Journal of Network and Systems Management (2023) 31:20
https://doi.org/10.1007/s10922-022-09710-3

How Low Can You Go? A Limbo Dance for Low-Latency Network Functions

Sebastian Gallenmüller[1] · Florian Wiedner[1] · Johannes Naab[1] · Georg Carle[1]

Received: 1 March 2022 / Accepted: 22 November 2022 / Published online: 28 December 2022

Abstract

Throughput is a commonly used performance indicator for networks. However, throughput may be considered insignificant if data is outdated or networks become unpredictable or unreliable. Critical services may even prioritize latency, predictability, and reliability at the expense of throughput to avoid detrimental effects on service operation. Latency, predictability, and reliability are distinct qualities realized in real-time systems. Real-time systems often require additional effort using non-standard interfaces, requiring customized software, or providing low throughput figures. This work picks up the challenge and investigates a single-server network function—a building block for end-to-end low-latency network applications. Assessing reliability and quantifying low latency is equally challenging, as sub-microsecond latency and $1/10^5$ loss probability leave little room for error. Both, our measurement and the investigated platforms, rely on Linux running on off-the-shelf components. Our paper provides a comprehensive study on the impact of various components on latency and reliability, such as the central processing unit (CPU), the Linux Kernel, the network card, virtualization features, and the networking application itself. We chose Suricata, an intrusion prevention system (IPS), representing a widely deployed, typical network application as our primary subject of investigation.

Keywords Ultra reliable · Low latency · DPDK · Network experiment · Intrusion prevention

This work is based on our paper presented at HiPNet 2021 [1] and previous low-latency studies [2, 3].

✉ Sebastian Gallenmüller
gallenmu@net.in.tum.de

Extended author information available on the last page of the article

1 Introduction

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 [4]. Investigating URLLC-compliance involves two challenges. First, we need measurement facilities and tools. Our experimental platform must be capable of accurately and precisely determining sub-ms latency. The reliability of the measurement system is key to observing rare events. Second, we want to investigate the design of a software stack capable of hosting URLLC applications. Thus, we need to understand the performance of multiple interacting system components impacting network input / output (IO), such as the network interface card (NIC), the central processing unit (CPU), the operating system (OS), and the application itself.

URLLC 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, applies, and measures various hardware acceleration techniques and an optimized software stack to provide URLLC-compliant service levels. In this work, we target a security-related network function common to many deployments, not only 5G. Here, we investigate Suricata [5], a widely deployed intrusion prevention system (IPS). This investigation considers both quality *and* throughput in the investigation of an IPS.

We aim to achieve the following goals in our paper:

- establishing a measurement methodology and platform that allows accurate and precise latency measurements with a particular focus on tail-latency behavior;
- creating a highly optimized software stack for predictable low-latency network functions on off-the-shelf hardware;
- demonstrating the performance of our software stack depending on various system components such as the Linux kernel, different NICs, and virtualization; and
- deriving guidelines to design and operate applications with a predictably low latency.

The remainder of the paper is structured as follows: We present a motivating example demonstrating the impact of latency and the importance of optimizing systems towards ultra-low latency in Sect. 2. Section 3 introduces related work and the underlying techniques. Based on these techniques, we create a low-latency software stack in Sect. 4. In Sect. 5, we present our measurement methodology and toolchain. Section 6 determines the impact of the previously described technologies and derives guidelines for creating low-latency systems. The limitations of the described solutions are discussed in Sect. 7. All experiment artifacts used in this paper are publicly available; a short introduction how to reproduce our research is given in Sect. 8. Finally, Sect. 9 concludes the paper.

2 Motivating Example

To demonstrate the impact of a typical packet processing task on latency, we measured an application running on a server based on off-the-shelf hardware. The investigated application is Suricata, an IPS. We measured the latency between the ingress and egress port of our IPS for each packet. To avoid packet losses due to overload, our measurement uses a constant packet rate of 10 kpkts/s, well below the maximum capacity that the IPS can handle. To determine the latency, we used hardware timestamps taken on a separate device to ensure that the observed latencies were caused by Suricata, not the measurement device.

A starting application may be subject to higher latencies caused by memory allocation or empty caches. We are only interested in steady-state behavior; therefore, we cut the first second of measurement data. Figure 1 shows a scatter plot of the measured forwarding latency. We filtered for the 5000 worst-case latencies, taken over a 60-second measurement run. We observed a median latency of 47 μs and sudden latency spikes up to 1.8 ms. Repeated measurements led to similar latency spikes. However, we could not determine a regular pattern for these spikes; thus, the latency may suddenly increase, randomly impacting the IPS latency.

This example measurement demonstrates that despite favorable conditions—moderate traffic and pre-loaded application—the investigated system neither provides low nor predictable latency. The height and the randomness of the spikes disqualify such a system to be used for reliable communication, such as URLLC. At the same time, the example highlights the potential for improvement. We measured a 38-fold difference between median and worst-case values. Therefore, we investigate various techniques that promise the realization of services with predictably low latency on off-the-shelf systems. If we improve worst-case latencies significantly, URLLC-compliant packet processing in software running on off-the-shelf hardware becomes feasible.

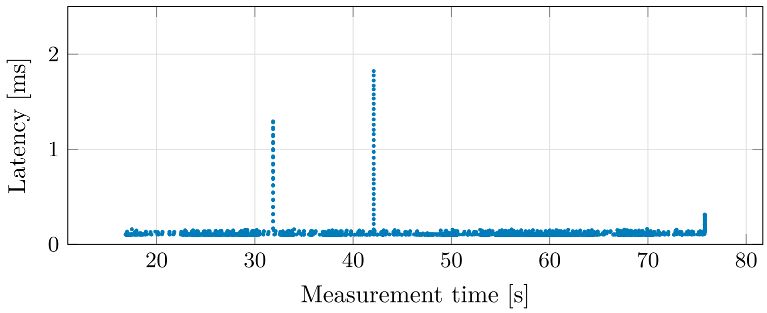

Fig. 1 Worst-case forwarding latencies of Suricata

3 Background and Related Work

This section introduces the background and related work investigating various studies and system components that may introduce latency for packet processing systems. We focus our investigation on software packet processing systems based on Linux and off-the-shelf hardware components. In addition, we examine intrusion prevention systems.

Low-Latency Measurements Several guides exist for tuning Linux [6–9] to reduce the latency for packet processing applications through measures such as core isolation, disabling virtual cores or energy-saving mechanisms, and reducing the number of interrupts. Li et al. [10] 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. Higher repetition rates increase the probability of observing seemingly rare events and their impact on the overall application performance. Popescu et al. [11] demonstrate that latency increases as low as 10 μs can have a noticeable impact on applications, e.g., Memcached. A study by Barroso et al. [12] demonstrates the need for μs-scale latency IO in data center applications. They propose a synchronized programming model to simplify application development for low-latency applications. In previous work [2, 3], we demonstrated that high reliability and low latency could be achieved on off-the-shelf hardware and virtualized systems, using a DPDK-accelerated Snort. However, the latency was still subject to interrupts causing latency spikes in the μs-range.

Hardware Properties HyperThreading (HT), also called simultaneous multithreading (SMT), is a feature of modern CPUs that allows addressing physical cores (p-cores) as multiple virtual cores (v-cores). Each p-core has its own physically separate functional units (FU) to execute processes independently. If multiple v-cores are hosted on a common p-core, FUs are shared between them. Zhang et al. [13] demonstrate that sharing FUs between v-cores can impact application performance when executing processes on v-cores instead of physically separate p-cores.

Another feature of modern CPUs is the support of sleep states, which lower CPU clock frequency and power consumption. Switching the CPU from an energy-saving state to an operational state leads to wake-up latencies. Schöne et al. [14] measured wake-up latencies between 1 μs and 40 μs for Intel CPUs depending on the state transition and the processor architecture.

Despite having physically separate FUs, p-cores share a common last-level cache (LLC). Therefore, processes running on separate p-cores can still impact each other competing on the LLC. Herdrich et al. [15] observed a performance penalty of 64% for a virtualized, DPDK-accelerated application when running in parallel with an application utilizing LLC heavily. The uncontended application performance can be restored for the DPDK application by dividing the LLC statically between CPU cores utilizing the cache allocation technology (CAT) [15, 16] of modern Intel CPUs.

Low-Latency VM IO Transferring packets into/out of a virtual machine (VM) leads to significant performance penalties compared to bare-metal systems.

Emmerich et al. [17] compared packet forwarding in bare-metal and VM scenarios, demonstrating that VMs can introduce high tail latencies of 350 μs and above. They also demonstrated that DPDK could help improve forwarding latencies but must be used on the host system and the VM.

Furthermore, modern NICs, supporting single-root IO virtualization (SR-IOV), can be split into several independent virtual functions, which can be used as independent NICs and bound to VMs exclusively. In this case, virtual switching is done on the NIC itself, minimizing the software stack involved in packet processing. In an investigation by Lettieri et al. [18], SR-IOV, among other techniques for high-speed VM-based network functions, is one of the fastest techniques with the lowest CPU utilization. Therefore, the latency performance of SR-IOV is superior to software switches; e.g., Xu and Davda [19] measured an almost 10-fold increase in worst-case latencies for a software switch. Xiang et al. [20] create and evaluate an architecture for low-latency network functions. Their architecture provides sub-millisecond latencies, but they do not investigate the worst-case behavior. Zilberman et al. [21] give an in-depth latency analysis of various applications and switching devices. They stress the need for tail-latency analysis to analyze application performance comprehensively.

The topic of VM-based network functions has been extensively researched in literature [18–20]. However, given our motivating example in Sect. 2 and the importance of the URLLC service, we argue, similar to Zilberman et al. [21], that the crucial worst-case behavior needs close attention. Hence, we aim to create the lowest latency system achievable by utilizing available applications on off-the-shelf hardware.

There are also embedded systems such as jailhouse [22] or PikeOS [23] that can partition the available hardware providing real-time guarantees for user processes or VMs. However, they are either incompatible with standard Linux interfaces such as libvirt or replace the host OS entirely. Therefore, the tool support for these specialized hypervisors is worse compared to more widespread solutions such as Xen or the kernel-based virtual machine (KVM) utilizing the libvirt software stack. Thus, we do not consider these specialized solutions for this work but rely on well-established software tools and hardware.

Kernel

Reghenzani et al. [24] present an extensive survey on the evolution and features of real-time Linux. Real-time capabilities are added to the regular or vanilla Linux kernel through a set of patches. Over time, these rt patches were incrementally added to the mainline kernel code. A significant feature of these patches is the predictability they introduce to the Linux kernel. They achieve this by increasing the preemptability of kernel code. By allowing preemptability to formerly non-interruptable parts of the kernel code, applications can be scheduled more regularly, avoiding long phases of non-activity.

The Linux kernel uses scheduling-clock interrupts, or short ticks, for scheduling processes [25]. With the introduction of the *tickless* kernel, these interrupts can be entirely disabled for specific cores if the no-hz-full mode is enabled. A no-hz-full core that runs a single process exclusively, disables its ticks and can execute this process in an almost interrupt-free mode. Hosting more than one process on such a core re-enables the tick. The no-hz-full mode can be enabled for all but one core on a system. The remaining core always operates in the non-tickless mode executing potential scheduling tasks for the other cores.

Because of the high relevance of interrupt handling on latency [1–3, 17], we want to investigate the latency impact of the different Linux kernel variants. In this work, we create Linux images with the vanilla, rt, and no-hz kernels being the only difference between them, to ensure comparable results.

Kernel Bypass Techniques Another possible cause for OS interrupts is the occurrence of IO events, e.g., arriving packets, to be handled by the OS immediately. Interrupt handling causes short-time disruptions for currently running processes. The ixgbe network driver [26] and Linux [27] employ moderation techniques to minimize the number of interrupts and, therefore, their impact on processing latency. Both techniques were introduced as a compromise between throughput and latency optimization. For our low-latency design goal, neither technique is optimal, as the interrupts—although reduced in numbers—cause irregular variations in the processing delay, which should be avoided.

DPDK [28], a framework optimized for high-performance packet processing, prevents triggering interrupts for network IO entirely. It ships with its own userspace drivers, which avoid interrupts but poll packets actively instead. This leads to execution times with only minor variation also due to DPDK's preallocation of memory and a lack of costly context switches between userspace and kernelspace. However, polling requires the CPU to wake up regularly, increasing energy consumption.

The Linux Kernel's XDP does not bypass the entire kernel but its network stack, offering throughput and latency improvements [29]. However, in a direct comparison with DPDK, they measured a higher forwarding latency for XDP (202 μs vs. 189 μs). XDP uses an adaptive interrupt-based process for packet reception. Though conserving energy, compared to DPDK's polling strategy, it leads to higher latencies for low packet rates.

PF_RING [30] is a packet processing framework that follows a design philosophy similar to DPDK, shifting the packet processing to userspace. The netmap [31] framework, like XDP, was designed with OS integration in mind. It uses system calls for packet reception and transfer. Though the number of system calls is reduced, netmap still has a higher overhead, increasing the cost and latency of packet IO. In a direct comparison between DPDK, PF_RING, and netmap, DPDK offered higher throughput than netmap and PF_RING, and the latency of DPDK was equal to PF_RING and lower than netmap's latency [32].

Measurement Methodology MoonGen [33] offers accurate and precise hardware timestamping on widely available Intel NICs (cf. Sect. 5). 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 [34]. Although the software solution can timestamp high throughput rates, its expressiveness is limited. The software timestamping process is subject to effects that impact measurements 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.

Dedicated timestamping hardware [35, 36] offers line-rate high-precision and high-accuracy timestamping on multiple 10G Ethernet ports but requires additional hardware, increasing the costs of the measurement setup. A study by Primorac et al. [37] 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.

Intrusion Prevention Intrusion prevention systems are a combination of a firewall with an intrusion detection system. IPSes detect and react to intrusions by identifying and blocking harmful network flows [38]. Security-related network functions like IPS can be subjected to quality of service requirements, for instance, in 5G URLLC.

Our previous studies have demonstrated a maximum latency of approx. 120 μs at a maximum packet rate of 60 kpkt/s for a DPDK-accelerated Snort IPS [2]. A reliable, low-latency service is possible using off-the-shelf hardware when certain operating conditions are met, e.g., exclusive access to system resources or the availability of sufficient compute resources.

A study by Albin et al. [39] measures the performance of the Suricata IPS [5] to be equal to or higher than the performance of Snort. Suricata's architecture allows an approximately linear growth in performance with the number of cores. Suricata supports various kernel bypass frameworks such as PF_RING, netmap, or XDP [40]. DPDK support for Suricata was introduced in December 2021 [41]. Its progressive software architecture and the recently added DPDK support promise equal or better performance than a DPDK-accelerated Snort. This combination makes Suricata an attractive subject for further investigation compared to our previous Snort-centered studies.

Evaluation of the State of the Art Increased requirements regarding latency and reliability demand a reevaluation of measurements and their methodology. Based on state-of-the-art technologies such as kernel bypass, real-time Linux kernels, and hardware-accelerated virtualization, 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.

4 Low-Latency System Design

This section describes the critical factors of our low-latency system design. We continue the work presented in previous studies [1–3] and present the derived low-latency system design based on various tuning guides [6–9]. In particular, we focus our investigation on an updated software stack relying on DPDK 21.11, Debian 11 using Kernel 5.10, and Suricata 7.0. Additionally, we investigate further system components impacting forwarding latency, such as the NIC, virtualization, or software architectures.

Suricata was chosen as a typical example of a security network function. To provide a secure network, such functions need to be applied to a significant portion of the traffic, thereby potentially increasing the latency of every investigated packet. In addition, these security functions may introduce high tail latencies (cf. Sect. 2), violating quality of service requirements such as the 1-ms goal of URLLC connections. In the past, we investigated the Snort 3 IPS [1–3]. In this work, we focus on the Suricata IPS. The architecture of Suricata is focused on multi-core architectures in contrast to Snort, impacting processing performance [42]. Because of these differences, Suricata was chosen as the primary target for this paper.

4.1 OS-Specific Techniques

In the following, we discuss various techniques that can be used to optimize OS settings to create a low-latency environment for hosting packet processing applications.

Figure 2 visualizes the distribution of CPU cores between host OS and the different applications used on top. For the given three-core CPU, the host OS uses P-core 0 exclusively; the intrusion detection software runs on P-cores 1 and 2, with P-core 1 dedicated to a management thread and P-core 2 to a worker thread performing the packet processing tasks. The *isolcpu* (ⓐ) boot parameter enforces this isolation by preventing the Linux scheduler from scheduling other processes onto the isolated cores. In Fig. 2, P-cores 1 and 2 are isolated. According to this configuration, the OS cannot schedule processes onto P-cores 1 and 2, creating the perfect environment for the uninterrupted execution of our packet processing application.

Our previous work [2] 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 specific cores when they are only executing a single thread. However, neither the vanilla nor the real-time (rt) kernel of Debian were compiled with 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.

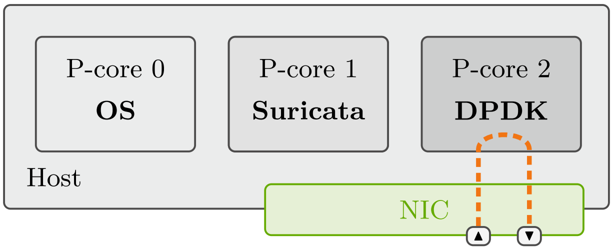

Fig. 2 Software stack architecture

The two boot parameters *rcu_nocbs* (ⓒ) and *rcu_nocb_poll* (ⓓ) shift in-kernel RCU handling to different cores, avoiding interrupts on the nohz-enabled cores.

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, avoiding network-induced interrupts for all other cores. The packets received via DPDK do not use this mechanism, but the receiving application polls for new packets.

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. [37].

Linux assumes the time stamp counter (TSC) clock to be unreliable and regularly checks whether the TSC frequency is correct. The option *tsc=reliable* (ⓚ) disables these regular checks avoiding interrupts [9]. 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 [43]. 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 infrastructure 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 [7].

We compiled a list of used parameters and the respective values in Table 1. Each parameter is labeled to link the explanation in the previous text with the table. This list briefly introduces the applied measures to lower unwanted interruptions for our packet processing application.

Some additional settings need to be set on the corresponding machine during runtime. We set the virtual memory statistics collector interval to 3600 s for reducing the time of recalculating those statistics. The Intel CAT tool [16] is used to statically assign the LLC to cores, reducing delays caused by cache contention.

4.2 Application-Specific Techniques

In this subsection, we discuss the techniques that should be considered when creating a low-latency network application. As an application framework, we suggest the usage of DPDK to reduce the impact of the Linux Kernel on networking applications. DPDK shifts the entire packet processing tasks, including drivers, to the user-space. DPDK's drivers poll the NIC for new packets, entirely avoiding interrupts. By preventing these packet reception interrupts, packet processing happens more

Table 1 Latency optimized bootparameters

	Parameter	Value	Description
ⓐ	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 wake-up
ⓔ	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

predictably. Several similar kernel bypass frameworks exist. However, DPDK's strictly polling-based reception promises the lowest possible latency compared to the other frameworks, such as XDP, netmap, or PF_RING (cf. Sect. 3). The Linux networking API (NAPI) reduces the number of interrupts generated but still relies on them [27]. Therefore, the NAPI itself will cause interrupts, impacting network performance and latency. To incorporate further NICs into our measurement, we use DPDK 21.11, which supports newer NICs, such as the Intel E810. The architecture of the NIC can have an additional impact on latency (cf. Sect. 6.2.2). For comparison, have we compiled the measurements using the same hardware setup as in previous works [1–3].

5 Measurement Methodology

This section presents the main challenges of performing sub-microsecond latency measurements. Afterward, we describe our toolchain and measurement setup for our subsequent case study.

Reliability We assess the reliability of a connection by quantifying its packet loss. In the context of this paper, the highest level of reliability is achieved if no packets are lost between the ingress and the egress port of an investigated system. Reliability is equally crucial for the measurement equipment, i.e., no packet loss should happen for the traffic sent to and received from an investigated system. A highly reliable, i.e., loss-free, measurement system is essential to measure rare latency events, as these events may be missed on a lossy measurement system.

Accuracy vs. Precision The quality of latency measurements can be evaluated along two dimensions—accuracy and precision. According to ISO [44], *accuracy* describes the "closeness of agreement between a test result and the accepted reference value" and *precision* refers to the "closeness of agreement between independent test results." Applying these definitions to our measurements, we consider accuracy as a measure to describe how close a measured timestamp is to the actual 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 error margin. The difference between accuracy and precision is visualized in Figure 3. 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.

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, which introduces additional latency, causing low accuracy. Without the optimizations mentioned in Sect. 4, interrupts caused by the OS may eventually delay the timestamping process of the CPU, causing 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. With hardware timestamping improving both, precision and latency, hardware timestamping is the superior measurement method compared to software timestamping.

MoonGen MoonGen [34] 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, X550, X710, or XL710 [45, 46]. 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 [47] offers hardware timestamping of all packets with a resolution of 12.5 ns. However, the NIC can only timestamp all the received packets, not the sent packets. To timestamp the outgoing traffic, we introduce an optical splitter or terminal access point (TAP) into our measurement setup. An example of such a setup is shown in Fig. 4. 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

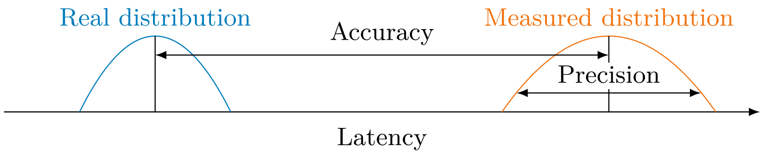

Fig. 3 Accuracy and precision for latency measurements

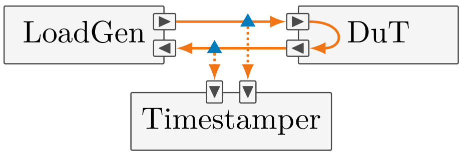

Fig. 4 Setup overview

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 medium propagation speed in optical fibers is 0.22 m/ns (based on 0.72 c [34]). The passive optical splitters do not introduce jitter, thus, do not impact the precision of our measurement; the impact on accuracy can be corrected if the fiber lengths are known. Using the propagation speed and the length of the fibers in a measurement setup would allow calculating the increased propagation delay. However, we did not perform this correction as the delay introduced by a few meters of additional fibers is still lower than the resolution of our timer. MoonGen supports the timestamping method of X550-based NICs through a userscript called MoonSniff. We determine the forwarding latency in three steps:

1. We use MoonSniff to record timestamped pcaps of the ingress and egress interface of a Device under Test (DuT).
2. We extract packet signatures from the pcaps and import them into a PostgreSQL database [48].
3. 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, such as an included packet counter, 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 packet transfer and loss rates, maximum and minimum latency values, latency percentiles, latency and jitter histograms, and worst-case latency time series.

This section has introduced the challenges for sub-microsecond latency measurements and a methodology to ensure the quality of these measurements. Based on these findings, we deduct the first recommendation, to be used to create a measurement setup optimized for sub-microsecond measurements:

Recommendation I: Measurement Setup Software timestamping on the measurement systems is subject to effects that may impact the quality of measurements. Hardware timestamping can help avoid these effects, ensuring high accuracy and precision. Both quality measures are essential for observing short and rare events, as minor deviations in the measurement may heavily impact the output. We recommend exclusively using hardware timestamping on affordable off-the-shelf hardware to minimize the measurement effort and cost while maximizing measurement quality.

6 Evaluation

This section introduces the measurement setup, our measurement methodology, and results.

6.1 Setup

The setup, shown in Fig. 4, is based on the presented measurement methodology. Our setup involves three nodes, the DuT hosting different applications, the LoadGen connected to the DuT via two 10G links, and the Timestamper that monitors both links passively via optical splitters. We kept hardware and software identical to our previous work [1–3], 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 was further equipped with three Intel NICs, based on the Intel 82599, X710, and E810 controllers, to investigate the impact of different NICs on latency. The DuT runs Debian bullseye (kernel v5.10) with the different kernels described in Sect. 3. We use KVM as hypervisor and DPDK version 21.11. We want to measure the packet loss and latency of applications with different complexity. The first investigated application is a basic L2 forwarder included in DPDK [28]. This basic packet processing application is investigated to provide an artificially simple example demonstrating the best-case performance. The second application is Suricata v7.0 [5], an example of a more complex, real-world packet processing application and its performance.

We test using constant bit-rate traffic with 64 B-sized packets. All measurements were repeated with packet rates between 10 kpkt/s and 250 kpkt/s. 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.

The experiments were conducted in our testbed using the pos framework utilizing an automated experiment workflow to ensure reproducible results [49].

6.2 Results

We try to determine the effects of specific system changes on latency. Therefore, we start our measurement with a simple forwarding application and gradually increase the complexity of our DuT.

6.2.1 Impact of the Linux Kernel

In this section, we want to determine the effect of the Linux kernel on latency. We investigate three different Debian Linux kernels: the vanilla kernel without any changes, the rt kernel provided via the Debian package repository, and a self-compiled kernel with the enabled `CONFIG_NO_HZ_FULL` flag. All three images are built using the identical kernel version 5.10.0-10, to keep the differences between the images minimal.

DPDK-l2fwd

Figure 5 presents the forwarding latency of a DPDK Layer 2 forwarder (DPDK-l2fwd) as a percentile distribution [50] at a packet rate of 10 kpkt/s. The plots for the higher packet rates are omitted due to their highly similar latency distribution. We did not observe lost packets, i.e., our system and the DPDK framework are powerful enough to handle the provided rates without overloading. Up to the 99.9th percentile, all kernels offer a stable latency of approx. 3.3 μs. For higher percentiles, the latency rises to 5.3 μs/5.5 μs for the vanilla/rt kernel. The latency of the no-hz kernel only rises to a value of 4.1 μs beyond the 99.99th percentile.

Figure 6 shows the 5000 worst-case latency events over the 60-second measurement time. All three measurements show a solid line at 3.3 μs, i.e., most latency events are on or below this line. A regular pattern above this line is visible for the vanilla and rt kernels. We identified OS interrupts, in this case, the local timer interrupt (loc), as the root cause for this behavior in previous work [2]. In the no-hz kernel, the interrupt can be disabled; therefore, the pattern disappears.

The pattern is the result of two clocked processes—OS interrupts and packet generation. We measure an increased delay on the DuT if the packet processing task is delayed due to an interrupt being triggered simultaneously. The observed pattern is an aliasing effect caused by undersampling, i.e., we can see a low-frequency signal that is not part of the original data. A more extensive description can be found in previous work [2].

In previous work [2, 3], we measured latencies for version 4.19 of the rt and vanilla kernels using the same scenario and hardware. There, we observed latencies of up to 13.6 μs. We attribute this reduction of more than 50 % to kernel optimizations of the interrupt handling. When comparing the results of Fig. 6 to our previous investigation of the no-hz kernel [1], we noticed a significant improvement. We successfully determined our interrupt monitoring tool as the source of

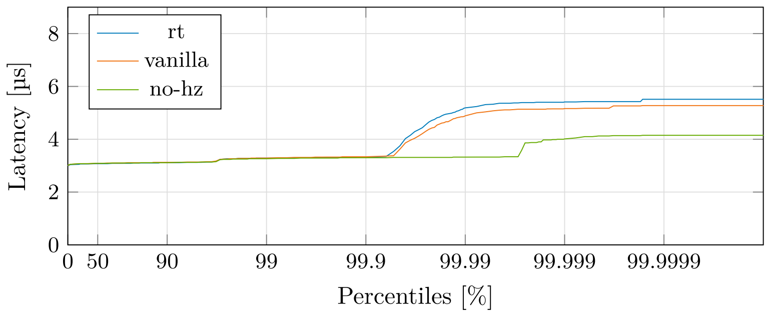

Fig. 5 Percentile distribution of the latency for DPDK-l2fwd using rt, vanilla, and no-hz Linux kernels at 10 kpkt/s

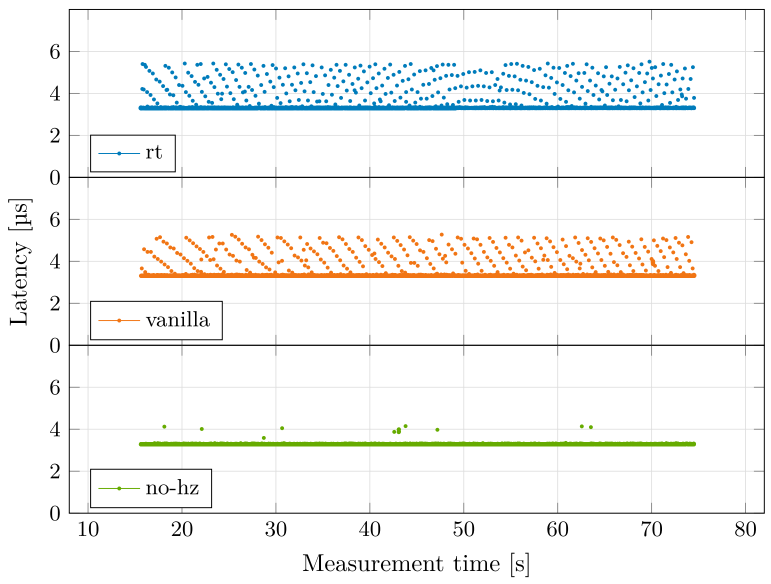

Fig. 6 5000 worst-case latency events for DPDK-l2fwd using rt, vanilla, and no-hz Linux kernels at 10 kpkt/s

a periodic 1-second latency increase. Without monitoring the interrupts during measurements, we could create a highly stable latency behavior.

Suricata-fwd

Suricata was chosen to measure the behavior of a real-world application. For this measurement scenario, we have disabled the ruleset in Suricata, turning the intrusion prevention system into a packet forwarding application. We use this measurement to determine the overhead of Suricata without the impact of rule application. Figure 7 shows the latency distribution of our measurement for the three investigated kernels.

For the rt kernel, we observe a higher latency than for the DPDK-l2fwd scenario, with a median latency of 3.8 μs. We notice a significant rise in latency beyond the 99.9th percentile to approx. 6.5 μs across all measured packet rates. For the highest packet rate of 250 kpkt/s, we measure an additional latency increase to 12.2 μs not present in lower rates.

For the vanilla and no-hz kernels, we observe an even higher rise in latency. To visualize the sharp tail-latency increase without concealing lower percentiles, we switched to a log scale for both kernels in Fig. 7. Up to roughly the 99.999th percentile, latencies are similar to the rt kernel. Beyond this point, the latencies of the no-hz and vanilla kernels rise up to 635 μs, a significant difference compared to the rt kernel.

Figure 8 shows two selected examples that visualize the worst-case forwarding latencies over the experiment. The first example shows the forwarding latency of the rt kernel at a rate of 250 kpkt/s. There, two latency spikes above 10 μs

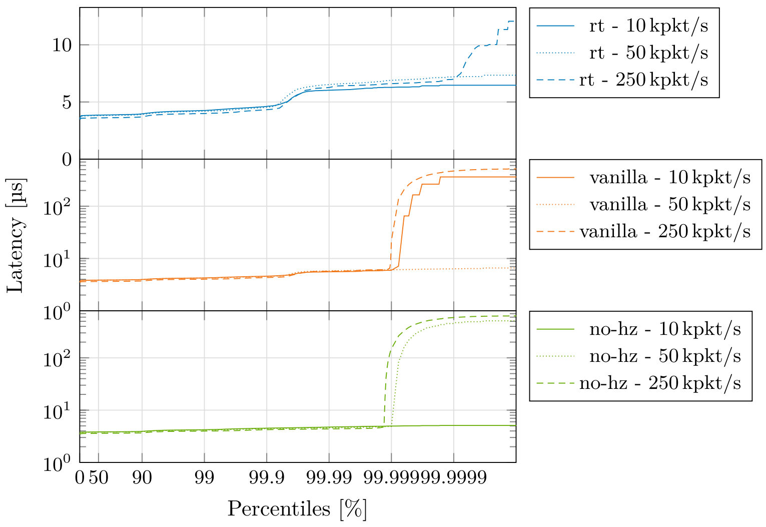

Fig. 7 Percentile distribution of the forwarding latency on Suricata (no ruleset)

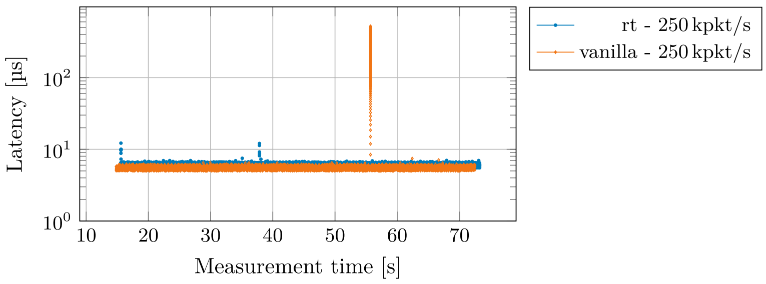

Fig. 8 5000 worst-case latency events for Suricata (no ruleset) using different Linux kernels at 250 kpkt/s

are visible. The other plot shows the latency of the vanilla kernel at a rate of 250 kpkt/s. There, a single spike is the reason for the latency increase.

Similar to our previous studies [2, 3], we assumed interrupts to be the root cause for the observed latency spikes. To verify our assumption, we used the Linux interrupt counters listed in `/proc/interrupts`. This list contains a counter for the different kinds of interrupts triggered since the start of the system. When running our experiment with different measurement times, we saw the latency spikes when the TLB shootdown counter was incremented. We further investigated the differences between the low impact of the TLB shootdowns on the rt kernel and the more significant impact on the vanilla and no-hz kernels. We observed that TLB shootdowns happen more rarely on the rt kernel; we attribute this lower number to the changes introduced by the rt patches.

The TLB shootdowns are mentioned by Rigtorp [8] 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 an interrupt and causes the observed latency spikes. Rigtorp [8] 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.

We attribute the occurrence of latency spikes to the memory management of Suricata and did not find a configuration to avoid them for the no-hz and vanilla kernels. We did not observe severe latency spikes for the rt kernel. Figure 7 includes examples of measurements without latency spikes. We attribute this lack of increased latency to our 1-minute measurement time. In other measurements, spikes were observed for these rates.

Another finding of our investigation is the similar behavior of vanilla and no-hz kernels, contradicting our previous measurements with the DPDK-l2fwd. The difference between both scenarios is the architecture of the investigated packet processing application. For the DPDK-l2fwd, we could dedicate a forwarding thread exclusively to one core. Without any other thread running on the same cores, the no-hz kernel disables almost all interrupts for this core. Suricata follows a more complex multi-threaded architecture involving management and worker threads. In combination with DPDK, we did not find a configuration that would allow us to create a dedicated worker core that was not interrupted by other threads. Without exclusive core usage, the no-hz kernel does not disable any interrupts on the packet processing core, acting the same way a vanilla kernel would—a behavior confirmed by our measurements. The rt kernel, in contrast, seems to handle the interrupt processing differently. For the rt kernel (cf. Fig. 7), we measured a higher jitter and a slightly increased latency. This behavior suggests that the rt kernel handles processing tasks during interrupts differently, thereby avoiding large spikes.

Suricata-Filter

Figure 9 shows the forwarding latency of Suricata applying its default ruleset. The increased complexity of the processing task raises latency. We measured a median latency close to 10 μs for all kernels and packet rates between 10 kpkt/s and 150 kpkt/s. For the no-hz and vanilla kernels, we noticed a steep increase in latency to approx. 600 μs starting at the 99.99th percentile. The cause for this increase are the same interrupts as for the previous scenario. However, these costly interrupts occur at a higher rate, lowering the percentile for the latency increase. For the rt kernel, we did not observe this behavior, leading to a more stable and overall lower worst-case latency.

For all three kernels, we measured an overload scenario, causing packet loss and an increase in latency up to 2 ms. This measurement shows that Suricata becomes overloaded at the same forwarding rate of approx. 200 kpkt/s regardless of the used kernel.

Recommendation II: Avoid Overload Our measurements show that overload must be avoided on packet processing nodes to keep latency reasonably low.

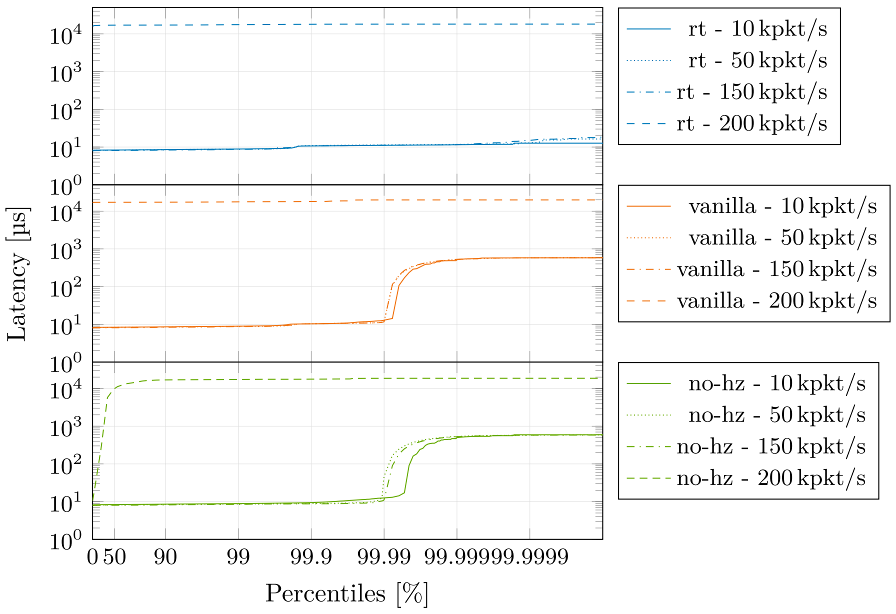

Fig. 9 Percentile distribution of the forwarding latency on Suricata (with ruleset)

Overload leads to packet loss and causes latencies in the ms-range. DPDK offers a low-overhead framework that can handle significant packet rates without overloading; complex applications such as the Suricata IPS can lead to overload scenarios if the processing capacity is exhausted. Therefore, a packet processing system should be designed with enough spare capacity to handle the expected packet processing load.

Recommendation III: Choice of Kernel Our measurements have shown that the Linux kernel can have a significant impact on the latency of packet processing applications. However, there is no kernel that offers consistently better performance across all investigated scenarios. The architecture of the packet processing application is a decisive factor for kernel selection. The no-hz kernel can disable almost all interrupts if an application thread or process can be hosted on a single core without the need to share it with other threads. In such a scenario, the no-hz kernel offers the best latency. For more complex applications, sharing cores among threads, no-hz does not allow disabling interrupts offering no benefit over the vanilla Linux kernel. However, our measurements showed that an rt kernel could optimize latency in such a scenario. The tail latency was lower, due to the lower impact of TLB shootdowns, causing high latency spikes for no-hz and vanilla kernels. Therefore, we recommend the rt kernel for complex applications like Suricata and the no-hz kernel for simple applications like the DPDK-l2fwd or Snort 3 [1]. We further noticed that the worst-case latencies improved when comparing a Linux kernel version 5.10 to version 4.19. Thus, we recommend checking different kernel versions, when optimizing for latency.

6.2.2 Impact of the NIC

This section investigates the impact of the NIC on the forwarding latency. To measure the isolated effects of the NIC, we select the DPDK-l2fwd on the no-hz kernel to minimize the impact of OS and application. We compare four different Intel NICs, the SoC-integrated X552 [47], the dual-port X520-DA2 [45], the quad-port X710-DA4 [46], and the E810-XXVDA4 [51]. The E810-based NIC supports 25 Gbit/s Ethernet but was used with a 10 Gbit/s link to ensure comparability with the other NICs. The rest of the measurement setup remained unchanged.

Figure 10 visualizes the forwarding latency for the different NICs at different packet rates as a percentile distribution. The oldest NIC in our comparison is

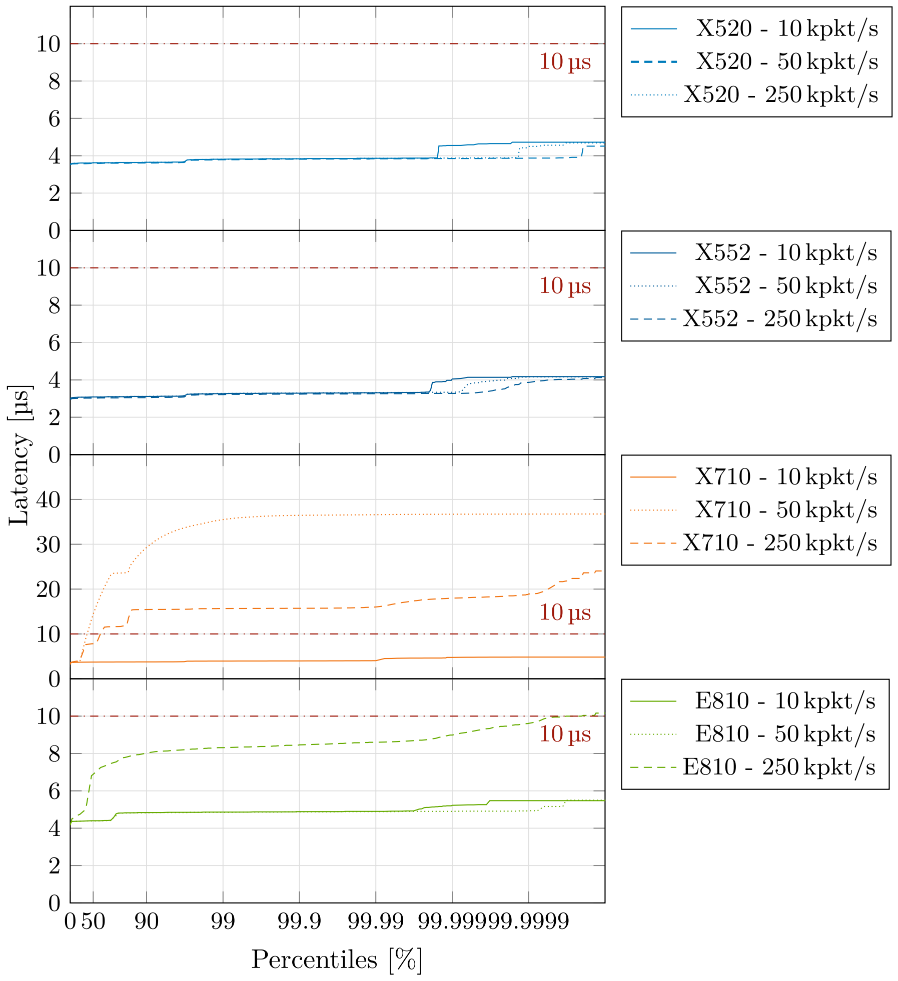

Fig. 10 Percentile distribution of the latency for different Intel NICs. The 10-μs bar is highlighted with a dash-dotted line for easier comparison

the X520, which offers a stable and highly similar latency across the investigated rates. A visible increase in latency starts around the 99.999th percentile; the latency increases from approx. 3.9 μs to 4.5 μs. The X552 NIC offers the same stability as the X520, and the latency distribution follows roughly the same shape. The absolute values are shifted by 0.5 μs, i.e., the X552 is faster than the X520.

For the X710 NIC, we see a stable latency behavior for a packet rate of 10 kpkt/s that stays below 5 μs. However, starting at a rate of 50 kpkt/s, we can see significant changes in the latency behavior where latency rises from a median of approx. 14 μs to 37 μs for the 99th percentile. A closer investigation of the egress traffic shows that the NIC begins sending bursts of packets. If we increase the rate, the latency sinks again to approx. 20 μs for the 99.9999th percentile; however, the bursty behavior remains. Thus, the latency is not as stable as it was for a rate of 10 kpkt/s. For the E810 NIC, we see stable latency behavior for rates of 10 kpkt/s and 50 kpkt/s. Higher rates are again subject to bursty behavior and latency increase.

We attribute the differences in stability to the increased complexity of the NICs and their controller architectures over time. An indicator for this increased feature set is the length of the respective data sheet that grew from approx. 1000 pages for the X520 [45] to over 2700 for the E810 [51]. Also, the size of the firmware present on the NICs grew over time. The X710 and the E810 possess firmware of several megabytes; the E810 additionally features a programmable parser that loads additional software during runtime. Where older NICs, such as the X520, possess a fixed processing path, newer NICs feature a higher degree of configurability for the packet processing path. This increased flexibility makes the packet processing path and latency on the NIC less predictable for newer NIC generations. We attribute the increased latency to the changes in NIC architecture.

For our tests, we relied on the default configurations provided by DPDK for PCIe and NIC drivers. Further optimizations were not considered. Please note that X710 and E810-based network controllers were designed for 40 Gbit/s and 100 Gbit/s bandwidths. In our scenario, we only investigated 10 Gbit/s to ensure compatibility across the different NICs and our measurement platform. The latency behavior may be different when operating at higher link bandwidths.

Recommendation IV: Choice of NIC We have shown that the choice of NIC can significantly impact latency and jitter. Older, low-complexity NICs, such as the Intel X520 or X552, offer less configurability leading to low, stable latency. Therefore, the impact of the NIC is low compared to other effects described in this paper. However, the impact changes when considering more complex NICs, such as the X710 and E810. Newer NIC generations offer a higher degree of flexibility, which in turn make predicting latency more challenging. Thus, we recommend carefully investigating the latency in application-specific scenarios, especially when using more recent NIC architectures.

6.2.3 Impact of Virtualization

We want to investigate the impact of virtualization on packet processing applications. Therefore, we measure the performance of a virtualized DPDK-l2fwd application. The application is run on a VM pinned to P-cores 1, 2, and 3 of our

DuT. The OSes on the DuT and the DuT VM use the same, previously described boot parameters and images. For the impact of virtualization, we only analyze the impact using the DPDK-l2fwd and the Intel X552 NIC. This simple setup allows measuring the impact of virtualization in isolation without potential effects caused by a complex software architecture. Further, the used CPUs could not run Suricata in a meaningful way. Our setup requires at least two separate p-cores for the operating systems—on the host and the VM. Suricata requires at least three cores to run with minimal core sharing among threads. For obvious reasons, this five-core requirement cannot be met on a quad-core CPU.

Figure 11 shows the effects of virtualization on packet processing applications. The measurement shows a stable latency of approx. 4 μs up to the 99th percentile. Beyond this point, a significant increase begins, and the different Linux kernel versions begin to differ. As shown in previous measurements (cf. Fig. 5), the rt Linux kernel latencies are higher than the latencies of the no-hz and vanilla Linux kernel. We measure tail latencies for no-hz and vanilla kernel at approx. 11 μs and 10 μs.

In previous work [1], we measured the latencies for version 4.19 of the no-hz Linux kernel using the same scenario and hardware. We observed a latency of up to 4.1 μs with virtualization on a no-hz Linux kernel. This shows that the impact of virtualization is significantly higher on newer kernel and software measured in this experiment with an increase of approx. 6 μs.

Figure 12 shows the worst-case latency behavior over the 60-second measurement period for the virtualized DPDK-l2fwd example on each of the measured Linux kernel versions. Further analysis of the worst-case events shows a familiar behavior; a solid horizontal line with most latency events either on or below this line, above this line, a regular pattern of events. In general, we can see that virtualization leads to an increased number of events in the area above the horizontal line. The number of events is increased because of the higher number of interrupts, caused by two running operating systems, the VM OS and the host OS. In addition, the interrupt processing time on the virtualized kernel is increased compared to non-virtualized setups. This leads to a visible impact of virtualization on the performance of low-latency packet processing systems.

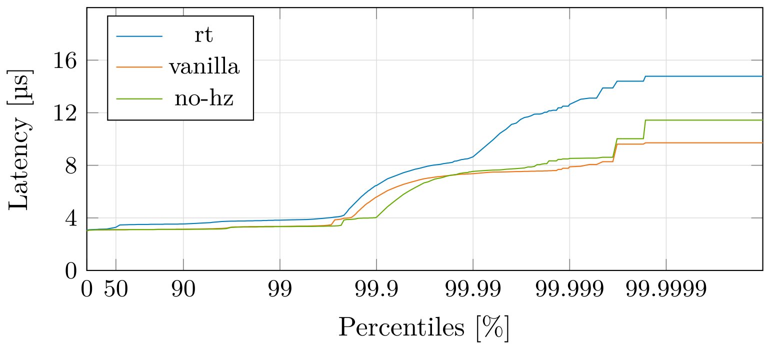

Fig. 11 Percentile distribution of the latency for a virtualized DPDK-l2fwd at 10 kpkt/s

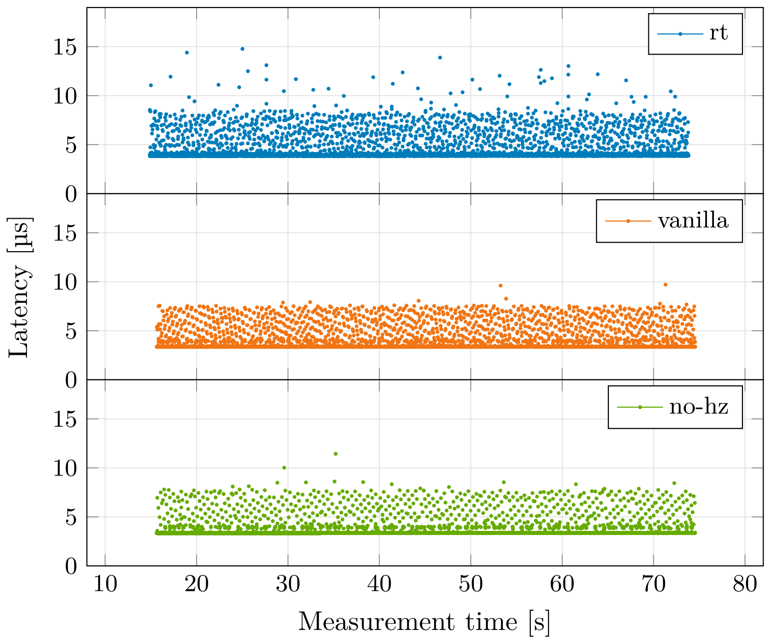

Fig. 12 5000 worst-case latency events for virtualized DPDK-l2fwd using rt, vanilla, and no-hz Linux kernels at 10 kpkt/s

Figure 12 shows almost identical behavior between the no-hz and the vanilla Linux Kernels. We assume the interrupts could not be disabled on the no-hz kernel in this environment. To verify our assumption that interrupts increase the latency, we performed a second measurement run, where we captured the interrupt counters in Linux during this specific measurement run. We found a correlation between the local timer interrupts and the increased latency events.

Recommendation V: Choice of Virtualization We have shown that choosing between virtualized and non-virtualized systems for packet processing impacts latency and jitter. At the same time, we have shown that its price, i.e., the impact of virtualization on latency, is limited to tail latencies. When using virtualization, the choice of Linux kernel matters. The virtualized no-hz kernel did not offer benefits over a virtualized vanilla kernel; the tail latency on the virtualized rt kernel is even higher. All three kernels performed worse from a tail-latency perspective than their non-virtualized counterparts. Due to the negligible impact on latency below the 99th percentile, latency considerations should not prevent the virtualization of applications in general. However, if tail latencies are the primary optimization goal, a bare-metal system can offer benefits.

7 Limitations

The measurements presented in this paper have shown that software packet processing systems can be tuned to provide sub-microsecond latency with low jitter. However, the low latency and jitter come at a price. The presented system configurations disable the CPU energy-saving mechanisms. In addition, DPDK actively polls the NIC, fully loading the allocated CPU cores. We measured the energy consumption in a previous paper [2]; the entire server consumed 31 W in an idle state and 47 W when executing a packet processing application. The majority of this 48-percent increase in energy consumption is caused by the CPU. While numbers are highly hardware specific, the increase and the CPU being its main factor can be transferred to other systems.

8 Experiment Data and Reproducibility

A major goal of our research is the creation of reproducible experiments [49]. Therefore, we created a website [52] that explains each measurement presented in the paper. The experiment artifacts are available in a GitHub repository [53]. The experiment artifacts include the experiment scripts, measurement data, plotting scripts, and plots. The investigated applications are open source on GitHub [5, 28]

9 Conclusion

Our measurements show that the latency limbo has much in common with the actual dance. A set-up latency bar can be easily touched or exceeded with seemingly minor alterations to the investigated software stack. Nevertheless, we show that, given the proper techniques, the latency bar remains intact. Therefore, we established our recommendations acting as guiding rails to create reliable, low-latency packet processing systems:

– Software-based timestamping methods are subjected to the same effects we investigated in our studies. A measured latency may be caused by the measuring *or* the measured system causing ambiguous measurement data. To avoid this problem entirely, we stress the need for hardware-based timestamping, to provide high accuracy and precision for measurements.
– Overloading a system leads to inevitable packet loss and filled buffers increasing latency; therefore, overload must be avoided. Our measurements have shown that DPDK and Suricata provide a throughput of several 100 000 packets per second on a single CPU core. However, when inserting a complex computation like the IPS rule application into the processing path, the limited

CPU resources may cause an overload. Providing enough CPU resources or limiting the number of packets are possible solutions to this problem.
- Our investigation of the different flavors of the Linux kernel has no clear winner. If a network application process can be hosted on a CPU core exclusively, the no-hz kernel provides stable and low latencies. The rt kernel offers superior performance if a core is shared between processes. However, we observed situations where the vanilla kernel performs best, if cores are shared, and TLB shootdowns did not occur. For the choice of kernel, there is no one-size-fits-all solution; it requires measurements or an in-depth investigation of the application architecture to find the best fitting kernel.
- In our comparison, we determined the Intel X552 as the NIC with the lowest and most stable latency. More modern cards were not only slower but also introduced jitter. If the described NICs are not an option, we recommend testing the designated NIC architecture before integration to avoid surprising effects on latency and jitter.
- For virtualization, we measured a noticeable impact on tail latency. The median or lower-percentile latencies were only slightly increased, demonstrating that virtualization is highly efficient and introduces little overhead to the packet processing path. When optimizing tail latencies, virtualization should be avoided, as we noticed irregular spikes for all our measurements.

Acknowledgements This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreements Nos. 101008468 and 101079774. Additionally, we received funding by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria. This work is partially funded by Germany Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK001K) and 6G-ANNA (16KISK107).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Gallenmüller, S., Wiedner, F., Naab, J., Carle, G.: Ducked tails: trimming the tail latency of(f) packet processing systems. In: 17th international conference on network and service management, CNSM Izmir, Turkey, October 25–29, IEEE, 2021. https://doi.org/10.23919/CNSM52442.2021.9615532
2. Gallenmüller, S., Naab, J., Adam, I., Carle, G.: 5G QoS: Impact of Security Functions on Latency. In: NOMS, IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20–24. IEEE **2020**, 1–9 (2020). https://doi.org/10.1109/NOMS47738.2020.9110422

3. Gallenmüller, S., Naab, J., Adam, I., Carle, G.: 5G URLLC: a case study on low-latency intrusion prevention. IEEE Commun. Mag. **58**(10), 35–41 (2020). https://doi.org/10.1109/MCOM.001.2000467
4. NGMN Alliance, 5G E2E Technology to Support Verticals URLLC Requirements, 2019
5. Suricata repository: https://github.com/gallenmu/suricata/tree/dpdk-21.11. Accessed 25 Nov 2022
6. AMD, Performance Tuning Guidelines for Low Latency Response on AMD EPYC-Based Servers Application Note. http://developer.amd.com/wp-content/ resources/56263-Performance-Tuning-Guidelines-PUB.pdf. Accessed 25 Nov 2022, Jun 2018
7. Mario, J., Eder, J.: Low Latency Performance Tuning for Red Hat Enterprise Linux 7. https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf. Accessed 25 Nov 2022
8. Rigtorp, E.: Low latency tuning guide. https://rigtorp.se/low-latency-guide/. Accessed 25 Nov 2022, Mar 2020
9. Beierl, M.: Nfv-kvm-tuning. https://wiki.opnfv.org/pages/viewpage.action?pageId=2926179. Accessed 25 Nov 2022
10. Li, J., Sharma, N.K., Ports, D.R.K., Gribble, S.D.: Tales of the tail: hardware, OS, and application-level sources of tail latency. In: Lazowska, E., Terry, D., Arpaci-Dusseau, R.H., Gehrke, J., Eds. Proceedings of the ACM symposium on cloud computing, Seattle, WA, USA, November 3–5, 2014. ACM, 2014, 9:1–9:14. https://doi.org/10.1145/2670979.2670988
11. Popescu, D., Zilberman, N., Moore, A.: Characterizing the impact of network latency on cloud-based applications' performance, 2017
12. Barroso, L.A., Marty, M., Patterson, D.A., Ranganathan, P.: Attack of the killer microseconds. Commun. ACM **60**(4), 48–54 (2017). https://doi.org/10.1145/3015146
13. Zhang, Y., Laurenzano, M.A., Mars, J., Tang, L.: SMiTe: precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In: 47th annual IEEE/ACM international symposium on microarchitecture, MICRO: Cambridge, UK, December 13–17, 2014. IEEE Computer Society **2014**, 406–418 (2014). https://doi.org/10.1109/MICRO.2014.53
14. Schöne, R., Molka, D., Werner, M.: Wake-up latencies for processor idle states on current x86 processors. Comput. Sci. R &D **30**(2), 219–227 (2015). https://doi.org/10.1007/s00450-014-0270-z
15. Herdrich, A., Verplanke, E., Autee, P., Illikkal, R., Gianos, C., Singhal, R., Iyer, R.: Cache QoS: from concept to reality in the Intel Xeon E5-2600 v3 Product Family. In: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12–16, 2016, 2016, pp. 657–668. https://doi.org/10.1109/HPCA.2016.7446102
16. Aleksinski, M. et al.: intel-cmt-cat. https://github.com/intel/intel-cmt-cat. Accessed 25 Nov 2022
17. Emmerich, P., Raumer, D., Gallenmüller, S., Wohlfart, F., Carle, G.: Throughput and latency of virtual switching with open vswitch: a quantitative analysis. J. Netw. Syst. Manag. **26**(2), 314–338 (2018). https://doi.org/10.1007/s10922-017-9417-0
18. Lettieri, G., Maffione, V., Rizzo, L., "A Survey of Fast Packet I, O Technologies for Network Function Virtualization," in High Performance Computing - ISC High Performance,: International Workshops, Frankfurt, Germany, June 18–22, 2017. Revised Selected Papers **2017**, 579–590 (2017). https://doi.org/10.1007/978-3-319-67630-2_40
19. Xu, X., Davda, B.: SRVM: hypervisor support for live migration with passthrough SR-IOV network devices. In: Proceedings of the 12th ACM SIGPLAN/SIGOPS international conference on virtual execution environments, Atlanta, GA, USA, April 2–3, 2016, 2016, pp. 65–77. https://doi.org/10.1145/2892242.2892256.
20. Xiang, Z., Gabriel, F., Urbano, E., Nguyen, G.T., Reisslein, M., Fitzek, F.H.P.: Reducing Latency in Virtual Machines: Enabling Tactile Internet for Human-Machine Co-Working. IEEE J. Select. Areas Commun. **37**(5), 1098–1116 (2019). https://doi.org/10.1109/JSAC.2019.2906788
21. Zilberman, N., Grosvenor, M.P., Popescu, D.A., Bojan, N.M., Antichi, G., Wójcik, M., Moore, A.W.: Where has my time gone? In: Passive and active measurement - 18th international conference, PAM 2017, Sydney, NSW, Australia, March 30–31, 2017, Proceedings, 2017, pp. 201–214. https://doi.org/10.1007/978-3-319-54328-4_15.
22. Ramsauer, R., Kiszka, J., Lohmann, D., Mauerer, W.: Look mum, no VM exits! (almost), CoRR, vol. abs/1705.06932, 2017. arXiv: 1705. 06932. [Online]. http://arxiv.org/abs/1705.06932
23. Kaiser, R., Wagner, S.: Evolution of the PikeOS Microkernel. In: First international workshop on microkernels for embedded systems, vol. 50, Jan 2007
24. Reghenzani, F., Massari, G., Fornaciari, W.: The Real-Time Linux Kernel: a survey on PREEMPT_RT. ACM Comput. Surv. **52**(1), 1–36 (2019). https://doi.org/10.1145/3297714

25. n.a., NO_{HZ}: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO%7B%5C_%7DHZ.txt. Accessed 25 Nov 2022
26. Emmerich, P., Raumer, D., Beifuß, A., Erlacher, L., Wohlfart, F., Runge, T.M., Gallenmüller, S., Carle, G.: Optimizing latency and CPU load in packet processing systems. In: Proceedings of the international symposium on performance evaluation of computer and telecommunication systems, Chicago, IL, USA, July 26–29, 2015, IEEE, 2015, 6:1–6:8. https://doi.org/10.1109/SPECTS.2015.7285275.
27. Salim, J.H.: When napi comes to town. In: Linux 2005 conference 2005
28. DPDK repository. https://github.com/gallenmu/dpdk-1/tree/21.11-low-latency. Accessed 25 Nov 2022
29. Høiland-Jørgensen, T., Brouer, J.D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., Miller, D.: The eXpress Data Path: fast programmable packet processing in the operating system kernel. In: Proceedings of the 14th international Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018, 2018, pp. 54–66. https://doi.org/10.1145/3281411.3281443.
30. Deri, L.: nCap: wire-speed packet capture and transmission. In: Third IEEE/IFIP workshop on end-to-end monitoring techniques and services, E2EMON: 15th May 2005. Nice, France, IEEE Computer Society **2005**, 47–55 (2005). https://doi.org/10.1109/E2EMON.2005.1564468
31. Rizzo, L.: netmap: a novel framework for fast packet I/O. In: Heiser, G., Hsieh, W.C., Eds., 2012 USENIX annual technical conference, Boston, MA, USA, June 13–15, 2012. USENIX Association, 2012, pp. 101– 112. https://www.usenix.org/conference/usenixsecurity12/ technical-sessions/presentation/rizzo
32. Gallenmüller, S., Emmerich, P., Wohlfart, F., Raumer, D., Carle, G.: Comparison of frameworks for high-performance packet IO. In: Proceedings of the eleventh ACM/IEEE symposium on architectures for networking and communications systems, ANCS 2015, Oakland, CA, USA, May 7–8, 2015, IEEE Computer Society, 2015, pp. 29–38. https://doi.org/10.1109/ANCS.2015.7110118
33. Emmerich, P.,Gallenmüller, S.,Raumer, D., Wohlfart, F., Carle, G.: MoonGen: a scriptable high-speed packet generator. In: Cho, K., Fukuda, K., Pai, B.S., Spring, N., Eds., Proceedings of the 2015 ACM internet measurement conference, IMC 2015, Tokyo, Japan, October 28–30, 2015. ACM, 2015, pp. 275–287. https://doi.org/10.1145/2815675.2815692
34. Emmerich, P., Gallenmüller, S., Antichi, G., Moore, A.W., Carle, G.: Mind the gap - a comparison of software packet generators. In: ACM/IEEE symposium on architectures for networking and communications systems, ANCS, Beijing, China, May 18–19, IEEE Computer Society, 2017, pp. 191–203. https://doi.org/10.1109/ANCS.2017.32
35. Antichi, G., Shahbaz, M., Geng, Y., Zilberman, N., Covington, G.A., Bruyere, M., McKeown, N., Feamster, N., Felderman, B., Blott, M., Moore, A.W., Owezarski, P.: OSNT: open source network tester. IEEE Netw. **28**(5), 6–12 (2014). https://doi.org/10.1109/MNET.2014.6915433
36. Silicom, Datasheet PE310G4TSF4I71. https://www.silicom-usa.com/wp-content/uploads/ 2016/08/PE310G4TSF4I71-Programmable-Application-Acceleration- 10G.pdf. Accessed 25 Nov 2022
37. Primorac, M., Bugnion, E., Argyraki, K.J.: How to measure the killer microsecond. Comput. Commun. Rev. **47**(5), 61–66 (2017). https://doi.org/10.1145/3155055.3155065
38. Zhang, X., Li, C., Zheng, W.: Intrusion prevention system design. In: 2004 international conference on computer and information technology (CIT 2004), 14–16 September 2004, Wuhan, China, IEEE Computer Society, 2004, pp. 386–390. https://doi.org/10.1109/CIT.2004.1357226
39. Albin, E., Rowe, N.C.: Realistic Experimental Comparison of the Suricata and Snort Intrusion-Detection Systems. In: 26th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2012, Fukuoka, Japan, March 26–29, 2012, L. Barolli, T. Enokido, F. Xhafa, and M. Takizawa, Eds., IEEE Computer Society, 2012, pp. 122–127. https://doi.org/10.1109/WAINA.2012.29
40. Julien V., et al.: Suricata User Guide. https://suricata.readthedocs.io/en/latest/. Accessed 25 Nov 2022
41. Julien, V., Simis, L.: dpdk: initial support with workers runmode. https://github.com/OISF/suricata/commit/a7faed12450b85e9108868861723741fc93716fa. Accessed 25 Nov 2022
42. Gupta, A., Sharma, L.S.: Performance Evaluation of Snort and Suricata Intrusion Detection Systems on Ubuntu Server. In: Proceedings of ICRIC 2019, Springer, 2020, pp. 811–821
43. Intel 64 and IA-32 Architectures Software Developer's Manual, 325462- 075US, Intel, Jun 2021
44. ISO 5725-1: 1994: Accuracy (Trueness and Precision) of Measurement Methods and Results-Part 1: General Principles and Definitions. International Organization for Standardization, 1994

45. Intel 82599 10 GbE Controller - Datasheet, 331520-005, Rev. 3.4, Intel, Nov 2019
46. Intel Ethernet Controller X710/XXV710/XL710 Datasheet, 332464-020, Rev. 3.65, Intel, Aug 2019
47. Intel Ethernet Controller X550 - Datasheet, 333369-005, Rev. 2.3, Intel, Nov 2018
48. PostgreSQL Global Development Group, PostgreSQL, Jul 2021. https://www.postgresql.org/
49. Gallenmüller, S., Scholz, D., Stubbe, H., Carle, G.: The pos Framework: A Methodology and Toolchain for Reproducible Network Experiments. In: CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7–10, ACM, 2021, pp. 259–266. https://doi.org/10.1145/3485983.3494841
50. Tene, G.: HdrHistogram: A High Dynamic Range Histogram. http://hdrhistogram.org/. Accessed 25 Nov 2022
51. Intel Ethernet Controller E810 Datasheet, 613875-005, Rev. 2.3, Intel, Sep 2021
52. Gallenmüller, S., Wiedner, F., Naab, J., Carle, G.: latency-limbo repository. https://gallenmu.github.io/latency-limbo. Accessed 25 Nov 2022
53. Gallenmüller, S., Wiedner, F., Naab, J., Carle, G.: latency-limbo repository. https://github.com/gallenmu/latency-limbo. Accessed 25 Nov 2022

Sebastian Gallenmüller received his Ph.D. in 2021 from the Technical University of Munich. He currently works as a PostDoc at the Chair of Network Architectures and Services led by Prof. Georg Carle at the Technical University of Munich. His main research interests are programmable packet processing systems and testbeds for network experiments with a focus on performance analysis and modeling of packet processing systems.

Florian Wiedner finished his Master of Science in Informatics in 2020 at the Technical University of Munich where he currently works as a Ph.D. student at the Chair of Network Architectures and Services. His research focuses on low-latency measurements and low-latency networking on partly virtualized systems as well as scalability.

Johannes Naab completed his Master of Science in Informatics in 2014 at the Technical University of Munich. In the same year, he started as a Ph.D. student at the Chair of Network Architectures and Services. His research focuses primarily on the development of large-scale cloud architectures and in his free time he performs Internet-wide measurements.

Georg Carle is Professor at the Technical University of Munich, holding the Chair of Network Architectures and Services. He studied at University of Stuttgart, Brunel University, London, and Ecole Nationale Superieure des Telecommunications, Paris. He did his Ph.D. in Computer Science at University of Karlsruhe, and worked as postdoctoral scientist at Institut Eurecom, Sophia Antipolis, France, at the Fraunhofer Institute for Open Communication Systems, Berlin, and as professor at the University of Tübingen.

Authors and Affiliations

Sebastian Gallenmüller[1] · **Florian Wiedner**[1] · **Johannes Naab**[1] · **Georg Carle**[1]

Florian Wiedner
wiedner@net.in.tum.de

Johannes Naab
naab@net.in.tum.de

Georg Carle
carle@net.in.tum.de

[1] Technical University of Munich, TUM School of Computation, Information and Technology, Boltzmannstr. 3, 85748 Garching Near Munich, Germany