Container Mapping and its Impact on Performance in Containerized Cloud Environments

Gaia Ambrosino
CeSMA–Center of Advanced Measurement Services
University of Naples Federico II
Naples, Italy
gaia.ambrosino@unina.it

Giovanni B. Fioccola
NetCom Group S.p.A.
Naples, Italy
g.fioccola@netcomgroup.eu

Roberto Canonico
DIETI Department
University of Naples Federico II
Naples, Italy
roberto.canonico@unina.it

Giorgio Ventre
DIETI Department
University of Naples Federico II
Naples, Italy
giorgio.ventre@unina.it

***Abstract*—The past few years have witnessed many software companies refactoring their monolithic applications into microservices to take advantage of containers in Cloud-native and DevOps-driven environments.**
Modern Cloud environments achieve high levels of scalability, flexibility, availability and business agility by combining two different kinds of virtualization: virtual machines and containers. The deployment of complex applications in such environments should be done by taking into account the interactions that may arise from the overlapping of these two different levels of virtualization. In this paper, we first describe a preliminary evaluation of a containerized application that has been obtained by refactoring a monolithic application into microservices. The application was deployed in a private Cloud system that relies on Kubernetes, which is a cluster management software for Docker containers that have been deployed on top of OpenStack-managed KVM virtual machines. Subsequently, in order to systematically investigate the possible interactions that can be produced when multiple containers running different kinds of workloads are activated on top of virtual machines, we present and analyze the results obtained by running different benchmarks in a testbed setup. From the analysis of the benchmark results, we derive some guidelines that should be taken into account by container schedulers when container orchestration platforms, such as Kubernetes, are deployed on top of a virtualized IaaS layer.

***Index Terms*—Cloud computing, containers, microservices**

I. Introduction

"Cloud computing represents a fundamental change in the way information technology (IT) services are invented, developed, deployed, scaled, updated, maintained and paid for" [1]. Virtualization, which is the technology that separates functions from hardware, is a key element of Cloud computing because it simplifies the delivery of services by providing a platform for managing IT resources in a scalable manner. By abstracting the physical characteristics of computing resources from their users, virtualization is useful to provide secure and isolated environments where computing capacity can be scaled up or down, on demand. Virtualization reduces maintenance costs (e.g. through automation of backup procedures and data replication) as well as facilitates the implementation of high availability architectures.

In current Cloud architectures, two different forms of virtualization are in use, one relying on virtual hardware and the other one relying on virtual operating systems. The former kind of virtualization is implemented by hypervisors, and leads to the execution of *virtual machines* (VMs), while the latter relies on so-called *containers*. Compared to VMs, containers are lightweight and can be built and destroyed faster than a virtual machine. On the other hand, VMs provide better isolated execution environments and thus offer better security and capabilities like live migration between physical servers. By consolidating containers and VMs, IT managers are able to accommodate a given workload on an infrastructure with a reduced footprint.

In the past, many companies have adopted VMs as virtualization elements to make development processes easier and more agile, and for reducing the costs. Today, with the growing popularity of DevOps-driven environments, most companies are re-engineering their monolithic applications by splitting them into multiple portable fragments executed as containers. Such an approach, based on microservices, offers advantages in terms of fault tolerance, isolation, overhead, required space, and scalability. VMs and containers, however, can coexist alongside each other. As a matter of fact, more and more enterprises recognize that these two technologies can be used synergistically [2].

In this paper, we investigate the problems that may arise when a complex application refactored as microservices is deployed in a private Cloud environment, in which containers are distributed on top of VMs. To evaluate the advantages deriving from the adoption of microservices, we first refactored a complex monolithic application and evaluated it when deployed in a private Cloud layered environment in which Docker containers are orchestrated by Kubernetes and instantiated on a pre-allocated set of VMs, which in turn are instantiated and managed by OpenStack [3]. Subsequently, in order to systematically investigate the possible interactions among containers activated on top of VMs, we have run a number of different benchmarks in a testbed setup, and finally we have analyzed the benchmark results.

The remainder of this paper is organized as follows. Section II presents the possible challenges and opportunities. Related work is discussed in Section III. In Section IV, a layered reference architecture is introduced, which allows to reach

978-1-7281-6972-9/20/$31.00 ©2020 IEEE
DOI 10.1109/SOSE49046.2020.00014

higher levels of scalability. Section V describes Docker and Kubernetes technologies, while in Section VI the Kubernetes scheduler component is illustrated. Section VII presents an experimental small scale testbed. In Section VIII, simulation results to evaluate the performance of the prototypical implementation are presented. Finally, Section IX concludes the paper by drawing our final remarks.

II. Challenges and opportunities

The process of containerization has become very important for enterprises that are modernizing their data centers. However, an increasing number of companies have chosen to adopt a model that allows to manage VMs with OpenStack and containers by using Kubernetes, because they want to orchestrate containerized apps while relying on their existing OpenStack powered infrastructure, and because of the presence of legacy applications that are deployed on virtual machines. To manage workloads by using Kubernetes and OpenStack, a first solution requires both technologies that can coexist in order to handle and monitor containers. This solution guarantees great performance, but the different resources cannot be accessed through a single unified interface. A second approach is more innovative and is based on a cloud-native environment, where OpenStack manages containers along with VMs by replacing Kubernetes. A third solution consists in running a Kubernetes cluster in VMs supervised by OpenStack, by leveraging the benefits of Kubernetes and Docker that fit seamlessly within a centralized OpenStack control system. This solution guarantees multi-tenancy and security advantages, but unexpected performance problems require further analysis and investigation, which is the scope of this paper.

III. Related work

The microservices architectural style is described in [4] as "an approach to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery". Microservices gained popularity in the last few years, thanks to the advent of the container technology, which allows to easily and flexibly deploy application components packaged together with the libraries they need to properly work with [5]. Containers encapsulate a lightweight runtime environment in which each application component is exposed to a consistent and independent software environment. In [6], authors show the advantages of containers over VMs in terms of boot-time. Since containers do not require the operating system to bootstrap, their initialization is much faster. Likewise, the time required for generating and distributing container images are shorter than for VMs.

Very recently, Boza et al. [7] have proposed to leverage performance-aware deployment strategies for containers to improve performance of containerized applications. In their paper, Boza et al. consider both the runtime and initialization time performance of containerized applications and show that default placement strategies provided by orchestrators are often inadequate. The experiments we present in this paper are in line with [7], but we explicitly take into account in our analysis the mapping of containers onto OpenStack VMs, and also the location of VMs onto physical cluster nodes. We claim that the "OpenStack and Kubernetes Better Together" proposition [2], that emerged from the OpenStack community, should be substantiated by favouring more informed container placement. In complex Cloud environments, however, optimal containers placement is still an open issue, while the same problem in the context of virtual machines has already been extensively investigated. The effective placement of VMs, in the context of Cloud infrastructures, is aimed to improve performance, resource utilization, and to reduce the energy consumption in data centers without SLA violation [8], [9], [10].

Most studies so far have considered container placement and VM placement as independent problems, while they are not. Such an approach leads to a scattered distribution of containers in the data center, which results in poor utilization of physical resources. The work [11] is the first study that we are aware of, which considers container and VM placement problems jointly. The authors propose a novel container placement strategy by simultaneously taking into account containers, VMs and physical machines. The proposed approach consists in a best-fit algorithm taking the minimum number of active physical machines as first goal and maximum resource utilization as second goal. Even though the paper presents a useful methodology, it only provides results obtained by simulation. In our work, instead, we verify how containers and virtual machines interact in a real implementation, under a realistic workload.

IV. A layered reference architecture

Recent evolutionary trends clearly show that modern Cloud environments will be based on layered architectures, where containers are executed on top of VMs. Such layered architectures allow higher levels of scalability and business agility through a flexible virtualized infrastructure.

In this paper, we assume a layered reference architecture as shown in Figure 1, whose three layers are: Orchestration Layer, IaaS Layer, and Physical Layer. The *Orchestration Layer*'s job is to execute applications in the form of containers, and to deploy and orchestrate containers according to the application requirements. This layer is the front-end for application managers and typically provides PaaS-style functionalities. The *IaaS Layer* instantiates a number of VMs for hosting the containers created by the Orchestration Layer. The amount of resources assigned to VMs (e.g. number of virtual CPUs, amount of RAM, storage space, and so on) limits the scalability of the applications. The *Physical Layer* consists of a collection of bare-metal resources, typically located in a datacenter facility, on top of which the IaaS Layer is built. It consist of servers running an hypervisor, storage systems (e.g. a Storage Area Network), and network devices in which

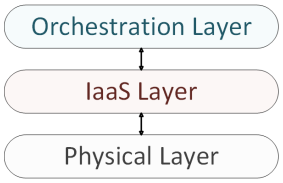

Fig. 1. Layered reference architecture

communications can be virtualized by means of network virtualization technologies and protocols (e.g. VxLAN or NVGRE).

V. Container technologies

In this section, we briefly describe the most relevant technologies that are available to deploy, manage and orchestrate containers in clustered environments.

A. Docker

A container is a software encapsulation of an operating system process that allows the process to have its own private namespace and computational resources, including memory and CPU [12] [13]. From a developer's perspective, a container is a software environment comprising an application and all its library dependencies and configuration files. Nowadays, all major public Cloud providers offer container-based Cloud services, also known as Container-as-a-Service (CaaS). Examples of CaaS services are: Google Container Engine, Amazon Elastic Container Service (ECS) and Microsoft's Azure Container Service. Application containers have been developed as an evolution of operating system virtualization. An OS-level virtualization technology method is LXC (LinuX Containers). By relying on Linux kernel's *cgroups* functionality, LXC allows creation and running of multiple isolated Linux Virtual Environments (VE) on a single host. Nowadays, the most popular container technology is Docker [14]. In Docker, the software component that hosts the containers is called *Docker Engine* and was created as an extension of LXC [15].

B. Kubernetes

Deploying a complex application is not only a matter of activating one single container. An effective adoption of the microservices paradigm requires proper tools for the automatic lifecycle management of possibly large sets of containers. This activity is referred to as *container orchestration*.

Kubernetes is "an open-source platform that is designed for automating deployment, scaling, and operations of application containers across clusters of hosts, providing a container-centric infrastructure" [16]. Kubernetes works as an orchestration system for Docker containers. Moreover, it actively manages different workloads so that users' declared objectives are satisfied. Thanks to Kubernetes, containers are easily created, destroyed, restarted and scaled up. Kubernetes allows to deploy containers across different machines and to create a communication network between them. A Kubernetes cluster consists of a combination of two kinds of nodes: worker nodes and masters. Figure 2 shows the main software components in each of the nodes of a Kubernetes cluster.

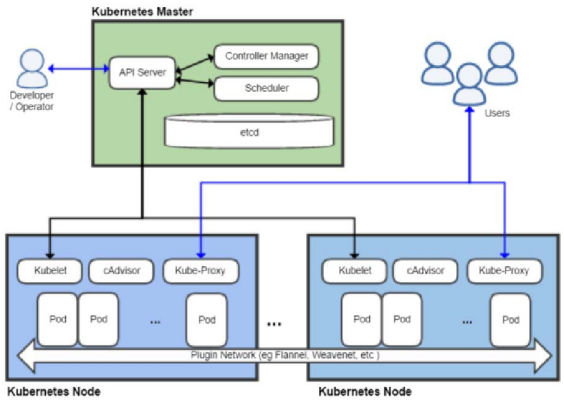

Fig. 2. Architecture of a Kubernetes cluster

The *Kubernetes Master* runs the basic components that are in charge of receiving requests from the users and correspondingly activate containers on worker nodes. These components are: *API Server* (`kube-apiserver`), *Controller Manager* (`kube-controller-manager`), *Scheduler* (`kube-scheduler`) and the key-value store *etcd*. The job of the Scheduler is to allocate a set of correlated containers, called *Pod*, onto a single Kubernetes node. A Pod is a set of one or more containers that share storage and network.

VI. Kubernetes Scheduler

The core component of Kubernetes is the *Kubernetes scheduler* that, while scanning the API server, monitors an object store for pods that have been created by a user or a controller and that have not been assigned yet to a worker node. In this case, such a new pod has not been assigned a *nodeName* yet. Then, the scheduler assigns the pod to a suitable node and it updates the *nodeName* parameter of the pod. The kubelet component running in the selected worker node, which monitors the object store for assigned pods, is notified that a new pod is in "pending execution" and it executes the pod. Finally, the pod starts running on the node.

A container scheduling is an optimization problem and there are several algorithms that the scheduler could use to decide the node it should select for running the pod. In detail, the scheduler has to choose a placement, which is a partial and non-injective assignment of a pod set to a node set. *Partial* means that there could be some pods that are not assigned to a node, while *non-injective* means that there could be some pods that are not assigned to the same node. Initially, the scheduler establishes the set of feasible placements that satisfy the constraints of the pod, then it determines which of these placements has the highest score.

To do this, the scheduler selects the nodes by using the node labels in the pod definition. When a node does not match the provided label, it is not chosen for deploying

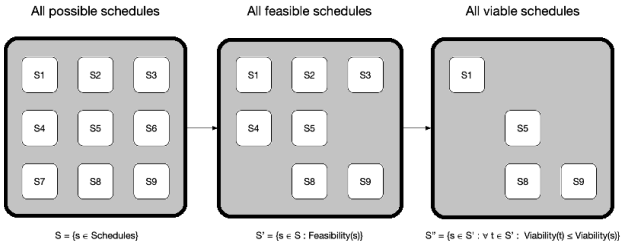

Fig. 3. A container scheduling process

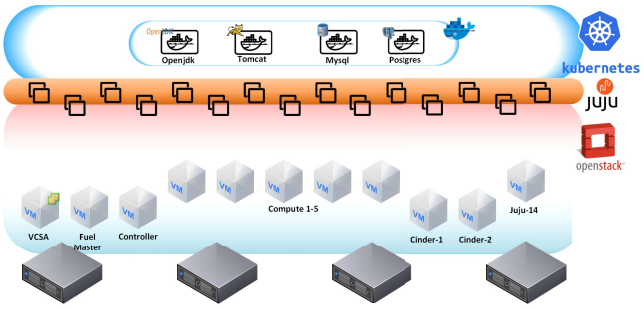

Fig. 4. Deployment of a containerized application in the testbed

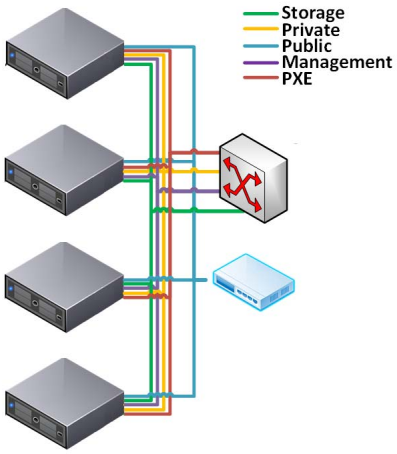

Fig. 5. Physical view of the testbed environment

the pod. This "predicate decision" translates into a true/false condition. Besides this, the scheduler favors a node where the pod image has already been pulled, and it also favors a node that does not include pods of the same service, in order to spread service pods on several nodes and better react to service failures. By using this approach, the scheduler assigns a priority to all possible nodes. Candidate nodes are ordered by their score with the highest ones on the top. At this point, the highest-scoring possible node gets chosen, in a round-robin fashion when there are nodes with the same score. After the highest priority node has been chosen, a scheduling step or a preemption step is performed. In the former case, there is at least a feasible node where the pod could be deployed. In the latter case, if no feasible nodes exist in the cluster, the preemption step is performed, and a subset of lower priorities pods running in the feasible node set must be deleted.

VII. Deployment of a Containerized Application

M3 is a hardware and software testing platform developed and owned by NetCom Group S.p.A. company. The M3 platform includes both a client and a server component. The whole platform is composed of four software components: two relational databases and two applications, DevTool and M3-TAM. In a traditional monolithic deployment, the application produces a CPU utilization of the host running the server component that increases linearly with the number of concurrent users, up to a saturation threshold. To evaluate the advantages of microservices, we refactored the application so that it could be deployed in the form of independent containers. The containerized application consists of four kinds of containers deployed as Kubernetes objects: PostgreSQL, MySQL, OpenJDK and Apache Tomcat. Figure 4 shows the deployment of the containerized application.

The application was run on an experimental small scale testbed consisting of four physical machines (see Figure 5). Two of the nodes are Dell PowerEdge R620 servers, each equipped with a single Intel Xeon CPU E5-2430-v2 running at 2.5 GHz, 16 GB RAM, two SATA internal hard disks and four Gigabit Ethernet NICs (Broadcom 4-ports 5720). Other two nodes are Dell PowerEdge R620 servers, each equipped with a single Intel Xeon CPU E5-2640-v2 running at 2.0 GHz, 32 GB RAM, two SAS internal hard disks and four Gigabit Ethernet NICs (Broadcom 4-ports 5720)

In the testbed, the IaaS layer is built on OpenStack. The OpenStack environment consists of a *Controller Node*, a *Cinder Node* and two *Compute Nodes*. The OpenStack cluster was automatically installed and configured by means of Mirantis Fuel [17]. As shown in Figure 5, the OpenStack networking infrastructure comprises five different virtual networks: PXE Network, Public Network, Private Network, Storage Network, and Management Network.

In the testbed, the automatic deployment, scaling, and lifecycle management of Kubernetes on top of OpenStack is performed by means of the Juju tool [18]. The Kubernetes cluster comprises two *Master Nodes* and seven *Worker Nodes*.

After the M3 application was deployed in Kubernetes, a load testing of the containerized application was performed. For this purpose, the Apache JMeter [19] tool was used to perform basic load testing of the application environment. Three different testing scenarios were designed by increasing the number of threads, i.e. the "number of users that JMeter will attempt to simulate", from 1 to 3. Each testing scenario consisted of HTTP requests made to the application endpoints.

Figure 6 shows the virtual CPU utilization measured on the virtual machine hosting the M3 container. The experimental

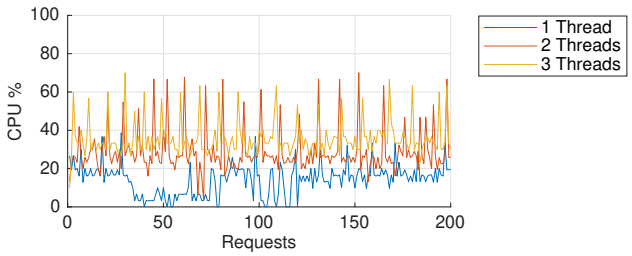

Fig. 6. CPU utilization of the VM hosting the M3 container

results show that the CPU utilization in the virtual environment is minimally sensitive to the number of concurrent users, mostly due to the fact that containerization helps to spread the workload among different physical servers.

Moreover, as the workload increases, the application is able to scale-up automatically in the proposed architecture, thanks to the Kubernetes Horizontal Pod Autoscaler [20], which adjusts the number of containers based on their CPU utilization. Hence, these tests show that the three-tier architecture is able to seamlessly manage the application and to deliver the required resources in an optimized way.

VIII. Performance Benchmarking

In this section, we aim at investigating how different container allocations of containerized applications can impact on performance, when they are executed in a layered Cloud system. For this purpose, we have created a simplified experimental environment consisting of three Kubernetes nodes installed as VMs in two physically distinct OpenStack compute nodes, as shown in Figure 7.

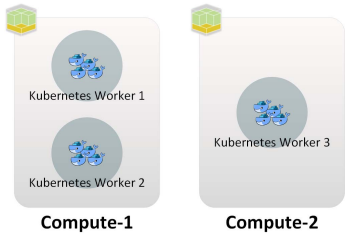

Fig. 7. Benchmarking environment

In such an environment, two different benchmarks in different scenarios are executed, producing a controlled and replicable workload. The first benchmark (network benchmarking) is structured as a client-server application, while in the second one (execution time, data transfer rate and memory benchmarking) the "dd" command-line utility is used. In the the first benchmark, the network performance is evaluated and three different scenarios are considered, characterized by: i) client and server in the same Kubernetes worker VM; ii) client and server in two different Kubernetes worker VMs located in the same physical OpenStack compute node; iii) client and server in two different Kubernetes worker VMs located in two physically distinct OpenStack compute nodes. In the second benchmark, the execution time, data transfer rate and memory performance are evaluated by starting an increasing number of concurrently active containers in a single Kubernetes worker.

A. Network Benchmarking

The first benchmark measures the network performance by using iPerf3 [21]. iPerf3 is "a tool for active measurements of the maximum achievable bandwidth on IP networks", which allows to measure TCP throughput among two endpoints. The iPerf tool is actually made of two distinct parts: a client and a server. In default mode, the client establishes a TCP connection to the server and acts as a sender of a stream of data. During a test execution, both the sender and the receiver show (at 1 second intervals) the average bandwidth during the test and the amount of data actually transmitted/received. A single test was configured to last 10 seconds.

Table I shows the average network bandwidth measured at both the sender and receiver side in ten different tests executed in the three previously described scenarios. Table I also shows the average values computed over the ten test executions for each scenario. Figures measured at the receiver side may be slightly less than those measured at the sender side. This is justified by the fact that, when the test finishes and the receiver is stopped, part of the transmitted data is still "in transit". As shown in Table I, in the first scenario the average network bandwidth of both sender and receiver is 12.9 Gb/s, and the average amount of data transferred is 14.9 GB. In this scenario, no TCP retransmissions (RTNS) were observed. This is justified by the fact that both sender and receiver are executed in containers running within the same VM.

In the second and third scenario, the average network bandwidth of both sender and receiver is about 0.8 Gb/s, and the average amount of data transferred is about 1 GB. These results are justified by the fact that Kubernetes worker nodes are executed as VMs configured with a 1 Gbps virtual NIC. Morover, in the third scenario the network bandwidth is also limited by the actual capacity of the server's physical NICs. In the third scenario, more retransmissions than in the second scenario were observed.

The tests have been repeated by linearly increasing the number of clients from 1 to 5 for each scenario: also in this case, results showed that the first scenario outperforms the other two in terms of network bandwidth.

B. CPU-intensive and I/O-intensive workloads

In the second benchmark campaign, we evaluate the execution time, data transfer rate and memory performance of multiple containers running on the same physical host. Two different benchmarks are executed: i) one producing an I/O-intensive workload; ii) another one producing a CPU-intensive workload. "dd" fetches a gigabyte of zeros from the Linux kernel and pipes them into a file on the file system. The average values of execution time, data transfer rate during write operations on the file system, and memory used were measured. The test was performed by starting an increasing number of concurrently active containers, up to 10.

TABLE I
NETWORK PERFORMANCE

First Scenario											
Network bandwidth [Gb/s]											AVG [Gb/s]
Sender	13.0	12.7	12.5	13.1	12.9	9.20	13.0	12.6	13.4	13.2	12.9
Receiver	13.0	12.7	12.5	13.1	12.9	9.20	13.0	12.6	13.4	13.2	12.9
Amount of data transferred [GB]											AVG [GB]
Sender	15.2	14.8	14.6	15.3	15.1	10.7	15.2	14.7	15.7	15.4	14.9
Receiver	15.2	14.8	14.6	15.3	15.1	10.7	15.2	14.7	15.7	15.4	14.9
RTNS	0	0	0	0	0	0	0	0	0	0	0
Second Scenario											
Network bandwidth [Gb/s]											AVG [Gb/s]
Sender	0.855	0.827	0.819	0.805	0.823	0.850	0.832	0.812	0.819	0.854	0.787
Receiver	0.852	0.824	0.818	0.803	0.821	0.848	0.829	0.810	0.817	0.852	0.784
Amount of data transferred [GB]											AVG [GB]
Sender	1.02	0.989	0.981	0.964	0.985	1.02	0.996	0.972	0.978	1.02	0.942
Receiver	1.02	0.986	0.979	0.961	0.983	1.02	0.993	0.969	0.978	1.02	0.939
RTNS	138	92	47	268	138	47	138	138	184	138	114
Third Scenario											
Network bandwidth [Gb/s]											AVG [Gb/s]
Sender	0.858	0.792	0.813	0.838	0.847	0.871	0.795	0.865	0.902	0.904	0.833
Receiver	0.856	0.790	0.811	0.836	0.845	0.869	0.793	0.863	0.901	0.903	0.831
Amount of data transferred [GB]											AVG [GB]
Sender	1.02	0.948	0.974	1.00	1.01	1.02	0.952	1.01	1.05	1.06	0.991
Receiver	1.00	0.946	0.972	1.00	1.01	1.02	0.949	1.01	1.05	1.06	0.989
RTNS	201	135	199	170	50	92	185	64	411	51	167

Each test has been repeated ten times. To generate an I/O-intensive workload, a Docker container was created in which the following Linux command was issued:

```
dd bs=1M count=1024 if=/dev/zero
of=/simple-container-benchmarks-writetest
conv=fdatasync
```

Figure 8 shows the average execution time when there are 1, 3, 5, 10 containers. Figure 9 shows the average data transfer rate when there are 1, 3, 5, 10 containers. Figure 10 shows the average amount of RAM used when there are 1, 3, 5, 10 containers. The experimental results clearly show that I/O performance sensibly degrade when multiple I/O-intensive containers are allocated onto the same physical node. To generate a CPU-intensive workload, a Docker container was created running the following Linux command:

```
dd if=/dev/urandom bs=1M count=1024 | md5sum
```

The "dd" command fetches 256 MB of random numbers from the Linux kernel and pipes them into the "md5sum" tool, which computes the MD5 hash of input data, so as to create a computationally expensive task. The average values of execution time, data transfer rate, and memory used were measured. The test was performed by starting an increasing number of concurrently active containers, up to 10.

Figure 11 shows the average execution time when there are 1, 3, 5, 10 containers. Figure 12 shows the average data transfer rate when there are 1, 3, 5, 10 containers. Figure 13 shows the average amount of RAM used when there are 1, 3, 5, 10 containers.

As for the I/O-intensive workload, also the CPU-intensive benchmark shows a performance degradation when multiple CPU-hungry containers are activated onto the same physical host.

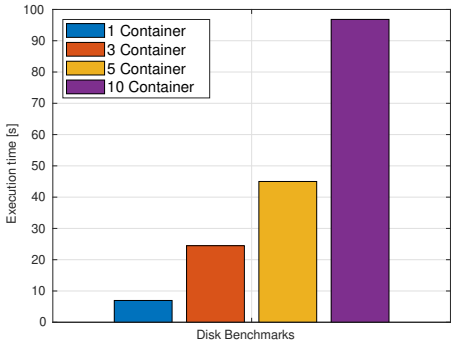

Fig. 8. Average execution time for I/O-intensive workloads

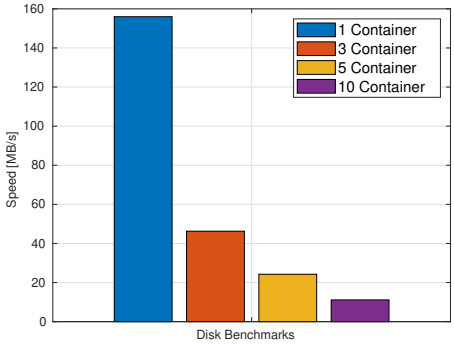

Fig. 9. Average data transfer rate for I/O-intensive workloads

IX. CONCLUSION

The reasons for migrating a monolithic application from a dedicated physical platform to a private Cloud are manifold. Nowadays, this *cloudification* process is mostly performed by

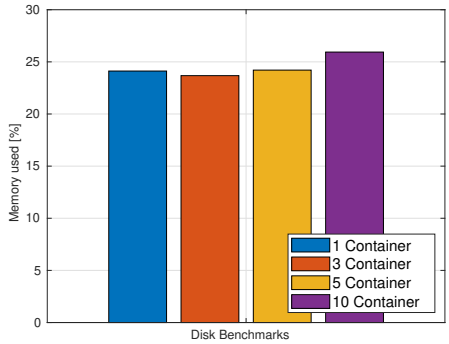

Fig. 10. Average memory used for I/O-intensive workloads

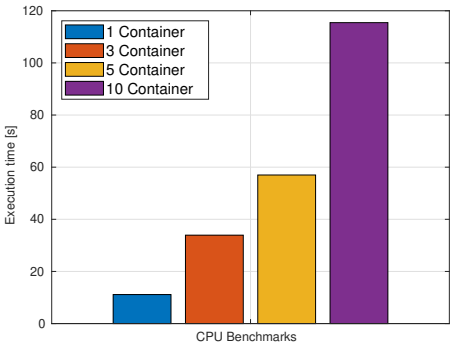

Fig. 11. Average execution time for CPU-intensive workloads

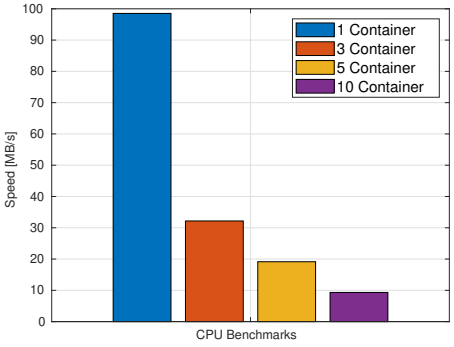

Fig. 12. Average data transfer rate for CPU-intensive workloads

decomposing the original application into small components, that may be instantiated separately as containers. In this paper, we address the advantages and risks that may arise from combining containers with VMs in a virtualized private infrastructure. While containerization clearly brings advantages with regard to flexibility, we also investigated the potential benefits in terms of scalability and effectiveness of resource utilization. In this work, the results of a series of experimental tests, conducted on a layered architecture combining Kubernetes and OpenStack, have been presented. Our results show that, while in principle a proper combination of VMs and containers gives the maximum flexibility, it is also important to take care on how these two virtualization layers interact, as a naive

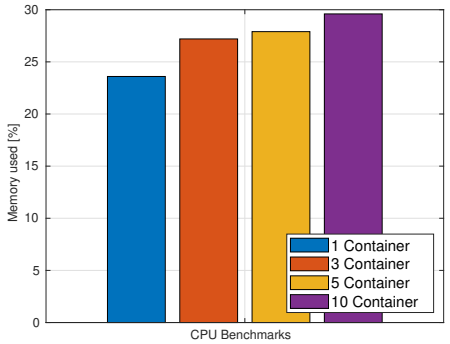

Fig. 13. Average memory used for CPU-intensive workloads

allocation of containers in such a virtualized infrastructure might easily bring to unacceptable performance degradation.

ACKNOWLEDGEMENT

This work was partially supported by Cisco Systems through the Sponsored Research Agreement "Research Project for Industry 4.0".

REFERENCES

[1] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing—the business perspective," *Decision support systems*, vol. 51, no. 1, pp. 176–189, 2011.

[2] R. LeFebvre, "Why openstack and kubernetes are better together," https://superuser.openstack.org/articles/openstack_kubernetes_better_together/, last accessed: 5 Feb 2020.

[3] "Openstack homepage," https://www.openstack.org/, last accessed: 5 Feb 2020.

[4] J. Lewis and M. Fowler, "Microservices – a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, last accessed: 5 Feb 2020.

[5] A. Simioni and T. Vardanega, "In pursuit of architectural agility: Experimenting with microservices," in *2018 IEEE International Conference on Services Computing (SCC)*, July 2018, pp. 113–120.

[6] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud," *Advanced Science and Technology Letters*, vol. 66, no. 12, pp. 105–111, 2014.

[7] E. F. Boza, C. L. Abad, S. P. Narayanan, B. Balasubramanian, and M. Jang, "A case for performance-aware deployment of containers," in *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, ser. WOC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–30. [Online]. Available: https://doi.org/10.1145/3366615.3368355

[8] Z. Usmani and S. Singh, "A survey of virtual machine placement techniques in a cloud data center," *Procedia Computer Science*, vol. 78, pp. 491 – 498, 2016, 1st International Conference on Information Security and Privacy 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050916000958

[9] G. B. Fioccola, P. Donadio, R. Canonico, and G. Ventre, "Dynamic routing and virtual machine consolidation in green clouds," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2016, pp. 590–595.

[10] P. D. Bharathi, P. Prakash, and M. V. K. Kiran, "Virtual machine placement strategies in cloud computing," in *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, April 2017, pp. 1–7.

[11] R. Zhang, A.-m. Zhong, B. Dong, F. Tian, and R. Li, *Container-VM-PM Architecture: A Novel Architecture for Docker Container Placement*. Springer International Publishing, 06 2018, pp. 128–140.

[12] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, September 2017.

[13] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, 2017.
[14] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.
[15] "Docker homepage," https://www.docker.com/, last accessed: 5 Feb 2020.
[16] "Kubernetes homepage," http://kubernetes.io, last accessed: 5 Feb 2020.
[17] "Fuel mirantis documentation homepage," https://docs.mirantis.com/fuel-docs/mitaka/userdocs/fuel-user-guide.htmll, last accessed: 5 Feb 2020.
[18] "Juju documentation homepage," https://jaas.ai/docs/getting-started-with-juju, last accessed: 5 Feb 2020.
[19] "Apache jmeter homepage," https://jmeter.apache.org/, last accessed: 5 Feb 2020.
[20] B. Hofmann and S. Pearce, "Auto scaling kubernetes clusters on openstack," https://www.openstack.org/summit/berlin-2018/summit-schedule/events/22884/auto-scaling-kubernetes-clusters-on-openstack, last accessed: 5 Feb 2020.
[21] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/, last accessed: 5 Feb 2020.