Control Groups Added Latency in NFVs: An Update Needed?

Florian Wiedner, Alexander Daichendt, Jonas Andre, Georg Carle
Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: {wiedner, daichend, andre, carle}@net.in.tum.de

***Abstract*—The Linux kernel is continuously developing, and the development teams constantly add new versions of libraries with additional features. The `cgroups` feature of the Linux kernel was recently updated. It is widely used by lightweight virtualization technologies such as Docker or LXC for resource isolation. `cgroups` underwent a significant revamp from version 1 (v1) to version 2 (v2). Researching the performance difference between these two versions regarding network latencies and throughput enables the usage of containers in time-critical applications for Network Function Virtualization (NFV). Previous work does not consider `cgroups` as a potential source of latency in NFV in their evaluations. In this paper, we use commodity hard- and software to measure the difference between v1 and v2 in isolation. Our experiments show that the two versions achieve the same degree of isolation. However, the tail latencies of v1 are higher than v2, which can be explained by a more efficient, 2.4 % less instruction-consuming implementation of v2. Based on our findings, we recommend using v2 for low-latency, lightweight virtualization network deployments wherever possible.**

***Index Terms*—low latency, container, virtualization, cgroups, network function virtualization**

I. Introduction

From inter-vehicle communication in self-driving cars to the coordination of assembly lines, critical applications require technology to operate at peak performance. In such scenarios, even short delays can result in catastrophic consequences. That is why low latencies enable the interaction and coordination of time-sensitive applications over shared networks. To achieve the lowest and most stable network latencies, investing in high-quality networking equipment and thoroughly reviewing the entire software stack is crucial. This investment can enable the migration of existing network applications towards real-time applications.

A common way of handling increasing complexity is to compartmentalize different software components into smaller pieces and run them in isolated environments. This splitting can be achieved through virtualization, using either heavy-weight virtual machines (VMs) or lightweight containers. To achieve optimal low-latency networking, a comprehensive grasp of the chosen virtualization technique is necessary. This understanding becomes particularly crucial when time-critical traffic needs to pass through networking functions, e.g. firewalls or intrusion detection systems. These networking functions are increasingly virtualized and interconnected in chains, making a deeper comprehension of their inner workings more important. Consequently, demand for virtualization solutions with low resource requirements but high isolation and performance features enabling real-time applications emerges. Two essential features to enable such optimizations are namespaces and control groups (`cgroups`).

Namespaces allow processes to have isolated and independent views of the system resources, such as the network, filesystem, or process IDs. `cgroups` provide a way to limit, allocate, and prioritize system resources among processes or groups of processes. Initially, `cgroups` were released in 2007 in kernel 2.6.24 [1] as version 1 (v1). They received an exhaustive revamp [2] with a second version (v2) released in kernel 4.5 with a reduced feature set.

This paper analyzes the latency and throughput performance differences for `cgroups` v1 and v2 towards their implemented features. For network operators and researchers, the paper recommends updating the `cgroups` version particularly when NFV infrastructure is serving real-time traffic. This paper provides the following:

1) A methodology to systematically analyze latency performance differences for containerized systems
2) A detailed evaluation of `cgroups` in container
3) A recommendation for the usage of cgroup versions in NFV

The paper is structured as follows: Section II presents the state-of-the-art with a extra focus on `cgroups` in Section III. Section IV presents the used methodology. Section V discusses the results, followed by Section VI providing the limitations of our used methodology and provided results. Section VII is showing condensed recommendations for further usage of containerized solutions with `cgroups`. Finally, we provide information about the reproducibility of our results in Section VIII, summarize our findings in Section IX, and propose future work.

II. Background and Related Work

Utilizing NFV is an essential step towards enhancing flexibility in network resource usage and planning, leading to more efficient resource usage overall. Presenting a comprehensive review of prior research on virtualization, network performance, and NFV establishes the foundation for this performance evaluation. Understanding the significance of `cgroups` as a vital feature for analysis is facilitated through this background information.

A. Network Function Virtualization

In recent years, the increasing adoption of virtualization techniques has led to the migration of appliances traditionally bind to dedicated hardware such as low-latency packet filtering into virtual environments [3]. This step towards virtualization supports, for example, slicing in 5G networks or on-demand, feature provisioning. Additionally, updating or replacing a system on-the-fly is easier. With specialized hardware instances, this was a complicated and costly procedure. NFV allows leveraging general-purpose commercial-off-the-shelf hardware to accomplish highly specialized networking objectives. This revolutionary approach is made possible through virtualization, enabling the utilization of current off-the-shelf hardware for a diverse range of networking tasks [3].

Each network function is packed as a Virtual Network Function (VNF) or Container Network Function (CNF) into an isolated, virtualized environment. On the one side, this approach requires a good positioning of VNFs to each other in a Service Function Chain (SFC) to have a high performance in traversing through the different functions in a pre-defined order. This positioning was analyzed in different related works such as [4]–[6]. On the other hand, especially for real-time traffic, challenging requirements towards the network traffic performance exists, analyzed in different related publications such as [7], [8]. Kourtis et al. [7] have analyzed the performance exploiting user-space networking using the Data-Plane Development Kit (DPDK). DPDK provides a significant performance improvement, as used in our measurements to mitigate the impact of external factors on the analyzed features.

B. Virtualization Techniques

The basis of CNFs, and VNFs is the ability to isolate applications from each other and the host system residing on the same hardware machine providing tailored services. Two commonly used architectures are available for virtualization: hypervisor- or container-based. Hypervisor-based virtualizations are known as VMs because the complete OS, including the kernel, is virtualized. Container-based virtualization is called lightweight virtualization because only parts of the OS are virtualized. In this case, the kernel is shared between the host OS and container, and mainly processes, files, and resource access are isolated [9]. In Figure 1, the base presents the hardware running the host OS and the top the different types of virtualization available; on the left side, the containerization, including the container engine used to manage the containers; Furthermore, on the right side, a VM shows the additional overhead of the guest OS residing within each system.

Throughput analysis of virtualization techniques is a common area of research. Barham et al. [10] studied the impact of CPU resources on XEN-based VMs, focusing on time-slices on the CPU-induced variations. Tran and Kim [11] found out that CPU core assignment and reservation of cores for specific containers within the system is crucial for improving throughput. Morabito et al. [12] draw a line towards container challenging traditional systems in resource usage and

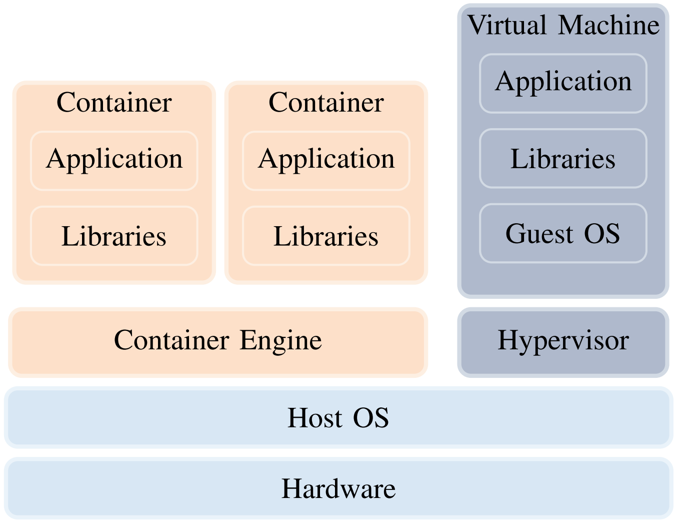

Figure 1. Comparison between container (left) and VMs (right) [9]

performance. In conclusion, extensive research is conducted on throughput and optimization, particularly concerning CNF deployments.

C. Linux Containers

LXC [13] is a low-level container runtime. It provides a minimalistic feature set to remain lightweight. LXC can be controlled using user tools, a C, or Python API. One downside of LXC is that convenience features such as layered images or orchestration are missing entirely. Conversely, this reduces additional overhead.The currently available version on Debian 10 is 4.0 [13].

Previous studies analyzed the performance overhead between different container solutions such as Docker and LXC, concluding that LXC performs best with a minimal overhead. However, the lack of additional features introduces overhead for the administrator in terms of work, orchestration, and management [14], [15]. Putri et al. [16] analyzed the performance differences between LXC, and Docker, concluding that, in general, LXC outperforms Docker containers. In the measured scenarios, the LXC container showed the best IO performance on heavy load. Based on that analysis, we use LXC to compare the performance differences using different cgroup versions on network performance metrics.

D. Performance of NFV inside container

Gedia et al. [8] analyzed the performance impact of software-defined networking applications on VNFs. Concerning networking, they focused on throughput comparing VMs and containers. Their results show that the throughput on containers was significantly higher than the throughput they could reach with VMs using the same setup otherwise. Moreover, Kourtis et al. [7] showed that the most significant improvement on network performance with CNFs could be made using DPDK in combination with direct hardware access to the network interface card (NIC). In conclusion, the kernel

Listing 1. cgroups v1 hierarchy [17]

```
/sys/fs/cgroup
├── cpuset
│   ├── cgroup0
│   │   ├── cpuset.cpus
│   │   └── cpuset.mems
│   └── cgroup1
│       └── cpuset.cpus
└── memory
└── cgroup0
└── memory.limit_in_bytes
```

Listing 2. cgroups v2 hierarchy [2]

```
/sys/fs/cgroup
├── cgroup0
│   ├── cpuset.cpus
│   ├── cpuset.mems
│   └── memory.max
└── cgroup1
└── cpuset.cpus
```

networking stack remains the bottleneck. Subsequently, using CNFs with user-space networking and hardware access to the NIC significantly improves the latency. Additionally, using containers instead of VMs enhances the performance. This analysis justifies deploying container in conjunction with a directly accessed NIC and user-space networking to evaluate `cgroup` versions induced latency.

III. `CGROUPS`

`cgroup` is a Linux kernel feature that controls CPU time, memory, and I/O between processes. Resources can be assigned, limited, prioritized, and isolated enabling fine-grained control for administrators, such as enabling only critical processes to access specific resources. For example, a background job should not compete for resources with a web server and potentially negatively affect the latency. With `cgroups`, the administrator can prevent resource contention by guaranteeing resources to the webserver and limiting non-critical processes.

Linux mounts both major `cgroup` versions in the same location, `/sys/fs/cgroup`, but their hierarchical structure differs. Listing 1 shows the hierarchy for v1. Each resource controller (controller) is represented by a separate mount point, and `cgroups` must be created per controller. In contrast, Listing 2 depicts the same hierarchy for v2. A unified, hierarchical structure represented by a single mount point of type `cgroup2` holding all controllers and groups. Each group can then hold any number of enabled controllers.

In v2, it is no longer possible to assign a process to an internal node of the tree hierarchy [18]. These properties can be verified on any modern Linux-based system by inspecting the output of `systemctl status`. The "no inner process" node clears up the hierarchy and makes it easier to understand.

In addition, several inconsistencies have been addressed in `cgroups` v2, leading to a higher degree of standardization. For instance, the renaming of `memory.limit_in_bytes` to `memory.max` is evident in Listing 1 and 2. These standardizations have been applied to all thresholds, resulting in a more uniform and consistent naming scheme.

Scheduler load balancing is a feature where a process may be migrated to a different core to balance the load equally in a multicore system, leading to latency spikes in critical application [19]. In v1 this behavior can be turned off by modifying the file `cpuset.sched_load_balance`. However, in v2, this option was initially removed, added only into recent kernels ($\geq$ 6.1) [20] again, which are not yet part of Debian 10 used in our experiments. We disable the feature for v1 and compare it to v2 with enabled load balancing.

Finally, in v1, each process thread could be assigned to a different cgroup, not possible in v2 due to its confusing nature [18]. Several container solutions use these `cgroups` for their isolation, such as Docker and Linux Container (LXC). Therefore, analyzing performance in combination with `cgroups` is commonly performed for different aspects. Zhuang et al. [21] analyzed the pitfalls of memory isolation in situations with a high memory demand, Xavier et al. [22] analyzed the performance of disk usage when limited using `cgroups` showing good isolation not interfering with each other, and Liu and Guitart [23] analyzed the latency performance of `cgroups` in Ethernet and Infiniband networks resulting in good usability for high-performance low-latency applications.

IV. Measurement Methodology and Setup

The tail-latency behavior for all packets is relevant for analyzing differences in low-latency areas, especially towards using NFV in 5G profiles such as Ultra Reliable Low-Latency-Communications. Moreover, precisely measuring the latency behavior of every packet requires additional effort. We derive our measurement methodology from Gallenmüller et al. [24]. They analyzed the tail-latency behavior for every packet traversing through a VM performing basic packet processing tasks. This was done to analyze the system itself and its surroundings, not the application. We utilize this to support tail-latency analysis of `cgroup` versions.

Figure 2 provides an overview of the three hardware hosts involved in the experiment: the Device under Test (DuT), the Timestamper, and the LoadGen. To generate packets on the LoadGen, we utilize MoonGen [25], a flexible high-performance packet generator based on DPDK capable of producing load at line rate with one CPU core. The Timestamper is connected to the ingress and egress lines via passive optical terminal access points, which add a negligible, constant delay on both sides. The DuT runs a single LXC version 4.0 container with direct access to the interfaces utilizing a minimal DPDK-based, libmoon [25] L2 forwarding application to measure performance differences caused by cgroup versions rather than the application.

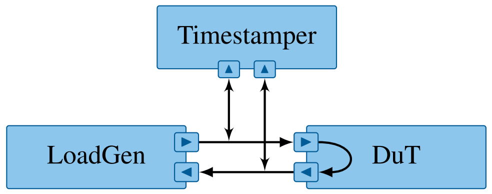

Figure 2. Experiment setup derived from [24]

The hardware specifications are as follows: The LoadGen features an Intel Xeon Silver 4116, 192 GB RAM, and a dual-port Intel 82599ES 10-Gigabit SFP+ NIC connected to the DuT with optical fibers. The DuT has an AMD EPYC 7551P, 128 GB RAM, and a dual-port Intel X710 10Gbe SFP+ NIC. The Timestamper is outfitted with an AMD EPYC 7542, 512 GB RAM, and an Intel E810-XXVDA4 25-Gigabit NIC with SFP flashed to 10 Gbit/s providing 1.25 ns precision.

We use the plain orchestration service (pos) by Gallenmüller et al. [26] to automate and make the measurements reproducible. Using pos enables us to reproduce the results as all experiments perform on a live system using individual scripts.

For evaluating the packets and correlating them between ingress and egress of the DuT, each packet carries a unique identifier to precisely evaluate its network latency. The NIC on the Timestamper adds a hardware timestamp to each packet, which are then matched on ingress and egress to measure latency. This methodology enables us to measure the latency without additional jitter in the measurement process. Subsequently, we evaluate the results for further parameters such as packet rates.

In our measurements, we utilize minimal-sized packets of 64 B as the processing cost of a single packet remains constant, irrespective of its size [25]. Therefore, the number of packets, not their size, is the predominant factor in processing delays. We measure with a packet rate of 1.52 Mpkt/s corresponding to 825 Mbit/s, and 6.24 Mpkt/s corresponding to 3.39 Gbit/s without overloading the network over a time of 45 s. On the DuT, we use Debian 10 with a real-time kernel version 5.10 and utilize the same optimizations described in [24] for VMs.

Using this measurement methodology, we can precisely obtain the latency induced in the network with a precision of 1.25 ns [27]. Analyzing more advanced setups, such as having a CNF application in the container is possible without further adoptions.

V. Evaluation

Each experiment evaluated in this Section is repeated multiple times; the result with the highest tail-latency is used as rare outliers are included. Figure 3 shows the packet rates before and after traversing through the container on the DuT over measurement time. As can be seen and verified by analyzing the raw data, even with the highest measured speed of 6.24 Mpkt/s, no packets are dropped. Consequently, further investigations into throughput are unnecessary, allowing us to shift our focus towards analyzing latency differences.

The latency of both `cgroups` versions with a packet rate of 1.52 Mpkt/s are shown in Figure 4. Both versions exhibit a nearly identical latency trend, with a slight difference emerging between the 99th and 99.9th percentiles. Specifically, the latency with v1 at the 99.9th percentile is slightly higher than with v2. However, towards the 99.99th percentile, the network latencies of both versions are similar again. When using 1500 B sized packets, we can reach a maximum packet rate of only 0.82 Mpkt/s due to the cable capacity of 10 Gbit/s. In Figure 5 the differences are outlined showing that even with higher packet sizes v2 outperforms v1. As fewer packets could be captured, outliers are more significant here.

Figure 6 presents the result of the experiment with a slightly higher packet rate of 6.24 Mpkt/s. The same trend as before is visible, albeit the spike in tail latency occurs slightly earlier. In Figure 7 the corresponding jitter is showing visible improvements from v1 to v2 with higher number of lower jitter values. We expected this behavior as the packet rate was four times higher than in the previous experiment. Likewise, v1 exhibits higher worst-case network latencies.

To further analyze the differences between both versions, we looked into the 5000 worst latencies over time, shown in Figure 8. As the HDR diagram suggested, they are similarly distributed for both versions, with slightly more outliers for v1, indicating that more packets are affected by higher delays. Our remaining measurement data suggests that extreme outliers, like those seen for v1 at the 18th second of measurement time, are more prevalent for v1 and less rare.

We verify the claim that `cgroups` v2 has a more efficient implementation by measuring the number of instructions executed with the Linux performance analysis tool `perf` [28].

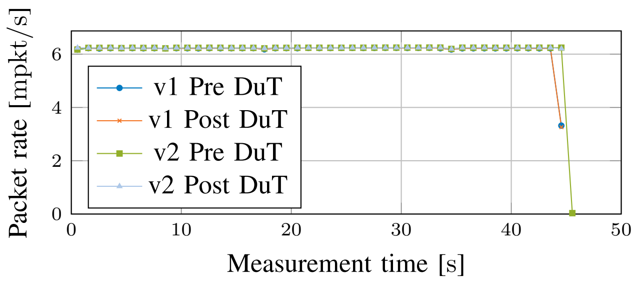

Figure 3. Packet rates before and after traversing through the DuT for `cgroups` versions

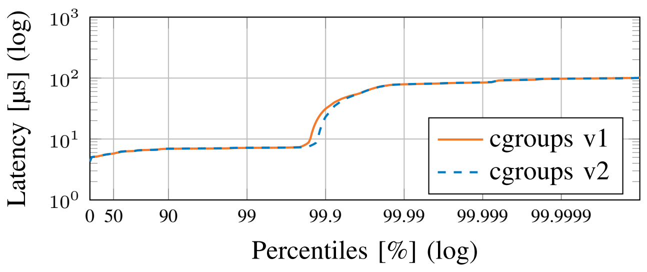

Figure 4. HDR diagram with packet rate of 1.52 Mpkt/s.

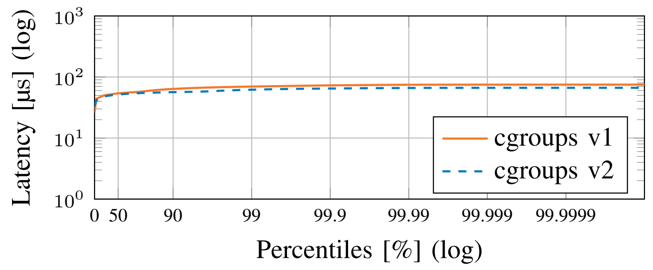

Figure 5. HDR diagram with packet rate of 0.82 Mpkt/s and 1500 B sized packets.

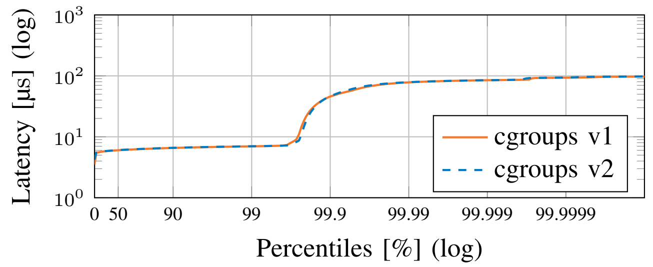

Figure 6. HDR diagram with packet rate of 6.24 Mpkt/s.

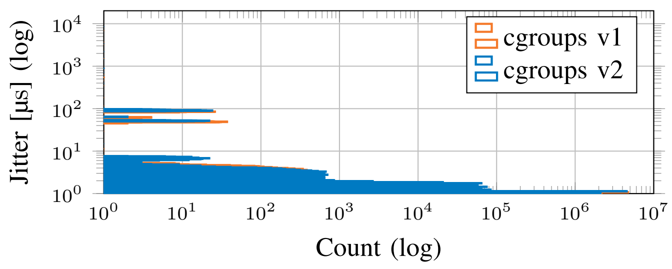

Figure 7. Histogram of jitter values with packet rate of 6.24 Mpkt/s.

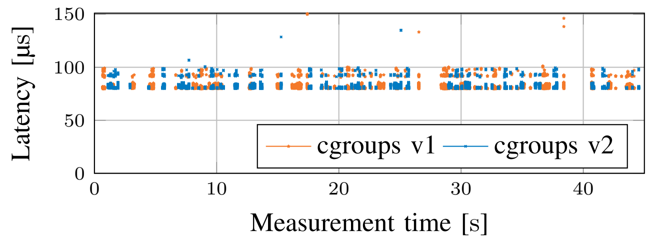

Figure 8. 5000 worst-case latency events over time at 1.52 Mpkt/s

The measurement includes the container startup, 60 s of packet forwarding at 1.52 Mpkt/s, and the teardown. We present the corresponding data in Table I representing the averages of three runs. We observe that for `cgroups` v1, the number of instructions executed is about 2.4 % higher, and about 2.2 % more conditional branches are executed compared. The difference in CPU migrations of 148 in v2 and 297 in v1 is noteworthy. Given that we turned off scheduler load balancing only in v1, this finding is unexpected based on the feature descriptions of `cgroups`. Using additional measurements, it

Table I
INSTRUCTION AND BRANCH USAGE ANALYSIS OF CGROUP VERSIONS.

cgroup version	**instructions**	**branches**	**migrations**
v1	674×10^9	98×10^9	297
std	4×10^9	6×10^8	24
v2	659×10^9	96×10^9	148
std	6×10^8	1×10^8	26
difference	-2.3 %	-2.1 %	-100.7 %

could be verified that the difference in CPU migrations is caused by the startup and teardown of the container itself, not during runtime.

While these differences are minor, we must note that our recorded difference in latencies only occurs at the 99.9^{th} percentile concerning a fraction of all packets.

VI. Limitations

We analyzed the performance of a single container with a basic forwarding application passing through the hardware NIC. Some limitations arise from this, affecting the results. We did not analyze concurrent containers or SFCs.

Moreover, analyzing more advanced packet processing applications using these measurements as a baseline are required to further enhance the development of CNFs. To extract limitations of `cgroups`, measuring without a particular application with additional workload was needed.

VII. Recommendations

Our results show that v2 is more efficient, uses fewer resources, reduces the number of instructions. The latencies when using NVFs on containers are lower for v2. This results in the recommendation to use v2 whenever possible, especially for ultra-low-latency targets. In specific applications, when features are required which are not yet implemented in `cgroups` v2, we suggest to use the older version. Beware that v1 is deprecated and might not receive the same amount of support as v2. The difference is not significant, but showing that the work done for refactoring `cgroups` was successful especially towards number of migrations performed.

VIII. Reproducibility

We provide the raw data of our experiments, the scripts used to generate them, instructions how to use them, and additional results in a website and repository[1].

IX. Conclusion

The Linux kernel constantly evolves, with bug fixing and continuing feature expansion. However, with these rapid changes, there is no extensive research available to evaluate the changes. Stability and tail-latency behavior are essential, especially for time-sensitive applications in the networking area moved to virtualization. This research is furthermore essential, for example, for self-driving cars or airplanes, as safety is a major concern; detailed studies are necessary to ensure their

[1]https://wiednerf.github.io/cgroups-nfv/

reliability and safety. In this paper, we have demonstrated that `cgroups` v2 is a superior choice for low-latency networking in general. While the latency behavior is identical to `cgroups` v1 up to the 99^{th} percentile, v1 performs worse with more packets having higher latency. Additionally, we showed that v2 is less prone to latency spikes and exhibits lower latencies and resource usage. V1 consumes $2.4\,\%$ more instructions for the same workload as the implementation is less efficient. It is worth noting, however, that v2 needs to include some of the features of v1, and upgrading to a modern kernel may only sometimes be possible. Production systems often utilize operating systems with long release cycles, especially in the networking area, as updates are often not feasible.

As part of our future work, we aim to analyze container and their latency behavior in concurrent situations and in trusted execution environments to enhance the safety of CNFs. Moreover, we want to analyze the concurrent usage of containers and VMs as part of SFC to combine their advantages as well as differences between CPUs from different vendors.

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-life, 16KISK002, and project 6G-ANNA, 16KISK107).

References

[1] J. Corbet, "Notes from a container," *LWN.net*, 10 2007, accessed on July 20, 2023. [Online]. Available: https://lwn.net/Articles/256389/

[2] R. Rosen, "Understanding the new control groups API," *LWN.net*, 3 2016, accessed on July 20, 2023. [Online]. Available: https://lwn.net/Articles/679786/

[3] B. Yi, X. Wang, K. Li, S. K. Das, and M. Huang, "A comprehensive survey of Network Function Virtualization," *Comput. Networks*, vol. 133, pp. 212–262, 2018.

[4] F. Monaco, G. Ognibene, F. Parola, and F. Risso, "Enabling Scalable SFCs in Kubernetes with eBPF-based Cross-Connections," in *IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2022, Phoenix, AZ, USA, November 14-16, 2022*, L. J. Horner, K. Tutschku, C. J. B. Cano, R. Bassoli, F. Esposito, R. Tkachuk, and T. Meuser, Eds. IEEE, 2022, pp. 33–38.

[5] D. Raumer, S. Bauer, P. Emmerich, and G. Carle, "Performance implications for intra-node placement of network function chains," in *6th IEEE International Conference on Cloud Networking, CloudNet 2017, Prague, Czech Republic, September 25-27, 2017*. IEEE, 2017, pp. 131–136.

[6] S. M. A. Araújo, F. S. H. de Souza, and G. R. Mateus, "A hybrid optimization-machine learning approach for the VNF placement and chaining problem," *Comput. Networks*, vol. 199, p. 108474, 2021.

[7] M. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration," in *IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2015, San Francisco, CA, USA, November 18-21, 2015*. IEEE, 2015, pp. 74–78.

[8] D. Gedia and L. Perigo, "Performance Evaluation of SDN-VNF in Virtual Machine and Container," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2018, pp. 1–7.

[9] Z. Li, M. Kihl, Q. Lu, and 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.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, M. L. Scott and L. L. Peterson, Eds. ACM, 2003, pp. 164–177.

[11] M. Tran and Y. Kim, "Network Performance Benchmarking for Containerized Infrastructure in NFV environment," in *8th IEEE International Conference on Network Softwarization, NetSoft 2022, Milan, Italy, June 27 - July 1, 2022*, A. Clemm, G. Maier, C. M. Machuca, K. K. Ramakrishnan, F. Risso, P. Chemouil, and N. Limam, Eds. IEEE, 2022, pp. 115–120.

[12] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering, IC2E 2015, Tempe, AZ, USA, March 9-13, 2015*. IEEE Computer Society, 2015, pp. 386–393.

[13] S. Graber. (2023, July) Linux Container - LXC - Introduction. Accessed on July 20, 2023. [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[14] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393.

[15] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[16] A. R. Putri, R. Munadi, and R. M. Negara, "Performance analysis of multi services on container docker, lxc, and lxd," *Bulletin of Electrical Engineering and Informatics*, vol. 9, no. 5, pp. 2008–2011, 2020.

[17] M. Kerrisk, "cgroups(7) - linux manual page," https://www.man7.org/linux/man-pages/man7/cgroups.7.html, 2021, accessed on Jul 20, 2023.

[18] C. Down, "cgroupv2: Linux's new unified control group system," QCON London, 2017.

[19] Debian. (2020, November) cpuset(7) - linux manual page. [Online]. Available: https://manpages.debian.org/bullseye/manpages/cpuset.7.en.html

[20] L. Waiman, "cgroup/cpuset: Add a new isolated cpus.partition type," https://github.com/torvalds/linux, commit f28e22441f353aa2c954a1b1e29144f8841f1e8a, Sep. 2022.

[21] Z. Zhuang, C. Tran, J. Weng, H. Ramachandra, and B. Sridharan, "Taming memory related performance pitfalls in linux cgroups," in *2017 International Conference on Computing, Networking and Communications, ICNC 2017, Silicon Valley, CA, USA, January 26-29, 2017*. IEEE Computer Society, 2017, pp. 531–535.

[22] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds," in *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, M. Daneshtalab, M. Aldinucci, V. Leppänen, J. Lilius, and M. Brorsson, Eds. IEEE Computer Society, 2015, pp. 253–260.

[23] P. Liu and J. Guitart, "Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study," *Clust. Comput.*, vol. 25, no. 2, pp. 847–868, 2022.

[24] 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.*, vol. 31, no. 1, p. 20, 2023.

[25] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, K. Cho, K. Fukuda, V. S. Pai, and N. Spring, Eds. ACM, 2015, pp. 275–287.

[26] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, "The pos framework: a methodology and toolchain for reproducible network experiments," in *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*, G. Carle and J. Ott, Eds. ACM, 2021, pp. 259–266.

[27] Intel Corporation, "E810 datasheet rev2.5." [Online]. Available: https://cdrdv2-public.intel.com/613875/613875_E810_Datasheet_Rev2.5.pdf

[28] "perf-stat(1) - linux manual page," https://www.man7.org/linux/man-pages/man1/perf-stat.1.html, accessed on July 20, 2023.