Virtualization Costs: Benchmarking Containers and Virtual Machines Against Bare-Metal

Saverio Giallorenzo, Jacopo Mauro, Martin Gyde Poulsen, Filip Siroky

▶ To cite this version:

Saverio Giallorenzo, Jacopo Mauro, Martin Gyde Poulsen, Filip Siroky. Virtualization Costs: Benchmarking Containers and Virtual Machines Against Bare-Metal. SN Computer Science, 2021, 2, 10.1007/s42979-021-00781-8 . hal-03337920

HAL Id: hal-03337920
https://hal.inria.fr/hal-03337920

Submitted on 8 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SN Computer Science (2021) 2:404
https://doi.org/10.1007/s42979-021-00781-8

ORIGINAL RESEARCH

Virtualization Costs: Benchmarking Containers and Virtual Machines Against Bare-Metal

Saverio Giallorenzo[1,2] · Jacopo Mauro[3] · Martin Gyde Poulsen[4] · Filip Siroky[4]

Received: 16 April 2021 / Accepted: 29 June 2021
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2021

Abstract
DevOps advocates the usage of Virtualization Technologies (VT), such as Virtual Machines and Containers. However, it is complex to predict how the usage of a given VT will impact on the performance of an application. In this paper, we present a collection of reference benchmarks that developers can use to orient when looking for the best-performing VT w.r.t their application profile. To gather our benchmarks in a resource-wise comprehensive and comparable way, we introduce VTmark: a semi-automatic open-source suite that assembles off-the-shelf tools for benchmarking the different resources used by applications (CPU, RAM, etc.). After performing a survey of VTs in the market, we use VTmark to report the benchmarks of 6 of the most widely adopted and popular ones, namely Docker, KVM, Podman, VMWare Workstation, VirtualBox, and Xen. To validate the accuracy of our reference benchmarks, we show how they correlate with the profile performance of a production-grade application ported and deployed on the considered VTs. Beyond our immediate results, VTmark let us shed light on some contradicting findings in the related literature and, by releasing VTmark , we provide DevOps with an open-source, extendable tool to assess the (resource-wise) costs of VTs.

Keywords Virtualization · Benchmarking · Cloud deployment

Introduction

Virtualization Technologies (VT) such as Virtual Machines and Containers are the pillars behind the growth of cloud computing and the adoption of the DevOps approach [1]. VTs opened the way to structure cloud-application portability, scalability, component isolation/security, etc.. Alas, those features come with different levels of overhead on the baseline performance of "bare-metal" deployments, i.e., single-tenant physical servers, which constitute the hardware where VTs execute upon.

Today, there are a plethora of alternatives to choose from when adopting VTs, even when limiting our scope to the mainstream VT categories of (hardware) hypervisors (hardware-supported/native and hosted) [2], and containers [3]. While this vastness of options is good for developers, since it gives them the flexibility to choose the best VT for their application profile, it becomes difficult for them to compare those options over a given set of relevant metrics (e.g., costs, performance). Indeed, the accurate solution for obtaining comparable VT metrics for a given application would entail adapting it to be deployed under the different VTs, benchmark them, and choose the best fitting one. Unfortunately, there are two main problems with this solution:

1. the developers must set their application upon the different VTs. This is a time-consuming activity in two respects: (i) developers must spend time acquiring

"This article is part of the topical collection "New Paradigms of Software Production and Deployment" guest edited by Alfredo Capozucca, Jean-Michel Bruel, Manuel Mazzara and Bertrand Meyer".

✉ Jacopo Mauro
mauro@imada.sdu.dk

Saverio Giallorenzo
saverio.giallorenzo2@unibo.it

Martin Gyde Poulsen
mgp@ordbogen.com

Filip Siroky
phillwide@gmail.com

1 Università di Bologna, Bologna, Italy

2 INRIA, Sophia Antipolis, France

3 University of Southern Denmark, Odense, Denmark

4 Ordbogen A/S, Odense, Denmark

Published online: 07 August 2021

enough knowledge of each technology to allow them to proceed with the setup, (ii) developers have to dedicate time to the setup and optimization of their application under each configuration and later collect the benchmarks;

2. the approach hardly scales, i.e., each application generates a benchmark apart and cannot be directly taken as a reference for other applications. In essence, there is either replication of similar work or measurements that cannot be reliably re-used within the community of developers.

Building towards a general solution, in this paper, we aim to provide a reliable reference that developers can use to orient when looking for the virtualization technology that performs the best—technically, the one that has the smallest overhead to bare-metal—given their application profile.

To gather our reference data, we assembled a semi-automatic suite, called VTmark, to benchmark different VTs reliably—here, the term "reliable" means to have consistent and repeatable measurements among the samples with low standard deviation. VTmark uses a set of renowned off-the-shelf benchmarking tools to collect the performance of the different VTs in a resource-wise comprehensive and comparable way. In developing VTmark, we also aim at the maximal automation possible for its deployment and its execution. Unfortunately, due to the nature of some considered VTs, we could not achieve a fully automatized deployment, e.g., some hypervisor solutions need some manual configuration before the execution of the suite.

Contributions of this paper include:

- the presentation of VTmark, its components, and the rationale that guided our choices in assembling it;
- the report and interpretation of the reference benchmarks obtained from running VTmark over different representative implementations of the three mainstream virtualization technologies of (hardware) native hypervisors, hosted hypervisors, and containers, addressing some contradictions reported in the literature;
- an initial validation of the reliability of the results obtained through VTmark. To do that, (i) we take a production-grade distributed application (the largest Danish online dictionary); (ii) we adapt it to be deployed over the VT implementations we used to collect our benchmarks (the ones in our reference report); (iii) we benchmark the performance of the various adaptations over synthetic-yet-realistic traffic loads and we discuss our results in relation to the reference data obtained through VTmark.

In Section "Preliminaries and Technology Selection", we provide background concepts on virtualization technologies, their popularity, and the list we selected for this study. We then position this contribution by comparing with related work in Section "Related Work". In Section "VTmark", we introduce VTmark, its components, and the relevant technical details on how they are used within VTmark. In Section "Results", we show and comment the results collected by benchmarking a set of VT implementations (presented in Section "Architecture and Supported Virtualisation Technologies") with VTmark. We then present the tests performed with a production-grade distributed application in Section "Case Study", comparing those results with the performance profiles gathered through VTmark. We conclude with the main closing remarks on this work and prospect its future continuations in Section "Conclusion".

Preliminaries and Technology Selection

According to the well-known survey on virtualization technologies by Chiueh and Brook [4]:

> Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, etc.

In this paper, we consider virtualization technologies (VT) as abstractions of the hardware which allows developers to have multiple coexisting running operating environments on one physical server. The abstraction itself can be on different levels and of different types.

In this section, we first briefly report a description of the categories of VTs that we consider mainstream in the context of the current Cloud Computing development landscape.

Virtualization Technologies Categories

There are mainly three categories of virtualization technologies for the Cloud: virtual machines, containers, and unikernels.

Virtual Machines Virtual Machines (VM) represent the first generation of virtualization technologies, with implementations available as early as the 1960s. Thanks to their long-time reliability, VMs are the backbone of the current commercially available Cloud offering. We can practically see VMs as physical servers, with their own complete and isolated operating system. A VM is created and managed by a *hypervisor*, also called VM monitor. There are two main types of hypervisors, which also determine the type (features and performance) of virtual machines we are dealing with [5]: (i) type-1, or *native* hypervisors, which run directly on the hardware (ii) type-2, or *hosted* hypervisors, which run on a host operating system.

Fig. 1 Popularity of type-1 hypervisors

	Stack Overflow	Google Trends	Stackshare stacks	Github stars
VMware vSphere	607	24.1	476	0
Xen Project	428	9.6	25	326
ESXi	428	83.28	0	0
Azure Virtual Machines	2019	0.7	40	10
Hyper-V	1413	35	0	0
KVM	998	45	38	0
Oracle VM	500	7.11	4	0
bhyve	11	0.5	0	15
Virtuozzo	77	0.1	2	0

Since both native and hosted VM solutions are widely adopted in the industry, we consider both types in an equal way in our benchmarks.

Containers Containers find their roots in technological advancements from the 1980s, which brought forward the concepts of process-level segregation file access in operating systems. Containers provide a more lightweight virtualization approach than VMs, because they come as a feature of a given operating system, whose kernel supports the co-existence of multiple isolated user-space instances—called containers. When creating a container, the user defines which resources and devices available to the operating system the container can access.

Since containers run interacting “directly” with the kernel,[1] they have a smaller footprint on performance. On the other hand, the increased performance of containers comes at the cost of a decrease in system flexibility (which is now dependent on the guest kernel functions) and security [6] than VMs.

Unikernels Unikernels are the most recent addition to the category, with proposals dating back to the late 1990s [7–9].

The unikernel approach is that of packaging a given application (e.g., a web server) with the minimal kernel functions and drivers the application requires to run as a sealed, immutable executable—called unikernel—directly on the hardware (or sometimes on a hypervisor).

Thanks to their minimality, unikernels seem a promising compromise between the flexibility of VMs and the performance of containers. Moreover, since unikernels support the execution of one process, they increase isolation—although unikernels run applications in kernel mode, making it unclear how they affect the exposure of the system to attacks and bugs—and provide a simpler model for performance prediction. From the software engineering point of view, while VMs and containers do not impose strong constraints on how programmers develop their applications, unikernels require applications to be written specifically for them [10].

Due to the limitations and early-stage issues mentioned above, there are no relevant commercial options for unikernels. Thus, we omit to consider solutions from this category in this work.

Virtualization Solutions and Popularity

We now present the most popular, state-of-the-art implementations of the categories of VTs considered in this work. To evaluate their popularity, we relied on information derived from Google searches trends at https://trends.google.com, the number of questions on https://stackoverflow.com, the number of stacks on https://stackshare.io, and GitHub Stars (for open source projects).

As a premise, we notice that there are many solutions for each VT category we consider. Broadly, while they share the same denominating features of the category they belong in, they differ in the set of features and performance they grant. For example, minimal solutions encompass the creation, configuration, deployment, and management of containers or VMs, while more sophisticated alternatives add functionalities such as advanced orchestration, scaling, and load balancing of the instances. In benchmarking these technologies, we abstract from the distinguishing features of specific implementations, which we do not address here.

When displaying the data on the popularity in tabular format (cf. Figs. 1, 2 and 3), we sort in descending order the entries in the row based on the sum of the column-marginal difference between the entry value and the maximal value for the column. Formally, let i be the row index of the table t, $c[i]$ the value of of column c in raw i, $\mathtt{max}(c)$ the maximal value in column c, the row score of i in t is obtained as follows.

[1] They use the same system call interfaces of traditional programs and avoid to logically emulate some hardware.

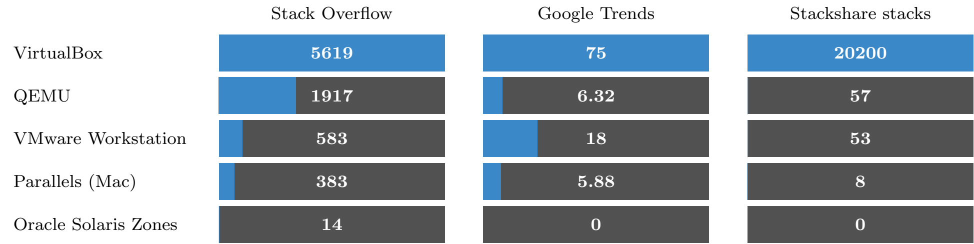

Fig. 2 Popularity of type-2 hypervisors

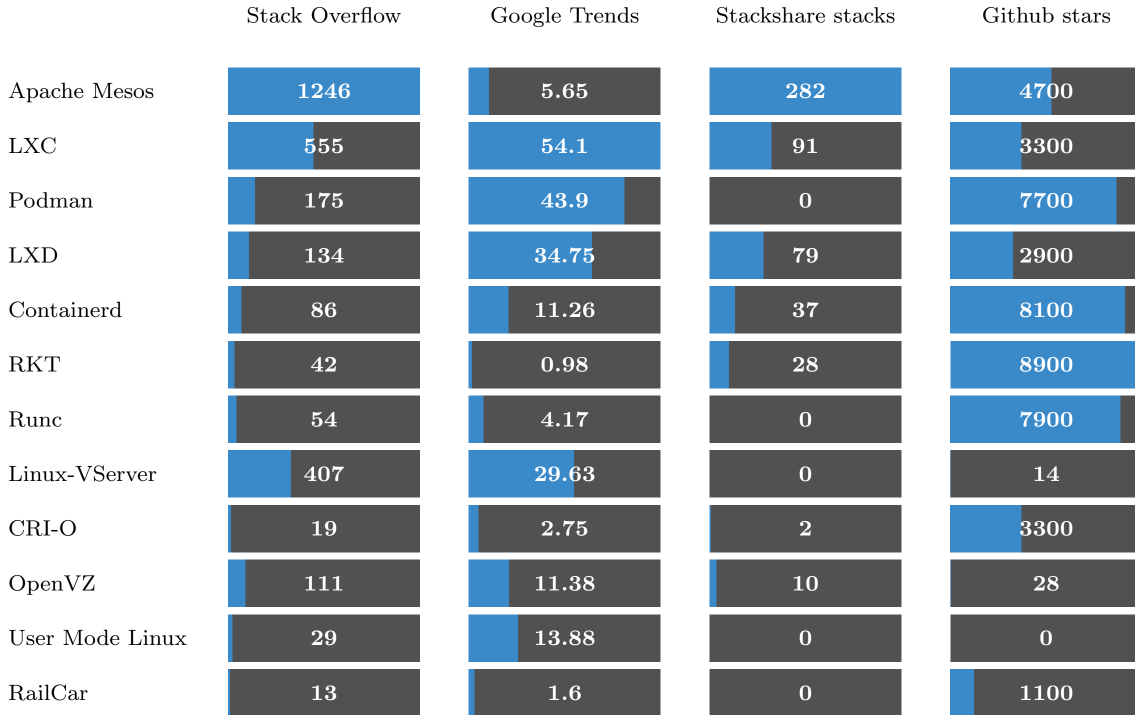

Fig. 3 Popularity of container engines (Docker and Kubernetes excluded)

$$row_score(i,t) = \sum_{c \in \texttt{columns}(t)} \frac{1}{|\texttt{columns}(t)|} * \frac{c[i]}{\texttt{max}(c)}.$$

Native Hypervisors (Type-1)

We report the list of the most popular native hypervisor (type-1) solutions in Fig. 1.[2] From the data, the most popular type-1 hypervisors is VMWare vSphere, followed by Xen Project and ESXi.

Hosted Hypervisors (Type-2)

We report the list of the most popular hosted hypervisor (type-2) solutions in Fig. 2.[3] The most popular type-2 hypervisor is VirtualBox, followed by QEMU and VMware Workstation.

We remark that some technologies in this category can work also as type-1 hypervisors. For instance, QEMU can become a type-1 hypervisor by using KVM [11]. In this work, we consider them in isolation and belonging to the type-2 category.

Containers

[2] Google Trends results are for a 12-month average relative to VMWare ESXi peak popularity between April 2020 and March 2021. XenServer and Xen have been, respectively, renamed into Citrix Hypervisor and Xen Project. For consistency, we refer to these with the latter.

[3] Google Trends results are obtained for a 12-month average relative to VirtualBox peak popularity between April 2020 and March 2021. The results on VMware Workstation combine the three main products that share the same underlying technology: VMware Workstation Player, VMware Workstation Pro, and WMware Fusion. All these projects are not open source and therefore in this plot, we omitted the Github star column.

This category of implementations is indisputably dominated by the de facto standard imposed by Docker [12], which is three orders of magnitude more popular than any competitor, according to Google trends. Similar (on a smaller scale) results come from Stack Overflow (75K threads) and Stackshare (50 K stacks).

Right after Docker (less popular, in absolute terms), we find Kubernetes, which has about 30% of the popularity of Docker, based on Google trends and https://stackoverflow.com.

For completeness and consistency with the previous sections, we report the list of the most popular containers solutions in Fig. 3,[4] excluding the most famous Docker and Kubernetes options above, to avoid skewing the scale due to their disproportionate popularity.

Looking at the chart, we can state that most container engines are Linux based and often tied to one particular distribution.

Technology Selection

We conclude by providing the rationale behind the choices made to select the VT implementations used to collect our benchmark. In summary, we look for the *popularity* and *affordability* of the VT technologies.

Popularity reflects the adoption and interest around a given technology demonstrated by users. The more popular the technology is, the larger is its adoption and the possible audience interested in comparisons including it. There is no consensus on defining popularity, but for our work, we relied on the information derived from Google searches trends, https://stackoverflow.com, https://stackshare.io, and GitHub Stars as reported in the previous section.

With *affordability*, we mean an expression of the ease of adoption that a given technology presents to the general audience of users; e.g., whether it is a free and open-source solution or rather comes with paid licensing or subscriptions and/or platform lock-ins. We deem more affordable solutions that are open source rather than closed source, solutions that are free rather than paid, and solutions that entail the fewest lock-ins. For example, if there are no relevant open-source projects in a given technology segment, the next most affordable options are closed-source projects that offer free/community versions of their solutions. Moreover, as an extension of the lock-in effect, when standards (proper or de facto) exist, we consider more affordable those solutions compatible with them. Similarly, when comparing closed-source options, we consider as most affordable those that users can deploy on their private infrastructure rather than requiring them to use some third-party infrastructure.

Our goal was to test at least one technology for every virtualization modality. Following the principles above we identified the following list of implementations.

Type-1 Hypervisors *Xen/Xen Project*. Since VMware vSphere is a closed-source project, we consider the more affordable second best solution, i.e., Xen Project—which is open source and has no dependencies on a specific platform. In the remainder, for brevity, we use "Xen" to refer to Xen Project.

Type-2 Hypervisors *Virtualbox*. Here, Virtualbox is the undisputed popularity and affordability "winner".

Containers *Docker*. As reported in the related section, Docker is by far the most renowned and affordable technology for containers.

Given the above choice of technologies, we also looked for comparable alternatives for each of them:

Type-1 Hypervisors *KVM*. Similarly to our choice for Xen, we orient our alternative for type-1 hypervisors mediating popularity with affordability. Hence, we exclude ESXi (a closed-source project by VMware) and Azure Virtual Machines and Hyper-V (tied to the Microsoft Azure platform). Hence, we consider *KVM* as the second-best type-1 hypervisor given its popularity and affordability—it is an open-source project and it does not dependent on a specific platform.

Type-2 Hypervisors *VMware Workstation*. Although QEMU appears to be the second most popular type-2 hypervisor, QEMU is a complex platform that can be used as a type-2 hypervisor but also configured to work as a type-1 hypervisor. For this reason, we discarded QEMU and selected VMware Workstation, which is the third most popular type-2 hypervisor—besides being a technology often compared in the literature.

Containers *Podman*. Regarding containers, in addition to Docker, we select Podman. The reason for this choice is that both the second and third most popular solutions—Kubernetes and Apache Mesos—are container-orchestration technologies rather than strictly container engines. The following entry, LXC, would be valid (it is a popular open-source project); however, its incompatibility with the Docker format, a de facto standard for container specification, makes it less affordable than the next-in-line, Podman, which adopts that standard.

[4] Google Trends results are for a 12-month average relative to LXC peak popularity between April 2020 and March 2021.

Table 1 Virtualization technologies used in related work and this paper

Technology	Soltesz et al. [14]	Barham et al. [13]	Mavridis et al. [15]	Granisze-wski et al. [16]	Langer et al. [17]	Morabito et al. [5]	Scheepers [18]	Sharma et al. [19]	Chae et al. [20]	Felter et al. [21]	Kamarainen et al. [22]	Barik et al. [23]	Li et al. [24]	**This paper**
Bare-metal														
	•	•	•			•				•	•		•	•
Type-1 Hypervisors														
Xen	3	<1	•		•									4.11.4
XenServer				6.1			•							
KVM			•		5.4	2		•	•	•	•			3.1.0
Hyper-V			•	•										
OVM				3.2.7										
ESXi				5.1	4.0									
Type-2 Hypervisors														
VMWare		F3.2			P7								F	15.5.6
VirtualBox				4.3.6	3.1.2							•		6.0.16
Containers														
Docker						1.3.2			•	•	•	•	•	19.03.8
Podman														1.7.0
LXC						1.0.6	•	•						
VServer	•													
UML		•												
Year Pub.	07	03	18	16	11	15	14	16	17	15	15	16	17	

Related Work

In this section, we position our work among other related contributions in the literature on benchmarking and comparisons of existing virtualization technologies.

We schematically represent the technologies considered in the relevant related work in Table 1. In the table, when a cell is marked with the sign •, we mean that the authors benchmarked that software but did not report which version they used; when the version is available, we replace • with that number. For VMware, if indicated by the authors, we also report whether they used the pro (P) or the free (F) version of that software. The last row with the title "Year Pub." indicates the year of the publication (last two digits after the year 2000).

From the table, the only other work comparing bare-metal, type-1 and -2 hypervisors, and containers is [13], which was published more than 15 years ago, barely after the creation of the first public clouds. Moreover, our paper is the only work that includes in its comparison Podman.

In addition to the data shown in the table, we report the fact that only 15% of those works included scripts used for running the benchmark (towards assembling a suite like VTmark), and only 8% listed the benchmark arguments, as we do here.

In the remainder of the section, we report the main findings summarized from the surveyed papers and divided by VT categories.

Native Hypervisor (Type-1) Performance

There is a common agreement on the performance impact of Type-1 Hypervisor on CPU and memory, which is reported as small for a single VM instance. However, when scaling, the overall performance drops significantly in contrast to containers. As far as disk performance is concerned, the performance drops somewhere between 10 and 50% depending on the different IO operations.

There are significant differences between the reported results, sometimes leading to contradicting outputs. For example, Soltesz et. al. [14], when comparing Xen against bare-metal using the synthetic benchmark DBENCH [25], found out a performance drop of 79% while Barham et al. [13], on the same setting, report a drop of performance of only 3%.

This huge difference might be due to Barham et al. working with a prototype and the overhead reported by Soltesz et. al could have been introduced later due to the Xen technology development.

From the majority of the work, native hypervisors seem to affect network performance. Mavridis et al. [15] and Barham et al. [13] report nearly identical throughput to bare-metal, when the CPU utilization is not stressed. The data suggest that type-1 hypervisors require more CPU to process packets, which is also supported, e.g., by the TCP_STREAM benchmark performance in [15]. Network performance seems to have significant overhead and depends mostly on the available CPU processing power to achieve the same throughput or response rate of bare-metal.

Note also that Mavridis et al. [15], benchmarking TCP and UDP request-response performance found out that KVM has almost the same performance as bare-metal. That report contradicts the results obtained by Morabito et al. [5] that also run the same comparison and reported a 43% performance drop between KVM and bare-metal.

Hosted Hypervisor (Type-2) Performance

The performance overhead of type-2 hypervisors vary significantly among reports in the considered related work.

In particular, CPU performance varies from paper to paper and among benchmarks. Li et al. [24] report a performance drop of only about 4.5% on average. Barham et al. [13] report no significant performance impact for the SPEC INT2000 benchmark[5] and 20% performance decrease for the Linux kernel compilation time. Barik et al. [23] report 45% for building Apache and 55% for building PHP in comparison to Docker performance. Lastly, Graniszewski et al. [16] report an average 22.5% performance drop against type-2 hypervisors in nbench. For kernel compilation, Graniszewski et al. state the performance drop is about 40% in comparison with type-2 hypervisors.

Memory performance drops seem small. The consensus is that, similar to type-1 hypervisors, the drop is smaller than 5%.

Disk operations suffer instead a performance drop of about 30% based on the results from the Bonnie++ benchmarks [26] reported by Li et al. and the DBENCH benchmarks [25] reported by Barham et al.. Contrarily, Graniszewski et al. [16] showed that VirtualBox doubled the performance of any of the type-1 hypervisors that they benchmarked. For web-server and mail-server benchmarks, the performance is usually less than 50% of the type-1 hypervisors.

From the reports, also the network suffers a significant performance overhead. Graniszewski et al. [16] netperf[6] reported the same or worse performance than Type-1 hypervisors, and Li et al. [24] using Iperf (TCP) reported a 55% performance drop. Barik et al. [23] measured a 33% performance drop for TCP against container with tbench.

[5] http://www.spec.org/.

[6] https://hewlettpackard.github.io/netperf/.

Table 2 Synthetic benchmarks used in VTmark

Benchmark	Resource	Version	Options
Dhrystone	CPU	2.2	2147483627
Whetstone	CPU	1996	5,000,000
Kcbench	CPU	0.4.1	
RAMspeed	RAM	3.5.0	-b 1...19 -g 32 -l 5
iPerf3	NET	3.6	-O 10 -i 60 -t 180 -b 0
			-O 10 -i 60 -t 180 -b 0 -b 1000000000
			-O 10 -i 60 -t 180 -b 0 -u
			-O 10 -i 60 -t 180 -b 0 -b 1000000000 -u
Bonnie++	Disk	1.98	-q -r 16000 -z 1580230295 -x 3 -n 2500 1000
DBENCH	Disk	4.0	-t 180 1/2/4/8 [-D /dbench]
			-t 180 1/2/4/8 [-D /dbench] -F
			-t 180 1/2/4/8 [-D /dbench] -s

To summarize, type-2 hypervisors are expected to have a significant impact on CPU, disk, and network performance in certain situations, while memory performance should not have significant drops.

Containers Performance

As expected, all reports agree that containers have a small impact on resources. The CPU and memory performance are almost unaffected: sometimes, the overhead is only 1%. More surprisingly, there are cases reporting containers performing better than bare-metal. Morabito et al. [5], for example, report a 1% improvement for Docker and LXC over bare-metal in Linpack benchmarks [27].

Soltesz et al. [14], and Morabito et al. [5] report the same performance impact of using containers on TCP stream. The performance of UDP streams and TCP and UDP request-response rates seem to be significantly impacted according to the results from Morabito et al. In particular, the UDP stream benchmark resulted in about a 40% performance drop while the request-response rate for TCP and UDP, respectively, are about 20% and 10% lower than the response rate of bare-metal. Li et al. [24] present contradicting results, showing only a 2% overhead.

Disk performance follows a similar trend to those of the network: they vary among operations and their drop can range from 7 to 50%. The considered related work report different amounts of overhead: Li et al. [24], using Bonnie++, recorded a 40–50% data throughput decrease, Morabito et al. [5], using Bonnie++ and dd, report a 7% performance drop for sequential reads and writes while for random writes and seeks the performance drop ranges between 10 and 50%. Soltesz et al. [14] contradict the results obtained from dd, Dbench, and Postmark benchmarks, where VServer performs better than bare-metal in DBENCH and Postmark and the same in dd.

Overall, containers seem to have a small impact on resources and scale well. The only exception is disk performance, where the overhead seems to vary.

VTmark

In this section, we present VTmark, a semi-automatic benchmarking suite to run a set of synthetic benchmarks over a target computational resource.

In the following, we give a summary of the synthetic benchmarking tools included in VTmark, we detail our architectural choices to (semi-)automatize the deployment and execution of VTmark, and describe the virtualization solutions selected in Section "Preliminaries and Technology Selection" as targets of this study.

Benchmarking Toolset

Inspired by the related literature (cf. Section "Related Work"), VTmark includes some of the most widely used benchmarking tools freely available on the main Linux distributions, which we briefly describe below. As a summary, Table 2 shows all the adopted tools, with their launch parameters used in VTmark.[7]

Dhrystone Dhrystone [28] was created in 1984 as a single-threaded program for measuring CPU performance. Initially written in ADA, it was later translated into C. In VTmark, we use version 2.2 (adjusted to use the current version of C compiler libraries) and we execute this benchmark 5 times to normalize its results. To achieve reliable results,

[7] Note that the `-D` option for DBENCH was used for container engines with volumes. When benchmarking iPerf3, we additionally specify the IP address and the port of the server we benchmark with the options `-c <ip>` and `--cport <port>`.

Dhrystone should be executed many times. The tool documentation suggests running at least 500 million iterations. We decided to have 2,147,483,627 repetitions.[8]

Whetstone Whetstone [29] measures floating-point operations performance and was written initially in Algol 60 in 1972 and rewritten in Fortran in 1973. The Fortran version was the first general-purpose benchmark setting the industry standard of performance [30]. The C version was written in 1996 and is available at http://www.roylongbottom.org.uk/whets.c. In VTmark, we adjusted the code (C version) and set a fixed number of passes to 5,000,000 to get consistent results. A pass for Whetstone is the execution of the original benchmark that is now required to accommodate for the introduction of faster CPUs.

Kcbench Kcbench[9] is a benchmark that measures the time needed for compiling the Linux kernel, monitoring the CPU utilization during the compilation. VTmark uses version 0.4.1 from Kcbench's GitHub repository. The benchmark compiles Linux 4.14 kernel three times by default.

RAMspeed RAMspeed [31] is a cache and memory benchmark for several UNIX-like operating systems. One can use it to benchmark integer, floating-point, MMX, and SSE instructions in two types of benchmarks: *mark benchmarks that pass sequential data streams to the relevant hardware units, and *mem benchmarks to measure the speed of simulated Copy, Scale, Add, and Triad tests[10] with the support for a batch run returning the average speeds. VTmark uses version 3.5.0 from http://www.alasir.com/software/ramspeed, setting the number of gigabytes per pass to 32 with `-g 32` and BatchRun mode with `-l 5`.

iPerf3 iPerf3 [32] is a network throughput benchmark for testing TCP, UDP, and SCTP. The binary contains a server and a client: the client sends data to the server and the server responds to the former. The benchmark provides reports from both the server and the client. We used the Debian Buster package. VTmark performs several benchmarks with iPerf3, setting the duration of the test to 3 min (`-t 180`) but discarding the first 10 seconds (`-O 10`) needed to warm up the system and to reach stable speed readings.

Bonnie++ Bonnie++ [26] is a benchmark that performs a set of simple tests on the file system and hard drive to measure their performance. The tests range over database-like access to one or more files, as well as creation, reading, deletion, and random seeks. VTmark uses the Bonnie++ Debian Buster package and specifies the amount of RAM (`-r 16000`), a seed for randomized operation (`-z 158230295`), the number of tests to 3 (`-x 3`), and the number of samples (`-n 2500`). In particular, among all the Bonnie++ benchmarks, we have considered the sequential output and sequential input micro-benchmarks testing the per-character, block input and output operations, and the output rewrites.

Table 3 Supported virtualization technologies in VTmark

Technology	Version	Debian Package
Xen	4.11.4	Y
KVM	3.1.0	Y
VirtualBox	6.0.16	Y
VMWare Workstation	15.5.6	N
Docker	19.03.8 Community	Y
Podman	1.7.0	N

DBENCH DBENCH [25] is an IO workload benchmarking tool that can be run for a desired period of time, using multiple processes to simulate concurrent users. VTmark uses DBENCH from the official Debian Buster repository. The benchmark is set to run for 3 min (`-t 180`) with 1, 2, 4, and 8 processes with no extra synchronization, fsync (`-F`), and O_SYNC (`-s`) options.[11]

Architecture and Supported Virtualization Technologies

Architecture and Bare-Metal Baseline As baseline comparison for the considered virtualization technologies, VTmark can be installed on Debian-based bare-metal servers. To streamline its execution, we recommend to pair VTmark with Openstack-baremetal, which helps in automating operations over bare-metal servers. Indeed, Openstack [33] is a cloud operating system that enables the creation of bare-metal machines, containers, and virtual machines over a pool of servers that it manages. It enables its users to manage storage, compute, and network resources through APIs and web interfaces with standard authentication. VTmark uses the state-of-the-art data collector Influx DB[12] to capture the benchmark resource consumption. In particular, it captures all types of CPU usage (system, user, IO wait, interrupts, etc.) for every core, disc reads and writes, total disc utilization as well as swap, and memory and network usage. The

[8] Nearing the maximal number of iterations reachable by a 32-bit integer counter.

[9] https://gitlab.com/knurd42/kcbench.

[10] Triad is a combination of Add and Scale operations: it reads data from the first memory location, scales it, then adds data from the second one and writes to a third place.

[11] Note that DBENCH does not impose requirements on the number of CPUs. Indeed, when 8 processes are run, these processes can be run concurrently also on only 1 CPU.

[12] https://www.influxdata.com/.

captured data are then visualized by Grafana,[13] a state-of-the-art technology for metrics visualization.

We now list the virtualization technologies supported by VTmark, which we summarize in Table 3. In the following, we detail also how VTmark deploys them.

Type-1 Hypervisors: Xen Xen (Project) is an open-source type-1 hypervisor [34]. The Xen hypervisor uses a microkernel design to have a small footprint and interface. It manages CPU, memory, and interrupts. IO operations and networks are managed by the control domain (the host). In VTmark, Xen packages are installed on an OpenStack bare-metal server. The guest OS is based on a file-based image in the host operating system using the sparse image option. The domain also has limited CPU and memory.

Type-1 Hypervisors: KVM KVM stands for Kernel-based Virtual Machine [35] and comes as a virtualization module of the Linux kernel to allows the latter to act as a hypervisor. KVM is open source and requires processors with hardware virtualization instructions to provide hardware-assisted virtualization and para-virtualization. Similar to Xen, in VTmark KVM packages are installed on an OpenStack bare-metal server. The user has to pass the image of the VM of the guest operating system to the KVM installation scripts that creates it. After this step, the user goes through the standard Debian installation.

Type-2 Hypervisors: Virtualbox Oracle VM VirtualBox is a type-2 hypervisor supporting the Windows, macOs, Linux, and Oracle Solaris operating systems. Virtualbox supports many hardware features, guest multiprocessing, running different operating systems, and managing VMs through both a graphical and command-line interfaces [36]. Oracle provides a Debian package for VirtualBox in its repository. For its setup, VTmark installs the additional package `xinit` and `fluxbox` to let its user access the VirtualBox GUI and set up the image manually (it was used to install the OS and set up an IP address[14]). Concretely, VTmark facilitates the download of a standard Debian installation image and it allows users to create a VM with the required settings. This VM is then used to run the benchmarks automatically.

Type-2 Hypervisors: VMWare Workstation VMWare Workstation (both the Pro and Player versions) is a type-2 hypervisor for Linux and Windows. VMWare Workstation supports high-performance 3D graphics, virtual networking, duplication, and snapshots and comes with a REST API for VM automation. For deploying VMWare Workstation in VTmark, we extended an already-existing Ansible role[15] to install additional packages (viz. libpcsclite1, linux-headers-4.19.0-6-amd64, libxml2, and policykit-1). The user of VTmark needs to set up the VMWare VM using the GUI as done for VirtualBox and deploy the benchmarks.

Containers: Docker Docker is an open-source container engine. It acts as a server–client application with the Docker daemon running in the background, and its command-line interface acting as clients. Docker builds on the containerd daemon, which relies on OCI-compliant container runtimes [37]. In VTmark Docker is installed from the official Docker repository. The containers are created based on the Ansible playbooks for deploying the benchmarks. In particular, VTmark uses the standard Docker Debian 10 image.[16]

Containers: Podman Podman is an open-source, daemonless container engine for OCI containers on Linux systems. The commands for Podman management are compatible with Docker, thus allowing a smooth transition between the two engines. It supports both OCI and Docker image formats and has support for managing groups of containers together within Pods [38].

Podman has no Debian package. VTmark deploys Podman using an Ansible role for installing it with all its dependencies. Thanks to the compatibility with Docker, VTmark reuses the scripts to build Docker images to also build the Podman images.

Using VTmark

VTmark considers as its target operating system Linux, and it is specifically programmed to run on Debian-based distributions—indeed, considering the main distributions of Debian and Ubuntu alone, ca. 65% of the whole market share of Linux-based web servers[17] are suitable to run VTmark. VTmark requires as minimal hardware resources 2 servers, one used for monitoring, with a recommendation of 2–4 cores and 2–4 GB of RAM, and the second one for the actual benchmarking. For the benchmarking, we recommend 8 cores, 16GB of RAM, and a gigabit network adapter.[18] The two servers need to be configured with Debian, preferably on the same network, and the IP addresses of the virtual machines and the containers must be reachable. We recommend managing those two servers with, e.g., OpenStack [33]—as did in our benchmarks—which provides features that help to automate the shift between different

[13] https://grafana.com/.

[14] Note that if a DHCP server is available, this procedure could be automatized. However, we did not want VTmark to rely on the presence of a DHCP server and for this reason we kept VirtualBox as a semi-automatized target in VTmark.

[15] https://github.com/vmware-archive/ansible-role-vmware-workstation.

[16] https://hub.docker.com/_/debian.

[17] https://w3techs.com/technologies/details/os-linux.

[18] Benchmarking with a less powerful server is possible but, in this case, some benchmark arguments might need adjustments.

virtualization technologies. To run the configuration scripts of VTmark, a workstation with installed Ansible 2.8 or newer version [39] is required.

The deployment of the monitor server is automated through Ansible, which requires the definition of the server IP address to the `telegraf_influxdb_instance` in the configuration file `ansible/group_vars/all/vars.yml` and into the `monitor host` in `ansible/inventory.ini`. Additional configuration options like the timezone information are available in `ansible/group_vars/all/vars.yml` and in `ansible/ansible.cfg`.

After setting up these parameters, the monitor can be deployed by running the following command.

```
ansible − playbook − iinventory.inimonitor.yml − lmonitor
```

When the monitor is deployed, it is possible to proceed with the deployment of the virtualization technologies to benchmark. As done for the monitor, the IP of the server in which to deploy the virtualization technology needs to be added into the `ansible/inventory.ini` file. As an example, in the following, it is possible to see a snippet of the `ini` file for KVM.

```
[kvm_host]
openstack-kvm ansible_host=10.0.0.1 ansible_become=true
```

To deploy the host roles, it is possible to execute the following command

```
ansible − playbook − iinventory.iniphase1.yml
−l < technology > −host,
```

where < technology > is the virtualization technology to test (i.e., among: `baremetal`, `openstack-baremetal`, `xen`, `kvm`, `virtualbox`, `vmware-workstation`, `docker`, and `podman`).

Bare-metal, Openstack bare-metal, Docker, and Podman do not require any further steps. Unfortunately, it was not possible to fully automatize the deployment for Xen, KVM, VirtualBox, and VMWare Workstation, which require the execution of a few manual steps for the creation and configuration of the VMs[19]. After these manual steps are executed, to finalize the deployment the following command should be invoked

```
ansible − playbook − iinventory.iniphase1.yml
−l < technology > −guest
```

Table 4 Idle performance

Benchmark	Memory (%)
Bare-metal	1.3
OpS bare-metal	1.6
Xen	2.4
KVM	6
VirtualBox	7.4
VMWare Workstation	2
Docker	2.5
Podman	1.8

after specifying the IP address of the virtual machine created in the inventory (the technologies to consider here that replace the < technology > place holder are `kvm`, `virtualbox`, `vmware-workstation`, and `xen`).

When the virtual machines and containers are deployed, the benchmark activity starts by running the `vtmark_executor.py` to run the benchmarks.[20] This will start to execute the benchmark in sequence and collect the results, which will be saved in the same directory as the script. The results are split based on the benchmark and every benchmark is saved in its own folder. The results can then be parsed with the utility script `vtmark_parser.py` that prints them into the standard output in comma-separated values format or alternatively into Latex tables.

Detailed instructions on how to deploy, run, and visualize the results of the benchmarks are available at https://gitlab.com/phillwide/vtmark/-/blob/master/PARSER.md.

Results

In this section, we present the reference benchmarks we obtained by running VTmark on a production-grade computation cluster. The cluster is composed of Supermicro SuperServer 5039MS-H8TRF nodes with Intel Xeon E3-1240 v5, 16GB 2133MHz RAM, Intel SSD DC S3520 Series 150GB and gigabit network adapters.

Settings Before presenting our results, we report information on the general setup of the benchmarks. When not using standard settings for a given technology, we also include their configuration and the reason for the deviation.

The version of Debian we used was Debian Buster with kernel 4.19.0. The image for OpenStack is a generic Debian cloud image with the same kernel. The cloud image had

[19] The interested reader can find those instructions at https://gitlab.com/phillwide/vtmark/-/blob/master/DEPLOYMENT.md.

[20] When more computational resources are available, VTmark can deploy and run in parallel all the benchmarks, except iPerf3 which, for reliability, needs the dedicated usage of the network to avoid interference among multiple instances running in parallel.

Fig. 4 Results in terms of percentage w.r.t. bare-metal from Dhrystone, Whetstone, Kcbench, RAMspeed, and Iperf3

	Dhystone	Whetstone	Kcbench	RAMspeed	Iperf3
OpS bare-metal	100%	100%	100%	101%	99.8%
Xen	92.3%	92.2%	80.7%	105.5%	99.4%
KVM	99.6%	99.8%	95.9%	100.4%	99.1%
VirtualBox	99.2%	92.5%	92.7%	95%	75.2%
VMWare	101.4%	101.1%	92.5%	103%	78.2%
Docker	100.1%	99.9%	97.7%	101%	100%
Podman	102.2%	102.5%	98%	103%	100%

to be adjusted by disabling the `discard` option from `/etc/fstab` root partition, as it was degrading disk performance. The benchmarks were run on a single node, virtual machine, or container according to the target virtualization technology.

For Xen, the server instance has 14GB of RAM, eight virtual cores and PyGrub for bootloader. The other settings are the default. Networking works through a network bridge. The hypervisor is pinned to a single CPU, has 2GB of memory and disabled autobaloon.

For KVM, the VM has 14GB of memory, eight virtual cores and uses the host CPU configuration, the VM image is not sparse and the virtualization is full. Networking works through a network bridge.

For VirtualBox and VMWare, considering also the settings used for the Xen server, both images were created with the default settings, a bridged network, 14,163 MB of RAM, and eight cores. For VirtualBox, we tested disk performance both having the drive set SSD and HDD in the storage settings. We used the default controller (SATA) and also the SCSI controller for a pre-allocated (static-size disk) and bridged network. For VMWare, the system disk of the VM was pre-allocated to avoid the performance drop of resizing the image, and we used the SCSI controller (recommended) and the VM in "headless" mode (i.e., an image without the GUI as it is not needed for running the benchmarks).

We now report and comment on the results of the benchmarks, starting from the idle performance of the system and then passing to compare the technologies by running VTmark and commenting the results from each tool in its set.

Idle State

Table 4 shows the RAM usage (in percent) in idle state for each of the targets. The memory consumption of VirtualBox and VMWare Workstation is calculated by adding the used memory from the VM and the memory on the host without the VM.

The resource consumption differs in a small percentage of RAM. We do not report the usage of CPU, network, nor disk as they are close to 0 for all technologies.

Dhrystone We report the results relative to bare-metal from Dhrystone in the first column of Fig. 4. The performance difference is minimal (under 3 %) with the exception of Xen that shows a performance degradation of 8%.[21]

Interestingly, there are virtualization solutions that perform even better than bare-metal. Podman, for instance, consistently outperforms bare-metal in this benchmark (2.2% performance increase). This is a surprising result which could be explained by the different Debian image used within the container and the Debian image used by bare-metal. Unfortunately, due to the complexity of the setting, we were not able to pinpoint the exact cause of this performance difference.[22]

As expected, OpenStack bare-metal achieves almost an identical performance to the one of bare-metal. As far as type-2 hypervisors are concerned, we notice that they cause more context switches and interrupts, but this does not seem to correlate to lower performance in the Dhrystone benchmark.

The performance of container engines matches the results of Morabito et al. [5]. Our results for type-2 hypervisors also match with what Barham et al. [13] reported for SPEC INT2000 (an alternative CPU integer benchmark). Our results for type-1 hypervisor performance is instead lower than expected, which might be caused by the system image being file-based—this matches the observation of the authors of the Xen Project, who mentions a degradation of performance for file backend storages[23]—although it is not clear whether the CPU performance is also affected, besides disk IOs.

Whetstone

As shown in the second column of Fig. 4, the results from running Whetstone are similar to those of Dhrystone. Podman performs best for both versions again. Most of the

[21] Similar results can be obtained when compiling the benchmark with the register storage class option (`-DPASS2 -DREG`) or with the optimization option (`-DPASS2 -O`).

[22] Docker shares the same image with Podman, but differently than Podman, it has a performance aligned with bare-metal.

[23] https://wiki.xenproject.org/wiki/Storage_options.

	Input		Output		
	Per Char	Block	Per Char	Block	Rewrite
Bare-metal	100% (1.39)	100% (0.94)	100% (1.01)	100% (3.23)	100% (2.17)
OpS bare-metal	99.3% (1.66)	102.3% (2.00)	99.7% (0.88)	101.2% (2.33)	102.6% (1.55)
Xen	34.8% (0.64)	108.7% (1.23)	54.1% (0.74)	94.8% (6.02)	104.7% (2.70)
KVM	96.9% (1.47)	98.4% (1.69)	93.8% (8.30)	100.2% (0.70)	106.9% (2.19)
VirtualBox	95.8% (1.58)	104.8% (1.75)	96.6% (0.45)	88.3% (2.95)	93.6% (0.88)
VMWare	81.2% (12.76)	48.6% (6.25)	94.2% (3.59)	90.4% (12.45)	64.6% (0.90)
Docker	58.7% (2.05)	101.5% (0.83)	61.3% (0.40)	99% (0.37)	100.8% (1.56)
Podman	59.2% (0.22)	100.9% (2.44)	60% (0.34)	100.9% (2.63)	99.1% (2.46)

Fig. 5 Percentage results w.r.t. bare-metal for the sequential input and output benchmarks of Bonnie. The relative standard deviation in percentage of the measures is added in brackets

technologies perform almost the same way as bare-metal. The only exceptions are VirtualBox and Xen, which perform significantly worse.

The resource consumption is the same as for the Dhrystone benchmark. The results generally follow our expectations, except for the differences between VirtualBox and VMWare Workstation. VMWare Workstation performs well even with floating-point operations, but VirtualBox falls behind. Xen has the same performance overhead in this benchmark as it did in Dhrystone.

Kcbench

Kcbench is a multi-threaded benchmark and, as shown in the third column of Fig. 4, it helps in highlighting the differences between the technologies studied w.r.t. the two previously benchmarks. Here, only OpenStack bare-metal and bare-metal perform the same, followed by the container engines with a performance drop of about 2.5–3%. The type-2 hypervisors have an overhead that results in about 8% lower performance. KVM is slightly better than type-2 hypervisors with a performance drop of 4% while Xen drops to 81% of bare-metal. The benchmark in this case utilizes 100% of CPU, uses around 5% of RAM, and under 1% disk IO.

Differently from Barham et al. [13], who report a 20% performance drop while compiling Linux kernel 2.4.21 for VMWare Workstation, we saw only an 8% decrease. Barik et al. [23] also reported a bigger difference between the performance of Docker and VirtualBox. Soltesz et al. [14] found no performance difference among bare-metal, Xen, and VServer when compiling the Linux kernel for single-core guests. They found a 7% performance drop for Xen when running SMP guests, where we found 19%. We conjecture that these differences are probably due to minor configuration differences among the various parameters involved in deploying the VTs.

RAMspeed

The fourth column of Fig. 4 reports the average RAM performance using RAMspeed. We compute the results by averaging the speeds of copy, scale, add, and triad operations of different memory types over five runs.

With the exception of VirtualBox, there is no performance drop for any of the technologies. All technologies achieve up to 6% better results than bare-metal, and mostly they do 11% better in the individual memory operation types. Xen performs best here with a 6% better performance than bare-metal. This surprising result may be due to the way the RAM is accessed by the various virtualization technologies and by the fact that RAMspeed does not stress the system too much, i.e., RAMspeed used only 25% of the CPU and about 1% of RAM for any virtualization technology without affecting other resources.

The results from RAMspeed match what we expected based on the related work. Even the 5% performance drop was observed for type-2 hypervisors.

Iperf3

We report in the fifth column of Fig. 4 only the UDP performance benchmarks obtained by running Iperf3, since their TCP counterparts are very close.

OpenStack bare-metal and container engines suffer no performance impact. Type-1 hypervisors have a limited performance impact of less than 1% while type-2 hypervisors instead achieve a significant decrease in performance of more than 20%.

Our bandwidth results for type-2 hypervisors are similar to the results of Barik et al. [23]. Container engine performance follows the trends from Li et al. [24], but are different from Soltesz et al. [14] and Morabito et al. [5]. Xen performance is not impacted, differently to what reported by in Soltesz et al. [14]. From the related work, we expected the performance might be linked to the CPU speed. This is supported by our results that report a performance drop when type-2 hypervisors use only one CPU core. Because iPerf3

needs less computation power to generate TCP traffic, the performance impact was not visible for the TCP benchmark.

Note that Iperf3 does not detect the limits correctly inside containers. In this case, they generate up to 4 times more data, which is then limited by the network interface. In the data reported in Fig. 4, for coherence, we set limits to network traffic for all the tested technologies.

Bonnie++

Figure 5 reports the results of sequential-outputs and -inputs of Bonnie++. Among all the benchmark used in this report that did not require setting the time window explicitly, Bonnie++ was the only one having a high variability, i.e., presenting a standard deviation greater than 2%. For this reason, in Fig. 5 we reported also not only the comparison of performance w.r.t. bare-metal but also the relative standard deviation in percentage of the measures obtained by running the benchmark 3 times.[24]

As we have seen previously for the CPU utilization, RAM memory, and network, OpenStack bare-metal has no performance impact for Bonnie++ sequential-outputs and -inputs.

Type-1 hypervisors and the container technologies have no significant degradation of performance w.r.t. bare-metal for block operations and rewrite output operations. The performance difference is no lower than 6% and, surprisingly, for the input-block benchmarks, they can achieve slightly better performance of bare-metal. In the per-character benchmarks, Xen has a significant degradation (almost half of bare-metal) and also the container technologies undergo performance degradations of around 40%.

VirtualBox is consistent with all the disk and controller settings, and thus we report in Fig. 5 only the results obtained with the SCSI controller. The per-character writing and reading and rewrite speeds are around 5% lower. The block writes benchmark performance is around 15% lower than the one of bare-metal while instead in the block reads it outperforms bare-metal by 2%. Compare to VirtualBox, the rewrite and block-read performance for VMWare Workstation drops by around 35% and 50%, respectively. Also the per-character write performance drops by around 10%. VirtualBox host uses 10% of CPU while its guest uses 5% CPU for system and 20% IO wait. VMWare Workstation uses significantly more CPU. The host peaks at 50% system usage and 15% IO wait, while the guest peaks only at 10% and 20% IO wait. On the other hand, Xen's control domain CPU utilization consists mostly of only IO wait (up to 20%). Xen guest peaks at 10% CPU utilization and 20% IO wait.

Compared to the literature, our results show VirtualBox outperforming the VMWare hypervisor, while Li et al. [24] reported the opposite observation. The swap in position could be explained by new releases of the software that either increased the performance of the former or decreased the ones of the latter. Another possible explanation could be the high variability of the performance of the VMWare Workstation with a standard deviation superior to 5% for some sub-benchmarks. Xen's performance is worse than expected for small operations, but block operations and rewrite results for both Xen and KVM are way better than intuitively expected.

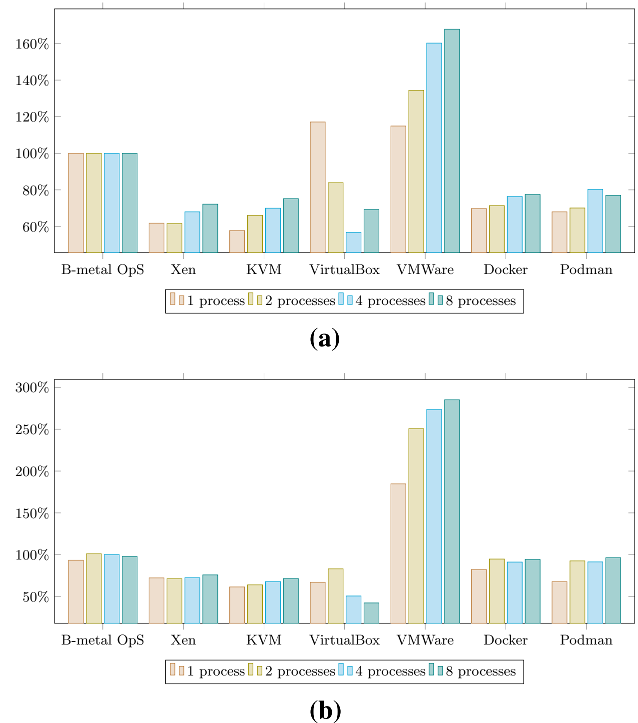

Fig. 6 DBENCH results: **a** plain DBENCH and **b** DBENCH FSYNC percentage w.r.t. bare-metal

DBENCH

We report in Fig. 6 the results from the DBENCH benchmark: the default load-file using different amounts of processes without synchronization (plain DBENCH in Fig. 6a) and with the fsync flag (DBENCH FSYNC in Fig. 6b) that makes DBENCH perform a fsync() to the file after each write operation.[25]

Here, the first surprising result is that VMWare Workstation consistently outperforms all other virtualization technologies and bare-metal. The synchronization option improves the performance of VMWare Workstation, which leads almost-three-times better performance than bare-metal.

[24] We excluded from our benchmarks the results of sequential-creates that we deemed too volatile and subject to significant performance differences.

[25] Note that similar setting instead the O_SYNC flag that makes DBENCH override the loadfile and use O_SYNC for all file open operations leads to similar results.

This outcome is puzzling and we are unsure if the hypervisor manages to properly write the data into the disk while executing the benchmark, or if the technology would lead to a greater data loss on an unexpected shutdown.

OpenStack bare-metal shows mostly no significant performance impact—lower than 3%.

Type-1 hypervisors perform within 60–75% of bare-metal and benefits a bit from the synchronization.

VirtualBox performs the worst out of all the technologies and it is the only technology to perform worse with more processors. VirtualBox reaches a drop of performance of almost 60% when 8 processors are used. As done for Bonnie++, also Fig. 6 presents the results obtained by using the SCSI controller. We noted that when using the SATA controller, the performance worsens further, reaching a drop of more 20% w.r.t. bare-metal.

The container engines achieve up to 77% of the performance of bare-metal with no synchronization. With synchronization, the performance improves to around 95% of bare-metal. We notice that container engines benefit from more processes as they generally perform better.

Considering the differences among the technologies in CPU usage in the benchmark, bare-metal spends proportionally more on IO wait than containers, while using about 12.5% less CPU during the peak. VMWare Workstation uses 70% CPU time on the host and 60% on the guest during its peak. Moreover, it uses only 2% of CPU on IO wait, which is the lowest among the technologies.

The VirtualBox CPU usage varies depending on the controller used during each benchmark, especially when the SCSI controller is used.

Comparing to the literature, in our tests, VMWare Workstation achieved results way better than any other technology. This is similar to what Graniszewski et al. [16] observed with VirtualBox using an alternative benchmark. However, even considering various disk setups, in our tests VirtualBox performs the worst overall. The decrease of performance of type-1 hypervisor reported by other authors was between 10% to 50%. In our setup, the impact is between 23 and 40%, which is within the expected range.

Threats to Validity

The first threat to the validity of our experiments is the inherent non-determinism of distributed systems, which play a major role in increasing the variability of benchmark results: context switches, networking latency, and caching effects are just some of the factors that are not under the direct check of VTmark, but whose effects can influence the results from the benchmarks. To account for the variability of the VTmark benchmarks, we aimed at repeating the measures, when possible, or run them for long enough to allow the stabilization of the results. In particular, the experiments with Dhrystone, Whetstone, and Kcbench were repeated 5, 5, and 3 times, respectively, and we obtained a standard deviation always below 2%. Bonnie++ had an higher variability but since each repetition could takes more than 20 min, we decided to run it only for 3 times. For the benchmark of RAMspeed, Iperf3, and DBENCH, which require them to run for a given allocated time, we decided to allocate 3 min. This is the amount of time we empirically observed it takes for the results to become stable.

In general, we encountered low variability with a standard deviation of above 5% only when benchmarking with Bonnie++ on VMWare Workstation.

Another threat to validity is the potential dependence of the results w.r.t. configuration parameters. All the tested technologies have a wide configuration space where combinations of different parameters result in benchmarks that are difficult to compare with each other. For example, [15] and [17] show that the choice of both host and guest OS can affect performance. In VTmark, we tried to follow the recommended guidelines whenever possible or chose the default configurations, trying to run the benchmark as closely as possible to a real production environment. For instance, Soltesz et al. [14] for their experiments disabled hyper-threading to achieve more stable results, while practitioners (e.g., [40]) suggest disabling those features that might cause a non-deterministic performance impact, like the Intel Turbo Boost feature which automatically changes CPU frequency as well as the hyper-threading and the `scaling_governor` profiles when it registers the running of demanding tasks. In VTmark we set the `scaling_governor` to the maximum performance profile, which is also the only adjustment done with the production servers at Ordbogen A/S.

Case Study

In this section, we compare the results obtained in Section "Results" with the performance profile of a real-world application used in production for nearly 10 years at Ordbogen A/S, here adapted to run on the VTs considered in Section "VTmark".

First, we describe the architecture, the components, and the resource profile of the application considered in this case study. We also report the peculiarities of adapting it for running our target technologies. We then move to briefly describe how we stress-tested each setup with synthetic-yet-realistic inbound traffic and we conclude by reporting our results and comparing them with the reference data from VTmark.

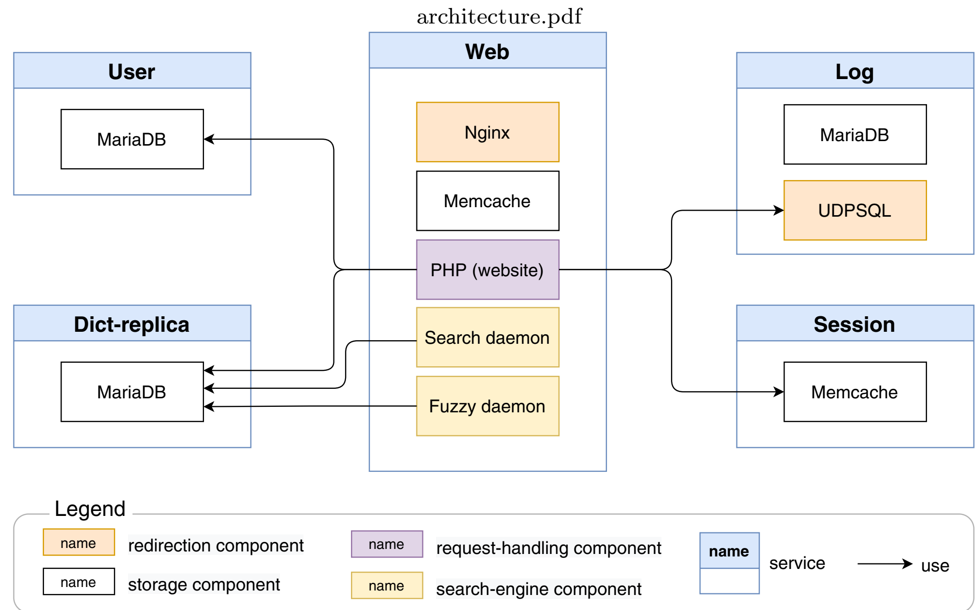

Fig. 7 Architectural diagram of the Ordbogen Website Infrastructure

The Ordbogen Website

The main aim of the Ordbogen A/S website is to serve the content of the digital dictionary (searches and pages). The use-case architecture—released in April 2010 and deprecated in February 2019—comes with an extensive testing system polished over the years to faithfully simulate the traffic generated at peak hours.

The use-case architecture consists of a Web service exposed to the customers and four internal services: Dict-replica, Log, User, Session. We report in Fig. 7 a representation of the architecture, with the service components and the relations between.

The Web service serves several Ordbogen websites, using Nginx for redirecting inbound requests. All websites are built with PHP and use Memcache for query optimization. All sites have 15 min of standard cache timeout for all semi-static data. Search and Fuzzy daemons are components for searching the Dict-replica indexes. The search daemon is a C++ Levenshtein-based search engine; while the Fuzzy daemon is a legacy C-based search engine. The web servers log each search to the Log database over UDP.

The other components are:

– the User service, which gives access to database for storing data related to users and companies such as username, email, password, licenses, etc.;
– the Session service, which helps relieving the load on the User service by caching (via Memcache) the ongoing user sessions;
– the Log service, which is an in-house UDPSQL service capturing logging data from the Web service over UDP. UDPSQL stores the data into a MySQL database, sending to the Log service each search made and the time it took to serve the request;
– the Dict-replica service, which gives access to database containing all dictionary products, letting the Search and Fuzzy daemons fetch search indexes and providing content for the websites.

From the logs recorded in production during peak hours, the Web service is the main bottleneck of the system under stress, hence, we pick that component for comparing our benchmarks.

Adaptation and Stress-Testing

Ordbogen A/S created a test suite that can simulate users behavior at peak hours. An instance of the test suite emulates the typical behaviors of more than 50,000 distinct users as separate, session-oriented HTTP requests. When stress testing https://ordbogen.com, the system would be usually tested with up to 200,000 current simulated users.

Each simulated user produces a randomized load pattern that approaches normal distribution when the number of simulated users increases. The users begin searching in the first 7 min, and each user conducts 25 searches 60–180 s apart.

For our use-case, we set up a separate benchmarking environment, running the Dict-replica, Log, User, Session in a unique node, since these are not bottlenecks for the application. We created a Docker image for each service, and with Docker compose, we built and ran the four services on a Dell PowerEdge R240 with four core Intel CPU E2124 and 16 GB DDR4 RAM and 1TB HDD.

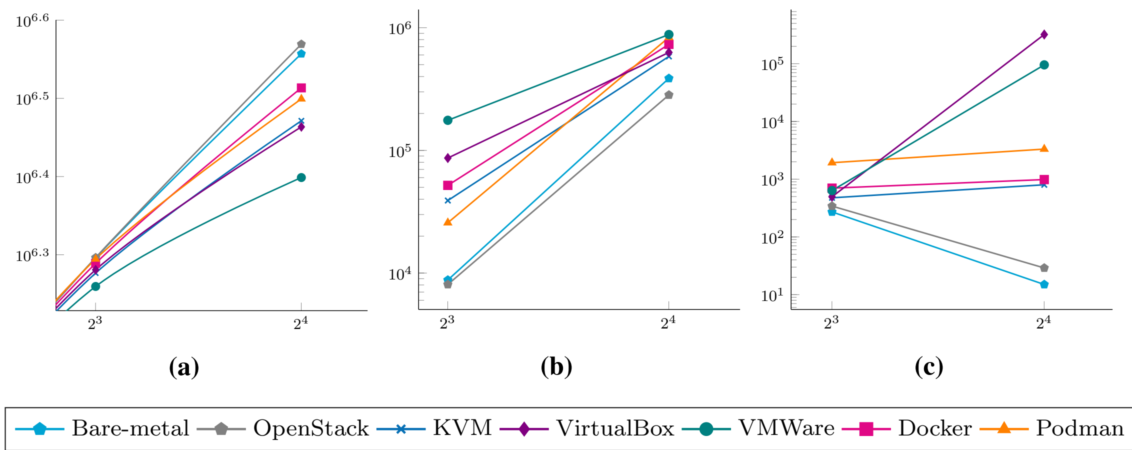

Fig. 8 Benchmarks of the case study. Axes are in log scale. The x-axes represent thousands of requests per second. Y-axes, respectively, represent: (a) # of HTTP 200 (successful) responses, (b) # of HTTP 50* (server error) responses, (c) # of non-HTTP errors (e.g., request timeout)

For creating the traffic, we used the Ordbogen A/S stress test with an increasing number of simulated users. We then measured the response rate and request execution time under load, looking into the different aspects of the considered virtualization technologies.

Empirical Benchmark Results

We report in Fig. 8 the results of our benchmarks, divided into three plots showing how many requests per second the various virtualization technologies sustain

- with a successful (HTTP 200) response, Fig. 8a;
- with a HTTP error (HTTP 50*) response, Fig. 8b;
- with a transport-level error (e.g., request timeout), Fig. 8c.

To test the various setups, we simulated 2000, 4000, 8000, and 160,000 concurrent synthetic users, each performing a simulated, realistic interaction with the Orbogen website (i.e., every user performed 25 search activities, each one composed of multiple HTTP request interactions). Concretely, each synthetic user loaded the page and performed a series of requests, amounting to 250 queries. If the synthetic user received a failure (a 50* error or a transport-level error, like a timeout) the follow-up HTTP request in the search activity was dropped, thus reducing the amount of total HTTP requests performed. The duration of the test varied between 1 h 8' and 2 h 16' due to the variability of the response times (request timeouts delayed the execution of the following searches for a simulated user).

When benchmarking Xen, we could not proceed beyond 2000 simulated users due to a high rate of connection errors, which caused the overall tests to fail. We put our best efforts to find the root cause of those failures, but, due to the complexity of the stress-test architecture, we could not pinpoint if the issue could be solved by adopting different Xen settings. For this reason, we decided to exclude Xen for the visualization of the results in Fig. 8.

For presentation sake, we visualize the results obtained simulating 8000 ($2^3 \times 1000$) and 16000 ($2^4 \times 1000$) users since the results with 4000 user or less did not present any major difference between the various technologies. Please note that the plots in Fig. 8 are in logarithmic scale, where the values in the x-axes represent thousands of requests per seconds.

In Fig. 8a—the higher the line, the better—bare-metal and OpenStack bare-metal provide the highest throughput of satisfied HTTP requests (HTTP 200). Docker and Podman follow nearby, while VirtualBox and KVM are close second-to-last and VMWare is the least performing technology.

Note that when simulating 8000 users, even bare-metal servers use 100% of CPU, and the first errors start occurring. This can be seen in Fig. 8b, c—the lower the line, the better. Looking at Fig. 8b, OpenStack bare-metal performs slightly better than bare-metal. We conjecture that the small difference in performance is due to the interplay between non-determinism and the random behavior of users generated by the test suite. While, at 8000 users, we find, respectively, Podman, KVM, Docker, VirtualBox, and VMWare as the best-performing technologies, we observe an inversion at 16,000 users, where VirtualBox has fewer 50* errors than Docker and Podman, and KVM is slightly better than Podman. Passing to Fig. 8c, we return to find bare-metal as the best-performing technology, followed by OpenStack bare-metal and KVM. Interestingly, for bare-metal and OpenStack bare-metal, the doubled amount of users (from 8000 to 16,000) corresponds to a decrease in the number of transport-level errors, which instead remain stable for the container technologies. As in Fig. 8b, there is an inversion

between the best-performing technologies at 8000 users—VirtualBox, and VMWare return slightly fewer errors than Docker, followed from a distance by Podman—while, at 16000, Docker and Podman sensibly distance VMWare and VirtualBox (note, here in inverted positions w.r.t.the 8000-calls figure). This phenomenon is probably due to the high depletion of the computational resource when simulating 16,000 users. At 8000 users, the system is still trying to satisfy their requests without reporting an error, thus causing the application to encounter transport-level errors. At 16,000 users, instead, the resources are depleted and the application returns errors reducing the number of messages that incur transport-level errors. Overall, the sum of all the errors of the container technologies is more than bare-metal and OpenStack bare-metal and less than the other technologies.

Comparison with VTmark The gathered results from our use case align with those found with VTmark. Indeed, the Ordbogen website is a CPU- and memory-intensive application and, as seen in Section “Results, we were expecting type-2 hypervisors to be the worse-performing, followed, respectively, by type-1 hypervisors, containers, and bare-metal or OpenStack bare-metal. Based on the results of our benchmarks, in particular, looking at Kcbench and DBENCH, we were expecting no decrease in performance between bare-metal and OpenStack bare-metal, a decrease of performance in the order of 2/5% at peak usage between bare-metal and the container technologies, and more than that for the other hypervisors 1 and 2 technologies. In our use-case, we measure the expected observation at 8000 users, which stress the servers and consume all the CPU resources.

As a side note, these results led the Orbogen’s DevOps to compare the running costs of the computing nodes and the performance differences between a bare-metal and a container solution, deeming the gap relevant enough (approximately 2%) to orient the adoption of OpenStack bare-metal VT for their deployment but considering for the future the possibility of adopting a container solution (Docker, Podman) to make the architecture more flexible for Cloud deployments.

Conclusion

We introduced VTmark, a suite that helps in benchmarking the performance of various virtualization technologies. Among the benefits of VTmark, it provides a way for developers to compare the performance of candidate deployment technologies without the need to adapt the actual system they want to deploy, but rather use the results from VTmark to orient their decision.

At the moment, developers can use VTmark to benchmark the most common solutions based on hypervisors and containers against bare-metal, using 7 well-known benchmarking tools for comparing the performance of CPU, memory, I/O, and networking.

As a preliminary validation of VTmark, we run it on the hardware used to run a production-grade distributed application, showing the performance trade-offs of various technologies and relating our findings with the existing literature—pointing out and discussing contradictory results. We have shown that the obtained measures correlate with the result of VTmark. The result strengthens our claim that developers can use VTmark to obtain reliable indicators to compare the performance profile of their applications with that of candidate virtualization technologies.

As future work, we plan to increase the number of virtualization technologies supported by VTmark and the number of benchmarking tools. Another direction regards the “ergonomics” of VTmark, i.e., making the execution of VTmark fully automated (e.g., using Cloud [41–43] and Serverless [44] coordination deployment tools) and parametrizing the execution of tests with respect to sets of constraints, like setting a maximal number of CPUs (memory size, etc.) used across the benchmarks, bounding the time given for the whole suite to run, as well as fixing per-tool time bounds, etc.

Regarding the selection of benchmarks, we believe that there is a need for more refined benchmarks, to distinguish and compare the performance of complex, distributed application. For example, in our experiments the benchmarks performed with RAMspeed for memory and with Dhrystone and Whetstone for single-core CPUs are not enough to show relevant performance nuances among the different technologies. While complex production-grade applications can work as benchmarks—like the one we used in our use-case—they come with their own application-specific performance profiles, making it difficult to use their results as indicators for other applications or difficult to adapt and use for a given virtualization technology. We deem this lack of proper tools a call for action for the community of researchers and practitioners spanning from Cloud Computing to DevOps to study and develop more refined benchmarks and benchmarking tools: (a) able to discriminate in finer ways the various virtualization technologies, (b) fast and openly available, c) reliable (i.e., which have limited variability), and (d) highly automated.

Declarations

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Hüttermann M. DevOps for developers. Apress; 2012.
2. Popek GJ, Goldberg RP. Formal requirements for virtualizable third generation architectures. Commun ACM. 1974;17(7):412–21.
3. Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on computer systems, vol 41. 2007; p. 275–87. https://doi.org/10.1145/1272996.1273025.
4. Chiueh SNT, Brook S. A survey on virtualization technologies. Rpe Report, 2005; p. 142.
5. Morabito R, Kjällman J, Komu M. Hypervisors vs. lightweight virtualization: a performance comparison. In: 2015 IEEE International Conference on cloud engineering, 2015; p. 386–93. IEEE. https://doi.org/10.1109/IC2E.2015.74.
6. Combe T, Martin A, Di Pietro R. To docker or not to docker: a security perspective. IEEE Cloud Comput. 2016;3(5):54–62.
7. Engler Dawson R, Frans Kaashoek M, O'Toole J, Jr. Exokernel: an operating system architecture for application-level resource management. ACM SIGOPS Oper Syst Rev. 1995;29(5):251–66.
8. Leslie IM, McAuley D, Black R, Roscoe T, Barham P, Evers D, Fairbairns R, Hyden E. The design and implementation of an operating system to support distributed multimedia applications. IEEE J Sel Areas Commun. 1996;14(7):1280–97.
9. Madhavapeddy A, Mortier R, Rotsos C, Scott D, Singh B, Gazagnaire T, Smith S, Hand S, Crowcroft J. Unikernels: library operating systems for the cloud. ACM SIGARCH Comput Arch News. 2013;41(1):461–72.
10. Goethals T, Sebrechts M, Atrey A, Volckaert B, De Turck F. Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications. In: 2018 IEEE 8th International Symposium on cloud and service computing (SC2), 2018; p. 1–8, Nov 2018.
11. Goto Y. Kernel-based virtual machine technology. Fujitsu Sci Tech J. 2011;47(3):362–8.
12. Merkel D. Docker: lightweight linux containers for consistent development and deployment. Linux J. 2014;2014(239):2.
13. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. SIGOPS Oper Syst Rev. 2003;37(5):164–77.
14. Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. SIGOPS Oper Syst Rev. 2007;41(3):275–87.
15. Mavridis I, Karatza H. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. Future Gener Comput Syst. 2019;94:674–96.
16. Graniszewski W, Arciszewski A. Performance analysis of selected hypervisors (virtual machine monitors - vmms). Int J Electron Telecommun. 2016;62(3):231–6.
17. Langer SG, French T. Virtual machine performance benchmarking. J Digit Imaging. 2011;24(5):883–9. https://doi.org/10.1007/s10278-010-9358-6.
18. Scheepers MJ. Virtualization and containerization of application infrastructure: a comparison. In: 21st Twenty Student Conference on IT, volume 21, 2014.
19. Sharma P, Chaufournier L, Shenoy P, Tay YC. Containers and virtual machines at scale: a comparative study. In: Proceedings of the 17th International Middleware Conference, pages 1–13. Association for Computing Machinery, 11 2016.
20. Chae MS, Lee HM, Lee K. A performance comparison of Linux containers and virtual machines using docker and kvm. Clust Comput. 2019;22(1):1765–75.
21. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on performance analysis of systems and software (ISPASS), 2015; p. 171–72, March 2015.
22. Kamarainen T, Shan S, Siekkinen M, Yla-Jaaski A. Virtual machines vs. containers in cloud gaming systems. In: 2015 International Workshop on network and systems support for games (NetGames), volume 2016-, pages 1–6. IEEE, 2015;2016.
23. Barik RK, Lenka RK, Rao KR, Ghose D. Performance analysis of virtual machines and containers in cloud computing. In: 2016 International Conference on computing, communication and automation (ICCCA), 2016; p. 1204–210. IEEE.
24. Li Z, Kihl M, Lu Q, Andersson JA. Performance overhead comparison between hypervisor and container based virtualization. In: 2017 IEEE 31st International Conference on advanced information networking and applications (AINA), pages 955–962. IEEE, 2017.
25. Tridgell A, Sahlberg R. Dbench. https://dbench.samba.org. Online; Accessed 22 Oct 2019.
26. Coker R. Bonnie++. 2001. https://www.coker.com.au/bonnie++/. Online; Accessed 22 Oct 2019.
27. Dongarra JJ, Moler CB, Bunch James R, Stewart Gilbert W. LINPACK users guide. SIAM; 1979.
28. Weicker Reinhold P. Dhrystone: a synthetic systems programming benchmark. Commun ACM. 1984;27(10):1013–30.
29. Curnow H, Wichmann B. A synthetic benchmark. Comput J. 1976;19:43–9.
30. Longbottom R. Classic benchmarks. http://www.roylongbottom.org.uk/classic.html. Online; Accessed 6 Mar 2020.
31. Hollander Rhett M, Bolotoff Paul V. RAMspeed, a cache and memory benchmarking tool. 2002. http://alasir.com/software/ramspeed/. Online; Accessed 22 Oct 2019.
32. Dugan J, Elliott S, Mah BA, Poskanzer J, Prabhu K. iperf—the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/. Online; Accessed 22 Oct 2019.
33. Openstack. What is OpenStack? 2020. https://www.openstack.org/software/, 2020. Online; Accessed 20 Apr 2020.
34. The Linux Foundation. Xen Project Overview. 2020. https://wiki.xen.org/wiki/Xen_Project_Software_Overview. Online; Accessed 28 July 2020.
35. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. kvm: the linux virtual machine monitor. In: Proceedings of the Linux symposium, volume 1, pages 225–230. Ottawa, Ontorio, Canada, 2007.
36. Oracle. Virtualbox manual: chapter 1 first steps. https://www.virtualbox.org/manual/ch01.html. Online; Accessed 05 Feb 2020.
37. The Linux Foundation. Open container initiative. 2020. https://opencontainers.org/. Online; Accessed 14 June 2020.
38. Containers Organization. What is podman? https://podman.io/whatis.html. Online; Accessed 5 Feb 2020.
39. Red Hat. Ansible. 2020. https://www.ansible.com/. Online; Accessed 20 Apr 2020.
40. Bakhvalov D. How to get consistent results when benchmarking on linux? https://easyperf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux. Online; Accessed 23 Oct 2019.
41. Bravetti M, Giallorenzo S, Mauro J, Talevi I, Zavattaro G. Optimal and automated deployment for microservices. In: Fundamental approaches to software engineering—22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on theory and practice of software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings. Springer; 2019, p. 351–368.
42. Bravetti M, Giallorenzo S, Mauro J, Talevi I, Zavattaro G. A formal approach to microservice architecture deployment. In:

Microservices, science and engineering. Springer; 2020, p. 183–208.
43. Bacchiani L, Bravetti M, Giallorenzo S, Mauro J, Talevi I, Zavattaro G. Microservice dynamic architecture-level deployment orchestration. In: Damiani F, Dardha O, editors, Coordination models and languages—23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on distributed computing techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings, volume 12717 of Lecture Notes in Computer Science. Springer; 2021, p. 257–75.
44. De Palma G, Giallorenzo S, Mauro J, Zavattaro G. Allocation priority policies for serverless function-execution scheduling optimisation. In: Kafeza E, Benatallah B, Martinelli F, Hacid H, Bouguettaya A, Motahari H, editors, Service-Oriented Computing—18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings, volume 12571 of Lecture Notes in Computer Science. Springer; 2020, p. 416–430.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.