Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Lightweight low-latency virtual networking

Alexander Daichendt

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Lightweight low-latency virtual networking

Leichtgewichtige virtuelle Netzwerke mit niedriger Latenzzeit

Author:	Alexander Daichendt
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Florian Wiedner, M. Sc.
	Jonas Andre, M. Sc.
Date:	August 15, 2022

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, August 15, 2022

Location, Date

Signature

Abstract

Networking with containers is becoming increasingly prevalent in many industries due to their lightweight and flexible approach to virtualization. We evaluate the viability of containers for reliable low-latency networking. Previous work measures latencies without high accuracy and precision in the µs range, focuses on heavy-weight virtualization, or prioritizes bandwidth analysis over latencies. This thesis compares network latencies of containers with virtual machines and specifically focuses on tail latencies. We build a program to orchestrate containers in the testbed, then use existing tooling to measure latencies and compare them with measurements of virtual machines and bare metal. The results of this thesis indicate that containers perform identically compared to virtual machines; however, they are less isolated and more susceptible to interference by programs running on the host. Interferences can cause spikes in tail latencies, which can be mitigated by using a real-time kernel.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

With Industry 4.0 the convergence of information technology (IT) and operational technology (OT) advances and forces the evaluation of existing technology under new conditions; for example, low-latency networking with containers is becoming increasingly prevalent. Industrial applications have strict latency requirements: on the one hand, a response must occur within 0.5 ms and on the other hand, the variance in response times should be minimized, even for the 99.99999^{th} (7 nines) percentile of packets [1]. Traditionally, a programmable logic controller (PLC) controls manufacturing processes, machines, and robots and provides data on the current state. A PLC is an industrial computer specialized in high availability, fault tolerance, reliability, and low latency [2]. Typically, a PLC has to meet hard real-time requirements: no deadline can be missed.

Some computers collect data generated by a PLC or other sensors and send them to another more powerful computer for analysis. When systems rely on these computed results, transmitting the data over the Internet is infeasible due to unreliable high latencies. Hence, the industry is moving towards edge computing, which moves some parts of the data analysis from the cloud closer to the manufacturing plants. An edge computer often features a wireless connection to widely distributed sensors. Depending on the application, an edge computer only meets soft real-time requirements [2], where missed deadlines are occasionally acceptable [3]. The collected data are processed for purposes such as logging, visualization, and monitoring. Typically, an edge computer runs on specialized equipment that is optimized for goals such as adaptability, scalability, safety, latency, and resilience. Multiple edge computers may depend on each other's data. Meeting real-time constraints between PLC and communication with other edge computers requires a reliable high-performance, low-latency interconnection.

Multiple services can run on one edge computer; however, they should not affect each other. Virtualization is creating an abstraction layer on top of a computer and its resources, so that multiple virtual computers may utilize these resources. Each virtual computer is isolated from the rest of the system and uses only the allocated resources. With virtualization, the deployment of an edge computer is standardized, scalable, and rapid. This increases maintainability, reduces errors, and saves money. There are two forms of virtualization: heavy-weight virtualization with virtual machines (VMs) and lightweight virtualization with containers. In recent years, container implementations such as Docker and Linux containers (LXC) have gained significant popularity. Containers facilitate the creation of portable software services by integrating the required environment and dependencies, but not every part of an operating system is virtualized. On the one hand, containers share the kernel with the host, and on the other hand, containers typically rely on kernel-level features of the host to isolate certain resources so that full virtualization of hardware is not necessary. Additional properties such as cold starts in milliseconds and hardware independence render containers a widely applicable technology. For these reasons, containers might be suitable as an underlying technology for certain edge computers.

This thesis investigates the viability of containers for virtual, ultra reliable low-latency communication (URLLC). To do so, we answer the following research questions.

- **RQ1**: Are containers viable for virtual URLLC?
- **RQ2**: How large is the difference in network latencies between containers and VMs?
- **RQ3**: What are the root causes for the observed differences, and what can we do to minimize them?

We use HVNet [4], a project for low-latency networking with VMs as a framework. We extend this project to support LXC and optimize these containers for predictable low latencies. Although Docker is more popular, we decide to use LXC, which mimics the behavior of a VM more closely. To answer **RQ1**, we take into account whether it is possible to run the measurement tooling in a container and whether we can achieve a sufficient amount of isolation. After integrating containers into HVNet, the same tooling is available for both VMs and containers. Additionally, we also conduct baseline experiments on bare metal and compare the network latencies of containers, VMs and bare metal. These measurements are the basis for the answer of **RQ2** and also contribute to **RQ1**. Tail latencies are of special interest to investigate how reliable containers are and are analyzed as part of **RQ3**. Furthermore, we analyze the differences between containers and VMs and how they could affect latencies.

1.1 Contributions

This thesis addresses the research questions by extending an existing framework for virtual URLLC with containers. We perform a structured comparison between the two virtualization techniques and gather quantifiable evidence for differences in network latencies. The main contributions of this thesis include:

- implementing tooling for low-latency network experiments with containers.
- proposing a framework for running networking tooling such as Data Plane Development Kit (DPDK) in containers. Often scripts are written with the assumption of running on bare metal, so that access to certain kernel features fails in containers.

1.2 Outline

The remaining thesis is structured as follows. Chapter 2 introduces background information such as CGroups and namespaces, HVNet [4], and DPDK. Next, Chapter 3 covers studies that have already been conducted related to this thesis. In Chapter 4 we analyze problems related to virtual networking, containers, and factors that influence latencies. The implementation process is detailed and presented in Chapter 5. In Chapter 6 the measurement methodology is introduced, our results are discussed and compared with VMs. Finally, in Chapter 7, we summarize our findings and suggest future work.

Chapter 2

Background

The main area of application for this thesis is low-latency networking with containers. An overview of the characteristics, advantages, and disadvantages of containers is presented in Section 2.1. Kernel-level features that are detrimental to containers are introduced in Sections 2.1.1 and 2.1.2. For low-latency scalable networking on a single host, we use single root I/O virtualization (SR-IOV), which we introduce in Section 2.2. In Section 2.3 we introduce HVNet, a set of scripts for low-latency virtual networking on a single host, which we extend in this thesis. Lastly, Section 2.4 explains DPDK, a framework for high-performance networking in userspace.

2.1 Container

In recent years, containers have been established as a competitive alternative to VMs. Both are a form of virtualization. There are three generic types of virtualization [3]: full, para, and operating system (OS) level. Containers are classified as OS-level, while virtual machines belong to the class of full virtualization. Virtualization creates an abstraction layer over the physical resources so that they can be shared across multiple processes that are independent and should be isolated from each other.

There are two types of containers [5]: application and system. Application containers focus on running a single service per container. They typically only bundle the minimum required libraries to run a single application [6]. The most prominent example is Docker. A system container mimics an entire operating system, including all libraries. Multiple processes may run at the same time. Both provide an environment that is separated from the remaining system in terms of libraries and hardware resources to perform

specific tasks. There are numerous reasons for classifying containers as lightweight virtualization.

The first reason is that containers do not emulate their own kernel; instead, they are controlled by the kernel of the host system. Figure 2.1 demonstrates the architectural differences between VMs and containers. Unlike a container, each VM has its own kernel, but requires a hypervisor to manage access to hardware [3]. Sharing the kernel with the host and other containers has implications on isolation. To remedy some concerns, modern kernels offer features that help build isolated systems. The most important features are CGroups and namespaces, which we discuss in Sections 2.1.1 and 2.1.2.

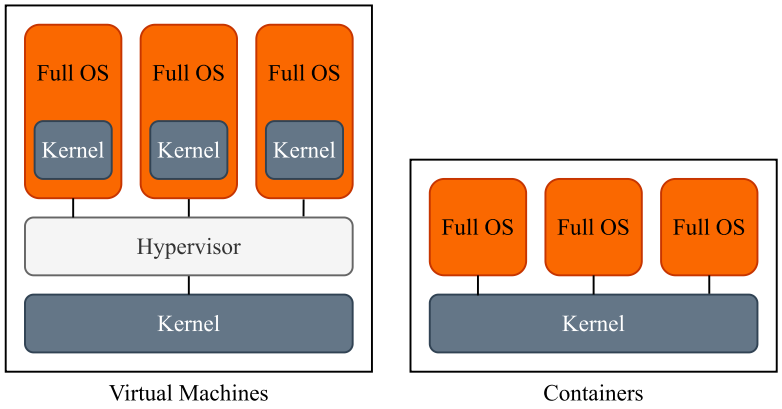


Figure 2.1: Architectural differences between virtual machines and containers

Containers do not need expensive abstraction layers that virtualize CPU, memory, and I/O [7]. The remaining overhead for isolation is low: In terms of performance, containers compete with bare metal [6], [8]. Due to the absence of heavy-weight abstraction layers, it is possible to better utilize physical hardware. A container only uses as many resources as it needs for the current task while respecting defined limits. Is a resource no longer required, it is immediately available to other containers or the host. This makes it easy to overprovision physical resources for more dense deployments.

However, there are downsides: it is not possible to run an OS from a different family in a container due to the inherited kernel. For example, it is impossible to run a Windows 11 container on top of a Linux kernel. Moreover, interaction with the kernel is limited. This implies that some features, such as the ability to load kernel modules, are not available in a container.

2.1.1 Control Groups

Modern versions of the Linux kernel ($>$ 3.10) support control groups, which limit, prioritize, control, and audit access to physical hardware resources for processes [6].

Listing 2.1: Listing the effectively assigned CPU cores

```
1 $ cat /sys/fs/cgroup/user.slice/cpuset.cpus.effective
2 0-5
```

There are two versions of CGroups available. Modern OS, such as Debian Bullseye, are based on version 2, which changes the internal workings and organization of the subsystems. Granted that version 1 is in the process of being deprecated, the remaining Chapter is applicable to version 2.

CGroups are organized hierarchically. This means that a child CGroup inherits all the properties of its parent, and it is impossible for a child to request more resources than the parent provides. On a Linux-based OS, CGroups are mounted as a pseudo-filesystem at `/sys/fs/cgroup`. The hierarchical structure is reflected in this folder structure. For example, the CGroup `test` is mapped to `/sys/fs/cgroup/test`.

For each resource, such as CPU time, CPU cores, memory, devices, etc., a subsystem exists. A subsystem restricts, audits, and freezes resources within a CGroup [9]. The subsystems are reflected as files at each level of hierarchy. An example for a subsystem is `cpuset`, which assigns CPU cores to a CGroup. The command in Listing 2.1 shows the effectively available CPU cores within the `user.slice` CGroup.

Most Linux distributions rely on systemd as the default init system. When an OS with systemd boots, systemd creates the root CGroup at `/sys/fs/cgroup/` [10]. Moreover, systemd acts as a manager for the CGroup hierarchy by creating slices that inherit all subsystems from the root CGroup [11]. A slice is an abstract concept for managing resources for groups of processes. Depending on the kind of process, it is assigned to a specific slice. The most important slices are as follows:

- `-.slice`: root slice, all other slics inherit from this CGroup
- `user.slice`: user session and all processes running in user space
- `system.slice`: services and scope units
- `machine.slice`: virtual machines and systemd containers

For every process, systemd creates a new CGroup as a child of a slice. With CGroups v2, the CGroup, which hosts a process, must not have a child [12]. The command `systemctl status` displays the current CGroup hierarchy of the system.

2.1.2 Namespaces

Namespaces provide different views on system resources by providing abstractions. Each process running in a namespace appears as the sole owner of the abstracted resources. Changes to isolated system resources in a namespace are neither visible to the host nor to other namespaces. Namespaces are completely unrelated to CGroups. Older versions of the Linux kernel support six different namespaces, while modern versions provide eight; the TIME and CGROUP namespaces were added recently [13]:

- NET: manages the network stack; own routing table; creates new interfaces
- PID: assigns new process identifiers (PIDs) starting from 1
- UTS: different host and domain name
- MNT: filesystem isolation; mounts new filesystems
- IPC: isolation of signals, pipes, and shared memory
- USER: isolates user and group IDs; maps root user within container to unprivileged user on host.
- TIME: virtualizes two system clocks of a Linux system: CLOCK_MONOTONIC and CLOCK_BOOTTIME.
- CGROUP: isolates CGroups, so that it appears that the namespace is the owner of the root CGroup.

A process may join multiple namespaces and multiple processes can join the same namespace. All things considered, namespaces isolate key OS resources, such as network stack, PIDs and filesystem. Subsequently, it is possible to create an isolated environment for lightweight virtualization by relying on kernel-level features.

2.2 Single Root Input/Output-Virtualization

For network experiments on a single host with realistic latencies, every virtual machine or container requires a network interface card (NIC). However, it is infeasible to assign one NIC to each node when more than a few nodes are used. On the one hand, computers will run out of PCIe lanes or slots, and on the other hand, it is expensive to buy dozens of high-throughput NICs.

SR-IOV, which is an extension of the PCI specification for virtualizing PCIe devices [14], solves this problem without significantly impacting networking performance. Figure 2.2 illustrates the arrangement of a physical function (PF) and virtual functions (VFs) on

one NIC. A VF is a lightweight PCIe device that has performance-critical resources: independent memory space, interrupts, and a direct memory access (DMA) stream to the process using the VF [3]. Each VF is associated with a PF [14], which is a fully configurable PCIe function. All VFs are connected by a virtual Ethernet (vEth) bridge, which sorts incoming packets according to the MAC or virtual local area network (VLAN) tag and delivers them to the right VF [15]. A container or VM controls a VF by binding it to a VF driver. This setup allows VFs to use DMA to make packets available to the container.

SR-IOV with VMs requires an input–output memory management unit (IOMMU) to correctly map the address spaces between the host and the guest. For containers, this is not necessary, since the address space is shared and no translation of physical to virtual memory is required.

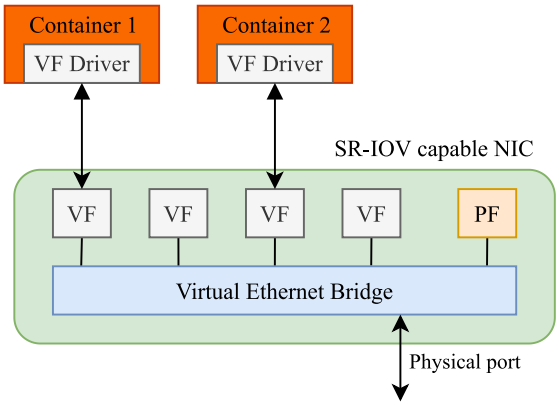


Figure 2.2: Conceptual illustration of SR-IOV

2.3 Hardware-Assisted Virtual Networking

HVNet [4] is a framework for performing network experiments on a single physical host with low latencies. Arbitrary topologies can be created by defining them in a configuration file. As nodes, HVNet is based on VMs, which are interconnected with SR-IOV. HVNet optimizes the software stack for low latencies by applying a wide variety of patches. It delegates orchestration and management of the hardware hosts involved in an experiment to plain orchestration service (pos) [16]. This includes setting up the desired Linux live images, boot parameters, and out-of-band control with Intelligent Platform Management Interface (IPMI).

HVNet offers a complete framework to run reproducible and automatic network experiments. To minimize the impact of the packet generator on the Device under Test (DuT), a separate host can be utilized. Since timestamping of every packet is not possible with

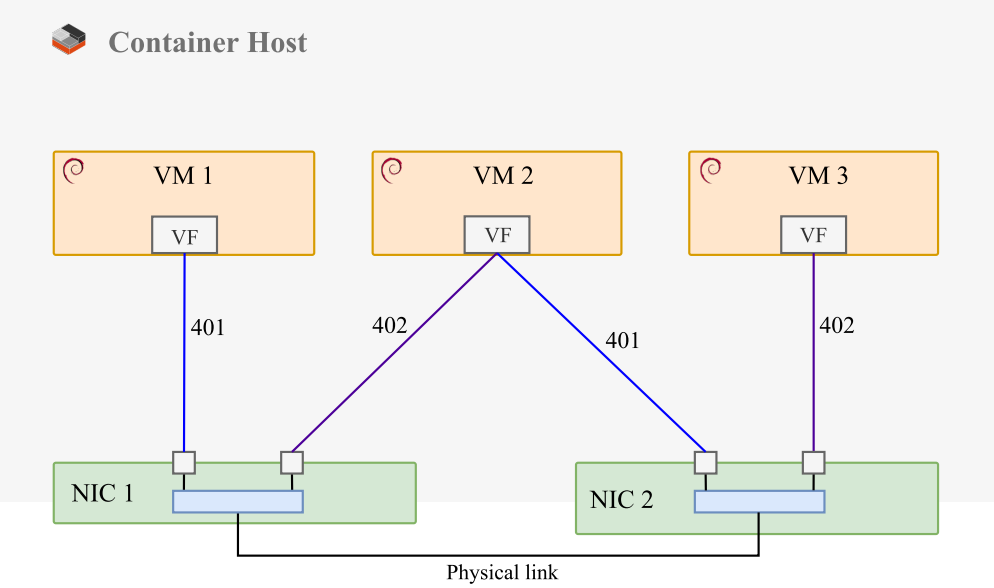


Figure 2.3: Three containers connected in a line with VFs and VLAN IDs

commodity hardware [17], a third host, the timestamper, may be used. Optical splitters duplicate traffic without adding a delay between the load generator and DuT, and send it to the timestamper. The timestamper then records the latency for every packet and outputs a packet capture (pcap) file.

For each virtual node, separate hardware resources, such as the number of CPU cores, memory, and scripts, may be defined. A separate script for setup, experiment, and teardown offers a high degree of flexibility in the application of HVNet. The assignment of CPU cores and memory can be optimized to be non-uniform memory access (NUMA) aware.

For networking, HVNet uses SR-IOV. Various factors, such as packet processing in the kernel or userspace, physical hardware, and a wire between network cards contribute to latencies. For this reason, HVNet assigns VFs from two NICs, which are interconnected with a physical cable, to VMs. Figure 2.3 shows how an example with three VMs connected in a line translates into the virtualized setup with VFs. In this setup, there is one connection between VM 1 and 2, and one connection between 2 and 3. Each connection is assigned a separate VLAN ID. A connection consists of two VFs located on different network cards with matching VLAN IDs. This setup forces packets to travel from one NIC to another via the physical link. Due to the VLAN IDs the packets must be serialized on the NICs, which adds latency, just like in a bare metal environment [4].

2.4 Data Plane Development Kit

The Data Plane Development Kit (DPDK) is a set of libraries to accelerate packet processing. Various techniques come into action to increase speed. The conventional Linux kernel network stack processes incoming packets by raising I/O interrupts, moving the packets into the kernel buffer, and then into the userspace buffer [18]. Therefore, with interrupt-based packet processing, packets are copied multiple times in memory, which is a reason for unpredictable delays. DPDK solves this problem with a userspace network driver that continuously polls for new packets, zero-copy to bypass the kernel when copying packets, and hugepages for efficient memory management.

2.4.1 Poll mode driver

The poll mode driver of DPDK is based on the `igb_uio` kernel driver, which is built on the userspace I/O framework (uio). The uio framework is an interface between the kernel and `igb_uio` protecting driver developers from unstable kernel API and internal kernel functions and macros [19]. Binding a device to the `igb_uio` driver creates the device file in `/dev/uioX` where `X` is the number of the device starting at 0. Through these device files, a developer can interact with PCI devices in userspace programs bypassing the kernel entirely. For this purpose, the following methods are exposed [19]:

1. **mmap()**: access to device memory
2. **read()**: blocking call to mimic the behavior of interrupts. When an interrupt is caught, the `read()` call returns.

A downside of the poll mode driver is that one CPU core is constantly occupied with polling. Due to the shift of networking to userspace, the kernel is no longer involved and fewer variations in latencies occur [20].

2.4.2 Zero-copy

Zero-copy is a mechanism that eliminates multiple copy actions. Each NIC has one or more receiving (RX) and transmitting (TX) queues, which are created in main memory. These queues do not store the entire packet but a descriptor that references the location of the entire packet. The packet is stored in a separate memory pool. When a packet arrives, it is copied directly into the memory pool with DMA, and the descriptor is added to the RX queue. Since DMA requires stable physical addresses, the memory pool is created in statically allocated 2 MB hugepages, which are excluded from the Linux page migration mechanism [21]. The memory pool is shared between the NIC and the application so that incoming packets are immediately available to the applications [18].

This thesis does not work directly with DPDK. Instead, the software packet generator MoonGen [22], which is built on top of DPDK, is used. MoonGen offers example scripts for generating and forwarding packets, which reduce the learning curve of interaction with DPDK. MoonGen is built on top of libmoon, a library that handles core functionalities and interfaces with DPDK.

Chapter 3

Related work

For many years, substantial interest has been shown in the research of containers and their underlaying technologies. However, most networking related research focuses on measuring bandwidth and not latencies. Section 3.1 presents literature related to our testbed and measurement methodology, which is essential for the prototype and evaluation. The literature in Section 3.2 outlines containers and their performance in low-latency networking. Finally, Section 3.3 highlights the existing work on optimizing the software stack for lower latencies.

3.1 Testbed setup and testing methodology

All experiments in this thesis are reproducible because they are implemented on top of the plain orchestration service (pos) [16]. pos is a novel approach to start a structured and reproducible experiment. Hosts are controlled out-of-band with IPMI, Intel vPro, or AMD's Pro features. Live images ensure that no host is tainted by previous experiments. Hosts can be parameterized with variables, and processes can be synchronized. HVNet is tightly integrated into pos, which is why pos is essential for this work.

For generating packets, we rely on the flexible packet generator MoonGen [22]. MoonGen relies on DPDK for fast packet processing, which is capable of saturating a 10 GbE link with minimum-sized packets using a single core. MoonGen's flexibility originates from using the scripting language Lua with a JIT compiler to configure the program. Due to many examples provided, the learning curve is not steep. In addition to being a packet generator, MoonGen can forward packets. We exclusively use these example scripts for our experiments.

3.2 Containers and performance evaluation

Watada et al. [6] conducted a comprehensive review of available research on container technology. Their work compares containers and VMs with respect to isolation, security, orchestration, and performance. Different container solutions use different isolation and resource allocation mechanisms. In summary, this work serves as an entry point for virtualization solutions and their enabling hard- and software features. Relevant for our work is the comparison of key technologies between containers and VMs and the performance comparison between container solutions and heavy-weight virtualization. They find that all container implementations perform similarly, but outperform VMs in many benchmarks.

Krishnakumar's work [15] measures latencies for Docker containers that are attached to SR-IOV VFs. Another host that generates packets with Pktgen is connected to the PF. In addition, they conduct experiments with OpenVSwitch. One of their goals is to develop a latency-sensitive scheduler for Kubernetes. In contrast to that, we are evaluating the viability of containers in the context of low-latency networking and focus on measuring and optimizing the latencies. Their latency measurements are only accurate in hundreds of µs which is sufficient for their Kubernetes scheduler. The first experiment, which assigns two VF from ingress and egress NIC to a Docker container, measures 800 µs. Another experiment utilizes an OpenVSwitch and vhostuser ports attached to one container; the same latency is measured. Our work does not look at OpenVSwitch at all, and furthermore, we rely on previous work and infrastructure that allows recording latencies in the range of ns.

A media gateway (MGW) plays a critical role in the 5G core network. By placing the MGWs closer to the user, lower latencies can be achieved. Naturally, internet service providers are looking for ways to minimize latencies for deployed network virtualized functions. Chen et al. [23] develop a Docker-based media gateway, which adjusts the Layer 3 and 4 information of each packet passing through. The work compares the throughput and average latency of the kernel network stack with DPDK-based networking for both bare metal and Docker containers. In conclusion, for bare metal 49 µs average latency is recorded, while for Docker the number is slightly higher at 69 µs. Since their implementation of the MGW rewrites parts of the packets, which adds processing delays, it is not directly comparable to our work.

The work of Mao et al. [24] compares the worst-case latencies of VMs, Docker containers, and bare metal for a real-time kernel, a low-latency kernel, and a generic one. Measurement is carried out with 500 000 cycles of Cyclictest, a network latency benchmarking tool, and DPDK. Under system load, they find that Docker delivers near bare metal

latencies (32 µs vs. 30 µs) when using the real-time Linux kernel. The VM shows much higher latencies with 23 614 µs. Without load, all kernels perform nearly identically with latencies of around 10 µs for Docker and bare metal. VMs are again much worse with latencies of more than 5300 µs. This indicates that we may expect lower latencies for our container prototype as well.

Zhang et al. [25] conducts a comparison between containers, VMs, and bare metal with the goal of evaluating the performance differences of SR-IOV and passthrough for high-performance computing (HPC) applications. They find that containers with passthrough outperform VMs for both passthrough and SR-IOV. Furthermore, they find a 9 % overhead of containers with passthrough over bare metal. No studies have been conducted with containers and SR-IOV. To fill this gap, our implementation supports topology creation with SR-IOV links. However, collecting these measurements is reserved for future work.

3.3 Low-latency optimizations

Related work indicates that containers might be competitive in low-latency networking. Most of the research around low-latency virtual networking focuses on VMs and makes numerous recommendations that we can reuse for our prototype. For example, the work of Gallenmüller et al. [17] optimizes latencies and jitter. To do so, they showcase the *nohz* kernel option and various kernel boot parameters, which isolate cores. Just as in our work, they utilize SR-IOV for realistic latencies. In conclusion, their results suggest that their optimizations deliver slightly lower and more stable latencies than a real-time kernel. They identify the translation lookaside buffer (TLB) shootdowns as a possible cause of tail latencies. We reuse their suggested low-latency optimizations and adjust them as necessary for containers.

HVNet by Wiedner et al. [4] creates a framework for conducting reproducible networking experiments on a single host. HVNet interfaces with pos to spawn virtual machines as nodes, which are interconnected with SR-IOV for low latencies. For evaluation, they collect latency data of 100 different topologies. The worst-case latencies are between 0.22 ms and 0.41 ms, which is an improvement compared to software solutions such as Mininet. This thesis integrates containers on top of HVNet so that the tooling and measurement scripts are reusable. In future work, a direct comparison between our solution and HVNet will be possible.

Chapter 4

Problem Analysis

This Chapter presents three problems related to virtual networking on a single host. Section 4.1 explains the first challenge, which is the technology for interconnecting nodes. For that, we discuss software network simulators, their solution to this problem, and why they are insufficient. As an alternative, we discuss hardware features such as SR-IOV. The second problem is introduced in Section 4.2: the choice of technology for virtual nodes. There are reasons why containers might perform better than VMs. Finally, Section 4.3 highlights possible sources of tail latencies, which should be considered to ensure applicability in real-time systems. Based on this analysis, we derive requirements for the prototype.

4.1 Virtual networking on a single host

There has been considerable interest in virtualizing network experiments on a single host. Network experiments on real hardware are expensive to set up: Each node requires a host with its own CPU, memory, mainboard, and one or more high-performance NICs. For each experiment, the cabling must be adjusted, which requires physical presence at the site. This approach is more prone to errors due to human interaction. We analyze approaches that virtualize or simulate networking with the goal of delivering low, reliable, and accurate latencies. Furthermore, the solution must be compatible with containers.

4.1.1 Network simulators

Over the years, numerous network simulators have been presented. A few examples are Mininet [26], ContainerNet [27], ns-3 [28], and Dockemu [29]:

Mininet's approach to virtualizing networking is lightweight: it creates process groups that are interconnected with vEth-pairs in network namespaces [26]. Through a Python interface, the user defines the topology and the scripts to run on each node. As demonstrated by Wiedner et al. [4], this approach produces tail latencies much higher than those of the alternatives and will not be considered.

Containernet is a Mininet fork that implements Docker containers as hosts [27]. Networking with vEth-pairs is inherited from Mininet. The work [30] shows that Docker performs worse in terms of I/O, raw CPU power, and various benchmarks for Hadoop and Spark compared to LXC. Additionally, Docker containers typically only bundle libraries to run one application at a time, which reduces the adaptability. A better comparison to a VM is a system container. As a consequence, we do not investigate Docker or Containernet further.

ns-3 is a discrete event network simulator written in Python and C++. Topologies and programs are defined in C++ scripts, which would require a rewrite of our measurement tooling. Although Beifuß et al. [28] showed that after implementing their own simulation model, latencies are simulated accurately except for edge cases, containers are not supported. The implementation of containers would require restructuring of the existing code and would not be trivial to do. In summary, ns-3 does not meet our requirements.

Dockemu is software implemented on top of ns-3, which integrates Docker containers as nodes into the simulation. The interconnection between the simulated node and Docker is carried out in two hops with a tap interface and a Linux bridge with a vEth-pair [29]. Consequently, since vEth-pairs have shown to deliver unreliable latencies, this approach might perform even worse due to another processing layer.

They all have different ways to simulate the network stack. Some of them factor in latencies introduced by the kernel; most of them allow adding delays to links. However, none of them let us integrate containers and, in addition to that, deliver accurate, low, and stable latencies.

4.1.2 Virtualized networking with low latencies

An important problem concerning virtual networking experiments is the lack of realism and stability due to reliance on software. Mininet [26] uses vEth pairs and network namespaces. As demonstrated by Wiedner et al. [4], virtual machines with VFs offer more stable and precise latencies compared to Mininet. The work presents a 618-fold improvement in worst-case latencies for two hop flows and a 1063-fold improvement for five hop flows. The higher the hop count, the higher the improvement, leading to the

conclusion that Mininet scales poorly compared to VFs. Up to the 50^{th} percentile in measured latencies, Mininet demonstrates latencies lower than HVNet. The results indicate that lightweight virtualization offers lower latencies when jitter and tail latencies are not considered. Further optimizations are necessary to stabilize the latencies.

Another problem is scalability. For utilizing real hardware, there are two options: passthrough and SR-IOV. With passthrough, the host transfers all control over a PCIe device to a guest who receives exclusive control. Research by Zhang et al. [25] suggests that passthrough is more performative for both containers and virtual machines. In particular, the overhead for the passthrough with containers over the native bare metal HPC benchmark is stated to be 9%. However, this approach is not scalable. A topology with n nodes requires n NICs, which is expensive. Furthermore, for larger n (such as $n > 30$), it is not feasible to connect so many individual NICs to a single host due to the absence of PCIe lanes and general physical logistics. In addition to that, by passing through one NIC for every virtual node, manual wiring is required to set up the links. We rule out pass-through as an exclusive way to interconnect virtual nodes.

By introducing hardware features such as SR-IOV, some overhead and jitter are eliminated, which occur when packets are processed exclusively in software. Previous work [4], [17], [25], [31] shows that the latencies introduced by SR-IOV are stable even for high percentiles such as 99.999 (5 nines) and the overhead is acceptable. As a consequence, SR-IOV is our first choice to achieve low-latency networking.

4.2 Containers as virtual nodes

For the realization of virtual nodes, two forms of virtualization come into question: heavy-weight virtualization with VMs, and OS-level virtualization with containers. The performance of VMs in the context of low-latency networking, optimization of hard and software stacks, and reduction of tail latencies was investigated by [4], [8], [14], [17], [25], [32]. In comparison, OS-level virtualization is not investigated as much, which is a reason why this thesis investigates containers. Table 4.1 provides an overview of the differences between VMs and containers and indicates the reasons why containers may have lower (tail) latencies.

VMs offer a higher degree of isolation due to the operation of an own kernel. Increased isolation comes with a trade-off in performance: Although there are modern hardware features, such as VT-x or AMD-v, which enable efficient handling of privileged instructions by opening a trap to the hypervisor [3], heavy-weight virtualization still inflicts a performance penalty. An interrupt is much more expensive to execute in a VM than on bare metal due to multiple context switches between virtual machine monitor (VMM)

	VM	Container
Kernel	own	shared with host
Paging	expensive	cheap
System call	expensive	cheap
Isolation	strong	lightweight with kernel-level features
Memory	usually pre-allocated	grows as needed
Hugepages	supported	must be created on host
Scheduler	own	by host
Start-up	seconds-minutes	milliseconds

Table 4.1: Comparison between VMs and containers

and VM [33]. Furthermore, the raw CPU compute performance is negatively affected. Ramalho et al. [34] measure 4.27 % worse memory performance for kernel-based virtual machine (KVM) VMs with NBench, 2.88 % worse integer performance and 1.44 % worse floating point performance. The same study finds that Docker containers for all NBench benchmarks are close to bare metal; in the worst case, Docker is 0.8 % worse than bare metal. In other benchmarks, containers consistently outperform KVM. Research by Li et al. [32] shows that the performance of VMs is inferior to that of containers for communication data throughput, computation latency, memory data throughput, and storage data throughput. In summary, related work demonstrates how containers achieve better synthetic benchmarks, which might translate into lower latencies because of more available computing power.

4.3 Factors influencing latencies

One reason for latencies is interrupts [1], [20], [35]. There are two types of interrupts: hardware interrupts, caused by physical I/O devices, and software interrupts, used for system calls [3]. The interrupt controller handles interrupt requests based on a priority. When it is time for an interrupt request, the controller issues an interrupt with the CPU. The CPU then stops working through the instructions and jumps to the interrupt handler, which defines the code that is executed due to the interrupt. Upon completion, the CPU returns to the location where it was interrupted and continues there.

Any interrupt of a CPU core engaged with packet processing will cause disturbance and induce unwanted delays. Therefore, our objective is to minimize interrupts by following the recommendations of Gallenmüller et al. [17] and Akkan et al. [35]. The suggestions of Gallenmüller et al. include kernel parameters to disable virtual cores, disable power

saving options, disable scheduling interrupts with the *nohz* kernel option, and various other configurations that affect latencies.

Gallenmüller et al. concluded that their tweaked *nohz* kernel provides better tail latencies than the real-time kernel. However, they test in a highly optimized environment. Since we introduce LXC containers that are built on different kernel and virtualization features, we consider both kernels *nohz* and real-time in our research.

Another factor is NUMA awareness. A PCI device is connected to a NUMA node. When memory or CPU cores from different nodes are used, data need to traverse NUMA nodes, which introduces latencies. Emmerich et al. [21] denote the added latency as 1.7 µs.

Chapter 5

Implementation

The practical implementation of this thesis uses HVNet [4] as a framework. In summary, we replace virtual machines in HVNet with containers, execute commands that require kernel interaction on the hypervisor instead of in the containers, and then run modified experiment scripts for measurements. The finished implementation can be found on GitLab [36]; branch `impl-lxc-container`.

Initially explained in Section 5.1 is the selection process for an implementation of LXC and basic interaction with containers. In Section 5.2 we integrate them into pos [16]. We mimic the VF setup of HVNet for containers; Section 5.3 presents the necessary configuration of a NIC for SR-IOV. Since it is not possible to initialize all resources for the usage of DPDK applications in a container, Section 5.4 structures the initialization into two steps: First, in Section 5.4.1, we execute commands that must run on the host. Second, in Section 5.4.2 we set up the container. Various options and optimizations for host and guests are highlighted in Section 5.5. Finally, we describe in Section 5.6 the integration process of our work in HVnet. Due to time and scope constraints, we are unable to port all features of HVNet to containers. Section 5.7 reviews features that are not tested or implemented.

5.1 Linux Containers

Linux Container (LXC) is a lightweight implementation for containers sponsored by Canonical. For isolation, LXC utilizes different kernel features, where the most important are CGroups and namespaces. LXC consists of userspace management programs, language bindings, and container templates.

Listing 5.1: Installation of the LXC userspace tools and utilities

```
1 $ apt-get -y install lxc debootstrap python3-lxc
```

Initially, we create container scripts with LXD. LXD is a wrapper around LXC, which adds a number of convenience and data center features [5]. However, after working with LXD, we decide to implement only LXC containers without the LXD wrapper. The reasons are that LXD adds complexity during setup and maintenance, and this thesis project does not require any of the additional features. Implementing LXD would add yet another layer of abstraction that could possibly fail, while providing almost no benefit to this thesis.

Since HVNet uses libvirt to manage virtual machines, we investigate the possibility of interacting with libvirt to orchestrate containers. This approach would make parts of the existing codebase reusable. For this purpose, RedHat offers the library *libvirt-lxc*. However, we cannot create containers with this bridge. The reason for this could be that the project was discontinued in 2015 [37] and was not tested with modern Linux distributions. After investigating the options mentioned above, we decided to use the LXC userspace tools and the related Python library.

5.1.1 Installation

Listing 5.1 shows the installation of LXC on Debian-based systems and the required dependencies. The *lxc* package installs the LXC userspace tools and three background services [38]:

1. *lxc-auto*: autostarts containers on boot
2. *lxc-net*: configures the lxcbr0 bridge interface (default option)
3. *lxc*: initializes the AppArmor profiles for unprivileged containers

The next package *debootstrap* bootstraps a Debian operating system that includes all libraries and binaries into a folder [39]. It is impossible to boot a container from an iso image file; therefore, the LXC userspace tools rely on debootstrap to create the root file system.

For the management of containers in code, there is the Python library *python3-lxc*. After carefully testing the library, we found it too unstable to use it for some tasks. First, container creation seemingly fails without pattern and without any error. Second, the command, which is supposed to add CGroup values to the configuration file, apparently ignores some CGroup version 2 values. Therefore, we write a wrapper class in which

Listing 5.2: Listing all installed containers and stats on the host

```
$ lxc-ls -f
NAME        STATE   AUTOSTART GROUPS IPV4          IPV6 UNPRIVILEGED
cnt-deb-00  RUNNING 0         -      172.16.142.2 -    false
cnt-deb-01  RUNNING 0         -      172.16.142.3 -    false
```

faulty commands are replaced by calls to the userspace tools or written in the container configuration file. Regardless of the faulty commands, we still utilize some working parts of the library because not reimplementing the entire library lowers the complexity of our code.

For this thesis, we are exclusively working with privileged containers. Unprivileged containers require more configuration (for example, AppArmor and user elevations [38]) but do not change the result of the experiments. Hence, we disable AppArmor on boot with the kernel parameter `apparmor=0`. AppArmor may be terminated during a running session with the command `aa-teardown`.

Upon creation of a new Debian system container with `lxc-create -n name -t debian` a configuration file is generated in `/var/lib/lxc/`. The container starts with `lxc-start -n name`. Stopping works analogously. The command to attach a shell to a running container is `lxc-attach -n name`. Listing 5.2 demonstrates the command to list the installed containers.

Using the aforementioned information, we create scripts that automatically install the dependencies, create containers upon request, and mirror the installed software and system settings of virtual machines. These scripts will be extended in the next chapters and finally integrated into HVNet.

5.1.2 Images for LXC: distrobuilder

Images of hosts and virtual machines, which power experiments with pos [16] are preconfigured with software and system settings. However, containers rely on different technologies, making it impossible to load images meant for physical hosts. As a consequence, we write a script that configures containers in the same way that images are set up for pos. This includes installing commonly used build tools and software, various network-related packages, configuring timezones, networking, secure shell (SSH) keys, and the SSH server. The containers are then in almost the same state virtual machines are in after booting.

Similarly, it is possible to create images for LXC. The LXC ecosystem offers distrobuilder, a program to create LXC and LXD images. One way to use distrobuilder is

Listing 5.3: Creating a container from an image

```
$ lxc-create -n cnt-deb-02 -t local -- --metadata /path/to/image/meta.tar.xz --
    fstree /path/to/image/rootfs.tar.xz
```

to first create a container, install the desired software on it, and then snapshot the state of the container as an image. Subsequently, we design a small wrapper script around distrobuilder. This script creates a Debian Bullseye container, installs the script previously drafted with pos defaults, and another user-defined script before finally packing the image. Distrobuilder outputs two compressed files: `meta.tar.xz` and `rootfs.tar.xz`. The meta file contains metadata and context information to correctly restore the image. The rootfs file contains the entire root filesystem, including all libraries and installed applications. Both files are mandatory to restore an image: Listing 5.3 shows how to create a container from an image.

As a result, we develop an easy way to image arbitrary software into LXC containers by providing an installation script.

5.2 Integration with pos

For reproducible experiments, we integrate LXC containers in pos. pos already supports creating and managing virtual machines. To simplify the setup, a container will pretend to be a virtual machine from the perspective of pos. Hence, containers will be named `$(hostname)-vmX`. Not all features of pos can be used with containers: boot parameters and requesting a boot image cannot be supported due to the way containers work. In Section 5.2.1 we configure the management network so that containers can be reached by SSH. Section 5.2.2 explains the necessary steps that allow pos to control the containers with commands, such as power-on, shutdown, and restart.

5.2.1 Management network setup

The management network's only purpose is to provide remote access to a container. It is not optimized for performance in any way. We set up a host-shared bridge that allows guests to interact with the network as if they were directly attached to it, without the container host in between.

We utilize a Linux bridge with the host management interface as master [40]; commonly referred to as a host-shared bridge. The bridge receives the same MAC address as the master interface. This bridge replaces the management interface that grants access to the host. Systemd-networkd handles DHCP for recent versions of Debian; hence, we

instruct it to request a DHCP lease for the newly created bridge interface. The bridge receives the same lease as the master, since the MAC address is the same. Finally, the IP address of the former management interface is deleted. The containers are then connected to the bridge with a vEth pair. All packets belonging, for example, to an SSH session now travel via the newly created bridge.

By default, LXC creates the masquerading or network address translation (NAT) bridge lxcbr0. This bridge assigns IP addresses from an internal subnet and forwards packets to clients [40]. Consequently, guests do not have unrestricted access to the host network, and the only way to reach containers is by forwarding ports. This implies that the containers will not be visible to pos. Since we are not using this interface, we are deleting it.

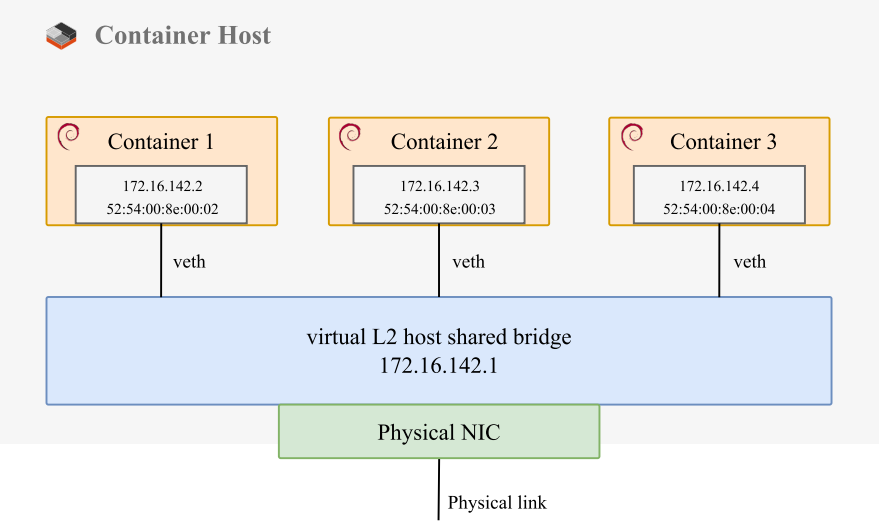


Figure 5.1: Overview of a host-shared bridge and special MAC addresses for containers

Each vEth interface inside a container must receive a special MAC address, so that pos [16] recognizes the container. The MAC address must match the following format with two parameters `52:54:00:{hex}:00:{num}`: `hex` is the third octet of the IP address of the host that can be queried with `hostname -I` and `num` is the number of the VM or container starting from two. Figure 5.1 demonstrates how the MAC addresses are set and which IP addresses each container finally receives. The MAC addresses `52:54:00:8e:00:0x` where x is the container number plus two are assigned to containers. The third octet of the host IP address `172.16.142.1` is 142 in base10, which translates into 8E in base8. When the container requests an IPv4 address with DHCP, pos assigns the next highest IPv4 address: If the host has the IP address 172.16.142.1, the first container receives 172.16.142.2, the second one 172.16.142.3 and so on.

Listing 5.4: Example override of the vBMC function for turning on a container

```
def power_on(self):
    container = lxc.Container(self.domain_name)
    if container.defined:
        if not container.running:
            container.start()
```

Listing 5.5: Registering a container with vBMC

```
$ vbmc add --username ADMIN --password blockchain --port 6001 container01
$ vbmc start container01
```

5.2.2 Control with vBMC

pos [16] controls nodes with IPMI commands. IPMI is a set of specifications that controls hosts out-of-band and is commonly implemented in the baseboard management controller (BMC). This means that even when not powered on, a user or program has the ability to take control of the device [41]. While this solution is adequate for hardware hosts, it is impractical for VMs: the BMC does not provide a way of targeting VMs running on top of a physical host. OpenStack's virtual baseboard management controller (vBMC) solves this problem. The vBMC is a lightweight simulation of a BMC in software, which interfaces with libvirt to forward IPMI commands to VMs.

At the time of writing this thesis, there is no comparable tool for interaction with LXC containers. Consequently, we rewrite Openstack's virtual BMC to forward the IPMI calls to LXC via the Python API and userspace tools instead of libvirt. The vBMC includes a wrapper class that implements methods such as *power_on*, *power_shutdown*, *get_power_state*. The Listing 5.4 demonstrates our override for the *power_on* function exemplarily. The other methods are implemented similarly.

Before installing the modified version of vBMC on the container host, we clone the repository and install its dependencies with pip. The project is built with an included Python script. Finally, we start the vBMC daemon, which forwards IPMI commands to the addressed container. Listing 5.5 demonstrates the registration procedure for an example container with the name `container01` and port 6001. After the registration, pos is able to control the container with IPMI.

Listing 5.6: Creating 8 VF on the interface eth0

```
1 $ echo 8 > /sys/class/net/eth0/device/sriov_numvfs;
2 $ sleep 5;
```

5.3 NIC configuration for SR-IOV

NICs, which are interconnected with a cable, must be configured specifically so that the packets travel back and forth between the two NICs. For that, we reuse the HVNet setup [4].

Listing 5.6 shows exemplarily how to create 8 VFs on the device `eth0`. Afterwards, a magic sleep is required, else the further configuration of the VFs fails.

Next, we allow unicast full promiscuous mode for VFs in the physical function, so that each VF has the ability to activate the option to read all ingress traffic [42], even traffic that is not addressed to the VF. Unicast full promiscuous mode for VFs requires a special NIC, such as an Intel X700 [4]. Furthermore, we set the VFs as trusted, which allows them to request promiscuous mode. We disable spoof checks, so that the VF does not discard packets with spoofed addresses. Since we rely exclusively on IPv4, we disable IPv6 for each VF. Finally, in Listing 5.7 we assign the VLAN ID 401 to the fourth VF of the interface `eth0`.

Listing 5.7: Setup VLAN 401 for the fourth VF of the interface eth0

```
1 $ ip link set eth0 vf 3 vlan 401;
```

5.4 DPDK in containers

In some of our experiments we rely on a layer 2 packet forwarder from the libmoon [22] packet generator. libmoon runs on top of DPDK, another library for fast packet processing. Running DPDK applications in a container requires a special setup, since it is not possible to load kernel modules or to create hugepages in a container. In summary, we prepare all kernel-dependent interactions on the container host and pass through the initialized resources to the container. We then rewrite the libmoon scripts to skip resource initialization in the container.

5.4.1 Host preparation

We derive the next steps from Krishnamurthy's guide published in the DPDK mailing list [43] to run DPDK in LXC containers. In the script `prepare_host_forwarder.sh`,

Listing 5.8: Binding a network interface card to the igb_uio driver

```
$ modprobe uio
$ insmod igb_uio.ko
$ python dpdk-devbind.py --bind=igb_uio "$DEVICE"
```

Listing 5.9: Major and minor IDs for device files

```
$ ls -la /dev/uio*
crw------- 1 root root 242, 0 Jul  8 14:01 /dev/uio0
crw------- 1 root root 242, 1 Jul  8 14:01 /dev/uio1
```

we gather all steps. Furthermore, container-related configuration is added to the script `prepare_container_host.py`.

First, the NIC, which will be used in DPDK, must be bound to the `igb_uio` driver. This driver is supplied by the DPDK framework and is compiled with `make`. Since `igb_uio` is built on top of the `uio` kernel module, `uio` has to be loaded. Listing 5.8 demonstrates the process of binding interfaces to `igb_uio` with the help of `dpdk-devbind`, a script supplied by DPDK. To bind a VF to the driver supported by DPDK, it is important to install the IAVF driver [44]. Otherwise, it is not possible to correctly interact with VF.

Second, DPDK requires preallocated hugepages. The libmoon library supplies a script to initialize 512 hugepages of 2048 kB on all NUMA nodes. In case multiple containers are running DPDK applications like libmoon and have cores from the same NUMA node assigned, the amount of allocated hugepages in the libmoon initialization script has to be adjusted.

At that point, the host is set up. Next, we pass through the initialized resources to a container. By binding the network interfaces to `igb_uio`, device files in `/dev/uioX` are created. These device files are assigned with CGroups to a container. For that, the major and minor ID of the devices must be known. Listing 5.9 reveals the major ID 242 and the minor ID 0 respectively for the two created device files.

In addition, the device nodes must be mounted or created in the container. There are two options for doing so: during runtime with the `mknode` command or by mounting the device file. We opt for the second option. The hugepages are mounted in the same way. Since binding to `igb_uio` creates files in `/sys` which are necessary to interact with the NICs, additional options are required according to Stéphane Graber in a Github issue [45]. With the LXC configuration shown in Listing 5.10, DPDK is capable of taking control of NIC within a container. The host setup is now complete. Next, we configure the containers to utilize the resources passed through.

Listing 5.10: LXC configuration file content for passing through device files

```
lxc.mount.auto =
lxc.mount.auto = proc:rw sys:rw
lxc.cgroup2.devices.allow = c 242:0 rwm
lxc.mount.entry = /dev/uio0 dev/uio0 none bind,create=file
lxc.mount.entry = /dev/hugepages dev/hugepages none bind,create=dir 0 0
```

5.4.2 Container preparation

Running libmoon inside a container requires some adjustments. By default, libmoon assumes to run on a bare metal or virtual machine. We have to make two adjustments to the `build.sh` script, which compiles libmoon and its dependencies and sets up the host:

1. Exclude the call to the `bind-interfaces.sh` script, which binds the network interfaces to the `igb_uio` driver. This step is not necessary since the passed-through device files are already available.
2. Exclude the compilation and loading of the `igb_uio` kernel module, which is not possible inside a container.

After compilation, libmoon is almost ready for experiments. One issue requires modification of libmoon: HVNet [4] assigns CPU cores to containers with NUMA awareness. However, the ingress and egress NICs are connected to different NUMA nodes, which causes libmoon to throw an error and terminate immediately. We develop a workaround for this issue: a new configuration option called `forceNumaNode` that overrides the internal check regarding NUMA optimality in libmoon.

Finally, we configure the DPDK library used in libmoon to match our requirements. Libmoon is configurable through the file `dpdk-conf.lua` where the arguments are passed to libmoon and DPDK. We configure four options:

forceNumaNode: Our added option; skips the NUMA assignment and verification process, and forces a specific NUMA node.

cores: This option assigns specific cores for packet processing. A single forwarder receives three CPU cores: one for OS services, handling interrupts, and other processes unrelated to packet processing, and two for processing the transmitting and receiving of packets.

–socket-mem: By default, libmoon will allocate all available hugepages [46]. This behavior makes running multiple forwarders impossible since there are no more hugepages

for the second forwarder available. With the option `-socket-mem` it is possible to define how many 2048 kB hugepages should be allocated to this instance of DPDK for each NUMA node.

–file-prefix: Two or more containers would interfere with the hugepages of each other despite the limit set with `-socket-mem` [46]. The option `-file-prefix` assigns a prefix to the allocated memory to avoid conflicts.

5.5 Optimizations

We apply a number of optimizations that improve performance and minimize interrupts on packet processing cores. Section 5.5.1 introduces our CGroup configuration, which affects the allocation of hardware resources to containers. In Section 5.5.2 we optimize containers to exclude processing cores from housekeeping tasks. Finally, Section 5.5.3 describes the optimizations applied to the host.

5.5.1 CGroup setup

Hardware resources are restricted for each container with CGroups. HVNet returns a blueprint that assigns cores and memory, including the NUMA socket, to each container. We assign the resources according to the blueprint. For that, we utilize specific CGroup options:

cpuset.cpus: The value is a comma-separated list or a range of core numbers. Each of the cores is assigned to all processes that inherit this CGroup option [47]. This option alone does not restrict the parent to schedule processes on the assigned cores.

cpuset.cpus.partition = 'root': The cpuset subsystem of CGroups offers another option, partitions, which offers a native solution to the problem of depriving CPU cores from other CGroups [47]. A partition acts as its own tree. The resources assigned to the subtree are only present in this subtree, but not in the parent. For example, a LXC container with the options `cpuset.cpus.partition = 'root'` and `cpuset.cpus = 3-5` is granted exclusive access to the cores 3-5, because the parent CGroup node is deprived of the cores 3-5. This can be verified by checking the value of `cpuset.cpus.effective`.

cpuset.mems: This option limits from which NUMA nodes memory can be assigned to processes. We exclude this option for experiments where we face the problem described in Section 5.4.2: libmoon requests memory from a different NUMA node than the

Listing 5.11: Configuring housekeeping tasks of a container to run only on the first CPU core

```
$ systemctl set-property user.slice AllowedCPUs=7;
$ systemctl set-property system.slice AllowedCPUs=7;
$ systemctl set-property init.scope AllowedCPUs=7;
```

Listing 5.12: Configuring a new slice with access to all CPUs for DPDK

```
$ cat <<EOF > /etc/systemd/system/libmoon.slice
[Slice]
AllowedCPUs=7-9
EOF
$ systemctl daemon-reload
$ systemd-run --slice=libmoon.slice echo "Hello␣Slice"
```

one from which it has cores assigned. This requires us to overwrite the NUMA node assignment in libmoon.

memory.max: Hard limit of the maximum assignable memory in bytes. If more memory is requested, the processes in the group are terminated by the out-of-memory killer [47]. Containers do not pre-allocate memory; instead, they only request memory when needed, leading to more efficient resource utilization.

5.5.2 Container optimization

A container runs its own instance of systemd. Systemd again creates slices and has scope units, which are permitted to use any core which is assigned to the container. We restrict this behavior so that these services only run on the first CPU core that is not involved in the packet processing. Assuming that the container is assigned CPU cores 7-9, we restrict systemd services to only use core 7 as demonstrated in Listing 5.11. Furthermore, there are certain processes of pos [16] that run in the `user.slice`. To eliminate potential context switches and interrupts, we restrict the cores assigned to this slice.

Since we restricted the `user.slice` to one core, it is no longer possible to run the forwarder. Instead, a new slice is created that has access to all cores. The Listing 5.12 creates a new slice called `libmoon.slice`, reloads the system daemon so that systemd creates the slice, and finally, we run an example program in this slice. Another unsuccessful approach is to assign the two processing cores 8 and 9 and `cpuset.cpus.partition = 'root'` to the `libmoon.slice`, which deprives the container of the aforementioned cores. However, a DPDK application requires another thread to handle occasional interrupts for link updates of the poll-mode driver [48]. When assigning only two cores, this thread cannot be scheduled and DPDK fails to start.

Listing 5.13: Kernel parameters of the container host

```
1 $ cat /proc/cmdline
2 mce=ignore_ce tsc=reliable idle=poll nohz=on audit=0 amd_iommu=off nosmt console=
      ttyS0,115200 apparmor=0 nohz_full=8,9,10,24,25,26 rcu_nocbs=8,9,10,24,25,26
      kew_tick=1 irqaffinity=0 intel_pstate=disable nmi_watchdog=0 nosoftlockup
      rcu_nocb_poll random.trust_cpu=on intel_idle.max_cstate=0
```

5.5.3 Host optimizations

The kernel parameters are used to configure the operating system for specific tasks. We reuse the propositions of related work [4], [17], [35]: These kernel parameters reduce the amount of timer activity on the cores, block read-copy update (RCU) from the processing cores, and disable power saving features that might affect performance, such as cstates or pstates. After integrating the parameters into the project, we present our configuration in Listing 5.13.

Some minor adjustments are necessary compared to the propositions of related work. The IOMMU must be disabled for containers, since it is incompatible with our driver stack. In addition to that, we do not require the IOMMU. Furthermore, the option `isolcpus` is excluded, as it interferes with the CGroup cpuset subsystem.

5.6 Integration with HVNet

Here we present details of the integration of containers in HVNet. This includes new program parameters, new configuration parameters, boot parameters, and finally container creation.

5.6.1 Program parameters

The parameters change the behavior of HVNet. When executing the Python `setup.py` script, parameters may be defined. We add more parameters to enable a more fine-grained control of LXC.

-lxc or –**enable-lxc**: Use LXC instead of VMs. Adjustments of the setup scripts might be necessary. Not all the functionality of HVNet has been tested. Enabling this option switches the VM setup scripts in `setup.py` to the container ones.

-lxc-di or –**lxc-disable-isolation**: Indicates whether our implemented host and container optimizations should be applied. This option can be used to test performance by letting LXC handle core assignments, the impact of NUMA awareness, and our core pinning and isolation mechanisms.

5.6.2 Configuration parameters

Initially, we added more options to the HVNet configuration. With this json-formatted configuration file, the user defines a topology, the scripts to run on each node, and the experiment setup. To integrate the previous work more efficiently, we add an option to define a script `lxc_host_setup` that runs on the container host. Every node may define its own host setup script, so that each node can add more hugepages, or various other changes it needs to be executed on the host. For the DPDK forwarder nodes, we may use the script `prepare_host_forwarder.sh` prepared previously.

With the second new option, the user can define the path to a LXC image. The path must be accessible from the host that executes HVNet and point to a folder containing the two files `meta.tar.xz` and `rootfs.tar.xz`, which define a LXC container. HVNet copies the folder with rsync to the container host, where the containers are instantiated with the images. Based on the json configuration file, HVNet derives a configuration object that provides detailed information about the topology and the host, including the NUMA-aware assignment of memory and CPU cores.

The program parameters are defined in the configuration object used internally in HVNet. For example, the variable `lxc` tells whether LXC is enabled or not. Furthermore, the Boolean variable `lxc_disable_isolation` indicates whether our isolation mechanisms should be omitted.

5.6.3 Boot parameters

We adjust the HVNet boot parameters. The IOMMU must be disabled. On the one hand, we do not have VMs which would require the IOMMU, and on the other hand, the creation of VFs fails with our setup when the IOMMU is enabled. Next, we disable CPU core isolation with the parameter `isolcpus`. According to [49], `isolcpus` are deprecated and instead the cpuset subsystem for CGroups should be used - just like we do.

5.6.4 Container creation

To match the setup of VMs in HVNet, we create two scripts: one to install all dependencies related to LXC and one to create the actual containers and VFs. With the program parameter `-lxc` we instruct the `setup.py` script to exchange the VM files for the corresponding container files.

The **prepare_dependencies.sh** script has the following purposes:

- Binding RCU tasks to the first CPU core. This will prevent the kernel from executing an RCU task on one of the cores assigned to a container.

- Updating the package database and installing the required packages.
- Installing NIC drivers: i40e version 2.17.15 and ice 1.7.16.
- Installing our modified version of the virtual BMC and running the daemon.

The **prepare_container_host.py** script sets up containers and all other related optimizations while respecting the values provided by the HVNet configuration object:

- Creating the management interface
- If necessary, creating VFs and setting them up accordingly
- Creating containers. Passing through the VFs and NIC device files. Applying optimizations to the container. If no image is provided, copies over SSH keys and executes the `prepare_container.sh` script, which installs the expected defaults by pos. Finally, registering the container with vBMC.

After these two scripts are executed, the container topology is set up and wired according to the configuration. Next, the setup scripts are executed, and finally the experiment is starting.

5.7 Limitations

Not all functionality of HVNet is implemented and tested, since many of the existing scripts require adaptation to be used in a container. A list of features that do not work:

- Node type *switch* - the OpenVSwitch scripts require adaptation since they load kernel modules. We started the adaptation of the setup scripts for containers, but even after extracting the loading of the kernel module, the program would crash.
- Link type *virtual* - with VMs, the vEth-pair is wrapped in the VirtIO driver, which virtualizes a PCIe device and therefore can be bound to DPDK. With LXC the interface is assigned to a namespace. It is impossible to bind it to the DPDK compatible device driver, since the container is aware that it is not a fully featured PCI function.
- atsbpr configs: untested

We have shown that it is possible to run low-latency networking software such as DPDK in a container and integrate LXC into the testbed infrastructure with pos. Containers are viable for low-latency networking from the perspective of tooling, which answers the first part of **RQ1**. The second part and the remaining research questions are analyzed by taking measurements in the next Chapter.

Chapter 6

Evaluation

This Chapter describes all measurements conducted within the scope of this thesis. Different configurations, kernel parameters, and CGroup options are tested for their impact on latency. In Section 6.1 we present our testbed and measurement methodology, so that our experiments can be reproduced. Our measurements, evaluation and comparison with related work are conducted in Section 6.2. For our measurements, we utilize a setup with a single container or VM, which forwards packets. Although we tested the implementation of SR-IOV for larger topologies involving multiple containers, we could not compare the results with VMs. The driver for VFs is malefunctioning in a VM and causes frequent crashes, so it is not possible to take measurements.

6.1 Setup

For all experiments, we utilized three hardware hosts. Containers are created on the DuT, the LoadGen generates packets with MoonGen [22] and the Timestamper utilizes the MoonSniff framework to track the latencies for every packet.

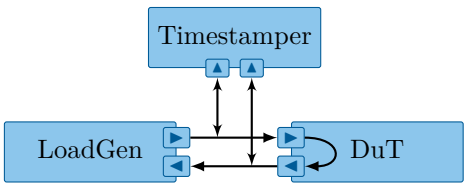


Figure 6.1: Experiment setup

The LoadGen is equipped with two Intel 82599ES 10 GbE, each connected to an Intel X710 10 GbE NIC on the DuT. Figure 6.1 visualizes the setup: The container network

Host	CPU	Memory	NIC
DuT	AMD EPYC 7551P	128 GB	2x Intel X710 10 GbE
LoadGen	Intel Xeon Silver 4116	192 GB	2x Intel 82599ES 10 GbE
Timestamper	AMD EPYC 7542	512 GB	2x Intel E810-C 25 GbE

Table 6.1: Hardware specifications of the testbed

on the DuT forwards packets from one interface to another so that it effectively creates a loop. MoonGen generates UDP packets on the LoadGen and sends them through its egress NIC to the ingress of the DuT. There they traverse a network of containers until the egress is reached, which is again linked back to the LoadGen. The Timestamper is attached to the connections between LoadGen and DuT with optical splitters. An optical splitter taps into the connection so that the Timestamper receives all packets that are exchanged. No latency is added, except for the propagation delay of the fiber cable, which is accounted for [17]. The Timestamper measures the latency for a packet by subtracting the measured timestamps of the two packets with the same unique identifier. Consequently, the Timestamper creates pcap files which are evaluated with another set of scripts by HVNet. By using commodity hardware such as the Intel 82599ES to timestamp packets at the LoadGen, only 1 kpkt/s can be timestamped. As a consequence, we can analyze tail latencies more effectively with optical splitters.

All experiments are carried out with a minimum packet size of 64 B. This translates into 1.52 Mpkt/s for approximately 1 Gbit/s with framing and 6.24 Mpkt/s for approximately 4.4 Gbit/s. The LoadGen cannot generate more than approximately 6.24 Mpkt/s at the smallest packet size [22].

For our experiments, we tested various OS versions. HVNet and other related work were tested with Debian Buster, which is why we included it in our tests. In contrast to that, our implementation is based on Bullseye. It was not possible to backport our work to Buster due to the default version of CGroups being version 1, and the package of LXC differing by a major version. As far as possible, we test with Buster and Bullseye on the DuT. The LoadGen always operates with Debian Buster, Linux kernel 4.19. For the forwarder, which runs in a container on the DuT, we use different versions of libmoon. Related work was tested with the original version [50] of libmoon, commit `2dbbcd9`, which is why we also use this version for our tests with Debian Buster. Unfortunately, this version of libmoon does not compile on Bullseye, which is why we use a modernized version of libmoon [51] from the branch `dpdk-20.08` on Bullseye. For all experiments on Bullseye, the commit `f2a859e` is checked out, except for the LXC experiments, where we require our NUMA patch, which is built on top of the commit

`a6525dd` of the mentioned branch. For the LXC experiments, we created an image with libmoon preinstalled. More figures for every conducted experiment can be found in the result repository [52].

6.2 Base experiment

The topology of the base experiment is simple: The ingress and egress NICs are passed through to a container or VM that forwards packets between the interfaces with the libmoon l2fwd script. Libmoon is the library underneath MoonGen that provides most of the logic. This experiment establishes a baseline using a single container without added complexity, so that other possible factors that influence the measurements are eliminated. Subsequently, we formulate recommendations for low-latency networking with containers.

6.2.1 Kernel with nohz patch

This experiment used a *nohz* enabled kernel according to the recommendation of related work [17]. We compare five different setups: VM with Debian Buster and Bullseye, bare metal (physical) with Buster and Bullseye, and LXC with Bullseye. The results are organized as follows: the results for 1.52 Mpkt/s are presented in Figure 6.2, and for 6.24 Mpkt/s in Figure 6.3. Both Figures are high dynamic range (HDR) diagrams with two logarithmic axes. For each percentile on the x-axis, the graph shows the recorded latency in µs. For example, in Figure 6.2, the 99^{th} percentile for Buster experiments are roughly at 5 µs, which means that 99 % of the recorded latencies are lower than 5 µs. The x-axis starts within the single-digit µs range and increases up to 1000 µs. The HDR plot highlights the tail latencies better than regular plots, where the outliers are barely visible.

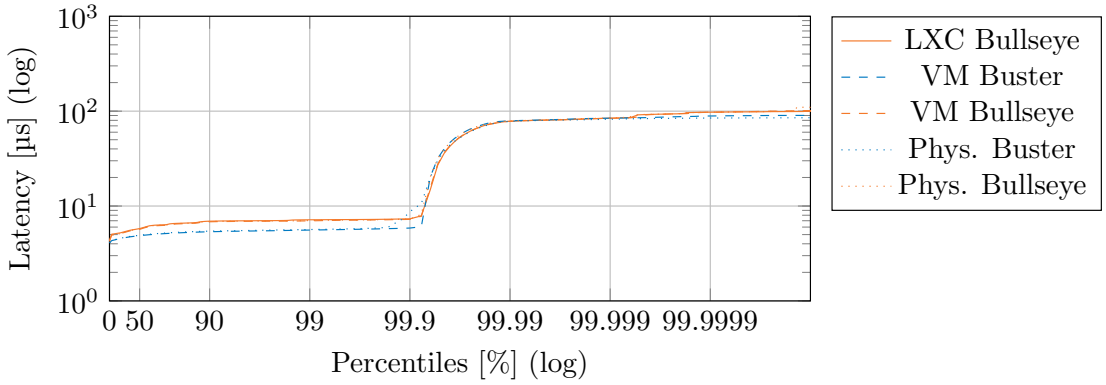


Figure 6.2: Results of the base experiment with 1.52 Mpkt/s and *nohz* kernel

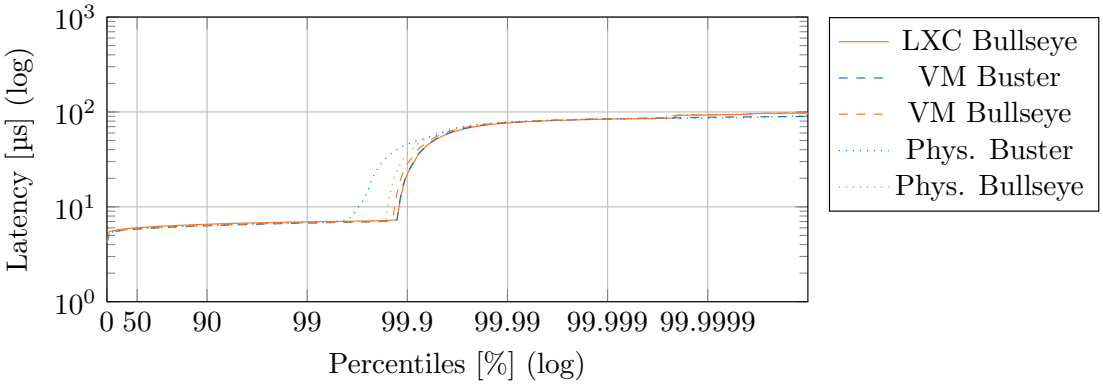


Figure 6.3: Results of the base experiment with 6.24 Mpkt/s and *nohz* kernel

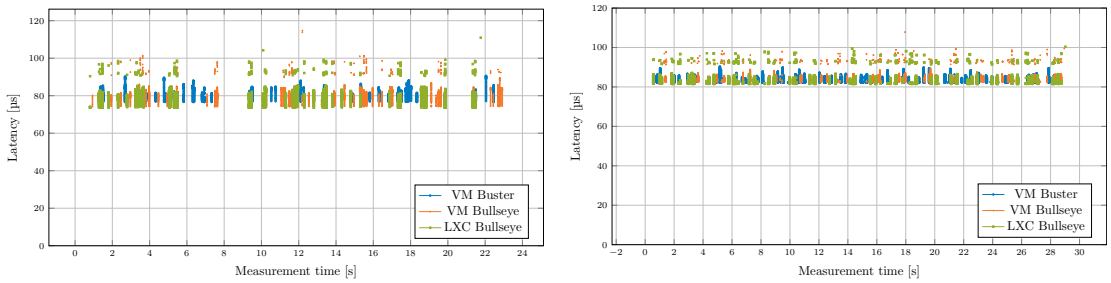
Depending on the packet rate, we observe a sharp increase in latencies in the 99.9$^{\text{th}}$ percentile. Gallenmüller et al. [17] assume that it is caused by the measurement application; however, they did not investigate further. For higher packet rates, this sharp increase moves further to the left of the graph. The early increase in latencies at the 99.9$^{\text{th}}$ percentile in Figure 6.2 for the bare metal Buster experiment is most likely not a consistent phenomenon due to the shape of the other measurements. We did not repeat this experiment for a cleaner data set.

The small variance in the results until the 99.9$^{\text{th}}$ percentile is explained by measurement inaccuracies. However, the trend that bare metal performs worse than both virtualization technologies in the 6.24 Mpkt/s experiment was persistent. Gallenmüller et al. [17] reported similar behavior for higher packet rates. Although they only measured up to 120 kpkt/s, the tendency for VMs to perform better was observed for higher packet rates.

It is also remarkable that we observe a difference in latencies between Debian Buster and Bullseye. The difference in the experiment with 1.52 Mpkt/s is 1.4 µs for the 99$^{\text{th}}$ percentile and remains roughly the same for other percentiles until the sharp increase occurs - independent of virtualization. We could not identify a reason for this observation. In the experiment with a higher packet rate of 6.24 Mpkt/s there was no difference measured outside the margin of error. Based on our collected data, we expect that LXC on Buster performs better than our Bullseye implementation.

To further analyze the difference between Buster and Bullseye, Figure 6.4 shows the 5000 worst-case latencies for a VM on Buster, Bullseye, as well as LXC. The latencies of LXC are in line with the results of a VM on Bullseye. There is no significant difference in the distribution of latencies: for both, the majority of the worst-case latencies are around 80 µs; we then observe a small gap where no latencies are recorded. Finally,

a few outliers are again recorded. This behavior was also shown by Gallenmüller et al. [17]. Surprisingly, we did not record this behavior for Buster, although Gallenmüller et al. tested with Buster.



(a) 1.52 Mpkt/s (b) 6.24 Mpkt/s

Figure 6.4: 5000 worst-case latencies for the base experiment with *nohz* kernel

We identify an issue in our setup that introduces latency. The ingress NIC is connected to NUMA node 1, the egress NIC to NUMA node 3; we assign CPU cores from node 3 and memory from node 3. According to Emmerich et al. [21], this setup adds 1.7 µs latency due to crossing NUMA boundaries. A more optimal scenario requires CPU cores from node 1. However, this would require further modification of the libmoon forwarder, which we reserve for future work. In Figure 6.2, we measured 5.3 µs for the 99.9^{th} percentile with Buster. Compared to that, Gallenmüller et al. [17] measured roughly 3.3 µs at a much lower packet rate for the same percentile. Consequently, unnecessary NUMA crossings could be an explanation for the observed difference compared to related work.

All things considered, we answer **RQ1** positively. Containers are viable for low-latency experiments. To answer **RQ2**, we showed that containers perform identically, within the margin of error compared to VMs when using the same OS and kernel. Cross-comparing Debian Buster and Bullseye reveals a difference of 1.4 µs. However, this difference is agnostic to virtualization and likewise measurable for bare metal.

6.2.2 Low packet rates

While experiments with high packets per second are more useful for observing hard and software bottlenecks, we still test lower packet rates. By lowering the data rate to 0.01 Mpkt/s, we observe worse latencies than in previous experiments with higher packet rates. Figure 6.5 shows the latencies with LXC and the libmoon forwarder for 0.01 Mpkt/s, 0.06 Mpkt/s, 0.12 Mpkt/s, and 1.52 Mpkt/s, which resembles the packet rates of related work [17]. It is remarkable that with increasing packet rates, the latencies improve. The latencies for the lower rates are independent of virtualization or OS. We

measured nearly identical latencies for the lower packet rates on Buster and Bullseye, and on containers and VMs.

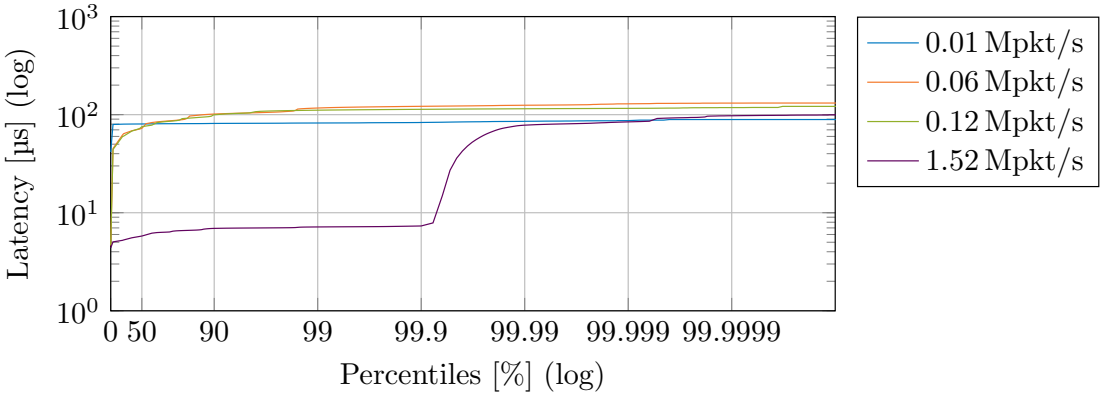


Figure 6.5: Low packet rates demonstrate worse latencies than higher ones

To further pin down the problem, we repeat the experiment with a different layer 2 forwarder. For that, we install DPDK version 22.11 and run the integrated example program `dpdk-l2fwd` on Debian Bullseye. However, the observed latencies did not improve. We conclude that this problem is related to our setup, but we cannot identify the root cause. Subsequently, we focus on experiments with 1.52 Mpkt/s and 6.24 Mpkt/s where we observe low latencies for most packets.

6.2.3 The impact of recording interrupts

HVNet provides a Python script that records the interrupts on VMs and the hypervisor. During our measurements, we noticed that the interrupt recording (IR) script was always executed on the hypervisor, but not on the VMs, regardless of the program parameter. This was the reason we initially observed large spikes in latencies for the containers, as shown in Figure 6.6. Since containers share the kernel with the host, running the interrupt recording script on the host affects the tail latency measured inside a container. In the VM we did not measure a difference.

In addition to the experiments with and without interrupt recording, we present in Figure 6.6 measurements for a real-time kernel. All measurements are performed with 1.52 Mpkt/s. A Linux kernel with the *nohz* patch cannot perform consistently for higher percentiles due to interference from other tasks. For systems which rely on the completion of a task, or else money or even humans will be lost, this is insufficient. A real-time kernel is used for scenarios where missing deadlines is unacceptable. It guarantees that a process finishes its calculation in a certain amount of time without being affected by the state of the remaining system. A real-time kernel completely mitigates outliers

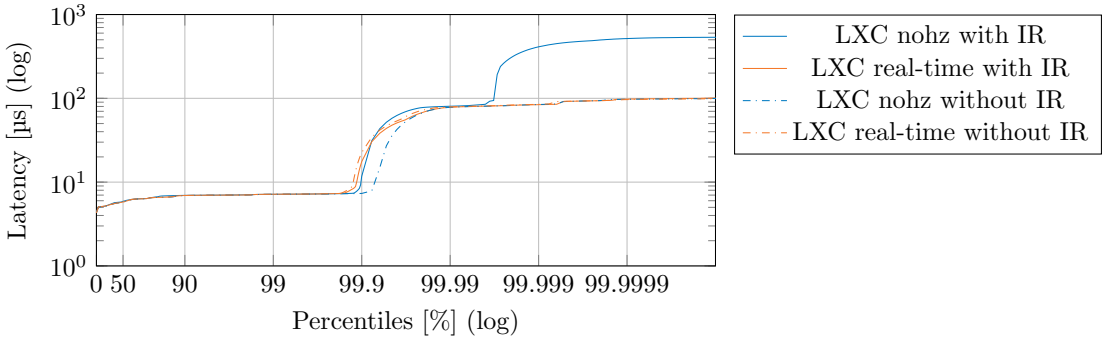


Figure 6.6: Impact of interrupt recording on LXC: real-time and *nohz* kernel

for higher percentiles, as can be seen in Figure 6.6. Even in the 99.999$^{\text{th}}$ percentile, containers achieve the same stable latencies of around 100 µs in the worst case, just like a *nohz* enabled kernel. More stable latencies come with a price: The latencies in the 99.9$^{\text{th}}$ percentile are slightly worse. The sharp increase with the *nohz* patch occurs after the 99.9$^{\text{th}}$ percentile, while with a real-time kernel it appears sooner. Finally, the raw data reveal that not a single data point with significantly higher latency was captured.

Therefore, the isolation for containers is inferior to that of VMs. Tasks can be scheduled on the cores assigned to a container even though we removed the cores from the host with CGroups. A reason why this is possible is scheduler load balancing. Scheduler load balancing is a mechanism of the kernel that moves tasks between cores. The Linux admin guide [53] specifically recommends to disable load balancing on real-time systems to minimize system overhead. The reason why this is not an issue for VMs is that the kernel parameter `isolcpus` removes the isolated cores from the load balancing mechanism. In CGroups v1 there is a flag to disable scheduler balancing; however, in v2 this flag is not yet implemented. Nevertheless, kernel developers are working to implement an option for the partition parameter of the cpuset subsystem that mimics the behavior of `isolcpus`. The last message in the discussion thread for the proposed patch was sent in June 2022 [54].

This insight answers **RQ3**. When running additional programs on the hypervisor or container host, latency spikes infrequently in a container. In contrast, a VM is not affected by our interrupt recording program running on the hypervisor. To reduce the difference in tail latencies, we recommend a real-time kernel. As demonstrated, a real-time kernel completely mitigates spikes in tail latencies.

6.2.4 Kernel parameter isolcpus

The kernel parameter `isolcpus` is used to exclude a core from the scheduler and load balancing, preventing processes from being scheduled on an isolated core. The official Linux documentation recommends not to use this parameter and instead to rely on the CGroups cpuset subsystem [49]. However, many older projects still rely on this parameter, so we tested the behavior of `isolcpus` in combination with CGroups.

Although the parameter excludes a core from scheduling, a process may still be scheduled on an isolated core if there is a relevant CGroup option, or the `sched_setaffinity` system call is used [35]. Furthermore, kernel tasks can be scheduled on an isolated core. An experiment with the `isolcpus` kernel parameter confirms these assertions. We isolate all cores that are passed to the container with the parameter `isolcpus=24,25,26` and keep the housekeeping threads exclusively on core 24 inside the container. No difference in latencies is observed.

In another attempt to investigate this parameter further, we isolate only the CPU cores that are processing packets and omit the housekeeping core for the container. Therefore, we use the parameter `islcpus=25,26`. Regardless, we do not record any difference. We conclude that the kernel parameter `isolcpus` in combination with CGroups is not relevant for further investigation.

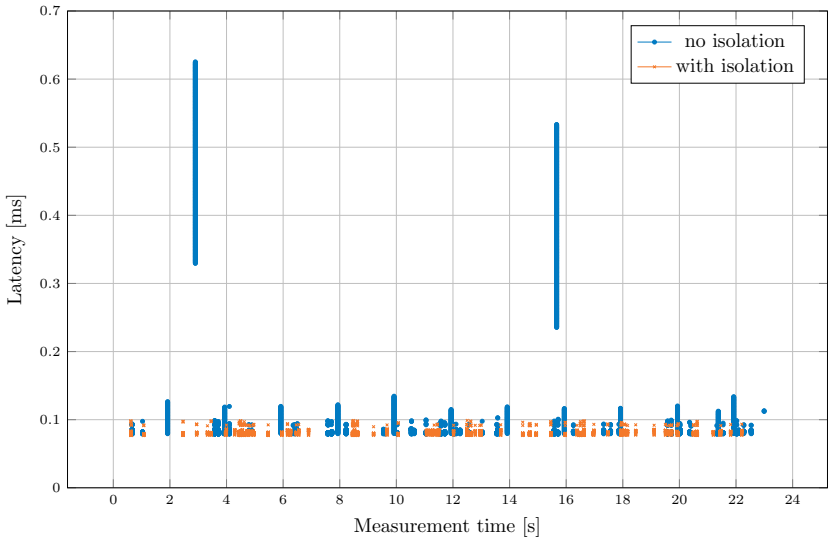
6.2.5 Disabling isolation optimizations

The experiments in this Section investigate the impact of previously established core isolation on network latencies. Without the CGroup and slice isolation efforts, the containers will have access to any CPU core. Furthermore, the host system also has access to all cores. Any thread on both the host and container could be scheduled on any core. This includes housekeeping threads, interrupts, and any other program running on the computer. The forwarder application tries to optimize the choice of CPU cores to match the NUMA nodes, where the NICs are attached. The results can be reproduced by setting the `-lxc-disable-isolation` flag when launching HVNet.

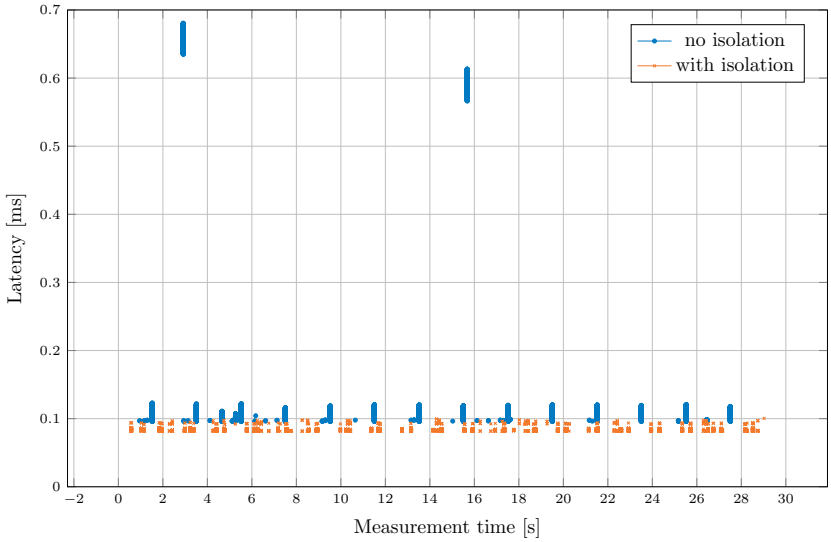
This experiment runs with a *nohz* enabled kernel with LXC. We plotted the 5000 worst-case measurements in Figure 6.7 and 6.8 for four events: our isolation mechanisms are disabled and then enabled for a *nohz* patched kernel and a real-time kernel. In all experiments, the worst-case latencies are much improved when the cores are isolated. The large spikes in latencies in Figure 6.7a and 6.7b could be caused by an interrupt or by another thread scheduled on the processing core. It is notable that both spikes occurred at the same timestamp within the experiment. The real-time kernel makes the

latencies more predictable. In Figure 6.8a and 6.8b we can see that all packets had in the worst case a latency of slightly more than 120 µs regardless of the packet rate.

In conclusion, our previously developed core pinning and isolation mechanisms reduce the worst-case latencies. We recommend using the presented optimizations for reliable low-latency networking with containers.

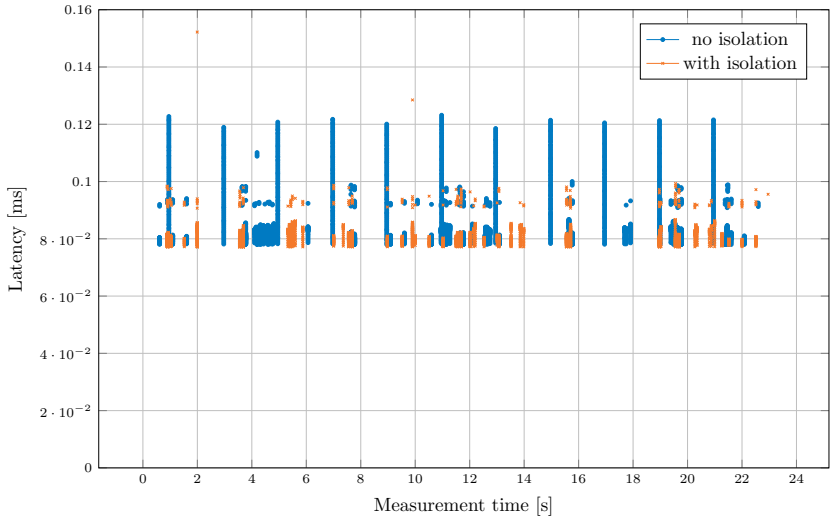


(a) *nohz* - 1.52 Mpkt/s

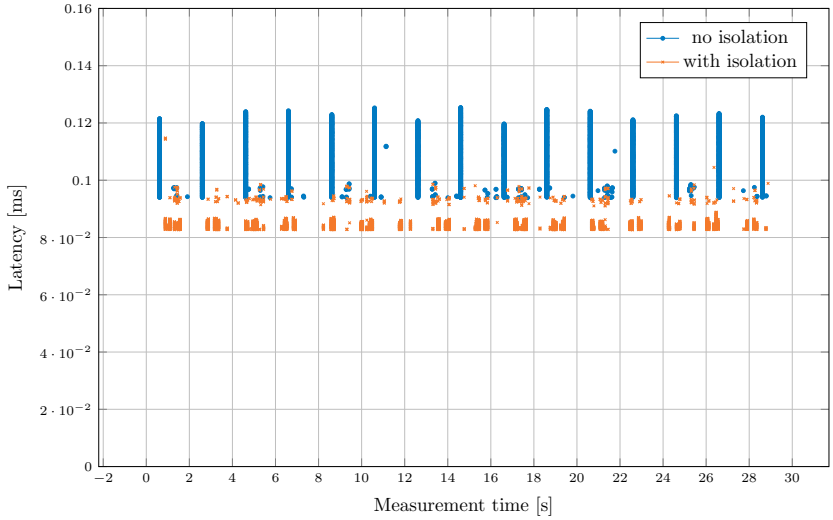


(b) *nohz* - 6.24 Mpkt/s

Figure 6.7: 5000 worst-case latencies with and without isolation for a *nohz* kernel



(a) real-time - 1.52 Mpkt/s



(b) real-time - 6.24 Mpkt/s

Figure 6.8: 5000 worst-case latencies with and without isolation for a real-time kernel

Chapter 7

Conclusion

This thesis aimed to evaluate the viability of containers for low-latency networking. Based on our implementation and latency measurements, it can be concluded that low-latency software frameworks like DPDK can run in a container and that the measured network latencies of containers and VMs are identical. However, the isolation of containers is not as effective as the one of VMs: latency spikes occur when running other software on the container host. This behavior can be mitigated by using a real-time kernel. Finally, we identified a difference in average latencies of 1.4 µs between Debian Buster and Bullseye - independent of virtualization.

Initially, we selected LXC as our reference implementation for containers and integrated it into the testbed, the orchestration service pos [16], and incorporated it into HVNet. We presented the setup necessary to pass through PCI devices and VFs to a container. By attaching VLAN IDs to VFs, we mimic HVNet and can create arbitrary network topologies on a single host. Various optimizations improving the isolation of the processing cores were applied, and we showed that these optimizations improved tail latencies.

With our measurements, we made numerous observations. First, virtualization can outperform bare metal. Second, we identified spikes in tail latencies when running other software on the container host. Third, a real-time kernel eliminates these spikes. Fourth, the latencies of containers are identical compared to VMs. Fifth, there is a difference of 1.4 µs in network latencies between Debian Buster and Bullseye. Sixth, measuring latencies accurately with high precision is not a trivial problem and requires careful consideration and a special hard and software stack.

As initially explained, edge computers collect and analyze sensors data in an industrial environment. On the basis of our research, we recommend containers as a virtualization

solution. However, threads running on the container host can affect the tail latencies of a real-time application inside a container. For that reason, containers fulfill only the requirements of a soft real-time system but not a hard real-time one. Heavy machinery or lives must not depend on reliable latencies of containers. Future work might be able to resolve the remaining issue so that containers may be deployed in critical systems.

Many adaptations, extensions, and experiments have been left to future work due to time and scope constraints. More research of containers in combination with low-latency networking could follow up on this thesis:

- **Experiments with SR-IOV**: Due to the comprehensive analysis of the base setup, we could not collect measurements for a more intricate topology with VFs. Our prototype implements SR-IOV in the same way as HVNet does and is functional according to early tests. However, we did encounter an issue with the VF driver `IAVF`, which causes occasional crashes. In future work, we can solve the issue and compare the impact of using SR-IOV on containers and VMs.
- **Flow based experiments**: The HVNet paper [4] attained the results by running flow-based experiments. Each VM is attached to a VF from the ingress and egress NICs, which allows injection of traffic into any node. Since this setup was previously only tested with the splitter setup, we reserve the exact comparison between our work and the results of HVNet for future work.
- **Comparison with other container solutions**: There is a wide array of container solutions available. Due to the lack of literature, a comparison in terms of tooling and network latencies between the most popular ones such as Docker, OpenVZ, and LXC could be an interesting study.
- **CGroups v1 with Buster**: With CGroups v1 it is possible to disable the load balancing mechanism of the scheduler. This could eliminate the observed latency spikes. Furthermore, when downgrading to CGroups v1, the container implementation could be backported to Debian Buster for the best comparability with other work [4], [17].
- **Impact of NUMA traversal on latencies**: When creating larger topologies, it is not always possible to assign CPU cores while respecting NUMA optimality. Previous work [21] already measured the impact of NUMA traversals on bandwidth and latency. However, they focused on bandwidth analysis and only denote the latency for a NUMA traversal as 1.7 µs without further details.

Chapter A

Appendix

List of acronyms

pos plain orchestration service, host and VM orchestrator for a testbed [16]

OT operational technology, hard- and software specialized on operating manufacturing plants and assembly lines

IT information technology, hard- and software involved with creating, processing, storing, securing and exchanging data; typically in a business context

PLC programmable logic controller, industrial computer specialized for controlling a production or manufacturing line; trimmed for reliability, low maintenance and low latency

VM virtual machine, form of virtualization, often called heavy-weight virtualization due to virtualization an entire OS and hardware

NIC network interface card, facilitate networking tasks

SR-IOV single root I/O virtualization, virtualizes physical hardware into multiple lightweight VFs

PF physical function, a PCIe function supporting the configuration of the SR-IOV specification

VF virtual function, a lightweight PCIe function attached to a physical function with SR-IOV

LXC Linux containers, a toolbox for creating and managing lightweight system containers

DPDK Data Plane Development Kit, a framework for accelerating packet processing with user space tooling

IPMI Intelligent Platform Management Interface, set of specifications for out-of-band control of computers

BMC baseboard management controller, dedicated embedded microcontroller; handles out-of-band communication with IPMI

vBMC virtual baseboard management controller, a BMC implemented in software only

VLAN virtual local area network, layer 2 partitioning of networks

IOMMU input–output memory management unit, maps device addresses to physical memory

PID process identifier, 16-bit ID for uniquely identifying a process

OS operating system, the operating system running on a PC

DuT Device under Test, a device that is investigated during experiments

pcap packet capture, an API for capturing network traffic; includes information of layer 2 - 7

VMM virtual machine monitor, manages access to physical resources for virtual machines

KVM kernel-based virtual machine, open source virtualization technology for Linux

URLLC ultra reliable low-latency communication

RCU read-copy update, synchronization mechanism of the Linux kernel

vEth virtual Ethernet, always created in pairs, connect namespaces or different vEth interfaces with a Linux bridge

DMA direct memory access, bypasses the CPU for memory access

HPC high-performance computing, highly parallelized computing procedures utilizing entire datacenters of computers

TLB translation lookaside buffer, caches translations of mappings from physical to virtual address space

NUMA non-uniform memory access, a set of CPU cores has its own memory controller bypassing the memory bottleneck

SSH secure shell, protocol for encrypting a connection; most frequently used to transmit commands to a remote host

NAT network address translation, maps from one IP address space into another

HDR high dynamic range, shows more details for higher values

Bibliography

[1] S. Gallenmüller, J. Naab, I. Adam und G. Carle, "5G URLLC: A Case Study on Low-Latency Intrusion Prevention," *IEEE Communications Magazine*, Jg. 58, Nr. 10, S. 35–41, Okt. 2020. DOI: 10.1109/MCOM.001.2000467.

[2] M. Sollfrank, F. Loch, S. Denteneer und B. Vogel-Heuser, "Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation," *IEEE Transactions on Industrial Informatics*, Jg. 17, Nr. 5, S. 3566–3576, 2021. DOI: 10.1109/tii.2020.3022843. Adresse: https://doi.org/10.1109/TII.2020.3022843.

[3] A. Tanenbaum und H. Bos, *Modern Operating Systems* (Always learning). Pearson, 2015, ISBN: 9780133591620. Adresse: https://books.google.de/books?id=9gqnngEACAAJ.

[4] F. Wiedner, M. Helm, S. Gallenmüller und 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) (INFOCOM WKSHPS CNERT 2022)*, Virtual Event, Mai 2022.

[5] LXD development community, *Introduction to LXD*, Available online at https://linuxcontainers.org/lxd/introduction/; last accessed on 2022/08/07., 2014.

[6] J. Watada, A. Roy, R. Kadikar, H. Pham und B. Xu, "Emerging Trends, Techniques and Open Issues of Containerization: A Review," *IEEE Access*, Jg. 7, S. 152 443–152 472, 2019. DOI: 10.1109/access.2019.2945930. Adresse: https://doi.org/10.1109/access.2019.2945930.

[7] A. M. Joy, "Performance comparison between Linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, IEEE, März 2015. DOI: 10.1109/icacea.2015.7164727. Adresse: https://doi.org/10.1109/icacea.2015.7164727.

[8] P. Sharma, L. Chaufournier, P. Shenoy und Y. C. Tay, "Containers and Virtual Machines at Scale," in *Proceedings of the 17th International Middleware Con-*

ference, ACM, Nov. 2016. DOI: 10.1145/2988336.2988337. Adresse: https://doi.org/10.1145/2988336.2988337.

[9] *cgroups(7) — Linux manual page*, Aug. 2021.

[10] D. D'Silva und D. D. Ambawade, "Building A Zero Trust Architecture Using Kubernetes," in *2021 6th International Conference for Convergence in Technology (I2CT)*, IEEE, Apr. 2021. DOI: 10.1109/i2ct51068.2021.9418203. Adresse: https://doi.org/10.1109/i2ct51068.2021.9418203.

[11] *systemd.slice(5) — Linux manual page*, Freedesktop, Aug. 2021.

[12] V. Šraier, *Transparent Restarts of Stateless Linux Services*, English, Bachelor's thesis, 2020. Adresse: http://hdl.handle.net/20.500.11956/119403.

[13] *namespaces(7) — Linux manual page*, Aug. 2021.

[14] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian und H. Guan, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, Jg. 72, Nr. 11, S. 1471–1480, Nov. 2012. DOI: 10.1016/j.jpdc.2012.01.020. Adresse: https://doi.org/10.1016/j.jpdc.2012.01.020.

[15] R. Krishnakumar, "Accelerated DPDK in containers for networking nodes," English, Master's thesis, Aalto University. School of Electrical Engineering, 2019, S. 65. Adresse: http://urn.fi/URN:NBN:fi:aalto-201903172255.

[16] S. Gallenmüller, D. Scholz, H. Stubbe, E. Hauser und G. Carle, "Reproducible by Design: Network Experiments with pos," in *KuVS Fachgespräch - Würzburg Workshop on Modeling, Analysis and Simulation of Next-Generation Communication Networks 2022 (WueWoWas'22)*, Würzburg, Germany, Juli 2022.

[17] S. Gallenmüller, F. Wiedner, J. Naab und G. Carle, "Ducked Tails: Trimming the Tail Latency of(f) Packet Processing Systems," in *3rd International Workshop on High-Precision, Predictable, and Low-Latency Networking (HiPNet 2021)*, Izmir, Turkey, Okt. 2021.

[18] H. Bi und Z.-H. Wang, "DPDK-based Improvement of Packet Forwarding," *ITM Web of Conferences*, Jg. 7, T. Gong, T. Yang und J. Xu, Hrsg., S. 01 009, 2016. DOI: 10.1051/itmconf/20160701009. Adresse: https://doi.org/10.1051/itmconf/20160701009.

[19] H. J. Koch und H. L. Gmb, "Userspace I/O drivers in a realtime context," in *The 13th Realtime Linux Workshop*, 2011.

[20] S. Gallenmüller, "Data-Driven Analysis and Modeling of Packet Processing Systems," Diss., Technical University of Munich, Feb. 2021. DOI: 10.2313/NET-2021-02-1.

[21] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl und G. Carle, "User Space Network Drivers," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*, Sep. 2019.

[22] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart und G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference (IMC) 2015, IRTF Applied Networking Research Prize 2017*, Tokyo, Japan, Okt. 2015.

[23] W.-E. Chen und C. H. Liu, "Performance Enhancement of Virtualized Media Gateway with DPDK for 5G Multimedia Communications," in *2019 International Conference on Intelligent Computing and its Emerging Applications (ICEA)*, IEEE, Aug. 2019. DOI: 10.1109/icea.2019.8858303. Adresse: https://doi.org/10.1109/icea.2019.8858303.

[24] C.-N. Mao, M.-H. Huang, S. Padhy u. a., "Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, Nov. 2015. DOI: 10.1109/cloudcom.2015.67. Adresse: https://doi.org/10.1109/cloudcom.2015.67.

[25] J. Zhang, X. Lu und D. K. Panda, "Performance Characterization of Hypervisor- and Container-Based Virtualization for HPC on SR-IOV Enabled InfiniBand Clusters," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, Mai 2016. DOI: 10.1109/ipdpsw.2016.178. Adresse: https://doi.org/10.1109/ipdpsw.2016.178.

[26] B. Lantz und B. O'Connor, "A Mininet-based Virtual Testbed for Distributed SDN Development," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ACM, Aug. 2015. DOI: 10.1145/2785956.2790030. Adresse: https://doi.org/10.1145/2785956.2790030.

[27] M. Peuster, H. Karl und S. van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, IEEE, Nov. 2016. DOI: 10.1109/nfv-sdn.2016.7919490. Adresse: https://doi.org/10.1109/nfv-sdn.2016.7919490.

[28] A. Beifuß, D. Raumer, P. Emmerich u. a., "A study of networking software induced latency," in *2015 International Conference and Workshops on Networked Systems (NetSys)*, 2015, S. 1–8. DOI: 10.1109/NetSys.2015.7089065.

[29] A. R. Portabales und M. L. Nores, "Dockemu: Extension of a Scalable Network Simulation Framework based on Docker and NS3 to Cover IoT Scenarios.," in *SIMULTECH*, 2018, S. 175–182.

[30] B. Ruan, H. Huang, S. Wu und H. Jin, "A Performance Study of Containers in Cloud Environment," in *Lecture Notes in Computer Science*, Springer International Publishing, 2016, S. 343–356. DOI: 10.1007/978-3-319-49178-3_27. Adresse: https://doi.org/10.1007%2F978-3-319-49178-3_27.

[31] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li und A. Apon, "Performance considerations of network functions virtualization using containers," in *2016 International Conference on Computing, Networking and Communications (ICNC)*, 2016, S. 1–7. DOI: 10.1109/ICCNC.2016.7440668.

[32] Z. Li, M. Kihl, Q. Lu und J. A. Andersson, "Performance Overhead Comparison between Hypervisor and Container Based Virtualization," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, IEEE, März 2017. DOI: 10.1109/aina.2017.79. Adresse: https://doi.org/10.1109/aina.2017.79.

[33] H. Guan, Y. Dong, K. Tian und J. Li, "SR-IOV Based Network Interrupt-Free Virtualization with Event Based Polling," *IEEE Journal on Selected Areas in Communications*, Jg. 31, Nr. 12, S. 2596–2609, 2013. DOI: 10.1109/JSAC.2013.131202.

[34] F. Ramalho und A. Neto, "Virtualization at the network edge: A performance comparison," in *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2016, S. 1–6. DOI: 10.1109/WoWMoM.2016.7523584.

[35] H. Akkan, M. Lang und L. M. Liebrock, "Stepping towards noiseless Linux environment," in *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '12*, ACM Press, 2012. DOI: 10.1145/2318916.2318925. Adresse: https://doi.org/10.1145/2318916.2318925.

[36] A. Daichendt, *HVNet Repository*, Available online at https://gitlab.lrz.de/tumi8-theses/ba-daichendt/hv-net; last accessed on 2022/08/07.

[37] Red Hat, *Linux Containers with libvirt-lxc (deprecated)*, Available online at https://access.redhat.com/articles/1365153; last accessed on 2022/08/07., 2015.

[38] Canonical, *LXC*, Available online at https://ubuntu.com/server/docs/containers-lxc; last accessed on 2022/08/07., 2021.

[39] M. Kraai, *debootstrap - Bootstrap a basic Debian system*, Debian Project, Apr. 2001.

[40] Debian wiki community, *Bridged setup for your container(s) network*, Available online at https://wiki.debian.org/LXC/SimpleBridge; last accessed on 2022/08/07., 2011.

[41] J. Frazelle, "Opening up the Baseboard Management Controller," *Queue*, Jg. 17, Nr. 5, S. 5–12, Okt. 2019. DOI: 10.1145/3371595.3378404. Adresse: https://doi.org/10.1145/3371595.3378404.

[42] Kernel development community, *Linux Base Driver for the Intel(R) Ethernet Controller 700 Series*, Available online at https://www.kernel.org/doc/html/

latest/networking/device_drivers/ethernet/intel/i40e.html; last accessed on 2022/08/07.

[43] K. Jambur, *[dpdk-dev] [README]:Running DPDK in a LXC-based Container*, Available online at http://mails.dpdk.org/archives/dev/2014-October/006373.html; last accessed on 2022/08/07., 2014.

[44] Intel Network Division, *Intel Ethernet Adaptive Virtual Function (AVF) Hardware Architecture Specification (HAS)*, Available online at https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ethernet-adaptive-virtual-function-hardware-spec.pdf; last accessed on 2022/08/07., 2018.

[45] S. Graber, *LXD Github Issue 3619: DPDK in LXD Container*, Available online at https://github.com/lxc/lxd/issues/3619#issuecomment-319430483; last accessed on 2022/08/07., 2017.

[46] DPDK documentation, *43.2.3. Running Multiple Independent DPDK Applications*, Available online at http://doc.dpdk.org/guides/prog_guide/multi_proc_support.html#running-multiple-independent-dpdk-applications; last accessed on 2022/08/07., 2022.

[47] T. Heo, *Control Group v2*, Available online at https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html; last accessed on 2022/08/07., 2015.

[48] DPDK, *Thread Safety of DPDK Functions*, Available online at https://doc.dpdk.org/guides-18.05/prog_guide/thread_safety_dpdk_functions.html#interrupt-thread; last accessed on 2022/08/07., 2018.

[49] *The kernel's command-line parameters*, Available online at https://docs.kernel.org/admin-guide/kernel-parameters.html; last accessed on 2022/08/07.

[50] P. Emmerich, *libmoon*, Available online at https://github.com/libmoon/libmoon; last accessed on 2022/08/07.

[51] F. Wiedner, *libmoon*, Available online at https://github.com/WiednerF/libmoon.git; last accessed on 2022/08/07.

[52] A. Daichendt, *LXC result data*, Available online at https://gitlab.lrz.de/tumi8-theses/ba-daichendt/lxc-results; last accessed on 2022/08/07.

[53] S. Derr, *Control Group v1 CPUSET*, Available online at https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cpusets.html#what-is-sched-load-balance; last accessed on 2022/08/07., 2004.

[54] W. Long, *[PATCH v11 0/8] cgroup/cpuset: cpu partition code enhancements*, Available online at https://lore.kernel.org/lkml/20220510153413.400020-1-longman@redhat.com/T/#m0054f46ac9bc975579e23306e785d1b88ec602c2; last accessed on 2022/08/07., 2022.