Utilizing Hardware-Supported Containers for Low-Latency Networking

Florian Wiedner, Alexander Daichendt, Jonas Andre, Georg Carle
Technical University of Munich
Garching, Germany
{wiedner, daichend, andre, carle}@net.in.tum.de

***Abstract*—Applications requiring low-latency packet processing are challenging for today's network and service management when resources are limited and must be shared. Containers are suitable, but achieving connectivity between containers exclusively in software is unsuitable for low-latency requirements. The impact of network latencies in containerized environments has received comparatively less attention, particularly in comparison to research on virtual machines. This paper analyzes throughput and network latencies in container-based networks on a single host featuring single-root input/output virtualization, Linux Containers, and commercial off-the-shelf hardware. We conduct measurements using a state-of-the-art measurement methodology to identify tail-latency behavior, achieving a resolution of $1.25\,\mu s$. We evaluate a single flow in a line topology with up to 64 containers and a complex topology with 38 flows and 12 container nodes. The experiments demonstrate that pinning interrupt request handlers to non-uniform memory access nodes increases throughput and decreases latencies. Furthermore, we identify data translation lookaside buffer misses, rescheduling interrupts, and soft interrupt floods as critical challenges causing spikes in latencies while isolation remains impossible. Our findings identify bottlenecks for real-time container applications. A comparison with VM measurements shows that containers can achieve latencies up to $60\,\mu s$ lower. We support in this paper, network and service management in deciding on the underlying virtualization technology for packet-processing applications by providing recommendations accordingly.**

***Index Terms*—low latency, container, LXC, virtualization, NUMA, single-root input/output virtualization, networking**

This work is based on our paper presented at CNSM 2024 [1] extending it towards complex scenarios and resulting recommendations.

I. Introduction

Virtualization enables resource sharing and on-demand provisioning. Testing and validation in isolated environments help mitigate potential issues that may arise later in the development process. For instance, networks used for automotive driving and their protocols require low-latency guarantees, and testing is a significant challenge.

Furthermore, building expensive, complete hardware-based systems or manual rewiring is only feasible for small systems without additional on-demand requirements. As the system scales, the complexity of setup and maintenance increases, and human error becomes more likely. Network virtualization and resource sharing help minimize costs by automating setup and provisioning procedures.

Virtual machines (VMs) provide comprehensive isolation by virtualizing a complete operating system (OS), including the kernel. The hypervisor, a software layer, uses controlled access and abstracted interfaces for VMs to access physical resources. Due to this nature, VMs require a significant amount of additional resources. Containers use kernel-level features to establish an isolated sandbox at the process level and share the kernel with the host OS as a lightweight alternative.

Deploying multiple containers on a single host is common in digital twin environments, 5G slicing, and virtualized service function chains using network function virtualization. These scenarios require scalable, low-latency virtual network nodes on shared hardware. Wiedner et al. [2] demonstrated that virtual, software-based networking cannot achieve stable, low latencies due to the unpredictable nature of virtual networking. Additionally, they analyzed a hardware-assisted multi-node network virtual setup showing that they achieved stable tail latencies across different flows. Furthermore, Wiedner et al. [3] demonstrated that, in general, a single container can provide low latency with user- and kernel space networking. We can leverage the benefits of low-overhead virtualization by using containers alongside physical networking equipment. This paper analyzes throughput and latencies by creating networks on a single host featuring single-root input/output virtualization (SR-IOV) for NICs, Linux containers (LXC), and commercial off-the-shelf (COTS) hardware. SR-IOV allows to share a PCIe device such as a NIC as multiple lightweight PCIe devices among virtual nodes such as containers or VMs. With this, we aim to address the extent to which hardware-supported LXC containers can perform similarly to VMs in terms of tail latency in complex, multi-flow networks on a single host, and what influences tail latency accordingly? With a single host, we aim for a flexible way to run complex, multi-flow scenarios without the requirement to rewire and reduce the amount of hardware resources needed. Wiedner et al. [2] analyzed as well a complex, multi-flow scenario on a single host, however, using VMs as nodes. To achieve comparable results with multiple containers, we first evaluate a line topology to identify bottlenecks and then a complex network featuring 38 flows—with a flow representing a group of packets with the same destination IP address and port—and 12 container nodes to gain insight into complex scenarios and uneven resource-sharing requirements. We use these two scenarios to identify causes of tail latency in containers, depending on the number of instances in use and the available physical resources, and to identify potential optimizations that overcome these influences. These experiments replicate situations

related to deploying multiple containers on a single host by chaining many containers to model worst-case load and reveal systemic bottlenecks. This extends the current state-of-the-art by identifying possibilities to provide a network on a single host using lightweight virtualization while achieving low tail-latency. To contextualize our findings, we compare our results for this complex topology with prior work based on VMs. These investigations enable providing recommendations on tuning possibilities, placements, scalability, and mitigation considerations for network and service management when utilizing container or VMs in low-latency networking.

a) Research Goal and Objectives: We aim to investigate with this article to what extent SR-IOV-supported container networking provides stable, low-latency performance in both simple and complex network scenarios, providing recommendations for their usage to improve the utilization of multi-container-based systems and resources for network and service management. This research goal also aims to introduce system-level mechanisms to reduce latency in multi-container setups. We especially want to test the hypothesis that SR-IOV-based networking for containers can match or outperform VMs in multi-flow topologies under comparable resource constraints, and to show that using selected IRQ pinning methods improves the overall latency and throughput of multi-container systems. Furthermore, we want to identify whether, in multi-flow topologies, tail latency is dominated by co-located high-rate flows rather than hop count, and whether multi-queue configurations reduce tail latency for multiplexed flows.

The main objectives of this paper are:

1) pinning interrupt request handlers (IRQs) to non-uniform memory access (NUMA) nodes increases throughput and decreases latencies, especially tail-latency, when using per-core pinning,
2) insight into causes of high tail-latencies when using container-based packet processing, such as data translation lookaside buffer (dTLB) misses, rescheduling interrupts, and soft interrupt floods,
3) identifying influences of multi-flow cross traffic on tail latency in concurring containers,
4) a comparison of latencies in container and VMs,
5) a detailed study of multi-flow, multi-node, and multi-path scenarios and container chaining influences,
6) evaluation of influence factors and their importance when scaling up the number of containers until 64,
7) recommendation for using containers and VMs based on selected requirements.

b) Relation to Original Published Work: This paper extends on Daichendt et al. [1] presented at the International Conference on Network and Service Management (CNSM) 2024 in Prague, Czech Republic. In the original conference paper, we analyzed only simple line topologies with a single flow traversing through them with no additional multiplexing. In this paper, we extend the analysis to a topology with multiple nodes, flows, and paths traversing it, examine the influence of lightweight container isolation, and compare it to the same analysis performed on a tuned VM-based setup. We extend our insights and recommendations, along with the related state-of-the-art analysis based on the new insights we have gained. This extended analysis enables us to offer in-depth insights into utilizing containers to assess networks, with a focus on services that require low latency. Furthermore, we focus in this article on the impacts on network management and summarize technical details targeted at Linux kernel engineers.

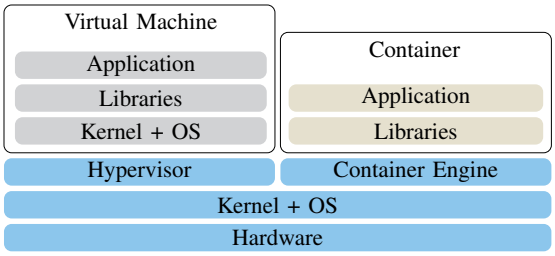


Figure 1. Comparison of VMs (left) and containers (right)

As such, we added the following new contributions, and research evaluations beyond Daichendt et.al. [1]:

1) A multi-flow topology with 12 nodes and 38 flows
2) A detailed analysis of IRQ scheduling effects under mixed-load conditions
3) A queue-configuration study
4) A direct comparison to VM-based networks under identical flow patterns
5) Strengthened recommendations for container versus VM deployment depending on latency requirements
6) Relation of the findings toward the usage possibilities in network and service management

c) Structure: The remaining paper is structured as follows: Sections II and III introduces the state-of-the-art and methodology. Sections IV and V evaluate and discuss the acquired data, including evaluations of both simple and complex scenarios. In Section VI, we recommend scenarios in which containers efficiently replace VMs and provide reproducibility information in Section VII. However, Section VIII disclaims limitations. We conclude in Section IX.

II. Background and Related Work

This section provides a comprehensive overview of background information and related work focusing on containers, SR-IOV, and the Linux network stack.

A. Virtualization Techniques

In cloud computing, software from numerous customers runs on shared machines in isolated environments. Two virtualization techniques are predominant: heavyweight VMs and lightweight containers, as shown in Figure 1. The base shows the hardware and the host OS, and the top part compares the two variants. The right side of Figure 1 shows containerization, including the container engine, and the left side shows a VM, highlighting the additional overhead of the guest OS within each system. VMs comprehensively virtualize an entire OS, encompassing a kernel, at the system level.

Yadav et al. [4] describes that VMs offer strict separation from the hardware and a complete OS. In contrast, containerization is less resource-intensive and highly flexible than

Table I
SUMMARY OF I/O PERFORMANCE CHARACTERISTICS OF VIRTUALIZATION IN THE STATE-OF-THE-ART

Technique	Sources	Latency	Throughput
KVM VM	[6]–[8], [10]	✓	✓✓✓
Docker	[5], [6]	✓✓	✓✓
LXC	[1], [5], [7], [8], [10]	✓✓✓	✓✓✓

VMs due to separating only parts such as processes and the networking space from the host OS. Containers are as shown in Figure 1 a lightweight OS-level isolation with a shared kernel. This work utilizes LXC due to its minimalistic approach. Morabito et al. [5] show that LXC performs marginally better than Docker but lacks convenience features.

Felter et al. [6] showed that a Docker container outperforms Kernel-Virtual-Machines (KVM)-based VMs in MySQL throughput due to minimized overhead. According to them, compute and memory access are nearly overhead-free, while OS interactions and Input/Output (I/O) cause overhead due to hypervisor involvement without direct passthrough of the hardware to the virtualization. Sharma et al. [7] found LXC to be 2% slower than bare metal across multiple benchmarks. Moreover, they found VMs to be 3% slower than LXC, though performance in I/O-bound processes was inferior. Bessera et al. [8] confirmed the assessment. Furthermore, Sharma et al. [7] revealed that LXC, compared to VMs, is more susceptible to noisy neighbors or adversarial workloads, leading to resource deprivation for co-located containers. These findings indicate that in I/O-bound and memory-intensive workloads, containers outperform VMs, but are more sensitive to concurrency. In our previous work [1], we identified that the placement of Central Processing Unit (CPU) cores and the memory used by the container are crucial for maintaining consistent latency under multiple containers and higher network load. Aqasizade et al. [9] analyzed key metrics such as memory, network, and CPU performance in comparison between different containerization and VM solutions, stating that Docker containers provided, on average, the best performance in comparison, without assessing latency. These results show that containers, in general, are a good choice for high-performance networking. The corresponding performance characteristics are summarized in Table I.

B. Single-Root Input/Output Virtualization

Sharing a single Peripheral Component Interconnect (PCI) device across multiple VMs or containers enables scalability and resource sharing. To avoid using the host OS to handle packet distribution, SR-IOV extends the PCI specification, enabling physical devices to present multiple lightweight virtual functions (VFs) to the PCIe bus and the OS. The device's physical function (PF) is a fully-featured PCI function for management and configuration. A VF has separate send-and-receive queues and shares the underlying physical resources. It can be interrupted independently from other VFs and the PF, resulting in minimal overhead, near line-rate throughput, and low network latency due to reduced overhead, as outlined by numerous studies [11]–[13].

Moreover, Dong et al. [14] demonstrated that SR-IOV significantly reduces CPU utilization and can achieve near line-rate throughput with up to 60 VMs in scale, but they did not analyze the influence on per-packet latency. Furthermore, Liu [12] benchmarked SR-IOV against bare metal and virtio devices—a virtual network device. Compared to bare-metal, SR-IOV exhibited slightly higher network latencies due to interrupt virtualization. Virtio performed roughly 5 times worse than SR-IOV, indicating that the technique is infeasible for real-time applications even without stressing the system. The SR-IOV throughput remained close to line rate. Gèhberger et al. [15] analyzed SR-IOV in combination with DPDK and other access methods, showing that it in general provides significant benefits, however, has challenges when focusing on communication on the same node by using the hardware switch as well as influences on each other from noisy virtual functions, limiting the possibilities compared to a direct hardware interface usage. In a separate attempt, Hwang et al. [16] have analyzed the feasibility of using DPDK instead of SR-IOV to accelerate network sharing through user-space networking and high-performance switching on the hypervisor, demonstrating a high-performance framework with higher scalability. However, it uses additional CPU and Memory resources on the hypervisor, reducing the maximum number of containers possible; it also requires cores on the NUMA node of the interfaces, reducing the benefits. As such, we utilize SR-IOV instead.

SR-IOV has achieved great success in high-performance networking and resource sharing. Related work states that PCI passthrough in combination with SR-IOV reduces network latencies due to less logic required to select the VF when using SR-IOV [11]. Wiedner et al. [17] analyzed the performance of internal, virtual connections versus SR-IOV in Mininet and identified the superior performance of SR-IOV in terms of latency, jitter, and throughput performance compared to complete virtual connections, allowing to share a network interface while still using the physical network connection. However, they did not evaluate container in a mixed setup with cross network traffic effects and their influences on tail latency.

C. Network Stack of the Linux Kernel

Direct memory access (DMA) transfers received packets from the PCIe bus into the main memory. A DMA descriptor determines the main memory location in the receive (RX) queue, a ring buffer holding pointers to physical memory [18]. Packet loss may occur if processing is not swift, i.e., during packet bursts, causing the hardware to overwrite data in the buffer.

When data is copied into the main memory, the NIC raises a hardware IRQ, notifying the CPU core handling the IRQ based on configured affinities, or the OS scheduler decides if not configured [19]. The OS prevents the NIC from raising another IRQ until it clears the current one, reducing the IRQ handler work to a minimum. The driver schedules a software

IRQ (softirq) to process the packets further and clear the hardware IRQ, allowing the NIC to raise another one. Data are copied into `sk_buff`, a kernel data structure used for buffering frames that contain additional metadata, presented as a double-linked list. This process leads to the networking stack validating the packet layer by layer until it reaches an application that further handles the packet, such as forwarding or processing.

Modern NICs can have multiple RX and transmit (TX) queues. Incoming packets are distributed using Receive Side Scaling (RSS). RSS calculates a hash based on header information to determine the processing queue [20]. Therefore, packets of a unidirectional flow are handled in the same queue to avoid out-of-order processing. Each queue has its own hardware IRQ, and the driver can merge all into `sk_buff`.

As the number of arriving packets increases, more time must be spent handling interrupts. As a mitigation, the Linux kernel implements busy polling, a form of interrupt coalescence, with the NAPI. After reaching a certain threshold of packets per second, the driver switches into poll mode and turns off hardware IRQs [21]. A separate thread periodically polls the RX queue for new packets. The number of packets to poll and the cycle are adjustable. NAPI significantly lowers the network latencies and increases throughput on busy systems, as shown by Salim et al. [22].

Beifuß et al. [23] created a model to predict latencies introduced by the Linux kernel and drivers and compared it against measurements. They observed that the interrupt rate decreased as the packet rate increased, indicating the successful transition from IRQ-based networking to NAPI.

Further, exists multiple options to accelerate the performance of networking with either moving the networking process from the Linux kernel to the user-space using for example DPDK which massively improves the performance of the networking stack as we identified in [3] especially for Intel-based machines; for the investigated AMD machine, the influence of one container when carefully optimizing was negligible. However, application support is required to bypass the kernel, and as such, these are typical kernel-based networking applications that are not directly supported [24]. Further acceleration techniques, such as eBPF or XDP, allow for increasing the performance of kernel-based networking but require either high-performance hardware support or application-specific hooks to utilize them to their full potential [25].

D. Network Performance with Containers

Among various network isolation mechanisms, such as Xen and OpenVZ, according to Xavier et al. [26], networking within LXC with network namespaces achieved the highest throughput and lowest latencies. While Linux-VServer performed identically to bare-metal, it provided less isolation because it sorted frames only by an identifier. Based on Xavier et al. [26], networking namespaces are a good choice for lightweight virtualized systems. Qi et al. [27] analyzed different container interface plugins on Kubernetes in detail, showing that, in general, accelerated networking outperforms traditional container networking approaches; they, however, did not analyze hardware-assisted techniques such as SR-IOV or network interface injection. Additionally, Santos et al. [28] evaluated the performance of Flanel as a typical container interface plugin in a Kubernetes cluster. They showed, however, that more analysis is needed to identify a valid solution for low-latency, low-jitter traffic within such networks. Furthermore, Mentz et al. [29] analyzed the performance of different virtual Docker container networking drivers, such as the host, bridge, macvlan, and overlay network modes, all of which utilize virtual drivers to add new network connections between containers. They conclude that the simpler the driver to be used and the more abstractions it provides, the lower the performance gain is. This insight results in the need to reduce abstraction layers in container networking as much as possible to improve overall performance. Struye et al. [30] analyzed the macvlan driver, bridge, and host networking in terms of network performance within LXC, Docker, and rk. They found that, on average, using the macvlan driver with LXC delivers the highest throughput and the lowest increase in latency compared to bare-metal, indicating that LXC is a valid choice for analyzing container network performance due to its lightweight design. They did, however, not focus on per-packet and per-flow tail-latency performance, required to identify rare events causing high latency spikes.

Ara et al. [31] evaluated the performance of software switches by using LXC on the sending and receiving sides. They compared kernel networking with VPP and SR-IOV, concluding that SR-IOV outperforms VPP in terms of throughput. Software solutions outperformed SR-IOV on a single host for latency, while SR-IOV exhibited lower latency in multi-host scenarios. However, their latency measurements were limited to mean values, lacked tail latency analysis, and did not include high packet rates. Wiedner et al. [17] analyzed the usability of namespaces in combination with SR-IOV and compared this to containers on a line topology, showing that much higher throughput can be achieved using SR-IOV. In contrast, latency remains stable with namespaces, with tail latency rising at the mean. De Oliveira et al. [32] analyzed the performance of SR-IOV in combination with a Docker container, showing that using a container only adds a small overhead compared to other virtualization solutions while providing stability benefits due to the isolation from other processes.

Rathore et al. [33] compared the performance of KVM and LXC containers as virtual routers. They benchmarked up to eight concurrent VMs or containers in terms of latency and throughput using SR-IOV and kernel networking. Although LXC achieves higher packet rates and slightly lower latencies than KVM, they advise against using containers; kernel networking is less isolated, and limiting CPU utilization spent on the network stack per container is impossible.

III. Measurement Methodology and Setup

Based on this background and state-of-the-art analysis, we are aiming to investigate to what extent SR-IOV-supported container networking provides stable low-latency performance in both simple and complex topologies. This examination enables the provision of recommendations for their use to

improve the utilization of multi-container-based systems and resources for network and service management. This research question also aims to introduce a system-level mechanism to reduce latency in multi-container setups. To achieve this goal, we describe in the first step a precise and accurate measurement methodology and hardware configuration that enables to investigate these questions.

The network performance of containers is evaluated using three hosts: a timestamping host, a load generation host (LoadGen), and the device under test (DuT) running containers. Figure 2 depicts the setup derived from [1], [2]. Using separate hardware hosts eliminates the influence of the measurement process on the results, enabling precise latency measurements.

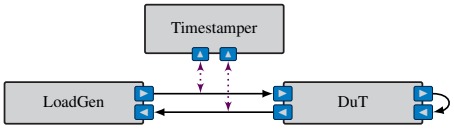


Figure 2. Experiment setup based on HVNet [2]

A. Overview

The LoadGen generates UDP datagrams based on IPv4 packets using MoonGen [34], a high-performance packet generator capable of saturating a $10\,\mathrm{Gbit/s}$ link with a single CPU core and precise timing using user-space networking with DPDK. The machine has an Intel Xeon Silver 4116, 192 GB of RAM, and a dual-port Intel 82599ES 10-Gigabit SFP+ NIC connected to the DuT via optical fibers.

The DuT hosts LXC 4.0.6 topologies on Debian 11 with real-time (RT) patches. According to Gallenmüller et al. [35], an RT kernel is preferable over others when cores are shared. Furthermore, an RT kernel aims to achieve deterministic behavior, especially in tail latency, and is therefore selected for this paper [35]. The DuT features an AMD EPYC 7551P, 128 GB RAM, and two interconnected dual-port Intel X710 10Gbe SFP+ NICs. One port on each NIC is connected to the ingress and egress of the LoadGen. Datagrams arrive at the ingress, are processed by the container topology, and are returned to the LoadGen.

The timestamper is connected to the ingress and egress via passive optical terminal access points, introducing twice the same constant delay, which is not visible in the latency calculations. Each packet carries an unique identifier, allowing the correlation of packets to calculate the exact latency through the topology. The calculations are performed by the MoonSniff framework [34] using the hardware timestamps of the corresponding NIC, unaffected by the software capturing the packets. This methodology allows latency to be gathered without inducing measurement artifacts. The timestamper is equipped with an AMD EPYC 7542, 512 GB RAM, and an Intel E810-XXVDA4 25-Gigabit NIC flashed to $10\,\mathrm{Gbit/s}$ providing $1.25\,\mathrm{ns}$ precision [36]. These NICs allow retrieving hardware timestamps for each packet on ingress, and therefore, we utilize a third machine connected via passive optical terminal access points to retrieve both times, before and after the DuT, in the ingress direction [36]. Each measurement was repeated at least three times.

To ensure reproducibility for the experiments, we use the plain orchestration service (pos) by Gallenmüller et al. [37]. pos relies on non-persistent live images to facilitate a pristine system for each series of experiments. The system controls bare-metal machines and VMs via IPMI. For containers, the VirtualLXCBMC is employed [10]. We use a set of scripts interacting with PostgreSQL to evaluate the collected packet dumps.

B. Optimizations

Several optimizations are applied to the OS kernel and the containers. Related work [38], [39] suggested turning off energy-saving features, read-copy updates, and activating poll mode when a core is idle. By turning off logging to reduce CPU usage, the latency for some packets can be reduced. Moreover, configuring IRQ affinity for core 0 steers interrupts to core 0, and we exempt this core from processing packets. Furthermore, simultaneous multithreading (SMT) is disabled, as it does not aid in handling interrupts [20]. The AMD HPC guidelines [39] suggest turning off C2 states, as it lowers power usage and puts a core into deep sleep. This sets the performance governor, preventing the CPU from scaling down its frequency and avoiding disruptions in packet processing.

The containers in the experiments have no additional applications running; the kernel and network namespaces handle all packet processing. We do this to ensure that we analyze the network and not the applications. Each container runs an array of libraries, including journaling, a DHCP client, an SSH server, and a dbus daemon, all of which require CPU time. Limiting all containers' user and system slices to CPU core 0 reduces work unrelated to processing IRQ handlers.

C. Arbitrary Flow Injection

Examining complex networking scenarios includes user-defined flows. A flow is a group of packets sharing the same destination IP address and port for this article. This objective can be accomplished by following the instructions by Wiedner et al. [2]. Their methodology involves SR-IOV on the ingress and egress interface of the DuT (cf. Figure 3) and passing a tuple of VF $(ingress\_vf, egress\_vf)$ to each VM. When the ingress NIC receives frames, they are associated with the VF's MAC address. Consequently, the NIC sets the upper bound for the number of nodes and logical connections supported to the maximum amount of VFs the NIC can provide, in our case, up to 64 nodes with one connection per node. In Figure 3 is the interface with its VFs connected to the egress of the LoadGen called `ingress`, and 0-2 on this interface represent the VF numbers which are attached to the ingress interface—`in` of each LXC container respectively. For the egress interface from our node, the naming is `egress` and `eg` with the reversed principle.

D. Network Topologies on a Single Host

The remaining two ports of the used NICs on the DuT are connected via a direct-attach cable to provide virtual

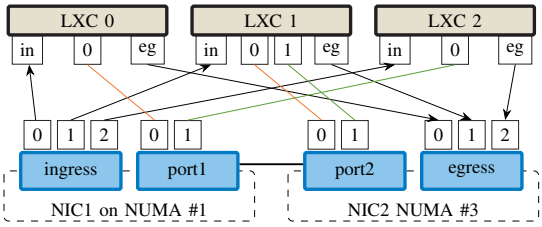


Figure 3. Overview of 3 hosts in line with SR-IOV on all NICs

topologies with real hardware. The logical connection between two nodes consists of one VF on each of the NICs sharing the VLAN tag. We use policy routing, a feature of the Linux kernel [21], to route packets based on the UDP destination port. This setup ensures a packet is transmitted over the physical wire, introducing delay due to propagation and serialization on ingress and egress.

In Figure 3, we illustrate how the physical hardware reflects a logical line topology with length three after combining the approach in Section III-C and the two interconnected NICs. Each container receives an ingress and egress VF from the respective physical NIC. Two lines of the same color indicate a logical connection between two containers with the same VLAN ID. For instance, LXC0 and LXC1 are connected with the VF0 on `port1` and VF0 on `port2`. Any packet between the two containers is sent over the physical wire connecting `port1` and `port2`. The interfaces in the container are numbered from 0 to 1, respectively, connected to the corresponding VFs, numbered from 0 upwards. We utilize SR-IOV VFs to provide connections to enable direct memory access, reducing the influence of the hypervisor on the latency, especially the tail-latency. Previous work [17] shows that SR-IOV, in combination with namespaces, improves latency performance significantly, similar to that used in LXC containers. As such, we are moving the interface into the container namespace and providing the container with direct access to the VF without further system influences, as it would be for container host networking using virtual interfaces in-between.

VFs from `port1` and `port2` are assigned to containers in a best-effort manner, resulting in assignments where memory must be copied between NUMA nodes. For example, in Figure 3, a flow with the hops 0-1-2 has to cross NUMA nodes once to copy the packets into the egress on node 3.

E. Methodology

To reduce complexity and focus on isolated effects, Section IV discusses a single flow in a line topology, stressing the system since every node processes every packet. We employ line lengths ranging from 1 to 64 containers, explore the maximum achievable packet rate, and measure network latencies. To further stress the system, we use constant-bitrate traffic with minimal packet sizes, ensuring that the performance observed is a lower bound to the achievable performance with mixed traffic and dynamic workload. With these experiments, we aim to employ a controlled way to reveal why degradation occurs when scaling container-based packet processing, and to achieve this, we stress the system to especially expose bottlenecks. Section V evaluates the performance of multiple flows in a complex topology. The results are compared with measurements from related work. To enable this, we use the same traffic patterns, topology, and packet sizes as in related work to facilitate a direct comparison between two systems that provide emulated topologies with low-latency support. Further information about the specific configuration, setup scripts, and additional measurement data is provided in Section VII. We further focus on Linux network space applications to retrieve insights for commonly usable applications and not specialized applications developed for user-space networking, such as DPDK. This focus allows us to identify performance improvements outside of adapting the application itself. The complex topology experiments evaluate whether the bottlenecks identified in the simple topology remain relevant in realistic multi-flow scenarios and whether containers remain competitive with VM-based approaches under such workloads. As such, we have not evaluated packet acceleration techniques such as XDP or eBPF, which should improve performance regardless of whether they are used in a container or on bare metal. However, analyzing these acceleration and their influence on multiple containers remains future work.

IV. Evaluation: Single Flow in Line Topology

This section presents the maximum achievable packet rate and identifies bottlenecks from a single flow in a line topology. We utilize constant-bitrate traffic with minimal-sized packets—64 B—for all measurements in this section to stress the system towards higher rates. With larger packet sizes, higher throughput is achievable; however, the stress of the system with the same packet rate remains similar according to previous work [3]. A more dynamic and realistic workload, enabled by higher packet-size diversity and different inter-packet gaps, delivers higher performance, as shown in this work, serving as a lower bound achievable when stressing the system itself. We use this series of experiments to compare different IRQ schedulers and node pinning approaches to answer the first part of the raised research question and hypotheses related to identifying how different pinning methods can improve the overall throughput and latency of a multi-container system. Using the single flow line topology, we can stress the system to reach worst-case values, clearly showing potential improvements leading to clear recommendations in the end. Additionally, while the measurements expose low-level kernel mechanisms such as IRQ handling, NUMA locality, and TLB behavior, these effects directly shape the manageability of large-scale, container-based network infrastructures. Understanding these bottlenecks enables operators to make informed scheduling, placement, and resource allocation decisions across many container-based networking tasks.

A. Packet Rate

We restrict the number of RX and TX queues to one per interface, as additional queues yield minimal benefits for one flow [20]. To precisely measure the packet rate and loss, we gradually increase the packet rate and count the number of packets sent and received. We measure packet loss for

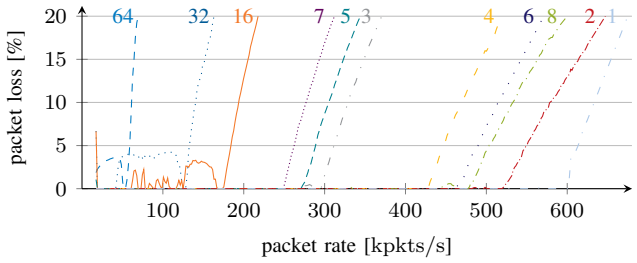


Figure 4. Packet loss at packet rate for line topologies with per node pinning. Each color represents a different length of the line-topology from 1 to 64.

20 s per rate before increasing the rate by roughly 1 kpkts/s. Figure 4 depicts the packet loss with per-node IRQ pinning. By pinning IRQs to all cores of the NUMA node to which the NIC belongs, memory copies are minimized. We further utilize the measurement methodology and setup, including the networking setup, container configuration, and optimizations described in Section III.

An interesting pattern emerges for long lines where a single core handles multiple IRQs. Before packet loss increases linearly with the packet rate, it can already be observed at some rates. For example, with 64 hosts, packet loss occurs in the interval 17.3 kpkts/s to 50 kpkts/s, but in the interval 50 kpkts/s to 55 kpkts/s no loss occurs. This behavior is caused by a combination of rescheduling interrupts and handling soft interrupt floods related to the NAPI mechanism to load balance between resource consumption and high performance. The NAPI moves from interrupt-based to a polling-mode to avoid situations where small packet loss occurs due to a receiving livelock caused by overloading single cores and therefore shifting soft interrupts with rescheduling between different cores [40]. This livelock causes an earlier drop in the processed packet rate before reaching the maximum processing capacity due to rescheduling interrupts and handling soft interrupt floods before switching to poll mode in NAPI [40]. This can result in a state where NAPI tries to overcome the receive livelock, moving to poll mode and then back to interrupt-based operation in a loop, causing additional interrupt rescheduling to reduce overloading a single core. Between 17.3 kpkts/s to 50 kpkts/s IRQs are processed regularly. The mechanism for dealing with soft interrupt floods explains that the stable packet rate increases linearly before [41], which is detailed in the appendix. Although no packets are lost, the latency becomes increasingly unpredictable due to the non-real-time scheduling policy, larger polling intervals, and scheduling latency. Conversely, the highest packet rates are in direct conflict with latency.

We analyze two variants of pinning IRQs towards their influence on latency and causing of additional effects such as dTLB misses: letting the scheduler decide how to handle IRQs (no pin) and pinning IRQs to all cores of the NUMA node the NIC belongs to (pin). For the first variant, we allow the kernel to schedule IRQs on all cores except the first one, as it handles other interrupts. To gather the data, we sample active IRQ threads for 10 s with `linux-perf` while the DuT is processing packets. The data presented in Figure 5 reveals that pinning IRQ threads to the correct NUMA nodes achieves a higher packet rate. We address this pinning in the further text as per-node IRQ pinning.

NUMA locality explains why lines with an even number of nodes generally achieve higher packet rates than those with an uneven number of nodes. The DuT, equipped with the first-generation AMD EPYC processor, has four NUMA nodes. Ingress is attached to node 1, while egress is attached to node 3 as depicted by Figure 3. When using an odd number of containers, the egress IRQ handler must copy memory from node 1 to node 3. However, memory is copied within the same node using an even number of containers. The PCIe bus delay remains constant in both cases; using an even number of containers bypasses the memory copy across NUMA domains at the final node in the line. In line with our data, Emmerich et al. [18] measured a penalty of 20 % on the packet rate for inadequate NUMA assignment.

With more nodes, the packet rate drops noticeably due to increased dTLB cache misses, especially with 64 containers, resulting in around 13 % dTLB misses. The dTLB caches recent mappings of virtual to physical memory addressing; more details about the dTLB cache are available in the appendix. If a mapping between a physical and virtual page is absent in the dTLB, an expensive page walk stalls the current cycle due to multiple slow memory accesses. Two factors contribute to increasing dTLB pressure: the high packet rate and core overcommitment when handling IRQs across more than eight containers. This makes it clear that dTLB cache misses due to a high packet rate or core overcommitment are a major bottleneck. To reduce this influence, network and service management tasks include avoiding core overcommitment and load balancing when high packet rates are expected. When using 16 containers, the packet rate drops to less than half that of 8 containers. This performance degradation stems from each core effectively processing two IRQ handlers resulting in inefficient scheduling.

None of the line lengths can saturate the maximum line rate of 10 Gbit/s divided by the number of used VFs per NIC. For example, a line with 64 containers with per-node pinning can achieve a throughput of 31 Mbit/s while the line rate per VF is at $\frac{10\,\text{Gbit/s}}{63} = 159\,\text{Mbit/s}$. Although SR-IOV incurs overhead, the OS and hardware bottleneck dominate the physical limitations.

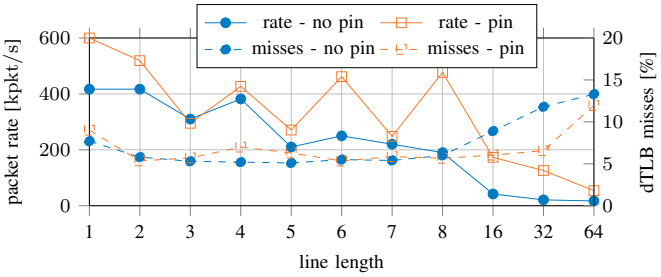


Figure 5. Packet rate and number of dTLB misses for line-topology of length between 1 and 64 container for both with and without IRQ pinning

B. Latency

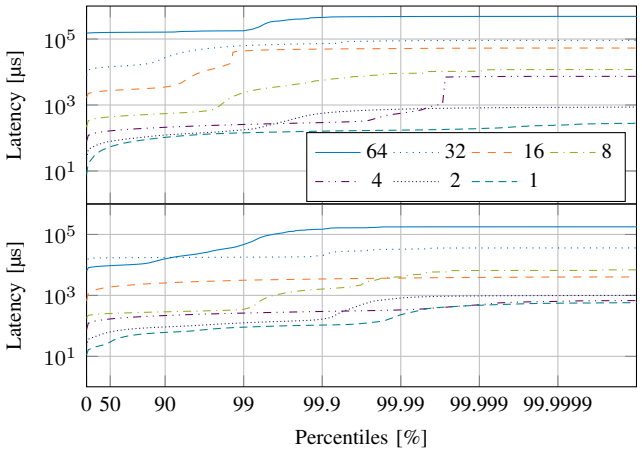


Figure 6. Latency HDR diagrams of line topology lengths for the maximum packet rate without packet loss; no IRQ pinning (top) and per node IRQ pin (bottom)

First, we measure at the maximum achievable packet rates reported in Figure 5 to stress the system while avoiding overload, which would harm latency [35], [42]. Second, a packet rate of $17.3\,\mathrm{kpkts/s}$ is employed as it is below the maximum packet rate for lines up to 64 and the lowest packet rate our script could generate. From multiple runs, the run with the highest worst-case latency is selected. High-dynamic-range (HDR) diagrams visualize the latency distribution by highlighting the impact of outliers in a dataset with logarithmic scales on both axes. This approach enables us to focus on the latency tail.

Figure 6 shows the latency distribution of a line topology with selected lengths without IRQ pinning in the top diagram and the bottom one with per-NUMA node IRQ pinning at the respective maximum packet rate. Due to varying packet rates and our Intel X710 NICs batching aggressively on lower packet rates [35], the latencies in the two HDR diagrams are not comparable. Without IRQ pinning, tail latency is less predictable, leading to frequent spikes. For example, at the 99.9^{th} percentile, a line with four nodes outperforms a line with two nodes. Furthermore, past the 99.99^{th} percentile, a sharp increase in latency is observed for four nodes, indicating that an interrupt was scheduled on the same core as the IRQ handler. Comparing the two HDR diagrams shows that IRQ pinning reduces latency and improves predictability. While the lower percentiles exhibit similar values, with IRQ pinning showing slightly lower latencies, the tail latency diverges past the 99.9^{th} percentile. For instance, the tail-latency of a four-node line is at $7400\,\mu\mathrm{s}$ without and $678\,\mu\mathrm{s}$ with IRQ pinning. With more than eight nodes, the effect of cores processing more than one IRQ handler becomes apparent. Even at the median, the latency of a 16-node line with $1094\,\mu\mathrm{s}$ diverges from the linearity of an 8-node line at $335\,\mu\mathrm{s}$.

To further investigate the impact of IRQ pinning, Figure 7 compares the tail-latency of IRQ pinning per-NUMA node and per-core. When pinning each IRQ handler to a single CPU core located on the same NUMA node, we are limited by the AMD EPYC 7551P to lines of length 8. Seven cores are necessary for the VFs on `port1` and `port2` (cf. Figure 3), and one core each is required for the ingress and egress. In scenarios where the number of containers in a chain is fewer than the number of per-NUMA CPU cores, per-core IRQ pinning results in stable, lower-tail latencies. While up to the 99^{th} percentile,

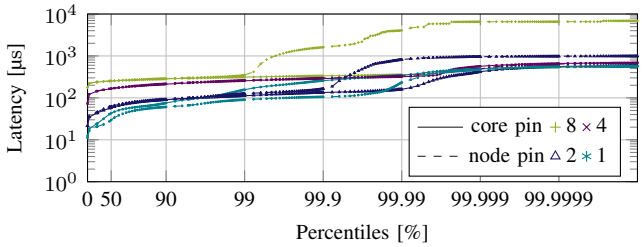


Figure 7. Latency HDR diagram of line topology lengths for the maximum packet rate without packet loss when utilizing either per-node or per-core pin

the latency is identical; rare rescheduling interrupts cause the tail latency to diverge when pinning IRQ handlers to NUMA nodes. A significant improvement is observed at percentiles above the 99.9^{th} percentile for eight nodes. When comparing lengths one and two, $99.99\,\%$ of the recorded events are relatively close, with diverging tail latency past the 99.99^{th} percentile. For only one container, the kernel forwards packets between two NICs; however, for two containers, all four ports of both NICs are processing packets with SR-IOV, introducing additional delay. In conclusion, IRQ pinning per core is strictly superior to per-NUMA-node pinning.

For network operators deploying many chained containers, per-core IRQ pinning provides the most predictable tail-latency envelope. However, it limits scalability as it consumes dedicated cores. Per-NUMA node pinning, although less stable, supports larger topologies. This tradeoff is essential when managing multi-tenant environments, where core allocation policies directly affect latencies.

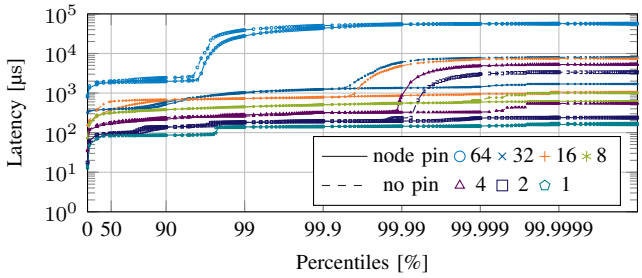


Figure 8. Latency HDR diagram of line topology lengths on a rate of $17.3\,\mathrm{kpkts/s}$ for no pinning of IRQs vs. per-node-pinning

The latency distribution for a packet rate of $17.3\,\mathrm{kpkts/s}$ is shown in Figure 8. Most events occur between $100\,\mu\mathrm{s}$ to $1000\,\mu\mathrm{s}$ while latency increases with the number of nodes. There is no significant difference between the two variants. Between the 90^{th} and 99^{th} percentile, the latency of a 64-node line is higher without IRQ pinning, converging with higher percentiles.

Most noteworthy are the infrequent, large latency spikes observed for both variants. For example, with node pinning,

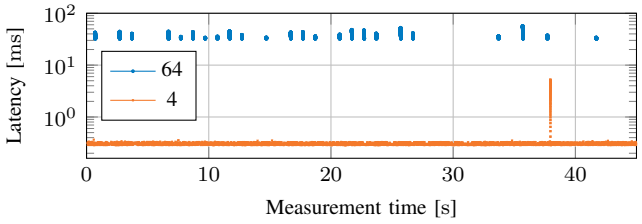


Figure 9. 5000 worst-case latencies on a rate of 17.3 kpkts/s with per-node IRQ pin

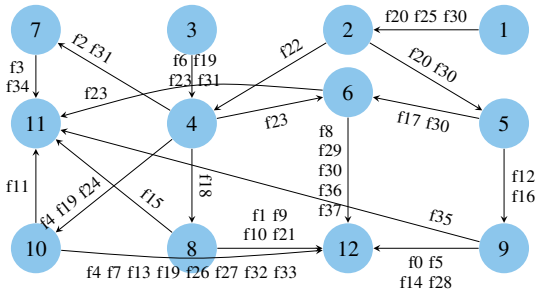


Figure 10. Topology `topo` of a network with 12 container nodes and 38 flows from [2]

a significant latency spike is observed, reaching 5.2 ms for 4 nodes and 60 ms for 64 nodes. Experiments with the maximum packet rate shown in Figure 6 do not exhibit such spikes. To further investigate this phenomenon, we display the 5000 worst-case latencies over time for lines with outliers and node pinning in Figure 9. One spike at 38 s is observed across four nodes, caused by a rescheduling interrupt. The same applies to the spikes displayed for 64 nodes. The kernel scheduler balances processes across all available cores, resulting in more rescheduling opportunities, a higher number of containers, and, consequently, a higher number of IRQs. We did not consistently observe these spikes in the remaining runs of the experiment for 1-32 nodes. For example, the other five repeats with four nodes did not show a spike; with 64 nodes, it occurred consistently in all repeats. Although the container system with IRQ pinning achieved a nearly loss-free maximum rate of 64 kpkts/s, when lowering the rate to 17.3 kpkts/s, occasionally packet loss occurs as shown in Figure 4. As Figure 9 displays, 64-node lines are more prone to rescheduling interrupts. Each core is overcommitted by almost 8 times, leading to increasing dTLB misses and worsening tail latency past the 90^{th} percentile. This shows that the networking subsystem has no threading bottlenecks and can be used in multiple containers simultaneously.

We recorded higher latency than related work [10] for a single host. While our measurements showed a median of 49 µs, related work reported a median of 13 µs. This disparity can be attributed to their use of PCI passthrough of ingress and egress, whereas we choose to utilize SR-IOV combined with PCI passthrough. Furthermore, for their measurements, they used a simple `tc`-based forwarder that copies packets at layer 2 between interfaces, bypassing the complexity of higher layers.

We found that pinning IRQs to cores yields lower tail latencies than per-NUMA-node pinning. In scenarios where more containers are deployed in a line than CPU cores per-NUMA node are available, overcommitting cores is an option when the packet rate is low; at higher rates, dTLB misses, and interrupts are more likely to cause higher spikes. This insight provides valuable considerations needed to manage networks when planning to employ multiple container nodes while aiming to achieve low tail latency.

V. Evaluation: Multiple Flows

Figure 10 shows a network with 12 nodes and 38 flows based on [2]. Each node is a container illustrated as numbered blue circle; the flows passing through a link are noted as edge description. The arrows of the edges represent the direction of the flows at this edge. As such, a flow traversing through two edges is named two times in the figure, a flow traversing through links three times. We selected this network for the examination to compare the results with Wiedner et al. [2] for VMs. As more queues support processing multiple flows, we use 4 RX and 4 TX. The ring sizes remain at 512. We aim to analyze the influence of multiplexing flows in our network on each other. This is especially important when analyzing the influences of different containers. We therefore aim to identify whether, in these multi-flow topologies, tail latency is dominated by co-located high-rate flows rather than hop count, and to analyze whether multi-queue configurations reduce tail latency for multiplexed flows. Moreover, we aim to analyze the impact of the number of queues on results when using multiple flows, allowing processing across more than one queue. This comparison and experiments allow us to verify the second hypothesis related to the research question, aiming to understand the differences between a VM-based and a container-based multi-flow multi-node scenario to analyze the influences of the previously analyzed findings from the line topology and potential differences when more than one flow in a complex topology is involved. This provides insights and will result in recommendations for multi-container network management.

We use a modified setup based on Section IV. As flows enter the network at different nodes, it is impossible to assign NUMA-optimized VFs to all containers. Therefore, we employ a simple approach by assigning VFs to containers via enumeration, which is not optimal for all flows; however, as shown in Figure 8, as the impact of NUMA locality is less pronounced at lower packet rates. We use the setup described in Section III, with SR-IOV VFs serving as flow injection points and for all connections between nodes. The software used in the container corresponds to that used for the network evaluation in Wiedner et al [2]. The flow rate and paths respect the maximum packet rate per link to avoid overloading. Each container is pinned to a specific core and handles multiple flows. The CPUs and IRQs are pinned to the same cores as in Wiedner et al. [2]. The packet size is 363 B and the packet rates range between 1 Mbit/s to 294 Mbit/s. Measurement duration is 175 s.

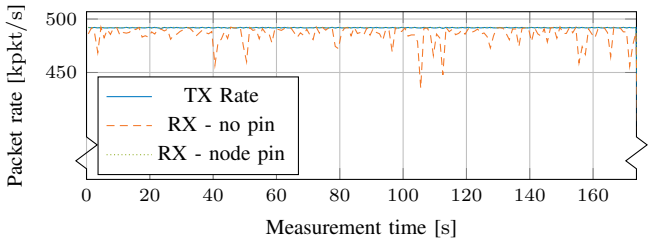


Figure 11. Aggregated packet rate of all flows traversed through `topo` comparing with and without pinning

Table II
5 MOST AFFECTED FLOWS BY WORST-CASE LATENCIES

Flow	hops	rate	worst-case latency
Flow 25	2 hops	293 Mbit/s	411 µs
Flow 31	3 hops	94 Mbit/s	394 µs
Flow 29	2 hops	68 Mbit/s	370 µs
Flow 37	2 hops	68 Mbit/s	369 µs
Flow 22	2 hops	73 Mbit/s	361 µs

A. Packet Rate

Initially, we measure the aggregated packet rate across all flows in two scenarios, as shown in Figure 11. First, the affinities of the IRQ threads are set to all NUMA node cores associated with the two NICs. Second, let the scheduler decide where to run the IRQ threads. The TX rate indicates the number of packets sent by the LoadGen, while the RX rate reflects the same after passing through the network on the DuT. The data reveal that packets are lost without IRQ pinning, resulting in 1.19 % of packets being lost. An unstable packet rate can be attributed to the scheduler assigning IRQs to cores that do not belong to the NIC's NUMA node, resulting in memory accesses across NUMA boundaries. The variant with per-node IRQ pinning achieves a stable packet rate almost identical to the TX rate, with only 6.7×10^{-4} % of the packets lost. Therefore, IRQ pinning is necessary to achieve a stable packet rate; we will use per-node IRQ pinning going forward.

B. Tail-latency

The tail latencies for the five flows with the highest latencies are coined by two large spikes shown in Figure 12, not observed for VMs [2]. Based on the recorded interrupts, there are two causes for the observed spikes: A TLB shootdown caused the first spike at 55 s as shown in Figure 12; details about TLB shootdowns are in the appendix. The second spike at approximately 120 s is caused by a rescheduling interrupt and a subsequent IRQ handler migration [43].

Table II presents more details, such as hop count, rate, and tail-latency for the selected flows. Flow 25, with the highest packet rate, has the highest latency; none of the flows with 1 Mbit/s are among them. Hop count is irrelevant to tail latency because each link processes packets in separate queues. Among the flows distributed across the network, only flows 29 and 37 traverse the same link between nodes 6 and 12, indicating that no resource competition occurs.

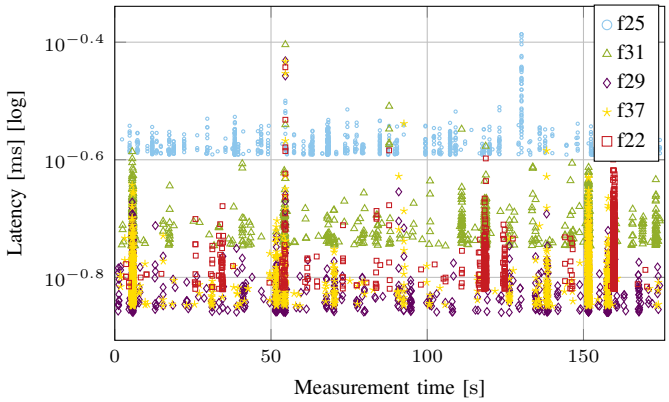
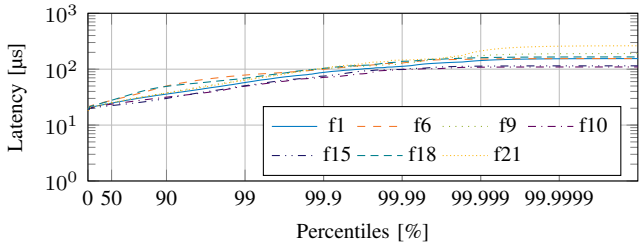
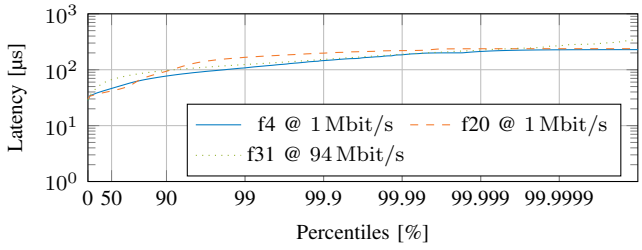


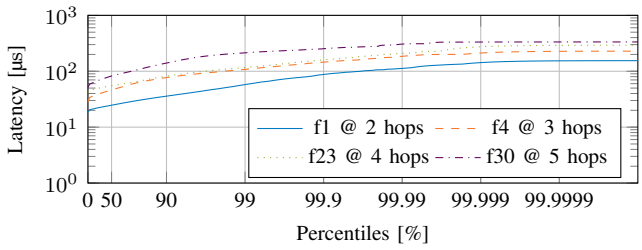
Figure 12. 5000 Worst-case latencies events for the 5 most affected flows after traversing through `topo`



(a) 2-hop measurement @ 1 Mbit/s



(b) 3-hop measurement



(c) 2 to 5-hop measurement @ 1 Mbit/s

Figure 13. Latency HDR diagrams of flows through `topo` with per-node pinning

C. Latency

Figure 13 presents the latency distribution. Figure 13a shows the latency distribution of 2-hop flows with a rate of 1 Mbit/s. The tail latencies are similar, except flow 21 ducks with 262 µs past the 99.999^{th} percentile compared to flow 9 with 190 µs. Flows 1, 9, 10, and 21 pass the same links between nodes 8 and 12, sharing VFs. Due to RSS [20], flows are distributed across queues with each having a separate IRQ handler. Scheduling the IRQs on multiple cores may introduce varying latencies,

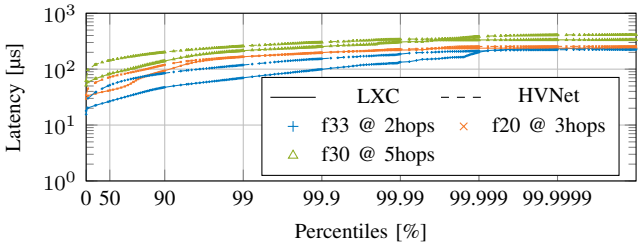


Figure 14. Latency HDR diagram comparing flows traversed through `topo` for both LXC and VMs as nodes [2]

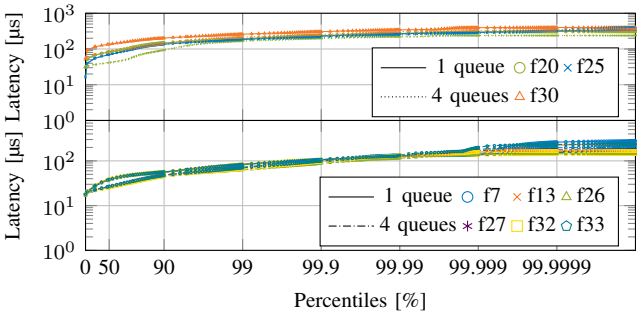


Figure 15. Latency HDR diagrams for queue configurations; node 1-2 (top) and node 10-12 (bottom)

which explains the higher latencies observed in flow 21.

Figure 13b shows the latency distribution of selected 3-hop flows. All 3-hop flows exhibit higher latency than the 2-hop flows. Flow 31 has a higher throughput of $94\,\mathrm{Mbit/s}$ compared to the other flows, which are at $1\,\mathrm{Mbit/s}$, not necessarily resulting in higher latencies. A more critical factor is the aggregated load of flows passing through the same node. For example, flow 20 runs alongside flow 25, with a rate of $293\,\mathrm{Mbit/s}$. Flow 31 has a higher median latency than Flow 20, but its worst-case latency is lower. Only for rare, singular events is the latency of flow 31 higher than that of flow 20. Figure 13c compares the latency distribution of selected two to five-hop flows. Latency increases with the number of hops, though not linearly. The difference between 2 and 3 hops for the flows is small, while the difference between 3 and 4 hops is more pronounced.

To contextualize the results, we compare the tail latencies of our approach with the results of HVNet [2]. Figure 14 emphasizes lower latencies for our approach for all percentiles. For up to the 99.9^{th} percentile, the difference is considerable, with a median of $82\,\mu s$ for our approach and $144\,\mu s$ for HVNet for flow 30. Due to space constraints, we only show the HDR diagrams for flows 20, 30, and 33. Towards the higher percentiles, the latencies of LXC and HVNet converge. For some flows, we observe slightly higher tail latencies than in HVNet, which is attributed to rare rescheduling interrupts. For example, flow 33 shows about $12\,\mu s$ higher tail latency than HVNet. VMs are less performant than containers due to the additional overhead required for memory management. The guest OS cannot control the physical machine memory for mapping virtual to physical pages; this is the hypervisor's job. Advanced features, such as nested page tables, are unavailable for our first-generation EPYC CPU. Hence, the hypervisor maintains a shadow page table for each guest OS. Whenever the hypervisor detects a guest page fault, it updates the shadow page table, which introduces additional overhead. In contrast, containers operate within their own virtual address space, eliminating memory management overhead.

D. Queue Configurations

Additionally, to provide recommendations toward queue configuration for network and service management, we investigate the impact of different queue configurations on the tail latency. Previously, we tested the default configuration with 4 RX and TX queues; now we compare it with 1 queue each.

First, the effect of one high-throughput flow on the latency of other low-throughput flows is investigated. As an example, the top diagram of Figure 15 shows the latency distribution of flows 20, 25, and 30 between nodes 1 and 2, where flow 25 has a throughput of $293\,\mathrm{Mbit/s}$ and flows 20 and 30 $1\,\mathrm{Mbit/s}$. Flow 25 remains unaffected by the number of queues. However, using only a single queue significantly affects the latency distribution of the other flow. Due to RSS, they are distributed across all queues, so a core other than the one handling flow 25 processes the IRQ handler. When using only a single queue, the IRQ handler processes 293 packets of the high-throughput flow for every packet of the low-throughput flow, resulting in higher latencies. Second, the effect of a high number of low-throughput flows over a single link. The link between nodes 10 and 12 provides an excellent example with seven flows. All flows have a throughput of approximately $2\,\mathrm{Mbit/s}$. No other flow passes through the same link. The bottom diagram of Figure 15 demonstrates clearly that for the percentiles lower than 99.9^{th}, four queues provide lower latencies than 1 queue, whereas this remains unclear for higher percentiles. As expected, the results show that more queues allow lower latencies for multiple flows.

VI. Recommendations

Containers are a more lightweight alternative to VMs for non-critical network applications with relaxed real-time requirements. We observed lower latencies for containers than VMs for the same topology. However, hardware choices must be carefully considered, as NUMA locality significantly impacts system performance, with the upper bound set by the CPU cores on the NIC's associated NUMA node. Latencies are stable when pinning IRQs to dedicated cores, followed by pinning them to the NIC's NUMA node as the second-best option, with latency spikes due to interrupt rescheduling. This is important when considering the usage of multiple containers with low-latency requirements. It is therefore required to distribute the load to improve the performance of pinning IRQs to the respective nodes without overcommitting resources.

We cannot recommend using multiple containers for applications with hard real-time requirements. Even if newer developments in the kernel, such as threaded NAPI for polling packets, may improve this situation, especially when packet rates change and NAPI switches between interrupt-based and

polling mode, as well as issuing softirqs during this time, reduce the benefits and harm, especially the tail latency. Our experiments demonstrated that containers cannot fulfill the strictest 5G URLLC requirement of $0.5\,\mathrm{ms}$ with the kernel networking stack. It is impossible to sufficiently isolate CPU cores from the kernel to avoid interrupts, confirming the assessment by Gallenmüller et al. [44]. To overcome this challenge, a more exclusive isolation technique, similar to VMs, would be needed to reduce the number of interrupts and their impact on different instances. However, applications with more relaxed real-time requirements can benefit from it. Examples include service-function-chaining [45], 5G network slicing with virtualized packet processing [46], and network digital twins [47]. Finally, we recommend using as many queues as flows per link, as this reduces latency. Although more queues increase context switching, but reduces the overall latency.

VII. Reproducibility

To reproduce our results, we provide setup scripts and all figures, including repeats not included in the paper due to space constraints, on our accompanying website[1].

VIII. Limitations

Hardware availability limited our experiments to a single setup. We have analyzed in Wiedner et al. [3] the differences for a single container between the setup used in this paper and an Intel-based setup, showing that the analyzed Intel machine is superior; however, even these systems are, in general, of the same generation as the used AMD machine. Due to current unavailability, future work will include experiments on monolithic Intel and AMD CPUs with faster interconnects. A CPU with more cores per-NUMA domain could enable a higher number of containers. Due to this limitation, the shown results correspond in detail to the hardware setup used.

In our examination, we focused on kernel-level packet forwarding, as introducing a more complex application would incur additional latency due to memory transfers to the user space. Moreover, additional delays would be introduced when using cores from another NUMA node. Furthermore, we analyzed only kernel-space and left user-space networking across multiple container instances for future work.

Furthermore, we focused on artificial traffic patterns and topologies. As such, a more realistic, dynamic workload influences results in terms of latency and achievable throughput. However, our measurements provide a lower bound on achievable throughput and an upper bound on achievable latency.

IX. Conclusion

Our work demonstrated that containers are a viable alternative to VMs for real-time networking when using the flexible kernel networking stack. Our results demonstrate that it is possible to reduce the impact of concurrent containers on one another through careful optimizations. Network emulators can benefit from the flexibility and scalability of containers while maintaining low latencies. Various solutions implementing containers, such as Containernet, already exist. However, they typically rely on kernel-based virtual networking, negatively impacting tail latencies [2]. Using real hardware and scaling with SR-IOV significantly improves latency. We have demonstrated that our approach can achieve stable, low latencies.

In this work, we measured the packet rate of containers in a line topology with up to 64 nodes. NUMA locality is crucial for achieving high packet rates; therefore, we recommend pinning IRQ handlers to the CPU cores on the NIC's NUMA node. We identified rescheduling interrupts as a significant factor for higher tail latencies. They can be mitigated by pinning IRQ handlers to the respective CPU cores to which the NIC is attached, thereby limiting the line length to the number of available cores on the NIC's NUMA node. When using long lines with 64 nodes, dTLB misses increase to around $12\,\%$, degrading the packet rate and tail latency.

Moreover, we measured the tail latency of a complex topology with multiple flows. We found that pinning IRQs to the NUMA node is necessary to achieve a stable packet rate without packet loss, as the scheduler is unaware of the NUMA topology. Comparing the same topology with VMs as the underlying virtualization technology, containers achieve lower latencies, with a median latency $62\,\mu\mathrm{s}$ lower on selected flows. This shows that, with careful configuration of the scenarios, similar or lower container latencies are achievable, even when different containers compete for the same resources. A per-flow worst-case analysis revealed that a flow's throughput is a significant factor for tail latency.

In the future, we plan to evaluate user-space networking, such as DPDK, in multiple containers. Previous work [10] suggests containers with DPDK can achieve lower latencies than with kernel networking, reducing the impact of the kernel. Additionally, we intend to evaluate how different kernels and schedulers affect the tail latency. Moreover, the potential impact of newer Linux kernel versions on latencies must be analyzed. Furthermore, we have yet to assess the newly introduced EEVDF scheduler in Linux 6.6, which is expected to offer reduced latencies due to a new latency-nice value. Finally, we plan to extend the evaluation to multi-host scenarios and evaluate the scheduling of resources on different nodes to stay in line with network management policies.

Acknowledgments

Our work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), projects HyperNIC (503359370) and SLICES-SUSTAINABILITY (566292327), by the EU Horizon Europe programme, project GreenDIGIT (101131207), by the German Federal Ministry of Research, Technology and Space (BMFTR), project 6G-life (16KIS2414), and by the Bavarian Ministry of Regional Development and Energy, project 6G Future Lab Bavaria. Results presented in this publication were obtained using the SLICES-DE research infrastructure.

[1] https://tumi8.github.io/extended-applicability-hwsupported-containers

APPENDIX

We outline previously summarized technical details.

a) dTLB cache misses: : The dTLB cache recent mappings of virtual to physical memory addressing to improve the data accesses, especially for parts that are accessed multiple times. This effect emerges when adding more containers in the chain, introducing new kernel data structures that must be touched for every packet. The aggregate memory footprint of these structures eventually exceeds the capacity of the per-core dTLB. When a dTLB entry is not present, the CPU must perform a multi-level page walk, which stalls the IRQ handler for several hundred cycles. Consequently, all TLBs are flushed, causing expensive pagewalks to find the pages to copy the arriving packets into. Under high packet rates, these stalls cause back-pressure inside the NIC driver's NAPI processing loop, reducing throughput and increasing queuing delay before IRQ completion.

b) soft interrupts: The system is overloaded when it spends more than $2\,\mathrm{ms}$ for ten repeats of processing soft interrupts. The kernel process `ksoftirqd/n` is scheduled to clear the remaining soft interrupts. Because `ksoftirqd/n` runs with the default, non-real-time scheduling policy, it can be preempted by other processes [42] Additionally, the `ksoftirqd/n` processes all remaining softirqs of any kind, which can result in higher, less predictable latencies for network softirqs. Moreover, Suo et al. [48] confirms that when CPU utilization is imbalanced, the kernel reschedules IRQs to other cores, causing rescheduling interrupts.

REFERENCES

[1] A. Daichendt, F. Wiedner, J. Andre, and G. Carle, "Applicability of hardware-supported containers in low-latency networking," in *20th International Conference on Network and Service Management, CNSM 2024, Prague, Czech Republic, October 28-31, 2024*, P. Varga, P. Celeda, T. Wauters, M. Tortonesi, J. François, and J. Galán-Jiménez, Eds. IEEE, 2024, pp. 1–7. [Online]. Available: https://doi.org/10.23919/CNSM62983.2024.10814577

[2] F. Wiedner, M. Helm, S. Gallenmüller, and G. Carle, "HVNet: Hardware-Assisted virtual networking on a single physical host," in *IEEE INFOCOM WKSHPS: Computer and Networking Experimental Research using Testbeds (CNERT 2022)*, Virtual Event, May 2022.

[3] F. Wiedner, M. Helm, A. Daichendt, J. Andre, and G. Carle, "Performance evaluation of containers for low-latency packet processing in virtualized network environments," *Perform. Evaluation*, vol. 166, p. 102442, 2024.

[4] A. K. Yadav, M. L. Garg, and Ritika, "Docker containers versus virtual machine-based virtualization," in *Emerging Technologies in Data Mining and Information Security*, A. Abraham, P. Dutta, J. K. Mandal, A. Bhattacharya, and S. Dutta, Eds. Singapore: Springer Singapore, 2019, pp. 141–150.

[5] R. Morabito, J. Kjallman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, Mar. 2015.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Mar. 2015.

[7] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale," in *Proceedings of the 17th International Middleware Conference*. ACM, Nov. 2016.

[8] D. Beserra, E. D. Moreno, P. T. Endo, and J. Barreto, "Performance evaluation of a lightweight virtualization solution for HPC I/O scenarios," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, Oct. 2016.

[9] H. Aqasizade, E. Ataie, and M. Bastam, "Experimental assessment of containers running on top of virtual machines," *IET Networks*, vol. 14, no. 1, p. e12138, 2025.

[10] F. Wiedner, M. Helm, A. Daichendt, J. Andre, and G. Carle, "Containing Low Tail-Latencies in Packet Processing Using Lightweight Virtualization," in *35rd International Teletraffic Congress (ITC-35)*, Oct. 2023.

[11] S. Huang and I. Baldine, "Performance Evaluation of 10GE NICs with SR-IOV Support: I/O Virtualization and Network Stack Optimizations," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 197–205.

[12] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support," in *2010 IEEE International Symposium on Paralle and Distributed Processing (IPDPS)*. IEEE, 2010.

[13] N. Pitaev, M. Falkner, A. Leivadeas, and I. Lambadaris, "Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, FD.IO VPP and SR-IOV," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, Mar. 2018.

[14] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, Nov. 2012.

[15] D. Géhberger, D. Balla, M. Maliosz, and C. Simon, "Performance evaluation of low latency communication alternatives in a containerized cloud environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 9–16.

[16] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Serv. Manag.*, vol. 12, no. 1, pp. 34–47, 2015.

[17] F. Wiedner, D. Kreutzer, J. Andre, and G. Carle, "Continuous integration for networks supporting low-latency using hybrid network emulation," in *36th International Teletraffic Congress (ITC-36)*, Trondheim, Norway, June 2025, pp. 1 – 9.

[18] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle, "User space network drivers," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 91–93.

[19] J. Bainbridge and J. Maxwell, *Red Hat Enterprise Linux Network Performance Tuning Guide*, mar 2015.

[20] T. Herbert and W. de Bruijn. (2023) Scaling in the linux networking stack. Accessed on June 20, 2024. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/scaling.html

[21] R. Rosen, *Linux Kernel Networking*. Apress, 2014.

[22] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proceedings of the 5th Annual Linux Showcase & Conference*. Oakland, California, USA: USENIX Association, November 5–10 2001.

[23] A. Beifus, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle, "A study of networking software induced latency," in *2015 International Conference and Workshops on Networked Systems (NetSys)*. IEEE, Mar. 2015.

[24] G. Ara, T. Cucinotta, L. Abeni, and C. Vitucci, "Comparative evaluation of kernel bypass mechanisms for high-performance inter-container communications," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science, CLOSER 2020, Prague, Czech Republic, May 7-9, 2020*, D. Ferguson, M. Helfert, and C. Pahl, Eds. SCITEPRESS, 2020, pp. 44–55.

[25] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with linux ebpf," in *30th International Teletraffic Congress, ITC 2018, Vienna, Austria, September 3-7, 2018 - Volume 1*. IEEE, 2018, pp. 209–217.

[26] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, Feb. 2013.

[27] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 1, pp. 656–671, 2021.

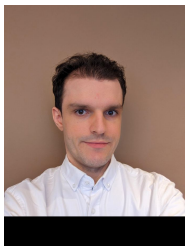
[28] J. Santos, B. V. Ninan, B. Volckaert, F. De Turck, and M. Al-Naday, "A comprehensive benchmark of flannel cni in sdn/non-sdn enabled cloud-native environments," *IEEE Transactions on Network and Service Management*, vol. 22, no. 6, pp. 5270–5284, 2025.

[29] L. L. Mentz, W. J. Loch, and G. P. Koslovski, "Comparative experimental analysis of docker container networking drivers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 2020, pp. 1–7.

[30] J. Struye, B. Spinnewyn, K. Spaey, K. Bonjean, and S. Latre, "Assessing the value of containers for nfvs: A detailed network performance

study," in *2017 13th International Conference on Network and Service Management (CNSM)*, 2017, pp. 1–7.
[31] G. Ara, L. Lai, T. Cucinotta, L. Abeni, and C. Vitucci, "A Framework for Comparative Evaluation of High-Performance Virtualized Networking Mechanisms," in *Cloud Computing and Services Science*, D. Ferguson, C. Pahl, and M. Helfert, Eds. Cham: Springer International Publishing, 2021, pp. 59–83.
[32] A. T. de Oliveira Filho, E. Freitas, P. R. do Carmo, E. Souto, J. Kelner, and D. F. Sadok, "Analysis of sr-iov in docker containers using rtt measurements," *Computer Communications*, vol. 228, p. 107961, 2024.
[33] M. S. Rathore, "KVM vs. LXC: Comparing performance and isolation of hardware-assisted virtual routers," *American Journal of Networks and Communications*, vol. 2, no. 4, p. 88, 2013.
[34] P. Emmerich, S. Gallenmüller, F. Wohlfart, D. Raumer, and G. Carle, "Moongen: A scriptable high-speed packet generator," *Proceedings of the 2015 Internet Measurement Conference*, 2014.
[35] S. Gallenmüller, F. Wiedner, J. Naab, and G. Carle, "How Low Can You Go? A Limbo Dance for Low-Latency Network Functions," *Journal of Network and Systems Management*, vol. 31, no. 1, Dec. 2022.
[36] Intel Corporation, "E810 datasheet rev2.6," 05 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/613875/intel-ethernet-controller-e810-datasheet.html
[37] S. Gallenmüller, D. Scholz, H. Stubbe, E. Hauser, and G. Carle, "Reproducible by design: Network experiments with pos," 2022. [Online]. Available: https://opus.bibliothek.uni-wuerzburg.de/28083
[38] S. Gallenmüller, F. Wiedner, J. Naab, and G. Carle, "Ducked Tails: Trimming the Tail Latency of(f) Packet Processing Systems," in *2021 17th International Conference on Network and Service Management (CNSM)*, 2021, pp. 537–543.
[39] Advanced Micro Devices, Inc., *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors*, Advanced Micro Devices, Inc., March 11 2020. [Online]. Available: https://www.amd.com/system/files/TechDocs/56305.zip
[40] C. D. Murta and M. A. Jonack, "Evaluating livelock control mechanisms in a gigabit network," in *Proceedings of 15th International Conference on Computer Communications and Networks*, 2006, pp. 40–45.
[41] J. Corbet. (2016) Threadable NAPI polling, softirqs, and proper fixes. Accessed on June 20, 2024. [Online]. Available: https://lwn.net/Articles/687617/
[42] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications," *ArXiv*, vol. abs/1809.02595, 2018.
[43] "Rescheduling interrupts," accessed on June 20, 2024. [Online]. Available: https://help.ubuntu.com/community/ReschedulingInterrupts
[44] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, "5G URLLC: A Case Study on Low-Latency Intrusion Prevention," *IEEE Communications Magazine*, vol. 58, no. 10, pp. 35–41, 2020.
[45] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *Journal of Network and Computer Applications*, vol. 75, pp. 138–155, 2016.
[46] X. Li, M. Samaka, H. A. Chan, D. Bhamare, L. Gupta, C. Guo, and R. Jain, "Network slicing for 5g: Challenges and opportunities," *IEEE Internet Computing*, vol. 21, no. 5, pp. 20–27, 2017.
[47] Y. Cui, J. Liu, M. Yu, J. Jiang, L. Zhang, and L. Lu, "Network digital twin," *IEEE Network*, vol. 38, no. 1, pp. 5–6, 2024.
[48] K. Suo, Y. Shi, A. Lee, and S. Baidya, "Characterizing networking performance and interrupt overhead of container overlay networks," in *Proceedings of the 2021 ACM Southeast Conference*. ACM, Apr. 2021.

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 and research associate at the Chair of Network Architectures and Services. His research focuses on low-latency networking, virtualization, routing, and the performance of post-quantum-safe TLS.



Alexander Daichendt finished his Master of Science in Informatics in 2025 at the Technical University of Munich. Currently, he works on low-latency software-defined media transport for the broadcasting industry.

Jonas Andre finished his Master of Science in Informatics in 2019 at the Technical University of Munich, where he worked until 2025 as research associate at the Chair of Network Architectures and Services. His research focuses on wireless networks.

Georg Carle is Professor at the Technical University of Munich (TUM), holding the Chair of Network Architectures and Services. He studied electrical engineering at the University of Stuttgart, with periods abroad at Brunel University London and ENST Paris, now Télécom ParisTech, and received his Ph.D. from the University of Karlsruhe, now KIT. Before joining TUM, he held positions at Institut Eurécom in Sophia Antipolis, France, at Fraunhofer FOKUS in Berlin, and at the University of Tübingen. His research focuses on networked systems and security.