SFC) by adding the timestamp information to the PPTMon header when a packet enters any VNF and then extracts all of the PPTMon timestamp data at the end of the service chain. The idea is inspired by how In-band Network Telemetry [14, 15] has been successfully done the end-to-end network monitoring in programmable switches. By doing this way, PPTMon can report end-to-end packet processing time measurement instead of per-hop measurement.

V. Conclusion

In an NFV system, Monitoring VNFs, particularly the VNF packet processing time, is crucial for NFV operation and management. In this paper, we proposed PPTMon - a solution for real-time and light-weight packet processing time monitoring. PPTMon works by attaching timestamp data to the packet header at the VNF ingress, then calculating the processing time at the VNF egress. We presented the detailed design and implementation of PPTMon. The evaluation results showed that PPTMon is accurate and light-weight. PPTMon has an overhead of 3.6% VNF throughput reduction on average when doing stress test.

We have several directions to improve PPTMon, as discussed in Section IV-E. Firstly, we plan to add UDP support to PPTMon. Secondly, we will modify PPTMon to support monitoring from the side of the physical hosts. Finally, we will develop the SFC monitoring function so that PPTMon can provide end-to-end measurements instead of per-hop measurements.

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] "Openstack." [Online]. Available: https://www.openstack.org

[4] P. Naik, D. K. Shaw, and M. Vutukuru, "NFVPerf: Online performance monitoring and bottleneck detection for NFV," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 154–160.

[5] S. Geissler, S. Lange, F. Wamser, T. Zinner, and T. Hoßfeld, "KOMon — Kernel-based Online Monitoring of VNF Packet Processing Times," in *2019 International Conference on Networked Systems (NetSys)*, 2019, pp. 1–8.

[6] F. Rath, J. Krude, J. Rüth, D. Schemmel, O. Hohlfeld, J. A. Bitsch, and K. Wehrle, "SymPerf: Predicting Network Function Performance," in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos '17, 2017, p. 34–36.

[7] K. M. Salehin and R. Rojas-Cessa, "Measurement of packet processing time of an Internet host using asynchronous packet capture at the data-link layer," in *2013 IEEE International Conference on Communications (ICC)*, 2013, pp. 2550–2554.

[8] 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.

[9] J. Postel, "Transmission Control Protocol," RFC Editor, STD, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[10] S. Bradner and V. Paxson, "IANA Allocation Guidelines For Values In the Internet Protocol and Related Headers," RFC Editor, BCP, March 2000. [Online]. Available: https://tools.ietf.org/html/rfc2780

[11] J. Touch, "Shared Use of Experimental TCP Options," RFC Editor, RFC, August 2013. [Online]. Available: https://tools.ietf.org/html/rfc6994

[12] J. Touch, "Transport Options for UDP," RFC Editor, Tech. Rep., 2019. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tsvwg-udp-options

[13] D. Borkmann, "Advanced programmability and recent updates with tc's cls bpf," *Proc. NetDev*, vol. 1, 2016.

[14] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[15] J. Hyun, N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Real-time and fine-grained network monitoring using in-band network telemetry," *International Journal of Network Management*, vol. 29, no. 6, 2019.