LXC Container Between `cgroups` v1 and v2: a Performance Evaluation

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

***Abstract*—The cgroups feature of the Linux kernel is widely used by lightweight virtualization technologies such as Docker or LXC to provide resource isolation. Recently, cgroups underwent a significant revamp from version 1 (v1) to version 2 (v2). Researching the performance difference between these two versions in terms of network latencies enables the usage of containers in time-critical applications. Previous work ignored cgroups as a potential source of latency in packet-processing systems. In this paper, we measure the performance difference between cgroups v1 and v2 in isolation using commodity hard- and software. Our experiments show that the two versions achieve the same degree of isolation, but the tail latencies of v1 are higher, which can be explained by a more efficient, 2.4 % less instruction-consuming implementation of v2. Therefore, we recommend the use of v2 for low-latency lightweight virtualization network deployments wherever possible.**

***Index Terms*—low latency, container, lxc, virtualization, dpdk, cgroups, packet processing**

1. 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 the slightest delay can result in catastrophic consequences. That is why network latencies are a crucial factor in enabling the interaction and coordination of sensitive applications. To achieve the lowest and most stable network latencies, it is crucial to invest in high-quality networking equipment and thoroughly review the entire software stack.

A common way of handling increasing complexity is to compartmentalize different software components into smaller pieces and run them in isolated environments. This can be achieved through virtualization, using either heavyweight virtual machines or lightweight containers. A deeper understanding of the inner workings of the chosen virtualization technique is necessary to optimize for low-latency networking. Two essential features for enabling 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 were completely revamped [2] with a second version (v2) released in kernel 4.5. Since cgroups v1 and v2 differ in their implemented features, this paper highlights the differences between them. Furthermore, we show the impact of the cgroup version on network latencies with experimental measurements.

The paper is structured as follows: Section 2 presents related work. In Section 3, we provide background information on cgroups and Linux containers (LXC). Section 4 details the specific optimizations we apply to increase isolation. Section 5 discusses the experimental setup and measurements. Finally, we summarize our findings in Section 6 and propose future work.

2. Related Work

In our previous work [3], we expanded the capabilities of HVNet by Wiedner et al. [4], a framework for orchestrating low-latency experiments on a single host with KVM, by incorporating LXC containers, initially developed by Wiedner et al. HVNet automates the setup of all involved hosts, configures them for low-latency networking, and synchronizes the measurement scripts. Virtual networking topologies can be defined in text files. Our measurements [3] of network latencies for LXC containers with cgroups v2 demonstrated comparable performance to VMs but with occasional spikes in tail latencies. To address this issue, we found that using a real-time kernel is effective. Our prior implementation and research serve as the foundation for this paper, where we introduce a new feature in HVNet to switch between cgroups v1 and v2 and compare the cgroup versions to evaluate their performance.

Abeni et al. [5] propose a real-time scheduler for the Linux kernel that is aware of cgroups, making it compatible with Docker and LXC. They demonstrate experimentally that their scheduler delivers lower average response times for a task set than KVM but with similar worst-case latencies. Notably, their scheduler is capable of migrating processes to another core that has processing time left; a feature unavailable in KVM-based systems since the hypervisor has no access to the scheduling of a VM guest. However, it provides lower network latencies for cgroup-based virtualization. Although their work shows promise, it may not be directly applicable to our setup. In our setup, a single core processes all packets of a network interface card (NIC) in userspace with the Data Plane Development Kit (DPDK). Process migration between different cores is not anticipated.

 doi: 10.2313/NET-2023-11-1_03

Listing 1: cgroups v1 hierarchy [6]

```
/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
```

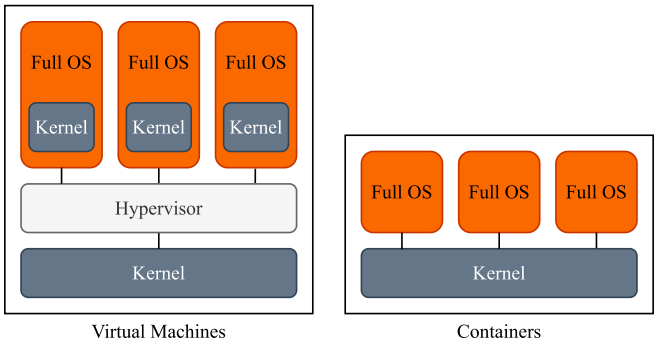

Figure 1: Architecture of VMs and containers [3]

3. Background

Section 3.1 presents a short introduction to cgroups, highlighting their key features and differences between v1 and v2. Section 3.2 introduces the concept of containers and Linux Containers (LXC), the container implementation we rely upon.

3.1. cgroups

cgroup is a Linux kernel feature that assigns resources such as CPU time, memory, and I/O between processes. Resources can be limited, prioritized, and isolated, enabling administrators fine-grained control over the system. There are scenarios where it is desirable to guarantee that one critical process has access to resources. For example, a background cronjob should not compete for resources with a web server and potentially negatively affect the latency of a request. With cgroups, the administrator can prevent resource contention by guaranteeing resources to the webserver and limiting non-critical processes.

Both cgroup versions are mounted in the same location, `/sys/fs/cgroup`, but differ in their hierarchical structure. Listing 1 shows the hierarchy for v1, where each controller is represented by a separate mount point, and a cgroup must be created for each controller individually. In contrast, Listing 2 depicts the same hierarchy for v2. A unified, hierarchical structure represented by a single mount point of type `cgroup2` holds all controllers and groups. Each group can 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 as claimed by Down [7]. 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.

Important for our network latency performance analysis is the scheduler load-balancing option. 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 [8]. In latency-critical applications, a single context switch can cause a spike in latency. In v1, this behavior can be disabled by modifying the file `cpuset.sched_load_balance`. However, in v2, this option was initially removed. Only recent kernels ($\geq$ 6.1) support this option which was introduced by Waiman [9]. This paper disables scheduler load balancing for v1 and compares it to cgroups v2 with load balancing. Testing this option for cgroups v2 is out of scope for this paper due to missing infrastructure for testing the latest kernels.

Finally, another change is that in v1, each thread of a process could be assigned to a different cgroup [7]. This behavior is considered confusing and unnecessary and is no longer present in v2.

3.2. LXC Containers

Containers are a lightweight alternative for virtualization. They are considered to be operating system level because they share the host kernel and operate on the same level as any other process in userspace. Figure 1 highlights this architectural difference between VMs and containers. A container does not virtualize its own kernel, while a VM does. Furthermore, no hypervisor is required. Sharing the kernel with the host and other containers has implications for isolation. However, modern kernels offer features that help to build isolated systems. The most important features are cgroups and namespaces.

LXC is a low-level container runtime being in active development since 2008. It provides a minimalistic feature set to remain lightweight with minimal overhead. Userspace tools for managing LXC containers are available. A C or Python API is available for controlling LXC for more advanced use cases. One downside of LXC is that convenience features such as layered images or orchestration are missing entirely. LXC images are typically larger than Docker images since they snapshot the entire root

filesystem of an OS installation - including the installed libraries.

We use LXC 4.0, the userspace tools, and the Python API for our implementation. To evaluate the performance difference of cgroups v1 and v2, we extend an existing framework for low-latency measurements: HVNet [4].

4. Implementation

Our original plan was to implement cgroups v1 and v2 on Debian Buster to enable a more seamless comparison with previous work by Wiedner et al. [4] and Gallenmüller et al. [10], [11]. However, Debian Buster runs on kernel 4.19, which does not yet include the cpuset controller [12]. Without the cpuset controller, the container is unisolatable from the rest of the system, making a performance comparison between cgroups v1 and v2 meaningless. Therefore, we focus our efforts on Debian Bullseye, which supports the legacy cgroups v1 and a more mature implementation of cgroups v2.

To switch between the two cgroups versions, a startup flag `--lxc-enable-cgroup-v1` is implemented in HVNet [4]. By setting this flag, the container host is booted with the kernel parameter `systemd.unified_cgroup_hierarchy=0`. This parameter disables the unified cgroups v2 hierarchy and enables the legacy cgroups v1.

The process isolation methodology differs between cgroups v1 and v2, but both achieve the same goal. For v2, we utilized two methods: first, the feature `cpuset.cpus.partition`, which removes CPU cores of a child from the parent cgroup. Second, we use systemd to restrict all processes to CPU cores unused by the container through the command `systemctl set-property user.slice AllowedCPUs=0-23;27-31`. In contrast, neither of these features are available with v1. Instead, we followed the instructions suggested by Weisbecker [13]. First, we create a new housekeeping cgroup and restrict its access to CPU cores, ensuring no cores are shared between housekeeping and the container. Next, we move all processes, including kernel threads unrelated to the container into this cgroup using the Python tool `cset` [14]. It is worth noting that the init process with PID must remain in the root cgroup; otherwise, it would be impossible to start a container. By applying these optimization techniques, we have ensured that the processing cores are fully isolated.

5. Evaluation

In the first Section 5.1, we introduce the experiment setup in detail. Subsequently, in Section 5.2 we present the findings from our measurements and provide a comprehensive analysis and discussion of the results.

5.1. Experiment Setup

Our experimental setup follows both HVNet [4] and our previous work [3]. Figure 2 provides an overview of the configuration for the three hosts involved in the experiment: the Device under Test (DuT), the Timestamper, and the load generator (LoadGen). To generate packets on the LoadGen, we utilize MoonGen by Emmerich et al. [15], a flexible high-performance packet generator written in Lua. The Timestamper is connected to the ingress and egress lines via passive optical terminal access points (taps), which add a negligible, constant delay. The DuT runs a single LXC container with direct access to the ingress and egress interfaces and runs a minimal DPDK L2 forwarding application.

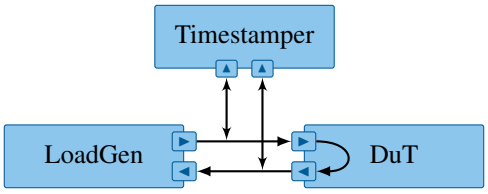

Figure 2: Experiment setup [3]

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 is equipped with 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 a dual-port Intel E810-XXVDA4 25-Gigabit flashed to 10-Gigabit NIC, providing 1.25 ns precision.

To automate and make the measurements reproducible, we use the plain orchestration service (pos) by Gallenmüller et al. [16]. This service enables us to control boot parameters, power status, and images of bare-metal hosts and VMs hosted by libvirt, utilizing IPMI. In our previous work [3], we developed virtualLXCBMC [17] to integrate LXC with pos, which enables us to control LXC containers with IPMI. However, since a container does not have its own kernel, it is impossible to set boot parameters.

Each packet carries a unique identifier to precisely evaluate its network latency. The packets are timestamped by their respective NICs. The Timestamper matches a packet on the ingress and egress and measures the duration. This methodology enables us to measure the processing latency without introducing latency by the measurement process itself. Subsequently, the Timestamper generates pcap files which scripts process further.

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 [15]. Therefore, the number of packets, not their size, is the predominant factor contributing to processing delays. We measure with a packet rate of 1.52 Mpkt/s corresponding to 825 Mbit s^{-1}, and 6.24 Mpkt/s corresponding to 3.39 Gbit s^{-1}, like in our previous work [3]. On the DuT we use Debian Bullseye with a real-time kernel 5.10.

5.2. Results

To measure the tail latencies of cgroups v1 and v2, we use the setup of Section 5.1, the optimizations described in Section 4 and [4]. Each experiment is repeated three times; the worst case is reported in this paper. We have made instructions for reproduction and additional measurement data available for inspection[1].

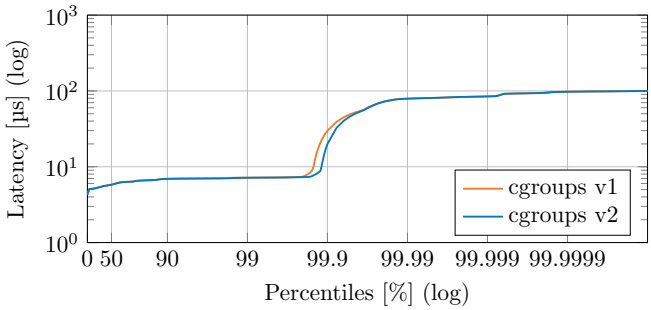

Figure 3: HDR histogram of cgroups v1 and v2 with 1.52 Mpkt/s on Debian Bullseye with real-time kernel.

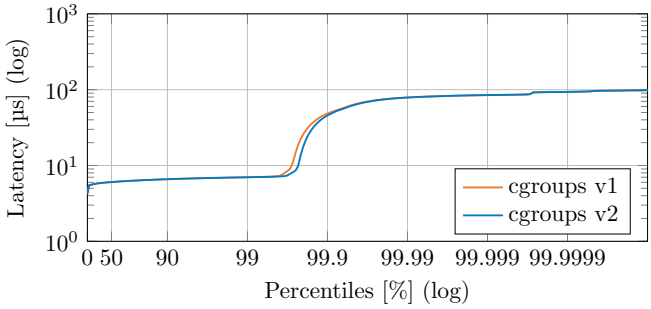

Figure 4: HDR histogram of cgroups v1 and v2 with 6.24 Mpkt/s on Debian Bullseye with real-time kernel.

The latency differences between cgroups v1 and v2 on Debian Bullseye with a real-time kernel, with a packet rate of 1.52 Mpkt/s, are shown in Figure 3. Both versions exhibit a nearly identical latency trend, with a slight difference emerging between the 99^{th} and 99.9^{th} percentiles. Specifically, the latency with v1 at the 99.9^{th} percentile is slightly higher than with v2. However, towards the 99.99^{th} percentile, the network latencies of both versions match again.

Figure 4 presents the result of the same experiment with a higher packet rate of 6.24 Mpkt/s. The same trend as in Figure 3 is visible, albeit the spike in tail latency occurs slightly earlier. This behavior is expected, as the packet rate is four times higher than before. Likewise, v1 exhibits higher worst-case network latencies.

The 5000 worst-case latencies, as shown in Figure 5, are similarly distributed for cgroups v1 and v2, as the HDR histograms already suggested. There are slightly more outliers for v1, indicating that more packets are affected by higher processing delays than for v2. Our remaining measurement data suggests that extreme outliers, like those seen for v2 at the 34^{th} second of measurement time, are more prevalent for v2.

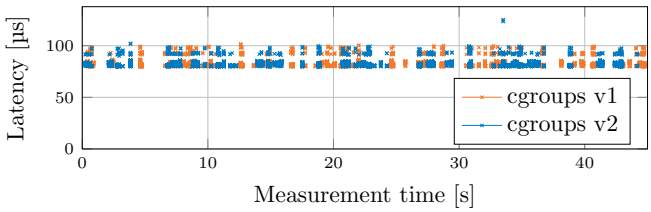

Figure 5: 5000 worst-case latencies of cgroups v1 and v2 with 1.52 Mpkt/s on Debian Bullseye with real-time kernel

TABLE 1: Performance analysis with `perf stat -B` between cgroups v1 and v2.

cgroups	instructions	branches	migrations
v1	674×10^9	98×10^9	297
v2	659×10^9	96×10^9	148

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` [18]. The measurement includes the container startup, 60 s packet forwarding at 1.52 Mpkt/s, and the shutdown. The experiment is repeated three times, taking the average value. The resulting data is presented in Table 1 and in the reproduction collection[1]. 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 to v2. The difference in process migrations of 148 in v2 and 297 in v1 is noteworthy. Given that we disabled scheduler load balancing, this finding is unexpected.

While these differences seem small, it is crucial to note that our recorded difference in latencies only occurs at the 99.9^{th} percentile, which concerns only a fraction of all packets.

6. Conclusion and Future Work

The Linux kernel is a constantly evolving system, with bug fixing and continuing feature expansion. However, with these rapid changes, there is no extensive research available to evaluate the changes. To ensure the reliability and safety of using containers and cgroups in low-latency systems like self-driving cars or airplanes, detailed studies are necessary. In this paper, we have demonstrated that cgroups v2 is a superior choice for low-latency networking. 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, v1 consumes 2.4 % more instructions for the same workload, indicating its less efficient implementation. Therefore, we conclude that cgroups v2 is the better choice for low-latency systems that require high performance and reliability. It is worth noting, however, that v2 lacks some of the features of v1, and upgrading to a more modern kernel may not always be possible without additional costs. Systems in production often utilize operating systems with long release cycles, which makes changes to packaged software like the kernel expensive.

As part of our future work, we aim to enhance and automate measuring and analyzing the instructions consumed by a container, the underlying technology, and the applications inside. Automation would enable us to assess the performance of different enabling technologies like cgroups more effectively. Additionally, we are interested in testing the kernel 6.1, which incorporates the new cpuset partition type `isolated`, and comparing it against cgroups v1 `cpuset.sched_load_balance`.

1. https://wiedner.pages.gitlab.lrz.de/iitm-seminar-daichendt-reproducibility/

 doi: 10.2313/NET-2023-11-1_03

References

[1] J. Corbet, "Notes from a container," *LWN.net*, 10 2007, accessed on March 23, 2023. [Online]. Available: https://lwn.net/Articles/256389/

[2] R. Rosen, "Understanding the new control groups API," *LWN.net*, 3 2016, accessed on March 23, 2023. [Online]. Available: https://lwn.net/Articles/679786/

[3] A. Daichendt, "Lightweight low-latency virtual networking," August 2022, B.Sc. thesis, found at https://gitlab.lrz.de/wiedner/iitm-seminar-daichendt-reproducibility/-/blob/master/data/thesis.pdf.

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

[5] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3373400.3373405

[6] M. Kerrisk, "cgroups(7) - linux manual page," https://www.man7.org/linux/man-pages/man7/cgroups.7.html, 2021, accessed on April 1, 2023.

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

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

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

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

[11] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, "5G URLLC: A case study on low-latency intrusion prevention," *IEEE Communications Magazine*, vol. 58, no. 10, pp. 35–41, Oct. 2020.

[12] H. Tejun, "Control Group v2," https://www.kernel.org/doc/html/v4.19/admin-guide/cgroup-v2.html, Oct. 2015, accessed on March 23, 2023.

[13] F. Weisbecker, "CPU isolation practical example, part 5," January 2022, accessed on March 23, 2023. [Online]. Available: https://www.suse.com/c/cpu-isolation-practical-example-part-5/

[14] A. Tsariounov, "cpuset," https://github.com/lpechacek/cpuset, 2018, accessed on 2023-03-23.

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

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

[17] A. Daichendt, "VirtualLXCBMC," August 2022. [Online]. Available: https://github.com/AnonymContainer/virtuallxcbmc

[18] "perf-stat(1) - linux manual page," https://www.man7.org/linux/man-pages/man1/perf-stat.1.html, accessed on March 23, 2023.