5G URLLC: A Case Study on Low-Latency Intrusion Prevention

Sebastian Gallenmüller*, Johannes Naab*, Iris Adam†, Georg Carle*
*Technical University of Munich, †Nokia Bell Labs
*{gallenmu, naab, carle}@net.in.tum.de, †iris.adam@nokia-bell-labs.com

***Abstract*—5G introduces different communication types to support novel applications, e.g., industrial control, depending on ultra-reliable, low-latency communication (URLLC) to provide their service. To realize this type of communication, network operators offer virtual end-to-end networks, called network slices, to their customers and Service Level Agreements (SLA) of the operators promise a certain quality of service for the provided network slices. The main challenge is the creation of network slices that adhere to these strict requirements despite virtualized resources shared across different network slices.**

In this article, we analyze the latency performance of typical virtualized network functions. Based on these results, we derive guidelines to lower latency and propose a system architecture for hosting low-latency network functions. We measure the latency performance of a security network function, an intrusion prevention system based on Snort 3, and demonstrate that URLLC-compliant latency performance is achievable. Our entire architecture relies on off-the-shelf hardware and widely adopted software components making our findings highly applicable to situations where low latency is crucial.

All artifacts used in this article, the investigated software, the pcap traces, and the experiments scripts, are publicly available at https://gallenmu.github.io/low-latency/.

***Index Terms*—5G, network slicing, URLLC, NFV, IPS, QoS, latency, DPDK**

This article is the extension of a paper originally presented at NOMS 2020 [1]. The original paper includes an analysis of batched packet processing and a model to calculate the throughput of a system to avoid overload.

I. Introduction

One of the main features of 5G is the definition of different communication types tailored to the specific requirements of applications such as IoT, self-driving cars, or industrial control systems. Three types are available: the *enhanced mobile broadband* (eMMB) service, which is comparable to current LTE networks but offers higher bandwidth, the *massive machine type communication* (eMTC), that supports cells with a large number of clients required for IoT deployments, and the *ultra-reliable low-latency communication* (URLLC), which is optimized for critical processes that cannot tolerate packet loss or high latency. Network operators may provide URLLC services via a virtual network to their customers, so-called network slices. The network slices can rely on virtualized resources shared across different slices and customers for cost efficiency. Resource sharing presents a challenge, especially to URLLC, regarding the requirements for its quality of service (QoS).

The name URLLC already reflects its two defining features of service quality: *ultra reliability* and *low latency*. Fixed values for both requirements do not exist but depend on specific use cases. A current draft [2] for the upcoming 5G release 17 stage 1 (scheduled for end of 2020) lists the requirements for various use cases as follows. In the most demanding scenarios, URLLC requests a reliability of 99.99999%, i.e., the percentage of packets successfully transmitted for a given system entity. For latency, the most challenging use cases require a maximum delay below $0.5\,\mathrm{ms}$. This delay is measured one-way, end-to-end in the radio access network, e.g., between the user equipment (UE) and a network function (NF) hosted on the edge of a mobile network.

This article focuses on a system architecture to host NFs for URLLC, specifically on the following three goals: *(i)* we demonstrate the issues that prevent a URLLC-compliant communication on current architectures and the challenges that arise from measuring and communicating the latency performance accurately; *(ii)* we analyze the root causes for poor latency performance and provide guidelines to avoid them; *(iii)* we perform a series of measurements on an NF, based on Snort 3, an intrusion prevention system (IPS), that demonstrates the latency performance of a latency-optimized architecture for NFs.

This article is organized as follows: Section II presents the issues of typical NFs concerning their latency behavior before introducing the underlying system architecture in Section III. We present our architecture for low-latency NFs in Section IV and examine its behavior in Section V through a series of measurements. Sections VI and VII discuss the limitations and potential disadvantages of our approach and whether containers or VMs are the better choice to deploy our system architecture. Section VIII contains a description on how to reproduce the experiments presented in this article. The article concludes in Section IX.

II. Motivation

As an initial measurement, we investigate the service quality of a typical security NF hosted in such an environment. We use a Linux server running Debian buster that uses KVM to virtualize NFs of different network slices. The NF runs Snort as an in-line IPS. We test with a moderate

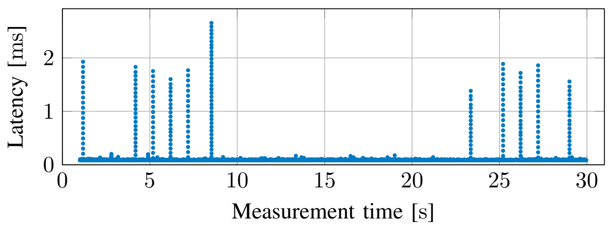


Fig. 1: Snort 3 forwarder worst-case latencies

packet rate of $10\,\mathrm{kpackets/s}$ to avoid any packet drops of the NF due to overload.

Figure 1 displays the worst-case latencies that occurred during our 30-second measurement period. This plot leads to two important conclusions. First, with the majority of latency measurements below $0.5\,\mathrm{ms}$, we need to direct our attention to the latency peaks. Second, though happening rarely, the peaks in latency are highly critical as they exceed the latency budget significantly, in our case, by more than 200%. The result of our measurement indicates that the 99.99th percentile of this measurement already violates the delay requirement.

Even this simplified example demonstrates that this NF, based on a typical system configuration, fails to provide URLLC-compliant QoS. In the following, we analyze the reasons for this system behavior and present guidelines for a system design that provides the necessary reliability and latency.

III. Background and Related Work

This section investigates how the latency performance of a typical off-the-shelf system can be improved. Extensive guidelines exist to tune systems for low latency [3]. We investigate the main system components, the network interface card (NIC), the processor (CPU), and the operating system (OS):

a) NIC: One 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 and Linux employ moderation techniques to minimize the number of interrupts and, therefore, the influence on processing latency [4]. 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, a framework optimized for high-performance packet processing, prevents triggering interrupts for network IO entirely. It ships with its own userspace driver, which avoids interrupts but polls packets actively instead. This leads to execution times with only little variation 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. For our investigation, we use a DPDK-enabled version of Snort that is already available (cf. Section VIII).

Virtualization can cause latencies of $350\,\mu\mathrm{s}$ and more for packet IO [5]. A study investigates several approaches for VM IO, where a hardware acceleration technique called single-root IO virtualization (SR-IOV) proved to be a low-latency approach causing the lowest CPU overhead [6]. SR-IOV splits the hardware into several virtual NICs, which can be attached to VMs exclusively to create a direct path between NIC and VM. This direct pass-through minimizes the software-layers between hardware and VM, providing a stable, low-latency IO. SR-IOV NICs, like the one used in Section V, integrate switches that can be used to form NF chains across VMs.

b) CPU: Hyperthreading (HT) is a technique that allows addressing physical CPU cores as several independent virtual processing cores. If two virtual cores are hosted on the same physical core, resource contention between processes of the seemingly independent virtual cores becomes possible. This resource contention can lead to delays in the processes running on virtual cores.

On current Intel CPUs, the first two cache levels are used exclusively by a single physical CPU core. However, the last level cache (LLC) is shared across all cores of a CPU, leading to resource contention on the LLC between processes running on different cores. The cache allocation technology (CAT) of modern Intel processors allows to partition the LLC across the available CPU cores. CAT can enforce exclusive LLC access for CPU cores and processes to restore the uncontended cache access latency [7].

To save energy, CPUs can lower their clock frequency and switch into sleep states. The lower the frequency and the deeper the sleep state, the longer it takes to switch the CPU back to the operational mode again. This can introduce additional delay for processes between $1\,\mu\mathrm{s}$ and $40\,\mu\mathrm{s}$ [8].

Multi-CPU systems present specific challenges when optimizing for latency. Resources, i.e., NICs or RAM, are always directly connected to one of the available CPUs. Such systems are often associated with the term non-uniform memory access (NUMA), because of the differences in resource access costs. Processes can access these directly attached resources faster than resources connected to a remote CPU. An NF should be pinned to the correct CPU core to use CPU-local NICs and RAM exclusively to minimize latency.

c) OS: DPDK prevents IO-related interrupts; however, the OS triggers interrupts for other housekeeping tasks, e.g., timers or scheduling. To trigger interrupts more predictably, Linux kernel patches exist, creating the so-called PREEMPT_RT kernel [9]. These patches lower the number of interrupts triggered by the OS or decrease the execution times of interrupts.

The Linux scheduler can cause delays for processes, for instance, if a process is migrated to another core. These effects can be prevented by isolating a CPU core from the Linux scheduler. After that, the OS user can dedicate a CPU

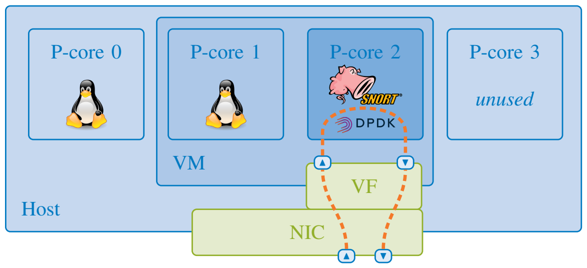


Fig. 2: Low-latency system architecture

core to a specific process exclusively, an approach known as CPU pinning.

Specialized embedded OSes exist, e.g., jailhouse [10], that offer real-time guarantees for user processes or VMs. We do not consider these specialized solutions despite their attractive features. These specialized OSes typically come with their own APIs and tools. We aim to use off-the-shelf toolchains and libraries, e.g., KVM and libvirt, to simplify migration from traditional to low-latency system architectures. Even the PREEMPT_RT kernel is available in the package repository of Debian.

IV. Low-Latency System Architecture

From Section III we can distill the following guidelines to create and configure systems for hosting low-latency NFs: *(i)* use specialized packet processing frameworks like DPDK, *(ii)* attach VMs using SR-IOV, *(iii)* disable HT and energy-saving mechanisms, *(iv)* apply Intel CAT to partition the LLC, *(v)* switch to an RT kernel, *(vi)* isolate cores hosting the low-latency application.

Applying the guidelines leads to a system architecture shown in Figure 2. Our example system uses a CPU with four physical cores, where all virtual cores and the energy-saving mechanisms are disabled in the BIOS. The test system has only a single CPU; therefore, we do not need to consider NUMA-related optimizations. We use a Debian with PREEMPT_RT kernel, on both the host and the VM, to minimize interrupt latencies for the virtualized packet processing application. Core isolation is used to restrict the OS processes and the application processes to specific cores minimizing the QoS impact on OS and application processes. The OS of the host runs on physical core (p-core) 0 exclusively. P-cores 1 and 2 are isolated for exclusive VM utilization. In the VM, the OS is restricted to p-core 1, isolating p-core 2 from host and VM OS alike. Our NF, i.e., DPDK and Snort, run on p-core 2. The core isolation feature complements DPDK's design philosophy of statically pinning packet processing tasks to cores. Utilizing SR-IOV, the NIC is split into virtual functions (VF). One VF is passed through to the VM attached to p-core 2. The critical network path and its associated CPU resources are isolated from OS tasks providing a stable service for latency-critical processes. We use Intel CAT to statically assign a large portion of the LLC to p-core 2.

V. Evaluation of our Low-Latency Architecture

We use a setup consisting of three nodes equipped with Intel Xeon D-1518 SoCs (quad-core 2.2 GHz) and X552 NICs ($2 \times 10\,\mathrm{Gbit/s}$). One node acts as a load generator (LG) directly connected to the device under test (DuT). The third node, the timestamper (TS), uses passive optical taps to timestamp the traffic between LG and DuT to determine the end-to-end delay caused by the DuT. Timestamps are collected passively, i.e., without introducing additional delay or variation in hardware with a resolution of $12.5\,\mathrm{ns}$. The DuT forwards the traffic between its two interfaces and sends it back to the load generator.

The DuT runs Debian buster (kernel 4.19), KVM as hypervisor, and a beta version of Snort 3 with a DPDK-enabled backend. LG and TS run MoonGen to generate and record the traffic. To measure the latency without the influence of protocol mechanisms, we use UDP for our measurements. As we want to investigate a realistic use case for intrusion prevention, we set the UDP destination port to 53 to trigger the DNS rules contained in our filtering ruleset. Bursty input traffic may cause short-time system overloads. Therefore, we rely on constant bitrate traffic for our experiments. We measured at different packet rates, $10\,\mathrm{kpackets/s}$ to $120\,\mathrm{kpackets/s}$, a typical range for a single-core Snort IPS. Section VIII lists the used software tools and versions.

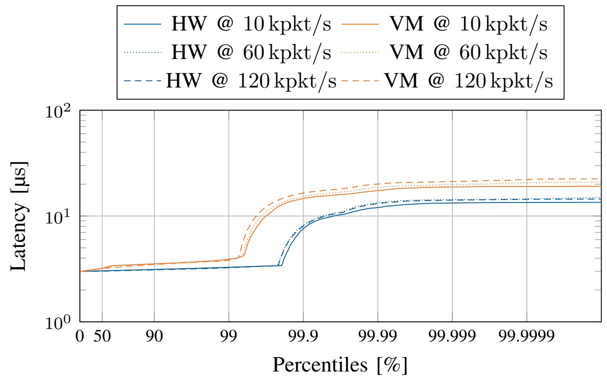
A. Latency Measurements

The measurements investigate the DuT's forwarding latency in three different scenarios. In the basic scenario, *DPDK-l2fwd*, traffic is forwarded between the DuT's two interfaces using only DPDK. This measurement determines the latency caused by the packet processing framework. For the second scenario, *Snort-fwd*, we run the Snort application on top of the DPDK framework and realize a forwarder based on Snort. The results of this measurement allow quantifying the overhead caused by the intrusion prevention framework. The most complex scenario is *Snort-filter*, where we activate a set of filtering rules. The outcome of this experiment is the impact of the rule application on the forwarding latency. We use the community ruleset provided by Snort for our investigation.

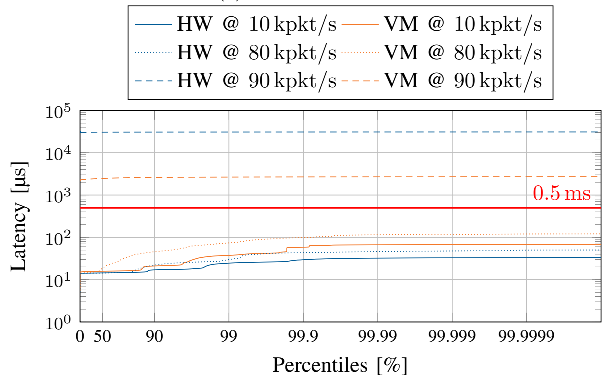
We tested each of the three different scenarios in two different system configurations. The first configuration, *HW*, deploys the application without any virtualization. The second one, *VM*, uses the configuration displayed in Figure 2. Both configurations were analyzed to determine the impact of virtualization on latency.

Figure 3 shows the forwarding latency of the three different scenarios, each for both configurations. The latency is plotted as an HDR-histogram, which displays the latency percentiles on a logarithmic x-axis.

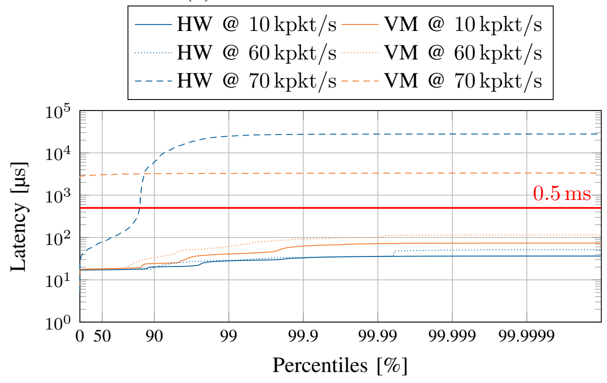
Figure 3a visualizes the latency performance of the DPDK forwarder. In this scenario, the different packet rates do not have a visible impact on latency. Latency stays



(a) DPDK-l2fwd



(b) DPDK + Snort-fwd



(c) DPDK + Snort + Snort-filter

Fig. 3: HDR-histogram showing forwarding latency of different forwarders

approximately constant up to the 99th percentile, for higher percentiles latency rises to approx. 15 µs (HW) and 20 µs (VM). The latency performance for the VM configuration is worse compared to the HW configuration, begins to rise at a lower percentile and has a higher maximum value. However, both values are well below the latency requirements of URLLC. This measurement demonstrates that the DPDK framework is an adequate foundation for low-latency packet processing. The negligible impact of the packet rates of DPDK is in line with our previous results [11], that demonstrate DPDK throughput at several million packets per second.

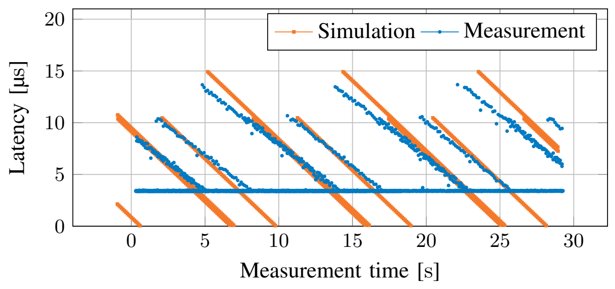


Fig. 4: 5000 worst-case latency events measured for DPDK-l2fwd (HW) at 10 kpackets/s

Building on the DPDK results, we use a DPDK-enabled version of Snort 3 for the measurement in Figure 3b. The most significant difference to the previous scenario is the impact of the packet rate on latency. For both configurations, latency exceeds the 0.5 ms goal if the packet rate is increased to 90 kpackets/s. The latency increases to approx. 3 ms for the VM configuration and to approx. 30 ms for the HW configuration. That difference is the result of the reduced NIC buffer for the VM configuration due to the activation of SR-IOV. The root cause of the high latencies is the overload situation caused by the increased throughput. Overloading the system causes packet losses, which violates the reliability criteria of URLLC. If the system is not overloaded, latency stays below 125 µs, i.e., fully compliant to URLLC requirements.

As the Snort framework can provide the required service levels, we investigate Snort 3 with an active community ruleset in Figure 3c. This system behaves similarly to the previous measurement. Latency increases to approx. 200 µs still meeting the expected service levels. If overloaded, the latency rises above 0.5 ms. However, the more complex processing rises packet processing costs, so this application is already overloaded at a packet rate of 70 kpackets/s.

Our measurements demonstrate that an overload situation has to be avoided to prevent packet loss and high latency.

B. A Closer Look on Latency

Figure 3 demonstrates that latency begins to increase even though it does not exceed our latency goal. To determine the root cause for this increase, we investigated the latency more closely. The latency increase is visible even for the most basic scenario, DPDK-l2fwd. Therefore, we use this scenario for our time-series analysis visualized in Figure 4.

There, we see a horizontal line containing the majority of latency measurements, and a regular pattern on top reaching latencies of approx. 14 µs. We found out that interrupts of the OS cause these high latencies. If triggered, OS interrupts pause currently running processes, in our case, the packet processing application. Packet processing resumes after the interrupt has been processed, leading to increased latency. Profiling the OS shows two different kinds of interrupts being processed on the CPU core of our forwarding application—local timer interrupts (loc) and IRQ work interrupts (iwi). Iwi and loc have different execution times, causing the two different maximum levels in Figure 4.

The regular pattern is a result of an interplay between two clocked processes, the generation of interrupts on the DuT at 250 Hz and the generation of the traffic on the LG at 10 kHz. The frequency of the generated traffic with its 100 µs inter-packet gaps is too low to sample the interrupts with an execution time below 15 µs. This process is widely known as undersampling, which leads to the creation of the observed pattern. We created a script, which simulates the two clocked processes. The resulting pattern, shown in Figure 4, is similar to the original pattern, which verifies our previous assumptions.

The latency for the virtualized forwarders starts to rise earlier and has a higher maximum value (cf. Figure 3). Performing the same analysis as for the HW case, we found out that the number of interrupts doubles and the execution time of interrupts increases by roughly 5 µs. The higher number of interrupts is a consequence of two OSes triggering interrupts—the OS of host and the OS of the VM. We attribute the higher execution times to the higher complexity for the interrupt processing in a virtualized environment.

VI. Limitations

Despite its benefits in terms of latency and jitter, the proposed architecture has disadvantages.

Disabling energy-saving mechanisms increases energy costs for the server. We measure a power consumption of 31 W/45 W (idle/processing) for our DuT with energy-saving mechanisms enabled. Disabling these mechanisms raises power consumption to 46 W/47 W (idle/processing). The consumption of the idling system increases by 48% and by 4% for a processing system. The latter, rather low figure, means that additional costs for a highly utilized system are comparatively low. The former, substantial difference for the idling system entails significant cost increases for under-utilized hardware.

One possible remedy against hardware under-utilization is the consolidation of several VMs onto a shared server. Overbooking the available hardware resources with virtual resources for the VMs works well if NFs can tolerate short delays caused by resource contention between VMs. However, our proposed architecture prevents resource sharing by statically assigning VMs to cores, especially for the isolated cores dedicated to URLLC NFs. This increases hosting costs for such a VM. Another solution to increase hardware utilization is VM migration. Migration allows efficient hardware utilization by migrating VMs onto systems with under-utilized hardware. VM migration with SR-IOV requires additional application support due to the non-trivial replication of the NIC's hardware state [6].

At first glance, our proposed solution has serious issues: disabling energy-saving mechanisms increases costs for under-utilized hardware, and at the same time, two possible solutions to that problem, VM overbooking and migration, potentially introduce short-time service interruptions violating URLLC requirements. However, we need to consider that URLLC is a solution for use cases with a specific set of requirements. Typical use cases that require URLLC are continually running, managing critical infrastructures such as industrial plants or power grids [2]. These critical use cases have a network demand that requires a continuously running, periodic exchange of messages. There are use cases that require URLLC service on demand, such as remote, robotic aided surgery [2]. These on-demand use cases only require URLLC communication during operation for a limited amount of time. However, during operation, these use cases have the same requirements as the other use cases, i.e., a highly deterministic, continuous flow of packets. The specific set of requirements makes the network demand of URLLC use cases less volatile and more predictable. This predictability enables network operators to assign hardware resources efficiently, ensuring high utilization. As we have shown, high utilization limits the additional energy costs for our proposed URLLC NF design. Thus, our design allows an ultra-reliable, low-latency, but cost-efficient and sustainable operation despite its previously mentioned limitations.

If use cases demand even lower latency than our system architecture can achieve, SmartNICs can be applied. These NICs offer tightly integrated processing capabilities, e.g., FPGAs or dedicated ASICs, that can provide continuously low latency for highly specialized tasks.

VII. Containers vs. VMs

Containers are an alternative approach towards virtualization that gained attraction in recent years. Network operators recognize the benefits of the more lightweight virtualization of containers. Containers start faster than VMs, which allows a faster initialization of NFs. Additionally, containers require less memory during operation, as they do not come with their own copy of the OS. Both properties make containers a perfect fit for hybrid cloud environments. In such an environment, containerized NFs can be moved and scaled efficiently between private and public clouds.

Despite the mentioned and other benefits like increased performance [12], why does our solution use VMs instead of containers? We argue that the benefit of containers for URLLC-compliant NFs is limited.

First, containers use the same hardware and underlying software as we use for our measurements. Therefore, containers face the same fundamental problems presented in this paper; they are subjected to OS interrupts, face the same CPU-related issues, and use the same IO interfaces. Our proposed system architecture provides solutions to all the mentioned problems that are also applicable to containers. The measurements show minor differences for throughput and latency between the bare-metal and the VM setup. These differences set the bounds for the latency and throughput achievable on a container-based NF. The containerized NF has a higher overhead than a bare-metal and a lower overhead than a VM-based deployment.

Second, a significant difference between containers and VMs is the way of resource isolation between different VMs and containers. Whereas VMs use hypervisor-based virtualization, containers use Linux cgroups for virtualization. A VM-based system architecture allows creating a strict *no*-resource-sharing policy between VMs and host OS. This policy avoids any harmful effects on latency (cf. Section V). All containers running on the same host, share the same instance of the kernel. Sharing may lead to unwanted resource sharing that we can exclude from the beginning by choosing VMs instead.

Third, one aspect mentioned in comparisons between VMs and containers is security [12]. VMs provide a higher grade of isolation between the hosting OS and the VM. Especially for security-related NFs, like the ones investigated previously, VMs are still an attractive way of deployment. However, the security flaws Spectre and Meltdown, published in early 2018, have shown that even the promise of strong VM-based isolation can be broken.

The choice between VMs and containers depends on the usage scenario. For URLLC-compliant NFs, we do not see a significant benefit switching to containers as performance gains are limited and resource sharing is harmful to our low-latency goal. For us, the previously mentioned aspects outweigh the potential benefits in cloud environments. Therefore, we chose VMs as a platform for our case study. However, containers may be the preferred solution if the benefits are prioritized differently or if URLLC-related restrictions do not apply.

VIII. Reproducibility

We see the reproducibility as a critical aspect of scientific experiments. Therefore, we provide all artifacts that were essential to the creation of this article. All investigated software components, i.e., DPDK, Snort, or filter rules, are publicly available. We further provide a repository for the experiment scripts, the evaluation scripts, and the used data on our GitHub page for interested readers [13].

IX. Conclusion

We demonstrated that adhering to URLLC-compliant QoS is a challenging endeavor. System interrupts can cause high tail latencies up to 2 ms that ruin the latency behavior of any NF. The maximum allowed latency for the most challenging URLLC applications is 500 µs. As NFs only make up a part of the entire network path, related work suggests a stricter latency goal of 350 µs [14]. Nevertheless, we demonstrated that even complex NFs, such as the demonstrated Snort IPS, can achieve the required service levels. We achieved a maximum latency of only 121 µs through an optimized system configuration and software architecture. Our system architecture can fulfill this even stricter requirement while relying on widely adopted tools and frameworks running on off-the-shelf hardware components.

The measurements were performed on a specific kind of NF, the Snort IPS. However, the fundamental challenges are rooted in the standard system components, such as Linux. Therefore, almost any NF based on the same standard components will suffer from the same problem. Fortunately, our proposed solution can solve all issues using only widely deployed components such as KVM or libvirt. This makes our results highly relevant to large scale cloud deployments for NFs that rely on the same components, such as OpenStack.

Our system configuration is applicable to container-based deployments, making our setup results relevant to that area. When comparing container-based and VM-based NFs for URLLC use cases, we argue that VM-based deployments are still superior, due to features such as their stronger resource isolation. However, container-based NFs may be advantageous for use cases where looser latency and reliability criteria apply.

Our proposed system architecture for low latency NFs relies on the most fundamental system configurations, such as CPU, NIC, or OS. Therefore, any form of deployment—container or VM—will profit from the proposed low-latency setup. For future work, we want to quantify the impact of containers and their isolation mechanisms on latency. A highly promising target for our investigation is called kata containers [15]. In kata, each container is executed by a hypervisor, thereby combining the benefits of both approaches, efficient portability and strong isolation.

References

[1] Sebastian Gallenmüller, Johannes Naab, Iris Adam, and Georg Carle. 5G QoS: Impact of Security Functions on Latency. In *2020 IEEE/IFIP Network Operations and Management Symposium (NOMS 2020)*, Budapest, Hungary, April 2020.

[2] 3GPP. 22.104 Service requirements for cyber-physical control applications in vertical domains V17.3.0. http://www.3gpp.org/ftp//Specs/archive/22_series/22.104/22104-h30.zip. Accessed: 2020-08-24.

[3] Mark Beierl. Nfv-kvm-tuning. https://wiki.opnfv.org/pages/viewpage.action?pageId=2926179. Accessed: 2020-08-24.

[4] Paul Emmerich, Daniel Raumer, Alexander Beifuß, Lukas Erlacher, Florian Wohlfart, Torsten M. Runge, Sebastian Gallenmüller, and Georg Carle. 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*, pages 6:1–6:8, 2015.

[5] Paul Emmerich, Daniel Raumer, Sebastian Gallenmüller, Florian Wohlfart, and Georg Carle. Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis. *J. Network Syst. Manage.*, 26(2):314–338, 2018.

[6] Giuseppe Lettieri, Vincenzo Maffione, and Luigi Rizzo. A Survey of Fast Packet I/O Technologies for Network Function Virtualization. In *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, pages 579–590, 2017.

[7] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family. In *2016 IEEE International Symposium on High*

Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016, pages 657–668, 2016.

[8] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science - R&D*, 30(2):219–227, 2015.

[9] Paul McKenney. A realtime preemption overview. https://lwn.net/Articles/146861/. Accessed: 2020-08-24.

[10] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no VM exits! (almost). *CoRR*, abs/1705.06932, 2017.

[11] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. 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*, pages 29–38, 2015.

[12] David Beserra, Edward David Moreno, Patricia Takako Endo, Jymmy Barreto, Djamel Sadok, and Stenio Fernandes. Performance analysis of LXC for HPC environments. In *Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015, Santa Catarina, Brazil, July 8-10, 2015*, pages 358–363. IEEE Computer Society, 2015.

[13] Sebastian Gallenmüller, Johannes Naab, Iris Adam, and Georg Carle. Reproducing Experimental Results. https://gallenmu.github.io/low-latency/. Accessed: 2020-08-24.

[14] Zuo Xiang, Frank Gabriel, Elena Urbano, Giang T. Nguyen, Martin Reisslein, and Frank H. P. Fitzek. Reducing Latency in Virtual Machines: Enabling Tactile Internet for Human-Machine Co-Working. *IEEE Journal on Selected Areas in Communications*, 37(5):1098–1116, 2019.

[15] A. Randazzo and I. Tinnirello. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214, 2019.

Sebastian Gallenmüller received his Master of Science in Informatics from the Technical University of Munich in 2014. There, he started as a Ph.D. student at the Chair of Network Architectures and Services. His main interests are the experimental evaluation of high-performance software packet processing systems and other programmable data planes with a focus on latency measurements.

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.

Iris Adam is a senior researcher at Nokia Bell Labs in Munich. She is mainly concerned with the investigation of security management and orchestration tasks. Her current research focus is the automated security management in 5th generation mobile networks.

Georg Carle is a professor at the Department of Informatics at 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 a postdoctoral scientist at Institut Eurecom, Sophia Antipolis, France, at the Fraunhofer Institute for Open Communication Systems, Berlin, and as a professor at the University of Tübingen.