

# Graphical Abstract

## Performance Evaluation of Containers for Low-Latency Packet Processing in Virtualized Network Environments

Florian Wiedner, Max Helm, Alexander Daichendt, Jonas Andre, Georg Carle


### Motivation

**Low-Latency Network Applications on Hardware**

- ▶ Expensive
- ▶ Limited availability
- ▶ Long delivery times

**Low-Latency Network Applications on Virtual Machines**

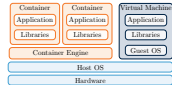
- ▶ Substantial overhead
- ▶ Complete Operating System for every machine
- ▶ Long boot times



### Background

Virtualization of systems is possible using:

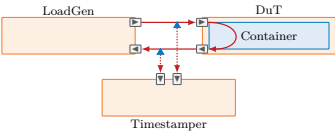
- ▶ Full virtualization, e.g., virtual machines (VMs)
- ▶ OS-level virtualization, e.g., container



Raise of latency is according to [2] mostly caused by:

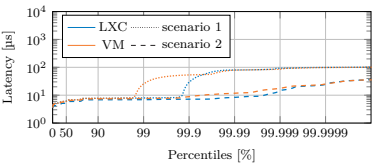
- ▶ Interrupts raised on the same core as the virtualization
- ▶ Energy-saving mechanism during idle times
- ▶ Other applications running on the same cores as the virtualization

### Measurement Setup



- ▶ Loadgen runs a packet generator (MoonGen [1]) creating UDP packets
- ▶ Device under Test (DuT) contains to be analyzed system
- ▶ Timestamper records ingress/egress traffic using passive optical traffic access points (TAPs)
  - Hardware-timestamping of entire network traffic (resolution 1.25 ns)
  - Determine worst-case latencies on a per-flow basis

### Evaluation: Optimized Container in two scenarios



*Virtual machines in comparison to LXC container on AMD and Intel Mainboards*

- ▶ Using optimizations such as to:
  - reduce interrupts
  - disable energy-saving-mechanism
- ▶ Latency until 110  $\mu$ s.
- ▶ Difference on AMD- and Intel-based system negligible

### Modeling: Extreme Value Theory

Platform	Opt.	RT	NoHz	Vanilla	Exceedances
VM	✓	x	x	✓	1.25
VM	x	x	x	✓	2.58
Container	✓	✓	x	x	1.42
Container	x	✓	x	x	7.67
Container	✓	x	✓	x	1.25
Container	x	x	✓	x	1.67
Container	✓	x	x	✓	2.92
Container	x	x	x	✓	2.29
Kernel Netw.	✓	✓	x	x	2.50
Kernel Netw.	x	x	x	✓	22.73

- ▶ Model tail-latencies using Extreme Value Theory
- ▶ Predict behavior four-fold into the future
- ▶ Optimized containers are the most predictable virtualization technique

### Conclusion

Container are possible for low-latency applications when:

1. the cache system is carefully selected
2. optimizations such as disabling energy-saving are utilized

*Carefully selection of Hardware architecture and optimizations required*

- ▶ Container are more relying on the underlying hardware system
- ▶ In general, more fine-grained optimizations on container needed

*More analysis especially towards concurrent container and optimizations required.*

[1] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[2] S. Gallenmüller, F. Wiedner, J. Naab, and G. Carle. How low can you go? A limbo dance for low-latency network functions. *J. Netw. Syst. Manag.*, 31(1):20, 2023.

## Highlights

### **Performance Evaluation of Containers for Low-Latency Packet Processing in Virtualized Network Environments**

Florian Wiedner, Max Helm, Alexander Daichendt, Jonas Andre, Georg Carle

- Overhead caused by virtualization, including light and full virtualization, is negligible in tail latencies.
- Containerized packet processing systems can traffic with low-latency requirements depending on the underlying hardware cache design and utilized optimizations.
- Latencies over time processed through container can be modeled more precisely than other virtualization techniques.
- Latencies on containers are more prone to influences from the underlying hardware than virtual machines.

# Performance Evaluation of Containers for Low-Latency Packet Processing in Virtualized Network Environments

Florian Wiedner\*, Max Helm, Alexander Daichendt, Jonas Andre, Georg Carle

*<sup>a</sup>Technical University of Munich  
School of Computation, Information, and Technology  
Department of Computer Engineering, Boltzmannstr. 3, Garching by  
Munich, 85748, Bavaria, Germany*

---

## Abstract

Packet processing in current network scenarios faces complex challenges due to the increasing prevalence of requirements such as low latency, high reliability, and resource sharing. Virtualization is a potential solution to mitigate these challenges by enabling resource sharing and on-demand provisioning; however, ensuring high reliability and ultra-low latency remains a key challenge. Since bare-metal systems are often impractical because of high cost and space usage, and the overhead of virtual machines (VMs) is substantial, we evaluate the utilization of containers as a potential lightweight solution for low-latency packet processing. Herein, we discuss the benefits and drawbacks and encourage container environments in low-latency packet processing when the degree of isolation of customer data is adequate and bare metal systems are unaffordable. Our results demonstrate that containers exhibit similar latency performance with more predictable tail-latency behavior than bare metal packet processing. Moreover, deciding which mainboard architecture to use, especially the cache division, is equally vital as containers are prone to higher latencies on more shared caches between cores especially when other optimizations cannot be used. We show that this has a higher impact

---

\*Corresponding author,

©2024, This manuscript version is made available under the CC-BY-NC-ND 4.0 license <https://creativecommons.org/licenses/by-nc-nd/4.0/>

*Email addresses:* [wiedner@net.in.tum.de](mailto:wiedner@net.in.tum.de) (Florian Wiedner), [helm@net.in.tum.de](mailto:helm@net.in.tum.de) (Max Helm), [daichend@net.in.tum.de](mailto:daichend@net.in.tum.de) (Alexander Daichendt), [andre@net.in.tum.de](mailto:andre@net.in.tum.de) (Jonas Andre), [carle@net.in.tum.de](mailto:carle@net.in.tum.de) (Georg Carle)

on latencies within containers than on bare metal or VMs, resulting in the selection of hardware architectures following optimizations as a critical challenge. Furthermore, the results reveal that the virtualization overhead does not impact tail latencies.

*Keywords:* low-latency, container, virtualization, packet processing

---

## 1. Introduction

Low-latency packet processing applications are driving improvements in areas such as autonomous driving or real-time industrial automation. These applications frequently utilize specialized hardware to fulfill the demands of these applications. However, specialized machines create scalability challenges, affecting the cost per service rate when real-time requirements are in play. The 5G ultra-reliable low-latency communications (URLLC) profile provides a framework for systems that require ultra-low latency, defined as <1 ms end-to-end latency and 99.999<sup>th</sup> percentile of traffic must be within this limit [1]. Using dedicated hardware to run applications for such purposes is not an economically efficient solution for the customer or provider.

Therefore, a solution that offers on-demand provisioning and resource sharing for low-latency network services is required. Virtualization of computer systems is one such solution requiring only general-purpose hardware. However, using virtual machines (VMs) with a complete operating system (OS) results in significant performance, memory, and disk space usage overhead. Gallenmüller et al. [2] compared packet processing between bare metal and VMs on commodity hardware and reported that tuning Linux to reduce interrupts and other influences significantly reduces tail latency in packet processing. VMs offer a high level of isolation that is unnecessary in many cases. Hence, a lighter version is preferred to improve resource usage.

Containers offer a lightweight virtualization alternative for resource sharing with other containers and host OS on the same system. While containers do not virtualize the complete OS, several lightweight software isolation mechanisms are available [3]. Both solutions, VMs and containers, and their induced latency may vary between base systems and vendors.

Given the significance of low latency and high reliability in critical systems, evaluating the tail-latency behavior of packet processing during the long-term execution of applications in containers is essential. Moreover, analyzing the influence of the general hardware mainboard architectural system

is needed for an in-depth understanding of influences outside of the used software systems towards bare-metal, container, or VMs as solutions. Herein, we provide

1. an investigation of the influence of optimization techniques on tail-latency using full- and light-virtualization utilizing different hardware system architectures,
2. a model for tail-latency behavior in packet processing within containers and VMs,
3. a comparison of using network packet processing applications on bare metal, containers, and VMs for low-latency optimized, commercial off-the-shelf systems for the use with URLLC, and
4. an analysis of virtualization techniques on selected hardware architectures and their influence on latency.

This work is based on our work presented at International Teletraffic Congress 35 (ITC-35) 2023 in Torino, Italy [4]. We extended our work with additional recommendations based on the cache model of the system’s architecture and an analysis of different vendor’s additional mainboard architectural system for a complete comparison and analysis. Moreover, we additionally analyzed a layer-3 forwarding application, extending our analysis of layer-2 forwarding applications towards the additional overhead due to more heavy packet processing on the upper layer.

The article is structured as follows: Section 2 offers background information and presents the current development and research progress in virtualization and latency optimizations, including software- and hardware-based ones. Section 3 outlines optimization techniques for containers. Section 4 describes the measurement setup and Section 5 evaluates the proposed approach’s results. We provide models of the tail latencies in Section 6. Section 7 recommends using specific virtualization techniques in low-latency packet processing. Sections 8 to 10 conclude the paper by presenting limitations, reproducibility information, a conclusion, and future scope for research and development.

## 2. Background and Related Work

This section analyzes relevant literature in containers, VMs, low-latency applications and optimization, and tail-latency models. For simpler readability are all acronyms used throughout the article summarized in Table 1.

Table 1: List of acronyms.

---

ADF	Augmented Dickey-Fuller
BM	Block Maxima
DPDK	Data Plane Development Kit
DuT	Device under Test
EVT	Extreme Value Theory
GEV	Generalized Extreme Value
GPD	Generalized Pareto Distribution
HDR	High-Dynamic-Range diagram
JS	Jensen-Shannon divergence
KPSS	Kwiatkowski-Phillips-Schmidt-Shin test
KVM	Kernel Virtual Machine
LXC	Linux Container
LoadGen	Load-Generator
NIC	Network Interface Card
NUMA	Non-Uniform-Access
OS	Operating System
POS	Plain Orchestration Service
PoT	Points over Threshold
RT	Real-Time
RX	Receiver
TAP	Terminal Access Point
TLB	Translation Lookaside Buffer
TX	Transceiver
URLLC	Ultra-Reliable Low-Latency Communications
VM	Virtual Machine
cgroup	Control Group
timestamp	Timestamping Machine

---

### 2.1. Containers and VMs

Using a single hardware machine for each customer or application is neither cost-effective nor flexible. Therefore, virtualization is a crucial technology that enables resource sharing and flexible, on-demand provisioning of resources. However, when executed in a virtual environment, applications with strict low-latency and reliability requirements should perform similarly to those implemented on bare metal.

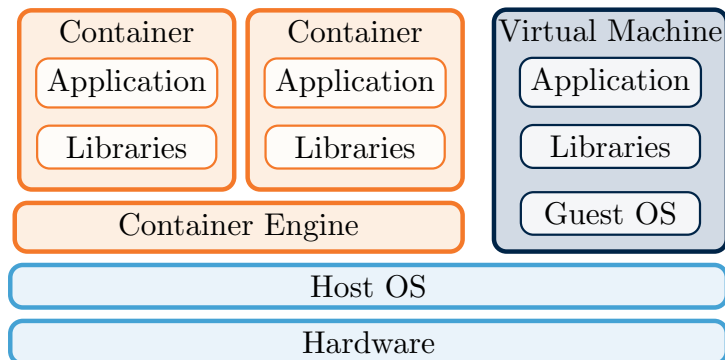


Figure 1: Comparison between container (left) and VMs (right) [7].

Two commonly used architectures for virtualization are hypervisor- and container-based. Hypervisor-based virtualizations (VMs) isolate the complete OS, including the kernel. We call containers lightweight or OS-level virtualizations as they share the kernel between host OS and containers [5]. Containers isolate mainly processes, files, and resource access [5]. As illustrated in Figure 1, the OS kernel and hardware features are commonly not emulated or paravirtualized [6]. In Figure 1, the base shows the hardware running the host OS, and the top shows the types of virtualization that are available; the left side depicts containerization, which includes the container engine used to manage the containers; Moreover, the right side depicts a VM, which shows the additional overhead of the guest OS residing within each system.

Yadav et al. [7] describe that VMs offer a strict separation using virtualized hardware, and a completely separate OS providing a high level of isolation and reducing the influence of customers on each other on the same physical machine. However, this isolation level results in a significant overhead in resource usage, making VMs ideal for experimental and high-security applications, with a trade-off between security and resources necessary for URLLC applications [8].

Moreover, Yadav et al. [7] specify containerization as flexible and less resource-intensive than VMs. With a shared kernel, containers offer quick startup and direct device access, as Gedia and Perigo [9] have demonstrated. This minimized overhead makes containers ideal for performance-critical scenarios where multiple applications must interact with each other [6]. Linux offers several container frameworks, such as Kubernetes, a cluster manager

that automates deployment and enhances application portability, or Linux containers (LXC). LXC integrates all libraries of a complete OS but uses a more complex setup than solutions like Docker and has less overhead [6]. Gedia and Perigo [9] have demonstrated that containers outperform VMs in provisioning time and memory utilization. Wiedner et al [10] detailed that the version of control groups (cgroup) matters for latency-sensitive systems as version 2 outperforms version 1. Control groups are an integral part of process isolation in LXC containers. Therefore, in our further analysis, we utilize cgroups version 2.

The throughput analysis of containers and VMs is a common area of research, which is evident from the considerable attention paid to it.

Barham et al. [11] studied the impact of CPU resources on XEN-based VMs, focusing on variations induced by time slices on the CPU. Furthermore, Abeni et al. [12] analyzed the effect of tuning Linux on the maximum packet rate of kernel virtual machine (KVMs) and achieved promising results by binding CPU affinity of interrupts to selected cores and the VMs to remaining cores. Similarly, Tran and Kim [13] found that CPU core assignment for containers is crucial for improving throughput. Morabito et al. [14] conclude that containers challenge traditional systems regarding resource usage and performance. Furthermore, Cha and Kim [15] employed containers to offer low-latency edge services and demonstrated that container setups achieve near-optimal throughput by utilizing hardware support. To conclude, research on packet processing in virtualized systems primarily focuses on throughput analysis or latency using overlay networks such as [16, 17].

Several studies have analyzed latency on VMs as demonstrated in [2, 18, 19, 20]. However, latency analysis for packet processing applications based on containers is typically not addressed in present studies despite the necessity to examine the influences of packet processing through containers. New research focuses on latency and real-time applications within containers. For example, Liu et al. [21] recently analyzed the usability of Docker’s overlay network compared to the host network mode for real-time applications. They conclude that the overall performance of the host-network mode is better on average, but tail latency was not further analyzed. Furthermore, Wiedner et al. [4, 10] analyzed the latency performance of LXC containers in comparison to VMs and hardware, concluding that optimized LXC can achieve similar tail latencies as the other two variants. This work extends the paper from Wiedner et al. [4], demonstrating that low-latency applications on containerized systems are possible. However, they used only one



potential system architecture of mainboards, leaving the comparison to other architectures for future work.

## *2.2. Low-latency Applications*

Packet-processing applications with end-to-end latency requirements of  $<1$  ms in general-purpose traffic networks are becoming increasingly important with new technologies such as the 5G URLLC profile [22]. Therefore, packet processing applications must improve latency and reliability.

Gallenmüller et al. [23] analyzed latency implications on intrusion detection systems and found that using a specific OS, reducing interrupts, and using a specific network interface card (NIC) help to reduce latency spikes. As Bozilov et al. [24] reported, adding security features into the network introduces additional latency, but they are increasingly important. Therefore, it is crucial to reduce network-induced latency to allow security mechanisms within networks. Jain et al. [25] have shown that improving the data plane on specialized network function virtualization systems utilizing the data plane development kit (DPDK) can significantly reduce latency.

Other examples of low-latency applications include data center internal communication [26], communication phases in distributed machine learning applications [27], and cloud systems providing centralized services to multiple users as outlined by Gandhi et al. [28].

## *2.3. Low-latency OS Optimizations*

Tuning and optimizing OS is essential to enable predictable, reliable, and low-latency applications on any system type, whether container, VM, or bare metal. Previous studies report that tunings in specific areas are possible, such as reducing the impact of processing IO on a container and reducing the influences of the system itself. For instance, Gallenmüller et al. [23] have demonstrated that interrupts on the packet-processing core have a significant impact on latency. Turning off timer ticks, isolating cores, and reducing energy-saving mechanisms on VMs can minimize the impact on packet processing through virtualization [8]. Which optimizations are available and needed depends on the used hardware and its architecture, including memory locality, cache design, or hardware-supported optimizations as depicted by [29]. Using poll mode instead of interrupt-based drivers improves latency and reduces the number of context switches. Handley et al. [30] found that using DPDK [31], a framework for poll-mode networking significantly reduces processing latency as the application can be isolated from the OS kernel.

Like VMs, container performance can be improved by reducing interrupts and adding predictability [32]. Herein, we evaluate and demonstrate the optimization potential for containers compared to packet processing on VMs or bare metal, including exploring further optimizations for low-latency-networking.

#### 2.4. Hardware-dependent optimizations

Software optimizations are needed for enabling low latency on generic-purpose hardware, but their impact depends heavily on the underlying hardware and its design [29]. Moreover, most detailed implementation information is not available to the public for hardware systems as vendors protect themselves for economic reasons such that most impacts leading to hardware details remain a black box for the researcher. Based on this, finding the reason behind the measurement results of hardware differences takes much work. Therefore, analyzing results on different systems and system architectures is required to draw clear conclusions on their influence.

Architectures such as AMD’s chiplet-based design, initially introduced with the first generation AMD EPYC processors, pioneered a new area of distributed processing and memory access [33]. In this case, each chiplet defines one non-uniform-access (NUMA) area with attached IO devices. Based on related work, this design provides improved performance but unlocked new challenges towards inter-chip communication, memory, cache access, and synchronization of hardware clock timings [33]. The interconnection mainly provides a higher cost regarding memory access latency, which also impacts network performance when not pinned to a core directly connected to the specific NUMA node. The cache is divided into the different NUMA nodes, which can be divided even within the NUMA nodes.

Another substantial vendor in computer mainboards is Intel, which provides the Intel Xeon server architectures. Intel adhered to a monolithic computer architecture, such as Skylake, without chiplets and the challenges of inter-chiplet communication, providing enhanced possibilities for optimization of cache access [34]. Those different systems provide advantages and disadvantages based on the specific use case. We conclude that analyzing differences based on hardware machines is done on a use-by-use basis, as the advantages and disadvantages significantly depend on individual needs.

Until now, only two selected systems have been presented, and the challenge is: On the one hand, vendors improve architectures incrementally, leading to new generations of system architectures. On the other hand,

companies such as ARM offer an entirely different architecture that is not binary-compatible with x86. Different vendors such as Intel, ARM, or AMD adapt their architectures over time to overcome their challenges and provide enhancements for their user [35]. Depending on the hardware machines, the conclusion drawn from measurements can be quite different, which makes the selection and evaluation process of hardware machines significantly harder.

Optimizing the systems differs due to hardware-specific optimizations and constraints. For example, Intel provides the Intel Cache Allocation Technology [36], a way to highly customize the cache’s access and usage. In contrast, AMD does not provide a similar scheme resulting from the physically more divided cache. This technology can be further used for improving the application’s performance, such as in [37]. Optimizing the assignment of cores to VMs or containers is more important for AMD due to its chiplet design based on NUMA-nodes [33]. Using these approaches, we will further analyze the systems using, in general, the same optimizations as applicable, but also leverage hardware-specific optimizations based on results from previous works.

### *2.5. Low-latency on container*

Using containers for low latency and highly reliable systems can be challenging but feasible. The processing time on the container node is prolonged due to interrupts needed for the container engine. To mitigate this issue, one method is to use poll-mode drivers that minimize context switches and interrupts. DPDK [31] provides user-space, poll-mode networking to accelerate packet processing, with a broad range of available applications such as MoonGen [38], a high-speed packet generator, and Snort [39], an intrusion detection system. Using user-space networking within containers requires additional tasks outside the container as selected operations require privileged access, such as binding interfaces to the user-space drivers [13]. They must be performed initially on the host OS, after which the interface can be moved into the container’s namespace. Tran and Kim [13] have demonstrated that DPDK applications can be used with containers after performing these additional steps.

Moreover, containers cannot be entirely isolated from the host OS as a shared kernel is used, and interrupts are needed on the specific cores, resulting in the challenge of optimizing them for low-latency operations while reducing the influence of operations performed outside of the containers. The

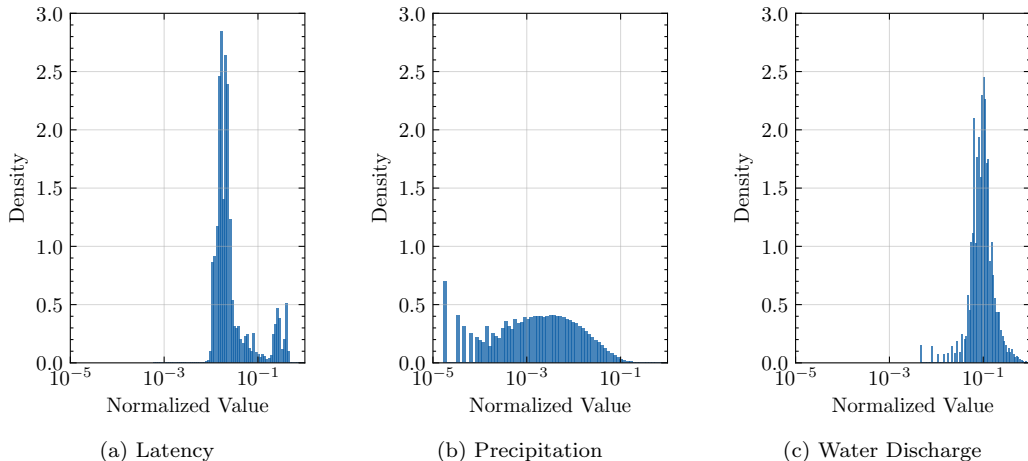


Figure 2: Histograms of normalized datasets for latency-, precipitation-, and water discharge measurements [40, 41]. The measurements were selected for their exemplary nature.

challenges presented herein illustrate the trade-off between resource sharing and URLLC.

### 2.6. Tail-latency Behavior and Models

Distributions of measured latencies in communication networks typically exhibit a long tail [8]. This long tail is not unique to latencies; for example, it exists in measured precipitation [40] and water discharge in river networks [41] as shown in Figure 2. We can observe three long-tailed distributions with varying scale and location factors; note the log scale. Both precipitation and water discharge are commonly modeled using Extreme Value Theory (EVT).

EVT is a statistical technique used to model the behavior of distribution tails [42]. It relies on historical data to predict the probability and magnitude of rare events like natural disasters, e.g., floods as indicated by precipitation or water discharge. In the networking domain, EVT can be utilized to model tail latencies since they constitute similarly rare events. Previous studies have successfully applied EVT to analyze time sequences in both wired and wireless networks [43, 44, 45, 46].

Within EVT, a distribution’s tail is determined using either the Block Maxima (BM) approach or the Points over Threshold (PoT) approach. The BM method uses a Generalized Extreme Value (GEV) distribution as a model, while the PoT method uses a Generalized Pareto Distribution (GPD)

model. The connection between the selection method and type of model is described by the Fisher–Tippett–Gnedenko- [47] and Pickands–Balkema–De Haan [48, 49] theorems respectively. The PoT method is less wasteful with data; therefore, it is typically preferred over the BM method [42]. As a result, we only consider the PoT method and GPD model.

Our study adopts a similar approach to previous studies to predict and validate the likelihood of latency spikes. The validation shows an accurate predictive power of such models when extrapolating tail-latency magnitudes and frequencies to extended measurement periods. Additionally, we evaluate the convergence of these predictions.

### 3. Optimization Analysis

In computer systems, processes can be affected, among others, by interrupts, the sleep state of CPUs, or concurrent processes. While throughput in packet processing is unaffected by these influences, latency, specifically tail latency, is significantly impacted. Having examined optimizations in both bare-metal and VM environments, we evaluate the suitability of these optimizations for container environments.

Several studies have examined network latency optimization techniques in VMs and bare-metal systems [8, 50, 29]. The host OS manages the scheduling of applications in containers, which means achieving complete isolation from interrupts is impossible. Consequently, optimizations such as a tick-less-kernel and isolation of selected CPUs are not feasible as a shared kernel is used, requiring access to the specific cores. Due to these difficulties in isolating containers, it is essential to explore new approaches and conduct assessments of the suitability of these optimizations in containers.

To minimize the impact of the host OS on the container and between containers, LXC provides a method for reserving cores and memory exclusively. Additionally, automatic load balancing in LXC can be turned off to reduce overhead and ensure no additional scheduling is needed [3]. To summarize, these specialized container isolation techniques help to minimize the external influence on a container.

Fine-tuning the poll mode for idle CPUs, disabling energy-saving mechanisms, and turning off audit messages can improve container performance like VMs. Interrupts affinity can be set to a specific core, and logging of backtraces can be reduced to improve latency. Turning off simultaneous multithreading improves latency for all systems. Table 2 presents the suggested

Table 2: Latency optimized boot parameters for Host OS running containerized systems.

Parameter	Value	Description
rcu_nocbs	[ <i>cores</i> ]	No RCU callbacks
rcu_nocbs_poll		No RCU callback threads wakeup
irqaffinity	0	Interrupts on specific core
idle	poll	Poll mode when core idle
tsc	reliable	Rely on TSC without check
mce	ignore_ce	Ignore corrected errors
audit	0	Disable audit messages
nmi_watchdog	0	Disable NMI watchdog
skew_tick	1	No simultaneous ticks for locks
nosoftlookup		Disables logging of backtraces
nosmt		Disables hyperthreading

list of boot parameters. The list is based on the presented container adoptions of optimizations for VMs in the study of Gallenmüller et al. [23]. Using the program *taskset*, it is possible to pin the affinity of all ready-copy-update processes to a core to reduce their scheduling on container cores. However, when resources are limited, the CPUs should be shared between containers, increasing tail latencies.

Furthermore, Intel provides additional optimization possibilities for its systems that are not available for AMD-based systems to our current knowledge. Gallenmüller et al. [23] performed their measurements on an Intel-based system, and their additional optimizations will be acquired in this paper for Intel-based machines. We additionally turn off the dynamic voltage and frequency scaling with *pstates*, which can introduce additional delay when a core is idling or not fully utilized on Intel-based machines [29]. Moreover, we will use all optimization boot parameters outlined in Table 2 on all analyzed systems. This setup enables us to compare results obtained from systems of different vendors towards latency performance with packet processing systems inside containers.

Moreover, different papers and articles describe additional performance optimizations for containers to improve the latency, such as [16, 17]. Several cache optimizations for overlay and bridge networks are available, such as [16]. We cannot use them directly in our case, as we plan to overcome this issue by using direct-attached NICs inside the containers providing direct-

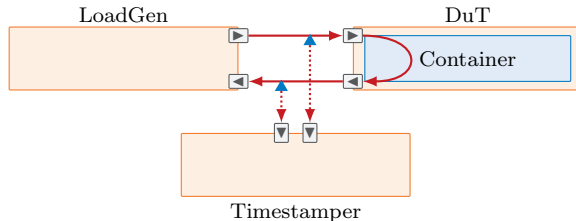


Figure 3: Measurement-setup structure.

memory access from the container and hardware device. Using this, we aim to directly attach the NIC to the container’s namespace. Therefore, isolate it from access to other containers, which is different than the host network mode Docker provides. Moreover, analyzing this paper’s performance improvements, we conclude that cache design and usage can be an issue based on different cache and system designs. However, our utilized methodology does not allow us to utilize the optimization techniques from [16, 17]. Direct access to the NIC reduced scalability by providing higher performance and reduced overhead.

#### 4. Measurement Setup

For precise measurements, load generation and timestamping are performed on separate machines, as depicted in Figure 3. We adopt two scenarios with distinct hardware to extract differences based on hardware architecture. The setup shown in Figure 3 is the same for both scenarios, only differing in utilizing other hardware machines. We utilize our results from [4] as **scenario 1**, and **scenario 2** provides insights into an additional hardware architecture when utilizing a container for low-latency networking.

In **scenario 1**, the load-generator (LoadGen) features an Intel Xeon Silver 4116 CPU, 192 GB RAM, and a dual-port Intel 82599ES 10-Gigabit SFP+ NIC connected to the Device-under-Test (DuT) using optical fibers. In **scenario 2**, the LoadGen runs on an Intel Xeon Gold 6130 CPU, 384 GB RAM, and the same NIC and connections as in **scenario 1**. The differences do not influence our results as we utilize only a single core for packet generation in both scenarios and use hardware rate-limiting based on the NICs capabilities. Moreover, using an external timestamping machine (timestamper), latencies are measured after the packets leave the LoadGen.

We use a timestamper linked to the fibers between DuT and LoadGen

with passive optical terminal access points (TAPs) to ensure high precision measurements per packet at line rate. These TAPs introduce a constant delay to the timestamps on both ends and can be neglected. The timestamper in **scenario 1** is equipped with an AMD EPYC 7542 32-Core Processor, 500 GB of memory, and an Intel E810-XXVDA4 25 Gbit/s NIC flashed to 10 Gbit/s offering a precision of 1.25 ns [51]. This ability takes timestamps in the hardware using the precision of the Intel E810 NIC. Timestamps taken in software are prone to the same interrupts and kernel operations as our DuT; therefore, using hardware timestamping significantly reduces the influence of the timestamping method on the results. In **scenario 2** a machine with the same CPU is used, 128 Gbit of RAM and instead of a 4-port Intel E810-XXVDA4 25 Gbit/s NIC we have a dual-port Intel E810-XXV 25 Gbit/s NIC as well flashed to 10 Gbit/s. Both cards provide the same capabilities but differ in the number of available ports. As we utilize hardware timestamping, the available memory does not influence the results as it is not a bottleneck. The original hardware from **scenario 1** was not longer available under our measurements for **scenario 2**.

The DuT in **scenario 1** is equipped with an AMD EPYC 7551P 32-Core Processor, 128 GB RAM, and  $2 \times$  Intel X710 10GbE SFP+ NICs, where one port each is linked to the LoadGen. In **scenario 2**, the DuT is equipped with two Intel Xeon Silver 4116 CPUs, 192 GB RAM, and  $2 \times$  Intel X710 10GbE SFP+ NICs, cabled in the same way as **scenario 1**. To utilize differences based on the hardware design of the DuT, we utilized different scenarios based on different DUTs with different mainboards.

With this setup, any measured differences can be traced back to the mainboard and CPU, as the same NICs are used in both scenarios. On the DuT, we execute our experiments using bare-metal, VM, or container. The used solutions access the interfaces directly. Finally, we summarize the hardware configurations in Table 3.

To retrieve more information about the architectural differences of both DuT scenarios, we utilized the tool `lstopo` from the `hwloc` package [52]. Figures 4 and 5 are showing information about architecture, cache design, and bus configurations. Figure 4 shows that **scenario 1** machine consists of one mainboard entity divided into four NUMA chiplets with four cores sharing a 8 MB layer three cache. The used NIC ports are distributed on node 1 (TX) and node 3 (RX) with `enp33s0f0` and `enp100s0f0`. This setup requires copying data to send and receive over the Infinity Fabric interconnects between NUMA nodes.



Table 3: Hardware configuration of `scenario 1` and `scenario 2`.

		scenario 1	scenario 2
DuT	CPU	AMD EPYC 7551P	Intel Xeon Silver 4116
	RAM	128 GB	192 GB
	NIC	2 × Intel X710 10GbE SFP+ NICs	
LoadGen	CPU	Intel Xeon Silver 4116	Intel Xeon Gold 6130
	RAM	192 GB	384 GB
	NIC	Intel 82599ES 10-Gigabit SFP+ NIC	
timestamper	CPU	AMD EPYC 7551P 32-Core	
	RAM	500 GB	128 GB
	NIC	Intel E810-XXVDA4 NIC	Intel E810-XXV NIC

In `scenario 2` (cfg. Figure 5), the NICs are both connected to node 0 (`enp24s0f0`, `enp25s0f0`) over the same PCIe bus. A second CPU is installed in another socket, declared as NUMA node 2. However, as no PCIe device is connected to it, it has no relevance to our measurements. Each CPU has only one shared layer three cache of 17MB. In conclusion, the significant differences between the two sampled systems lie in different amounts and kinds of NUMA nodes and cache architectures.

The configuration in Figure 3 facilitates precise analysis of packet processing tail-latency and lets us compare different hardware and software configurations accurately. We use PostgreSQL for evaluation to enable easy extension and evaluation of the analysis. Further, we employ MoonSniff scripts [38] of MoonGen on LoadGen and timestamper to transmit and record minimally sized packets with 64 B. We assign identifier numbers to transmitted packets for correlation and timestamp the packets using the respective hardware timestamping features. Previous studies suggest that the primary factor relevant for packet processing is the number of packets, and not their size [18, 20].

We use Debian Bullseye 11 (kernel 5.10) and execute a libmoon [38] layer two (l2) forwarding application to minimize the impact of the application itself unless specified otherwise. The forwarding application runs inside the container to analyze the impact of packet processing within the container itself. All experiments employ the packet rates from 10-1000 kpackets/s to analyze the effect on latencies. The number of data points collected depends

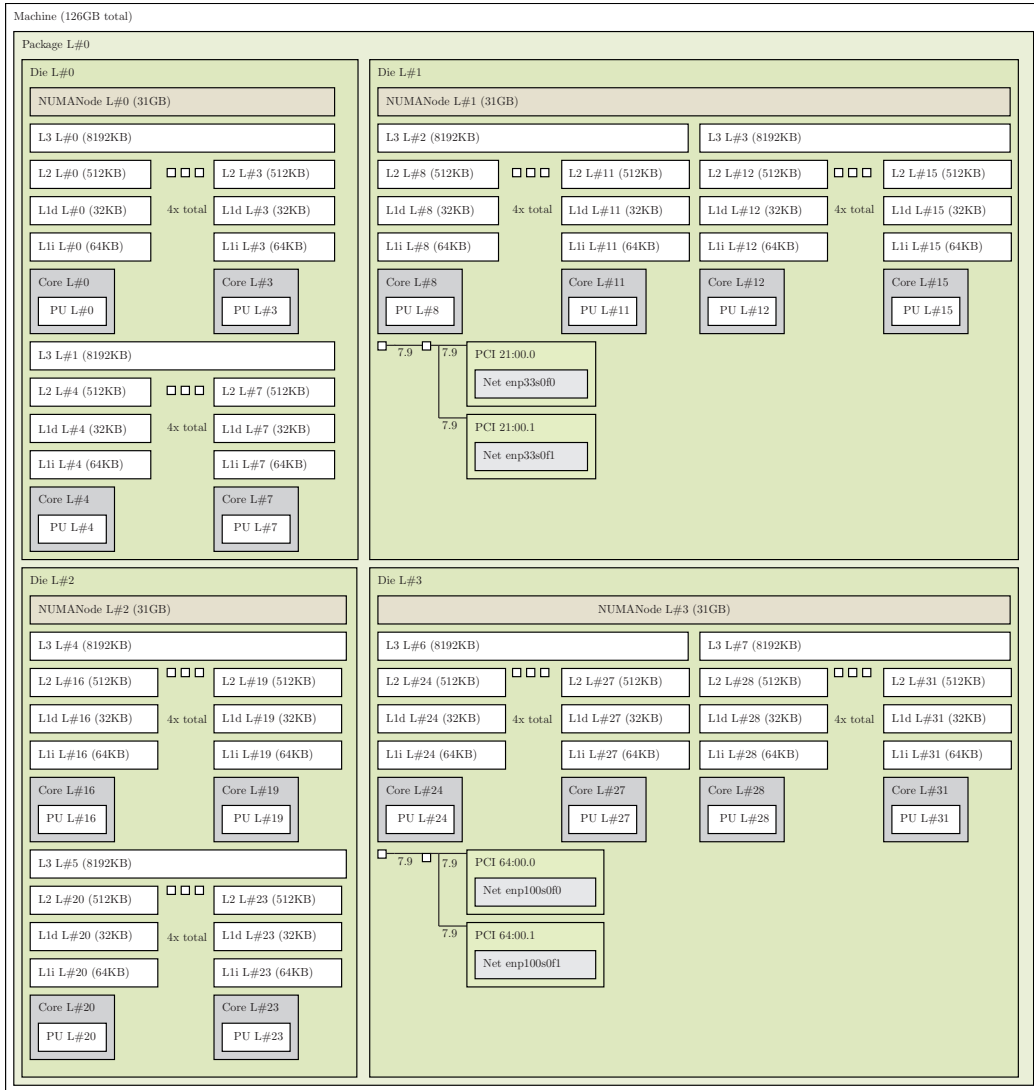


Figure 4: Logical View of the hardware machine in Scenario 1 with only used PCI lanes.

on the packet rate; for example, with 1 Mpkts/s, we collect 160 million data points per experiment in 160 s.

To provide simple automation and reproducible test execution, we adopt the plain orchestration service (POS) for testbed management and experiment execution [53]. Using this concept, we could use the same scripts and technologies as in [4] to perform comparable measurements. In Section 9,

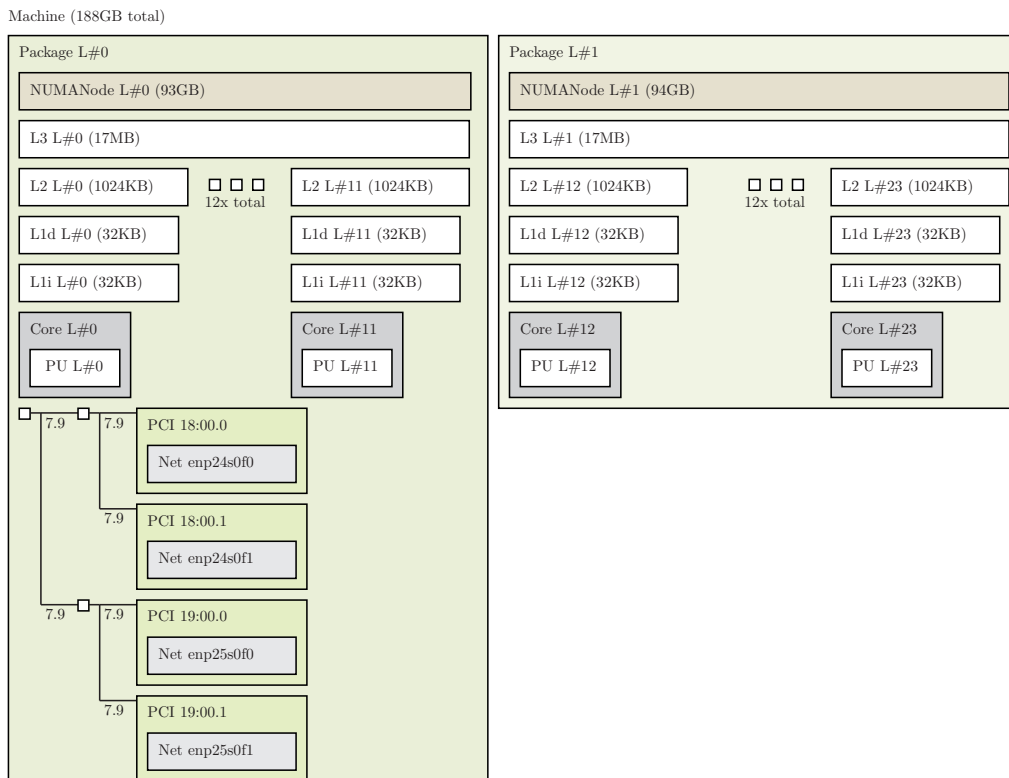


Figure 5: Logical View of the hardware machine in **Scenario 2** with only used PCI lanes.

we describe the access to the different data and scripts to enable reproducing all of our results simply and easily. We encourage readers of this paper to analyze the webpage<sup>1</sup> and the provided extra materials to gain a deeper understanding of the evaluation and the utilized technologies.

## 5. Evaluation

Our evaluation of tail latency behavior in packet-processing containers utilizes the optimizations described in Section 3 and focuses on the two hardware scenarios described in Section 4, including the influence of OS kernel variants, such as a real-time (RT) kernel, and a vanilla kernel. We assess different packet rates, compare the outcomes to those of VMs and bare-metal

<sup>1</sup><https://wiednerf.github.io/container-in-low-latency/>

and analyze if the observed behavior allows the usage within URLLC applications. Furthermore, we evaluate all measurements in comparison between an AMD architecture in **scenario 1** and an Intel architecture in **scenario 2**. **scenario 1** and its evaluation have already been published in [4]; we use the results obtained there to compare them to **scenario 2**.

### 5.1. Scenario

We devised a straightforward scenario depicted in Figure 3 to examine precise low-latency behavior in container setups. Packets are generated externally, transmitted to the DuT, and forwarded through a basic packet processing application—a l2 forwarding application of libmoon—before being sent back over another link. The forwarding application operates within the analyzed system.

With the libmoon l2 forwarding application, we examine the latency induced by network processing, hardware, and virtualization rather than the application itself. Utilizing, for example, layer three (l3) forwarding application adds a constant additional delay on top of the results examined in this article such as [23] have described for Surricata, an intrusion detection system operating on layer three. Moreover, all l3 and upper layer forwarding and packet processing applications must process at least l2 and manage the forwarding tasks. Therefore, a l2 forwarding application provides a valid baseline for our measurements and comparison of optimization techniques. We analyzed a l3 sample forwarding application and compared the results with the l2 variant resulting in lower maximum packet rate and shifted latency. This approach enables the evaluation of the effect of optimization techniques and the underlying hardware system in isolation. We compare the same application on VMs and bare-metal. Through these results, we provide recommendations for using low-latency container applications and identify any limitation

We investigated packet processing in containers using a vanilla Debian OS without optimizations and studied the occurrence of latency spikes over time, as is shown in Figure 6. The figure displays the 5000 worst latency events over time with high peaks induced by interrupts and rescheduling. High latency spikes occur at specific points in time, with all other latencies below 200  $\mu$ s. A periodic, recurring pattern can be seen in the worst-case delays below 200  $\mu$ s caused by rescheduling interrupts regularly emitted on all cores. When we analyze now the worst-case latency based on the requirements of

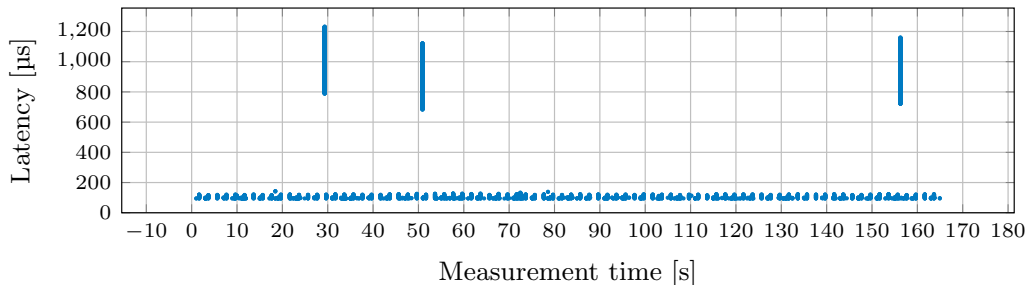


Figure 6: 5000 worst latency events for measurements using LXC-containers based on Debian 11 in `scenario 1` with 1 Mpkts/s.

URLLC for flows to have end-to-end-latencies of  $<1$  ms at the 99.999<sup>th</sup> percentile, we already break these requirements with a one-hop scenario without further optimizations. To establish the validity of our findings, we repeated all experiments multiple times, selecting the measurement with the worst tail-latency for evaluations to ensure the capture of rare events. These results show why examining different options, optimizations, and hardware configurations is essential to provide valid suggestions for using specific technologies for different use-case scenarios.

## 5.2. Packet Rates

We analyze the latency by comparing the effects on systems with and without optimizations discussed in Section 3, starting with comparing different packet rates. Figure 7a illustrates the behavior of non-optimized vanilla compared to the optimized RT variant presented in Figure 7c in `scenario 1`. The logarithmic plots show the latency against the percentiles using high-dynamic-range (HDR) diagrams [54]. By using HDRs in our evaluation (e.g., Figures 7a and 7c), we focus on tail latency events to analyze rare latency spikes by providing a logarithmic scale on both axes to focus on the differences in the tail-latencies specifically. Across all the rates, the optimized and non-optimized systems in `scenario 1` exhibit similar tail-latency behavior at the 99.99<sup>th</sup> percentile and 99.9995<sup>th</sup> percentile, respectively. Lower packet rates are more susceptible to high latencies in lower percentiles; for instance, at 200 kpkts/s, more than 50% of all packets reach a maximum latency of 110  $\mu$ s. Additionally, all measurements indicate higher median latencies when the packet rate is decreased, which is suspected to be due to the lower number of measured packets. The measurements of `scenario 2`, presented in

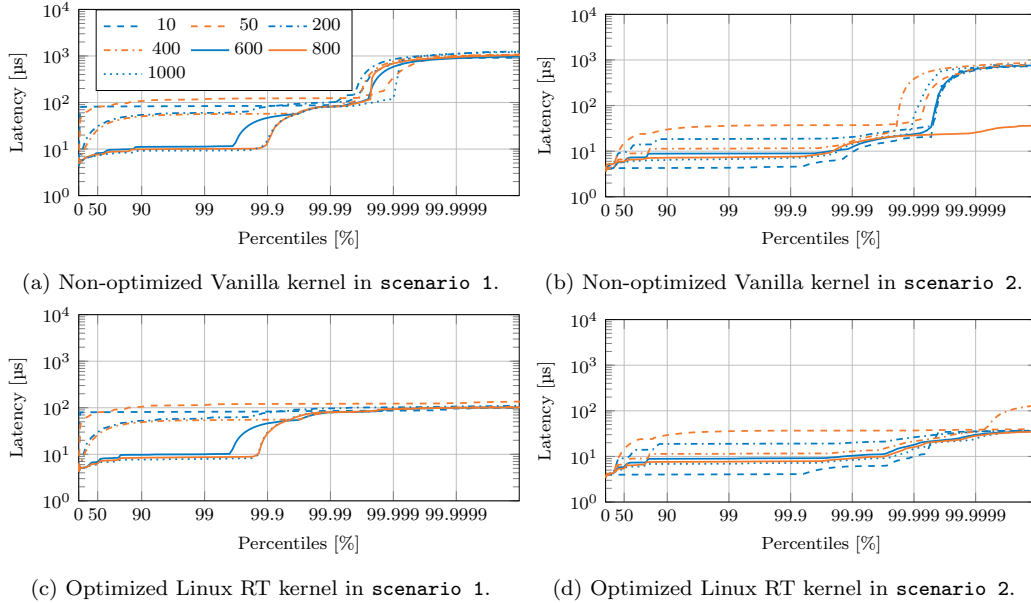


Figure 7: HDR diagram of latency for selected packet rates (kppts/s), legend in Figure 7a.

Figures 7b and 7d, depict a similar behavior until the 99.99<sup>th</sup> percentile is the optimized and non-optimized behavior similar to each other.

In general, in the non-optimized variant, the results in both scenarios show significant spikes. These spikes are caused by a higher number of Translation Lookaside Buffer (TLB)-shutdown interrupts on the respective core and rescheduling events. These shutdowns are executed when the processor changes the mapping between virtual and physical memory addresses to notify all other processors with associated caches to invalidate their respective mapping. This high number of TLB shutdowns issuing a higher spike difference in **scenario 2** is caused by the different designs of the layer three caches between cores in our used machine in Figure 5 and, therefore, results in the recommendation to consider cache design when deciding if containers can be used for low-latency especially when optimizations are not possible due to scalability issues. Figure 8 shows the corresponding recorded interrupts over time in a normalized function and the 5000 worst-case events of the exact measurement using a vanilla non-optimized kernel in **scenario 2**. Here, the highest outliers clearly result from a mixture of TLB shutdowns and rescheduling interrupts called next to each other.

At lower packet rates, the impact on percentiles for rare events is higher

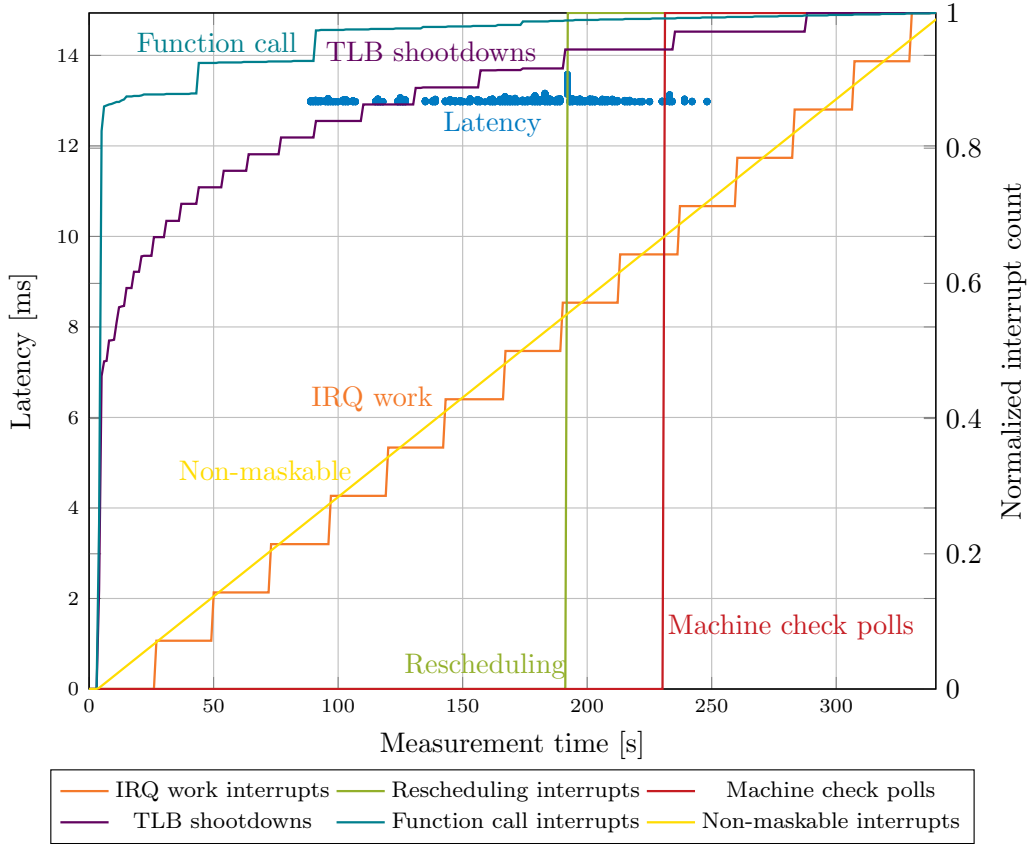


Figure 8: 5000 worst case events over time and corresponding interrupt events normalized over time in **scenario 2**.

because fewer packets are captured within the same measurement time and the amount of packets leaves more space for delays due to the reduced number of packets processed. Hence, we conclude that rare occurrences have a higher impact at lower rates, and no further observations of reasons for the system were made. Therefore, higher packet rates are more suitable for tail-latency analysis, whereas higher rates are insufficient for median analysis due to the significant differences in median behavior. Similar to the measurements in **scenario 1**, the differences in **scenario 2** between the different rates are more significant until the 99.99<sup>th</sup> percentile than in the tail-latency. The tail-latency ranges from 5  $\mu$ s to 50  $\mu$ s across the different rates. Overall, measurements indicate that all evaluated variants can process 1 Mppts/s with minimally sized packets without packet loss used in the following evaluations.

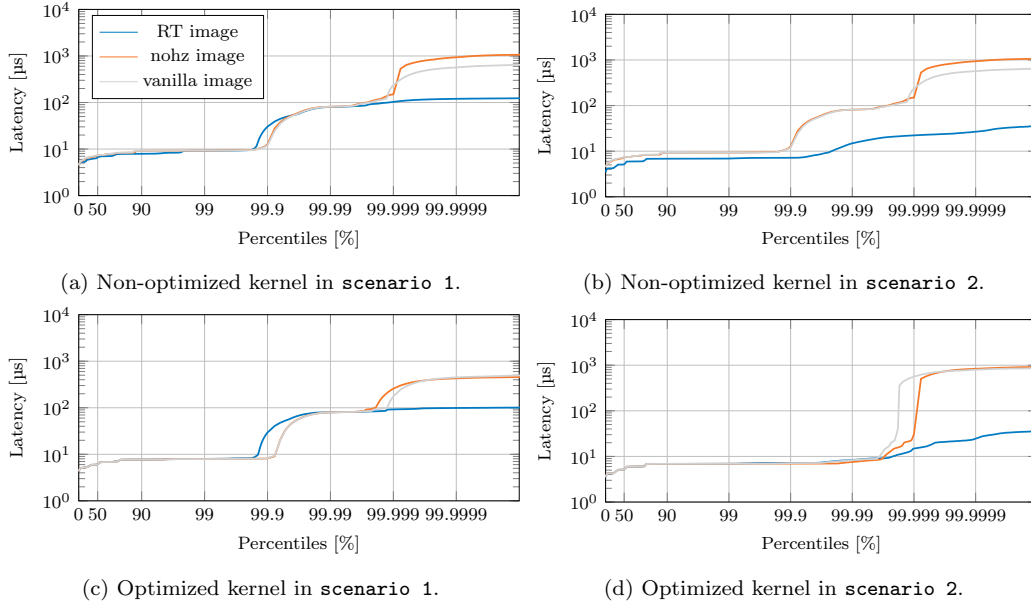


Figure 9: HDR diagram of Debian kernel variants at 1 Mpkt/s, legend in Figure 9a.

### 5.3. Optimizations

Additionally, we investigate the difference in OS kernel variants by including experiments with vanilla, *nohz*, and RT kernel. The *nohz* kernel uses a kernel configuration parameter to remove timer ticks from cores with only one active process. To utilize this kernel features, the kernel must be built with the `CONFIG_NO_HZ_FULL` parameter and the `nohz_full` bootparameter needs to be added. The bootparameter describes which cores should be isolated from the timer tick, similar to the `isolcpu` parameter. In contrast, the RT kernel provides deterministic behavior, which should improve latency reliability behavior. For the RT kernel we used the specific RT-preempt kernel shipped with Debian 11 available via the package repositories. This allows us to utilize a common-used RT image available for broad usage without requiring us to build the kernel extra.

The results of the three kernel variants are presented in Figure 9. Figure 9a shows the measurement results for **scenario 1** for the non-optimized OS in the three kernel variants, revealing a high tail-latency spike towards  $1000\ \mu\text{s}$  at the 99.999<sup>th</sup> percentile when using the *nohz* and vanilla kernel. All kernel variants exhibit latency spikes at the 99.9<sup>th</sup> percentile. The RT kernel variant records a maximum latency of around  $140\ \mu\text{s}$ . When comparing



these results to **scenario 2** in Figure 9b, we can see similar behavior with the RT variant compared to the vanilla variant providing lower latencies as in **scenario 1** until at least the 99.999<sup>th</sup> percentile; The RT variant is in **scenario 2** in general lower compared to **scenario 1**, which is in line with results from previous studies on similar machines such as [23]. In contrast, Figure 9c shows the results for the optimized OS in **scenario 1**, where the RT variant had a slightly lower maximum latency of 110  $\mu$ s compared to the non-optimized RT variant. Moreover, the *nohz* and vanilla kernel variants lead to significantly lower tail latencies at around 510  $\mu$ s compared to the measurements using kernels without optimization. When analyzing the optimized results of **scenario 2**, the latencies up to the 99.99<sup>th</sup> percentile remain below the limit of 10  $\mu$ s compared to the non-optimized version and the optimized version in **scenario 1** which only stays until the 99.5<sup>th</sup> percentile below this margin. Moreover, the tail-latency for **scenario 2** is still receiving multiple TLB-shutdowns over time, resulting in a high tail-latency spike reaching even higher tail-latencies in vanilla and *nohz* kernel-variants compared to the non-optimized version due to more batched interrupts. The optimized RT kernel variants, however, do not receive such interrupts due to better isolation and show stable tail latencies. In all cases, the RT kernel provides the lowest average and tail latencies throughout the tested scenarios and is holding the requirements for URLLC.

Concerning tail latency, the optimized variants outperform the not optimized ones, which are affected by interrupts and rescheduling. Similar to the findings of related work on bare-metal [23], our measurements indicate that the *nohz* variant attains nearly the same tail latencies as the vanilla one on Debian 11. With our measurements, we can even provide further insights: the *nohz* variant with Debian 11 provides no longer improvements independent of the underlying hardware systems in **scenario 1**, **scenario 2**, and in the paper [23]. The RT variant displays similar results to the vanilla one up to 99.99<sup>th</sup> percentile of latencies. However, latencies do not increase further, limiting the tail latency in **scenario 1** and **scenario 2**. The *nohz* kernel can only provide benefits when no scheduling is needed, but LXC requires this to schedule the container engine on the relevant cores. Meanwhile, the RT kernel provides more stable tail latencies due to the deterministic behavior of the OS kernel operations.

In Figure 10a, the 5000 worst-case events of our baseline measurements using the non-optimized vanilla variant are compared to those of an optimized RT one in **scenario 1**. The behavior of the optimized RT kernel variant is

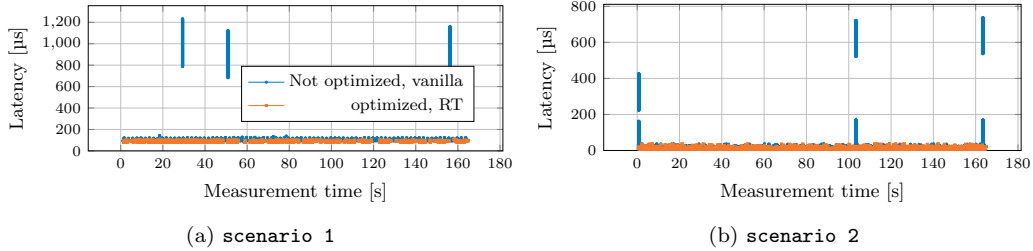


Figure 10: 5000 worst latency events for measurements using LXC-containers based on Debian 11.

similar to that of the non-optimized vanilla kernel variant without the rare, significant outliers that are caused by rare interrupts scheduled on the packet processing core, which is similar to the results in Figure 10b for **scenario 2**, especially in the optimized RT variant. Here, the outliers correlate in terms of time nearly to the distances seen in the non-optimized version. Due to different setup times, this observed movement of worst-case events over time cannot be depicted in the optimizations, whereas the reduced higher latencies, especially after 4 s, are dependent on rescheduling interrupts as well as the optimizations in this case as well show that the outliers are minimal lower in comparison. In all scenarios when utilizing the optimized variant with RT-kernel, we can stay below the 1 ms at the 99.999<sup>th</sup> percentile supporting the URLLC target in comparison to the vanilla, non-optimized variant.

#### 5.4. Container vs. VMs

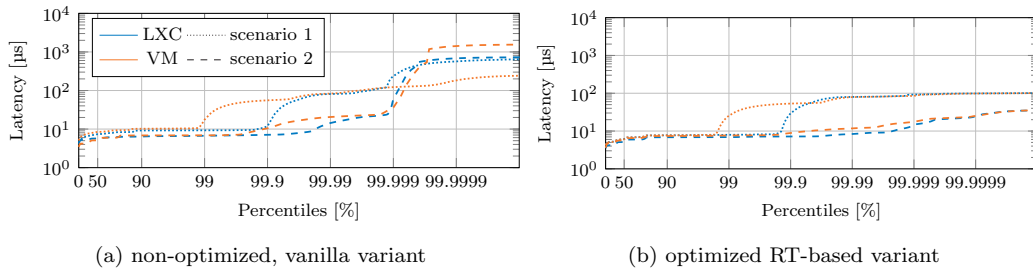


Figure 11: HDR diagram of latency on VMs and LXC containers.

Figure 11 compares measurements between container and VM setups using an optimized RT and a non-optimized vanilla variant. We execute the forwarding application for this experiment inside the VM compared to inside

the container. The VMs are operating as KVM instances on the DuT. In our measurements, the RT kernel variant demonstrates better results, which we attribute to the utilization of Debian 11 in contrast to Debian 10 and, subsequently, a newer kernel used in related work [8, 2].

As presented in Figure 11a, the performance differences between VMs and containers are insignificant concerning tail latencies in both scenarios. When we compare the results from both scenarios, the result for VMs as described in related work [8] shows a significant improvement using the optimizations compared to the non-optimized variant of reaching a maximum of less than  $30\ \mu\text{s}$ . This results from the better isolation properties of the Intel-based CPU compared to reducing the kernel influence, which is impossible for LXC containers due to their usage of a shared kernel. It results in a general evaluation that VMs in the vanilla variant show lower tail latencies due to better isolation by default in both scenarios. However, VMs with higher overhead reach the same maximum tail latency at lower percentiles for all measurements and scenarios. By carefully isolating and optimizing containers, results similar to VMs can be achieved, requiring fewer resources.

### 5.5. Container vs. Bare-metal

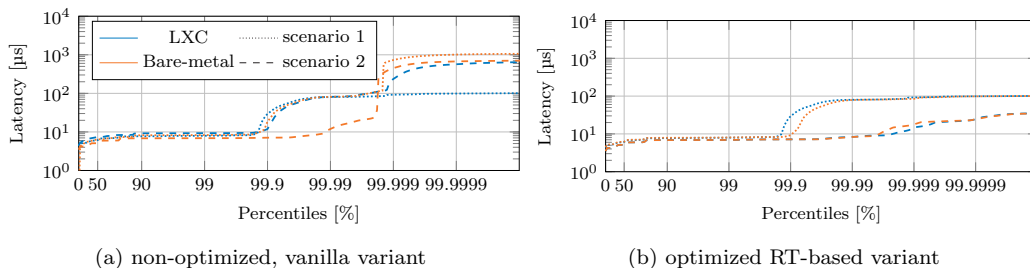


Figure 12: HDR diagram of latency on bare-metal and LXC containers.

The forwarding application is executed directly on bare metal compared to running inside a container for this comparison. The vanilla variant for containers, which provides minimal container isolation by default, yields slightly lower tail latencies (Figure 12) in both scenarios. Meanwhile, optimized bare-metal experiments slightly outperform optimized containers primarily due to a higher degree of isolation of the forwarding application from interrupts. Our findings differ from those reported in [23] as we use a different mainboard. Specifically, [23] used an Intel mainboard with Intel CAT for pinning

the cache to a specific core, which AMD does not support to our current knowledge. Furthermore, they used a DPDK l2fwd application without Lua as a wrapper, which can result in additional latencies caused by the wrapper. Although bare-metal configurations are generally preferable due to better resource isolation, containers may be used when resource sharing is necessary. We now utilize an Intel-based mainboard in our **scenario 2**. In Figure 12 in both sub-figures, the same assumptions as by [23] can be drawn for the bare-metal case, resulting in promising results with a much lower tail-latency in the optimized variant of lower than  $30\ \mu\text{s}$  similar to the results using VMs. This result stays in line with previously presented results.

To conclude, containers exhibit only a minimal overhead compared to bare-metal, contingent on the underlying hardware, which may vary. The overhead and improvement possibilities of containers in comparison to bare-metal and VMs vary depending on the cache design of the individually selected hardware node. Consequently, hardware selection is critical when ultra-low-latencies are required.

### 5.6. L3 vs. L2 forwarding application

Moreover, to show our results on a more complex and processing-heavy application, we additionally analyzed it using the l3-sample-forwarding application of the DPDK project. This forwarding application adds additional overhead and processing as to an opposite to the l2 forwarding scenario; the program has to perform parsing of the packet until and including the l3 header. The l3 application performs a lookup process using longest-prefix-matching in the forwarding information base. Due to unavailability to the original timestamper in **scenario 1** for the l3 forwarding scenario, we used for both **scenario 1** and **scenario 2** the timestamper from **scenario 2**, with specifications as shown in Table 3.

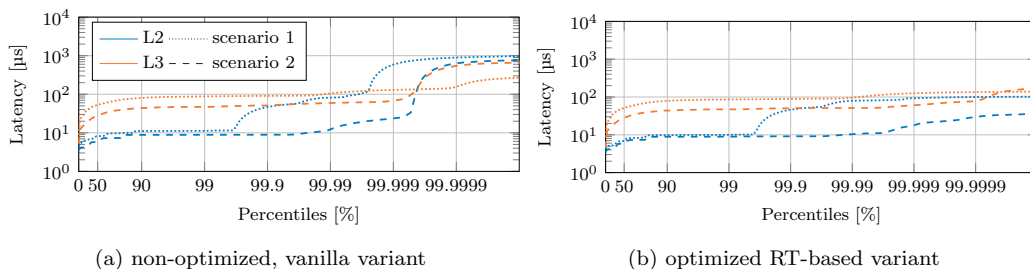


Figure 13: HDR diagram of latency using l3 vs. l2 forwarding application on 600 kpkts/s.

Figure 13 shows a comparison between l2 and l3 forwarding application with 600 kpkts/s as this was the last rate below reaching an overload scenario in the non-optimized variants of the l3 forwarding application. This results in the first major difference: the l3 forwarding application on our systems, due to its additional processing overhead, cannot process as many packets per second as the simple l2 forwarding application.

In Figure 13a the non-optimized, vanilla results for both scenarios with l2 and l3 forwarding applications are shown. Until the 99.99<sup>th</sup> percentile all, we see a clear latency shift due to the higher processing overhead. In the worst-case delay, this differs between scenarios and l2 and l3. In **scenario 1** we reach even lower latencies in the worst-case scenario. This lower tail latency could be due to not processing a packet during a TLB shutdown or a rescheduling interrupt. Whereas in **scenario 2**, the influence of those system interrupts is the same for both forwarding applications. This result could be due to the already high added latency from the interrupts and two processing cores in the l3 application compared to only one in the l2 example.

When comparing this results to Figure 13b with the RT-optimized variant, we see during the whole period a clear shift similar to the non-optimized variants, which is reduced after the 99.99<sup>th</sup> by the results from the l2 forwarding application and the non-optimized variants. All results show a reduced influence of the higher processing costs on tail latencies. This evaluation shows that analyzing a simple l2 application is not enough for the average cases but already shows nearly the same results for tail latencies due to the small added processing time for our sample l3 application. With this, the influence of the network, when the processing overhead is not considerable, is more significant. Analysis of our l3 sample application shows that retrieving recommendations based on our baseline results is valid for further analysis, even for applications with higher processing times.

### 5.7. User-space vs. Kernel-space Network Driver on Container

Thus far, networking in user space has been examined. We investigated a Linux traffic control mirroring application for comparison with kernel-space networking. The application enables the assessment of the impact of kernel-space networking.

Figure 14 shows the tail latency for kernel-space networking compared to user-space networking. The tail latency after the 99.999<sup>th</sup> percentile was similar for both variants of packet processing in **scenario 1**. The optimized

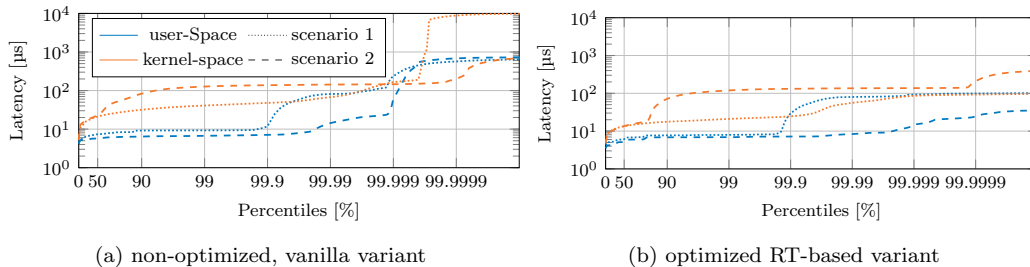


Figure 14: HDR diagram of latency on kernel- versus user-space packet processing.

variants in **scenario 1** measured a maximum latency of  $110\ \mu\text{s}$  in experiments. Compared to the 99.8<sup>th</sup> percentile for user-space packet processing, the latency increases gradually for the 50<sup>th</sup> percentile. This result suggests that the optimizations employed for user space can be used for kernel-space processing. Thus, container isolation enables the use of kernel drivers for low-latency applications. Similar to the experiments on the container with **scenario 1**, kernel-space networking in a container in **scenario 2** are significantly differing, wherein the optimized variant until the 99.99<sup>th</sup> percentile user-space networking is significantly better than kernel-space until the tail-latency, where kernel-space is exhibiting an additional spike compared to user-space networking.

In the non-optimized variant, the tail latency in kernel-space networking exhibits worse performance, with a difference of at least one order of magnitude in **scenario 1** as presented in Figure 14a. In **scenario 2** kernel- and user-space networking are in the non-optimized variant performing in the tail-latency similar to each other. However, comparing kernel- and user-space packet processing in containers reveals that both are valuable for low-latency applications in containers, provided that optimization techniques are carefully used and the hardware machines are carefully selected. Evaluating, in general, using user-space networking is an advantage as it outperforms kernel-space in the non-optimized variant. The tail latency in the optimized variant does not differ significantly depending on the machine, even though the kernel networking outperforms user-space.

## 6. Tail Latency Model

We apply EVT to create models of the tail-latency behavior. We evaluate the predictive capability of these models on a four-fold time horizon.

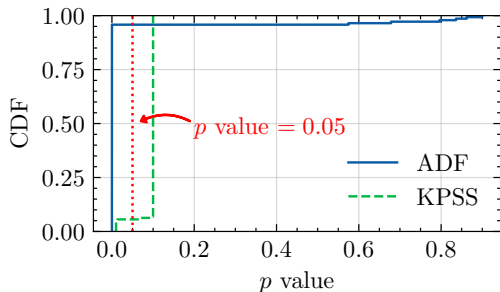


Figure 15:  $p$  values of both ADF- and KPSS tests, evaluated over the EVT part of all datasets.

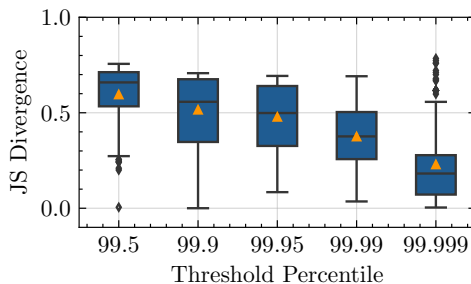


Figure 16: Jensen-Shannon divergence between GPD model and measured latencies over all datasets.

### 6.1. Prerequisites

The condition for the applicability of EVT is that the dataset needs to be identically distributed and stationary [55]. We assume identical distribution and apply two tests to verify stationarity. The two tests are the Augmented Dickey-Fuller (ADF) test [56], and the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test [57]. ADF tests for the presence of a unit root while KPSS tests for the absence of a unit root. The absence of a unit root is interpreted as stationarity. Figure 15 shows the distribution of the  $p$  values of the two tests as applied to all datasets. We only consider the part of the datasets that is used to generate the EVT models, not the part that is used to evaluate them. We can observe that 95% of  $p$  values of the ADF test are less than 0.05 and 95% of  $p$  values of the KPSS test are larger than 0.05. This leads us to reject the null hypothesis of the ADF and accept the null hypothesis of the KPSS test, meaning that 95% of our datasets are stationary. We refer to Wasserstein et al. [58] for a discussion on issues of using  $p$  values. Furthermore, we compared the test statistics of both tests with their respective critical values, leading to the same conclusion of stationarity.

### 6.2. Methodology

Each model is generated using the first 20% of measurement data points and validated using the following 80%, equivalent to a four-fold time horizon. The model is derived using the Peaks-over-Threshold (PoT) method with a variably set threshold. The PoT method classifies all data points larger than a given threshold as belonging to the tail of the latency distribution. A less-used alternative is the Block Maxima approach, which selects

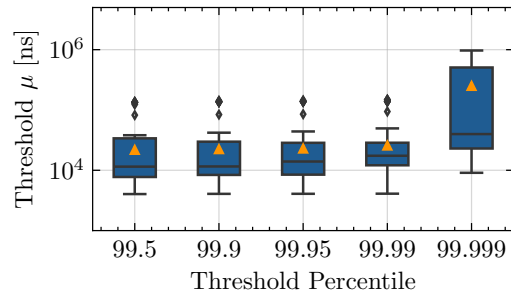


Figure 17: The absolute threshold latency values of all EVT models over the different threshold percentiles.

maximum values from fixed-sized intervals instead. The threshold for the PoT method is selected from the set  $\mathbb{T} = \{99.5, 99.9, 99.95, 99.99, 99.999\}$  using an entropy-based method. The entropy-based method is the Jensen-Shannon (JS) divergence [59]. It is an adaption of the commonly used Kullback–Leibler divergence [60] with the additional advantages of being symmetric and not yielding infinite values. It is used to measure the distance between two probability distributions. We are using it as a goodness-of-fit test for each threshold. An alternative method for threshold selection is the parameter-stability approach [42, 44].

Figure 17 shows the absolute threshold latency values of all EVT models for each threshold percentile. We can observe that the mean threshold value is approximately  $70 \mu\text{s}$ , i.e., latencies above  $70 \mu\text{s}$  are considered for the EVT model.

The data points obtained from the PoT method are then fit to a Generalized Pareto Distribution (GPD) using a maximum likelihood estimator with a confidence level of 95%. A GPD is characterized by three parameters: the threshold ( $\mu$ ), the scale ( $\sigma$ ), and the tail ( $\xi$ ). The JS divergence is then used to select the best-fitting model between the five thresholds. Figure 16 shows the JS divergence for all datasets and all thresholds.

This GPD model can be used to calculate the return level  $x_m$  [42] for an arbitrary return period  $m$ , as shown in Equation (1), with the threshold  $\mu$ , scale  $\sigma$ , tail  $\xi$ , the number of latency measurements  $D$ , and the number of latency values above the threshold  $D_{d>\mu}$ . The fraction  $\frac{D_{d>\mu}}{D}$  is the proportion of latencies larger than the threshold. The return level represents the magnitude of latencies, which is expected to be exceeded exactly once during the return period [42]. It can be considered a lower bound on the expected



worst-case for a given run time of the system. This can be seen as a complementary measure to worst-case latency bounds obtainable by analytical methods, such as network calculus.

$$x_m = \mu + \frac{\sigma}{\xi} \cdot \left[ \left( m \cdot \frac{D_{d>\mu}}{D} \right)^\xi - 1 \right]. \quad (1)$$

The limit of this function for  $m \rightarrow \infty$  describes the behavior of the tail latencies on an arbitrarily long time horizon. In the following, we will especially consider whether a model converges or diverges. This can be used as an indicator of bounded or un-bounded expected tail latencies.

We validated the models by comparing the convergence behaviors and the relation between the return levels and the remaining 80% of the data.

### 6.3. Results

Table 4 shows the results of applying the EVT models to the evaluation data of **scenario 2**. First, we will consider the number of exceedances per model. They can be used to classify the models into two classes: models with good predictive capabilities and models with poor predictive capabilities. We can observe models with good predictions for bare-metal, optimized VMs, optimized containers with RT kernel, and NoHz containers. Decent predictive performances are associated with models for containers and VMs with vanilla kernel, as well as optimized kernel networking stack with a RT kernel. Poor predictive performance is observed for containers with RT kernel but without any optimizations as well as for the unoptimized version of the kernel networking stack. This is mostly in line with the observed experimental results in Section 5.

Next, we will consider the percentage of bounded models. A bounded model has a finite return level for arbitrary return periods. We can observe that all models, except for kernel networking, have at least 50% bounded return levels, while most models have a higher percentage. Overall, there is no clear correlation between mean number of exceedances and percentage of bounded models with a Pearson correlation coefficient of  $-0.02$ .

A more detailed comparison of the tail parameter of the EVT model and the boundedness of the corresponding model is shown in Figure 18. Table 5 shows the same evaluation for **scenario 1**, as described in [4].

We conclude that measured tail-latencies on almost all variations of containers, optimized and unoptimized, are more predictable than on VMs

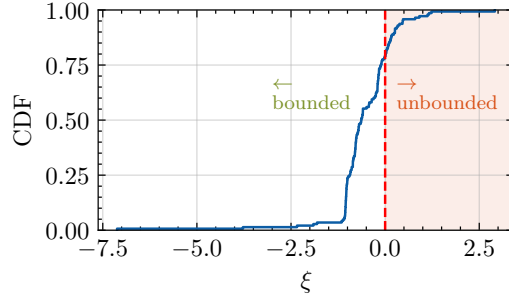


Figure 18: Distribution of tail parameter values  $\xi$  compared to the bounded and unbounded model regions.

Table 4: Results of applying the derived EVT models to the evaluation data from **scenario 2**. The exceedances are mean values for the number of packet bursts that have exceeded the predicted return level. This evaluation extrapolates the model to a four-fold time horizon. The expected number of exceedances is exactly 1. The bounded column indicates the percentage of EVT models that have a finite return value for an infinite return period.

Platform	Opt.	RT	NoHz	Vanilla	# Exceedances	Bounded
Bare-metal	✓	✓	✗	✗	0.83	58.3%
Bare-metal	✗	✗	✗	✓	1.33	100.0%
VM	✓	✓	✗	✗	1.25	50.0%
VM	✗	✗	✗	✓	2.58	66.7%
Container	✓	✓	✗	✗	1.42	83.3%
Container	✗	✓	✗	✗	7.67	100.0%
Container	✓	✗	✓	✗	1.25	75.0%
Container	✗	✗	✓	✗	1.67	83.3%
Container	✓	✗	✗	✓	2.92	75.0%
Container	✗	✗	✗	✓	2.29	71.4%
Kernel Netw.	✓	✓	✗	✗	2.50	33.3%
Kernel Netw.	✗	✗	✗	✓	22.73	63.6%

and in the kernel networking stack. The predictability between bare-metal and containers is roughly equal.

## 7. Recommendations for Low-latency-sliced Applications

Table 6 presents the tail-latency of the non-optimized vanilla and the optimized RT variant for each technology and scenario. We recommend a

Table 5: Results of applying the derived EVT models to the evaluation data for **scenario 1**.

Platform	Opt.	RT	NoHz	Vanilla	Exceedances	Bounded
Bare-metal	✓	✓	✗	✗	3.30	60.0 %
Bare-metal	✗	✗	✗	✓	0.33	16.7 %
VM	✓	✓	✗	✗	4.00	58.3 %
VM	✗	✗	✗	✓	2.58	25.0 %
Container	✓	✓	✗	✗	3.83	66.7 %
Container	✗	✗	✗	✓	1.50	16.7 %

Table 6: Tail-latency values for non-optimized vanilla and optimized RT version for all three systems with user-space networking and kernel-space networking on containers only.

Technology	Scenario	non-optimized vanilla	optimized RT
<i>container</i>			
kernel-space	1	9969.08 $\mu$ s	100.19 $\mu$ s
	2	744.72 $\mu$ s	414.52 $\mu$ s
user-space	1	659.25 $\mu$ s	108.86 $\mu$ s
	2	373.73 $\mu$ s	36.98 $\mu$ s
<hr/>			
<i>VM</i>	1	840.63 $\mu$ s	124.12 $\mu$ s
	2	1560.47 $\mu$ s	37.02 $\mu$ s
<hr/>			
<i>Bare-metal</i>	1	1077.44 $\mu$ s	101.81 $\mu$ s
	2	716.16 $\mu$ s	36.74 $\mu$ s

top-down strategy for choosing a system for URLLC based on the presented findings. While a bare-metal solution is best suited for commodity hardware, responding to on-the-fly demands can be resource-intensive and challenging. Our measurements demonstrated that VMs and containers can perform similarly to each other when selecting the best underlying hardware architecture for the selected technology. Therefore, we recommend analyzing additional aspects such as security, resource usage, and hardware system design. We recommend using containers to ensure reduced resource usage and high flexibility when the hardware provides a fine-granular separated cache per core group. However, if higher security and isolation of applications are required, VMs are recommended in all hardware scenarios. Containers and VMs can be

hosted on the same hardware system, providing both to customers. When analyzing those scenarios toward holding the URLLC requirements, it becomes visible that especially our non-optimized vanilla scenario in all combinations cannot hold the URLLC requirements, whereas the optimized variant of the RT image can hold the requirements in all versions. Using bare-metal or VMs provides, on the other side, the most headroom for additional application-induced latency, which needs to be kept in mind. This is, for example, already needed when utilizing l3 forwarding instead of l2 forwarding due to the added constant delay.

Table 7: Summarized Recommendations with ranks for each category from ✓✓✓ (best) to ××× (worst).

Technology	Cache	Latency	Security	Resources
<i>VM</i>				
optimized	Separated	✓	✓✓	×
	Shared	✓	✓	×
non-optimized	Separated	×××	✓✓	✓
	Shared	×××	✓	✓
<i>Container</i>				
optimized	Separated	✓✓	✓	✓✓
optimized	Shared	✓✓	×	✓✓
non-optimized	Separated	×	✓	✓✓✓
non-optimized	Shared	××	×	✓✓✓
<i>Bare-metal</i>				
optimized	Separated	✓✓✓	✓✓✓	××
optimized	Shared	✓✓✓	✓✓✓	××
non-optimized	Separated	××	✓✓✓	××
non-optimized	Shared	×	✓✓✓	××

Table 7 summarizes the recommendations. We have ranked the categories for non-optimized and optimized VMs, containers, and bare-metal solutions for each evaluated cache type. Generally, optimized variants perform better than non-optimized variants in all systems, whereas the choice of technology ultimately depends on the infrastructure, requirements, and available resources.

## 8. Limitations

Our analysis of virtualization overhead was based on single instances, not concurrent ones. Only simple applications were analyzed since we focused on technology-induced latency.

We did not examine shared network resources, such as previous studies [8] and [2] have analyzed for VMs. Further studies are necessary to explore potential improvements and provide a more in-depth analysis of shared system resources.

Our analysis focuses exclusively on the behavior of LXC containers since previous studies indicate that alternative solutions introduce additional overhead [14, 6]. However, this has yet to be verified further, and comparing container solutions for improving latency is part of future research.

Moreover, the approach of direct access to the NIC is not generally scalable. In this article, we did not analyze scalability solutions further, such as single-root IO virtualization, which allows splitting one PCIe device into multiple ones. We leave this analysis to future work.

## 9. Reproducibility

The scripts, raw data, and analysis results required to reproduce our findings are available online, including the raw PCAPs, the data extracted from these PCAPs, and the plots obtained for each measurement<sup>2</sup>. The scripts enable reproducing all measurements and calculations on other systems, provided the necessary hardware is available for per-packet timestamping and passive optical TAPs.

## 10. Conclusion and Future Work

Reliable and predictable low latency is critical in applications such as autonomous driving or remote medical procedures. Resource sharing and on-demand service provisioning, such as network slices in 5G, are also vital. This study demonstrates that lightweight virtualization is a suitable alternative for high-reliability, low-latency applications. However, achieving this requires a tactful selection of optimization parameters, such as rt-kernels.

---

<sup>2</sup><https://wiednerf.github.io/container-in-low-latency/>

We can use a user-space networking application to achieve high reliability and significantly reduce tail latency. Furthermore, VMs and containers exhibit comparable performance; however, containers require fewer resources but cannot be completely isolated. For containers, carefully selecting the underlying hardware architecture, especially the cache design, is much important than for VMs. Therefore, we can recommend containers if the underlying hardware supports the isolation of container resources and provides a corresponding cache system utilizing divided spaces for different containers and CPU cores. We employed an EVT model to assess the predictability of tail latencies in containers. We found that an optimized system in a container with an RT kernel converges more in models than any configuration based on VMs or bare metal. This study presents the first in-depth analysis of packet-processing latency in containers.

Further, we plan to analyze the influence of concurrent containers and CPU resource sharing and evaluate the potential of SR-IOV-based NIC sharing to improve low latency and high availability of resources. We also plan to enhance predictability using statistical methods over time to enable accurate planning. Finally, we plan to investigate and compare different container solutions and their latencies beyond current findings and the effect of using VMs and containers on the same system.

## Acknowledgments

This work was supported in part by the European Union Horizon 2020 research and innovation programme (project SLICES-SC, 101008468, and SLICES-PP, 101079774), the Bavarian Ministry of Economic Affairs, Regional Development and Energy (project 6G Future Lab Bavaria), and the German Federal Ministry of Education and Research (project 6G-ANNA, 16KISK107, and project 6G-life, 16KISK002).

## References

- [1] 5G: Study on Scenarios and Requirements for Next Generation Access Technologies, Last accessed: May 22, 2023 (2017).
- [2] S. Gallenmüller, J. Naab, I. Adam, G. Carle, 5G QoS: Impact of Security Functions on Latency, in: NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020, IEEE, 2020, pp. 1–9.

- [3] RedHat, `cpuset` - Red Hat Enterprise Linux 6, URL: [https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/6/html/resource\\_management\\_guide/sec-cpuset](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpuset), Last accessed: Feb 29, 2024 (2023).
- [4] F. Wiedner, M. Helm, A. Daichendt, J. Andre, G. Carle, Containing Low Tail-Latencies in Packet Processing Using Lightweight Virtualization, in: 35th International Teletraffic Congress (ITC-35), Torino, Italy, 2023, pp. 1 – 9.
- [5] Z. Li, M. Kihl, Q. Lu, 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), 2017, pp. 955–962.
- [6] D. Bernstein, Containers and Cloud: From LXC to Docker to Kubernetes, *IEEE Cloud Computing* 1 (3) (2014) 81–84.
- [7] A. K. Yadav, M. L. Garg, Ritika, Docker containers versus virtual machine-based virtualization, in: A. Abraham, P. Dutta, J. K. Mandal, A. Bhattacharya, S. Dutta (Eds.), *Emerging Technologies in Data Mining and Information Security*, Springer Singapore, Singapore, 2019, pp. 141–150.
- [8] S. Gallenmüller, F. Wiedner, J. Naab, G. Carle, Ducked Tails: Trimming the Tail Latency of(f) Packet Processing Systems, in: P. Chemouil, M. Ulema, S. Clayman, M. Sayit, C. Çetinkaya, S. Secci (Eds.), 17th International Conference on Network and Service Management, CNSM 2021, Izmir, Turkey, October 25-29, 2021, IEEE, 2021, pp. 537–543.
- [9] D. Gedia, L. Perigo, Performance evaluation of sdn-vnf in virtual machine and container, in: 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks, 2018, pp. 1–7.
- [10] F. Wiedner, A. Daichendt, J. Andre, G. Carle, Control Groups Added Latency in NFVs: An Update Needed?, in: F. H. P. Fitzek, L. J. Horner, M. Gharbaoui, G. Nguyen, R. Gu, T. Meuser (Eds.), *IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2023*, Dresden, Germany, November 7-9, 2023, IEEE, 2023, pp. 40–45.

- [11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: M. L. Scott, L. L. Peterson (Eds.), Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003, ACM, 2003, pp. 164–177.
- [12] L. Abeni, C. Király, N. Li, A. Bianco, Tuning KVM, to enhance virtual routing performance, in: Proceedings of IEEE International Conference on Communications, ICC 2013, Budapest, Hungary, June 9-13, 2013, IEEE, 2013, pp. 3803–3808.
- [13] M.-N. Tran, Y. Kim, Network Performance Benchmarking for Containerized Infrastructure in NFV environment, in: 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), 2022, pp. 115–120.
- [14] R. Morabito, J. Kjällman, M. Komu, Hypervisors vs. Lightweight Virtualization: A Performance Comparison, in: 2015 IEEE International Conference on Cloud Engineering, 2015, pp. 386–393.
- [15] J.-G. Cha, S. W. Kim, Design and Evaluation of Container-based Networking for Low-latency Edge Services, in: 2021 International Conference on Information and Communication Technology Convergence (ICTC), 2021, pp. 1287–1289.
- [16] S. Lin, P. Cao, T. Huang, S. Zhao, Q. Tian, Q. Wu, D. Han, X. Wang, C. Zhou, XMasq: Low-Overhead Container Overlay Network Based on eBPF, CoRR abs/2305.05455 (2023).
- [17] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, T. Anderson, Slim: OS kernel support for a Low-Overhead container overlay network, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA, 2019, pp. 331–344.
- [18] T. Zhang, L. Linguaglossa, J. Roberts, L. Iannone, M. Gallo, P. Giaccone, A benchmarking methodology for evaluating software switch performance for NFV, in: 2019 IEEE Conference on Network Softwarization (NetSoft), 2019, pp. 251–253.



- [19] G. K. Lockwood, M. Tatineni, R. Wagner, SR-IOV: Performance Benefits for Virtualized Interconnects, in: S. A. Lathrop, J. Alameda (Eds.), Annual Conference of the Extreme Science and Engineering Discovery Environment, XSEDE '14, Atlanta, GA, USA - July 13 - 18, 2014, ACM, 2014, pp. 47:1–47:7.
- [20] J. Liu, Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support, in: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–12.
- [21] H. Liu, W. Li, Y. Pang, R. Pei, Y. Hu, Y. Liu, L. Suo, K. Li, Accelerating data delivery of latency-sensitive applications in container overlay network, *IEEE Trans. Parallel Distributed Syst.* 34 (12) (2023) 3046–3058.
- [22] NGMN Alliance, 5G E2E Technology to Support Verticals URLLC Requirements (2019).
- [23] S. Gallenmüller, F. Wiedner, J. Naab, G. Carle, How Low Can You Go? A Limbo Dance for Low-Latency Network Functions, *Journal of Network and Systems Management* 31 (20) (Dec. 2022).
- [24] D. Bozilov, M. Knezevic, V. Nikov, Optimized threshold implementations: securing cryptographic accelerators for low-energy and low-latency applications, *J. Cryptogr. Eng.* 12 (1) (2022) 15–51.
- [25] V. Jain, H.-T. Chu, S. Qi, C.-A. Lee, H.-C. Chang, C.-Y. Hsieh, K. K. Ramakrishnan, J.-C. Chen, L<sup>2</sup>5GC: A Low Latency 5G Core Network Based on High-Performance NFV Platforms, in: Proceedings of the ACM SIGCOMM 2022 Conference, SIGCOMM '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 143–157.
- [26] Y. J. Liu, P. X. Gao, B. Wong, S. Keshav, Quartz: A New Design Element for Low-Latency DCNs, *SIGCOMM Comput. Commun. Rev.* 44 (4) (2014) 283–294.
- [27] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, J. S. Rellermeyer, A survey on distributed machine learning, *ACM Comput. Surv.* 53 (2) (mar 2020).

- [28] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, M. Zhang, Duet: Cloud Scale Load Balancing with Hardware and Software, SIGCOMM Comput. Commun. Rev. 44 (4) (2014) 27–38.
- [29] J. Mario, J. Eder, Low Latency Performance Tuning for Red Hat Enterprise Linux 7, URL: <https://access.redhat.com/sites/default/files/attachments/201501-perf-brief-low-latency-tuning-rhel7-v2.1.pdf>, Last accessed: Feb 29, 2024 (Nov. 2017).
- [30] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, M. Wójcik, Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 29–42.
- [31] DPDK Project, Home - DPDK, uRL: <https://www.dpdk.org/>, Last Accessed: Feb 29, 2024 (2024). URL <https://www.dpdk.org/>
- [32] C.-N. Mao, M.-H. Huang, S. Padhy, S.-T. Wang, W.-C. Chung, Y.-C. Chung, C.-H. Hsu, 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), 2015, pp. 611–616.
- [33] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, S. White, Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families : Industrial product, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 57–70. doi:10.1109/ISCA52012.2021.00014.
- [34] S. Hammond, C. Vaughan, C. Hughes, Evaluating the intel skylake xeon processor for hpc workloads, in: 2018 International Conference on High Performance Computing and Simulation (HPCS), 2018, pp. 342–349. doi:10.1109/HPCS.2018.00064.
- [35] K. B. Rao, Computer systems architecture vs quantum computer, in: 2017 International Conference on Intelligent Com-

- puting and Control Systems (ICICCS), 2017, pp. 1018–1023. doi:10.1109/ICCONS.2017.8250619.
- [36] Intel, URL: <https://github.com/intel/intel-cmt-cat>, Last accessed at Feb 29 2024.
- [37] Y. Kim, A. More, E. Shriver, T. Rosing, Application performance prediction and optimization under cache allocation technology, in: 2019 Design, Automation and Test in Europe Conference and Exhibition (DATE), 2019, pp. 1285–1288. doi:10.23919/DATE.2019.8715259.
- [38] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, G. Carle, MoonGen: A Scriptable High-Speed Packet Generator, in: K. Cho, K. Fukuda, V. S. Pai, N. Spring (Eds.), Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015, ACM, 2015, pp. 275–287.
- [39] Snort, Snort - Network Intrusion Detection and Prevention System, uRL: <https://www.snort.org/>, Last Accessed: Feb 29, 2024 (2024). URL <https://www.snort.org/>
- [40] M. Ziese, A. Rauthe-Schöch, P. Finger, E. Rustemeier, S. Hänsel, U. Schneider, Gpcc full data daily version 2022 at 1.0c: Daily land-surface precipitation from rain-gauges built on gts-based and historic data. 2022.
- [41] I. Godfried, K. Mahajan, M. Wang, K. Li, P. Tiwari, Flowdb a large scale precipitation, river, and flash flood dataset (2020). arXiv:2012.11154.
- [42] S. Coles, J. Bawa, L. Trenner, P. Dorazio, An Introduction to Statistical Modeling of Extreme Values, Vol. 208, Springer, 2001.
- [43] T. Hsing, On Tail Index Estimation Using Dependent Data, The Annals of Statistics 19 (3) (1991) 1547–1569.
- [44] M. Helm, F. Wiedner, G. Carle, Flow-level Tail Latency Estimation and Verification based on Extreme Value Theory, in: M. Charalambides, P. Papadimitriou, W. Cerroni, S. S. Kanhere, L. Mamatras (Eds.), 18th International Conference on Network and Service Management, CNSM 2022, Thessaloniki, Greece, October 31 - Nov. 4, 2022, IEEE, 2022, pp. 359–363.

- [45] N. Mehrnia, S. Coleri, Wireless Channel Modeling Based on Extreme Value Theory for Ultra-Reliable Communications, *IEEE Trans. Wirel. Commun.* 21 (2) (2022) 1064–1076.
- [46] M. Bennis, M. Debbah, H. V. Poor, Ultrareliable and Low-Latency Wireless Communication: Tail, Risk, and Scale, *Proc. IEEE* 106 (10) (2018) 1834–1853.
- [47] R. A. Fisher, L. H. C. Tippett, Limiting forms of the frequency distribution of the largest or smallest member of a sample, *Mathematical Proceedings of the Cambridge Philosophical Society* 24 (2) (1928) 180–190. doi:10.1017/S0305004100015681.
- [48] J. P. III, Statistical Inference Using Extreme Order Statistics, *The Annals of Statistics* 3 (1) (1975) 119 – 131. doi:10.1214/aos/1176343003. URL <https://doi.org/10.1214/aos/1176343003>
- [49] A. A. Balkema, L. de Haan, Residual Life Time at Great Age, *The Annals of Probability* 2 (5) (1974) 792 – 804. doi:10.1214/aop/1176996548. URL <https://doi.org/10.1214/aop/1176996548>
- [50] AMD, Performance Tuning Guidelines for Low Latency Response on AMD EPYC-Based Servers Application Note, URL: <https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/56263-EPYC-performance-tuning-app-note.pdf>, Last accessed: Feb 29, 2024 (Jun. 2018).
- [51] Intel Corporation, E810 datasheet rev2.5, URL: [https://cdrdv2-public.intel.com/613875/613875\\_E810\\_Datasheet\\_Rev2.5.pdf](https://cdrdv2-public.intel.com/613875/613875_E810_Datasheet_Rev2.5.pdf), Last Accessed: Feb 29, 2024.
- [52] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: A generic framework for managing hardware affinities in HPC applications, in: M. Danelutto, J. Bourgeois, T. Gross (Eds.), *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*, IEEE Computer Society, 2010, pp. 180–186.
- [53] S. Gallenmüller, D. Scholz, H. Stubbe, G. Carle, The pos framework: a methodology and toolchain for reproducible network experiments, in:

- G. Carle, J. Ott (Eds.), CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021, ACM, 2021, pp. 259–266.
- [54] G. Tene, HDRHistogram: A High Dynamic Range Histogram, URL: <http://hdrhistogram.org/>, Last accessed: Feb 29, 2024 (Aug. 2021).
- [55] L. Santinelli, J. Morio, G. Dufour, D. Jacquemart, On the sustainability of the extreme value theory for WCET estimation, in: 14th International Workshop on Worst-Case Execution Time Analysis, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.
- [56] D. A. Dickey, W. A. Fuller, Distribution of the estimators for autoregressive time series with a unit root, *Journal of the American statistical association* 74 (366a) (1979) 427–431.
- [57] D. Kwiatkowski, P. C. Phillips, P. Schmidt, Y. Shin, Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root?, *Journal of econometrics* 54 (1-3) (1992) 159–178.
- [58] R. L. Wasserstein, A. L. Schirm, N. A. Lazar, Moving to a world beyond “ $p_i 0.05$ ” (2019).
- [59] J. Lin, Divergence measures based on the Shannon entropy, *IEEE Transactions on Information theory* 37 (1) (1991) 145–151.
- [60] S. Kullback, R. A. Leibler, On information and sufficiency, *The annals of mathematical statistics* 22 (1) (1951) 79–86.