Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing

Ilias Mavridis, Helen Karatza

PII: S0167-739X(18)30576-4
DOI: https://doi.org/10.1016/j.future.2018.12.035
Reference: FUTURE 4656

To appear in: *Future Generation Computer Systems*

Received date : 16 March 2018
Revised date : 5 December 2018
Accepted date : 15 December 2018

Combining Containers and Virtual Machines to Enhance Isolation and Extend Functionality on Cloud Computing

Ilias Mavridis, Helen Karatza

Department of Informatics,
Aristotle University of Thessaloniki,
54124 Thessaloniki, Greece

Abstract

Virtualization technology is the underlying element of cloud computing. Traditionally, cloud computing has employed virtual machines to distribute available resources and provide isolated environments among users. Multiple virtual machines, with their own operating system and services, can be deployed and run simultaneously on the same physical machine on the cloud infrastructure. Recently, a more lightweight virtualization technology is being rapidly adopted and it is based on containers. A key difference between virtual machines and containers, is that containers share the same underlying operating system.

In this work, we studied the combination of these two virtualization technologies by running containers on top of virtual machines. This synergy aims to enhance containers' main drawback, which is isolation and, among others, to simplify the system management and upgrade, and to introduce the new functionalities of containerized applications to virtual machines. The benefits of this method have been recognized by big cloud companies, which have been employing this approach for years.

With this paper, we aimed to present the advantages of running containers on virtual machines and to explore how various virtualization techniques and configurations affect the performance of this method. Although, in bibliography there are a few papers that used this method partially to conduct experiments, there is not any research which considers this method from our integral aspect of view. In this study, we ran Docker containers and evaluated their performances, not only on KVM and XEN virtual machines, but also we ran Linux containers on Windows Server. We adopted an empirical approach, to quantify the performance overhead introduced by the additional virtualization layer of virtual machines, by executing various benchmarks and deploying real world applications as use cases. Furthermore, we presented for first time, how isolation is applied on virtual machines and containers, we evaluated different operating systems optimized to host containers, mechanisms of storing persistent data and, last but not least, we experimentally quantified the power and energy consumption overhead of containers running on virtual machines.

Keywords: Docker, VMs, Containers, KVM, XEN, Hyper-V, Performance Evaluation, Energy Consumption

1. Introduction

Cloud computing is a popular and commonly used computing paradigm. The core of cloud computing is the virtualization technologies that allow different users to independently access and share the same computing infrastructure. Cloud offers virtually unlimited resources that allow various services, such as log analysis [1], to scale on-demand and fulfill the growing user needs. Cloud providers adopt virtualization to improve resource utilization and management, to enhance security and portability, as well as to reduce the energy consumption and total cost of data centers [2], [3]. However all these come with a performance penalty, due to the additional overhead of the virtualization layer and the rise of system complexity.

In cloud environments there are two main virtualization technologies, a hardware-based and an operating system-based. In hardware-based technology, multiple virtual machines (VMs) can be deployed and co-exist on the same physical machine. A VM runs its own operating system (OS) and is isolated from other VMs that may be hosted on the same physical machine. VMs are the key element of Infrastructure as a Service (IaaS) cloud services and can be created on-demand by users. In the operating system-based model, containers share a single host OS and its depending libraries, drivers or binaries. Services can run in containers, in a fraction of the overhead introduced by VMs, due to lack of hardware abstraction. Containers can be described as tools for packaging, delivering and orchestrating software services and applications [4]. VMs and containers are complementary and place virtualization at different layers; therefore containers' main advantage is low performance overhead, whereas VMs offer strong isolation [5], [6].

In order to enhance security, extend functionality and promote an effective development process, we propose to combine VMs and containers by running containers on top of VMs. In case there is an acceptable performance overhead, there are several additional reasons to encourage running containers on top of VMs. For instance, by maintaining containers on top of VMs, the management and upgrade of the system become

Email addresses: imavridis@csd.auth.gr (Ilias Mavridis), karatza@csd.auth.gr (Helen Karatza)

easier, and hardware-software incompatibilities of the physical server are overcome by the hardware virtualization layer. In this way, one can handle the lack of suitable hardware of a private cloud infrastructure or the necessity to use already existing infrastructure. Additionally, with this method, the majority of public cloud providers that offer VMs as products can be used to host container-based services. Big cloud provider companies, like Amazon Web Services, Microsoft Azure and Google Container Engine, have deployed containers on VM instances already [7], [8]. Also virtualization companies, like VMware, embraced this approach and even submitted related patent applications [9].

In this work, we extended our previous study [2] in the following various directions. First, we presented how isolation is enforced in various virtualization technologies and methods. Second, in order to create and manage VMs, except for KVM on Linux, we studied XEN on Linux and we also explored how Windows Server attains to create and run Linux containers, by exploiting the Hyper-V hypervisor. Third, whereas in our previous work we ran containers on popular server-oriented Linux OSes, like Ubuntu Server and CentOS, this time we deployed five different container-optimized Linux OSes and also Windows Server on a more powerful infrastructure. Fourth, we used a new set of benchmarks with various parameters and configurations to stress the system in several ways. Fifth, we deployed SQL and NoSQL database services as use cases to measure how the performance overhead, which is introduced by the additional virtualization layer of the VM, is affected by the type of workload, type of storage mechanism, number of VMs, number of containers and users. And finally, we gathered power data to quantify the energy consumption overhead for the whole system and CPU separately, for system in idle state and under stress.

Our goal is to study and provide an insight into the isolation, performance and energy consumption of containers running on top of VMs. The remainder of this paper is organized as follows. Section 2 provides an overview of the related research. Section 3 presents virtualization techniques and how virtualization provides isolation. In Section 4 the experimental results are demonstrated and in Section 5 one can find the performance overhead results of our use cases. Section 6 presents power consumption overhead results. Finally, concluding remarks are presented in Section 7.

2. Related Work

By reviewing the literature, we noticed that virtualization technologies is an interesting topic and subject of study for many papers. We also identified that VMs and recently containers do play a vital role in cloud computing and in other emerging distributed computing paradigms [10]. Overall, we recognized three main categories. First, there are studies that analyzed and compared different hypervisors (also called virtual machine monitors - VMMs). Second, there are studies that compared container-based virtualization technologies and finally there are studies where both VMs and containers are compared and in some cases combined.

A performance comparison of VMware, XEN and Hyper-V hypervisors is presented by Abdellatief et al. [11]. They conducted various experiments with SQL workloads for numerous VMs and system loads. Their measurements showed that for light load, VMware had the highest SQL database performance (lower CPU utilization and Higher Order Per Minutes), then followed XEN and last came Hyper-V. For heavy load and higher number of VMs per hypervisor, VMware still had the highest SQL database performance but, in this case, the second higher performance was achieved by Hyper-V and lastly XEN. Also, they measured the host server CPU utilization and memory consumed by the VM(s). They found that for light load, XEN was the most efficient, followed by Hyper-V and then VMware. On the contrary, for higher load, the most efficient was Hyper-V, followed by VMware and XEN. Specifically in the case of XEN, the system's memory reached 98% consumption and made the server respond rather slowly.

Hwang et al. in [12] compared Hyper-V, KVM, vSphere and XEN. They concluded that the performance of each hypervisor can vary significantly, depending on the type of application, the workload and the resources assigned to each VM.

Nussbaum et al. [13] studied the I/O performance of KVM and XEN. They found that, in some areas, one outperforms the other. In [14] the authors, based on KVM's near-native performance and feature-rich experience, claimed that KVM is a better choice compared to XEN and Virtual Box. In [15] the three most widespread virtualization frameworks, XEN, KVM, and VMware ESXi were analyzed from a HPC perspective. Based on these hypervisors, they concluded that their virtualization layer introduces a substantial performance cost in every case.

The performances of the three open source hypervisors OpenVZ, XEN and KVM was measured and analyzed by Che et al. [16]. The authors took measurements, both as a black box and as a white box, for their macro and micro performances on the virtualization of the operating system. The experimental results showed that, for their setup, OpenVZ achieved the best performance.

The container-based virtualization has been studied by Xavier et al. [17] who compared the Linux VServer, OpenVZ and Linux Containers (LXC) as container-based systems, by running MapReduce workloads. They found that all three systems reached a near-native performance for these workloads and had many management capabilities, such as performance isolation, checkpoint and live migration. However, it is not recommended to use containers for security isolation [7].In [18] Docker containers were used in workflows. This way the authors observed a great reduction in required resources and they highlighted that it is possible for non-computer science experts to integrate third-party applications into new or existing workflows.

In more recent years, researchers analyzed and compared both hypervisors and containers. Estrada et al. [19] benchmarked the KVM and XEN hypervisors, as well as LXC, based on sequence alignment software that arranges sequences of DNA. They identified the differences and similarities of the runtime performance of each virtualized environment mentioned and they compared their runtime against a physical server.

Kozhirbayev et al. [20] used the NeCTAR Research Cloud to deploy Docker and Flockport (LXC) containers. The authors compared the performance of CPU, memory, network bandwidth and latency and storage overheads of these two container technologies to the performance of native NeCTAR Research Cloud instance. Their various experiments showed that there were roughly no overheads on memory utilization or CPU for both container technologies. Whereas, I/O and operating system interactions incurred some overheads.

KVM hypervisor was compared to Docker container manager in [21]. The authors ran several benchmarks and took performance measurements for CPU, memory, storage, and networking. Their benchmark results showed that containers achieved equal or better performance than VMs in almost all cases. In [22] the authors made a similar analysis with [21] but they also studied LXC as a container-based solution and OSv as a new type of lightweight Guest OS. They concluded that the containers' overhead is almost negligible, whereas the other virtualization methods provide higher security.

Xavier et al. [23] presented the container-based virtualization as an alternative to hypervisors in HPC. They conducted several experiments and found that XEN was outperformed by container-based virtualization implementations, such as Linux VServer, OpenVZ and LXC in HPC environments. They also experimentally evaluated the trade-off between performance and isolation.

The authors of [24] provided a performance comparison of KVM, XEN hypervisors and Docker on ARM architecture. The experimental results showed a slightly better performance for containers, however it was mentioned that hypervisors provide high isolation given by hardware extensions. In [25] the authors evaluated bare metal, VMs and Docker containers, that can be provisioned in OpenStack, in terms of CPU, networking, memory, disk I/O and boot-up times. Their evaluation showed that Docker hosts are characterized by the fastest boot-time and overall performance, similarly to those of bare metal, with network bandwidth being the only exception.

The authors of [26] used a HTTP proxy scenario to provide a performance analysis of KVM and Docker. They evaluated the total time needed to perform HTTP requests and responses, when the proxy was running on both virtualization technologies and the physical server was under heavy CPU load. They found that a Docker container can be up to twice as fast as a KVM-based VM.

In [27] Docker containers and VMs were deployed on Amazon cloud environment and their performances were analyzed in terms of throughput, response time and CPU utilization. The authors surprisingly found that VM-based web services outperformed container-based web services. As they pointed out, this happened mainly because Amazon cloud ran containers on top of EC2 VMs and not directly on bare metal physical hosts.

Plauth et al. [28] compared the performances of Docker and LXD containers, Rumprun, OSv and MirageOS unikernels, as well as XEN and KVM hypervisors. They recognized that running containers on VM is a common practice in order to run containers on top of IaaS cloud. They evaluated the performances of Nginx and Redis containers for both running on bare metal and on VM. Their results showed that unikernels performed equally or better than containers.

Finally, the performances and performance isolation of containers and VMs were studied in [29]. The authors conducted experiments and measured the performances overhead of KVM hypervisor and LXC containers. LXC achieved higher performance in almost all cases. It is worth mentioning that, in a small part of their experiments, they ran a LXC container on a KVM VM and evaluated its performance.

3. Virtualization Isolation

3.1. Virtualization and Isolation of x86 Architecture

Although big cloud providers examine the adoption of low-power ARM processors, today the x86 architecture is the leading processor architecture used in cloud [30], [31]. In x86 architecture there are three modes, the real, the protected and the virtual 8086 mode [32]. The protected mode is the predominant one because it offers some key advantages compared to the others, for example it supports multitasking. The x86 architecture in protected mode uses four privileged levels or rings, as it can be seen in Fig. 1. The ring 0 is the most privileged level to run instructions within the processor, it can communicate with the hardware and it is used for kernel space. Ring 3 is the least privileged and only used by the OSes for user space. The intermediate rings 1-2 are usually not used by the majority of the OSes.

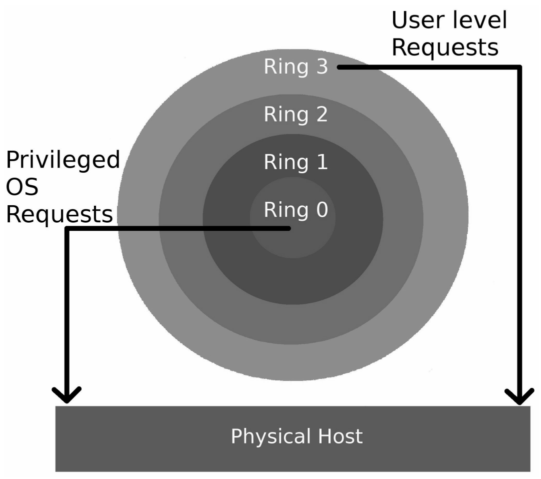


Figure 1: x86 architecture privilege rings.

x86 virtualization was introduced for the first time in 1998, by VMware [33]. They managed to combine binary translation and direct execution techniques on the processor, to run multiple isolated Guest OSes on the same hardware. In this approach, the hypervisor runs in ring 0 and the Guest OS in ring 1 Fig. 2. The hypervisor provides each VM with a complete simulation of the existing hardware, including virtualized memory management and virtual devices. The Guest OS is unmodified and unaware that it is running on a VM. Any privileged and protected CPU operation of the Guest OS is intercepted by the hypervisor, modified and executed in the underlying hardware. This technique is called full virtualization, does not require Guest OS modification, hardware or OS assistance and offers the highest isolation.

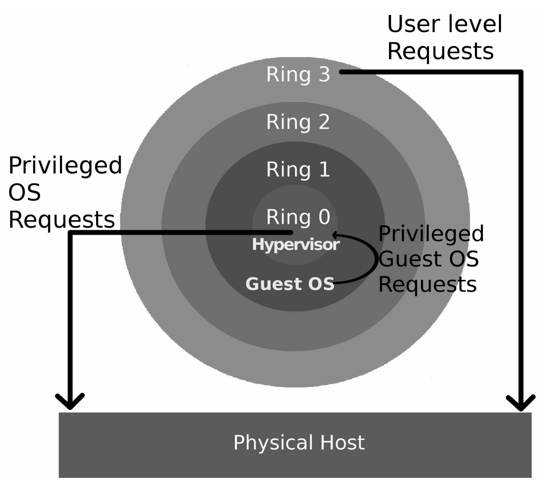


Figure 2: Full virtualization privilege rings.

Another virtualization technique which, in contrast to full virtualization, requires Guest OS modification, is called paravirtualization. In this technique the Guest OS is aware that it is running on a VM and its privileged operations are replaced by calls to the hypervisor (hypercalls), Fig. 3. The Guest OS can also make hypercalls for memory management, interrupt handling and other crucial operations [33]. Paravirtualization has lower overhead compared to full virtualization, due to the lack of binary translations, while the modified Guest OS communicates through hypercalls with the hardware. On the other hand, the Guest OS modification might introduce maintenance and compatibility issues.

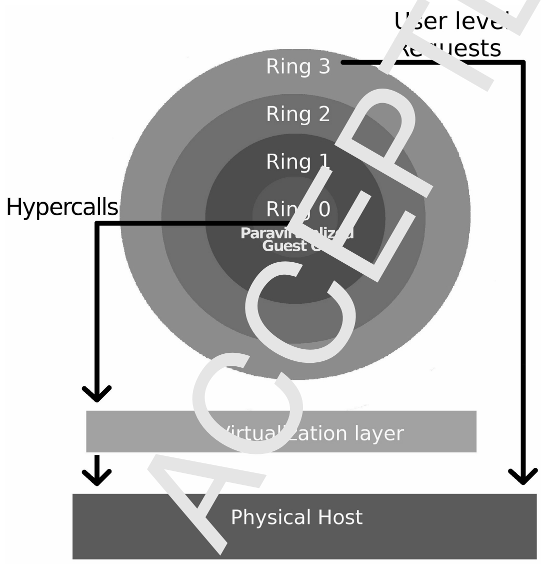


Figure 3: Paravirtualization privilege rings.

Last but not least, there is the hardware assisted virtualization technique. Although hardware assisted virtualization is originated from 1972 (IBM System/370) [34], hardware assisted virtualization for x86 CPUs became available in 2006 with Intel VT and AMD-V technologies. Intel and AMD extended the instruction set of their processors to provide hardware-based virtualization support. Hardware assisted virtualization does not require a modified Guest OS or software emulation and it improves the system's overall performance. Intel VT and AMD-V technologies have many similarities. However, as our system is equipped with Intel processor, we focus on Intel VT. Intel, with its extended instruction set, introduced two Virtual Machine Extension (VMX) modes, the Root Mode and the Non Root Mode (Fig. 4). The Non Root Mode is for the Guest OS and the Root Mode is for the hypervisor, in both modes there are 0-3 rings. The Guest OS is not aware that it is running on Non Root Mode and it does not need any modification. To execute Guest sensitive instructions, CPU stops running on Non Root Mode and switches to VMX Root Mode. This is called VMexit. Next, the hypervisor emulates the Guest's instruction and afterwards the CPU switches again from Root Mode to Non Root Mode. This is called VMentry.

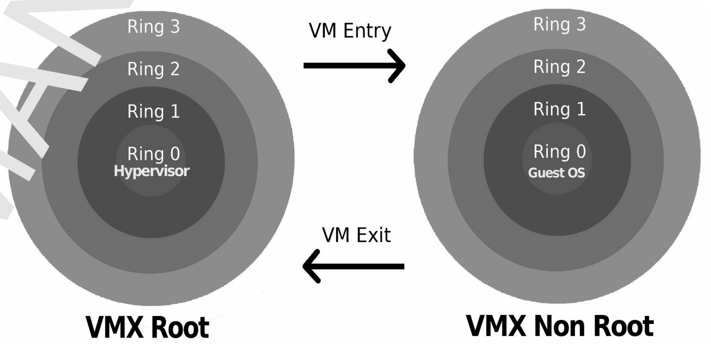


Figure 4: Hardware assisted virtualization privilege rings.

In our study we used KVM and XEN to deploy VMs on Linux Ubuntu Server and Hyper-V which is used by Windows Server. We chose KVM and XEN hypervisors because they are open-source, free and widely adopted. The Hyper-V was used by default from Docker for Windows, to run Linux containers on Windows. It is worth noting that these hypervisors are also used by some of the biggest cloud provider companies. Till the 6th of November 2017 Amazon employed XEN to deploy VMs, then it presented a new EC2 instance based on KVM. Google Compute Engine also uses KVM, while Microsoft Azure uses a customized version of Hyper-V.

We exploited the hardware assisted virtualization capabilities of KVM and XEN, to deploy VMs in both cases. Although XEN and KVM have essential differences, they both support hardware emulation with QEMU and paravirtualization for I/O devices.

XEN hypervisor is based on microkernel architecture and it consists of a privileged VM, which is called Domain-0 or Dom0, and other unprivileged VMs, which are called Domain-U or DomU [35]. Dom0 can directly access the hardware, it provides interfaces for DomU to use I/O devices, controls resource allocation policies, etc. Each DomU communicates with the XEN hypervisor, which is running on top of hardware and serves as an interface between the hardware and the VMs. CPU and main memory accesses are managed directly by the XEN hypervisor, whereas I/O is managed by Dom0 [36]. In hardware assisted virtualization, the XEN hypervisor is running in VMX Root in ring 0, whereas Dom0 is running in Non VMX Root and its kernel is running in ring 0.

Hyper-V comes as a role on Windows Server 2016 or as a stand-alone product. It is based on microkernel design and it requires CPU which supports hardware assisted virtualization [37]. Hyper-V isolates VMs in terms of a partition [38]. A partition is a logical unit of isolation, in which operating systems are running [38]. The virtualization stack is running in a parent partition, which has direct access to the hardware devices. Moreover, there are child partitions which host the Guest OSes. Hyper-V is running in VMX Root in ring 0, whereas root and child partitions are running in Non VMX Root and their kernels are running in ring 0. Hyper-V is used by Windows Server, in order to assist Docker to run Linux containers on Windows OS. Hyper-V creates a VM, with a lightweight OS just enough to run containers, for each running container. With this technique, each container runs isolated from the others, as it is the only one that shares the VM's kernel.

Instead of creating a hypervisor and necessary components like scheduler, memory manager and device drivers, KVM just loads a kernel module into Linux kernel and transforms the host into a hypervisor [39]. Every VM is treated by the host OS as an ordinary process and any virtual CPU is treated by Linux scheduler as a normal thread. KVM uses a modified QEMU to emulate the I/O bus, network interface etc. At the same time it supports paravirtualization with Virtio. Virtio allows the guest's devices to be aware of the host's hypervisor and to communicate directly with the hypervisor [40]. In hardware assisted virtualization, the hypervisor and driver are running in ring 0 of VMX Root mode, because KVM places the hypervisor and driver module in the Linux kernel. To further improve the isolation between VMs, the access control module SELinux can be used.

XEN and Hyper-V provide a greater isolation between the hypervisor, the drivers and the guest VMs compared to KVM, as they follow a microkernel architecture. On the other hand, KVM follows a monolithic architecture and although KVM does not have as strong isolation between drivers and hypervisor, as XEN and Hyper-V do, still it provides a great degree of isolation between the hypervisor and Guest VMs. In addition, the more complex structure of a microkernel-based hypervisor may lead to significant performance overhead and may even negatively affect the system's overall security, as more complicated systems have more lines of code and possibly more bugs.

3.1.1. Memory Virtualization and Isolation

VMs that are deployed on the same physical system share its memory. Even if there is only an unvirtualized OS on a x86 system, its applications will still see a virtual address space. In modern x86 CPUs, hardware maps multiple virtual address spaces into a possibly smaller amount of physical memory [41]. There is a memory management unit (MMU) which translates virtual to physical addresses and a translation lookaside buffer (TLB) which stores the most used parts of the page tables and accelerates the process [32]. In Linux systems there is memory isolation at the process level.

To deploy VMs in x86 architecture, the MMU has to be virtualized. Modern CPUs support mapping guest physical memory to the host physical memory with hardware (AMD NPT and Intel EPT). If the processor does not support these technologies and, as a result, does not have the dedicated hardware for MMU virtualization, the Guest OS maps guest virtual addresses to guest physical addresses. As it is expected, because physical memory is virtualized, the guest physical address is not the actual host physical address and, consequently, the hypervisor is responsible for mapping guest virtual memory to host physical memory. The hypervisor creates guest physical addresses internally, to host physical addresses mapping and it creates shadow page tables which contain guest virtual memory to host physical memory mapping. With this technique, the hypervisor can map guest virtual addresses to physical host addresses and assure that every VM will access only the part of the host's physical memory that it owns [41]. Although hardware TLB is used to cache guest virtual memory to host physical memory address translations and to accelerate the mapping process, extra overhead is introduced when the shadow page needs to be updated. A paravirtualization method can be used in order to improve the memory performance. Paravirtualization requires modification of the Guest OS, in order to be possible for Guest virtual memory to be mapped to host physical memory [42].

In our case, the system's CPU supports Intel EPT technology and is equipped with special purpose hardware for MMU virtualization. In hardware-assisted MMU virtualization, a two level mapping is performed in hardware, as it can be seen in Fig. 5. The first one, from guest virtual memory to guest physical memory, is done by the Guest OS and the second, from guest physical memory to host physical memory addresses, is controlled by the hypervisor and is unseen by the Guest OS [41]. In this way there is no need for the hypervisor to keep shadow page tables, since these two sets of page tables are exposed to the hardware and can be used to find the host physical addresses from the guest virtual addresses. Furthermore, to speed up future guest virtual addresses to host physical address translations, TLB is used. Moreover, the nested page tables of hardware-assisted virtualization are mostly static and are not updated when the VM creates or modifies page tables. Hardware-assisted virtualization reduces memory consumption and has higher performance for most workloads [43]. On the other hand, this technique's performance is heavily affected by workloads that stress the TLB, because the virtual to real physical address translation is a lot more complex, therefore each TLB miss is much more expensive [44]. However, the increased TLB miss cost can usually be overcome by using large pages.

When a VM asks for memory access, the processor will perform a translation using the EPT. A hypervisor can define the memory, which the VM has the right to access, by setting bits in the EPT memory structures [45]. If a VM tries to compromise

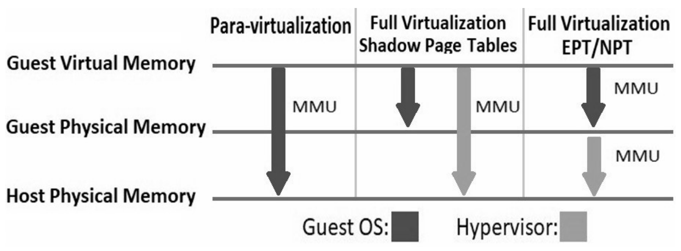


Figure 5: Hardware assisted virtualization privilege rings.

memory isolation and to access part of the memory that is not allowed to, then an EPT violation will occur and the hypervisor will intervene to handle the incident. By analyzing the EPT violations, one might recognize a potential attack or an isolation thread.

3.1.2. I/O Virtualization and Isolation

One of the key functions a hypervisor implements is sharing the system's devices and providing isolated I/O to the deployed VMs. There are three ways to provide I/O virtualization, full virtualization with device emulation, paravirtualization and direct I/O [46].

In full virtualization, real hardware devices will be emulated and the hypervisor will expose their interfaces to the VM. This way the Guest OS will simply use its driver for the emulated device, but as all the device operations will be emulated in software, this will cause a performance degradation. QEMU emulates a set of different existing devices and can be used under XEN and KVM hypervisors. KVM was originally designed to use the full virtualized I/O method. KVM starts a QEMU process in user-mode for each VM and attaches to these specific emulated devices. Every I/O operation from a VM is redirected by the KVM to the corresponding QEMU process.

The paravirtualization method has higher I/O performance compared to full virtualization. In this method, there is a lightweight front-end driver that is running in the VM and managing its I/O requests. Also, there is a back-end driver which is running in the host and actually communicates with the I/O devices. Moreover, this method may require special drivers or tools to function properly, which however are already available in most Linux distributions. Although this method is commonly used in XEN, KVM also supports it by leveraging Virtio. In a common XEN configuration, the back-end driver and the actual device drivers, which manage the devices, are running in Dom0 and the front-end driver is running in DomU. The front-end and back-end driver transfer data to each other via a block of shared memory, called Grant Tables. Because every data transfer is related to a request or response between the front-end and back-end driver, a buffer ring is used to manage the requests or responses. These techniques ensure that each VM will exclusively have access to its devices.

Finally, in direct I/O method, the VM can access the hardware device directly. As it would be expected, in this way we can achieve maximum performance, as the hypervisor does not intervene into the data-path. However, because the hypervisor is not involved in the data-path, useful functions such as exposure of file-image, virtual disk, network tunneling or live migration will no longer be possible.

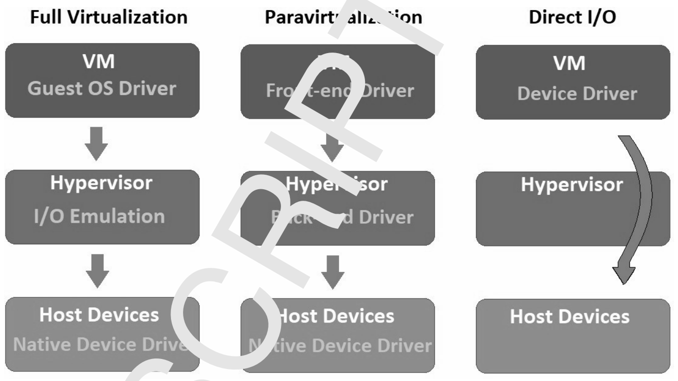


Figure 6: I/O virtualization methods.

In all the above methods (Fig. 6) the isolation and the restricted access to devices is a very important factor. Intel and AMD, in order to support the isolation and restricted access to virtualized devices with hardware, introduced the VT-d and AMD-Vi virtualization technologies respectively. In this work we focus on Intel's VT-d technology. The key capabilities of VT-d technology are I/O device assignment, direct memory access (DMA) remapping, interrupt remapping and reliability features [47].

The DMA remapping enhances isolation by restricting DMA of the devices to pre-assigned domains or physical memory regions [48]. When an I/O device asks for access to a certain memory location, DMA remapping hardware checks for its permissions to access that physical address. If the device has the corresponding right, it will access the memory, otherwise it will be blocked. To boost performances, frequently used remapping-structure entries are cached. In the direct assignment of I/O device, the hypervisor does not participate in every I/O request but only in requests for protected resources [48]. The DMA remapping hardware is used to convert the guest physical addresses, which the Guest OS provide, to host physical addresses. To further improve performance by reducing hypervisor overhead, interrupt remapping control can be directly assigned to VM. Intel VT-d can improve reliability and security of hypervisors by isolating different devices and also set access to specific memory ranges for each of them.

3.1.3. Containers and Isolation

Containers offer a logical packaging mechanism, which allows applications to be easily and consistently deployed in different environments, from personal computer to public cloud [49]. Containers do not run an instance of a complete OS, as VMs do, but they run multiple isolated processes in the same host [50]. They access the same OS, including root file system, libraries and common files. Mainly because the virtual hardware layer is missing, containers can be created significantly faster than VMs and introduce less overhead compared to VMs [21].

A container engine handles the container image, which maintains the necessary dependencies and binaries for a container to be executed. To control resource consumption and to organize processes, containers use Linux Control Groups (cgroups). With cgroups, it is possible to limit the CPU and memory consumption per container and resize or even terminate it. To isolate containers with no visibility or access to external objects, the namespace feature of the kernel is applied.

In our study, we focus on Docker as an open source software container platform [51], which has recently grown very popular. The main feature of Docker, that is not present in most other container platforms, is that it uses layered file system images, such as AUFS (Advanced Multi Layered Unification Filesystem) [52], [53]. This feature allows different containers to use the same basis layer repeatedly, boosting performance, saving network bandwidth and space in host's storage unit. Docker has great compatibility and maintainability, it is supported by many cloud platforms, like Amazon Web Services and Google Compute Platform and additionally, Docker dominates other container technologies in bibliography, [4], [6], [8], [21], [22], [24], [54], [55], [56], [57] and [58].

However, containers do not excel at providing isolation as VMs do [59]. The Docker Engine runs in user space (ring 3) as a root application, whereas namespaces and cgroups are applied by the Linux kernel in ring 0. Because all running containers share the same kernel, a compromised kernel leads to compromised containers. Moreover as containers run on the same level (ring 3) with other applications, any of them can disrupt their operation. To enhance security, a container can run on top of a VM. Thus, it can achieve multi-tenant isolation, having the VM as boundary of security, yet with the corresponding performance cost [60]. In addition, by running containers on VMs, supplementary security tools can be used by the host to monitor VMs and intercept malicious events [61],[62], [63].

4. Performance Evaluation

In this section, we experimentally quantify the performance overhead that was introduced by running Docker containers on KVM, XEN and Hyper-V VMs. We deployed a series of benchmarks to measure how the additional layer of the VM affects the CPU, memory, disk and network performances. We performed each experiment twenty times and we calculated mean values and standard deviation. At first, we deployed containers on bare metal and used these measurements as a basis. Secondly, we executed the same benchmarks, but this time containers were running on top of VM (Fig. 7).

All the experiments are executed on a 3.7 GHz Intel CORE I3-6100 processor with 8GB of DDR4 DIMM RAM and 100 Mbps Ethernet. The OSes which are installed on physical node and VMs are Alpine 3.6.2, Atomic Host based on CentOS 7, CargOS 2016.08, CoreOS 1465.8.0 and RancherOs v1.1.0. In all cases where VMs were deployed, we used Ubuntu Server 16.04.3 LTS as host. In order to examine the overhead of Linux Docker containers on Windows, we used Windows Server 2016 as host.

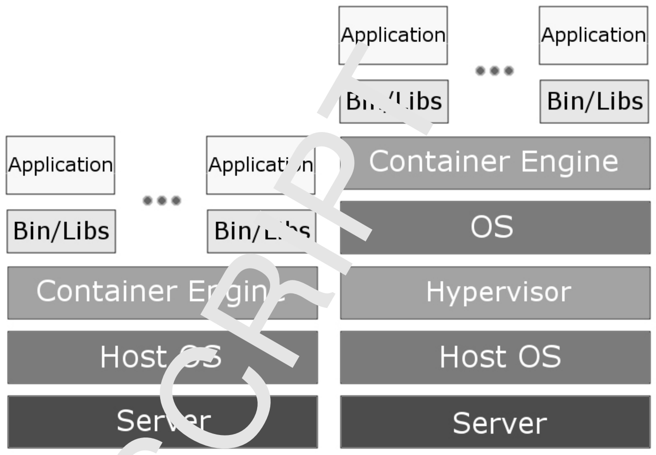


Figure 7: Containers and containers on VM.

Each VM that we created was configured to use all the available resources of the physical node. The Docker containers were created without any cgroups restrictions, in order for them to also use all available resources.

4.1. Container-Centric Operating Systems

As Docker's popularity and adoption is growing rapidly, in the last years some new Linux distributions, optimized to run containers and mainly Docker, were released. As deploying a full-fledged Linux distribution on a host that will only run Docker appears to be exaggerated, these container-centric OSes are minimalistic. The idea of minimalistic OSes is not new, but it finds a new appliance in this field.

Many of these OSes besides being characterized by small size, small overhead and small attack surface, have additional features that support the deployment of Docker containers. Some of these OSes run Docker daemon by default, have automatic updates to deal with inconsistencies and can update roll back in case of unsuccessful OS update. In our study, we deployed Docker containers on the free container-centric CoreOS [64], RancherOS [65], Atomic Host [66], CargOS [67] and the minimalistic Alpine OS [68].

CoreOS is an open source container-centric OS and it is also part of a commercial product called Tectonic [64]. CoreOS automatically configures the container's runtime on each CoreOS machine, updates itself and detects every new Docker container which will be connected to the network. RancherOS is actually consisted of just the kernel and Docker and it runs all system services as Docker containers. Atomic Host is Red Hat's container-centric minimalistic OS, it has built-in support for SELinux and it has replaced the yum package manager with the rpm-ostree package manager, something that allows software roll-back. CargOS [67] provides only the essentials to deploy Docker and it executes any other service as container. Alpine [68] was not originally designed to host Docker, but it is a secure and efficient OS which can be used to deploy Docker containers on it.

4.2. Benchmarks

We employed four different benchmarks to individually measure the performances of the four main server components. We used LINPACK [69] for CPU performance overhead measurements, STREAM [70] to measure memory performance, IOZONE [71] for block devices and file systems and NETPERF [72] to measure network performance.

LINPACK is a collection of Fortran subroutines which solve various systems of linear equations and measure the floating point operations per second (flops) [69]. The main LINPACK function solves a system of linear equations "Ax = b" (random matrix A, vector b) by performing lower upper decomposition of numerical analysis with partial pivoting.

STREAM is a simple synthetic benchmark which measures sustainable memory bandwidth by performing simple operations on vectors [70]. It executes four different operations, Copy, Scale, Add and Triad, which are presented in Table 1. Although STREAM recognizes a strong relation between the evaluated throughput and the size of the CPU cache, it measures the main memory bandwidth and not the cache bandwidth. According to STREAM's general rule, each array has to be at least four times the accessible cache memory size.

Table 1: STREAM operations.

Name	Kernel
COPY	a(i) = b(i)
SCALE	a(i) = q*b(i)
SUM	a(i) = b(i) + c(i)
TRIAD	a(i) = b(i) + q*c(i)

We used IOZONE [71] to measure the performances of various disk operations, such as Read, Write, Re-Read, Re-Write, Reverse Read, Stride Read, Random Read, Random Write, Fwrite and Fread. We executed this benchmark with 100 MB file size, originally on a container running on bare metal and then on a container running on top of a VM for each of the six OSes.

Finally, we took unidirectional throughput and end-to-end latency measurements with NETPERF [72]. Because NETPERF is based on the client-server model, we hosted the NETPERF server in an additional node and the NETPERF client in our initial testing node. Both machines were connected with 100 Megabit Ethernet Link through a ZTE H108NS router.

4.2.1. CPU

First, we measured CPU performance. We ran LINPACK on Docker container on bare metal and LINPACK on Docker container on top of VMs. We executed a set of experiments on six different OSes with LINPACK array sizes of 200x200, 2000x2000 and 8000x8000. For this type of experiments we used three different hypervisors and we defined the number of vCPUs (virtual processors) same as the number of logical cores of the physical machine. In case of XEN, one of the four total vCPUs was occupied by Dom0.

Fig. 8 shows the performance measurements of the LINPACK benchmark for various system configurations. In Fig. 8 we observe that, on the one hand, CPU achieved the best performance for the smallest array (200x200). In this case (of the smallest array), we notice that our measurements show also the highest standard deviation. This probably happens because the results are affected by the existing high speed memory. On the other hand, for bigger arrays of 2000x2000 and 8000x8000, which are not affected so much by the cached data on the high speed memory, we took lower but more stable values.

In general, from Fig. 8 we can see that when containers ran on bare metal or on XEN VM, they performed almost the same. Windows Hyper-V and KVM VMs follow shortly after them. KVM appeared to have slightly worse CPU performance compared to the others. This probably happens because KVM is not Type-1 hypervisor like the other two, it runs at a different level and is treated by the host OS as an ordinary process.

If we closely examine the case of 8000x8000 array, where we took the most stable measurements, the most significant difference is in KVM and bare metal CPU performance. For the biggest array, bare metal was 4,89% faster than KVM, and 11.12% in average for all different array sizes. Regarding OSes, Alpine achieved the best CPU performance in 77% of our cases, mostly with a relatively small difference of 3% from the others. It is worth noting that, whereas on bare metal CoreOS was the fastest, it was slower than Alpine, when running on a VM. This probably happens because the VM hides the nature of system information from the execution environment [5].

4.2.2. Memory

We evaluated the memory performance with STREAM benchmark. The available cache memory of our system (L3 cache) was 3 MB. Following the STREAM's rule, we set much larger arrays of about 2.3 GB and total memory required at 7 GB. As illustrated in Fig. 9 the Copy function, which is the simplest of the four different functions that STREAM uses (Table 1), achieved the highest performance for all different system configurations. By observing Fig. 9, we can also easily recognize that there is not a big difference in memory performance for Docker containers which run either on bare metal or on top of a VM.

Our memory performance measurements, for all four functions on all OSes, show that, in average, bare metal achieved 3.74% higher memory performance compared to KVM, 1.63% compared to XEN and 1.95% higher memory performance compared to Hyper-V. The different type of KVM hypervisor seems to affect the memory performance and the memory management services of the host. The six OSes achieved almost the same memory performance. Nevertheless, Alpine was the one showing slightly higher memory performance on 66% of our cases. CoreOS, similarly to CPU measurements, is persistently faster on bare metal.

4.2.3. Disk

We evaluated the disk performance with the IOzone benchmark and we configured it to generate files of 100MB. We ran it first on containers on bare metal and secondly on containers on top of a VM. We measured various operations such as Read, Write, Re-Read, Re-Write, Reverse Read, Stride Read,

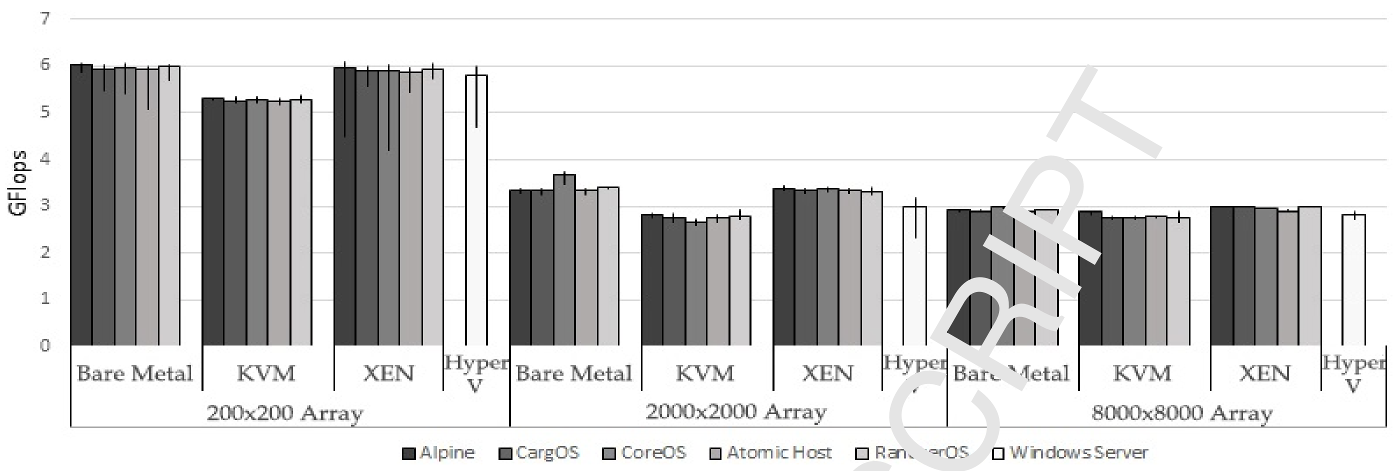


Figure 8: CPU performance.

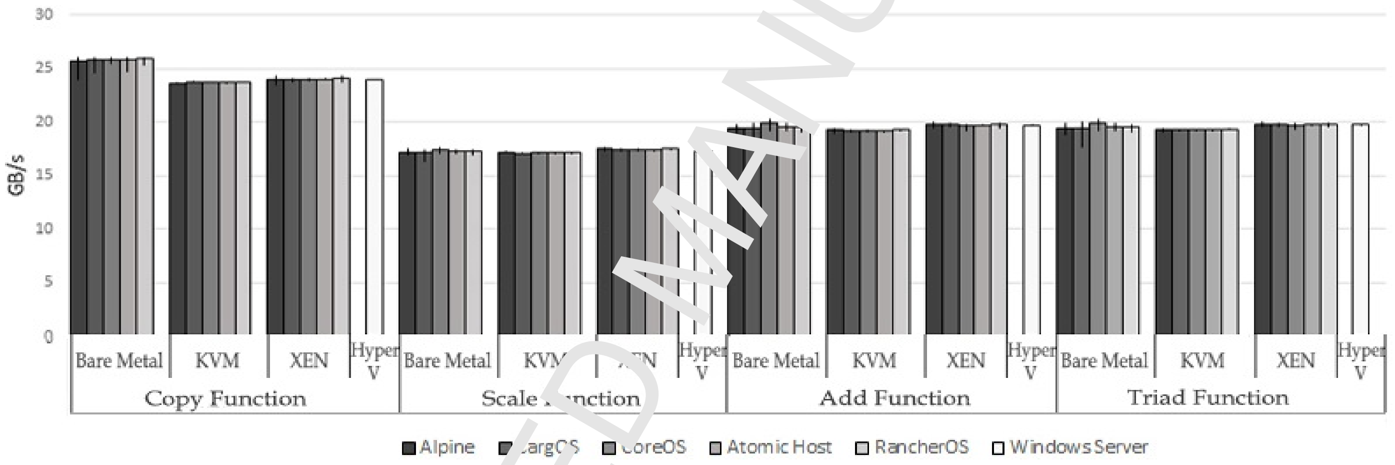


Figure 9: Memory performance.

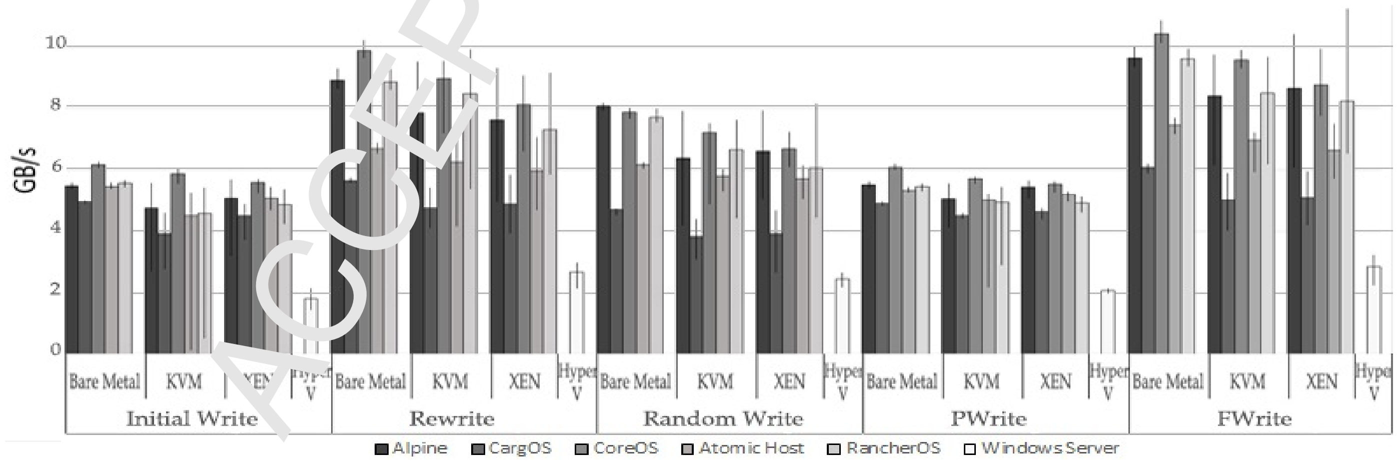


Figure 10: Disk write performance.

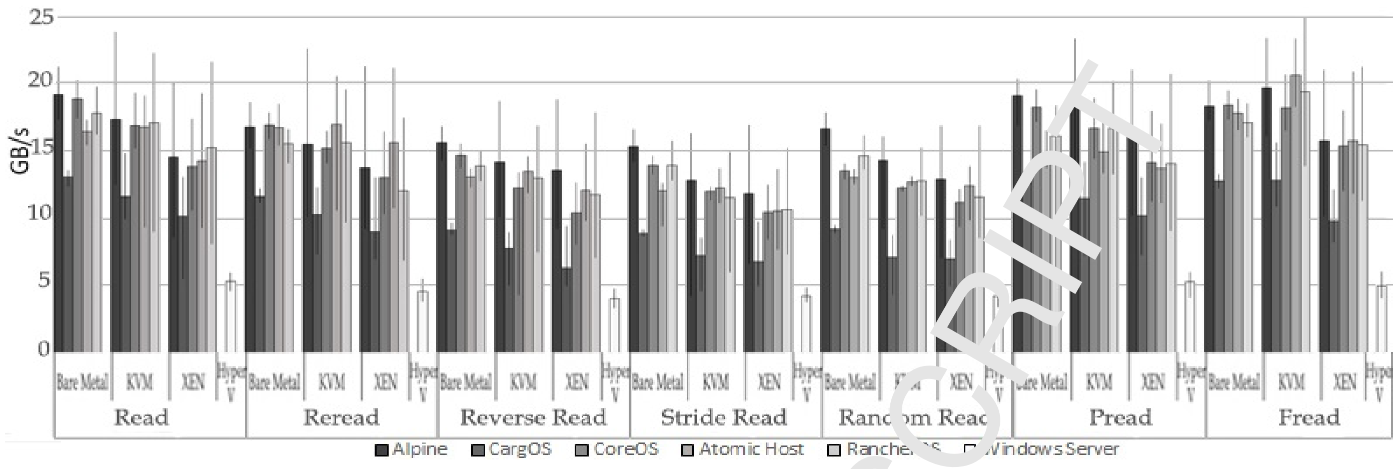


Figure 11: Disk read performance.

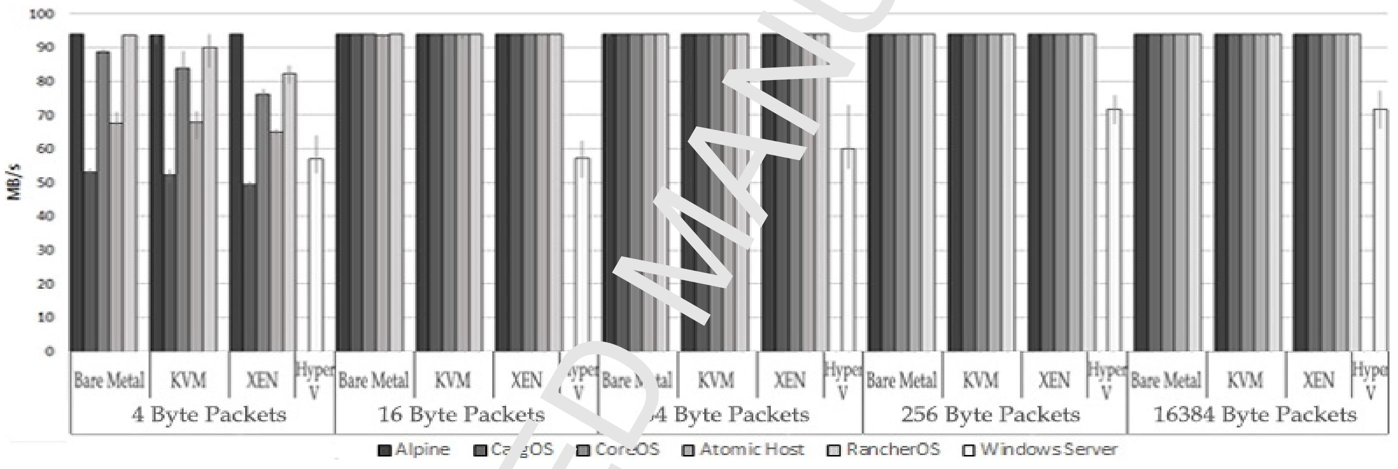


Figure 12: TCP_STREAM network performance.

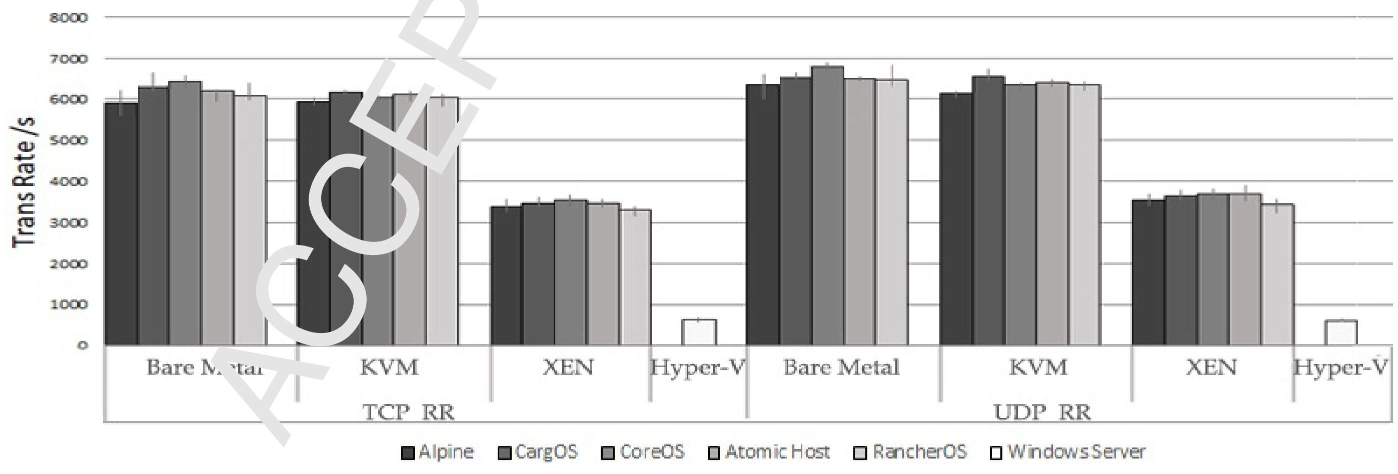


Figure 13: Request-Response network performance.

Random Read, Random Write, Fwrite and Fread. For these series of experiments, although we deployed full virtualized VMs, we used paravirtualized drivers for disk and network devices to boost the performances of KVM and XEN VMs.

The experimental results of write to disk functions can be seen in Fig. 10 and read from disk functions in Fig. 11. As it was expected also in this case the bare metal achieved the highest overall performance.

Fig. 10 shows a significant difference in the performances of write to disk operations between Windows Hyper-V and the other hypervisors. Generally, bare metal achieved the highest performance among all write operations, followed, with a short distance, by the KVM and XEN hypervisors. The performance gap between Windows Hyper-V and the other hypervisors is up to twice as bad as the lowest performance measurement. This probably happens because the files are stored on the Windows NTFS file-system and some translations may be needed [73]. According to Microsoft, incompatibilities for some applications may occur because some file-system operations are currently partially or even not at all implemented [73].

The performance measurements of read from disk functions are presented in Fig. 11. As well in this case, for almost all read operations, the performance of bare metal was the highest one, followed by KVM and XEN. Windows Hyper-V comes last with significantly lower performance. XEN disk performance measurements were slightly lower than KVM in almost all cases. This probably happens because only Dom0 has direct access to the disks when any other DomU can access the disks through Dom0.

4.2.4. Network

With NETPERF, we measured the performance of TCP_Stream, TCP RR (request-response) and UDP RR. We recorded the throughput of transmitting TCP packets for packet sizes of 4, 16, 64, 256 and the default 16384 Bytes. In the cases of KVM and XEN hypervisors, network paravirtualized drivers were used. As illustrated in Fig. 12, our measurements for TCP_Stream on bare metal, XEN and KVM are almost the same for all different OSes. The only exception is the case of the 4 byte packet (which is the smallest one), where caching and the much higher number of transmitting packets probably heavily affected the results. As it can be seen in Fig. 12 and Fig. 13, containers which ran on Windows Hyper-V showed the lowest network performance in almost all cases. This probably happens because of the differences on network stack of Windows and Linux-based OSes.

The TCP RR and UDP RR performance measurements are shown in Fig. 13. As we can see, for both TCP RR and UDP RR measurements, bare metal and KVM achieved almost the same performances, followed by XEN and finally by Hyper-V. XEN achieved lower performance compared to KVM, probably because every DomU must use the network driver that is located in Dom0 [12]. Also in this case, there was a significant difference of network performances of the Docker containers that ran on top of Windows Hyper-V and the Docker containers that ran on the other VMs.

4.3. Benchmarks Results

Containers that ran on KVM had the highest CPU performance overhead. In average, KVM introduced 13.02% higher CPU overhead compared to XEN and 7.4% higher CPU overhead compared to Hyper-V. Concerning container-centric OSes, almost all of them achieved the same CPU performance. Alpine achieved the highest one and Atomic Host the lowest one by 1,72% compared to Alpine.

We recorded similar results for the memory overhead. All OSes achieved almost the same memory performance with less than 0.6% difference between them. Container on bare metal reached 1,63% higher memory performance compared to XEN, 1,95% higher performance compared to Hyper-V and 3,74% higher performance compared to KVM.

In disk performance we observed high deviation in the recorded values. The highest GB/s achieved by Alpine and the lowest by CargOS, by 68.93% less compared to Alpine. Hyper-V on Windows had by far the lowest performance, which was 3.5 times lower compared to Alpine. KVM had in average 2,63% lower disk performance than bare metal and 16,73% higher disk performance compared to XEN.

The network benchmark results for TCP_Stream show that Alpine was the most performant one and the least was CargOS, by 9.9% compared to Alpine. KVM and XEN performances were almost the same as bare metal's network performance, whereas Hyper-V achieved 43.36% lower network performance compared to bare metal.

From these measurements we can conclude that, for different kinds of applications there are different optimal solutions. For CPU intensive applications, the combination with the highest performance is Alpine and XEN. For memory intensive applications, the best choice would be CoreOS and XEN. Lastly, for disk and network intensive applications, Alpine and KVM can be combined, in order to achieve higher performance.

5. Use Cases - Further study

In the previous section, as in the first stage of our analysis, we ran containers on bare metal and containers on top of VMs. We separately evaluated how the performance of CPU, main memory, disk and network are affected by the type of hypervisor and OS. In the following section, we used a case study approach to further our research and enhance our understanding on how different parameters affect the overall performance of containers running on top of VMs. In this set of our analysis, we explored how the number of VMs, the number of containers per VM, the number of threads, the type of processing and the mechanism of storing data, affect the overall performance of containers.

By combining the experimental results obtained so far, we observed that Alpine, CoreOS and RancherOS show the highest CPU performance. Although there is no big difference in CPU measurements, Alpine is the fastest one in seven of total nine Linux OS cases. Regarding the main memory, Alpine, CoreOS and RancherOS show the highest performance on bare metal, KVM and XEN. Alpine is the fastest one in six of nine cases.

Finally, disk and network experiments revealed that Alpine, CoreOS and RancherOS also show the highest disk and network performance. By taking these findings into consideration and having as unique selection criterion the overall mean performance, we adopted Alpine OS in all use cases.

It is important to note that containers were initially designed for stateless applications and ephemeral data. By default a container only has access to its own file-system and loses all of its data when it is terminated or crashed. For some applications this design may seem ideal, but for applications in which persistent storage is essential, like databases, it is totally unsuitable. Docker provides mechanisms to separate the data from containers and to store persistent data.

Because databases are common cloud application and in order to examine if Docker successfully deals with persistent data, we deployed two database applications on containers and we studied them as use cases. We deployed MySQL as a relational database and MongoDB as a noSQL database example. Before proceeding to use cases, it will be helpful to present how Docker containers store data.

5.1. *Docker Data Storage*

Docker uses union file-system architecture for images and containers. A union file-system represents a logical file-system by grouping directories and file-systems in branches. In Docker each branch is a layer and a Docker image is built up from different layers. Each layer, except for the very last one, is read-only. The last layer is initially empty but later it stores all the changes that are made to the file-system. Because each container has its own writable layer, different containers can share the same underlying image. In this way, Docker avoids duplication of data and optimizes disk space usage. When a container is removed, actually the top layer is deleted, leaving the underlying layers unaffected.

The writable layer is designed to store a small amount of data and to promote building and sharing images. Although it is possible to store data in this layer, it is not recommended to store persistent or shared data, mainly, because of three reasons. First, data in the writable layer cannot be easily moved out of the container. Second, the data will be lost if the container stops running and third, it is not optimal concerning the container's performance, because a storage driver is required to manage the file system. Storage drivers are based on copy-on-write technique which boosts the performances of read intensive applications, but it adversely affects the performances of heavy writing workloads. As a result extra overhead is added in comparison to storing data directly in the host's file-system.

Docker provides a mechanism which is designed for persistent data, it bypasses the container's union file system and is called "data volume". Data volumes are not controlled by the storage driver and, as a result, operate at host speeds. A Docker data volume is a directory or file that is mounted into a container but stored in the Docker host's file-system [74]. The data is stored outside of the container and will be still available if a container stops running or if it is removed.

There are some different options for data volumes and persistent data storage techniques. The first one is to create a data volume along with the creation of a Docker container or at any time with the relevant command. Although the data volume is stored in Docker host directory, it is created and controlled by the Docker daemon, it is isolated from the host machine and it is only managed by Docker. Usually data volumes are created in the directory /var/lib/docker/volumes/, as illustrated in Fig. 14 for Container 2 [75]. In order to be easily manageable, from Docker version 1.9, data volumes can have names and containers which are associated with specific data volume can be listed.

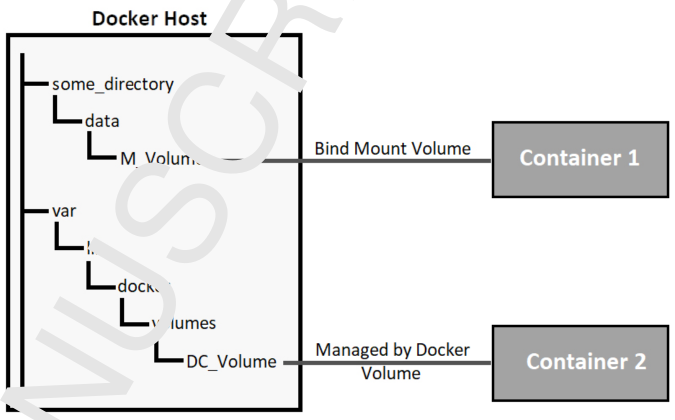


Figure 14: Docker mounted volume and managed by Docker data volume.

The second option is to create a container whose only purpose will be to own data volume. Its data will be accessible by other containers even if this container stops running. This kind of containers can be used for backup purposes or with the purpose to provide other containers with simultaneous access to its data. The third option is to mount a local host file or directory into a container. The source directory or file may be any file of the host as shown in Fig. 13 for Container 1. In this case extra caution is needed because any modification of the host's files might affect its operation. Last but not least, there are third-party volume plugins that allow containers to be integrated in external storage systems and extend its capabilities. For example there are plugins for Azure File Storage [76] and Google Cloud Engine [77].

5.2. *Use Case 1 MySQL*

MySQL is a widely used open-source relational database management system (RDBMS) [78]. With Docker Hub repositories, it is easy to create, share or download and run container images, such as MySQL, in a few seconds. We used the official MySQL repository to deploy our own MySQL Docker containers and we took various measurements for different configurations.

We used the Sysbench [79] benchmark to measure the performance of MySQL 5.7.20 database. The Sysbench can be used to separately evaluate different system components. Furthermore, it provides an OLTP (online transaction processing) component that is frequently used to measure the performances of databases. We created with Sysbench a database of twenty million rows, of about 5GB total size of random data, to run our experiments.

We stored the persistent data which Sysbench created in three different ways. First, we saved the data in data volumes by exploiting Docker's internal volume management. Second, by mounting a host's directory to a container. Finally, by creating a dedicated data volume container. We used InnoDB as storage engine and took performance measurements for various InnoDB buffer sizes. Moreover, we modified the number of threads from 1 to 150, the number of containers from 1 to 6 and the number of VMs from 1 to 3.

In the default mode, Sysbench executes read and write operations to the database. It selects queries from five SELECT queries, two UPDATE queries, a DELETE query and an INSERT query. As we noticed in our results, the ratio was about 65% reads, 25% writes and 10% other queries.

5.2.1. MySQL Docker Containers on Bare Metal

Fig. 15 shows the operation throughput as a function of the number of threads for different buffer sizes. Initially, we ran this set of experiments with default buffer size of 125MB. As Fig. 15 presents, for the default buffer size, the throughput was increased up to 60 threads and then the system was saturated. Thus, the throughput was decreased. To explore how the buffer size affects the overall performance and in order to achieve better utilization of the available resources, we increased the buffer size by several GigaBytes. For sizes of 2, 4 and 6 GB the overall performance was almost identical. We observed that the available memory can successfully deal with the increasing number of threads and, as a result, the operation throughput was steadily increased. The bigger buffer led to a significantly higher overall performance by reducing the disk I/O.

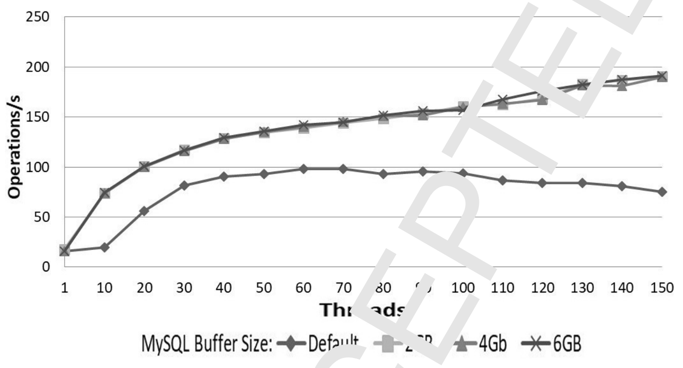


Figure 15: MySQL operations throughput on bare metal.

Table 2 presents the operation throughput for three different mechanisms which provide persistent storage. We configured our system to use 2GB of InnoDB buffer and we used Sysbench to measure the operation throughput for 50 threads. First, we employed Docker's internal volume management system to store data in data volumes. Second, we mounted a host's directory to a container and finally, we created a dedicated data volume container. It is worth noting that the dedicated data volume container was not even running during the experiments, therefore we used an additional container to access and manage its data. After the system warm-up, we ran Sysbench experiments 40 times and the experimental results are presented in Table 2. Using host directory resulted to 0.54% more operations per second compared to using data volume container and 6.84% more operations per second compared to using Docker volume. Even though these tests did not show any significant difference among the different mechanisms of storing persistent data, host directory seems to be the most performant method to adopt.

Table 2: MySQL operations per second on bare metal for different mechanisms of persistent storage

	Docker Volume	Host Directory	Data Volume Container
Bare-Metal (Operations/s)	128.95	137.77	137.03

Another factor which affects the operation throughput is the number of containers running parallelly on the same system. To investigate this factor, we simultaneously ran from 1 to 6 identical containers on our system and we observed how the overall performance was scaling. As illustrated in Fig. 16, the higher the number of running containers the more increased is the overall throughput and the more decreased is the individual container's throughput. We can see that, in average, as a new container was added, the overall performance was increased by 13.81% while the individual performance was decreased by 19.54%. We can also observe that containers running in parallel achieved almost the same performances, which means that there is a high level of fairness and great resource management. However, we cannot increase our system performance endlessly by adding more containers, due to saturation and performance bottlenecks.

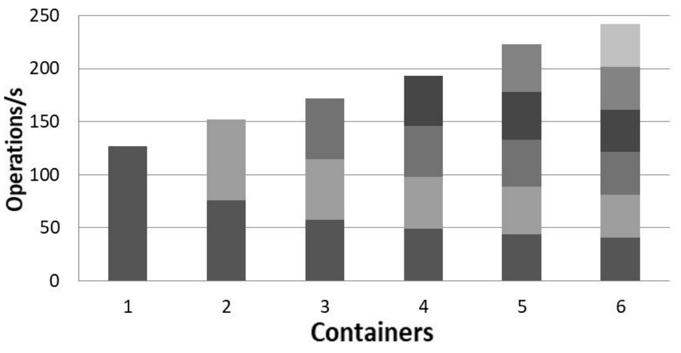


Figure 16: MySQL operations throughput for 1-6 containers on bare metal.

5.2.2. MySQL Docker Containers on VMs

We performed a similar series of experiments to investigate how the same factors, which we previously examined on bare metal, affect the performances of containers running on top of VMs. We deployed Docker containers on KVM and XEN VMs and we configured VMs to utilize all the available resources.

The results obtained from the Sysbench experiments for different buffer sizes and number of threads are presented in Fig. 17 and Fig. 18. From these two figures, we can observe that the two hypervisors show similar performances. For the default buffer size of 125MB, for both hypervisors, the throughput was increased up to 70 threads and then, due to saturation, the operation throughput was decreased. For buffer sizes of 2, 4 and 6 GB, we also observed an identical behavior as the significantly bigger buffer reduced the disk I/O and led to better performance.

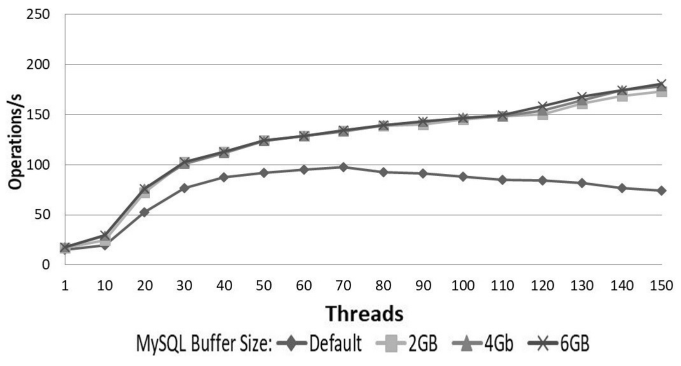


Figure 17: MySQL operations throughput on KVM VM.

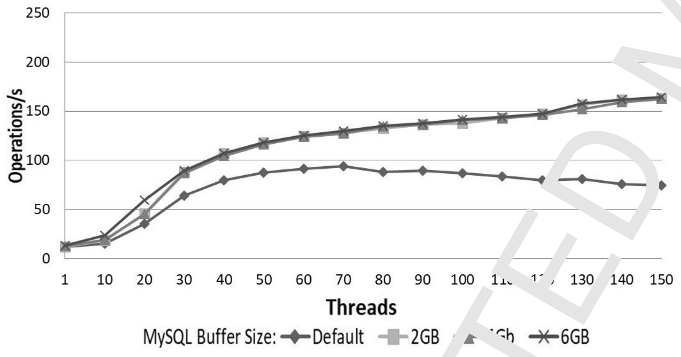


Figure 18: MySQL operations throughput on XEN VM.

To determine how the additional layer of hardware virtualization affects the performances of different mechanisms that provide persistent storage, we replicated the experiments that we previously executed on bare metal, this time on KVM and XEN VMs. As well in this case, we configured our system to use 2GB of InnoDB buffer and we used Sysbench to measure the operation throughput for 50 threads. Table 3 compares the mean operation throughput of containers which were running on top of KVM and XEN VMs and employed three different mechanisms to store persistent data. One can see on the table below that KVM achieved 4.83% better mean operation throughput, in all cases, compared to containers running on XEN. Host directory achieved in average, for both KVM and XEN 3.84% higher operation throughput compared to Docker volume and 0.08% higher operation throughput compared to data volume container.

Table 3: MySQL operations per second on KVM and XEN VMs for different mechanisms of persistent storage.

	Docker Volume	Host Directory	Data Volume Container
KVM (Operations / s)	119.47	123.92	123.83
XEN (Operations / s)	113.8	118.3	118.2

We also performed a number of MySQL tests to investigate the impact of the amount of VMs on total operation throughput. We progressively deployed from 1 to 6 containers on 1 to 3 VMs. We configured the VMs to use all available resources in every case and we deployed separately KVM and XEN VMs. We also set the InnoDB buffer of MySQL to 2GB and Sysbench to measure the operation throughput for 50 threads.

In the case where there were two VMs deployed, when there was one container, this was running on one VM, while the second VM was running as well, without a container. In the case where two containers were running, each container was running on each VM. For three running containers, two containers were deployed on the first VM and the third container was running on the second VM. For four running containers, on each VM two containers were deployed. For five running containers, three containers were running on the first VM and two containers were running on the second VM. Last, for six running containers, on each VM three containers were deployed. We followed the same method to distribute the containers on three VMs.

The results obtained from KVM experiments are presented in Fig. 19 - 21. Fig. 19 shows how the overall system throughput was increased, as the number of running containers on a single KVM VM was increased as well. Fig. 20 and Fig. 21 present the results of the same experiments, but this time the containers were running on two and three similar VMs respectively. As it would be expected, for the various amount of VMs, the higher number of containers led to better overall performance. However, by observing Fig. 19 - 21, one can see that each additional VM added overhead to the system, especially when the number of containers was higher. The same containers which were running on one VM, achieved 3.1% better operation throughput compared to those running on two VMs and 5% better operation throughput compared to those running on three VMs.

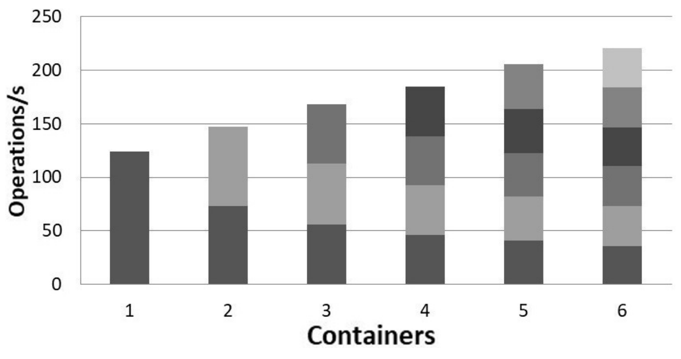


Figure 19: MySQL operations throughput on 1 KVM VM for 1-6 containers.

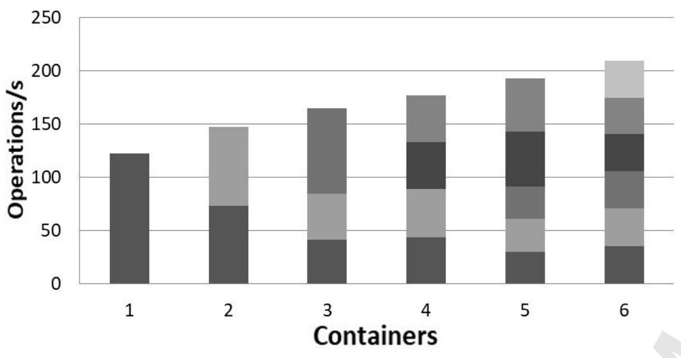


Figure 20: MySQL operations throughput on 2 KVM VMs for 1-6 containers.

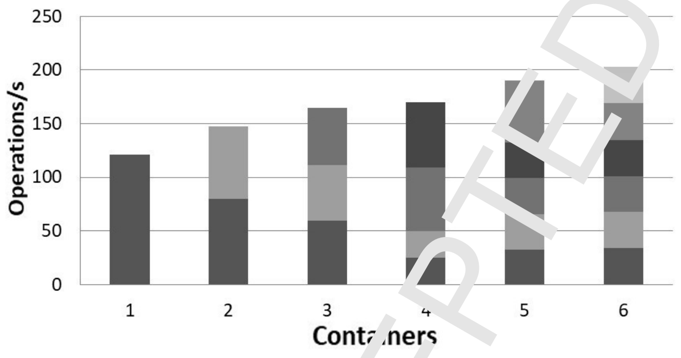


Figure 21: MySQL operations throughput on 3 KVM VMs for 1-6 containers.

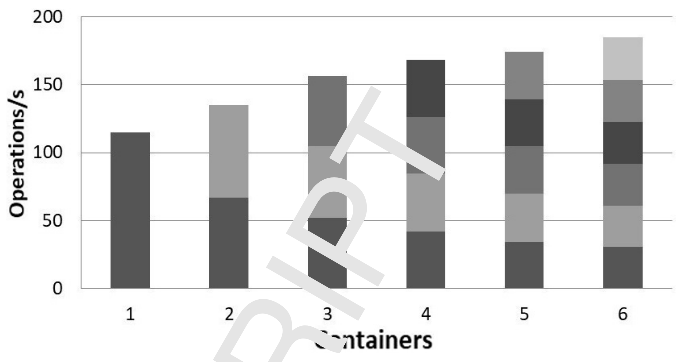


Figure 22: MySQL operations throughput on 1 XEN VM for 1-6 containers.

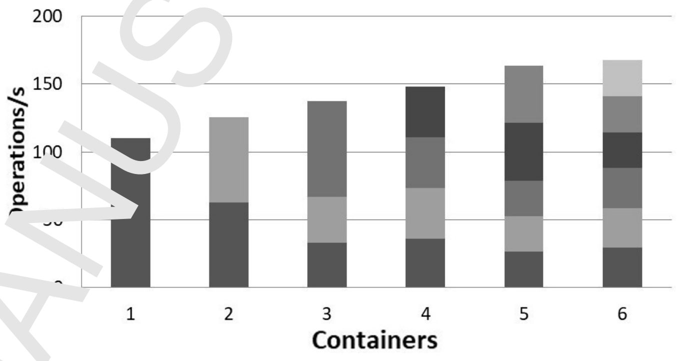


Figure 23: MySQL operations throughput on 2 XEN VMs for 1-6 containers.

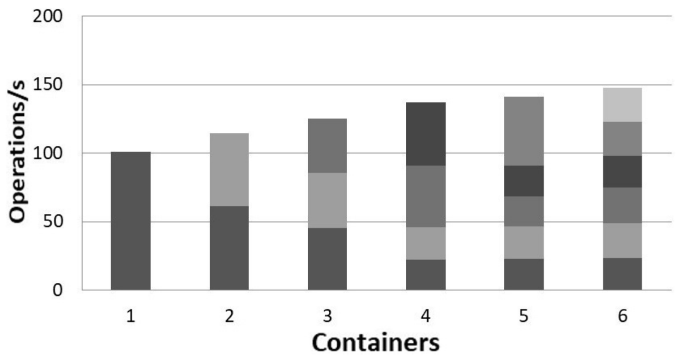


Figure 24: MySQL operations throughput on 3 XEN VMs for 1-6 containers.

By using Sysbench, we also investigated how additional XEN VMs affect the overall operation throughput of containers. In Fig. 22 - 24 we see the total system throughput and the individual container operation throughput for different numbers of containers and VMs. Also in this case, the increasing number of containers resulted to higher operation throughput. For XEN VMs, any additional VM led to higher overhead. The same containers that were running on one VM achieved 9.39% better operation throughput compared to those running on two VMs and 21.44% better operation throughput compared to those running on three VMs.

Running containers on a higher number of VMs resulted to lower performance throughput for MySQL services on both KVM and XEN hypervisors. This happens because every additional VM consumes further resources in order to run its own operating system and services. For this type of service we found that KVM achieved overall 19.9% higher performance compared to XEN.

5.3. *Use Case 2 MongoDB*

A new set of extensive evaluation experiments was performed on one of the most commonly used NoSQL databases, MongoDB [80]. In order to provide a more comprehensive view of containers running on top of VMs, we deployed MongoDB services on containers which were running either on bare metal or on top of KVM and XEN VMs. We evaluated container performances for different types of workloads, mechanisms of persistent storage, number of VMs, containers and threads, by exploiting the Yahoo! Cloud Serving Benchmark (YCSB) [81].

MongoDB is document-oriented and stores data as binary-encoded JSON documents, called BSON. Contrary to SQL databases, where the table schema must be declared before inserting data, MongoDB uses an equivalent to SQL tables that is called ”collections”, where the table schema does not have to be declared in advance. Documents within a collection might have different fields. In our evaluation, containers were running MongoDB v3.4.10 and WiredTiger storage engine.

Firstly, we exploited YCSB to enrich the database with almost 5GB of randomly generated records. Then we used four of the predefined types of workloads, which can emulate different application types, to test our system. The selected YCSB workloads are presented on Table 4.

Table 4: YCSB Workloads.

Workload	Operation ratio
A	Read/update: 50/50
B	Read/update: 95/5
C	Read/update: 100/0
F	Read/read-modify-write: 50/50

5.3.1. MongoDB Docker Containers on Bare Metal

To provide an insight on how the system performs, as the number of threads increases, we took measurements for all four workloads from 1 to 150 threads. The operation throughput of every workload is presented in Fig. 25. As it would be expected, regardless the number of threads, we see that workloads with higher read ratio achieved steadily higher operation throughput. Also in Fig. 25 we can observe that, when the number of threads was increased, the operation throughput was increased as well. The load of the system was not enough to lead to saturation and the system responded effectively to the increasing workload.

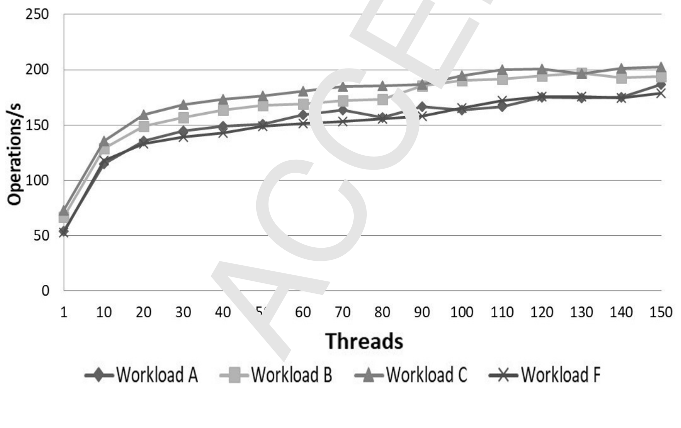


Figure 25: MongoDB operations throughput on bare metal.

We also experimentally evaluated the mechanisms of providing persistent storage with YCSB. Table 5 shows that there was no significant difference in operation throughput for persistent data stored in Docker volume, host directory or in data volume container. Nevertheless, host directory mechanism achieved 5.3% higher operation throughput compared to Docker volume and 3.48% higher operation throughput compared to data volume container.

Table 5: MongoDB operations per second on bare metal for different mechanisms of persistent storage.

	Docker Volume	Host Directory	Data Volume Container
Bare-Metal (Operations / s)	151.35	159.38	154.01

We selected workload A, whose operation ratio is 50/50 read/write, to investigate how the overall system operation throughput is affected by the number of simultaneously running containers. We deployed from 1 to 6 containers on bare-metal, which executed similar workloads and the results of these experiments are presented in Fig. 26. As we can see, there is high level of fairness among the parallelly running containers. Any additional container, which was running in parallel with others, led to 15.63% increment, in average, of the overall operation throughput.

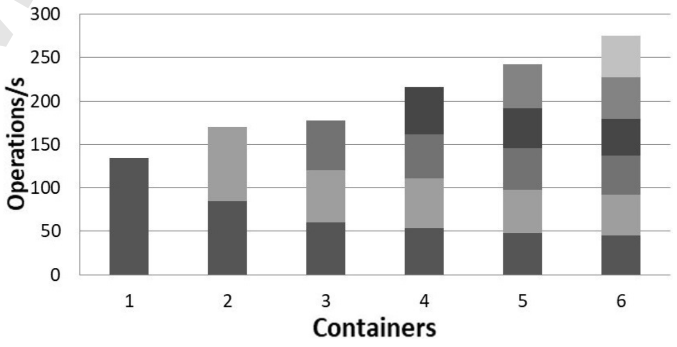


Figure 26: MongoDB operations throughput for 1-6 containers on bare metal.

5.3.2. MongoDB Docker Containers on VMs

Fig. 27 and Fig. 28 present the YCSB results for MongoDB containers on KVM and XEN VMs, running Workloads A, B, C and F, for 1 to 150 threads respectively. By observing Fig. 27 and Fig. 28 we can see that the operations per second were gradually increasing as the number of threads was also increasing.

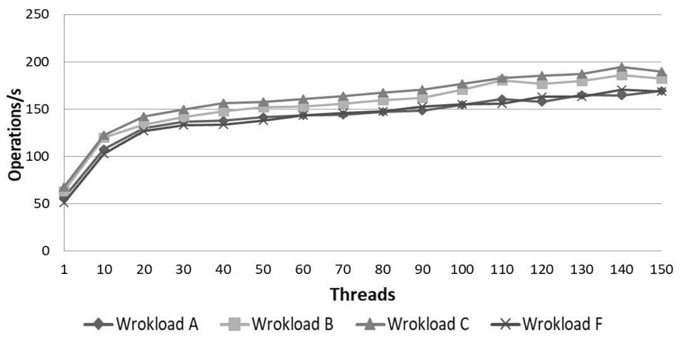


Figure 27: MongoDB operation throughput on KVM VM for different types of workload.

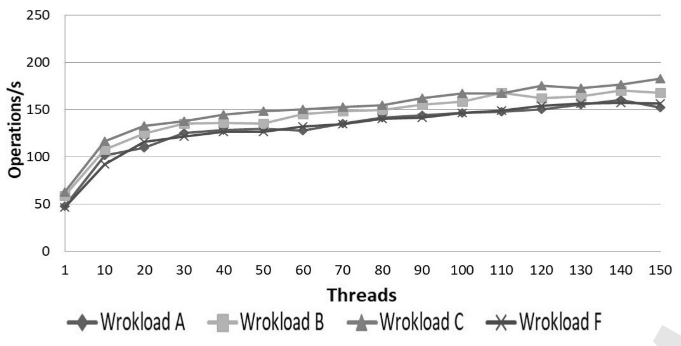


Figure 28: MongoDB operation throughput on XEN VM for different types of workload.

Table 6 compares the mean operation throughput of MongoDB containers running on top of KVM and XEN VMs, as three different mechanisms to store persistent data were employed. We used YCSB benchmark with workload A and 50 threads. Our measurements show that KVM achieved 8.45% better mean operation throughput, in all cases, compared to containers running on XEN. Host directory achieved in average, for both KVM and XEN, 3.19% higher operation throughput compared to Docker volume and 0.08% higher operation throughput compared to data volume container.

Table 6: MongoDB operations per second on KVM and XEN VMs for different mechanisms of persistent storage.

	Docker Volume	Host Directory	Data Volume Container
KVM (Operations / s)	141.88	144.51	143.68
XEN (Operations / s)	12[illegible].79	134.63	133.16

Considering the previous MySQL experiments, which demonstrated that there was an overhead introduced by additional VMs, we repeated this experimental scenario this time for MongoDB NoSQL database. In this set of experiments, we deployed from 1 to 3 VMs and we progressively ran from 1 to 6 containers on top of them. We used YCSB benchmark with workload A and 50 threads to measure how MongoDB performs for different configurations.

Fig. 29 - 31 provide the experimental results for MongoDB containers which were running on 1 to 3 KVM VMs respectively. In all three cases, independently of the number of VMs, the additional containers led to higher overall system performance. On the contrary, as the same number of containers was deployed on more VMs, we observed a decrease on operation throughput. The MongoDB containers which were running on one VM achieved [illegible].75% better operation throughput compared to those running on two VMs and 9.09% better operation throughput compared to those running on three VMs.

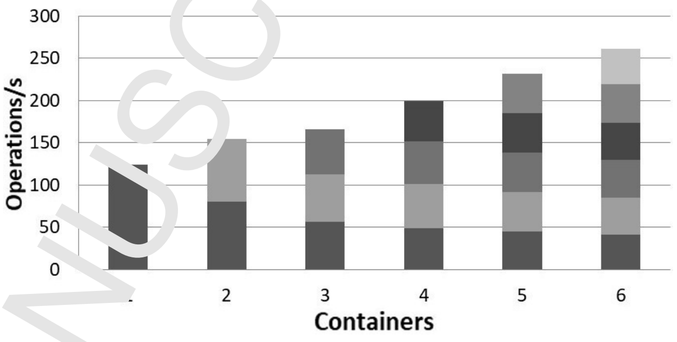


Figure 29: MongoDB operations throughput on 1 KVM VM for 1-6 containers.

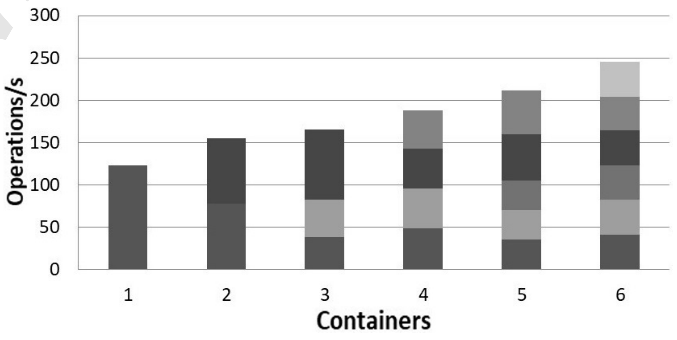


Figure 30: MongoDB operations throughput on 2 KVM VMs for 1-6 containers.

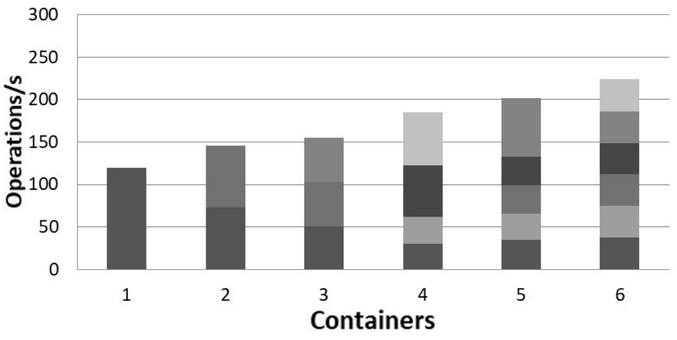


Figure 31: MongoDB operations throughput on 3 KVM VMs for 1-6 containers.

Corresponding to MongoDB containers on KVM experiments, we executed similar measurements for XEN VMs. In Fig. 32 - 34 the results of those measurements are presented. As it would be expected, also in this case, the increasing number of containers resulted to higher operation throughput. On the contrary, the additional VMs led to a higher overhead. The same containers, which were running on one VM, achieved 9.93% better operation throughput compared to those running on two VMs and 20% better operation throughput compared to those running on three VMs.

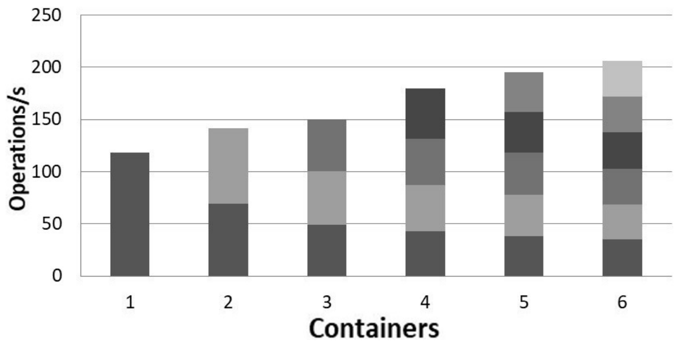


Figure 32: MongoDB operations throughput on 1 XEN VM for 1-6 containers.

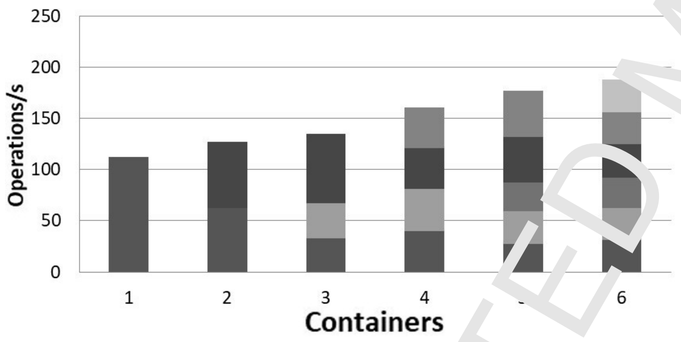


Figure 33: MongoDB operations throughput on 2 XEN VMs for 1-6 containers.

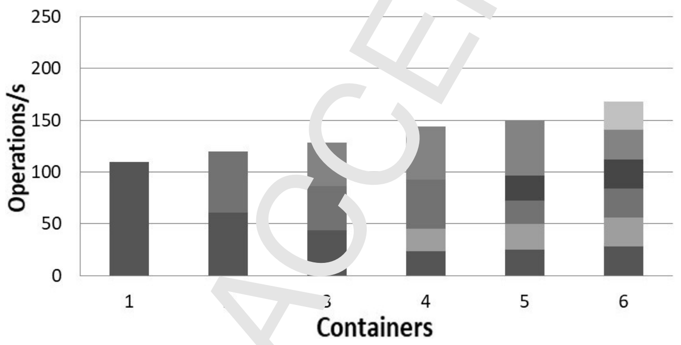


Figure 34: MongoDB operations throughput on 3 XEN VMs for 1-6 containers.

5.3.3. *Experimental Results*

Various experiments showed that Docker successfully stores persistent data. Regarding the methods of storing persistent data, host directory achieved the highest performance. Then data volumes follow and last comes host directory, by 1.06% and 4.37% lower performance respectively.

The I/O benchmark experiments of the previous section, showed that containers, which ran on KVM, achieved 16,73% higher disk performance in average, compared to XEN. As it would be expected, because MySQL and MongoDB are I/O intensive applications, MySQL and MongoDB containers on KVM achieved higher performance compared to XEN. Experiments showed that, for both services, for all different configurations, KVM achieved 20% higher performance compared to XEN. Moreover, any additional KVM VM led to 3.81% performance degradation. Whereas, any additional to the system XEN VM led to 10.16% performance degradation. From the previous experiments we can claim that, for I/O intensive applications, the combination of KVM and Alpine OS seems to be the optimal solution. On the contrary, XEN and CargOS will introduce the highest performance overhead on the system.

6. Power Consumption Analysis

Another critical aspect of cloud computing, is energy efficiency. Due to data centers' current sizes, software-based and hardware-based techniques and architectures are needed to moderate the high energy consumption, without heavily affecting the QoS requirements [82]. Virtualization in cloud computing, is a technique used to enhance resource consolidation, improve energy efficiency and promote green cloud computing [83]. Virtualization can increase CPU utilization by 40-60% and combined with other Green House Data techniques, such as opportunistic free cooling and hot/cold aisle containment, can lead up to 64.5% energy savings compared to an average cloud data center [84]. Furthermore, VM migration techniques allow workloads to migrate on-site or on geo-dispersed sites in order to utilize available renewable energy sources [84].

To better understand and quantify the power consumption overhead introduced by containers running on VMs, we conducted a new series of experiments. In line with our previous experiments, we deployed Docker containers on XEN and KVM VMs, since these virtualization technologies are free and widely used in real-world production environments. The energy efficiency of containers and VMs has been separately evaluated and compared in the past. However, there is no study which investigated the power consumption overhead of the VM-container combination.

In bibliography, there are several studies investigating power consumption of virtualization technologies. In [85] the authors compared KVM and Docker in terms of QoS and energy efficiency. They deployed a LAMP stack (Linux-Apache-MySQL-PHP) service and found that, for their configuration, Docker used 11.33% less energy than KVM. They concluded that Docker outperforms KVM in both QoS and energy efficiency.

Jiang et al. [86] evaluated the energy consumption of different hypervisors and Docker containers on various platforms and workloads. They found that there is not a single hypervisor to outperform the others in terms of power, energy consumption, or performance in all cases. Moreover, they pointed out that, for computation-intensive workloads, containers had similar power efficiency to VMs, even though container virtualization is lighter than the one of VMs.

In [87] OpenStack VM and Docker container were compared. Experiments showed that Docker container consumed less power while executing scientific workflow and while it was in idle state. Also, the container achieved better performance for CPU and memory intensive jobs compared to OpenStack VM.

Morabito [88] compared the power usage of KVM and XEN hypervisors to container-based virtualization Docker and LXC. He also found that, in idle state and in CPU/Memory heavy workloads, containers used marginally less power than VMs. Nevertheless, for network intensive experiments, containers achieved noticeably less power consumption compared to VMs. As the author noted, this happened because network packets have to be processed by extra layers in a hypervisor environment, compared to a container environment.

Finally, Shea et al. [89] compared the power consumption of network tasks on KVM, XEN and OpenVZ to bare metal. Their experiments revealed that the two hypervisors, KVM and XEN, consumed considerably more energy for this kind of tasks. On the other hand, OpenVZ consumed nearly the same amount of power as a non-virtualized system. The authors also proposed an adaptive packet buffering in KVM that could reduce the energy consumption caused by network transaction.

6.1. Power Consumption Experiments

6.1.1. System Setup

As mentioned, our test system is composed of a 3.7 GHz Intel CORE I3-6100 processor, 8GB of DDR4 DIMM RAM and 250GB hard disk. The Intel's x86 architecture, which is used in our system, is dominating in the CPU market and is used by major cloud service providers, such as Amazon [89]. This kind of CPU has low power consumption and is equipped with Running Average Power Limit hardware counters (RAPL), to accurately record CPU power consumption. In our series of experiments, we recorded the overall power consumption of the system and CPU separately and we calculated the total energy consumption of different configurations and workloads. We also measured in real time the internal power consumption for core devices and dram, using the RAPL counters [90]. However, since RAPL counters do not measure the overall system power consumption, we used the Bearware 302717 power meter to measure the total system power consumption (or "wall power") per second. We also recorded utilization metrics and how Dynamic Voltage and Frequency Scaling $DVFS$ adjusted the frequency/voltage of processors between 16 distinct P-states $740 - 3700MHz$ to conserve power.

6.1.2. System in Idle State

The virtual resources of a data center may be idle during periods of low demand. Taking this into consideration, we measured and compared the idle power consumption of VMs and containers running on VMs. We deployed 1 and 6 containers on 1 and 2 KVM and XEN VMs and we took power consumption measurements separately. The VMs were running Ubuntu Server and Alpine. Before recording any power measurement, the system was left for 5 minutes to be balanced and then we recorded the wall power consumption per second for 5 minutes. Table 7 shows the average power consumption results for every case.

It can be seen from the data in Table 6 that the different virtualization technologies under evaluation have roughly the same power consumption in idle state. In average, an idle container on a KVM VM increased the system's power consumption by 0.06% (0.03 Watts), 6 idle containers on a KVM VM increased the system's power consumption by 0.28% (0.14 Watts) and 6 idle containers on 2 KVM VMs (3 containers per VM) increased the system's power consumption by 0.34% (0.17 Watts).

Similar experiments for XEN hypervisor showed that an idle container on a XEN VM increased the system's power consumption by 0.08% (0.04 Watts), 6 containers on a XEN VM increased the system's power consumption by 0.35% (0.17 Watts) and 6 containers on 2 XEN VMS increased the system's power consumption by 0.36% (0.18 Watts). In average, for all different cases, the idle power consumption of XEN was 0.19% higher than KVM, 0.71% higher than Ubuntu bare metal and 1.36% higher than Alpine bare metal. KVM idle power consumption was on average 0.52% higher than Ubuntu bare metal and 1.16% higher than Alpine bare metal.

Our results on KVM and XEN idle power consumption are in alignment with earlier study [88], in which the authors had also found that, in idle state, XEN consumes a bit more power compared to KVM. Because both hypervisors take advantage of the standard Linux power saving system, they consume almost the same amount of power in idle state [88]. Although XEN, due to its architecture (Domain-0), was slightly more power-demanding in idle state compared to KVM. In our use cases, deploying idle containers on both XEN and KVM VM, led from 0.06% to only 0.36% increment of total power consumption.

6.1.3. System Under Stress

We deployed 11 different configurations to study the power consumption overhead of containers running on VMs compared to containers running on bare metal. As shown in Table 8, we took power measurements for different workloads, number of containers and VMs. To ensure a fair analysis between all virtualization technologies, we deployed a similar setup in all cases, on the same server hardware.

For MySQL experiments, for the smallest database size, we set the innodb buffer of MySQL to 2GB and for the bigger one, we set the innodb buffer to 1GB per container. We used the Sysbench benchmark to stress the system and we took power consumption measurements per second for the total system and the CPU. We ran 19 iterations of 1 minute Sysbench tests, with 50 threads per container, for both the small and the big MySQL dataset. Sysbench executed about 65% reads, 25% writes and 10% other queries. For MongoDB experiments, we stressed the system with the YCSB benchmark and we also took power

Table 7: System power consumption in Watts for system in idle state.

	Bare-Metal Ubuntu	Bare-Metal Alpine	KVM		XEN	
			1VM	2VMs	1VM	2VMs
Without Docker	48,90	48,59	49,10	49,21	49,20	49,28
1 container	48,96	48,67	49,13	-	49,24	
6 containers	49,02	48,69	49,24	49,38	49,37	49,46

Table 8: Power consumption experimental configurations.

		Configuration	Number of VMs	Number of containers	DB total size	MySQL InnoDB buffer size
Bare Metal	MySQL	1	-	1	5 GB	2GB / container
		2	-	6	24 GB	1GB / container
	MongoDB	3	-	1	5 GB	-
KVM	MySQL	4	1	1	5 GB	2GB / container
		5	1	6	24 GB	1GB / container
		6	2	6	24 GB	1GB / container
	MongoDB	7	1	1	5 GB	-
XEN	MySQL	8	1	1	5 GB	2GB / container
		9	1	6	24 GB	1GB / container
		10	2	6	24 GB	1GB / container
	MongoDB	11	1	1	5 GB	-

consumption measurements per second for the whole system and the CPU separately. We stressed the system by running 19 iterations with 10.000 operations for workload A (see Table 4), with 50 threads per container. In all cases where containers were running on VMs, we configured the virtualized nodes to exploit all the available resources of the server.

Experimental Results

Power data for the whole system was gathered over time. Fig. 35 - 37 provide the power measurements for the whole system, while 19 MySQL Sysbench tasks were executed consecutively for different system configurations (see Table 8). As it can be seen from Fig. 35 - 37, in most cases, bare metal achieved the lowest power consumption. Also, we can observe that the power consumption of running containers on one VM is lower, compared to running the same containers on two VMs of the same technology (Fig. 36 - 37). As the experiments progress over time, we can see that this difference is getting even higher. Moreover, we note that, especially for the biggest dataset, containers running on VMs consumed more time to perform the same tasks, compared to containers running on bare metal.

To provide an aggregated view of the above empirical measurements, we calculated the mean power and the mean energy consumption for every experimental configuration; the calculated values are presented in Fig. 42 - 43. As it can be seen in Fig. 42, a container running on a KVM VM has 1.48% power overhead compared to the same container running on bare metal. The same one running on a XEN VM, has 3.38% power overhead compared to bare metal. Regarding energy consumed by a container when running the same benchmark tasks, the container running on KVM VM consumed 2.25% more energy compared to bare metal and the container running on a XEN VM consumed 4.25% more energy compared to bare metal as well.

Experiments with higher number of containers and VMs have a different power consumption impact on the system. We observe from figure Fig. 43 that running containers on VMs is marginally less power-demanding than running the same containers on bare metal. In case of containers running on a KVM VM and a XEN VM, this leads to 1.44% and 2.23% less power consumption compared to running the same containers on bare metal respectively. This case is also observed by Jin et al. [91] and may happen because of different processes simultaneously running on the CPU cores.

In Fig. 36 - 37, we can distinguish two very often observed values of about 50 and 65 Watts, nevertheless this pattern is not appeared in Fig. 35. Assuming that this is due to CPU's DVFS mechanism, we focused on two representative cases and we recorded the values of RAPL counters to further investigate how the DVFS mechanism affects the CPU power consumption and, as a result, the overall system power consumption. In Fig. 38 we can see the total power consumption of the system, the CPU's power consumption, the power consumption of CPU's dram and cores, as well as the CPU frequency for the experimental configuration 4 (see Table 8). Similar measurements were taken for the experimental configuration 6. These results are presented in Fig. 39.

Both Fig. 38 and Fig. 39 reveal that there is a strong relation between CPU and the overall system's power consumption. Higher CPU frequency led to higher CPU power consumption, which in turn led to higher overall system's power consumption. For example in Fig. 38 from 265 to 309 seconds, CPU operated with 0.95 GHz average frequency and the average system power consumption was 51.78 Watts. On the other hand, we can see in Fig. 39 that, from 113 to 137 seconds, CPU operated at 3.38 GHz and the average system power consumption

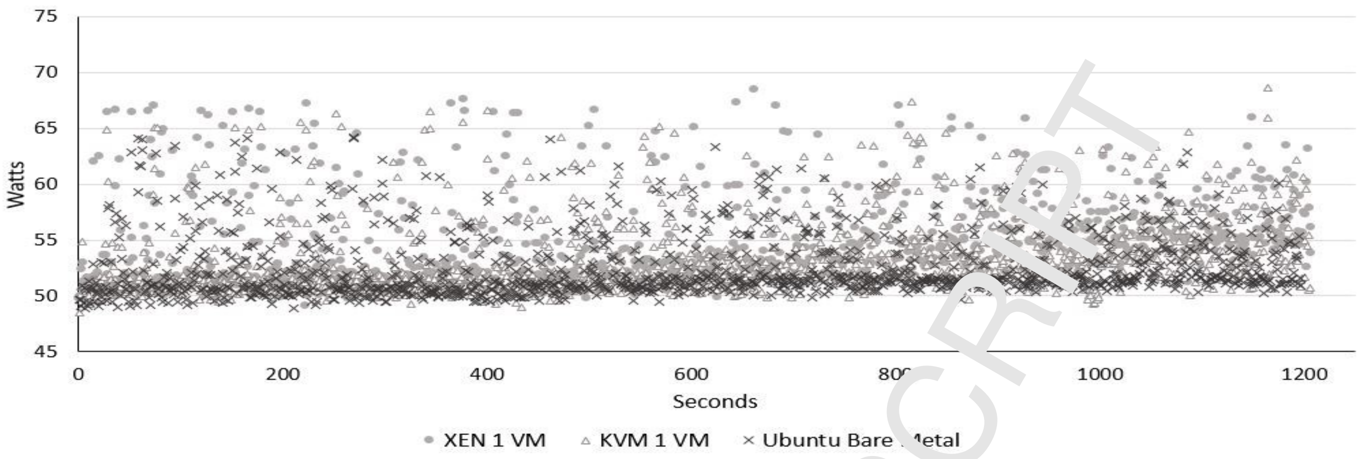


Figure 35: System's power consumption of running MySQL container with small dataset on bare metal and on a VM.

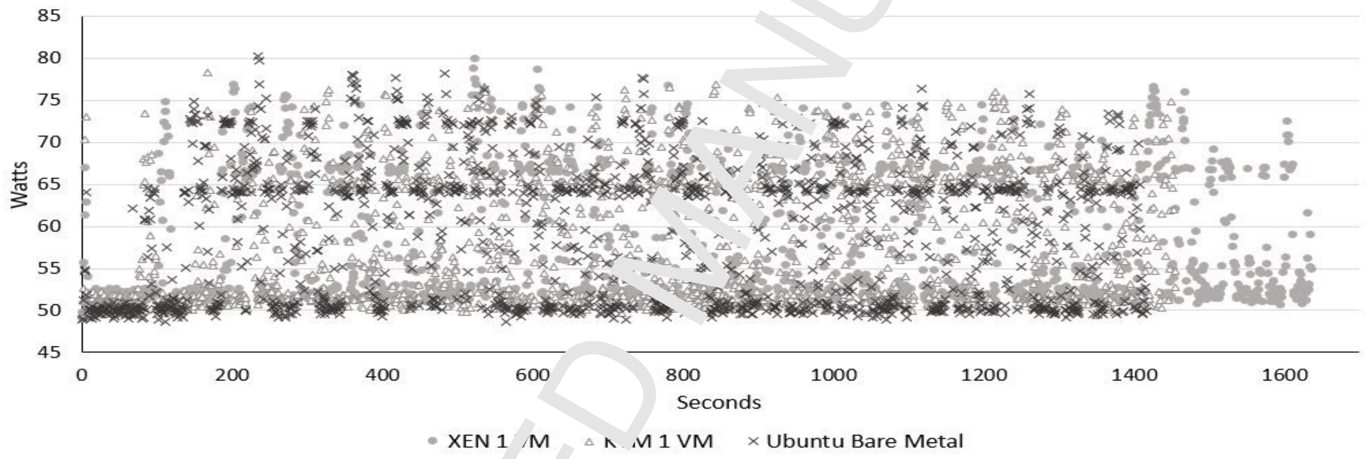


Figure 36: System's power consumption of running MySQL container with big dataset on bare metal and on a VM.

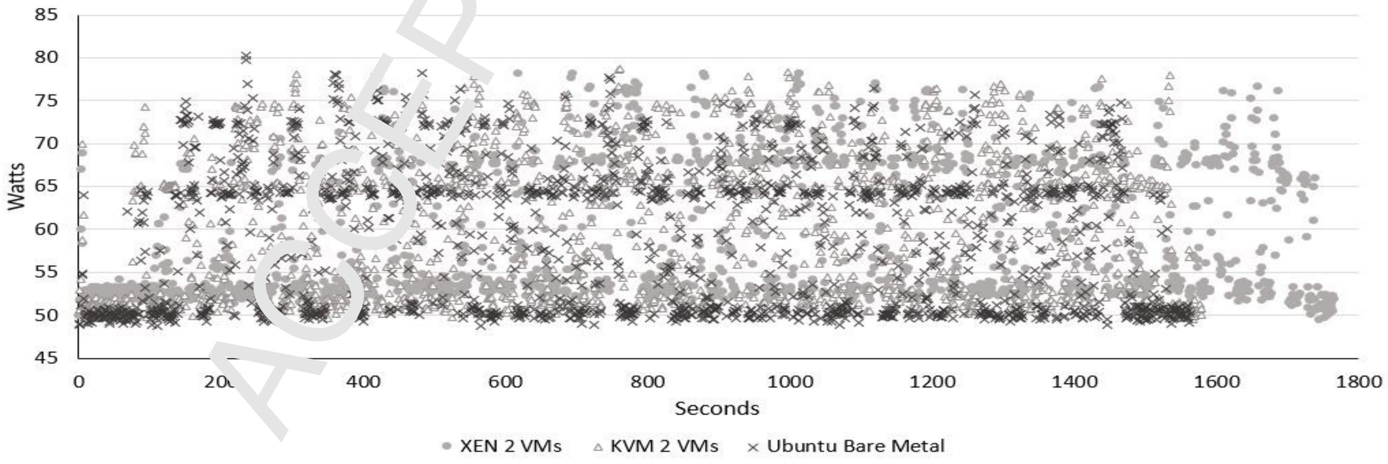


Figure 37: System's power consumption of running 6 MySQL containers with big dataset on bare metal and on 2 VMs.

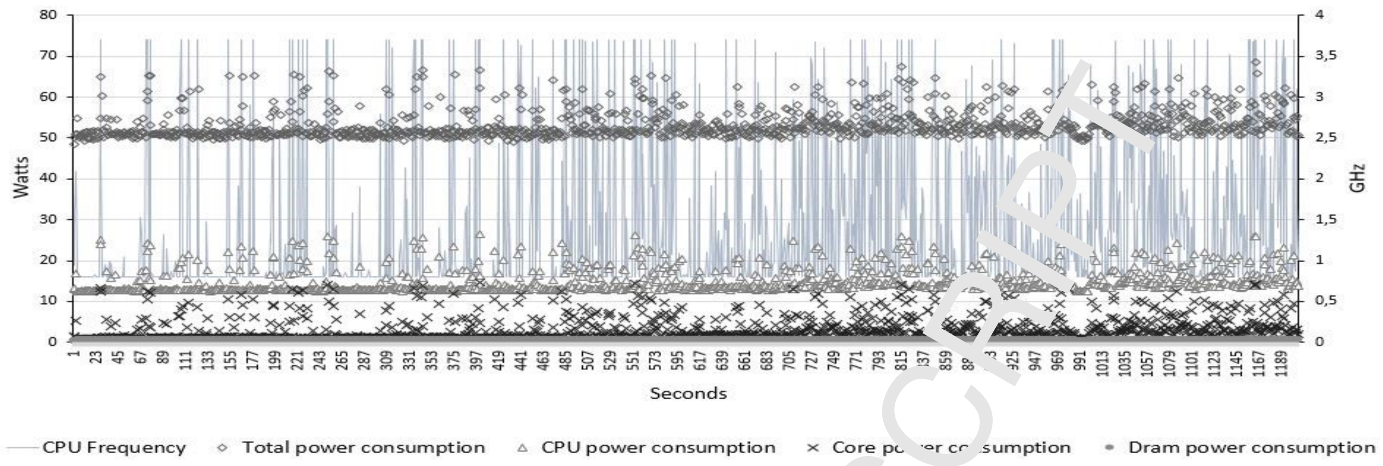


Figure 38: CPU frequency and power consumption of the whole system, dram and cores, for system running a MySQL container on KVM VM.

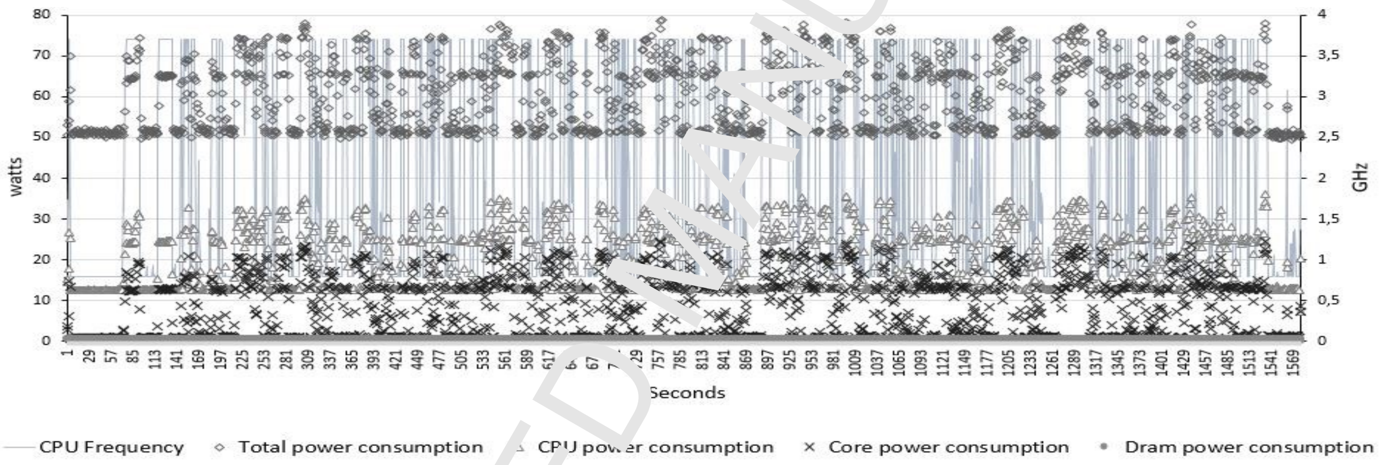


Figure 39: CPU frequency and power consumption of the whole system, dram and cores, for system running 6 MySQL containers on 2 KVM VMs.

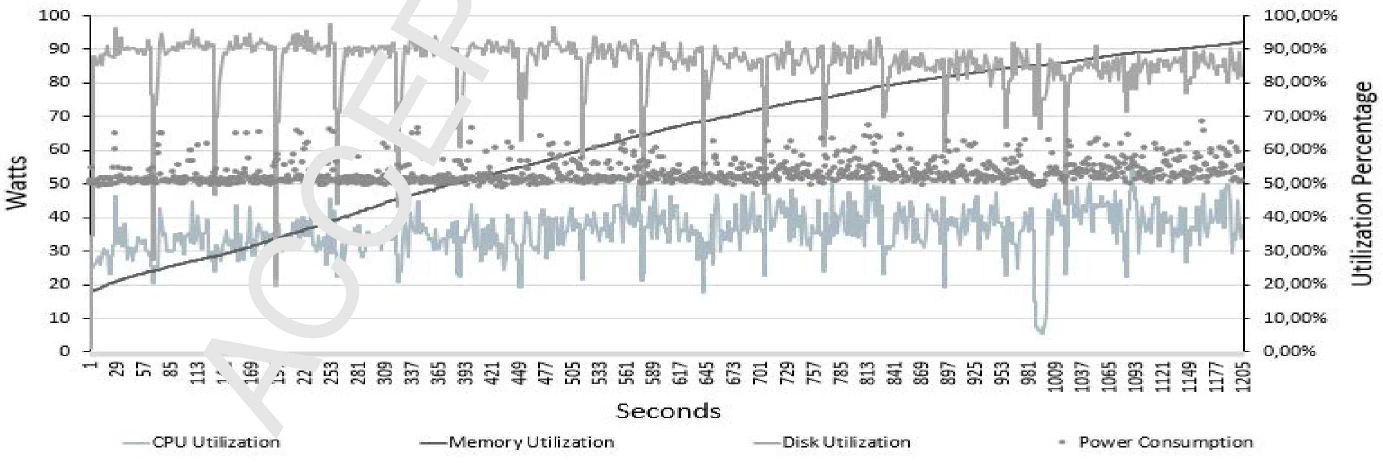


Figure 40: System power consumption and CPU, disk and memory utilization of system running a MySQL container on KVM VM.

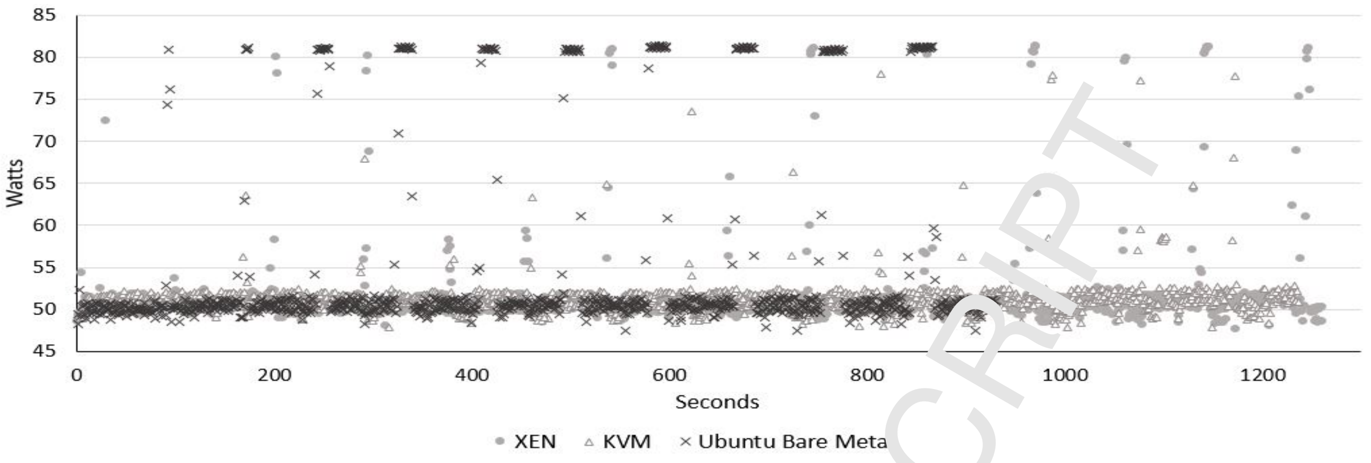
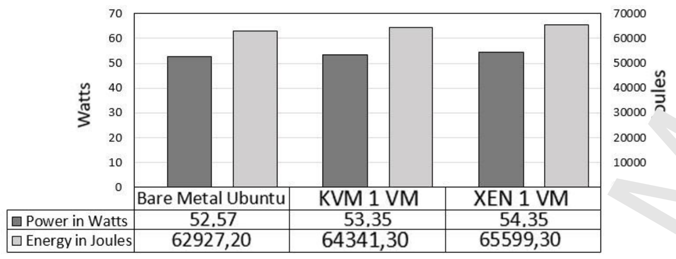
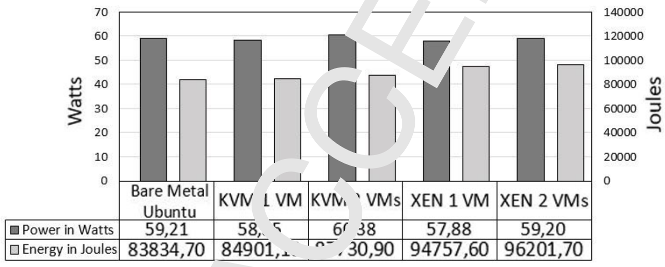


Figure 41: System's power consumption of running MongoDB containers on bare metal and on a VM.



	Bare Metal Ubuntu	KVM 1 VM	XEN 1 VM
▩ Power in Watts	52,57	53,35	54,35
▢ Energy in Joules	62927,20	64341,30	65599,30

Figure 42: Average power and energy consumption of MySQL small dataset containers.



	Bare Metal Ubuntu	KVM 1 VM	KVM 2 VMs	XEN 1 VM	XEN 2 VMs
▩ Power in Watts	59,21	58,[illegible]5	60,38	57,88	59,20
▢ Energy in Joules	83834,70	84901,[illegible]	8[illegible]30,90	94757,60	96201,70

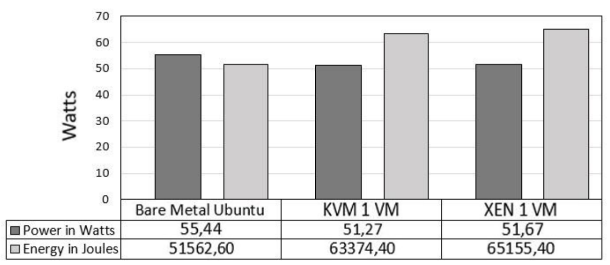
Figure 43: Average power and energy consumption of MySQL big dataset containers.

was 62.72 Watts. For the experimental configuration 4 the average CPU frequency was 1.38 GHz, whereas for experimental configuration 6 the system was under heavier load and the average CPU frequency was 2.32 GHz. For this reason we can see that, in cases where 6 containers are running with larger databases, CPU was operated at higher frequency, consuming more power. This applies also in case of running containers without additional virtualization layer, as we can see for bare metal measurements in these figures.

Our empirical analysis shows that CPU affects the power consumption behavior of the system rather heavily. The disk is recognized in bibliography [92] as the second highest energy consumption impact factor, especially for this kind of heavy I/O applications. Fig. 40 shows the correlation between the system's total power consumption and CPU, disk and memory utilization in time, for the experimental configuration 4 (see Table 8). We can observe that, in periods with high CPU utilization, the disk was also highly utilized and the system was more power demanding. There is a strong correlation between CPU frequency and disk power consumption, which is also noted by Arjona et al. [92]. On the other hand, even though memory utilization was steadily increased from 20% to 90% during this experiment, we do not see any noticeable impact on the system's power consumption.

Power consumption measurements were also taken for the MongoDB service. To stress the system, we executed 50% read and 50% update operations by using the YCSB benchmark. Fig. 41 shows that also in this case, the system operated mainly in two power states, in a lower power state at about 50 Watts and in a higher power state at about 80 Watts. This specific experimental configuration did not stress the system heavily and, as a result, for most of the time, the system's power consumption was about 50 Watts.

Regarding energy consumption of the specific YCSB tasks, although power consumption of a container running on bare metal was higher than the one of a container running on VM, more execution time was consumed by the container running on VM while executing the same benchmark tasks. Fig. 44 provides the mean power and energy consumption for these MongoDB experiments. As it can be observed, the container running on KVM VM consumed 22.91% more energy, compared to the one running on bare metal and also the container running on a XEN VM consumed 26.36% more energy, compared to bare metal.



	Bare Metal Ubuntu	KVM 1 VM	XEN 1 VM
■ Power in Watts	55,44	51,27	51,67
□ Energy in Joules	51562,60	63374,40	65155,40

Figure 44: Average power and energy consumption of MongoDB containers.

In conclusion, power consumption measurements showed that, for the system in idle state, all configurations had almost the same power consumption. The power consumption of both KVM and XEN was almost the same for the system under stress. However, regarding the energy consumed by containers to complete the same tasks, containers running on KVM consumed 7,11% less energy compared to containers running on XEN. So, KVM not only achieved the highest performances for MySQL and MongoDB containers, but also consumed the least energy in comparison to XEN.

7. Conclusions

In this study we combined the two most widely used virtualization technologies on cloud computing, VMs and containers. We highlighted that these two technologies are complementary and we proposed to run containers on VMs in order to improve containers' isolation, resource utilization, system's management and functionality. This approach offers some great benefits, however they come at a performance cost. In contrast to the majority of studies in bibliography, which separately compared VMs and containers, we evaluated their coexistence under various conditions. There are a few works that also conduct experiments by combining VMs and containers, but there is no research which used our approach to present the benefits of this method or which investigated how important factors, such as the number of VMs, the number of containers per VM and the Guest's OS, affect the performance and also the energy consumption of the system.

To quantify the performance overhead introduced by the additional layer of VMs, we conducted several experiments with various hypervisors, container-centric OSes and configurations. Our measurements showed a minor CPU and memory performance overhead of containers running on VMs, especially on XEN VMs. The disk and network performance is affected more heavily by the additional virtualization layer of VM. KVM hypervisor achieved the highest disk and network performance compared to other hypervisors. In addition, as far as OSes are concerned, Alpine, CoreOS and RancherOS achieved the highest performances.

Additionally, we deployed two different types of DBMS services, MySQL and MongoDB as use cases. Because database services are common in cloud and containers were initially designed for stateless applications, we considered DBMS as an interesting type of service to examine. After several experiments, we confirmed that Docker successfully manages persistent data in different ways. The experimental results showed that the overall performance of containers running on VMs is affected by the type of applications, the number of VMs, the number of containers, the guest's OS, the storage mechanism and the different types of workloads. There is a noticeable overhead introduced by the VMs, especially as the number of VMs and parallelly running containers is getting higher.

Furthermore, we studied the energy consumption impact of running containers on VMs. We took power consumption measurements for the whole system and for the CPU separately, for the system being in idle state and under stress, for various experimental configurations. For the system in idle state, as may expected, there was almost no overhead. But by stressing the system with running both MySQL and MongoDB services, we noticed that the XEN hypervisor is more energy demanding compared to KVM. Also we did not only measure the total system power consumption but we investigated how factors, such as CPU frequency and disk utilization, affect it.

In this work we combined VMs and containers to enhance containers' isolation and extend VMs' functionality. As the main drawback of this approach we can consider the performance and energy overhead of the additional virtualization layer. We experimentally quantified these overheads and presented how various factors affect it. The factors presented above, can be modified and adapted in different situations, in order to further improve the overall performance or isolation. For example, for security sensitive applications, running one container per VM can be considered as a good approach while, for performance demanding applications, running a number of containers per VM could be a better solution.

The findings of this study will serve as a base for future work. An interesting research topic is to investigate how the results of our analysis can be exploited by the currently available solutions for (multi-)cloud governance. Future directions may include the extension of the open reference architecture presented in [93], to support VM-container deployments. Also, we could integrate the findings of our study in cloud application automation standards like, TOSCA [94], which enables the creation, automated deployment and management of portable cloud applications. We could even extend tools, as in [95], to use TOSCA-based representation and VM-containers approach, to specify and orchestrate complex application in heterogeneous cloud platforms.

References

[1] I. Mavridis, H. Karatza, Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark, Journal of Systems and Software 125 (2017) 133–151. doi:10.1016/j.jss.2016.11.037.

[2] I. Mavridis, H. Karatza, Performance and overhead study of containers running on top of virtual machines, in: 2017 IEEE 19th Conference on Business Informatics (CBI), Vol. 2, IEEE, 2017, pp. 32–38. doi:10.1109/CBI.2017.69.

[3] G. L. Stavrinides, H. D. Karatza, A cost-effective and qos-aware approach to scheduling real-time workflow applications in paas and saas clouds, in: 2015 3rd International Conference on Future Internet of Things and Cloud (FiCloud), IEEE, 2015, pp. 231–239. doi:10.1109/FiCloud.2015.93.

[4] R. Peinl, F. Holzschuher, F. Pfitzer, Docker cluster management for the cloud-survey results and own solution, Journal of Grid Computing 14 (2) (2016) 265–282. doi:10.1007/s10723-016-9366-y.

[5] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2015, pp. 171–172. doi:10.1109/ISPASS.2015.7095802.

[6] R. K. Barik, R. K. Lenka, K. R. Rao, D. Ghose, Performance analysis of virtual machines and containers in cloud computing, in: 2016 International Conference on Computing, Communication and Automation (ICCCA), IEEE, 2016, pp. 1204–1210. doi:10.1109/CCAA.2016.7813925.

[7] J. Higgins, V. Holmes, C. Venters, Securing user defined containers for scientific computing, in: 2016 International Conference on High Performance Computing & Simulation (HPCS), IEEE, 2016, pp. 449–453. doi:10.1109/HPCSim.2016.7568369.

[8] T. Combe, A. Martin, R. Di Pietro, To docker or not to docker: A security perspective, IEEE Cloud Computing 3 (5) (2016) 54–62. doi:10.1109/MCC.2016.100.

[9] S. H. Davis, A. Sweemer, C. Greenwood, B. J. Corrie, G. Hicken, Z. Yang, Using virtual machine containers in a virtualized computing platform, uS Patent App. 14/505,349 (Apr. 7 2016).

[10] D. Tychalas, H. Karatza, High performance system based on cloud and beyond: Jungle computing, Journal of Computational Science. doi:10.1016/j.jocs.2017.03.027.

[11] A. Elsayed, N. Abdelbaki, Performance evaluation and comparison of the top market virtualization hypervisors, in: 2013 8th International Conference on Computer Engineering & Systems (ICCES), IEEE, 2013, pp. 45–50. doi:10.1109/ICCES.2013.6707169.

[12] J. Hwang, S. Zeng, F. y Wu, T. Wood, A component-based performance comparison of four hypervisors, in: 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), IEEE, 2013, pp. 269–276.

[13] L. Nussbaum, F. Anhalt, O. Mornard, J.-P. Gelas, Linux-based virtualization for hpc clusters, in: Montreal Linux Symposium, 2009.

[14] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, G. C. Fox, Analysis of virtualization technologies for high performance computing environments, in: 2011 IEEE International Conference on Cloud Computing (CLOUD), IEEE, 2011, pp. 9–16. doi:10.1109/CLOUD.2011.29.

[15] S. Varrette, M. Guzek, V. Plugaru, X. Besseron, P. Bouvry, Hpc performance and energy-efficiency of xen, kvm and vmware hypervisors, in: 2013 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, 2013, pp. 89–96. doi:10.1109/SBAC-PAD.2013.18.

[16] J. Che, C. Shi, Y. Yu, W. Lin, A synthetical performance evaluation of openvz, xen and kvm, in: 2010 IEEE Asia-Pacific Services Computing Conference (APSCC), IEEE, 2010, pp. 587–594. doi:10.1109/APSCC.2010.83.

[17] M. G. Xavier, M. V. Neves, C. A. F. De Rose, A performance comparison of container-based virtualization systems for mapreduce clusters, in: 2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE, 2014, pp. 299–306. doi:10.1109/PDP.2014.78.

[18] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai, et al., Skyport: container-based execution environment management for multi-cloud scientific workflows, in: Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds, IEEE Press, 2014, pp. 25–32. doi:10.1109/DataCloud.2014.6.

[19] Z. J. Estrada, Z. Stephens, C. Pham, Z. Kalbarczyk, R. K. Iyer, A performance evaluation of sequence alignment software in virtualized environments, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, 2014, pp. 730–737. doi:10.1109/CCGrid.2014.125.

[20] Z. Kozhirbayev, R. O. Sinnott, A performance comparison of container-based technologies for the cloud, Future Generation Computer Systems 68 (2017) 175–182. doi:10.1016/j.future.2016.08.025.

[21] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2015, pp. 171–172. doi:10.1109/ISPASS.2015.7095802.

[22] R. Morabito, J. Kjällman, M. Komu, Hypervisors vs. lightweight virtualization: a performance comparison, in: 2015 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2015, pp. 386–393. doi:10.1109/IC2E.2015.74.

[23] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, C. A. De Rose, Performance evaluation of container-based virtualization for high performance computing environments, in: 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE, 2013, pp. 233–240. doi:10.1109/PDP.2013.41.

[24] M. Raho, A. Spyridakis, M. Paolino, D. Raho, Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing, in: 2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), IEEE, 2015, pp. 1–8. doi:10.1109/AIEEE.2015.7367280.

[25] C. G. Kominos, N. Seyvet, K. Vandikas, Bare-metal, virtual machines and containers in openstack, in: 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), IEEE, 2017, pp. 36–43. doi:10.1109/ICIN.2017.7899247.

[26] R. S. Eiras, R. S. Couto, M. G. Rubinstein, Performance evaluation of a virtualized http proxy in kvm and docker, in: 2016 7th International Conference on the Network of the Future (NOF), IEEE, 2016, pp. 1–5. doi:10.1109/NOF.2016.7810144.

[27] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, Y. Al-Hammadi, Performance comparison between container-based and vm-based services, in: 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), IEEE, 2017, pp. 185–190. doi:10.1109/ICIN.2017.7899408.

[28] M. Plauth, L. Feinbube, A. Polze, A performance survey of lightweight virtualization techniques, in: European Conference on Service-Oriented and Cloud Computing, Springer, 2017, pp. 34–48. doi:10.1007/978-3-319-67262-5_3.

[29] P. Sharma, L. Chaufournier, P. Shenoy, Y. Tay, Containers and virtual machines at scale: A comparative study, in: Proceedings of the 17th International Middleware Conference, ACM, 2016, p. 1. doi:10.1145/2988336.2988337.

[30] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html, [Online; accessed 29-Sept-2018].

[31] https://cloud.google.com/compute/docs/cpu-platforms, [Online; accessed 29-Sept-2018].

[32] O. S. Nagesh, T. Kumar, V. R. Vedula, A survey on security aspects of server virtualization in cloud computing, International Journal of Electrical and Computer Engineering (IJECE) 7 (3) (2017) 1326–1336. doi:10.11591/ijece.v7i3.pp1326-1336.

[33] D. Marshall, Understanding full virtualization, paravirtualization, and hardware assist, VMWare White Paper (2007) 17.

[34] I. Studnia, E. Alata, Y. Deswarte, M. Kaâniche, V. Nicomette, Survey of security problems in cloud computing virtual machines, in: Computer and Electronics Security Applications Rendez-vous (C&ESAR 2012). Cloud and security: threat or opportunity, 2012, pp. p–61.

[35] D. Petrovic, A. Schiper, Implementing virtual machine replication: A case study using xen and kvm, in: 2012 IEEE 26th International Conference on Advanced Information Networking and Applications (AINA), IEEE, 2012, pp. 73–80. doi:10.1109/AINA.2012.50.

[36] A. Binu, G. S. Kumar, Virtualization techniques: a methodical review of xen and kvm, Advances in Computing and Communications (2011) 399–410doi:10.1007/978-3-642-22709-7_40.

[37] https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/get-started/install-the-hyper-v-role-on-windows-server, [Online; accessed 29-Sept-2018].

[38] https://msdn.microsoft.com/en-us/library/cc768520(v=bts.10).aspx, [Online; accessed 29-Sept-2018].

[39] N. Regola, J.-C. Ducom, Recommendations for virtualization technologies in high performance computing, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2010, pp. 409–416. doi:10.1109/CloudCom.2010.71.

[40] S. G. Soriga, M. Barbulescu, A comparison of the performance and scalability of xen and kvm hypervisors, in: 2013 RoEduNet International Conference 12th Edition Networking in Education and Research, IEEE, 2013, pp. 1–6. doi:10.1109/RoEduNet.2013.6714189.

[41] O. Agesen, Software and hardware techniques for x86 virtualization, Palo Alto CA, VMware Inc.

[42] X. Wang, J. Zang, Z. Wang, Y. Luo, X. Li, Selective hardware/software memory virtualization, ACM SIGPLAN Notices 46 (7) (2011) 217–226. doi:10.1145/2007477.1952710.

[43] X. Chang, H. Franke, Y. Ge, T. Liu, K. Wang, J. Xenidis, F. Chen, Y. Zhang, Improving virtualization in the presence of software managed translation lookaside buffers, in: ACM SIGARCH Computer Architecture News, Vol. 41, ACM, 2013, pp. 120–129. doi:10.1145/2485922.2485933.

[44] N. Bhatia, Performance evaluation of intel ept hardware assist, VMware, Inc.

[45] I. Korkin, Hypervisor-based active data protection for integrity and confidentiality of dynamically allocated memory in windows kernel, arXiv preprint arXiv:1805.11847.

[46] K. Hwang, J. Dongarra, G. C. Fox, Distributed and cloud computing: from parallel processing to the internet of things, Morgan Kaufmann, 2013.

[47] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, J. Wiegert, Intel virtualization technology for directed i/o., Intel technology journal 10 (3).

[48] https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices, [Online; accessed 29-Sept-2018].

[49] https://cloud.google.com/containers/, [Online; accessed 29-Sept-2018].

[50] X. Xu, H. Yu, X. Pei, A novel resource scheduling approach in container based clouds, in: 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE), IEEE, 2014, pp. 257–264. doi:10.1109/CSE.2014.77.

[51] https://www.docker.com, [Online; accessed 29-Sept-2018].

[52] http://aufs.sourceforge.net/aufs2/man.html, [Online; accessed 29-Sept-2018].

[53] A. Tosatto, P. Ruiu, A. Attanasio, Container-based orchestration in cloud: state of the art and challenges, in: 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), IEEE, 2015, pp. 70–75. doi:10.1109/CISIS.2015.35.

[54] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi, Cloud container technologies: a state-of-the-art review, IEEE Transactions on Cloud Computing. doi:10.1109/TCC.2017.2702586.

[55] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, N. Thoai, Using docker in high performance computing applications, in: 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), IEEE, 2016, pp. 52–57. doi:10.1109/CCE.2016.7562612.

[56] A. T. Gjerdrum, H. D. Johansen, D. Johansen, Implementing informed consent as information-flow policies for secure analytics on ehealth data: Principles and practices, in: 2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE), IEEE, 2016, pp. 107–112. doi:10.1109/CHASE.2016.39.

[57] N. Naik, Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems, in: 2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA), IEEE, 2016, pp. 1–8. doi:10.1109/MESOCA.2016.9.

[58] C.-N. Mao, M.-H. Huang, S. Padhy, S.-T. Wang, W.-C. Chung, Y.-C. Chung, C.-H. Hsu, Minimizing latency of real-time container cloud for software radio access networks, in: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2015, pp. 611–616. doi:10.1109/CloudCom.2015.67.

[59] https://docs.docker.com/, [Online; accessed 29-Sept-2018].

[60] C. de Alfonso, A. Calatrava, G. Moltó, Container-based virtual elastic clusters, Journal of Systems and Software 127 (2017) 1–11. doi:10.1016/j.jss.2017.01.007.

[61] A. Baliga, L. Iftode, X. Chen, Automated containment of rootkits attacks, Computers & Security 27 (7-8) (2008) 323–334. doi:10.1016/j.cose.2008.06.003.

[62] H. Moon, H. Lee, I. Heo, K. Kim, Y. Paek, B. B. Kang, Detecting and preventing kernel rootkit attacks with bus snooping, IEEE Transactions on Dependable and Secure Computing 14 (2) (2017) 145–157. doi:10.1109/TDSC.2015.2443803.

[63] X. Xie, W. Wang, Rootkit detection on virtual machines through deep information extraction at hypervisor-level, in: 2013 IEEE Conference on Communications and Network Security (CNS), IEEE, 2013, pp. 498–503. doi:10.1109/CNS.2013.6682760.

[64] https://coreos.com/, [Online; accessed 29-Sept-2018].

[65] http://rancher.com/, [Online; accessed 29-Sept-2018].

[66] https://www.projectatomic.io/, [Online; accessed 29-Sept-2018].

[67] https://carbos.io/, [Online; accessed 29-Sept-2018].

[68] https://www.alpinelinux.org/about/, [Online; accessed 29-Sept-2018].

[69] J. J. Dongarra, P. Luszczek, A. Petitet, The linpack benchmark: past, present and future, Concurrency and Computation: practice and experience 15 (9) (2003) 803–820. doi:10.1002/cpe.728.

[70] https://www.cs.virginia.edu/stream/ref.html, [Online; accessed 29-Sept-2018].

[71] http://www.iozone.org/, [Online; accessed 29-Sept-2018].

[72] http://www.netperf.org/, [Online; accessed 29-Sept-2018].

[73] https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers, [Online; accessed 29-Sept-2018].

[74] https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#data-volumes-and-the-storage-driver, [Online; accessed 29-Sept-2018].

[75] https://docs.docker.com/engine/admin/volumes/#more-details-about-mount-types, [Online; accessed 29-Sept-2018].

[76] https://github.com/Azure/azurefile-dockervolumedriver, [Online; accessed 29-Sept-2018].

[77] https://github.com/mcuadros/gce-docker, [Online; accessed 29-Sept-2018].

[78] http://www.oracle.com/technetwork/database/mysql/index.html, [Online; accessed 29-Sept-2018].

[79] https://github.com/akopytov/sysbench, [Online; accessed 29-Sept-2018].

[80] https://www.mongodb.com/, [Online; accessed 29-Sept-2018].

[81] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with ycsb, in: Proceedings of the 1st ACM symposium on Cloud computing, ACM, 2010, pp. 143–154. doi:10.1145/1807128.1807152.

[82] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, S. U. Khan, Survey of techniques and architectures for designing energy-efficient data centers, IEEE Systems Journal 10 (2) (2016) 507–519. doi:10.1109/JSYST.2014.2315823.

[83] J. Shuja, R. W. Ahmad, A. Gani, A. I. A. Ahmed, A. Siddiqa, K. Nisar, S. U. Khan, A. Y. Zomaya, Greening emerging it technologies: techniques and practices, Journal of Internet Services and Applications 8 (1) (2017) 9. doi:10.1186/s13174-017-0060-5.

[84] J. Shuja, A. Gani, S. Shamshirband, R. W. Ahmad, K. Bilal, Sustainable cloud data centers: a survey of enabling techniques and technologies, Renewable and Sustainable Energy Reviews 62 (2016) 195–214. doi:10.1016/j.rser.2016.04.034.

[85] I. Cuadrado-Cordero, A.-C. Orgerie, J.-M. Menaud, Comparative experimental analysis of the quality-of-service and energy-efficiency of vms and containers' consolidation for cloud applications, in: 2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), IEEE, 2017, pp. 1–6. doi:10.23919/SOFTCOM.2017.8115516.

[86] C. Jiang, Y. Wang, D. Ou, Y. Li, J. Zhang, J. Wan, B. Luo, W. Shi, Energy efficiency comparison of hypervisors, Sustainable Computing: Informatics and Systems. doi:10.1016/j.suscom.2017.09.005.

[87] A. Jaikar, S. Bae, H. Han, B. Kong, S. A. R. Shah, S.-Y. Noh, Openstack and docker comparison for scientific workflow wrt execution and energy.

[88] R. Morabito, Power consumption of virtualization technologies: An empirical investigation, in: 2015 IEEE/ACM 8th International Conference

on Utility and Cloud Computing (UCC), IEEE, 2015, pp. 522–527. doi:10.1109/UCC.2015.93.

[89] R. Shea, H. Wang, J. Liu, Power consumption of virtual machines with network transactions: Measurement and improvements, in: 2014 Proceedings IEEE INFOCOM, IEEE, 2014, pp. 1051–1059. doi:10.1109/INFOCOM.2014.6848035.

[90] https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl, [Online; accessed 29-Sept-2018].

[91] Y. Jin, Y. Wen, Q. Chen, Energy efficiency and server virtualization in data centers: An empirical investigation, in: 2012 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2012, pp. 133–138. doi:10.1109/INFCOMW.2012.6193474.

[92] J. Arjona Aroca, A. Chatzipapas, A. Fernández Anta, V. Mancuso, A measurement-based analysis of the energy consumption of data center servers, in: Proceedings of the 5th international conference on Future energy systems, ACM, 2014, pp. 63–74. doi:10.1145/2602044.2602061.

[93] A. Brogi, J. Carrasco, J. Cubo, F. DAndria, E. Di Nitto, M. Guerriero, D. Pérez, E. Pimentel, J. Soldani, Seaclouds: an open reference architecture for multi-cloud governance, in: European Conference on Software Architecture, Springer, 2016, pp. 334–338. doi:10.1007/978-3-319-48992-6_25.

[94] A. Brogi, J. Soldani, P. Wang, Tosca in a nutshell: Promises and perspectives, in: European Conference on Service-Oriented and Cloud Computing, Springer, 2014, pp. 171–186. doi:10.1007/978-3-662-44879-3_13.

[95] A. Brogi, L. Rinaldi, J. Soldani, Tosker: A synergy between tosca and docker for orchestrating multicomponent applications, Software: Practice and Experience 48 (11) (2018) 2061–2079. doi:10.1002/spe.2625.

Helen Karatza is a Professor Emeritus in the Department of Informatics at the Aristotle University of Thessaloniki, Greece. Dr.Karatza's research interests include Computer Systems Modeling and Simulation, Performance Evaluation, Grid and Cloud Computing, Energy Efficiency in Large Scale Distributed Systems, Resource Allocation and Scheduling and Real-time Distributed Systems. Professor Karatza is the Editor-in-Chief of the Elsevier Journal "Simulation Modeling Practice and Theory", Area Editor of the "Journal of Systems and Software" of Elsevier, and she has been Guest Editor of Special Issues in multiple International Journals.

Ilias Mavridis received his BSc in Computer Science from the Department of Computer Science and Biomedical Informatics from the University of Thessaly in 2012. He received his MSc in Networks-Communications-Systems Architecture from the Department of Informatics at the Aristotle University of Thessaloniki in 2015. Currently he is a PhD Student in the Department of Informatics at the Aristotle University of Thessaloniki, under the supervision of Professor Helen Karatza. He is also a member of the Parallel and Distributed Systems Group, under the coordination of Professor Helen Karatza. His research interests include cloud computing and distributed processing.

Highlights

1. Combination of virtual machines and containers, by running containers on top of virtual machines.

2. We highlighted the benefits of combining these two complementary virtualization technologies and we recognized the performance overhead as main drawback.

3. We deployed various experiments and use cases to empirically quantify the performance and energy consumption overhead introduced by the additional virtualization layer of virtual machines, when containers run on top of them.

4. The experimental results showed how different factors such as type of hypervisors, container-centric OSes, applications, system configurations and number of users affect the overall system's performance and energy consumption.