Network traffic measurement tools

Elia Azar* and Leonardo Linguaglossa*
*Télécom Paris
elia.azar@telecom-paris.fr, linguaglossa@telecom-paris.fr

***Abstract*—Telemetry is a fundamental process in network environments. With the growing popularity of softwarized functions that replace hardware middleboxes, it is essential to monitor network conditions in software. While the flexibility of software components provides the means for quickly develop and deploy monitoring solutions, this usually comes at the expenses of limited performance. In particular, custom ASIC-based measurement tools can guarantee that 100% of the incoming traffic will be analyzed. This is usually not the case for software monitoring tools, which may in addition alter the state of the system. Our goal is to evaluate non-invasive approaches in a low-resource setting to determine the best measurement tools for the analysis of network traffic. We take into account popular tools that rely on kernel-bypass, but also kernel-native approaches, which have the advantage of a complete stack with no implementation effort.**

***Index Terms*—DPDK, P4, EBPF, XDP, Moongen**

I. Introduction & Background

In this paper, the main focus is to test the efficiency of non-invasive methods in a low-resource environment in order to assess the best network traffic measurement tools.

For years, developers have depended on proprietary hardware's better performance for product deployment while using software packet processing for rapid prototyping and functional testing. This situation has improved over the last years, thanks to the acceleration of Software-Defined Networking (SDN) and Network Function Virtualization (NFV). It is now commonly understood that replacing costly, proprietary, and inflexible hardware middleboxes with software equivalent may result in considerable cost reductions.

The development of high-speed I/O frameworks that use acceleration techniques such as kernel bypass and batch processing to reach performance similar to proprietary hardware appliances has been a key driver of advancement.

For that reason, this paper tackles eight different software approaches in order to determine the best solution for such an environment. The goal of this study is to offer a methodology for comparing the performance of measurement tools in terms of Packet loss.

We deploy a basic setup typically used for performance comparisons. In this scenario, a network assessement tool is deployed behind a port of a 10 Gbps network interface cards (NICs).

We begin by examining the design space of eight software measuring tools namely Tcpdump [11], Moongen [4], DPDK pdump tool [1], xdp-dump from XDP-tools [16], BCC [14], P4-eBPF [8], P4-XDP [15] and DPDK custom code [6], to gain a fundamental knowledge of their designs.

Tcpdump [11] is a very popular and widely used command-line packets sniffer which is used to capture or filter TCP/IP packets received on a specific interface. It runs on the majority of Linux/Unix-based operating systems. This tool prints out a description of the contents of packets on a network interface that match the filter expression; the description is preceded by a time stamp.

MoonGen [4] "is a scriptable high-speed packet generator built on libmoon. The whole load generator is controlled by a Lua script: all packets that are sent are crafted by a user-provided script. Thanks to the incredibly fast LuaJIT VM and the packet processing library DPDK [6], it can saturate a 10 Gbit/s Ethernet link with 64 Byte packets while using only a single CPU core. MoonGen can achieve this rate even if each packet is modified by a Lua script. It does not rely on tricks like replaying the same buffer".

DPDK-pdump [1] "is a Data Plane Development Kit (DPDK) tool that runs as a DPDK secondary process and is capable of enabling packet capture on DPDK ports. The DPDK-pdump tool can only be used in conjunction with a primary application which has the packet capture framework initialized already. In DPDK, only the testpmd is modified to initialize packet capture framework, other applications remain untouched. So, if the DPDK-pdump tool has to be used with any application other than the testpmd, user needs to explicitly modify that application to call packet capture framework initialization code".

XDP-dump [16] "is a simple XDP packet capture tool that tries to behave similar to tcpdump, however, it has no packet filter or decode capabilities". To make up for this, XDP-dump can be used along with XDP-filter as a packet filtering utility powered by XDP. The latter can achieve very high drop rates: tens of millions of packets per second on a single CPU core.

BCC [14] "is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. It makes use of extended BPF (Berkeley Packet Filters), formally known as eBPF, a new feature that was first added to Linux 3.15. Much of what BCC uses requires Linux 4.1 and above. BCC makes BPF programs easier to write, with kernel instrumentation in C (and includes a C wrapper around LLVM), and front-ends in Python and lua. It is suited for many tasks, including performance analysis and network traffic control ".

P4-eBPF [8] "is a P4 compiler backend targeting eBPF. P4C (P4 compiler) is a reference compiler for the P4 programming language. It supports both P4-14 and P4-16. The back-end accepts only P4-16 code written for the ebpf_model.p4 filter model. It generates C code that can be afterwards compiled into eBPF using clang/llvm or bcc".

P4-XDP [15] "is a P4 compiler backend targeting XDP, the eXpress Data Path. P4 is a domain-specific language describing how packets are processed by the data plane of a programmable network elements, including network interface cards, appliances, and virtual switches. With P4, programmers focus on defining the protocol parsing, matching, and action executions, instead of the platform-specific language or implementation details. When the device observes an incoming packet, before hanging the packet to the Linux stack, the user-defined XDP program is triggered to execute against the packet payload, making the decision as early as possible".

DPDK custom app [6] "The main goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Users may use the code to understand some of the techniques employed, to build upon for prototyping or to add their own protocol stacks. The framework creates a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL), which may be specific to a mode of the Intel® architecture (32-bit or 64-bit), Linux* user space compilers or a specific platform".

Then we create three test scenarios to get results that are relevant to various features of a traffic monitoring instrument. Finally, we present experimental measurements of packet loss with unidirectional traffic.

The results seen in section V prove that methods bypassing the kernel are best fit for low-resource environments, thus being the best measurement tools for network traffic.

It's crucial to remember that these testing findings are very dependent on the specific hardware and software versions employed in our platform, and hence are merely suggestive.

Many of the scripts and guidelines for our tests have been posted on GitHub [3] to encourage reproducibility. We highly encourage researchers and engineers to use this to replicate the same series of experiments on their own servers and draw on this foundation to gain a better understanding.

The remainder of this paper is organised as follows. Section II introduces the related work while Section III discusses the meaurement tools design space. Section IV presents the associated experimental setup and evaluation. Section V discusses the results obtained, and finally, Section VI concludes this paper.

II. Related work

Various previous works making the connection between network traffic monitoring and performance analysis are available in the bibliography. A summary is presented in the following.

Overall, the focus of those works is on introducing traffic monitoring tools for network experiments. In contrast, in this work, the focus is on determining the best software network traffic measuring tools in terms of packet loss minimization.

As of today, a lot of tools and methods exist for traffic monitoring and measurement, but the cornerstone of this paper is the analysis and comparison of the set of tools to assess the best software tools in a network environment.

Paper [13] introduced CeMOC, a cost-efficient flow measurement system in distributed controller deployment for datacenter networks. In the implementation of SDN distributed controllers, their technique is a realistic option for a cost-effective measuring system.

CeMOC was designed as a standard SDN northbound RESTful API in a floodlight controller that can handle nearly all elements of monitoring. It can collect near real-time traffic with a considerable decrease in communication cost, message interaction, and controller overhead, according to the results of an extended experiment in in-band network deployment. However, CeMOC is used as a standard SDN northbound RESTful API, therefore deviating from our perspective.

In [7], the authors argue that combining pos and libmoon/-MoonGen creates an ideal platform for network experiments. Based on their findings, this platform has a low cost, a lot of versatility, is simple to use, and allows you to create repeatable tests. MoonGen, as well as the underlying framework libmoon, are two extremely significant tools for the research community. The latter provides a simple framework for creating modular high-performance apps and other utilities. MoonGen allows you to reliably and precisely manage the produced traffic as well as evaluate the latency and throughput of the received traffic, being critical criteria for executing and reproducing network experiments.

In [17] the authors designed FloWatcher-DPDK, a DPDK-based high-speed software traffic monitor provided to the community as an open source project. FloWatcher-DPDK, in a summary, provides configurable fine-grained statistics at the packet and flow levels. Experiments show that FloWatcher-DPDK can maintain high precision per-flow statistics at 10 Gbps line rate with 64B packets, using only 2 CPU cores while utilizing a little amount of resources with tolerable packet loss.

Article [18] mainly introduces and analyzes the structure and key technologies of DPDK. In this work, data processing issues under the conventional network protocol stack are investigated. A high-performance data processing system is suggested that focuses on these fundamental issues. DPDK may significantly minimize packet loss and increase data packet processing rate, according to this study. For traffic collection and distribution, it can help decrease resource waste and network overhead.

III. Measurement tools design space

Before discussing how the eight typical state-of-the-art solutions fit into a design space taxonomy, we first discuss the necessity of examining the various design objectives of alternative traffic measurement tools.

Table I
TAXONOMY OF STATE-OF-THE-ART HIGH-PERFORMANCE TRAFFIC MEASUREMENT TOOLS

Methods	Architecture		Programmability	Difficulty	Programming language
	in-kernel	kernel-bypass			
Tcpdump	x		None	One-liner	
MoonGen		x	Medium	Normal	Lua
XDP-dump	x		None	One-liner	
DPDK-pdump		x	None	One-liner	
P4-eBPF	x		Low	Easy	P4
P4-XDP	x		Low	Easy	P4
BCC	x		Medium	Normal	Python, C, Lua
DPDK		x	High	Hard	C

A. Design objectives

It's critical to grasp the major architectural differences between the measurement tools under consideration before conducting a comparison analysis. This might include analyzing the tool's processing model or establishing whether it was designed for a certain application, such as SDN or NFV. This takes time, but it appears to be a necessary requirement for avoiding incorrect estimate of the relevance of small performance factors.

Rather than going into depth on implementation, we'll look at each tool design in relation to a variety of technical factors that impact packet filtering and dumping performance in this section.

The goal is to obtain knowledge on how to create useful experimental settings. Table I summarizes this taxonomy and the traffic measurement tool categorization it corresponds to.

B. Architecture

" The Linux kernel has always been an ideal place to implement monitoring/observability, networking, and security. Unfortunately this was often impractical as it required changing kernel source code or loading kernel modules, and resulted in layers of abstractions stacked on top of each other. eBPF is a revolutionary technology that can run sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules.

By making the Linux kernel programmable, infrastructure software can leverage existing layers, making them more intelligent and feature-rich without continuing to add additional layers of complexity to the system or compromising execution efficiency and safety " [2].

Another solution for addressing the restrictions of the in-kernel network stack is programmable packet processors. They enable the execution of user-defined code in both the operating system and the hardware. Linux's programmable packet processor is known as XDP. It enables a user-defined eBPF application to handle a packet before it is sent to the kernel's network stack. The eBPF program can either process the packet in its entirety, conduct some preprocessing, and send it to the in-kernel stack, or transfer the packet to a userspace memory buffer once it has been processed.

" XDP provides bare metal packet processing at the lowest point in the software stack which makes it ideal for speed without compromising programmability. Furthermore, new functions can be implemented dynamically with the integrated fast path without kernel modification.

The XDP packet process includes an in kernel component that processes RX packet-pages directly out of driver via a functional interface without early allocation of skbuff's or software queues. Normally, one CPU is assigned to each RX queue but in this model, there is no locking RX queue, and CPU can be dedicated to busy poll or interrupt model. BPF programs performs processing such as packet parsing, table look ups, creating/managing stateful filters, encap/decap packets, etc " [12].

However, since networks are growing faster than CPUs, the network stack design is undergoing a revolution. The performance of a single-threaded CPU has reached a plateau. NICs, on the other hand, are becoming increasingly fast. Today, NICs are already capable of 2000 Mbps and are continuing to improve.

The hardware and OS network stack architecture are also challenged by very high packet rates. In summary, the typical in-kernel network stack architecture is unable to handle the high rate of packets.

In-kernel network stacks have well-known overheads. Operations are implemented as system calls in most in-kernel network stacks. That is, programs use system calls to transmit control to the kernel for both control plane and data plane activities. Because system calls have high overheads, they are an issue for network-intensive applications.

By transferring protocol processing to userspace, kernel-bypass networking removes the overheads of in-kernel network stacks. Packets travel from the NIC to userspace with minimum intervention from the OS.

The Data Plane Development Kit (DPDK) is one of the major kernel bypass solutions.

" DPDK is an open source software project managed by the Linux Foundation. It includes data plane libraries and polling-mode network interface controller drivers for offloading packet processing from the operating system kernel to user-space processes. Using the interrupt-driven processing offered in the kernel, this offloading enables improved compute efficiency and packet throughput " [6].

Similarly, MoonGen is built on libmoon [9], a Lua wrapper for DPDK.

C. Programmability, Difficulty & Programming Language

Performance needs and programmability concerns may force you to choose one programming language over another.

Some of the software frameworks for high-speed measurement tools lets you write, compile and run your scripts in C, Python, Lua or P4.

Tools that don't offer a programmability are very easy to use since running them can be done with a single command line. These tools are namely Tcpdump, XDP-dump from XDP-tools and DPDK-pdump. However, a major drawback is the lack of creating and adding new features.

Tools that offer programmability are not only performant, but also feature-rich and cross-platform compatible.

Higher-level programming languages such as Python and Lua are also used by some measurement tools. For example, " BCC makes BPF programs easier to write, with kernel instrumentation in C (and includes a C wrapper around LLVM), and front-ends in Python and lua. It is suited for many tasks, including performance analysis and network traffic control " [14].

MoonGen, on the other hand, " let its users write custom Lua scripts for their experiments. It is recommended to make use of hard-coded setup-specific constants in your scripts. The script is the configuration, it is beside the point to write a complicated configuration interface for a script " [**MoonGen-imc2015**].

An alternative would be programming in P4, " which is a domain-specific language describing how packets are processed by the data plane of a programmable network elements, including network interface cards, appliances, and virtual switches. With P4, programmers focus on defining the protocol parsing, matching, and action executions, instead of the platform-specific language or implementation details " [15]. P4 allows you to quickly and easily prototype network components.

P4C has an eBPF backend " that accepts only P4_16 code written for the ebpf_model.p4 filter model. It generates C code that can be afterwards compiled into eBPF using clang/llvm or BCC " [2].

Similarly, VMware presents P4C-XDP, a P4 compiler backend targeting XDP.

Finally, writing a script for DPDK is done in C, " it makes low-level code more accessible to programmers. All drivers in DPDK are written in C as large parts of them are derived from kernel implementations DPDK consists of more than drivers: it is a whole framework for building fast packet processing apps featuring code from utility functions to complex data structures — and everything is written in C. This is not an unreasonable choice: C offers all features required for low-level systems programming and allows fine-grained control over the hardware to achieve high performance. But with great power comes great responsibility: writing safe C code requires experience and skill. It is easy to make subtle mistakes that introduce bugs or even security vulnerabilities. Some of these bugs can be prevented by using a language that enforces certain safety properties of the program " [5].

IV. Experimental setup

In this section, we first describe the testbed over which we execute our experiments.

A. Measurement platform

Our testbed includes a commodity server equipped with two Intel Xeon L5640 @ 2.27GHz CPUs (with 2 threads per core and 6 cores per socket, and 32K/32K/256K/12288K L1d/L1i/L2/L3 caches), and two Intel dual-port 10-Gbps NICs spread over two NUMA nodes. The server runs Ubuntu 18.04 with Linux 4.15.0-140-generic kernel.

For each tested tool, the latest stable version at the time of writing has been used, namely: DPDK (version 20.11); BCC (v0.11.0); XDP-TOOLS (v1.0.0-beta3); P4C (commit 4747f35); P4C-XDP (commit c5e3370); and Moongen (commit 525d991).

We also reserve 2GB Hugepages per socket to minimize TLB misses.

Network measurement tools are always deployed on a single core on NUMA node 0 to ensure a fair comparison. By default, we use MoonGen as traffic generator/receiver (TX/RX) for all our scenarios. It's worth noting that using the same server for traffic generation and reception does not result in erroneous interference since the server's NUMA architecture efficiently isolates cores and memory.

Since packets are carried over physical NICs, the theoretical bottleneck for these cases is their maximum capacity (10Gbps).

B. Packet loss

It is critical for network operators to understand the impact of packet loss.

Network packets are data units that travel through a network. These units are simply parts of the larger message being sent, which have been organized into numerous layers. When these packets are joined, however, they become meaningful. To get to their destination, packets must pass through a number of hubs.

Packet loss is the failure of data packets to reach their destination after being sent over a network. Packet loss is often caused by network congestion, hardware problems, program bugs, and a variety of other causes.

In this paper, the area of interest is packet loss due to the deployment of traffic monitoring tools on a network component or an end-device. When faced with a large number of high-speed and high-throughput network environments, standard packet capture methods and processing capabilities are unable to keep up, resulting in substantial packet loss.

C. Test methodology

In our workflow, network traffic is generated with MoonGen and sent out of from one physical port from the first NIC. This traffic is sent to its desired destination which is the the physical port of the second NIC connected to the first one. Packets that are generated by MoonGen use UDP as a transport protocol, IPv4 for the network layer which are then encapsulated in Ethernet headers.

Both NICs are plugged on the same server, where the traffic monitoring tools and packet generator are installed and implemented. The packet generator is configured to run on 9 cores in order to achieve high performance by reaching up to 14.88 Mpps. For traffic monitoring tools, we were able to configure the devices that use kernel-bypass methods to use 5 physical cores when deployed, whereas no additional configuration was done for in-kernel methods.

Network measurement tools capturing packets on a given interface can either print them on the terminal, save them in a pcap file, or do nothing with them (which will be denoted by /dev/null later on). In addition, another functionality would be to filter the packets before passing or dropping them.

Evaluating the bare packet loss rate between two physical interfaces thus provides a useful baseline reference.

D. *bpftool*

Bpftool allows for inspection and simple modification of BPF objects on the system, specially eBPF programs and maps [10].

This tool was used along with some of the tools making use of BPF maps, namely P4-eBPF and P4-XDP.

The above-mentioned tools use not only BPF maps to store the value of the received packet counter as well as the dropped packet counter, but also to store and modify the values inside the hash tables used to perform match action on ingress packets.

Before running the tests, we make sure to reset all the counters to zero by specifying the map id as well as the keys for this map.

E. *Benchmark*

1) Unicast Method comparison: In this scenario, we test all the functionalities of the aforementioned methods to provide a proper comparison benchmark. It is important to note that some of the methods don't have all the functionalities available, like filtering or dumping the packets to a pcap file.

We test all the methods with 4 different configurations (if possible) in order to assess the performance of all the functionalities.

The following is the list of configurations:

1) Print the packets
2) Save the packets to a pcap file
3) Do nothing with the packets
4) Filter the packets

It should be noted that for the fourth configuration, once the packets are filtered, we decide to pass the packets without printing them nor saving them. This way, we can see how filtering can alter the tool's performance by comparing it with the third configuration. It is worth mentioning that all the incoming packets match the filter, meaning that dropped packets won't be caused due to filter mismatch, but rather to the deployed tool's performance.

The following expression was used as filter: udp dst port 320.

We compute the packet loss by taking into account the number of packets received and treated on the second physical port, and the number of packets sent out the first physical port.

2) Processing rate: In this case, we describe how packet loss varies depending on the throughput, which ranges from 1 to 14.88 Mpps. This process was automated by using bash script to run this benchmark 50 times per throughput by modifying MoonGen's packet generator configuration.

The amount of network packets that the networking equipment can process is referred to as the processing rate, which is quite similar to the forwarding rate of a switch. Packets per second is the unit of measurement for processing rate (pps).

This test assists us in determining the processing rate for each tool in our collection.

3) Packet Loss as a function of Filter Matching: In this scenario, we describe how packet loss varies with regard to the percentage of packets matching the filter by maintaining a stable throughput of 14.88 Mpps.

We were able to vary the percentage of matching packets by creating a function that uses the random library in Lua, and by passing the percentage threshold of packets matching the filter as an argument when running the packet generator.

Only tools that support filtering were tested, namely Tcpdump, MoonGen, XDP-dump, P4-eBPF, P4-XDP and BCC.

V. Results

In this section, we present the results obtained by applying our test methodology to the eight software network traffic measurement tools identified in Section I and analyzed in Section III.

We configure MoonGen to transmit synthetic traffic at 10 Gbps from the first NIC to the second. Packets are transmitted at the highest possible pace. By collecting outbound traffic from the first NIC and inbound traffic to the second NIC, we assess the percentage of packet loss.

The packet loss experienced by the software measurement tools in the three test scenarios under synthetic unidirectional traffic with a 64-byte packet size is presented and discussed in the following.

A. *Unicast Method Comparison*

Fig. 1 shows the packet loss variation for the Unicast Method Comparison benchmark. Considering unidirectional traffic, we can clearly see that the overall performance of the network traffic measurement tools are the best when the packets are passed directly to the upper layers, which is denoted by /dev/null in Fig. 1. A small degradation in performance occurs when packets are filtered before being passed. In terms of saving packets in a pcap file, we encounter a big discrepancy in performance, and printing them to stdout is the poorest option.

MoonGen outperforms all the other methods with an overwhelming performance of 40% of packet loss when the packets are passed without being printed, filtered nor saved.

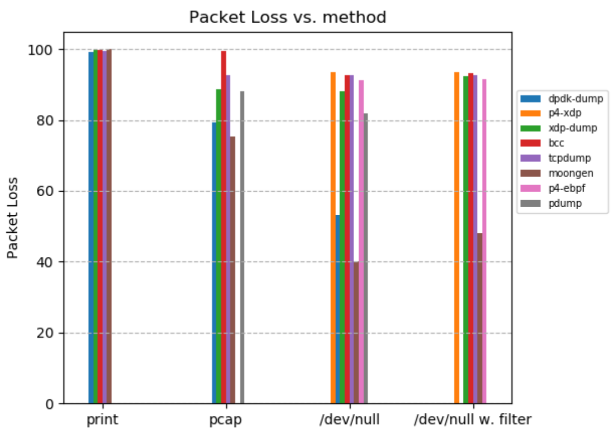

Figure 1. Packet loss vs method

MoonGen is followed closely by our DPDK custom application with a 53% packet loss under the same test scenario.

Based on these findings, we can determine that kernel-bypass methods outperforms in-kernel solutions when it comes to unidirectional traffic with a 64-byte packet size.

B. Processing rate

Fig. 2 illustrates the processing rate of available methods, except DPDK-pdump, for unidirectional traffic with 64-byte packets.

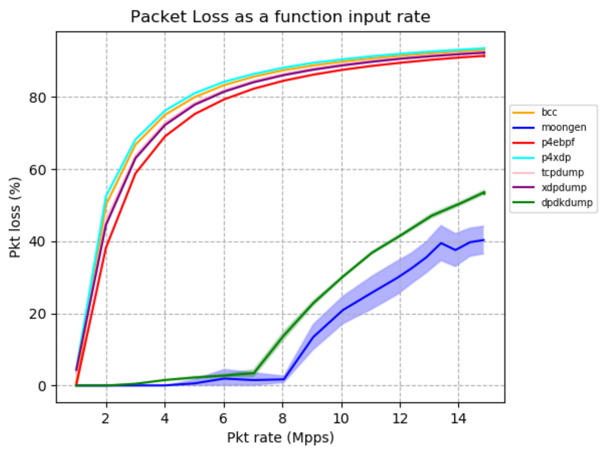

Figure 2. Processing rate

MoonGen achieves the best performance with a processing rate of 8 Mpps, closely followed by our custom DPDK application with a processing rate of 7 Mpps.

As for the other methods, they all have a processing rate lower than 1 Mpps, once again proving that kernel-bypass solutions are best suited for handling high input rate packets in such environments.

C. Packet Loss as a function of Filter Matching

Fig. 3 and 4 illustrate the packet filtering effect on packet loss. We split the data into two graphs since MoonGen's superior performance would hinder the plots of the other tools.

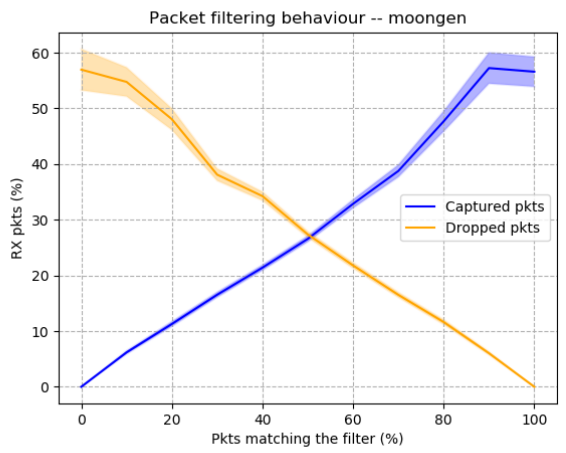

Figure 3. MoonGen's Packet filtering behaviour

It is seen in Fig. 3 that if we sum the number of packets processed by MoonGen and intentionally dropped or captured at any given time, the result will fluctuate around the value 55%, indicating that MoonGen processed 55% of the traffic while losing 45% of the packets, as shown in Fig. 1.

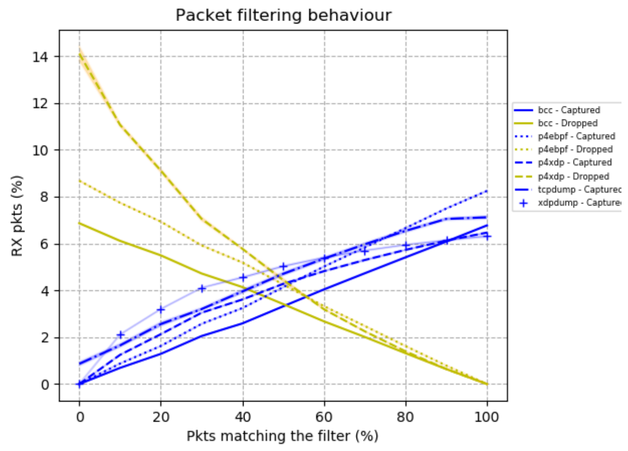

Figure 4. Packet filtering behaviour

Finally, Fig. 4 illustrate the benchmark result for the remaining methods, namely BCC, P4-eBPF, P4-XDP, Tcpdump and XDP-dump. We can see that the first three methods are represented by two plots, just like the previous figure with MoonGen, whereas Tcpdump and XDP-dump are only represented by one plot because it is not possible to retrieve the number of intentionally dropped packets due to the filter's presence.

It seems that the in-kernel solutions have comparable performance, with P4-XDP slightly outperforming the rest.

P4-XDP has an intriguing phenomena in that it appears to process more packets when none of them match the filter than when all of the packets do.

VI. Conclusion & Future work

The spread of NFV and the introduction of high-speed packet I/O frameworks has prompted much study into the construction of software network measurement tools that operate on COTS systems. To capture and handle communication between NICs in NFV applications, several alternative architectures have been suggested and deployed.

In this paper, we provide a performance assessment technique and provide example findings for eight state-of-the-art suggestions in order to better understanding of the performance of these designs. The study is built around three test scenarios: Unicast Method Comparison, Processing Rate, and Packet Loss as a Function of Filter Matching, all of which are intended to investigate the performance of common network traffic processing tools. This paper covers the experimental set-up in full, including the specifications of tested software and hardware versions, as well as the packet creation and monitoring tools employed, in the interest of repeatability. On GitHub [3], you can find all of the scripts and instructions we used to perform our tests.

The measurement results reveal that kernel-bypass solution prevail in all scenarios. This is to be anticipated, given that the latter avoids the overhead of interacting with the kernel of the operating system. This provides userspace with the kernel's raw performance. The given findings and accompanying discussion allow for more informed decision-making and should drive the development of viable improvements to alleviate limitations.

Our planned future work will include consideration of implementing P4-DPDK and P4-uBPF solutions. Another main axis of improvement would be creating a better version of our custom DPDK application by implementing a pipeline with one producer and two consumers. The producer would listen on the interface and save all the incoming traffic in a ring. Next, the first consumer will perform match/action operations to keep packets matching the filter. Finally, the second consumer will take the remaining packets and print them or save them in a pcap file before passing them to the application.

It's worth emphasizing that the current wide-ranging comparison took a lot of work, both to grasp the details of the network measurement tools in question and to set up and run the tests. As a result, we believe that other researchers can benefit from our expertise in further studying the performance characteristics of present and upcoming software tools, as well as in improving the assessment technique.

References

[1] *dpdk-pdump Application*. 2020. URL: https://doc.dpdk.org/guides-20.08/tools/pdump.html.

[2] "ebpf". In: URL: https://ebpf.io.

[3] Elia Azar and Leonardo Linguaglossa. *Network traffic measurement tools*. https://github.com/elia-azar/PRIM. 2021.

[4] Paul Emmerich et al. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, Oct. 2015.

[5] Paul Emmerich et al. "The Case for Writing Network Drivers in High-Level Programming Languages". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–13. DOI: 10.1109/ANCS.2019.8901892.

[6] Linux Foundation. *Data Plane Development Kit (DPDK)*. 2015. URL: http://www.dpdk.org.

[7] Sebastian Gallenmüller et al. "High-performance packet processing and measurements". In: *2018 10th International Conference on Communication Systems Networks (COMSNETS)*. 2018, pp. 1–8. DOI: 10.1109/COMSNETS.2018.8328173.

[8] P4 language. "p4c". In: URL: https://github.com/p4lang/p4c.

[9] "libmoon". In: URL: https://github.com/libmoon/libmoon.

[10] "Man page of BPFTOOL". In: URL: https://man.archlinux.org/man/bpftool.8.en.

[11] *Man page of TCPDUMP*. URL: https://www.tcpdump.org/manpages/tcpdump.1.html.

[12] IO Visor Project. "XDP". In: URL: https://www.iovisor.org/technology/xdp.

[13] Hamid Tahaei et al. "Cost Effective Network Flow Measurement for Software Defined Networks: A Distributed Controller Scenario". In: *IEEE Access* 6 (2018), pp. 5182–5198. DOI: 10.1109/ACCESS.2017.2789281.

[14] IO Visor. "BPF Compiler Collection (BCC)". In: URL: https://github.com/iovisor/bcc.

[15] VMware. "p4c-xdp". In: URL: https://github.com/vmware/p4c-xdp.

[16] xdp-project. "xdp-tools - Utilities and example programs for use with XDP". In: URL: https://github.com/xdp-project/xdp-tools.

[17] Tianzhu Zhang et al. "FloWatcher-DPDK: Lightweight Line-Rate Flow-Level Monitoring in Software". In: *IEEE Transactions on Network and Service Management* 16.3 (2019), pp. 1143–1156. DOI: 10.1109/TNSM.2019.2913710.

[18] Wenjun Zhu et al. "Research and Implementation of High Performance Traffic Processing Based on Intel DPDK". In: *2018 9th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*. 2018, pp. 62–68. DOI: 10.1109/PAAP.2018.00018.

Appendix

A. Moongen packet generator -> tcpdump

For tcpdump, we make sure that the interface 04:00.0 is using the following driver: ixgbe. If not, we do the following:

Listing 1. bash version
```
$ dpdk-devbind -b ixgbe 04:00.0
$ ifconfig enp4s0f0 up
$ tcpdump -i enp4s0f0
```

For this subsection and all the following ones, once we finish setting up the packet receiver, we open a new terminal and configure a NIC port in order to run moongen packet generator over dpdk. We do the following:

Listing 2. bash version
```
# bind an interface to dpdk
$ dpdk-devbind -b uio_pci_generic 05:00.1
$ timeout 60s PRIM/src/moongen.sh
```

It should be noted that packet generator is defined in throughput_one_port.lua with the following configuration:

- cores = 13,14,15,16,17,18,19,20,21
- pciWhitelist = "0000:05:00.1"
- "–socket-mem", "256,256"

B. Moongen packet generator -> Moongen dump

With Moongen, we can specify a packet receiver with lua. We make sure that the interface 04:00.0 is using the following driver: uio_pci_generic. If not, we do the following:

Listing 3. bash version
```
$ dpdk-devbind -b uio_pci_generic 04:00.0
```

Once the interface is set up, we run Moongen using the following lua script in libmoon: dump-pkts.lua with the following configuration:

with the following configuration:

- cores = 0,1,2,3,4
- pciWhitelist = "0000:04:00.0"
- "–socket-mem", "256,0"

For simplicity's sake, we can run dump-pkts.lua with Moongen by using the following command:

Listing 4. bash version
```
$ PRIM/src/moongen_dump_pcap.sh
```

C. Moongen packet generator -> dpdk pdump

In order to use dpdk-pdump, the interface 04:00.0 should be under dpdk's control.

Next, we enter dpdk's directory and run the following code:

Listing 5. bash version
```
$ ./dpdk-testpmd -l 0,1-5 -n 4 -a
0000:04:00.0 -- -i --port-topology=chained
```

Afterwards, the testpmd's terminal pops up. Forwarding rules should be set so the traffic can be processed by dpdk-pdump.

Listing 6. bash version
```
$ testpmd> set fwd io
$ testpmd> start
```

Then, we open a new terminal and run dpdk-pdump as a secondary process to testpmd.

Listing 7. bash version
```
$ ./dpdk-pdump -l 6,7-10 -n 4 -- --pdump
'port=0,queue=*,rx-dev=/opt/pcap/capture.pcap'
```

D. Moongen packet generator -> xdp-dump

First, clone the following repository and build its tools: https://github.com/xdp-project/xdp-tools

To use xdp-dump, we make sure that the interface 04:00.0 is using the following driver: ixgbe.

Then, run the following:

Listing 8. bash version
```
$ cd $XDP_TOOLS/xdp-dump
$ xdp-filter load enp4s0f0 -f udp -p deny
$ xdp-filter port 320
$ xdpdump -P -i enp4s0f0 -w <file>
```

E. Moongen packet generator -> bcc

First, clone the following repository and build its tools: https://github.com/iovisor/bcc

BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. BCC makes BPF programs with kernel instrumentation in C and front-ends in Python and lua. It is suited for many tasks, including performance analysis and network traffic control.

In our case, we make sure that the interface 04:00.0 is using the following driver: ixgbe.

Then, run the following:

Listing 9. bash version
```
$ cd PRIM/bcc/
$ ./pkt-filter.py [-i <if_name>] [-m <mode>]
[-f <filter>]
```

F. Moongen packet generator -> p4-ebpf

In this section, we are going to write our packet receiver in P4, transform it to C, and then compile it to an eBPF bytecode to be loaded in the Linux kernel for packet filtering on the listening interface.

First, clone the following repository and build its tools: https://github.com/p4lang/p4c

P4c is a reference compiler for the P4 programming language. We are going to use p4c-ebpf.

In our case, we make sure that the interface 04:00.0 is using the following driver: ixgbe.

The P4 file has been defined in PRIM/p4-ebpf/ under pkt-filter.p4

We run the following:

Listing 10. bash version

```
$ cd PRIM/p4-ebpf/
$ p4c-ebpf pkt-filter.p4 -o pkt-filter.c
$ make -f $P4C/backends/ebpf/runtime/kernel.mk
BPFOBJ=pkt-filter.o P4FILE=pkt-filter.p4
$ tc qdisc add dev enp4s0f0 clsact
$ tc filter add dev enp4s0f0 ingress bpf da obj
pkt-filter.o section prog verbose
```

Once the bytecode loaded in the Linux kernel, we track the id of the program to inspect the map ids with bpftool.

Listing 11. bash version

```
$ tc filter show dev enp4s0f0 ingress
$ bpftool prog show id <prog_id>
$ bpftool map dump id <map_id>
```

G. *Moongen packet generator -> p4-xdp*

In this section, we are going to write our packet receiver in P4, transform it to C, and then compile it to an eBPF bytecode to be loaded in the Linux kernel for packet filtering on the listening interface. The difference lies between this section and the above is the P4 model used due to change in the compiler.

We clone the following repository and build its tools: https://github.com/vmware/p4c-xdp

P4c-xdp presents a P4 compiler backend targeting XDP, the eXpress Data Path.

We make sure that the interface 04:00.0 is using the following driver: ixgbe.

The P4 file has been defined in PRIM/p4-xdp/ under pkt-filter.p4

We run the following:

Listing 12. bash version

```
$ cd PRIM/p4-xdp/
$ cp pkt-filter.p4 Makefile $P4C-XDP/tests/
$ make
$ ip link set dev enp4s0f0 xdp obj pkt-filter.o
```

Once the bytecode loaded in the Linux kernel, we track the id of the program to inspect the map ids with bpftool.

Listing 13. bash version

```
$ ip -d link show enp4s0f0
$ bpftool prog show id <prog_id>
$ bpftool map dump id <map_id>
```

H. *Moongen packet generator -> DPDK custom code*

We create our custom packet dump application using C by including DPDK's headers. We make sure that the interface 04:00.0 is using the following driver: uio_pci_generic.

Afterwards, we run our application using the following command:

Listing 14. bash version

```
$ cd $RTE_SDK/examples/dpdk-dump
$ make
$ ./build/dpdkdump --lcores='(0,1)@(0-5)' -n 4
-m 1024 -w 0000:04:00.0 --file-prefix dpdk_dump
-- -m <mode>
```

I. *Moongen packet generator -> t4p4s -> tcpdump*

In this subsection, we use the l2fwd.p4 script defined in t4p4s to run the switch on ports 04:00.0 and 04:00.1.

So first, we make sure that both interfaces 04:00.0 and 04:00.1 use the uio_pci_generic driver with dpdk-devbind.

Then, we modify the following line in opts-dpdk.cfg:

cores=2 -> ealopts += -l 0,1-7 -n 4 -w 0000:04:00.0 -w 0000:04:00.1 -m 1024 –file-prefix l2t4p4s

Then, we run the following command inside t4p4s directory:

Listing 15. bash version

```
$ ./t4p4s.sh :l2fwd
```

We run tcpdump on enp4s0f0 and moongen packet generator on 05:00.1.

J. *Moongen packet generator -> p4-bmv2 -> tcpdump*

In this subsection, we use the behavioral-model switch bmv2 on both enp4s0f0 and enp4s0f1.

So we make sure that the interfaces 04:00.0, 04:00.1 and 05:00.0 use the driver ixgbe.

Then, we use p4c compiler to compile our p4 script (basic.p4) so that it can be loaded on a bmv2 architecture. Once the compilation done, we run the software switch with simple_switch and access its control plane with simple_switch_CLI.

Listing 16. bash version

```
$ p4c -b bmv2 basic.p4 -o basic.bmv2
$ simple_switch --interface 0@enp4s0f0
--interface 1@enp4s0f1 basic.bmv2/basic.json &
$ simple_switch_CLI
```

Before running the packet generator, we create an entry in the p4 table so that our incoming packets on the interface 04:00.0 get forwarded through port 04:00.1.

Listing 17. bash version

```
$ RuntimeCmd: table_add MyIngress.ipv4_lpm
switch_ports 10.0.0.0/8 => 1
```

Once the entry has been added to the table, we can run tcpdump and moongen packet generator.