computers

Article

Toward Optimal Virtualization: An Updated Comparative Analysis of Docker and LXD Container Technologies

Daniel Silva, João Rafael and Alexandre Fonte *

Instituto Politécnico de Castelo Branco, Av. Pedro Álvares Cabral, nº 12, 6000-084 Castelo Branco, Portugal; dsilva@ipcbcampus.pt (D.S.); j.rafael@ipcbcampus.pt (J.R.)
* Correspondence: adf@ipcb.pt

Abstract: Traditional hypervisor-assisted virtualization is a leading virtualization technology in data centers, providing cost savings (CapEx and OpEx), high availability, and disaster recovery. However, its inherent overhead may hinder performance and seems not scale or be flexible enough for certain applications, such as microservices, where deploying an application using a virtual machine is a longer and resource-intensive process. Container-based virtualization has received attention, especially with Docker, as an alternative, which also facilitates continuous integration/continuous deployment (CI/CD). Meanwhile, LXD has reactivated the interest in Linux LXC containers, which provides unique operations, including live migration and full OS emulation. A careful analysis of both options is crucial for organizations to decide which best suits their needs. This study revisits key concepts about containers, exposes the advantages and limitations of each container technology, and provides an up-to-date performance comparison between both types of containers (applicational vs. system). Using extensive benchmarks and well-known workload metrics such as CPU scores, disk speed, and network throughput, we assess their performance and quantify their virtualization overhead. Our results show a clear overall trend toward meritorious performance and the maturity of both technologies (Docker and LXD), with low overhead and scalable performance. Notably, LXD shows greater stability with consistent performance variability.

Keywords: containers; Docker; LXC; LXD

check for updates

Citation: Silva, D.; Rafael, J.; Fonte, A. Toward Optimal Virtualization: An Updated Comparative Analysis of Docker and LXD Container Technologies. *Computers* **2024**, *13*, 94. https://doi.org/10.3390/computers13040094

Academic Editors: Paolo Bellavista and Wenbing Zhao

Received: 24 January 2024
Revised: 4 April 2024
Accepted: 7 April 2024
Published: 9 April 2024

1. Introduction

The increasing demand for highly available computing resources and the increasing adoption of Cloud computing have driven the adoption of virtualization technologies. The benefits offered by virtualization in controlling total acquisition costs (also known as capital expenditures or CapEx), and infrastructure maintenance costs (also known as operating expenses or OpEx) have contributed to this interest and to the fact that it has become one of the core blocks necessary to build large data centers and cloud infrastructure [1,2].

Traditional virtualization, based on modern hypervisors, is recognized as a mature and widely adopted technology in support of these infrastructures. However, the addition of an abstraction software layer for virtualization, allocation, and scheduling of host system resources (CPU, memory, or storage) to multiple virtual machines (VMs) incurs an overhead that has clear performance implications. Despite the latest hardware advances aimed at improving the performance of VMs, in certain contexts, VMs fail to provide the necessary scalability, agility, and flexibility. This is especially clear when running microservice applications or other modern applications that demand the employing of continuous integration/continuous deployment (CI/CD) best practices for agile development, where teams need to automate and accelerate the process of application development, testing, and deploying to eliminate the traditional approach of large and infrequent releases [3].

In response to these challenges and overhead, container-based virtualization presents itself as a compelling alternative to VMs and is becoming increasingly popular [4].

Unlike VMs, which virtualize resources at the hardware level, containers operate at the operating system (OS) level, sharing the same OS kernel with reduced overhead and eliminating the need for hypervisors. Containers are lightweight, requiring only the necessary dependencies for the application and a minimal execution environment, thus offering higher performance compared to VMs.

The most successful containers are Docker, CoreOS Rtk (also known as Rocket), LXC/LXD, Podman, and OpenVz, while orchestration tools such as Kubernetes, Apache Mesos, and Docker Swarm, allow for the deployment of container-based applications, managing thousands of containers, and balancing loads across clusters.

Containers can be categorized into two main types, application containers and system containers, each serving a different purpose. Application containers encapsulate a single process of an application, while system containers behave as a complete OS environment.

The goal of this article is to investigate the performance of recent container-based virtualization solutions, selecting as a target two of the most representative instances of each type, Docker (application) and LXD/LXC (system), contrasting and quantifying the overhead in relation to a native environment while trying to find a balance between performance and the purpose of its adoption. For this, a variety of benchmarks and workloads stressing these environments will be performed. Furthermore, this article aims to demonstrate the high maturity state of both container virtualization technologies.

The main work contributions are as follows:

- Reviewing key concepts about the container-based form of virtualization;
- Analyzing the advantages and limitations of application and system containers;
- Formalizing and quantifying any overhead introduced;
- Conducting an up-to-date comparison of container technologies using modern instances and recent hardware;
- Contrasting performance between container types (application vs. system);
- Investigating the variability in hardware–performance combinations;
- Demonstrating the maturity of container-based virtualization.

The rest of this paper is organized into six sections. Section 2 provides background information, discusses the concepts of container virtualization, and analyzes the two candidate container instances selected for evaluation. In Section 3, the scope of the study and related work are presented. Section 4 presents formal considerations regarding the expected overhead. Sections 5 and 6 detail the implementation of the adopted testing methodology and discuss the experimental results of the performance evaluation. Finally, Section 7 discusses major findings, presents conclusions, and suggests avenues for future work.

2. Background

This section aims to provide an overview of container-based virtualization technology, contrasting it with hypervisor-based virtualization, and highlights the key differences between Docker and LXD/LXC containers, which are the focus of this study.

2.1. Hypervisor Types and Containers

In traditional virtualization, a hypervisor, also known as a virtual machine monitor (VMM), is implemented as a software layer situated between the host system hardware and the virtual machine (VM) [5], as illustrated in Figure 1. The role of the hypervisor is to manage the allocation and scheduling of host hardware resources (CPU, memory, and input–output (I/O) devices) among multiple VMs. A hypervisor can provide VMs with high performance and a highly isolated and secure environment, without the behavior of one VM affecting the others. Examples of such hypervisors [6], which are usually HW-assisted, are kernel-based virtual machines (KVMs) and include Microsoft Hyper-V and VMware ESXi. Additionally, there are Type 2 or hosted hypervisors, which run as a host application/extension on top of the host OS, adding more overhead, and are the VMware Workstation/Fusion and the Oracle Virtual Box [7].

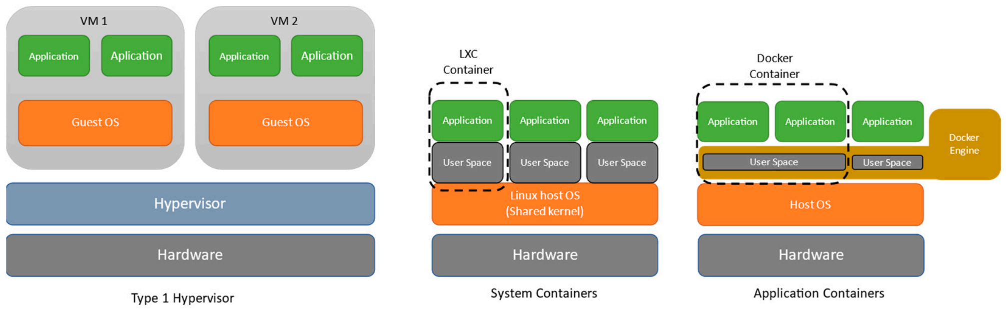


Figure 1. Hypervisors vs. containers.

Container-based virtualization offers a lightweight alternative to hypervisor-based virtualization. Unlike a VM that runs a full operating system, a container can encapsulate an individual process or application or behave like a full OS. As illustrated in Figure 1, this virtualization solution precludes the use of a hypervisor to function as an intermediary to take control and share resources. In container-based virtualization, the need for an intermediary hypervisor is eliminated. Instead, virtualization occurs at the kernel level of the host operating system (OS). Multiple isolated and independent slices of the host OS are created for containerized guests, allowing applications to share the same OS and, in some cases, common libraries. This approach reduces overhead and streamlines booting and shutdown times, as there is no need to boot a complete guest OS [8].

There are essentially the following three types of containers: process or application containers, system containers, and embedded containers. Embedded system containers serve to alleviate complexity in the IoT (internet of things) or industrial systems but are outside the scope of the present work. All these approaches use the host OS kernel. Figure 1 also shows the comparison between application containers (e.g., Docker or Rocket) and system containers (e.g., LXD/LXC or OpenVz), which are used in different circumstances, as described below.

Application containers, like Docker or Rocket, are process containers that bundle and run one process or service per container. These containers include all the necessary dependencies, configuration files, and libraries required for their operation, thus ensuring consistent operation in multiple environments.

Originally, containerization in Linux relied on tools like chroot to isolate processes; however, modern containerization leverages Linux kernel namespaces (an advanced form of chroot that changes the root directory of running processes) to isolate resources, both network and process identifiers (PID), and on cgroups to limit the use of resources and distribute them to containers.

On Windows, containerization principles differ, including process containers (isolated view of system resources within the same kernel), Hyper-V containers (running within a separate, stateless Hyper-V guest VM), and HostProcess containers (running directly on the host's network namespace) [9,10].

To conclude, Table 1 summarily contrasts both forms of virtualization, with benefits on both sides.

While containers offer lower complexity and overhead due to closer access to OS services, they may exhibit lower levels of isolation and security compared to hypervisor-based virtualization. Hypervisor-based solutions typically include advanced security features like data encryption, network isolation, and granular access control. However, potential OS-level vulnerabilities in containers necessitate additional security measures to mitigate risks [11].

However, containers can offer a high-density solution and allow for a fast and consistent deployment of applications, given the greater portability of applications, since all the necessary dependencies are already grouped in the container. Both solutions tend to provide an organization's infrastructures, including Cloud with lower CapEx and OpEx costs, and can meet the following four major design objectives of these infrastructures: scalability, reliability, dependability, and security compliance [2].

Table 1. Hypervisors vs. containers comparison.

Feature	Virtual Machines	Containers
Isolation	Increased isolation between virtual machines	Less effective. Shares the kernel of the host operating system
Size	Larger, include a complete operating system and dependencies	Smaller; the image includes only application-specific dependencies
Booting and Shutdown	Slower due to OS size and startup	Faster due to the use of the shared kernel
Management	Managed as independent entities	Managed on a large scale in clusters by using orchestrators.
Scalability	Vertical scaling; requires dedicated resources for each virtual machine	Horizontal scaling; can be managed on a large scale
Portability	Less portable; depends on the hypervisor compatible with the operating system	They can be easily moved between development and production systems and environments.
Security	Greater isolation between machines, hypervisors also have advanced features such as encryption, network isolation, and granular access control.	Potential risk of a compromised container affecting other containers, due to OS vulnerabilities
OS flexibility	Increased OS flexibility; depends only on hypervisor and host architecture	Flexibility limited by OS; a container needs to use the same OS

2.2. Docker and LXD/LXC

Docker is an open-source container platform that provides a comprehensive set of tools to create and manage the lifecycle of containers, including monitoring their state [12].

One of Docker's key strengths is its ability to streamline the deployment, testing, and scaling of containerized applications, making it well-suited for environments adopting the microservice design architecture pattern, as it facilitates the decomposition of applications into individual, isolated services that can scale horizontally, providing flexibility in application development and deployment.

Docker provides a deployment model based on a layered file system (Unionfs), which enables the creation of sharable images or libraries through public registry servers, such as Docker Hub. Docker images facilitate the portability of applications or sets of services between various environments, since the resulting container instance encapsulates the application or microservice along with all required dependencies for execution on any server (e.g., Linux or Windows). For example, in an application that requires the Apache TomCat application server and a MySQL database, the developer only needs to create a Dockerfile consisting of a sequence of commands to assemble a base image with these two items. This image can then be distributed to other computers.

In technical terms, Docker is lightweight, and it was initially built on Linux containers (LXC) on Linux systems, but it has gone its own direction, creating its own runtime environment called libcontainer, which is integrated within the Docker engine. This environment interfaces with kernel features such as namespaces and cgroups to provide efficient containerization. Docker uses the QEMU emulator when needed and, in Windows, leverages new Hyper-V/WSL2 technologies. In contrast to LXC, which launches an init OS in each container and can then launch other processes, Docker provides an OS environment by the Docker engine for running applications, specified in the Docker image.

At the core of the Docker architecture are several high-level components that enable the creation, management, and execution of containers. These are the Docker CLI, Docker daemon, and its REST API, which have collectively contributed to Docker's widespread adoption.

The Docker CLI serves as the primary interface for users to interact with Docker and send commands. These are translated into requests to the Docker daemon via the REST API, which then executes the necessary management operations (e.g., pull or run) on Docker objects, such as images and containers.

The two components that support these operations and the execution of containers at the lowest level are containerd and runC. The containerd provides a high-level execution environment and manages the lifecycle of all containers on the host system and abstracts the calls to the different OS kernels (Linux, Windows, or Solar). The runC component is the lowest-level component used to run the container; for example, it uses native Linux features to create and run containers, and runC includes libcontainer.

Finally, these low-level components, containerd and runC, follow the specifications of the Open Container Initiative that aims to promote and ensure interoperability between container platforms [13,14]. In this way, it is possible that the Kubernetes orchestration tool can use the Docker containerd high-level execution environment, or CRI-O environment, created for Kubernetes [15,16].

Linux LXC is an operating system-level virtualization technology that allows for the creation and execution of multiple virtual environments (VEs) or Linux containers, which are fully isolated on a single host [17]. These containers can be used in three ways, either as a sandbox for running specific applications or microservices within a VE, as application containers, or to virtualize a complete operating system as system containers.

LXC uses the namespaces mechanism to isolate the containers and the native Linux functionality, cgroups, to share its kernel with the containers and limit the CPU, memory, disk, and network between containers. This approach removes the overhead imposed by an additional kernel and consumes fewer resources, thus having higher efficiency and better performance [18].

In the case of system containers, LXC distinguishes itself from Docker by allowing for a single init process to create an isolated virtual environment that supports multiple application processes in a single container.

LXC can be combined with container management tools, such as Docker and LXD, thus allowing the creation of multiple complex virtual environments with ease, increasing their accessibility.

LXD is an extension of LXC and aims to turn it into a modern management tool for containers and virtual machines, similar to Docker, by utilizing a REST API to communicate with the LXC software library (libxlc), enhancing the LXC features, such as improved security and isolation (e.g., rogue container detection), live migration of containers, and limiting resources used by each container [19]. LXD is also image-based, offering a wide range of images of various Linux distributions, enabling flexibility and scalability, supporting various backends and network types, and HW from laptops to rackmount servers in the Cloud.

LXD platform enables an easy creation and management of containers, offering an additional layer of container management that enables more refined control of each container and its settings, including limiting resources, creating snapshots and migrating containers between hosts, using a command line tool, using its REST API, or using third-party or integrated tools [20,21].

3. Scope and Related Work

The primary focus of this research is to study container-based virtualization and to assess its maturity through performance evaluation of modern containers, including both process and system containers, which play a critical role in OS-level virtualization.

Traditional virtualization techniques face performance challenges due to the added overhead compared to native environments. Initially, system VMs had significant overhead, but this has been reduced over the years due to software optimizations and hardware advances. By recognizing this, extensive performance evaluations of hypervisors and non-virtualized execution have been conducted and found, in some scenarios, significant overhead [22–26].

More recently, other forms of virtualization, such as container-based virtualization have been the focus of attention, and extensive performance evaluations have been conducted, especially considering Docker due to the limited adoption of LXC in its early years. Nevertheless, previous comparisons of containers mostly relied on older versions and out-of-tree patches of containers [27–30].

This paper specifically provides an updated performance assessment of containers, considering two modern application and system containers platforms (Docker vs. LXD/LXC) for evaluation. Notably, LXD is included due to its recent advances and efforts in promoting adoption [19]. Thanks to LXD, LXC has been revitalized as a part of the system container ecosystem. This evaluation will primarily assess the added overhead on a resource-by-resource basis. To the best of our knowledge, this is also the first work to lay the groundwork for formally describing the added overhead by container-based virtualization, breaking it down into its main components and incorporating considerations related to certain evaluation limitations.

We restrict our study to base container technologies. For instance, a recent assessment of networking solutions for containers was conducted in [31]. Despite the performance findings showing the superior performance of SR-IOV, the study focused on Kubernetes and, apparently, their results and may not be replicated for both types of containers (applicational and system).

Finally, assessing emerging solutions that are extending containerization into specialized domains such as high-performance computing (HTC) environments [32,33] or addressing security challenges raised by containers [34] are also out of the scope of this work.

4. Expected Overhead of Containers

As discussed in Section 2, containers are an abstraction placed directly on top of the host OS for different processes or device namespaces, and, in the case of system containers, it is possible to virtualize multiple, completely new hosts within a single operating system kernel. For this reason, although there is a significant cost inherent to the configuration of containers, the expected overhead is reduced.

To validate this expectation, this study aims to analyze the performance of these two containerization solutions, using two of the most representative instances of application and system containers, Docker vs. LXD, contrasting them and using the native machine as a reference.

The different benchmarks and the different workloads they provide can be used during the tests to stress and evaluate performance aspects of the containers. Thus, for each workload, W_i, of a benchmark, a specific performance metric, P_i, will be measured on a performance feature basis (computing, I/O, and network communications). When analyzing each P_i value, it should be considered that it may be generically affected by several overhead components sourced from different virtualization mechanisms or abstractions. Though identifying all overhead components in a very fine-grained manner and providing a full formal analysis of overhead components is beyond this study's scope, we will next delve a bit deeper.

To start, as discussed in Section 2, Docker and LXD containers are very lightweight when compared to virtual machines (VMs), given that containers differ from VMs in the way they use host resources. Instead of virtualizing system hardware, Docker, and LXD share the kernel of their host, even though we have already seen that this is not exactly the case. Docker and LXD/LXC containers use kernel features (e.g., kernel namespaces, chroot, and cgroups) to create isolated processes and file systems, providing a completely isolated virtual environment (VE) or container.

Initially, Docker used LXC to create isolation from the host system but later switched to its own libraries (libcontainer). Despite this, Docker and LXD/LXC share many aspects. Docker offers an abstraction for machine-specific settings, by abstracting the storage, networking, and other features. These are part of the Docker engine and make Docker containers more portable, as they rely less on the underlying physical machine. Docker supports layered containers by default due to its layered file system; this means that the resulting container is the sequential combination of changes made to the file system.

In contrast, LXD complements LXC by handling networking and data storage, it should be expected that LXD would add negligible overhead given that LXC uses kernel extensions, operating similarly to a VM. With the absence of a virtualization hypervisor, the containerized machine still has limited access to the hardware (compared with a native OS), but the resources are accessed in a much easier fashion compared to a VM. The main issue with LXD is its lower scalability due to the size of container images when compared with Docker.

Even with all mentioned extra layers of abstraction, significant performance issues are not expected in most cases when using Docker or LXD.

Consequently, with a focus on Docker, the overall performance of containers should be influenced by some factors, such as the file system of the container image and the c-lib component, which consist of libraries and dependencies that need to be loaded for each container. For example, in a read-and-write benchmark test on the file system, the measured metric P_i for a workload Wi of an I/O benchmark should be given by Equation (1).

$$P_i = f(W_i, \textit{e-gm}, \textit{hOS-Fs}, \textit{c-FS}, \textit{c-lib/dependencies-loading}, \textit{U-p}), \tag{1}$$

where e-gm is the management or processing component of the container engine, hOS-Fs is the performance component of the host's file system, c-FS is the performance of the layered file system of the container image, and c-lib is the component of the libraries/dependencies that need to be loaded for each container. The e-gm component would be almost negligible. Finally, the U-p component includes the case of overhead due to unforeseen causes, such as the existence of a driver that does not allow optimal utilization of the hardware. For instance, to mitigate this component and enhance the performance and efficiency of the Docker file system, it is recommended to use optimized storage drivers, like overlay2 or aufs drivers. The first one was used in our experimental set-up.

Another important component that can impact container performance is the network overhead sourced by additional network mechanisms, which are necessary for internal communication between containers and with the public network. Different Docker network configurations, such as Bridge and NAT, can result in varying performance outcomes. Similarly, LXC also allows for the creation of custom network devices for containers, and multiple methods can be used to setup a network, such as a host bridge or a NAT bridge using a service call lxc-net.

Therefore, the value of the measured metric P_i for each workload Wi of a network benchmark when the resource is the network is expected as shown in Equation (2). The component that will potentially affect performance the most will be introduced by the additional network mechanisms (Bridge, NAT, among others) and p-vNet, which are needed for internal communication between containers and between containers and the public network. The remaining components have a similar meaning regarding I/O benchmarks.

$$P_i = f(W_i, \textit{e-gm}, \textit{p-hNet}, \textit{p-vNet}, \textit{c-lib/dependencies-loading}, \textit{U-p}), \tag{2}$$

where e-gm is the management or processing component of the container engine, p-hNet is the performance of host network communication stack, p-vNet is the performance of virtual network mechanisms, c-lib is the component of the libraries/dependencies that need to be loaded for each container, and U-p is the component that includes the case of overhead due to unforeseen causes.

Finally, after listing and measuring the performance metric P_i for a workload W_i of a benchmark, it is important to quantify the overhead O_i in a percentage (%). In effect, we must treat a positive overhead value O_i in % as an indicator of performance loss, compared typically to the performance of the native system or other systems used as a baseline, as suggested by Equation (3). Conversely, a negative overhead value O_i in % is an indicator of a performance gain.

$$O_i = (P_b - P_i)/P_b \times 100\%, \tag{3}$$

where O_i is the performance overhead in %, P_b is the baseline performance, and P_i is the actual performance metric P_i for a workload W_i.

Equation (3) assumes that an increasing P_b or P_i is better. However, in some cases, both have a reverse behavior, where lower is better (e.g., latency measurement or access times), and the modification suggested in Equation (4) can be adopted. It is important to note, however, that in our present study, we did not utilize any kind of these metrics.

$$O_i = (1 - P_b/P_i) \times 100\%. \tag{4}$$

5. Methodology and Experimental Environment

This section describes the implementation aspects of the testing methodology, including the research questions, the performance features to be evaluated, and the experimental environment factors such as host systems, operating systems, and configurations, as well as the benchmarks and workloads used.

5.1. Research Questions and Performance Features

The purpose of this study is to investigate the performance of recent container-based virtualization solutions by selecting two instances, Docker vs. LXD/LXC. Objectively, we aim to perform a quantitative comparison of performance in different usage scenarios. Empirically, we aspire to answer the following three research questions (RQs):

RQ1: Does Docker induce the same overhead on different operating systems?
RQ2: How do different container environments compare with native systems in terms of performance?
RQ3: How do Docker and LXD/LXC containers compare with each other in terms of performance?

The features or performance aspects of the containers to be evaluated are composed of the physical part and its capacity. For instance, the processing (CPU) and its capacity is typically measured by the CPU speed or CPU score. In the present study, we identify three candidate features essential to the performance evaluation of containerization solutions (the processor, the network communications, and I/O storage).

Processing (CPU): This refers to the processing offered by each type of container analyzed. Evaluating the CPU performance associated with a system container or application contender will reflect the overhead in processing.

I/O Storage: This refers to data transfers within the data storage system.

Network Communications: This refers to data transfers between container clients and between different containers or between different container instances. The evaluation of data transfer rates will reflect the overhead of network resources.

5.2. Host Systems, Operating Systems, and Configurations

This section details various experimental factors, such as the characteristics of the computer systems and operating systems used in the tests and important considerations in the configurations.

5.2.1. Host Systems

During the creation of the testbed, two computer systems with standard characteristics (CPU, RAM, GPU, storage, and network) were chosen to perform and repeat different evaluation experiments and stress the virtualized environment. It was intentional to choose two distinct platforms and two different vertices in terms of performance but of typical consumer use. The characteristics of the host systems used in the benchmarks are as follows:

(1) Intel Laptop

- CPU: Intel Core i7 6700 HQ;
- RAM: 16 GB (2 × 8 GB) DDR4 2133 Mhz CL15;
- Storage: SSD NVMe Gen3 × 4 Kingston NV2 500 GB;
- Network: Killer E2400 Gigabit.

(2) AMD Desktop

- CPU: AMD Ryzen 7 5900X @5.0 Ghz;
- RAM: 16 GB (2 × 8 GB) DDR4 3600 Mhz CL16;
- Storage: SSD NVMe Gen4 × 4 Samsung 980 Pro 1 TB;
- Network: Realtek 8111H Gigabit.

5.2.2. Operating Systems

In the configuration step of the benchmark environments, Docker Desktop was installed on both host operating systems used (Windows 10 22H2 and Garuda Linux), and LXD was installed on the Linux host operating system running an Arch-based system container. These were chosen for the following reasons.

Windows [35]: Windows 10, despite being replaced by Windows 11, is still currently one of the most widely used OSs for personal use, with a large and extensive documentation. Windows 10 is compatible both as host OS and guest OS, with different virtualization and containerization platforms, such as Oracle VirtualBox, Hyper-V, and Docker, which makes it one of the preferred OSs in virtualization, containerization, and testing environments.

Garuda Linux Xfce [36]: Among several existing options (e.g., Debian, Ubuntu, Garuda), Garuda Linux Xfce was chosen, because it is an OS based on Arch that is light, stable, and efficient. Also, it is a "ready-to-use" distribution, with several packages already pre-installed with good device driver support. Xfce is a stable and lightweight desktop environment that is visually appealing and easy to use.

Arch Linux [37]**:** Arch Linux is a lightweight and highly customizable Linux distribution known for its simplicity and flexibility. It follows a "do-it-yourself" approach, allowing users to build their systems from the ground up. With its minimalistic design and focus on user control, Arch was chosen as the base image for all LXD containers.

5.2.3. Specific Configurations

The host systems where the tests were conducted were equipped with all software updated to the date of execution, including all programs, system updates, and drivers.

Docker and LXD were configured to use only four threads, and, for better performance and compatibility on the systems, VT-d was enabled in the BIOS of the Intel computer and secure virtual machine (SVM) on the AMD computer.

5.3. Benchmarks and Workloads

In this study, the selection of software testing tools or utilities, also called benchmarks, which allow for analyzing and comparing the performance between several systems, was based on specific requirements. These included the need for multiplatform support and compatibility with the containerization technologies targeted of our analysis, as well as offering reliable measurements of relevant metrics.

Table 2 outlines the benchmarks selected by the category of resource to be evaluated and the performance metric measured. Some of tools may test more than one performance aspect.

Table 2. Selected benchmarks.

Physical Properties	Benchmarks	Performance Metrics
Processing	Geekbench6	Single-core/multi-core score
Network communications	Iperf3	Network speed, packet loss
I/O Disk	CrystalDiskMark	Read/write speed

Next, certain considerations are described and made for each tool selected.

GeekBench6 [38]: Geekbench6 is a cross-platform CPU benchmarking utility. It runs a set of different CPU tests to evaluate different aspects of CPU performance. Two key CPU performance values are returned, single and multi-core scores. The most relevant tests, which deserved our focus, were Clang, Asset Compression, and Ray Tracer, because they showed the best scalability in terms of the number of threads and memory usage.

CrystalDiskMark [39]: CrystalDiskMark is a tool that allows for evaluating the performance of storage drives, such as hard disks and USB flash drives. It performs read-and-write tests with different block sizes and conditions. Sequential read/write tests can be performed to measure the maximum read and write speeds that the unit can reach or random read/write tests.

iPerf3 [40]: Iperf3 is a tool used to perform network tests, which can create TCP and UDP streams and then measure the corresponding throughput of a network. Iperf3 allows for the definition of various parameters to test or optimize a network. Like CrystalDiskMark, it was chosen, because it is a simple, portable, and cross-platform tool.

The workloads used to stress each performance aspect of the containers and the native machines used as reference include the following factors.

Iterations and durations: Each instance (native system, Docker, and LXD/Linux) and each performance aspect (CPU, I/O storage, and network) was tested five times. A two-minute waiting period was introduced between tests to minimize interference from previous processes or cached data in the system's performance; additionally, all containers were restarted between tests to ensure consistency and validity of the results.

To obtain repeatable, comparable, and scientifically relevant results, we kept as many of the parameters as similar as possible. The major example was CPU tests, where each test was conducted with each environment limited to four threads to ensure an easier and fair comparison with each other, as some containerization environment setups are limited on the number of threads that can be assigned to them.

Workload size: The workload sizes were defined by the benchmark tools used. For benchmarking I/O device performance with CrystalDiskMark, it was configured to use 2 GB blocks and to saturate the 1 GB cache of both SSDs. The number of loops was set to five to demonstrate consistent performance. In network benchmarks using iPerf3, the maximum bandwidth was set for both TCP and UDP tests to accurately measure the network throughput.

In this the study, we analyzed the differences in performance by calculating a geometric mean of the various test runs, which were executed multiple times to ensure data accuracy according to best practices in benchmarks [41].

6. Experimental Results and Analysis

In this section, we present the experimental results obtained based on the employed metrics measured to quantify the performance and overhead differences between both native systems, the various containerization environments, and distinct hardware configurations. Finally, our analysis seeks to identify trends present within the collected data.

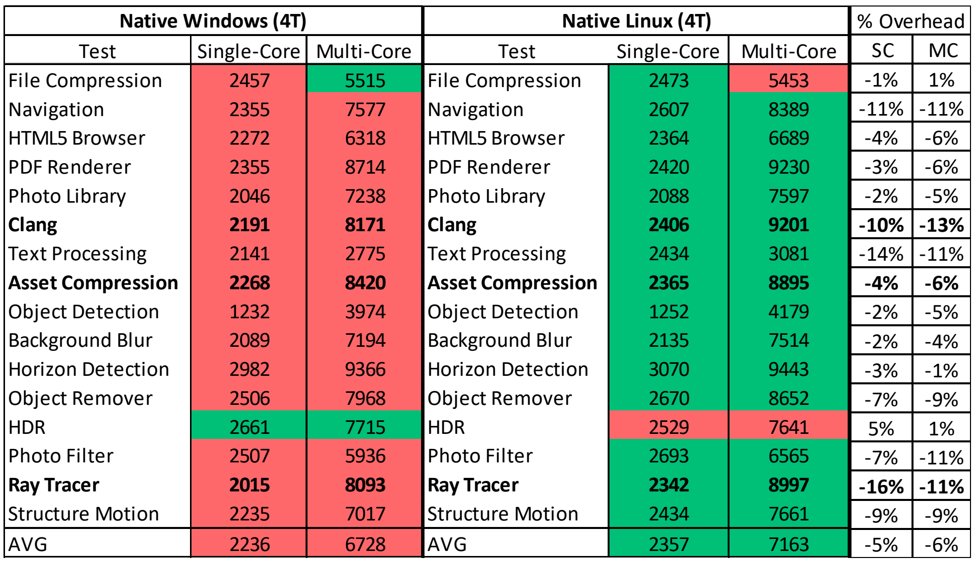
6.1. CPU Benchmarks

The CPU benchmark results will be presented in various tables with color-coded heatmaps to facilitate reading comprehension. Within each table, the last two columns will present the overhead results in percentages, where a positive number reflects an uplift and a negative number represents an overhead (as discussed in Section 4).

In the following tests, the benchmarking tool Geekbench6 was employed to assess the performance of the processor in the machines and containers. This tool encompasses multiple tests aiming to evaluate various aspects of processor performance. As mentioned earlier in Section 5.3, the Clang, Asset Compression, and Ray tracer tests of the Geekbench tool were given relevance, as they proved to be more scalable in terms of the number of threads. The results of these particular tests will be highlighted in bold.

6.1.1. Native System CPU Benchmarks

For the initial set of tests regarding the CPU benchmarks, we conducted evaluations on native systems, and the resulting performance results are shown in Figure 2. The data presented is from tests performed on the AMD desktop system and compares the performance between Windows and Linux OS.



Native Windows (4T)			Native Linux (4T)			% Overhead	
Test	Single-Core	Multi-Core	Test	Single-Core	Multi-Core	SC	MC
File Compression	2457	5515	File Compression	2473	5453	-1%	1%
Navigation	2355	7577	Navigation	2607	8389	-11%	-11%
HTML5 Browser	2272	6318	HTML5 Browser	2364	6689	-4%	-6%
PDF Renderer	2355	8714	PDF Renderer	2420	9230	-3%	-6%
Photo Library	2046	7238	Photo Library	2088	7597	-2%	-5%
Clang	**2191**	**8171**	**Clang**	**2406**	**9201**	**-10%**	**-13%**
Text Processing	2141	2775	Text Processing	2434	3081	-14%	-11%
Asset Compression	**2268**	**8420**	**Asset Compression**	**2365**	**8895**	**-4%**	**-6%**
Object Detection	1232	3974	Object Detection	1252	4179	-2%	-5%
Background Blur	2089	7194	Background Blur	2135	7514	-2%	-4%
Horizon Detection	2982	9366	Horizon Detection	3070	9443	-3%	-1%
Object Remover	2506	7968	Object Remover	2670	8652	-7%	-9%
HDR	2661	7715	HDR	2529	7641	5%	1%
Photo Filter	2507	5936	Photo Filter	2693	6565	-7%	-11%
Ray Tracer	**2015**	**8093**	**Ray Tracer**	**2342**	**8997**	**-16%**	**-11%**
Structure Motion	2235	7017	Structure Motion	2434	7661	-9%	-9%
AVG	2236	6728	AVG	2357	7163	-5%	-6%

Figure 2. Native Windows (4T) vs. native Linux (4T).

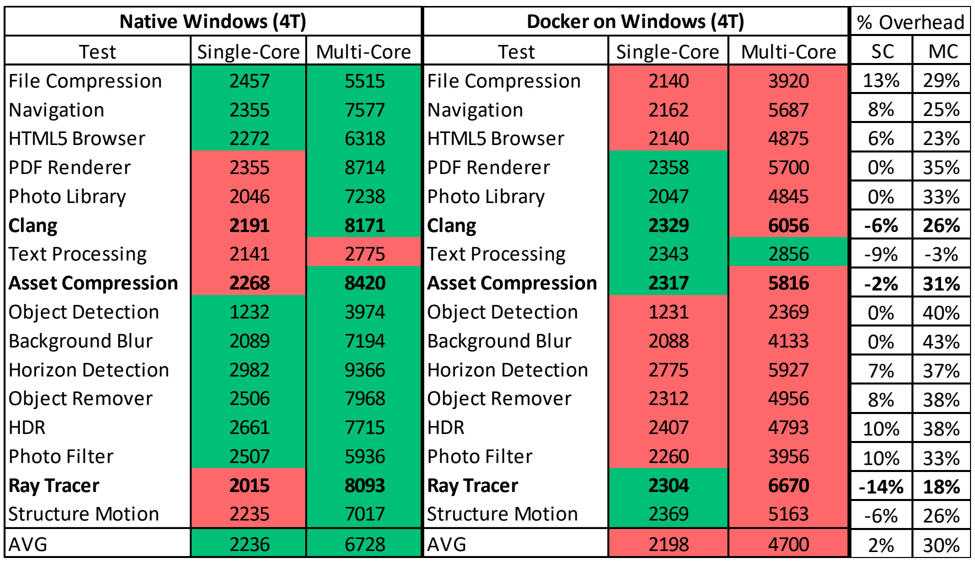
Upon analyzing the data from native Windows versus native Linux in CPU benchmarks, it was observed that Linux OS achieved better processor performance, exhibiting a performance gain of 5% in single-core and 6% in multi-core over OS Windows, as indicated by the negative sign of overhead. In the most scalable tests, denoted in bold, the performance increase was slightly above average, increasing 8% in single-core and 9.5% in multi-core. Notably, Clang and Ray Tracer workloads clearly show Linux OS outperform Windows OS, indicating Linux OS can handle highly demanding developer tasks, such as compiling code as well as intensive image synthesis tasks, such as creating artificial images. The results heatmap, along with the last two columns of the figure representing overhead, or in this case mostly uplifts, reveals that the Linux OS natively makes more efficient use of system resources. Only on the HDR and file compression tests, Linux shows a slight downward trend; however, it more than makes up for it in other areas.

6.1.2. Overhead Implications of Docker

As stated in Section 2.2, the Docker containerization software is implemented differently depending on the operating system being used. In this section, we evaluate how well these different implementations are employed.

First, we started with the Windows OS running Docker and running on the AMD desktop system, with both environments being limited to four threads, as referred previously. From the analysis of the results, we can infer that the Docker implementation on Windows suffers from various overhead issues.

As shown in Figure 3, while the single-core scores reveal only slight performance degradation with only about a 2% of overhead compared to their native counterparts, there is a huge downward trend in the multi-core scores, corresponding to about 30% overhead. From the more scalable results, marked in bold, we can see that they are higher on average than the overall mean, both on single-core and multi-core results. This suggests that it is not solely an issue related to different scalability but rather to inferior resource management.

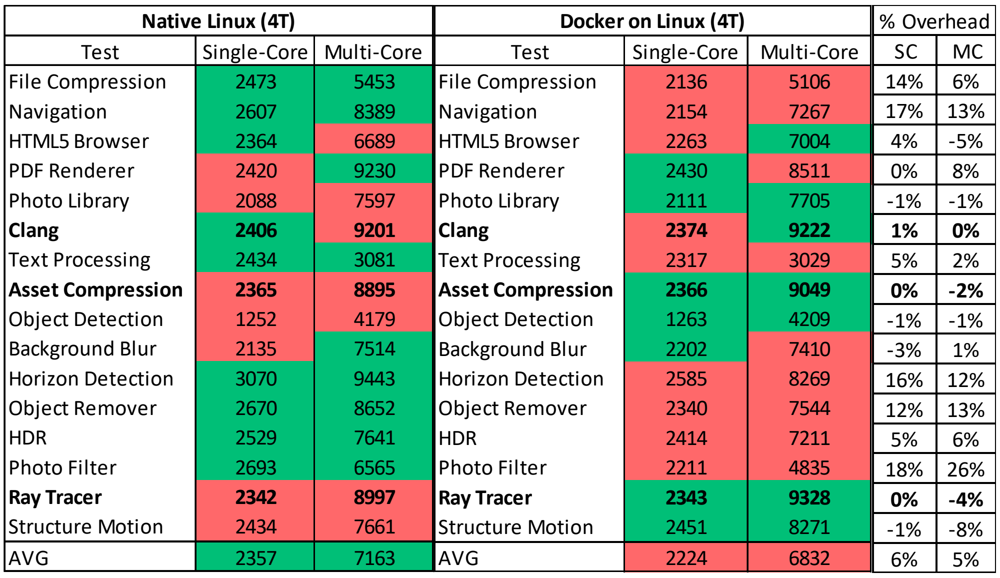


Native Windows (4T)			Docker on Windows (4T)			% Overhead	
Test	Single-Core	Multi-Core	Test	Single-Core	Multi-Core	SC	MC
File Compression	2457	5515	File Compression	2140	3920	13%	29%
Navigation	2355	7577	Navigation	2162	5687	8%	25%
HTML5 Browser	2272	6318	HTML5 Browser	2140	4875	6%	23%
PDF Renderer	2355	8714	PDF Renderer	2358	5700	0%	35%
Photo Library	2046	7238	Photo Library	2047	4845	0%	33%
Clang	**2191**	**8171**	**Clang**	**2329**	**6056**	**-6%**	**26%**
Text Processing	2141	2775	Text Processing	2343	2856	-9%	-3%
Asset Compression	**2268**	**8420**	**Asset Compression**	**2317**	**5816**	**-2%**	**31%**
Object Detection	1232	3974	Object Detection	1231	2369	0%	40%
Background Blur	2089	7194	Background Blur	2088	4133	0%	43%
Horizon Detection	2982	9366	Horizon Detection	2775	5927	7%	37%
Object Remover	2506	7968	Object Remover	2312	4956	8%	38%
HDR	2661	7715	HDR	2407	4793	10%	38%
Photo Filter	2507	5936	Photo Filter	2260	3956	10%	33%
Ray Tracer	**2015**	**8093**	**Ray Tracer**	**2304**	**6670**	**-14%**	**18%**
Structure Motion	2235	7017	Structure Motion	2369	5163	-6%	26%
AVG	2236	6728	AVG	2198	4700	2%	30%

Figure 3. Native Windows (4T) vs. Docker Windows (4T) AMD Desktop.

It sounds like the promising Hyper-V/WSL2 technologies employing in Windows OS may need additional refinements or special attention from users during Docker configurations, even though single-core results show negligible overhead or performance gain when using certain workloads.

Analyzing the Linux results on the AMD desktop system, with both environments being restricted to four threads, a different set of conclusions emerges when comparing the native Linux performance to running Docker on Linux. Both environments are seen trading blows depending on the workload and the workload type, single- or multi-core, as can be seen on the results heatmap presented in Figure 4.



Native Linux (4T)			Docker on Linux (4T)			% Overhead	
Test	Single-Core	Multi-Core	Test	Single-Core	Multi-Core	SC	MC
File Compression	2473	5453	File Compression	2136	5106	14%	6%
Navigation	2607	8389	Navigation	2154	7267	17%	13%
HTML5 Browser	2364	6689	HTML5 Browser	2263	7004	4%	-5%
PDF Renderer	2420	9230	PDF Renderer	2430	8511	0%	8%
Photo Library	2088	7597	Photo Library	2111	7705	-1%	-1%
Clang	**2406**	**9201**	**Clang**	**2374**	**9222**	**1%**	**0%**
Text Processing	2434	3081	Text Processing	2317	3029	5%	2%
Asset Compression	**2365**	**8895**	**Asset Compression**	**2366**	**9049**	**0%**	**-2%**
Object Detection	1252	4179	Object Detection	1263	4209	-1%	-1%
Background Blur	2135	7514	Background Blur	2202	7410	-3%	1%
Horizon Detection	3070	9443	Horizon Detection	2585	8269	16%	12%
Object Remover	2670	8652	Object Remover	2340	7544	12%	13%
HDR	2529	7641	HDR	2414	7211	5%	6%
Photo Filter	2693	6565	Photo Filter	2211	4835	18%	26%
Ray Tracer	**2342**	**8997**	**Ray Tracer**	**2343**	**9328**	**0%**	**-4%**
Structure Motion	2434	7661	Structure Motion	2451	8271	-1%	-8%
AVG	2357	7163	AVG	2224	6832	6%	5%

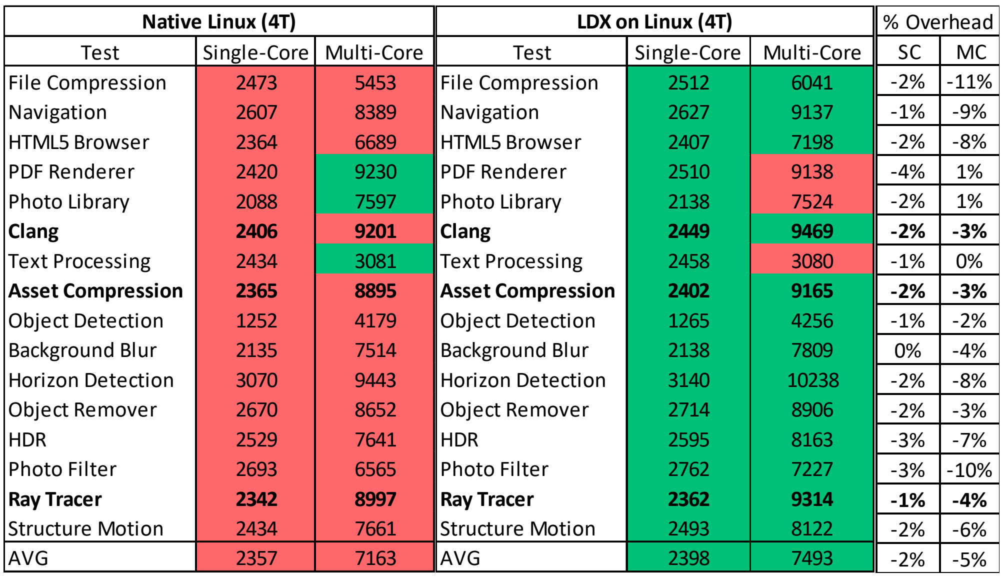
Figure 4. Native Linux (4T) vs. Docker Linux (4T) AMD Desktop.

On average, there was a 6% decrease for single-core workloads and a 5% decrease for multi-core workloads. While these results still represent a negative trend for the containerized environment, it is notably less pronounced compared to its Windows counterpart. Only on certain workloads the performance behavior may be considered poor (e.g., navigation, file compression, or photo filter). However, notably when analyzing the more scalable results, denoted in bold, we can observe an average difference remarkably close to zero percent. Based on this, we can conclude that the Linux implementation of Docker does not suffer from many scalability issues. This clearly suggests Docker containers on Linux are well-suited to encapsulate applications for highly demanding developer tasks, also including asset compression, such as 3D textual and geometric assets, as well as intensive image synthesis.

6.1.3. Overhead Implications of LXD

The last batch of tests related to CPU performance is centered on the LXD environment, introduced earlier in Section 2.2. As this environment is exclusive to Linux, there is an expectation that the results should be more favorable, especially in scenarios of multiple processes within a container and multi-core processors. In contrast, Docker should be faster while using single-core processors, because Docker is a single process per container.

Upon analyzing the results presented in Figure 5 and the resulting heatmap, we can see that that expectation is indeed met. The LXD environment on Linux stands out as the only one to consistently provide on average better performance than its native environment. This is most likely attributed to the fact that disabling threads on a native system usually disables some cache access, something that does not happen in a software-limited case like our LXD environment. Although the performance gain over the native system is modest, only 2% in single-core workloads and 5% in multi-core workloads, it represents the best or optimal scenario to run containerized software.



Native Linux (4T)			**LDX on Linux (4T)**			% Overhead	
Test	Single-Core	Multi-Core	Test	Single-Core	Multi-Core	SC	MC
File Compression	2473	5453	File Compression	2512	6041	-2%	-11%
Navigation	2607	8389	Navigation	2627	9137	-1%	-9%
HTML5 Browser	2364	6689	HTML5 Browser	2407	7198	-2%	-8%
PDF Renderer	2420	9230	PDF Renderer	2510	9138	-4%	1%
Photo Library	2088	7597	Photo Library	2138	7524	-2%	1%
Clang	**2406**	**9201**	**Clang**	**2449**	**9469**	**-2%**	**-3%**
Text Processing	2434	3081	Text Processing	2458	3080	-1%	0%
Asset Compression	**2365**	**8895**	**Asset Compression**	**2402**	**9165**	**-2%**	**-3%**
Object Detection	1252	4179	Object Detection	1265	4256	-1%	-2%
Background Blur	2135	7514	Background Blur	2138	7809	0%	-4%
Horizon Detection	3070	9443	Horizon Detection	3140	10238	-2%	-8%
Object Remover	2670	8652	Object Remover	2714	8906	-2%	-3%
HDR	2529	7641	HDR	2595	8163	-3%	-7%
Photo Filter	2693	6565	Photo Filter	2762	7227	-3%	-10%
Ray Tracer	**2342**	**8997**	**Ray Tracer**	**2362**	**9314**	**-1%**	**-4%**
Structure Motion	2434	7661	Structure Motion	2493	8122	-2%	-6%
AVG	2357	7163	AVG	2398	7493	-2%	-5%

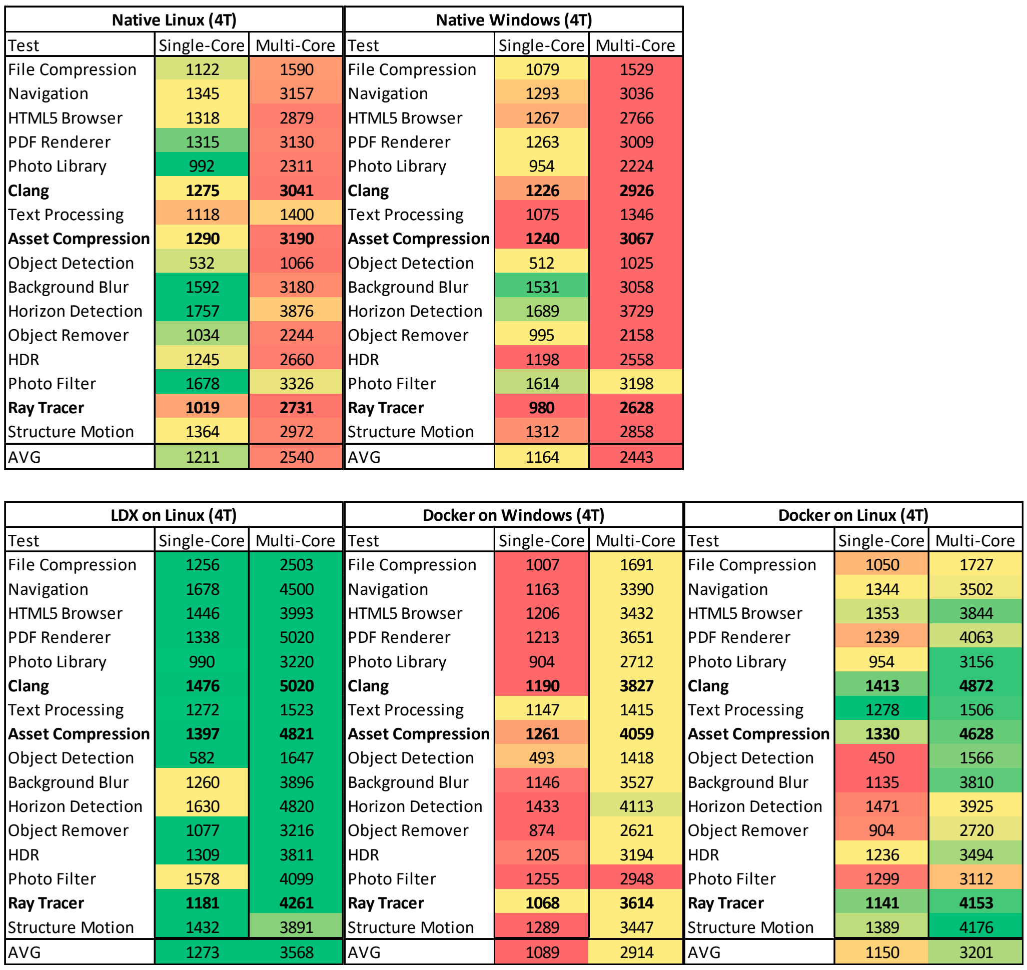
Figure 5. Native Linux (4T) vs. LDX Linux (4T) AMD Desktop.

Furthermore, taking a closer examination of the results, we can observe that the more scalable workloads, denoted in bold, have a similar average and closely mirror the global results. This, once again, suggests the fact that containerization on Linux does not present any scalability issues. Thus, as Docker containers on Linux, LXC/LXD containers are well-suited to encapsulate highly demanding developer workloads.

6.1.4. Replicability of the Results on a Different System

To ensure the robustness of our findings and to confirm if observed trends are hardware-dependent, we conducted the same benchmarks on a different system, presented in Section 5.2.1, while maintaining consistent software specifications

To summarize the results obtained, we opted again to compile them into the various heatmaps, presented below in Figure 6. These are formatted in a heat-map where the best result across each test is shaded in green, while the worst result is shaded in red.



Native Linux (4T)			Native Windows (4T)		
Test	Single-Core	Multi-Core	Test	Single-Core	Multi-Core
File Compression	1122	1590	File Compression	1079	1529
Navigation	1345	3157	Navigation	1293	3036
HTML5 Browser	1318	2879	HTML5 Browser	1267	2766
PDF Renderer	1315	3130	PDF Renderer	1263	3009
Photo Library	992	2311	Photo Library	954	2224
Clang	**1275**	**3041**	**Clang**	**1226**	**2926**
Text Processing	1118	1400	Text Processing	1075	1346
Asset Compression	**1290**	**3190**	**Asset Compression**	**1240**	**3067**
Object Detection	532	1066	Object Detection	512	1025
Background Blur	1592	3180	Background Blur	1531	3058
Horizon Detection	1757	3876	Horizon Detection	1689	3729
Object Remover	1034	2244	Object Remover	995	2158
HDR	1245	2660	HDR	1198	2558
Photo Filter	1678	3326	Photo Filter	1614	3198
Ray Tracer	**1019**	**2731**	**Ray Tracer**	**980**	**2628**
Structure Motion	1364	2972	Structure Motion	1312	2858
AVG	1211	2540	AVG	1164	2443

LDX on Linux (4T)			Docker on Windows (4T)			Docker on Linux (4T)		
Test	Single-Core	Multi-Core	Test	Single-Core	Multi-Core	Test	Single-Core	Multi-Core
File Compression	1256	2503	File Compression	1007	1691	File Compression	1050	1727
Navigation	1678	4500	Navigation	1163	3390	Navigation	1344	3502
HTML5 Browser	1446	3993	HTML5 Browser	1206	3432	HTML5 Browser	1353	3844
PDF Renderer	1338	5020	PDF Renderer	1213	3651	PDF Renderer	1239	4063
Photo Library	990	3220	Photo Library	904	2712	Photo Library	954	3156
Clang	**1476**	**5020**	**Clang**	**1190**	**3827**	**Clang**	**1413**	**4872**
Text Processing	1272	1523	Text Processing	1147	1415	Text Processing	1278	1506
Asset Compression	**1397**	**4821**	**Asset Compression**	**1261**	**4059**	**Asset Compression**	**1330**	**4628**
Object Detection	582	1647	Object Detection	493	1418	Object Detection	450	1566
Background Blur	1260	3896	Background Blur	1146	3527	Background Blur	1135	3810
Horizon Detection	1630	4820	Horizon Detection	1433	4113	Horizon Detection	1471	3925
Object Remover	1077	3216	Object Remover	874	2621	Object Remover	904	2720
HDR	1309	3811	HDR	1205	3194	HDR	1236	3494
Photo Filter	1578	4099	Photo Filter	1255	2948	Photo Filter	1299	3112
Ray Tracer	**1181**	**4261**	**Ray Tracer**	**1068**	**3614**	**Ray Tracer**	**1141**	**4153**
Structure Motion	1432	3891	Structure Motion	1289	3447	Structure Motion	1389	4176
AVG	1273	3568	AVG	1089	2914	AVG	1150	3201

Figure 6. Tests of various environments in the Intel system.

Upon analyzing these heatmaps, we can confirm most, if not all, trends noticed before. As a native system, Linux offers better performance compared to Windows in containerized environments. Docker on Linux outperforms Docker on Windows, and the best performance for containerized environments is achieved using LXD on Linux.

The only oddity is presented in the comparison between a native Windows system and Docker running on Windows, where, against expectations, the containerized environment performs better than its native counterpart.

6.2. I/O Benchmarks

In this category of tests, the benchmarking tool CrystalDiskMark was used to run various tests and ascertain the various performance metrics of the input and output (I/O) system of storage. Two specific tests were chosen, sequential writes and random reads, as they best represent the average use case for a normal user.

These tests were run multiple times with a two-minute break in between on the native environments and a complete reboot on the containerized environments. This procedure was performed to ensure that no past results were cached, providing an accurate evaluation of the environment's true performance. As mentioned before, these tests were also conducted with an increased file size to saturate the SSD controller's cache.

The results will be presented next in the upcoming subsections. Now, we opt to present the averaged raw results for each system graphically, where higher values indicate better performance. Additionally, we provide extra graphs illustrating the performance overhead as a percentage (%) of performance loss compared to the baseline system, again for the Native Windows, where a positive number represents an effective overhead, and a negative number represents a gain.

6.2.1. Sequential Writes

The sequential write test results are now presented in Figures 7 and 8. Firstly, it is observed that, when using the Windows OS, both systems provided performance close to the theoretical maximum for each disk, 2800 MB/s for the Intel system and 5000 MB/s for the AMD system. This does not occur in the Linux OS, which demonstrates a decrease in write speeds on the Intel system, with a corresponding overhead of over 20%. This observation remained consistent in all tests, leading to the conclusion that this OS may have less compatibility with this hardware combination, as in the AMD system the decrease was only about 5%.

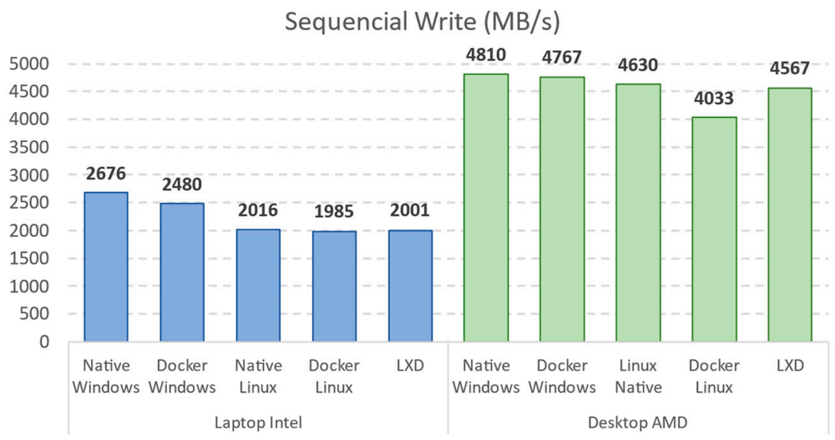


Figure 7. I/O benchmarks sequential write (MB/s).

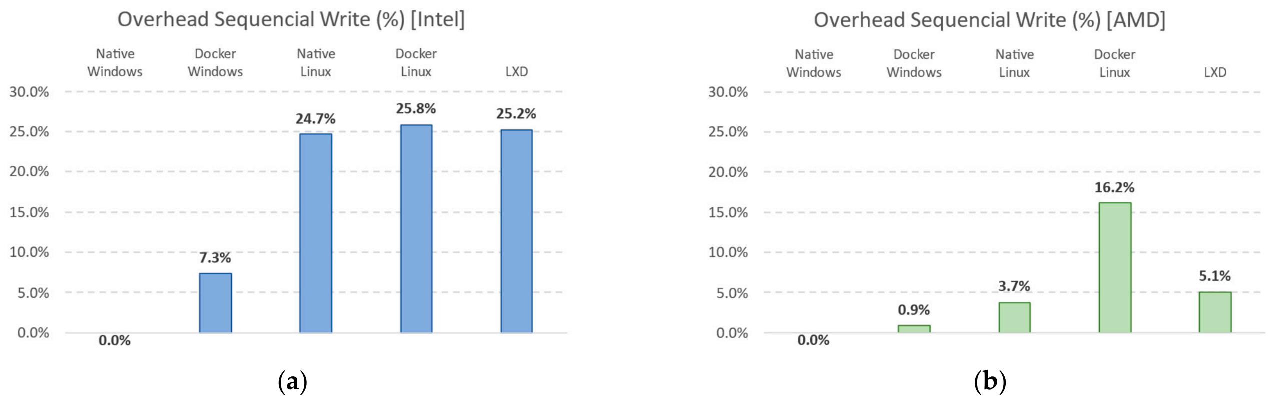


Figure 8. I/O benchmarks overhead of sequential write (%). (**a**) Overhead on an Intel-based system; (**b**) overhead on an AMD-based system.

Secondly, in both Docker implementations for both systems, lower write speeds were observed, which is expected, as Docker is more susceptible to overhead due to the inefficiency of the layered file system used in the containers. The AMD-based system demonstrated less overhead, benefiting from its higher CPU performance.

In contrast, the LXD environment outperforms Docker and closely approaches the performance of the native Linux system. In particular, it exhibits a superior performance on the AMD system.

6.2.2. Random 4K Read

Figures 9 and 10 show the results of the 4K random read test. Their analysis shows that the overhead caused by Docker on Windows affects the performance of both systems, with 27% overhead on AMD systems and 65% overhead on Intel systems.

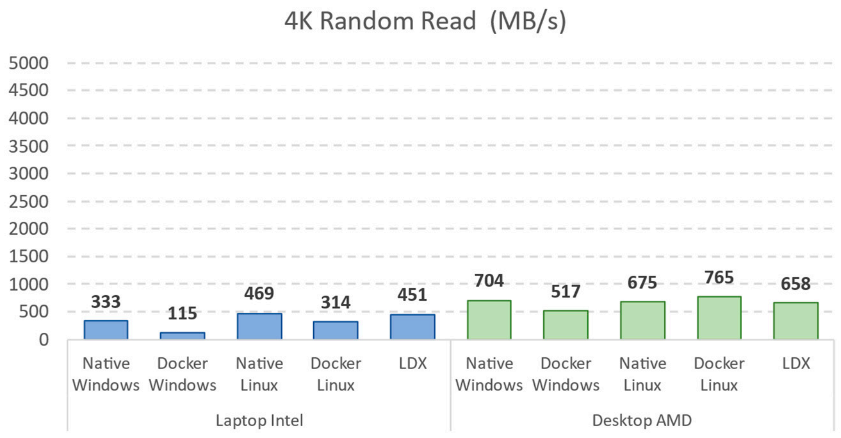


Figure 9. I/O benchmarks random reads (MB/s).

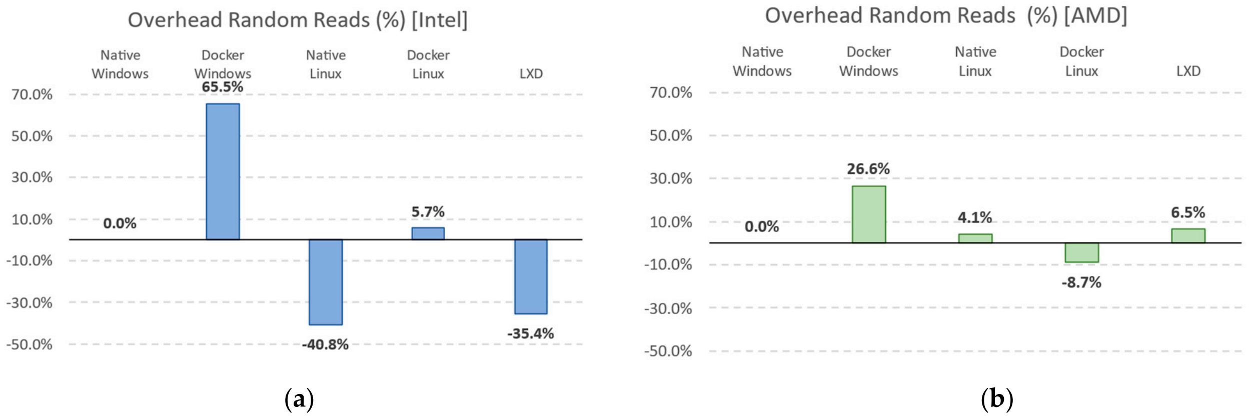


Figure 10. I/O Benchmarks overhead and uplifts of random reads (%): (**a**) Intel-based system; (**b**) AMD-based system.

On Linux, there are two opposite trends. On the Intel system, there is a notable performance uplift of 41% while using a native Linux OS compared to the baseline performance of native Windows and 35% while using a LXD environment. In other words, LXD adds a 6% overhead. On the other hand, using Docker on Linux shows a 6% performance overhead, which, when combined with the 41% performance uplift from Linux, effectively transforms that overhead into 47%, nearly halving its performance.

The other Linux trend is present in the AMD-based system, where it is compared to the baseline values of Windows. Moving to a Linux-based system introduces an overhead of 4% instead of a significant uplift. LXD results also show a similar trend. Finally, the Linux-based Docker results also show an inverse trend to the Intel results, gaining a 9% uplift instead of experiencing a small decrease.

6.3. Network Benchmarks

In the network benchmarks, to ascertain the network performance of the studied instance types (native, Docker, and LXD/LXC), the iPerf3 tool was used, and configured to use unlimited bandwidth. The tests were performed on a 1 GB/s internal network.

To ensure accuracy, these tests were conducted multiple times with a two-minute break in between on the native environments and a complete reboot on the containerized environments. This procedure was employed to eliminate any potential influence from past cached results, ensuring an accurate evaluation of the true performance of the environment.

In the following subsections, we present the network results graphically, considering TCP and UDP traffic performance. The average results of each system are presented with raw numbers.

6.3.1. TCP Bandwidth

Figure 11 shows the results of the network benchmarks in inbound TCP traffic. From these very similar and consistent results, it can be deduced that in a common network setup, the use of Docker and LXD tools adds minimal to no overhead during transfers on both Windows and Linux OS.

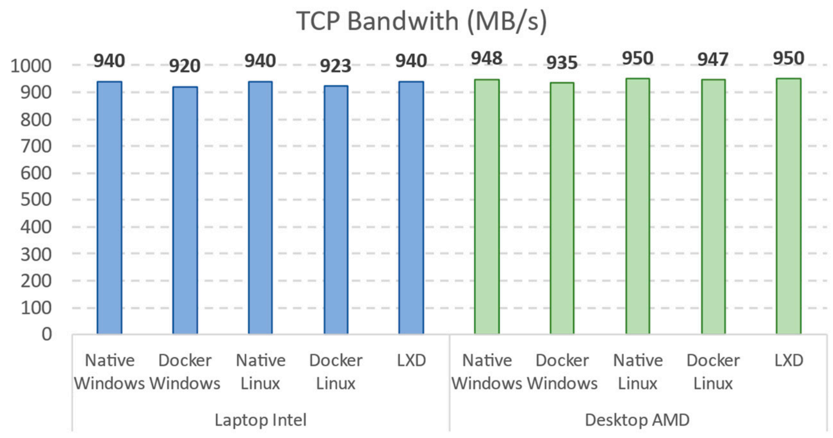


Figure 11. TCP bandwidths (MB/s).

6.3.2. UDP Bandwidth and Packet Loss

For the case of the tests involving UDP-based inbound traffic, Figure 12 shows the measured bandwidths, while Figure 13 presents the corresponding packet loss rates in the various environments.

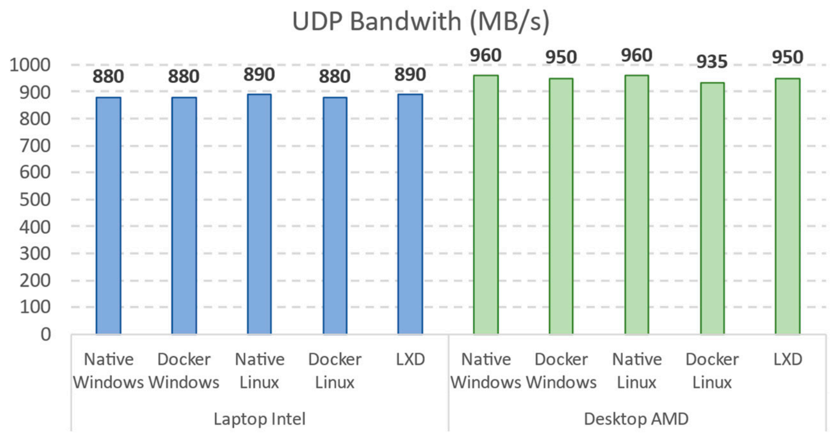


Figure 12. UDP bandwidths (MB/s).

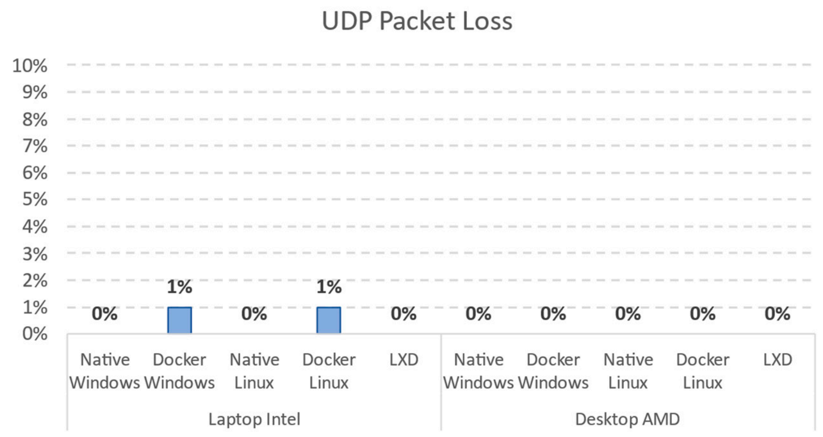


Figure 13. UDP packer loss (%).

Similar to the TCP results, in a conventional network setup, the use of Docker and LXD adds minimal to no overhead on network UDP communications in both environments. Only a slight packet loss was observed in the Docker tests on both Windows and Linux OSs, specifically on the less powerful Intel-based system, where it experienced a barely noticeable packet loss of 1%.

7. Discussion, Conclusions, and Future Work

The results and the analysis conducted in this study lead us to the conclusion that Docker and LXD are both powerful and useful container solutions in various contexts and circumstances. In effect, through extensive benchmarks and adopting well-known workload metrics such as CPU scores, disk speed, and network throughput, our results showed a clear overall trend toward meritorious performance and the maturity of both technologies; even we also observed cases of divergent variations and trends.

In network performance tests, in contrast to what was expected for container technologies, the containerization solutions studied performed identically, and they virtually equaled native systems and contributed to negligible overhead, making them not a limiting factor on network performance.

In I/O tests, the same overhead was expected, mainly due to the use of layered file systems or the use of non-optimized drivers/HW. After completing the analysis of the I/O benchmarks results, it was not possible to indicate a general optimal recommendation for all usage types.

In our results, divergent variations and trends were observed; even though LXD tends to produce less overhead, it didn't perform well under all circumstances. However, it is important to note that, in environments with good processing power, containers demonstrated good scalability. It is, therefore, recommended that containers for I/O-intensive tasks be pre-tested and implemented case-by-case. In tasks where sequential write performance is essential, like audio or video production, the difference becomes less noticeable, thus making the use of containers recommended. In tasks where 4k write performance is essential, like database management, a careful evaluation of the type of virtualization is also recommended.

In closing, it was observed that, of the tested tools, LXD consistently obtained superior results, being the most stable platform in terms of performance. This, as well as the operating system where it is located, Linux points to being an excellent choice for containerization, something already expected due to the large adoption of Linux in this area.

While this study provides valuable insights into the performance and overhead of Docker and LXD containers, there are some limitations to consider. First, comparing workload metrics, such as CPU values, disk speeds, and network throughput, does not provide the full range of performance characteristics for all types of applications and scenarios. Different workloads can behave differently, and additional metrics and real-world use cases can provide a more complete understanding of container performance.

Additionally, these research results are based on a specific set of hardware and software configurations, and results may not be directly applicable to different environments. Factors such as underlying infrastructure, network architecture, and host configuration can affect container performance and should be considered when generalizing results. It is also worth noting that there is a lack of research regarding the performance of containers in ARM-based systems. As ARM-based systems gain popularity in various domains, investigating container performance on such platforms becomes increasingly important for a comprehensive understanding of container technology's capabilities and limitations.

To mitigate these limitations, further research should explore a broader range of metrics, incorporating a variety of workloads and real-world use cases.

Furthermore, to improve further investigations and to offer a more exhaustive testing suite, things of note we recommend being improved would have been the following.

Firstly, to increase the maximum bandwidth of the network tests, even though our methodology is thorough, it is limited to a consumer-grade gigabit connection, while in some containerized use cases, mostly cloud-based ones, this connection speed could easily exceed the 10-gigabit range. Further network intensive benchmarks may also play an important role when selecting certain drivers (e.g., bridge, host, or overlay), considering factors such as container communication requirements, network isolation, compatibility with host hardware, or complexity of configuration.

Secondly, to test further scalability of containers benchmarks, multiple instances may be deployed and run concurrently on a single system or across a cluster system. This would involve utilizing orchestration tools for automating the deployment, scaling, and management of containerized applications. Despite the fact that LXD lacks built-in orchestration tools, Docker provides robust orchestration options like Docker Swarm and Kubernetes that can be the target of several compelling benchmark studies, including analyzing the latency impacts on users' experiences and the throughputs achievable with these platforms. However, conducting these studies would require more capable hardware to test these tools and technologies' true limits but would offer important insights for use case in on-premises or cloud environments, especially when adopting CI/CD best practices.

Finally, the assessment of energy efficiency of both studied container-based solutions emerges as another crucial research direction to balance or control the energy consumption, while maintaining quasi-optimal performance across diverse workloads.

Author Contributions: Conceptualization, D.S., J.R. and A.F.; methodology, D.S., J.R. and A.F.; software, D.S. and J.R.; validation, D.S., J.R. and A.F.; formal analysis, D.S., J.R. and A.F.; investigation, D.S., J.R. and A.F.; resources, D.S. and J.R.; data curation, D.S. and J.R.; writing—original draft preparation, D.S. and J.R.; writing—review and editing, A.F.; visualization, A.F.; supervision, A.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. AbdElRahem, O.; Bahaa-Eldin, A.M.; Taha, A. Virtualization security: A survey. In Proceedings of the 2016 11th International Conference on Computer Engineering & Systems (ICCES), Cairo, Egypt, 20–21 December 2016; pp. 32–40. [CrossRef]
2. Dan Marinescu, D. *Cloud Computing: Theory and Practice*, 3rd ed.; Elsevier: Amsterdam, The Netherlands, 2022.
3. Silva, D.; Rafael, J.; Fonte, A. Virtualization Maturity in Creating System VM: An Updated Performance Evaluation. *Int. J. Electr. Comput. Eng. Res.* **2023**, *3*, 7–17. [CrossRef]
4. Casalicchio, E.; Iannucci, S. The Emiliano state-of-the-art in container technologies: Application, orchestration and security. *Concurr. Comput. Pract. Exp.* **2020**, *32*, 17. [CrossRef]
5. Popek, G.; Goldberg, R. Formal requirements for virtualizable third generation architectures. *Commun. ACM* **1974**, *17*, 412–421. [CrossRef]
6. Kim, S.; Park, H.; Choi, J. Direct-Virtio: A New Direct Virtualized I/O Framework for NVMe SSDs. *Electronics* **2021**, *10*, 2058. [CrossRef]
7. Muench, D.; Isfort, O.; Mueller, K.; Paulitsch, M.; Herkersdorf, A. Hardware-based I/O virtualization for mixed criticality real-time systems using PCIe SR-IOV. In Proceedings of the IEEE 16th International Conference on Computational Science and Engineering, Sydney, Australia, 3–5 December 2013; pp. 706–713.
8. What Is Windows Containers ? | Definition from TechTarget. Available online: https://www.techtarget.com/searchwindowsserver/definition/Microsoft-Windows-Containers (accessed on 20 January 2024).
9. Yellin, N. Modern Containers Do Not Use Chroot (Updated), Updated 11 November 2022, Robusta. Available online: https://home.robusta.dev/blog/containers-dont-use-chroot (accessed on 20 January 2024).
10. Van Laere, T. Exploring Windows Containers, 30 June 2021. Available online: https://thomasvanlaere.com/posts/2021/06/exploring-windows-containers/ (accessed on 20 January 2024).
11. Edge, J. A Seccomp Overview, 2 September 2015. Available online: https://lwn.net/Articles/656307/ (accessed on 20 January 2024).
12. Docker: Accelerated, Containerization. Available online: http://www.docker.com/ (accessed on 10 June 2023).
13. Containerd—An Industry-Standard Container Runtime with an Emphasis on Simplicity, Robustness and Portability. Available online: https://containerd.io (accessed on 20 January 2024).
14. Open Container Initiative. Available online: https://opencontainers.org (accessed on 20 January 2024).

15. Docker vs. Containerd vs. CRI-O: An In-Depth Comparison. Available online: https://phoenixnap.com/kb/docker-vs-containerd-vs-cri-o (accessed on 20 January 2024).
16. Lightweight Container Runtime for Kubernetes. Available online: https://cri-o.io (accessed on 20 January 2024).
17. Linux Containers. Available online: https://linuxcontainers.org/ (accessed on 20 January 2024).
18. Singh, S.; Singh, N. Containers & Docker: Emerging roles & future of Cloud technology. In Proceedings of the 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology, iCATccT 2016, Bengaluru, India, 21–23 July 2016; pp. 804–807. [CrossRef]
19. Run System Containers with LXD. Available online: https://canonical.com/lxd (accessed on 22 March 2024).
20. Jain, H. LXC and LXD: Explaining Linux Containers, 2 June 2016. Available online: www.sumologic.com/blog/lxc-lxd-linux-containers/ (accessed on 20 January 2024).
21. Andrei, A. What Is the Difference between Docker, LXC, and LXD Containers? 22 August 2022. Available online: https://kodekloud.com/blog/what-is-the-difference-between-docker-lxc-and-lxd-containers/ (accessed on 20 January 2024).
22. Hwang, J.; Zeng, S.; Wu, F.; Wood, T. A component-based performance comparison of four hypervisors. In Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, 27–31 May 2013; pp. 269–276.
23. Pawar, S.; Singh, S. Performance comparison of VMWare and Xen hypervisor on guest OS. *Int. J. Innov. Comput. Sci. Eng.* **2015**, *2*, 56–60.
24. Vojnak, D.; Eordevic, S.; Timcenko, S.; Strbac, M. Performance Comparison of the type-2 hypervisor VirtualBox and VMWare Workstation. In Proceedings of the 2019 27th Telecommunications Forum (TELFOR), Belgrade, Serbia, 26–27 November 2019; pp. 1–4. [CrossRef]
25. Dordevic, B.; Timcenko, V.; Sakic, D.; Davidovic, N. File system performance for type-1 hypervisors on the Xen and VMware ESXi. In Proceedings of the 2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH), Sarajevo, Bosnia and Herzegovina, 16–18 March 2022; pp. 1–6. [CrossRef]
26. Rahman, H.; Wang, G.; Chen, J.; Jiang, H. Performance Evaluation of Hypervisors and the Effect of Virtual CPU on Performance. In Proceedings of the 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), Guangzhou, China, 8–12 October 2018; pp. 772–779. [CrossRef]
27. Yadav, A.; Garg, M.; Ritika. Docker containers versus virtual machine-based virtualization. In Proceedings of the Emerging Technologies in Data Mining and Information Security 2018 (IEMIS 2018), Kolkata, India, 23–25 February 2018; Springer: Singapore, 2019; Volume 3, pp. 141–150.
28. Bhardwaj, A.; Krishna, C. Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey. *Arab. J. Sci. Eng.* **2021**, *46*, 8585–8601. [CrossRef]
29. Li, Z.; Kihl, M.; Chen, Y.; Zhang, H. Two-Stage Performance Engineering of Container-based Virtualization. *Adv. Sci. Technol. Eng. Syst. J.* **2018**, *3*, 521–536. [CrossRef]
30. Arango, C.; Dernat, R.; Sanabria, J. Performance evaluation of container-based virtualization for high performance computing environments. *Rev. UIS Ing.* **2019**, *18*, 31–42. [CrossRef]
31. Liu, H.; Luo, Y.; Chen, B.; Yang, Y. Performance Evaluation of Container Networking Technology. In Proceedings of the 2023 IEEE 3rd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA), Chongqing, China, 26–28 May 2023; pp. 815–818. [CrossRef]
32. Zhou, N.; Zhou, H.; Hoppe, D. Containerization for High Performance Computing Systems: Survey and Prospects. *IEEE Trans. Softw. Eng.* **2022**, *49*, 2722–2740. [CrossRef]
33. Tesser, R.; Borin, E. Containers in HPC: A survey. *J. Supercomput.* **2023**, *79*, 5759–5827. [CrossRef]
34. Yang, Y.; Shen, W.; Ruan, B.; Liu, W.; Ren, K. Security Challenges in the Container Cloud. In Proceedings of the 2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), Atlanta, GA, USA, 13–15 December 2021; pp. 137–145. [CrossRef]
35. Explore Windows 10 OS, Computers, Apps & More | Microsoft. Available online: https://web.archive.org/web/20210203032518/http://www.microsoft.com/en-gb/windows/ (accessed on 10 June 2023).
36. Garuda Linux | Home. Available online: https://garudalinux.org/index.html (accessed on 20 January 2024).
37. Arch Linux. A Simple, Lightweight Distribution. Available online: https://archlinux.org/ (accessed on 20 January 2024).
38. Geekbench 6—Cross-Platform Benchmark. Available online: https://www.geekbench.com (accessed on 20 January 2024).
39. CrystalDiskMark. Available online: https://crystalmark.info/en/software/crystaldiskmark/ (accessed on 20 January 2024).
40. iPerf—The Ultimate Speed Test Tool for TCP, UDP and SCTP. Available online: https://iperf.fr/ (accessed on 20 January 2024).
41. Fleming, P.J.; Wallace, J.J. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM* **1986**, *29*, 218–221. [CrossRef]