Ducked Tails: Trimming the Tail Latency of(f) Packet Processing Systems

Sebastian Gallenmüller, Florian Wiedner, Johannes Naab, Georg Carle
Department of Informatics
Technical University of Munich
Garching near Munich, Munich
{gallenmu | wiedner | naab | carle}@net.in.tum.de

***Abstract*—Latency can be caused by delayed processing of packets on the nodes of a computer network. Latency figures tend to fluctuate, eventually creating substantial spikes leading to a long-tailed latency distribution. The absolute latency value and its distribution over time impact the service quality of computer networks—an essential requirement for novel services such as networked industrial control systems or remote medical procedures.**

In this work, we present our measurement methodology for packet processing systems to determine the latency reliably, and more importantly, its distribution, using highly accurate and precise hardware timestamping on off-the-shelf network interface cards (NICs). Further, we introduce an optimized software stack to run low-latency applications on regular Linux servers. Our investigation focuses on realtime features of the Linux kernel. The performance of our optimized software stack is demonstrated using a real-world application, the Snort intrusion prevention system (IPS). Across various scenarios, we achieve a maximum worst-case latency as low as 25 µs. This result is an almost 5-fold reduction of the measured tail latencies compared to a previous study.

***Index Terms*—5G, URLLC, IPS, latency measurements, DPDK, MoonGen**

I. Introduction

Typical examples of critical networked systems are industrial control systems or remote medical procedures. In these systems, networks have become an integral part, where network degradation may cause damage to equipment or even human life. Facing such consequences, networks for these applications require special attention to ensure reliable, secure, and smooth operation.

5G networks provide a dedicated service for ultra-reliable and low-latency communication (URLLC) that requires end-to-end reliability up to 6-nines and latency as low as 1 ms [1]. These requirements are especially challenging for systems that involve software packet processing systems. Packet processing tasks, hosted on off-the-shelf hardware, are subject to slow memory accesses, operating system interrupts, or system resources shared across different processes. Such adversities may introduce undesirably high latency, preventing the successful operation of URLLC. This work describes and employs various hardware acceleration techniques and an optimized software stack to provide URLLC-compliant service levels.

The design of low-latency software packet processing systems is only part of our investigation. Equally challenging is the design of measurement infrastructures to investigate such systems. Measurement equipment needs to handle the traffic load of modern systems handling up to 10 Gbit/s or even more. At the same time, rare latency spikes need to be measured reliably as URLLC does not tolerate them because of their potentially destructive effects. This requires the latency measurement of all observable packets.

In this work, we present:

- a highly optimized software stack for low-latency network functions on off-the-shelf hardware,
- the creation and application of a measurement methodology for the investigation of low-latency network functions, and
- a case study of our software stack and measurement methodology that demonstrates the performance and high precision.

The remainder of the paper is structured as follows. Section II presents related work. In Section III, we introduce our low-latency software stack architecture. We give an overview of our measurement methodology in Section IV. Our case study in Section V investigates the performance of our proposed system. We explain the steps necessary to reproduce the presented experiments in Section VI. Finally, Section VII concludes the paper.

II. Background and Related Work

In this section, we introduce work investigating the latency of packet processing applications on off-the-shelf hardware and measurement methodologies for low-latency systems.

a) Low-latency measurements: Several guides exist for tuning Linux [2]–[5] to reduce the latency for packet processing applications through measures such as core isolation, disabling of virtual cores or energy-saving mechanisms, and reducing the number of interrupts. Li et al. [6] investigate the latency of Nginx and Memcached, focusing on rare latency events. Their investigations stress the importance of tail latency analysis, especially considering network applications that perform the same tasks with high repetition rates. The repetition rates increase the probability of observing seemingly rare events and their impact on the overall application performance. Popescu et al. [7] demonstrate that latency increases as low as 10 µs can have a noticeable impact on applications, e.g., Memcached. In previous work [8], [9], we demonstrated that

high reliability and low latency could be achieved on off-the-shelf hardware and virtualized systems, using a Data Plane Development Kit (DPDK)-accelerated Snort IPS. However, the latency was still subject to interrupts causing latency spikes in the µs-range. Hardware and software systems specially built for the embedded domain can offer low latencies without tails even for virtualized systems, e.g., jailhouse [10]. We will not consider those solutions here, as they require adapted or redesigned software. For our solution, we prefer to use regular applications on standard Linux systems running on off-the-shelf hardware.

b) Measurement methodology: MoonGen [11] offers accurate and precise hardware timestamping on widely available Intel NICs. However, due to hardware limitations, most 10G NICs cannot timestamp the entire traffic, but a small fraction of it (approx. 1 kpkt/s). We also demonstrated that creating reliable timestamp measurements using software packet generators is challenging [12]. Although the software solution can timestamp high throughput rates, its expressiveness is limited. The software timestamping process is subject to the same effects, such as interrupts, causing latency spikes on the investigated system. This behavior makes it hard to attribute latency spikes to either the investigated system or the load generator. Specialized hardware [13], [14] offers line-rate high-precision timestamping on multiple 10G Ethernet ports but requires additional hardware. A study by Primorac et al. [15] compared MoonGen's timestamping to various software and hardware timestamping solutions. They concluded that MoonGen's hardware timestamping method offers a similar accuracy and precision compared to a professional timestamping hardware solution. Further, they recommend hardware timestamping solutions for investigating latencies in the µs-range.

Increased requirements regarding latency and reliability demand a reevaluation of measurements and their methodology. In this work, we aim to create a software stack architecture that removes interrupts entirely, to provide ultra-reliability combined with low latency. At the same time, we need a powerful measurement infrastructure and measurement approach to observe these systems with the necessary accuracy and precision. Therefore, we present a measurement methodology that can handle the challenging scenario of high packet throughput paired with precise and accurate latency measurements.

III. Low-Latency Software Architecture on Commodity Hardware

This section describes the key factors of our latency-optimized software architecture. The presented software architecture is derived from various tuning guides [2]–[5] and our own previous work [8]. In this work, we focus our investigation on reducing the tail latency by decreasing the number of OS interrupts as much as possible on virtualized and non-virtualized setups. We have already investigated a low latency system [8], using all the parameters we present in the following with the exception of the *nohz_full* option.

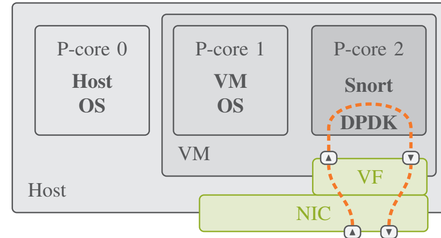

Fig. 1. Software stack architecture

Therefore, we focus our investigation on the *nohz_full* option and its effect on tail latency.

We only consider real CPU cores or physical cores (p-cores) for our investigation. Virtual cores (v-cores) are disabled via the *nosmt* kernel boot parameter. Disabling is necessary to avoid the impact of unwanted resource contention between p- and v-cores.

Figure 1 visualizes the distribution of CPU cores between VM and host operating system (OS) and between OS and application. For the given three-core CPU, the host OS uses P-core 0 exclusively; the VM runs on P-cores 1 and 2, with P-core 1 executing the VM OS and P-core 2 the investigated application, e.g., Snort. The *isolcpu* boot parameter enforces this isolation by preventing the Linux scheduler from scheduling processes onto specific cores. P-cores 1 and 2 are isolated from the host OS, P-core 2 is isolated from the VM OS. In this configuration, neither OS can schedule processes onto P-core 2, creating the perfect environment for the uninterrupted execution of our packet processing application.

Our previous work [8] shows that OS interrupts happen on isolated cores causing latency spikes up to approx. 20 µs. The boot parameter *nohz_full* disables scheduling interrupts on the specific cores when they are running only a single application. Neither the default nor the realtime (RT) kernel of Debian have the necessary options enabled. Therefore, the kernel must be recompiled with the configuration options *CONFIG_NO_HZ_FULL* and *CONFIG_RCU_NOCB_FULL* activated. The read-copy-update (RCU) is a synchronization mechanism in the Linux kernel, that may cause callbacks handled by interrupts on specific cores. The two boot parameters *rcu_nocbs* and *rcu_nocb_poll* shift in-kernel RCU handling to different cores, avoiding interrupts.

Devices, such as NICs, can trigger interrupts to signal the reception of new packets. Setting the *irqaffinity* to P-core 0 forces them to be handled on the designated OS core. The packets received via DPDK do not use this mechanism, but the receiving application polls the NIC directly.

To keep the CPU always in its most reactive state, we use the options *idle* and *intel_idle.max_cstate*. In addition, the intel pstate driver is disabled to avoid switching the CPU into power-saving states (*intel_pstate*). Switching off energy-saving mechanisms can improve latency beyond the 99.99th percentile by approx. 10 µs according to Primorac et al. [15].

Linux assumes the time stamp counter (TSC) clock to be unreliable and regularly checks whether the TSC frequency is

TABLE I
LATENCY OPTIMIZED BOOTPARAMETERS

Parameter	Value	Description
nosmt[1]		Disable virtual CPU cores
isolcpus	[*cores*]	Isolate from kernel scheduler
nohz_full	[*cores*]	No timer ticks
rcu_nocbs	[*cores*]	No RCU callbacks
rcu_nocbs_poll		No RCU callback threads wackup
irqaffinity	0	Interrupts on specific core
idle	poll	Poll mode when core idle
intel_idle.max_cstate	0	Limit CPU to c-state
intel_pstate	disable	Power state driver disabled
tsc	reliable	Rely on TSC without check
mce	ignore_ce	Ignore corrected errors
audit	0	Disable audit messages
nmi_watchdog	0	Disable NMI watchdog
skew_tick	1	No simultaneous ticks for locks
nosoftlookup		Disables logging of backtraces

[1]Only used on VM host

correct. The option *tsc=reliable* disables these regular checks avoiding interrupts [5]. These checks can be disabled safely for modern Intel core-based microarchitectures, where the TSC is invariant, i.e., independent of the CPU's clock frequency [16]. Correcting errors and scanning for errors can cause additional periodic latency spikes in our measurements, *mce=ignore_ce* ignores corrected errors. The parameter *audit=0* disables the internal audit subsystem, which causes load on each core, interrupting programs.

In addition, using *nmi_watchdog=0* disables another watchdog. This watchdog uses the intrastructure of the perf profiling utility, causing additional overhead for our low-latency system. The option *skew_tick=1* shifts the periodic ticks between different CPU cores. This helps to avoid resource contention initiated by a tick happening on all CPU cores simultaneously. For diagnostic purposes, the Linux kernel creates logs for long-running processes. The parameter *nosoftlockup* disables these logs, as we want to avoid the logging overhead for our investigated application [3].

We compiled a list of our used parameters and the respective values in Table I. This list briefly introduces additional measures to lower unwanted interruptions for our packet processing application. All parameters, except *nosmt*, are used for both, the VM and the VM host.

Some settings need to be set on the corresponding machine during run-time. We set the virtual memory statistics collector interval to 3600 s for reducing the time of recalculating those statistics. Intel CAT is used to statically assign the last level cache (LLC) to cores, reducing delays caused by cache contention.

We use DPDK to reduce the impact of the Linux Kernel on networking applications. DPDK shifts the entire packet processing tasks, including drivers, to the userspace. DPDK's drivers poll the NIC for new packets, entirely avoiding interrupts. By preventing these packet reception interrupts, packet processing happens more predictably. The Linux networking API (NAPI) tries to reduce the number of interrupts generated but still relies on them [17]. Therefore, the NAPI itself will cause interrupts impacting network performance and latency. To handle network IO on VMs, we use single root IO virtualization (SR-IOV). SR-IOV splits NICs into independent virtual functions (VFs) that can be directly bound to VMs. VFs reduce latency by bypassing the handling of packets by the VM host.

IV. MEASUREMENT METHODOLOGY

This section presents the main challenges of performing high-performance latency measurements for packet processing applications. Afterward, we describe our toolchain and measurement setup for our subsequent case study.

a) Accuracy vs. precision: The quality of latency measurements can be evaluated along two dimensions—accuracy and precision. For our measurements, we consider accuracy as a measure to describe how close a measured timestamp is to the real event. Precision is defined as the statistical variability between different measurements, i.e., how close the individual measurements are to each other. Low latency measurements require high accuracy, as the already low measurement values reduce the tolerable margin of error. A low precision measurement system may heavily impact tail latency measurements through statistical errors introduced by the measurement system itself. Therefore, high precision is essential to measure rare events reliably.

b) Software timestamping vs. hardware timestamping: Packet reception on modern servers happens asynchronously, i.e., received packets are copied from NIC to RAM and reception is signaled to the CPU eventually. Software timestamping can only happen after the reception is announced to the CPU, causing low accuracy. Without the optimizations mentioned in Section III, interrupts caused by the OS may eventually delay the timestamping process of the CPU, causing a low precision. The previously mentioned problems do not impact hardware timestamps: packets are timestamped shortly and accurately after reception on the NIC itself, and they are timestamped precisely, not impacted by OS interrupts.

c) MoonGen: MoonGen [11] is a packet generator that supports hardware timestamping without relying on specialized and expensive hardware. It uses the hardware timestamping features of widely deployed Intel 10G and 40G NICs, such as the X520, X710, or XL710 [18], [19]. The hardware timestamping feature was integrated into these NICs to provide precise timestamps for the precision time protocol (PTP). NICs that implement PTP in hardware do typically not support timestamping all packets at line rate. Therefore, MoonGen relies on a sampling process, i.e., only up to 1 kpkt/s are timestamped. This is a severe limitation, as the sampling would require extensive measurement times to observe rare latency events reliably.

To capture tail latencies more effectively, we prefer timestamping the entire packet stream. The Intel X550 NIC [20] offers hardware timestamping of all packets with a resolution of 12.5 ns. However, the NIC can only timestamp all received packets, not the transferred packets. To timestamp the outgoing traffic, we introduce an optical splitter or terminal access point

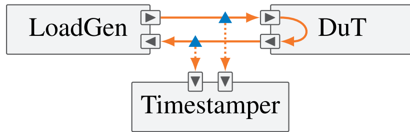

Fig. 2. Measurement setup with external timestamper

(TAP) into our measurement setup. An example of such a setup is shown in Figure 2. In this setup, a separate timestamper is introduced that taps into the optical fiber connection. This setup allows timestamping the entire ingoing and outgoing unidirectional traffic between the two other network nodes. The optical splitter allows a third interface to tap into an optical fiber connection and timestamp all packets sent by another interface. Tapping works passively, therefore, only a static offset is introduced to our latency measurements due to slightly longer fibers for the measurement setup. The impact of the additional fibers can be calculated using the length of the fiber and the medium propagation speed (approx. 0.72 c [11]). The passive component does not impact the precision of our measurement; the impact on accuracy can be corrected if the fiber lengths are known.

MoonGen supports the timestamping method of X550-based NICs through a userscript called MoonSniff. We determine the forwarding latency in three steps. First, we use MoonSniff to record timestamped pcaps of the ingress and egress interface of a device under test (DuT). Second, we extract packet signatures from the pcaps and import them into a PostgreSQL database [21]. Third, we match the packets from the ingress pcap to their respective counterpart of the egress pcap. This kind of matching can be efficiently computed using database joins. The join operation can be adapted to consider specific parts of the packet header to identify matching packets. After the matching packets have been identified, the database can calculate the forwarding latency using the packets' timestamps. In this database-driven approach, different analyses are realized as SQL statements. We use PostgreSQL to calculate maximum and minimum values, percentiles, latency and jitter histograms, and worst-case latency time series.

V. Case-Study

In the following, we use our measurement methodology to evaluate our proposed low-latency architecture. We focus our analysis on the impact of the *nohz_full* option, we investigated the other optimizations (cf. Section III) in previous work [8], [9].

A. Setup

The setup, shown in Figure 2, is based on the presented measurement methodology. Our setup involves three nodes, the DuT hosting different applications, the load generator (LoadGen) connected to the DuT via two 10G links, and the timestamping device (Timestamper) that monitors both links via optical TAPs. We kept hardware and software identical to our previous work [8], [9], to generate easily comparable results. All three nodes use the Intel Xeon D-1518 SoC (4×2.2 GHz) and its integrated Intel 10G dual-port X552 NIC. The DuT runs Debian buster (kernel v4.19) with the self-compiled kernel described in Section III. We use KVM as hypervisor, DPDK version 18.11 as forwarder, and version 3.0.0-beta of Snort [22].

The test traffic uses constant bit rate (CBR) with 64 B sized packets. We select UDP to avoid any impact of TCP congestion control on latency. The payload of the generated traffic contains an identifier for matching the different packets for the subsequent latency calculation.

B. Case-study scenario

Our case study investigates the differences between network applications in three different scenarios: *DPDK-l2fwd* is the baseline scenario running the integrated L2 forwarder of DPDK with as little processing overhead as possible, *Snort-fwd* represents a simple application scenario running the Snort intrusion-detection system without an active ruleset, acting as forwarder to analyze the additional overhead of Snort, and *Snort-filter* with Snort applying the community ruleset for intrusion detection. Traffic is not filtered to allow timestamping. This scenario quantifies the rule application overhead.

Our system is prepared according to the guidelines introduced in Section III. All measurements are performed in a bare-metal and a virtualized environment.

C. Measurements

We measure the selected scenarios for packet rates between 10 kpkt/s and 120 kpkt/s (using a step width of 10 kpkt/s between measurements). DPDK-based measurements offer consistent and stable performance even for short measurements. Therefore, we perform our investigations for a specific packet rate over a measurement time of 30 s. We report three results for each scenario: the lowest rate as a common baseline across all scenarios, the highest rate without packet loss on the DuT and the lowest rate under overload. For the DPDK forwarder, the reported rates are 10 kpkt/s, 60 kpkt/s, and 120 kpkt/s, as even with 1 Mpkt/s, the DPDK forwarder could not be overloaded.

D. Latency Evaluation

We present our latency figures as high dynamic range histograms (HDR histograms) [23]. HDR histograms report the measured latency (y-axis) in relation to their percentiles (x-axis). The characteristic feature of HDR histograms is their logarithmic x-axis that highlights the tail latency behavior of the observed application.

1) DPDK-l2fwd: The DPDK forwarder, shown in Figure 3, demonstrates the results for three packet rates 10 kpkt/s, 60 kpkt/s, and 120 kpkt/s on *HW* and *VM*. Latency steadily increases from approx. 3 µs to 3.4 µs up to the 99th percentile across all measurements. After that, there is a sudden, steep increase in latency up to 4.1 µs. The increase happens earlier for lower packet rates. It also happens earlier for the virtualized application, compared to the respective rate on the bare-metal application.

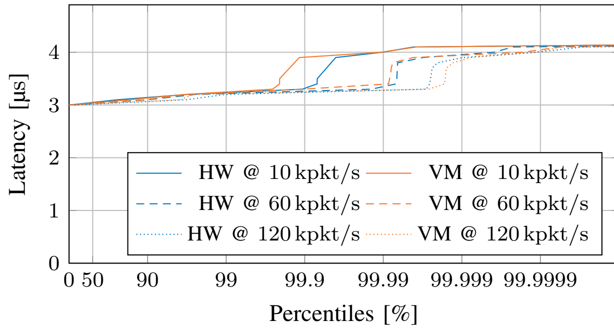

Fig. 3. Forwarding latency of DPDK forwarder for different packet rates on bare-metal (HW) and virtualized (VM) scenario

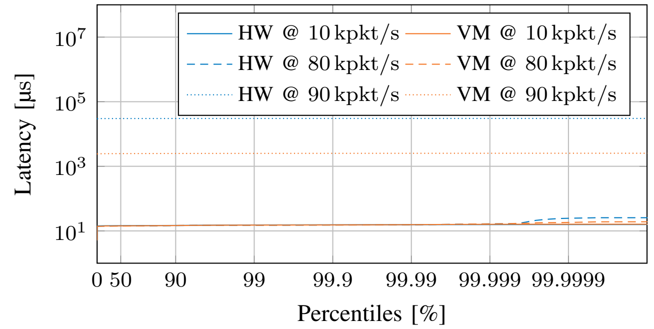

Fig. 5. Forwarding latency of Snort forwarder for different packet rates on bare-metal (HW) and virtualized (VM) scenario

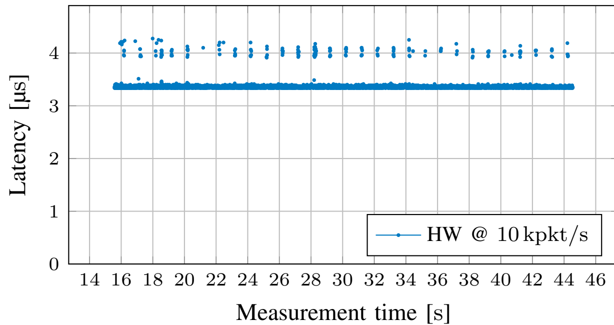

Fig. 4. 5000 worst-case latency events of the DPDK forwarder (HW) measured at a packet rate of 10 kpkt/s

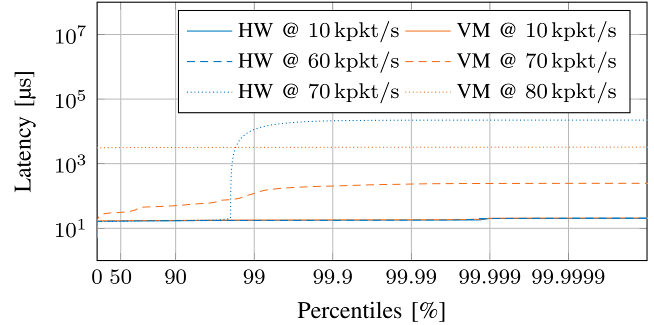

Fig. 6. Forwarding latency of DPDK forwarder for different packet rates on bare-metal (HW) and virtualized (VM) scenario

To analyze the latency events further, Figure 4 shows the 5000 worst-case events for the measurement with 10 kpkt/s over the measurement time. We omitted the plots for the other packet rates due to their high similarity. We exclude the initial 15 s from each measurement because we want to measure the steady-state behavior of our application without the effects caused by the packet generator start-up. Figure 4 shows two levels of latency, the first level at approx. 3.4 µs the second level at approx. 4.1 µs. In between the two levels, there is a gap with very few events. This almost empty gap is the reason for the steep increase in latency shown in Figure 3. The latency level at 4.1 µs shows a regular pattern. With all interrupts disabled, we assume the cause for the regular latency increase to be internal to the forwarding application. However, due to the low impact on latency, we did not investigate this phenomenon any further.

2) Snort-forwarding: Figure 5 shows the latency performance of the Snort forwarder. The Snort forwarder, on HW and VM, offers a median forwarding latency of approx. 14 µs. Up to a packet rate of 80 kpkt/s, the application is not overloaded. For the HW scenario, the worst-case latency is approx. 46 µs, and approx. 20 µs for the VM scenario.

Increasing the packet rates even further leads to an overloaded system; packet loss occurs and latency rises significantly to a level of 30 ms (HW) and 3 ms (VM). The reason for the difference between HW and VM scenarios is the smaller buffer size for the virtualized setup. There, the buffer is split among multiple VFs leading to the observed latency decline.

3) Snort-filtering: The latency of the Snort filtering application is displayed in Figure 6. Due to the more complex processing, the median forwarding latency rises to approx. 17 µs and the application becomes overloaded for lower packet rates. The worst-case latency of the bare-metal forwarder rises to approx. 20 µs for packet rates up to 60 kpkt/s. For the virtualized forwarder, the worst-case latency increases to approx. 76 µs for packet rates up to 70 kpkt/s. In this scenario, the virtualized forwarder works more efficiently, leading to a higher non-overloaded packet rate than its bare-metal counterpart. However, the higher tail latency for the VM scenario and beginning packet loss indicates that the VM setup is on the brink of overloading at the given packet rate. If the packet rate is increased on both systems, respectively, latency again rises significantly to a level of 30 ms (HW) and 3 ms (VM).

E. Impact of nohz_full

In previous work [9], we performed measurements with an identical setup and the same applications running a Linux RT kernel. Figure 7 shows the latency improvements that could be achieved when switching from the RT-based to a NOHZ-based kernel investigated in this paper. Like the HDR plots of our previous measurements, the x-axis lists the percentiles, the y-axis the relative latency improvement of the NOHZ kernel compared to the RT kernel. At the horizontal 100-%-line, both

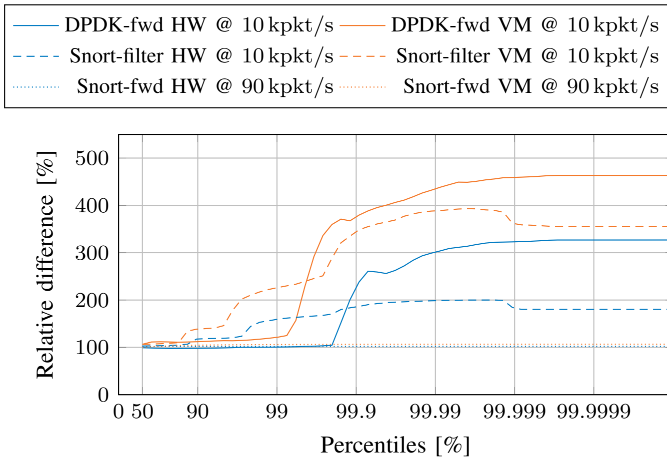

Fig. 7. Relative improvement between RT-based and NOHZ-based kernel for selected scenarios

kernels forward packets with the same latency. Above 100 % the NOHZ kernel offers lower latency, below the RT kernel.

To compare the results, we selected measurements representing three distinct scenarios. In the first scenario, the DPDK forwarder (HW), we see almost the same latency up to the 99th percentile. For higher percentiles, the latency improves by up to 320 %, i.e., the latency of the NOHZ kernel drops to roughly 1/3 compared to the RT kernel. In the VM scenario, the latency of the NOHZ kernel improves across all investigated percentiles. Up to the 99th percentile, the latency improvement remains below 4 %, however, the improvements rise to over 450 % for the higher percentiles. Our previous measurements can help explain the better performance of the NOHZ kernel. On the NOHZ kernel, we measure only minimal overhead for the virtualized scenarios. On the RT kernel, we continuously measured higher latencies in virtualized scenarios. Without this virtualization overhead, the latency figures for the NOHZ kernel are consistently lower than their counterparts measured on the RT kernel.

The second scenario investigates the latency differences for the Snort filter. In the non-virtualized scenario, the latency improvements start to show at the 90th percentile and rise to almost 200 %, i.e., the latency is approximately halved. For higher percentiles, the latency improvements drop to a level of 190 %. In the VM scenario, latency improvements are more significant. The improvements reach approx. 400 % at the 99.99th percentile and drop to 350 % for higher percentiles. When comparing the virtualized and the non-virtualized results, the improvements for the virtualized scenario are visible across all investigated percentiles. These improvements result from the lower virtualization overhead of the NOHZ kernel compared to the RT kernel.

The third scenario compares the latency of the Snort forwarder in an overload scenario. Here, the differences between the RT and the NOHZ kernel are lower than 3 % for any given percentile for either VM or HW. In this case, the NOHZ kernel offers hardly any improvement over the RT kernel.

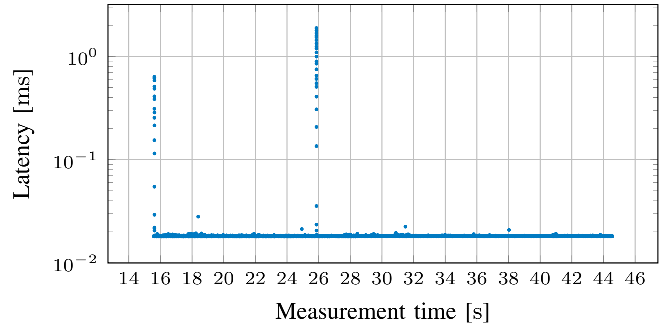

Fig. 8. 5000 worst-case latency events of the Snort forwarder (HW) measured at a packet rate of 10 kpkt/s

F. Latency spikes

The previous section demonstrated significant latency improvements that can be achieved using our optimization techniques. Especially the NOHZ kernel prevents any interrupts on specific CPU cores and offers consistently lower latency than the already optimized RT kernel. However, we noticed very rare, but at the same time, highly impactful interrupts in some of our measurements. Figure 8 presents the 5000 worst-case latency events that happened over a single measurement run. There, we measure an overall worst-case latency above 1 ms. A closer investigation identified the TLB shootdown interrupt as the root cause for this latency spike.

The TLB shootdowns are mentioned by Rigtorp [4] as a potential source of latency and jitter. The transition lookaside buffer (TLB) is a cache that accelerates virtual memory address translation by caching previous translation results. Certain events, such as memory unmapping or changing memory access restrictions, require a flush of the TLB for all CPU cores. This flush is realized as interrupt and causes the observed latency spikes.

Rigtorp [4] mentions several cases where the usage of RAM is reorganized, causing TLB shootdowns. Releasing memory from an application back to the kernel can cause TLB shootdowns and should, therefore, be avoided. He further recommends not using other techniques such as transparent hugepages, memory compaction, kernel samepage merging, page migration between different NUMA nodes, or file-backed writable memory mappings.

Our measurements show that (re-)starting our forwarding application after each measurement run causes TLB shootdowns. We can limit the number of TLB shootdowns by starting the forwarding applications only once at the start of our experiment. We do not restart our forwarding application between the individual measurement runs. In that case, the TLB shootdowns happen in the first few minutes during application runtime. After that, all subsequent measurements happen without any further interrupts due to TLB shootdowns. In this paper, we only present the measurements after the initial TLB shootdowns.

VI. Reproducibility

We consider reproducibility a key factor of experimental research. Therefore, we created a github repository [24] and a website [25] that provide all artifacts necessary to reproduce the presented measurements: the experiment and plotting scripts, and the pcap files that were used to create the plots of this paper. The investigated software components such as DPDK or Snort are available publicly.

VII. Conclusion

Measuring the tail latency of packet processing systems is essential. Despite their scarcity, the absolute value of these tail latency events may render a packet processing system unfit for critical URLLC-driven services such as industrial control systems. Our paper presents a methodology to measure these tail latencies and a software stack to lower the tail latencies of packet processing applications.

Our measurement methodology relies on hardware timestamping to accurately and precisely determine latencies, which cannot be achieved using software timestamping. For the first time, we present a function for MoonGen that allows timestamping all, rather than a few selected packets during experiments. Our entire measurement setup relies on free, open-source software and affordable off-the-shelf hardware. The presented low-latency software stack utilizes various techniques applied to typical Linux server systems to provide consistent low-latency packet processing. Our case study demonstrates that a non-overloaded Snort IPS can achieve a consistent forwarding latency below 25 µs.

A particular focus of our case study is the investigation of the NOHZ kernel and its impact on latency. A direct comparison between the NOHZ kernel and an already optimized RT kernel shows a negligible impact on the median latency. However, the impact on tail latency is significant, with tail-latency reductions to as low as roughly 1/5 of their original value on the RT kernel. We still observe sporadic latency events that may cause spikes in the ms-range. However, these events only happen within minutes after the start-up of an application. Afterward, this kind of interrupts are not observed and latency remains at consistently low values.

Our investigations uncoverd regular patterns (cf. Figure 4) that we cannot explain yet. We plan to do further measurements to determine the root cause of the observed pattern.

Acknowledgment

The German Research Foundation funded our research partially (Modanet, grant no. CA595/11-1). Additionally, we received funding by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria.

References

[1] NGMN Alliance, "5G E2E Technology to Support Verticals URLLC Requirements," 2019.

[2] AMD, "Performance Tuning Guidelines for Low Latency Response on AMD EPYC-Based Servers Application Note," Jun. 2018, Last accessed: Sept. 22, 2021. [Online]. Available: http://developer.amd.com/wp-content/resources/56263-Performance-Tuning-Guidelines-PUB.pdf

[3] J. Mario and J. Eder, "Low Latency Performance Tuning for Red Hat Enterprise Linux 7," Nov. 2017, Last accessed: Sept. 22, 2021. [Online]. Available: https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf

[4] E. Rigtorp, "Low latency tuning guide," Mar. 2020, Last accessed: Sept. 22, 2021. [Online]. Available: https://rigtorp.se/low-latency-guide/

[5] M. Beierl, "Nfv-kvm-tuning," Last accessed: Sept. 22, 2021. [Online]. Available: https://wiki.opnfv.org/pages/viewpage.action?pageId=2926179

[6] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[7] D. Popescu, N. Zilberman, and A. Moore, "Characterizing the impact of network latency on cloud-based applications performance," 2017.

[8] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, "5G QoS: Impact of Security Functions on Latency," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020, pp. 1–9. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110422

[9] ——, "5G URLLC: A Case Study on Low-Latency Intrusion Prevention," *IEEE Commun. Mag.*, vol. 58, no. 10, pp. 35–41, 2020. [Online]. Available: https://doi.org/10.1109/MCOM.001.2000467

[10] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits!(almost)," *arXiv preprint arXiv:1705.06932*, 2017.

[11] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, K. Cho, K. Fukuda, V. S. Pai, and N. Spring, Eds. ACM, 2015, pp. 275–287. [Online]. Available: https://doi.org/10.1145/2815675.2815692

[12] P. Emmerich, S. Gallenmüller, G. Antichi, A. W. Moore, and G. Carle, "Mind the Gap - A Comparison of Software Packet Generators," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2017, Beijing, China, May 18-19, 2017*. IEEE Computer Society, 2017, pp. 191–203. [Online]. Available: https://doi.org/10.1109/ANCS.2017.32

[13] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, G. A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski, "OSNT: open source network tester," *IEEE Netw.*, vol. 28, no. 5, pp. 6–12, 2014. [Online]. Available: https://doi.org/10.1109/MNET.2014.6915433

[14] Silicom, "Datasheet PE310G4TSF4I71," Last accessed: Sept. 22, 2021. [Online]. Available: https://www.silicom-usa.com/wp-content/uploads/2016/08/PE310G4TSF4I71-Programmable-Application-Acceleration-10G.pdf

[15] M. Primorac, E. Bugnion, and K. J. Argyraki, "How to measure the killer microsecond," *Comput. Commun. Rev.*, vol. 47, no. 5, pp. 61–66, 2017. [Online]. Available: https://doi.org/10.1145/3155055.3155065

[16] *Intel 64 and IA-32 Architectures Software Developers Manual*, Intel, 6 2021.

[17] J. H. Salim, "When napi comes to town," in *Linux 2005 Conf*, 2005.

[18] *Intel 82599 10 GbE Controller - Datasheet*, Intel, 11 2019, rev. 3.4.

[19] *Intel Ethernet Controller X710/ XXV710/XL710 Datasheet*, Intel, 8 2019, rev. 3.65.

[20] *Intel Ethernet Controller X550 - Datasheet*, Intel, 11 2018, rev. 2.3.

[21] PostgreSQL Global Development Group, "PostgreSQL," Jul 2021. [Online]. Available: https://www.postgresql.org/

[22] "Snort - Network Intrusion Detection and Prevention System," Last accessed: Sept. 22, 2021. [Online]. Available: https://www.snort.org/

[23] G. Tene, "HdrHistogram: A High Dynamic Range Histogram," Last accessed: Sept. 22, 2021. [Online]. Available: http://hdrhistogram.org/

[24] S. Gallenmüller, F. Wiedner, J. Naab, and G. Carle, "Reproducing Evaluation Results," Last accessed: Sept. 22, 2021. [Online]. Available: https://github.com/gallenmu/hipnet21

[25] ——, "Reproducing Evaluation Results," Last accessed: Sept. 22, 2021. [Online]. Available: https://gallenmu.github.io/hipnet21/