Linux Networking Performance Profiling Towards Network Function Virtualisation Improvements

Tariro Mukute[*], Joyce Mwangama[*], Albert A. Lysko[†] [*]
[*]*Department of Electrical Engineering, University of Cape Town*
[1]`mkttar001@myuct.ac.za`
[2]`joyce.mwangama@uct.ac.za`
[†]*Council for Scientific and Industrial Research, South Africa*
[3]`alysko@csir.co.za`

***Abstract*—The mobile networking industry has proposed the adoption of network function virtualisation in the various components of the core network, including the Gi-LAN which houses a large set of network functions. However, virtualisation introduces performance cost or additional processes that degrade the performance of the resultant network functions. In this work we consider the network stack for the most common virtualisation technology, Linux. We model the Linux networking stack based on the detailed analysis and monitoring of the performance of the various processes that occur in the network stack as the packets are forwarded to the network functions for processing. Based on the resultant model, we suggest and evaluate approaches for reducing the performance cost or avoiding unnecessary processes, in the context of Gi-LAN network functions.**

***Index Terms*—performance, profiling, networking, VNF, Virtual Network Function, Linux, eBPF, extended Berkeley Packet Filter, bpftrace**

I. Introduction

The recent growth in mobile networking traffic and the preceded diversity of the traffic coupled with the high Capital Expenditure (CAPEX) in the telecommunication industry have promoted the re-engineering of the current and future communication technologies [1]. Among these re-engineering efforts is the migration from expensive hardware based communication Network Functions (NF) to software based NF deployed on cheaper standard off-the-shelf hardware/servers. Migrating to flexible software-based NF entails running the NF on virtualised platforms. Virtualisation often degrades the performance of the resultant Virtual Network Function (VNF). However, in order to serve this increased growth in network traffic whilst solving for the lower CAPEX, the re-engineered virtualised platforms need to either maintain or improve the performance of the current and emerging communication networks. This entails a deep dive and understanding of the networking of the most common virtualisation technology, the Linux kernel.

The kernel's primary function is to enable the sharing of the hardware resources. This results in the virtualisation of each hardware resource so that it can be accessed by multiple competing processes in execution. The networking consists of the network drivers and the network stack. The network drivers are responsible for moving the packets from the Network Interface Controller (NIC) hardware to the kernel, and the network stack allows the sharing of the NIC hardware resources by maintaining multiple per connection queues and per-NIC queues. To maintain memory isolation between the queues and to absorb the different processing speeds the data is copied across the NIC queues and connections queues. The resultant virtualisation of hardware resources up for competing access; the multi-queue per NIC and per connection approach; the data copies and the context separation between the processes mentioned earlier and VNF form a complex system that affects the performance of the deployed VNF. In order to lessen performance degradation, it is valuable to identify the key sources of reduced performance, e.g. via profiling of the performance of the various processes mentioned above. Such quantification needs to be drawn down first. The results allow to put an extended focus on either improving these processes or adapting them to the context of operation. This can be particularly important in the telecommunication industry where some of these generic networking processes can be omitted or modified resulting in overall improvement.

The profiling of the above functions means tracing the processes inside the kernel. There are a number of tracing tools with different levels of observability of the underlying kernel. In order to get a good performance profiling, the tools need to offer high observability. This study will use the observability capability offered by the extended Berkeley Packet Filter (eBPF) to the Linux kernel. The eBPF allows safely executing untrusted user-defined eBPF programs inside the kernel. These programs can be written to collect metrics and can be attached to different points in the kernel. The possible attachment points to collect metrics are: i) Kernel functions with *kprobes*, ii) userspace functions with *uprobe*, iii) system calls with *seccomp*, and iv) *tracepoints*. The high observability feature becomes particularly important in the performance profiling as most of the kernel processes mentioned in the earlier paragraph are kernel functions with *kprobes* and/or *tracepoints* as we will illustrate in the later sections.

This study gives a detailed analysis of the networking of the recent Linux kernel version 5.4. We detail the receive path of the network traffic/packets, and state the various kernel functions and processes applied to the network traffic/packets, from the point it is received on the NIC hardware to the destined application. Thereafter we profile the performance of these functions and processes using our custom eBPF programs that we attach to the *kprobes* and *tracepoints* of these functions and processes. We define performance using the metrics frequency and time taken to execute each of the kernel functions and processes. These metrics are collected by our eBPF programs. We collect the metrics at different network bandwidths whilst the CPU is operating at full load. Lastly, this study presents the performance profiling of the networking of the Linux kernel version 5.4 and recommendation on processes that can be improved or omitted in the context of the telecommunication industry applications like, Firewalls etc.

The following section gives a detailed analysis and models the receive path of the Linux kernel networking based on the study and reference of the Linux source code. Thereafter Section III introduces eBPF tracing and the different attachment points used in this study. The next is Section IV, which describes the set-up for collecting the metrics, i.e., traffic generation and the System Under Test (SUT). Section V then presents and discusses the collected performance profile. Lastly, Section VI gives recommendations and offers conclusions on the study.

II. Related Works

Performance profiling of the Linux kernel has been an ongoing methodology for understanding the Linux kernel for years. This methodology has been enabled by the various tracing tools that were developed over the years, which including *SystemTap* [2], *top*, *iostat* and *vmstat* [3] among many other tools. On top of these tools, the profiling can also be derived from reading and monitoring the statistics collected into system files by the kernel. For example, reading and monitoring the `/proc/interrupts` can help us see how the number and rate of hardware interrupts change as packets arrive. However, these tools generally have limited observability of the Linux kernel. They cannot trace some parts of the Linux kernel, limiting the effectiveness of the performance profiling methodology. In addition, Linux will process a fair amount of packets in the context of whatever is running in the CPU when Software Interrupt Request (SoftIRQ) is handled. Therefore, in most cases, system accounting will attribute those CPU cycles to the process that was running at that moment. For example, *top* can report that a process is using 99+% CPU while, in reality, maybe 60% of that time is spent process packets. Only when there is more work for New API (NAPI) and the work is deferred to another SoftIRQ cycle that the system becomes more transparent and processes the packets under the context of SoftIRQ [4].

The authors in [5] describe the networking receive path of the Linux kernel version 2.6. The NIC and Device Driver Processing are modelled as a token bucket algorithm and the rest modelled as queuing processes. The authors look at the key factors that affect Linux systems' network performance correlating it to the models. During this process, the tracing is limited to the number of used packet descriptors and the transmit and receive rates of the system. Limited information is provided regarding the performance of the various functions and processes identified during the analysis of the networking receive path of the Linux kernel.

Another study by Joe Damato [6] considers the receive path of the Linux kernel version 3.13.0 and Intel's *igb* network driver. In addition to giving a detailed description of the functions and processes that occur as a packet transverses through the receive path, the author states the system files that can be monitored for performance profiling. Although the author provides more observability of the Linux kernel than the previous study, the observability can still be improved. Additionally, the author only gives a description and does not present any results.

In this study, we analyse the Linux kernel version 5.4 and Intel's *i40e* network driver. We use the eBPF programs for tracing, which we attach to the Kernel functions' *kprobes* and *tracepoints* to collect metrics for performance profiling. The eBPF programs can be attached to monitor most kernel functions (through *kprobes* and *tracepoints*) hence allow for better observability as compared to other tools and methods mentioned earlier. Additionally, because we author these eBPF programs, we control the metrics we collect, and how we collect them. There are various ways of writing the eBPF programs [7]. Another important attribute is that by using eBPF programs, we can obtain the time that has been spent on a particular packet processing function and that we do not need to rely on the context the CPU is running under. From the eBPF programs, we can also tell in which program's context the packet processing is occurring.

In this study, we make use of *bpftrace* to write our eBPF programs. *Bpftrace* was created by Alastair Robertson. It uses LLVM [8] as a backend to compile scripts to BPF-bytecode and makes use of BPF Compiler Collection (BCC) for interacting with the Linux BPF system, as well as existing Linux tracing capabilities: kprobes, uprobes, and tracepoints [7]. We use *bpftrace* because it makes writing eBPF tracing programs easier and is well suited for short scripts and ad-hoc investigations [7]. These benefits fit the scope of our work.

III. Linux Networking Stack

The Linux networking stack on packet ingress carries the packet through various functions in order to deliver it to the destination application. This process can generally be dived into three parts:

- Packet is read from the NIC and put into kernel buffers for further processing (*NIC and Device Driver Processing*).
- The packet goes through protocol processing and is delivered to the destination socket (*Packet Protocol Processing*).
- The application listening on the destination socket receives the packet (*Application Processing*).

During these processes, the packets are applied to different functions defined across the networking stack, which either drop the packet or continue with processing. Fig 1 depicts the network stack, from packet arriving on the NIC and being delivered to the destination socket being read by the intended application. These processes are described in detail in the following subsections.

A. NIC and Device Driver Processing

This is the initial part of the network stack for processing network packets. The network packets are received by the NIC and transferred into kernel buffer network data structure (`struct sk_buff`) for further processing up the network stack. The process is managed and controlled by the NIC and device driver. Packets are first transferred from the NIC using Direct Memory Access (DMA) into a DMA-able region on the Random Access Memory (RAM). The memory region for receiving the packets is a ring, `rx_ring`, divided into buffers, `rx_buffer`, referenced by packet descriptors. The `rx_ring` is allocated $\mathcal{D}$ buffers and the respective packet descriptors of each buffer, δ, where the received packets are transferred to. The number of packet descriptors, $\mathcal{D}$, is a design parameter of the NIC and driver. To be able to receive packets, the packet descriptors should be in a ready state, meaning they should be initialised and pre-allocated. When a packet is received (rx), it is transferred into the packet descriptors with a ready state, and if none of the packet descriptors are in ready state, the packet is dropped. Once in the `rx_buffer`, the packet is copied across to the `sk_buff` in the kernel. The packet descriptors are refilled as the used `rx_buffer` are read during NAPI poll and prepared for reuse/recycle. Therefore the packets are dropped when `rx_buffer` are not cleaned out timely.

The process described above is triggered by a hardware Interrupt Request (IRQ), raised to let the system know that there is a packet in the `rx_ring`. The IRQ is processed by an interrupt handler which does minimal work and leaves the rest of the packet reception to a SoftIRQ handler.

Hardware interrupts tend to be expensive in terms of central processing unit (CPU) usage. The NAPI was designed to mitigate this by allowing the driver to go into polling mode instead of being hardware interrupted on every packet. The interrupt is only raised when NAPI [9] needs bootstrap, i.e., when it's not enabled. In this case, `napi_schedule` is called, which wakes up the NAPI subsystem to read packets from DMA'd memory region. During this step, further IRQ is disabled to allow the NAPI subsystem to process packets without interruption from the device. The NAPI function `napi_schedule` then raises a SoftIRQ (`NET_RX_SOFTIRQ`), which run the registered SoftIRQ handler (`net_rx_action`) to poll the packets using the device driver's NAPI poll function. Before finally copying the packets into `sk_buff`, early packet processing functions can be applied to the packet using eXpress Data Path (XDP). These are user-defined dynamically loaded hook functions.

To detail the above process we look at the source code for the device driver from intel *i40e*. The IRQ is handled by the function `i40e_intr` shown in Listing 1. The function handles the different types of interrupts (Legacy/MSI/MSI-X) and ultimately enables NAPI using the function `napi_schedule_irqoff` an alias for `napi_schedule` as can be seen in Listing 2. The handler and the NAPI do minimal work and triggers SoftIRQ to do the heavy lifting as shown in Listing 2. The triggered SoftIRQ then runs the handler function `net_rx_action`, which uses NAPI to read packets. If the network packets are not exhausted, another SoftIRQ is raised. The effective poll function called by `net_rx_action` is `i40e_napi_poll` and is defined by the device driver shown in Listing 3. This function, as shown, reads the packets from the `rx_ring` and ensures that the budget (i.e., the number of packets read in a single poll) is not exceeded. From the code, we can see that the packets can be polled using zero copy (`i40e_clean_rx_irq_zc`), allowing for early actions to be performed on the packet without copying it into `sk_buff`. Alternatively, early actions can be performed on the packet after copying it into the kernel.

When zero copy is enabled, the driver runs the XDP hook function(s) without copying the packets to an `xdp_buff`. XDP functions can return four types of responses, `XDP_PASS`, `XDP_DROP`, `XDP_TX`, `XDP_REDIRECT`, `XDP_ABORTED`. As shown in the Listing 5, the `sk_buff` is only created when the XDP returns `XDP_PASS`, saving packet processing time and overheads for certain packets. Once that is done, the `napi_gro_receive` function is called to perform Generic Receive Offloading (GRO). If zero copy is not enabled in (please refer to Listing 6), the driver starts by creating and copying the packet to an `xdp_buff`. It then runs the XDP function and, depending on the response, it can either create a `sk_buff` (build/construct) or continue to the next packet in the `rx_ring`. When the XDP action is `XDP_PASS`, after creating the `sk_buff`, the `napi_gro_receive` function is called to perform GRO. This polling process is repeated until either the budget has been finished or there are no more packets. When the polling process is done, `napi_complete_done` is called, which passes that packet for protocol processing as detailed in the next subsection. This packet flow, along with the packet flow processing described in the next subsection, is depicted in Figure 2.

B. Packet Protocol Processing

This processing follows after the packet has been copied into the `sk_buff`. The network packets in the `sk_buff` are delivered to packet taps devices or the protocol layer before queuing the packet data to a socket `sock`. This packet protocol processing can be initiated by SoftIRQ when interrupts, IRQ, are enabled through the function `netif_receive_skb` (see Listing 8). It can also be initiated when `napi_poll` completes through `napi_complete_done` (see Listing 7). *The first case happens when the scheduler has preempted the NAPI poll function has been preempted by the scheduler before passing all the packets (`sk_buff`) up the networking stack.* During this process, for each IP packet that is dequeued from the `rx_ring`.

The steps start by checking if Receive Packet Steering (RPS) is enabled and enqueues the `sk_buff` on another CPU where RPS is enabled. We will focus on the case when RPS is not enabled. In this case, if the system is has a generic XDP function(s) defined, the XDP hook functions are applied to the packet, and the packet is either dropped or continued up the stack, depending on the action returned by the XDP function. The XDP functions here run after the packets have been copied and to `sk_buff`. Thereafter, if

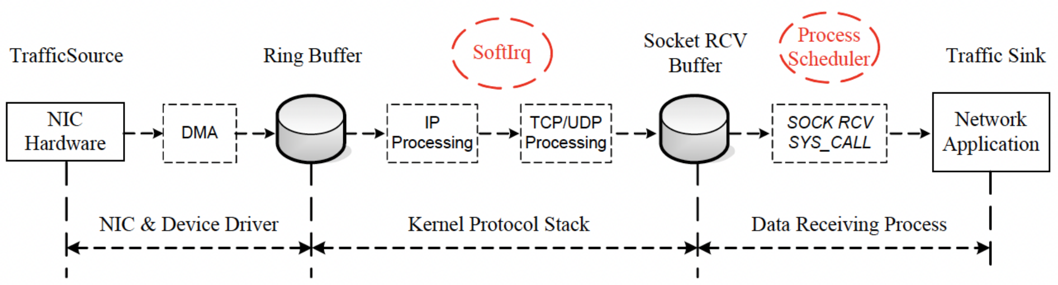
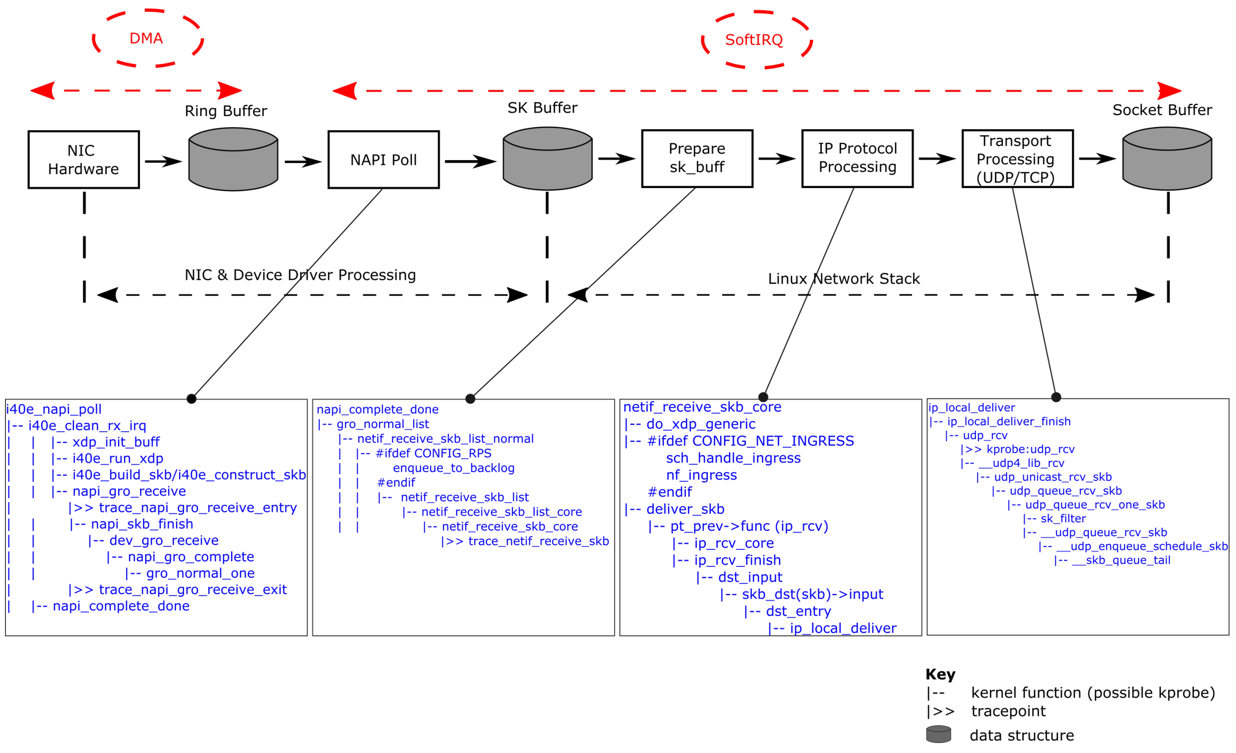


Fig. 1: Linux Networking Subsystem: Packet Receiving Process [5], from receiving by network interface card (NIC) to passing the received data to Network Appicatoin, via various stages in the kernel. DMA stands for direct memory access. SoftIrq stands for soft interrupt request. SOCK RVC denotes socket receive. SYS_CALL points to system call(s).



```
i40e_napi_poll
|-- i40e_clean_rx_irq
|    |   |-- xdp_init_buff
|    |   |-- i40e_run_xdp
|    |   |-- i40e_build_skb/i40e_construct_skb
|    |   |-- napi_gro_receive
|            |>> trace_napi_gro_receive_entry
|    |       |-- napi_skb_finish
|    |           |-- dev_gro_receive
|    |               |-- napi_gro_complete
|    |                   |-- gro_normal_one
|            |>> trace_napi_gro_receive_exit
|    |-- napi_complete_done
```

```
napi_complete_done
|-- gro_normal_list
    |-- netif_receive_skb_list_normal
    |   |-- #ifdef CONFIG_RPS
    |   |       enqueue_to_backlog
    |   |   #endif
    |   |   |--  netif_receive_skb_list
    |   |        |-- netif_receive_skb_list_core
    |   |            |-- netif_receive_skb_core
                         |>> trace_netif_receive_skb
```

```
netif_receive_skb_core
|-- do_xdp_generic
|-- #ifdef CONFIG_NET_INGRESS
        sch_handle_ingress
        nf_ingress
    #endif
|-- deliver_skb
    |-- pt_prev->func (ip_rcv)
        |-- ip_rcv_core
        |-- ip_rcv_finish
            |-- dst_input
                |-- skb_dst(skb)->input
                    |-- dst_entry
                        |-- ip_local_deliver
```

```
ip_local_deliver
|-- ip_local_deliver_finish
    |-- udp_rcv
        |>> kprobe:udp_rcv
        |-- __udp4_lib_rcv
            |-- udp_unicast_rcv_skb
                |-- udp_queue_rcv_skb
                    |-- udp_queue_rcv_one_skb
                        |-- sk_filter
                        |-- __udp_queue_rcv_skb
                            |-- __udp_enqueue_schedule_skb
                                |-- __skb_queue_tail
```

Key

|--	kernel function (possible kprobe)
|>>	tracepoint
⛁	data structure

Fig. 2: Linux Receive Packet Processing Flow, with details around bufferisation and related functions NIC refers to the network interface card. DMA stands for direct memory access. SoftIRQ stands for soft interrupt request.

required, the packet is delivered to the tap device. Following this, if `CONFIG_NET_INGRESS` is enabled, Linux traffic control (TC) classification and actions are applied to the packet. These function can be defined using eBPF or Linux tc command. Right after, netfilter ingress functions are applied to the packet as well. If the packets are not dropped by TC or netfilter, the packet is delivered to the protocol layer by calling `deliver_skb`. This process is shown by the extracted code shown in Listing 8.

The protocol functions to call depend on the packet type. In our case, we will consider IP protocol, which is thereby called by `deliver_skb` through `pt_recv->func`, which calls `ip_rcv`. The Listing 9 shows the processing functions for the IP protocol. The function `ip_rcv_core` does the heavy lifting, in processing and is called from within the `ip_rcv` entry function. The receive function is ended with a call to `ip_rcv_finish`, which is executed through a Netfilter hook function. If netfilter doesn't drop the packet, the process continues in `ip_rcv_finish`. This continues the processing, depending on the destination of the packet. If the packet's destination is the local system, `dst_entry` calls the `ip _local_deliver` function, which calls the network transport function depending on the type of the packet. Like earlier, this function is called from a Netfilter hook (`NF_HOOK`); therefore it's only called if the packet is not dropped by Netfilter. The register network transport functions can be seen from the Listing 10.

We will consider the UDP transport protocol. As shown from the Listing 10, the handler function for UDP is `udp_rcv` and it relies on `__udp4_lib_rcv` to do the heavy lifting. The Listing 11 with function `__udp4_lib_rcv` shows that is the destination socket was predetermined the packet is delivered to that socket; otherwise, the destination socket is looked up first. The process subsequently calls `udp_queue_rcv_one_skb` which first applies socket level eBPF program through `sk_filter_trim_cap`. If the socket level eBPF called in `__udp_queue_rcv_skb` doesn't drop the packet, `__skb_queue_tail` is called, which will put the packet on the receiving socket. The processing listening on the socket is then notified that the packet is now available.

IV. Experiment Set Up

In order to collect our results, we set up two servers, Server 1 and Server 2, on the same subnet, both running Linux kernel version 5.4. Server 1 serves as our System Under Test (SUT) and Server 2 serves as our traffic generator. We generate the traffic using IPerf [10]. As mentioned earlier in Section II, Linux processes packets in the context of whatever program is in the CPU when IRQ is handled. We use two different approaches to this problem. First, we use of eBPF programs, which provide better observability, by monitoring specific packet processing functions. Second, we infer the networking overhead using *openssl speed*, which is a well-known workload that reports how much CPU access it actually gets. We pin the *openssl speed* to a particular CPU, and we use NIC traffic steering to send all traffic from IPerf to the CPU running *openssl speed*. Additionally, in our eBPF programs, we also collect metrics for the CPU that *openssl speed* is pinned on. The CPU will be same CPU IPerf packets are being processed on. We collect context switching overhead which we measure across all CPUs.

In our eBPF programs, we collect the following metrics;

- **total packet poll time**: this is the amount of time spent getting the packets from the `rx_ring` to an `skb_buff`. This is measured from the time `net_rx_action` is called to poll packets, to when `napi_complete_done` is called, which is called when the budget has finished or there are no more packets.
- **skb-ip prep time**: this is the time taken to prepare the `skb_buff` created earlier for processing to the IP protocol. This is measured from when `napi_complete_done` is called to when `netif_receive_skb` is invoked.
- **total IP processing time** this is the time taken to apply IP protocol based functions to the `skb_buff`/packet plus the time taken to execute functions that decide where the packets are to be delivered. Just before the process starts, the following hooks can be applied in the following order: i) generic XDP, ii) tc, iii) netfilter ingress. When the hooks functions are defined, the time the take to execute affects the **total IP processing time**. The textbftotal IP processing time is recorded from when `netif_receive_skb` is called to when `ip_local_deliver` is invoked.
- **total total IP to transport protocol time protocol time**: this the time taken from the moment the packet is set to be delivered locally until the transport protocol to use has been established. In our experiment, this is from the instance `ip_local_deliver` was called, to when `udp_rcv` has been invoked.
- **total ip processing**: is the sum of the times above.
- **cs time**: this is the total amount of time taken processing context switching across all the CPUs
- **total packet processing time**: this is the CPU time that has been from the *openssl speed* due to packet processing. This includes both the receive and transmit path of the networking.

We assume and approximate the receive and transmit path to be asynchronous. We then verify the results from the eBPF programs (first approach) with the results inferred from the *openssl speed* (second approach). The first approach gives the CPU time taken by the receive path processes. The is reported by **total IP processing**. The second approach gives the CPU time taken by both the receive and transmit path processes. This is reported by (**total packet processing time**). Therefore, the **total IP processing** CPU time reported by the eBPF programs should be approximately half of the CPU time reported by *openssl speed*, **total packet processing time**.

V. Results and Discussion

We run our experiment as described in the earlier section. The summary results from the experiment can be shown in Figure 3. From the results in Figure 3, we can see that the processing of IP packets to their respective transport protocol, UDP in this case, takes the most time. This is an example of a packet processing that can be omitted for certain use cases in the core network. For example, consider NF like firewalls which can be deployed in core network. These NF (applications) may not require a continued session as they block the initial packets of a TCP or UDP session and therefore can do without this part.

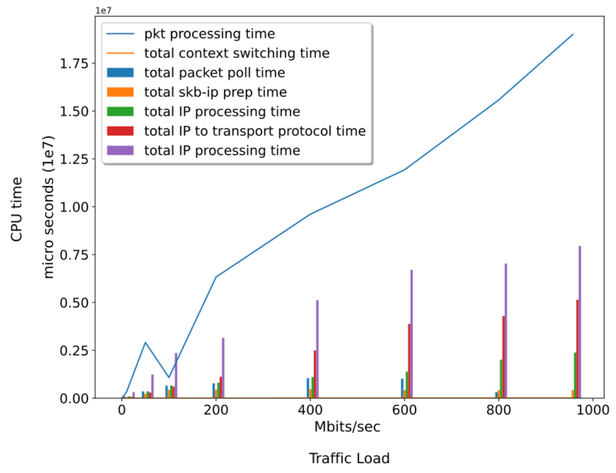


Fig. 3: Linux networking performance profile

Another observation is the **total packet poll time** goes down as the bandwidth increases. This may be because, under high bandwidth, most packets are processed in a single NAPI poll as part of the same budget. Therefore, the NAPI poll function is not called many times. This means that the virtual deployments should be on a path with a network bandwidth above a certain amount, in this case, 800 Mbits per second.

The **total IP processing time** and the **total IP to transport protocol time** increase linearly with the increase in network bandwidth. This observation can be used to approximate the CPU time of those processes under different network bandwidth, enabling better VNF deployment decisions as well as hardware resources provisioning decisions.

An additional observation that does not show up in Figure 3 explicitly is how context switching overhead increases as the bandwidth increases. We show these results in Figure 4. The context switching can be seen to take a lower overhead. However, since it increases linearly, context switching can, at a certain bandwidth, account for a significant part of the CPU cost of networking in the Linux kernel.

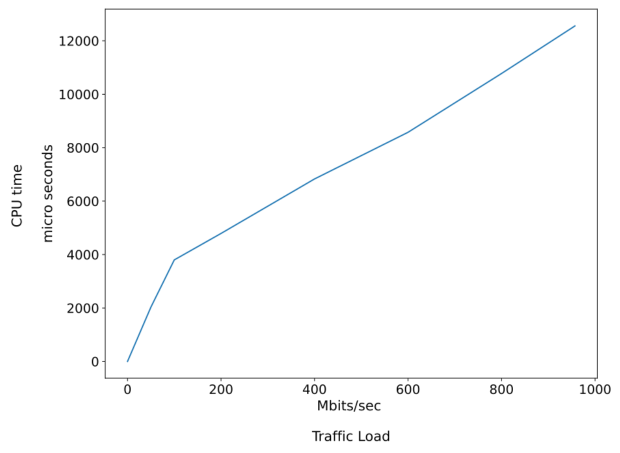


Fig. 4: Linux networking context switching

VI. Conclusion

In this study, we considered the migration to virtual network function in the telecommunication industry, i.e. deploying NF on virtualised platforms, using the Linux kernel. However virtualisation degrades the performance of the resultant network functions. We considered the need to maintain or improve improve the virtual deployments' performance to serve the growing traffic. To address this need, we attempt to quantify the performance degradations in various parts of the networking of the Linux kernel. We adopted performance profiling, reading and understanding the source code. Related works that used performance profiling, have had limited visibility/observability of the underlying functions and processes of the Linux networking. In our study, we used eBPF programs that we attach to Kernel function *kprobes* or *tracepoints*. We write these programs to collect metrics, which we then use for performance profiling. We also make use of inferred performance profiling, where we run a user program and compare its CPU time before and after network traffic. We then use these results to verify the results obtained from our eBPF programs.

Based on our results, we identify processes that take the most time and the processes that can be omitted or modified in the context of the telecommunication industry. In our future work, we plan to attempt to model performance profiling mathematically to approximate the CPU time of networking at different loads across different parts of the Linux networking.

Acknowledgement

The authors acknowledge the support received from TELKOM SA via the Telkom Centre of Excellence (CoE) in Broadband Networks Applications at UCT.

References

[1] J. Costa-Requena, J. L. Santos, V. F. Guasch, K. Ahokas, G. Premsankar, S. Luukkainen, O. L. Perez, M. U. Itzazelaia, I. Ahmad, M. Liyanage, M. Ylianttila, E. M. de Oca, "SDN and NFV integration in generalized mobile network architecture," in *2015 European Conference on Networks and Communications*, June 2015, pp. 154–158.

[2] F. C. Eigler and R. Hat, "Problem solving with systemtap," in *Proc. of the Ottawa Linux Symposium*. Citeseer, 2006, pp. 261–268.

[3] B. Gregg, "Linux performance analysis and tools," Technical report, Joyent, Tech. Rep., 2013.

[4] D. Ahern, "The CPU Cost of Networking on a Host," 2020, accessed on Jun 2021. [Online]. Available: https://people.kernel.org/dsahern/the-cpu-cost-of-networking-on-a-host

[5] W. Wu, M. Crawford, and M. Bowden, "The performance analysis of linux networking–packet receiving," *Computer Communications*, vol. 30, no. 5, pp. 1044–1057, 2007.

[6] J. Damato, "Monitoring and tuning the linux networking stack: Receiving data," 2016, accessed on Jun 2021. [Online]. Available: https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/

[7] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–7.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[9] The Linux Foundation. "napi". Accessed on Jun 2021. [Online]. Available: https://wiki.linuxfoundation.org/networking/napi

[10] A. Tirumala, "Iperf: The TCP/UDP bandwidth measurement tool," *http://dast. nlanr. net/Projects/Iperf/*, 1999.

Tariro Mukute received his BSc degree in Electrical and Computer engineering in 2016 at the University of Cape Town. He is currently pursuing a PhD in the Department of Electrical Engineering at the University of Cape Town, Cape Town, South Africa.