Measuring Docker Performance: What a mess!!![*]

Emiliano Casalicchio
Blekinge Institute of Technology
Department of Computer Science and Engineering
Kalrkskrona, Sweden
emiliano.casalicchio@bth.se

Vanessa Perciballi
University of Rome Tor Vergata
Department of Civil Engineering and Computer Science Engineering
Rome, Italy
v.perciballi@gmail.com

ABSTRACT

Today, a new technology is going to change the way platforms for the internet of services are designed and managed. This technology is called container (e.g. Docker and LXC). The internet of service industry is adopting the container technology both for internal usage and as commercial offering. The use of container as base technology for large-scale systems opens many challenges in the area of resource management at run-time, for example: autoscaling, optimal deployment and monitoring. Specifically, monitoring of container based systems is at the ground of any resource management solution, and it is the focus of this work. This paper explores the tools available to measure the performance of Docker from the perspective of the host operating system and of the virtualization environment, and it provides a characterization of the CPU and disk I/O overhead introduced by containers.

Keywords

Docker, Microservices, Container, Monitoring, Performance evaluation, Internet of Service

1. INTRODUCTION

Operating system and application virtualization, also known as container (e.g. Docker [12]) and LXC [8]), becames popular since 2013 with the launch of the Docker open source project (docker.com) and with the growing interest of cloud providers [5, 1] and Internet Service Providers (ISP) [14]. A container is a software environment where one can install an application or application component (the so called microservice) and all the library dependencies, the binaries, and a basic configuration needed to run the application. Containers provide a higher level of abstraction for the process lifecycle management, with the ability not only to start/stop but also to upgrade and release a new version of a containerized service in a seamless way.

Containers became so popular because they potentially may solve many Internet of Service challenges [4], for example: the dependency hell problem, typical of complex distributed applications. The application portability problem; a microservice can be executed on any platform supporting containers. The performance overhead problem; containers are lightweight and introduce lower overhead compared to Virtual Machines (VMs). For all these reasons, and more, the Internet of Service industry adopted the container technology both for internal usage [3, 1, 17] and for offering container-based services and container development platforms [5]. Examples are: Google container engine [3], Amazon ECS (Elastic Container Service), Alauda (alauda.io), Seastar (seastar.io), Tutum (tutum.com), Azure Container Service (azure.microsoft.com). Containers are also adopted in HPC (e.g. [19]) and to deploy large scale big data applications, requiring high elasticity in managing a very large amount of concurrent components (e.g. [7, 15, 18]).

The use of container as base technology for large-scale systems opens many challenges in the area of resource management at run-time, for example: autoscaling, optimal deployment and monitoring. Specifically, monitoring of container based systems is at the ground of any resource management solution, and it is the focus of this paper.

In literature, the performance of container platforms has been mainly investigated to benchmark containers versus VMs and bare-metal (e.g. [6]) or in cloud environments (e.g. [9]). The main finding of those research results is that the overhead of containers is much less than the overhead of VMs. At the best of our knowledge, there is lack of studies on the measurement methodology, measurement tools and measurement best practices, and on the characterization of the container overhead. Addressing those issues is a prerequisite for building run-time resource management mechanisms for container-based systems.

The goal of this paper is to answer to the following research questions:

- considering the many available alternatives, what are the more appropriate tools to measure the workload generated by a containerized application in terms of CPU and disk I/O performances?
- What are the characteristics of the overhead introduced by Docker containers in term of CPU load and

[*]This work is supported by the Knowledge Foundation grant num. 20140032, Sweden, and by the University of Rome Tor Vergata, Italy. The experiments in this work has been conducted in the IMTL lab at the University or Roma Tor Vergata. The authors would like to thanks Prof. Salvatore Tucci for the fruitful discussions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE '17 Companion, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4899-7/17/04. . . $15.00
DOI: http://dx.doi.org/10.1145/3053600.3053605

disk I/O throughput? The overhead is intended versus the native host operating system environment (c.f. Figure 2). Is there any correlation between the induced workload and the overhead?

A summary of the obtained results is reported in what follows. The available container monitoring methodologies and tools generate etherogeneous results that are correct per-se but must be duly interpreted. Besides, the specialized tools for monitoring container platforms are weak in measuring the disk I/O performances. In term of performance degradation, when the CPU load requested by the application is between the 65% and 75% the overhead of the container can be roughly quantified as around the 10% with respect the host operating system. Moreover we find a correlation between the CPU quota of a container and the overhead. Concerning the disk I/O, the overhead range from 10% to 30% but we have not found any pattern or dependency between the overhead and the size of the input.

The paper is organized as follows. Section 2 provides the background on container technologies and Section 3 discusses the most important related works. The measurement tools compared in the study, the monitoring architecture we set up and the measurement methodology we used are presented in Section 4. The experimental environment, the performance metrics and the results are discussed in Section 5. Finally, in Section 6, we summarize the lesson learned and we report our conclusions.

2. CONTAINER TECHNOLOGIES

The idea of containers dates back to 1992 [16] and have matured over the years with the introduction of Linux namespace [2] and the LXC project [10], a solution designed to execute full operating system images in containers. Application containers [12] are an evolution of operating system virtualization. Rather than packaging the whole system, containers package application or even application components (the so called microservices) which introduce a new granularity level of virtualization and thus become appealing for PaaS providers [5]. The main idea behind containers is the possibility of defining a container specific environment where to install all the library dependencies, the binaries, and a basic configuration needed to run an application.

There are several management tools for Linux containers: LXC, systemd-nspawn, lmctfy, Warden, and Docker [5, 12]. Furthermore, rkt is the container management tool for CoreOS. The latter is a minimal operating system that supports popular container systems out of the box. The operating system is designed to be operated in clusters and can run directly on bare metal or on virtual machines. CoreOS supports hybrid architectures (e.g., virtual machines plus bare metal). This approach enables the Container-as-a-Service solutions that are becoming widely available.

3. RELATED WORK

Performance profiling and performance evaluation is a topic of increasing interest for the containers' research community. The first seminal work on the subject [6] provides an extensive performance comparison among a native Linux environment, Docker and KVM. In this work are compared the three environments in presence of CPU intensive, I/O intensive, Network intensive workload. Moreover, the authors have compared the performances of the systems under study when running NoSQL and SQL workloads. The main intention of the work is to assess the performance improvement of running workloads in containers rather then in VMs, that is the authors want to give an estimation of the container overhead. The comparison is based on the performance metrics collected by the benchmarking tools. A similar study, aimed at comparing the performance of containers with hypervisors is [13]. The authors use a set of benchmark, and only the metrics evaluated by the benchmark tools, to assess the performance of Docker, KVM and LXC.

In [9] the authors studied the performance of container platforms running on top of a cloud infrastructure, the NeCTAR cloud. Specifically the authors compared the performance of Docker, Flockport (LXC) and the "native" environment, that is the VM, when running different types of workloads. The comparison was intended to explore the performance of CPU, memory, network and disk. For that purpose, a set of benchmarks has been selected and, as in [6], the results was based on the metrics measured by the benchmarking tools.

In [11] the authors proposed a study on the interference among multiple applications sharing the same resources and running in Docker containers. The study focus on I/O and it proposes also a modification of the Docker's kernel to collect the maximum I/O bandwidth of the machine it is running on.

With respect to the literature our study is aimed at characterizing the workload generated by the containerized application and at quantifying the performance overhead introduced by Docker versus the native environment. Moreover, we analyze also the influence of the measurement methodology on the performance results.

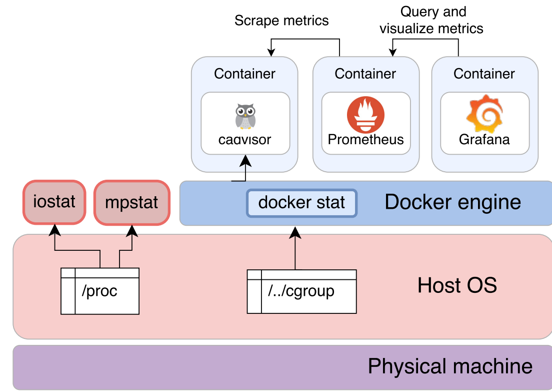

Figure 1: The monitoring infrastructure

4. PERFORMANCE MEASUREMENTS

4.1 Monitoring tools

To collect performance data we have used four open source performance profilers: `mpstat`, `iostat`, `docker stats` and `cAdvisor`. The first two are standard tools available for the Linux OS platform. `docker stats` and `cAdvisor` are tools specifically designed to monitor containers.

Figure 1 shows the monitoring architecture we set up. `mpstat` and `iostat` are part of the `sysstat` package and collect information from the Linux `/proc` virtual file system. Performance statistics for the Docker containers are stored in

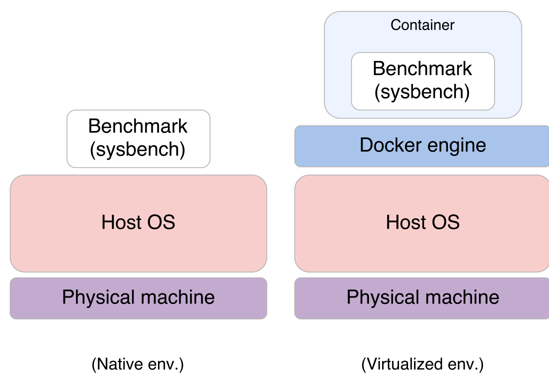

Figure 2: The native and virtualized experimental environments

the `/cgroups` virtual file system. `cAdvisor` runs in a container and it uses the Docker Remote API to obtain the statistics. `docker stats` is a Docker command, it runs in the Docker engine and it queries directly the `/cgroups` hierarchy. `Prometheus` (prometheus.io) and `Grafana` (grafana.org) are used to extract data from `cAdvisor`. The impact of those tools on the CPU utilization is negligible.

In what follows we provide a brief description of the container's specific performance profilers. We omit the description for `mpstat` and `iostat` because widely known tools.

The `docker stats` command returns a live data stream for running containers, that is: the CPU utilization, memory used (and the maximum available for the container), the network I/O (data generated and received). No file system I/O statistics are reported. It is possible to track all the containers or a specific one.

`cAdvisor` (container Advisor) is a running daemon that, for each container, keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. We didn't use cAdvisor's disk I/O metrics because, at the time we run the experiments, a software bug was reported. To extract the data sampled by `cAdvisor` each 1 second we use `Prometheus`, an open-source systems monitoring and alerting toolkit that scrapes metrics from instrumented jobs and store the resulting time series. Finally, `Grafana` query the data extracted by `Prometheus` and enable the export and the visualization of data.

4.2 Measurement methodology

For each performance test case:

- we did N runs to account for system uncertainty ($N = 10$ in our specific case);
- we sample performance data each Δt_{sample} seconds ($\Delta t_{sample} = 1$ in our specific case);
- we store the time series of performance data and the benchmark results in separated log files.

The procedure for the performance measurements in the native environment (cf. Fig. 2) is the following (repeated N times):

1. we start the benchmark
2. after $5 \times \Delta t_{sample}$ seconds (warm-up interval) we start collecting performance data with `mpstat` and `iostat` (the warm-up interval is strictly dependent on the specific benchmark, hence this is not a general recommendation, perhaps a more long warm-up period may be required)
3. our script trigger the termination of the benchmark and stop the monitoring tool.

The procedure for the performance measurements in the virtualized environment (cf. Fig. 2) is the following:

1. we activate `cAdvisor` and we continuously collect monitored data with `Prometheus` and `Grafana`;
2. we start the benchmark
3. after $5 \times \Delta t_{sample}$ sample intervals (warm-up interval) we start collecting performance data with `mpstat`, `iostat` and `docker stats`
4. our script trigger the termination of the benchmark and stop the monitoring tool.
5. we repeat steps 2 – 4 for N times. Than, we stop `cAdvisor`, `Prometheus` and `Grafana`.

Post processing of performance logs:

- we remove the cold-down phase by discarding the last 5 observations from the time series collected with `mpstat`, `iostat` and `docker stats` (this is appropriate for the specific benchmark we use and it is not a general recommendation, perhaps more long cold-down period may be required)
- we split the time series from `Grafana` in N different periods representative for the N runs.
- for all the performance data we compute the mean value, the mean square error (MSE) and the performance metrics presented in Section 5.1.

5. EXPERIMENTS

In our study we consider two use cases: CPU intensive workload; and disk I/O intensive workload.

The workload is generated using `sysbench` (https://github.com/akopytov/sysbench). The CPU intensive workload consists of verifying prime numbers by doing standard division of the input number by all numbers between 2 and the square root of the number. The disk I/O intensive workload consist in doing sequential reads, writes or random reads, writes, or a combination on files of large dimension respect to the RAM size, to avoid that caching could effect the benchmark results.

The features of the experimental environment (cf. Fig. 2) are described in Table 1. Docker is configured without any quotas on the use of the resources, that means a container can use as many resources are available.

5.1 Performance metrics

Because the purpose of this performance study is to quantify the Docker's overhead we have used a wide range of system metrics:

Table 1: Experimental environment characteristics

Processor	MD Turion(tm) II Neo N40L Dual-Core @800MHz
# of CPU, cores/socket, threads/core	2, 2, 1
RAM	2GB @1333MHz
Disk (file system type)	ATA DISK HDD 250GB (ext4)
Platforms	Ubuntu 14.04 Trusty, Docker v 1.12.3
Monitoring tools	Grafana 3.1.1, Prometheus 1.3.1, cAdvisor 0.24.1

- CPU utilization ($\%CPU$). This metric is measured using `docker stats`, `cAdvisor` and `mpstat`. The first two tools provide the value of the % of CPU used by the monitored application. `mpstat` provides the percentage of CPU utilization ($\%user$) that occurred while executing at the user level (application) and $\%CPU = \%user - \epsilon$. While executing experiments in our controlled environment we have empirically estimated $\epsilon = 2.5\%$

- Execution Time (E) measures the time taken to execute the benchmark and is calculated by `sysbench`.

- tps indicates the number of transfers per second that were issued to the device. A transfer is an I/O request to the device. Multiple logical requests can be combined into a single I/O request to the device. A transfer is of indeterminate size.

- kB_r/s, kB_w/s Indicate the amount of data read and written to/from the disk drive expressed in kilobytes per second. This metic is measured only with `iostat`. As before mentioned `docker stats` and `cAdvisor` do not provide enough and stable disk I/O metrics.

- CPU_{ovh} is the CPU overhead expressed as a fraction of the $\%CPU$ in the native environment. It is defined as

$$CPU_{ovh} = \frac{|\%CPU_{docker} - \%CPU_{native}|}{\%CPU_{native}}$$

- IO_{ovh} is the disk I/O throughput overhead expressed as a fraction of the kB_r/s or kB_w/s in the native environment. It is defined as

$$IO_{ovh,r} = \frac{|(kB_r/s)_{docker} - (kB_r/s)_{native}|}{(kB_r/s)_{native}}$$

for the read throughput and as

$$IO_{ovh,w} = \frac{|(kB_w/s)_{docker} - (kB_w/s)_{native}|}{(kB_w/s)_{native}}$$

for the write throughput.

- E_{ovh} is the execution time overhead expressed as a fraction of the E in the native environment. It is defined as

$$E_{ovh} = \frac{|E_{docker} - E_{native}|}{E_{native}}$$

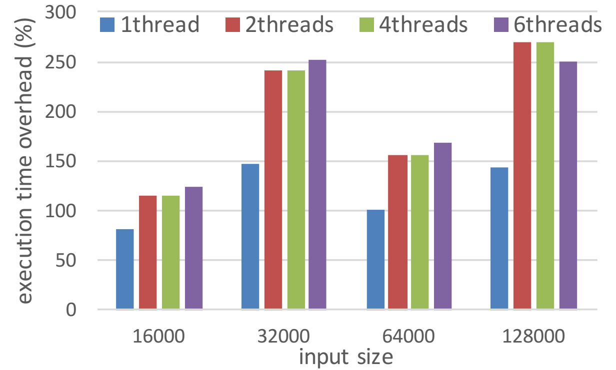

Figure 3: Execution time overhead E_{ovh}

5.2 CPU intensive workload

We run `sysbench` with *input number*={16000, 32000, 64000, 128000}. Following the approach used in literature (e.g. [6, 9]) we first analyze the execution time E and the overhead (E_{ovh}) for increasing input sizes and for increasing number of threads (1, 2, 4 and 6) used to process the input (cf. Figure 3). Increasing the number of threads has a significative impact on the CPU utilization. From these results it emerges that Docker heavily penalizes the execution time with an E_{ovh} ranging between the 80% and the 270%. Unfortunately that measure is useless in the context of the resource management, for example to parameterize adaptation models or to take auto-scaling decisions at run-time. Hence, we have measured the CPU utilization $\%CPU$ (cf. Figures 4) and the CPU overhead CPU_{ovh} (cf. Figure 5).

When the benchmark works only with 1 thread the reference CPU load of the system, hereafter $\%CPU_{native}$, is moderate, it is around 60%. In that scenario, `docker stats` provides a measure of the CPU load that is near to $\%CPU_{native}$ while `mpstat` and `cAdvisor` provide measures of the CPU load that is about 30% higher (cf. Figure 4 and 5). When the system is heavy loaded, e.g. for 4 or 6 threads, the reference load ($\%CPU_{native}$) increases to about 80% – 90%. In that case, all the measurement tools provide approximately the same results (see Figure 4), and therefore the CPU overhead goes below the 5% for the 4 threads case and below the 2.5% for the 6 threads scenario.

What does this mean? Does the overhead disappear? Is there any bias in the measurement methodology and tools?

The most logical explanation of that behavior is the following. Docker, if configured without any quotas in the use of resources, always "use" as much CPU as possible, that is between the 80% and 90%, also if the threads running inside the container are not demanding the CPU for the same amount of time. Therefore, `docker stats` allows to observe the amount of CPU demanded by the threads running inside the container, let us call that $\%CPU_{requested}$. Instead, `mpstat` and `cAdvisor` measure the effective CPU used by the container and give an effective measure of the workload on the system, that is $\%CPU$.

5.3 Disk I/O intensive workload

The purpose of these experiments is to understand the container workload when running a I/O intensive application. Specifically we use `sysbench` to do random read and

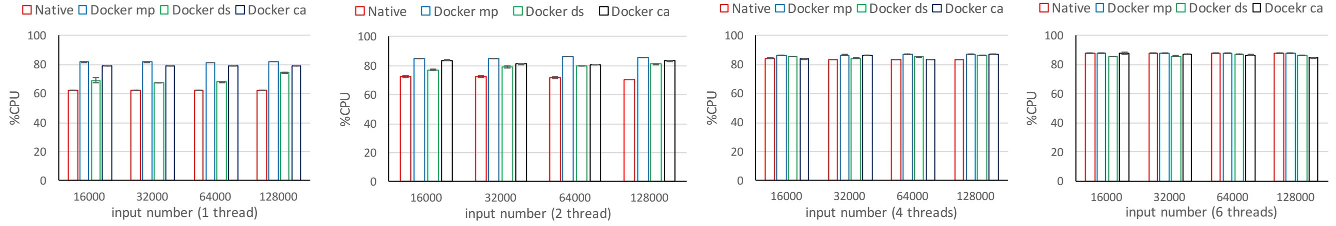

Figure 4: CPU load measured by means of: `mpstat` (Native and Docker mp), `docker stats` (Docker ds) and `cAdvisor` (Docker ca)

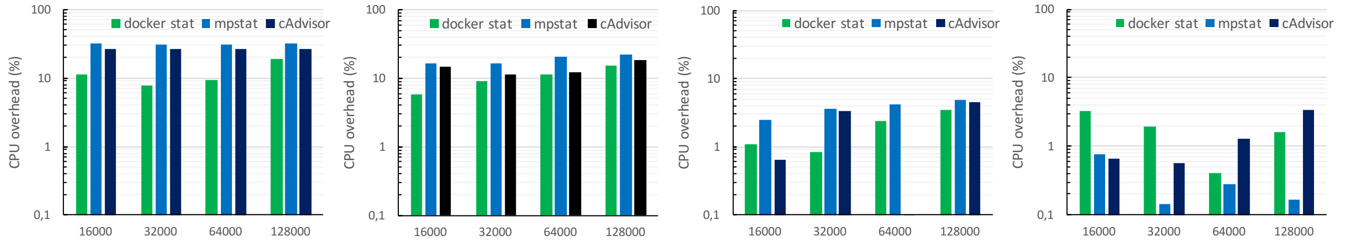

Figure 5: CPU overhead computed comparing $\%CPU_{native}$ with $\%CPU$ measured with `Gs`, `mpstat` and `cAdvisor`

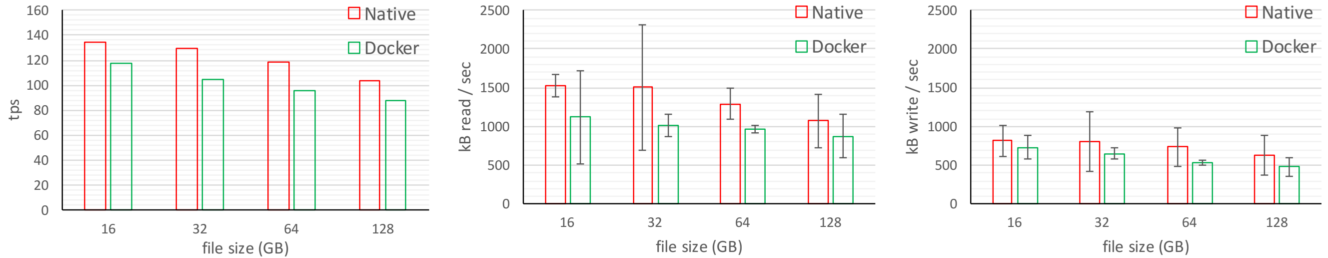

Figure 6: Disk I/O throughput measured in transactions per seconds (`tps`), kByte read per second (kB_r/s) and kByte written per second (kB_w/s)

write operations on files of the following sizes 16 GB, 32 GB, 64 GB and 128 GB. Considering that the RAM of the server we used for the experiments is 2GB we have the certainty that the OS caching mechanisms will not affect the measurements.

As before mentioned, the measurement are done only with **`iostat`** because **`docker stats`** does not collect disk I/O related data and **`cAdvisor`** is unstable for monitoring I/O.

Figure 6 reports the throughput in tps, kB_r/s and kB_w/s measured for the benchmark running on the native system and in the Docker container. As expected, the throughput in the Docker environment is lower compared with the native system, however there is not a clear dependency between the size of the dataset and the IO_{ovh}.

In terms of bytes read per seconds the overhead of Docker is between the 18% and the 33% and for bytes write per seconds the overhead of Docker is between the 10% and the 27% (c.f. Fig. 7). Moreover, it results that for larger datasets (the 64GB and 128GB cases) the overhead for read and write throughput is about the same.

We can conclude that for the disk I/O **`iostat`** (or native OS monitors) are the only available tool, and that the Docker overhead range between 10% and 30%.

6. CONCLUDING REMARKS

Measuring container performances with the goal of characterizing overhead and workload is not an easy task, also because there are not stable and dedicated tools that cover a wide range of performance metrics. From our measurement campaign, we have learned what follow:

1. the available container monitoring tools give different results that are correct *per-se* but must be duly interpreted. Moreover setting-up a monitoring infrastructure for containers requires the interconnection of at least three tools (cf. Fig. 1).

2. **`cAdvisor`** and **`mpstat`** measure the effective workload generated by the container on the CPU, i.e. $\%CPU$.

3. **`docker stats`** measures the amount of CPU requested (CPU_{req}) by the threads running inside the container and that amount can be lower than the effective CPU used by the container.

4. There is a correlation between the CPU quota set for the container and the CPU_{ovh} In case no quota is set, when the CPU_{req} is between the 65% and 75% the

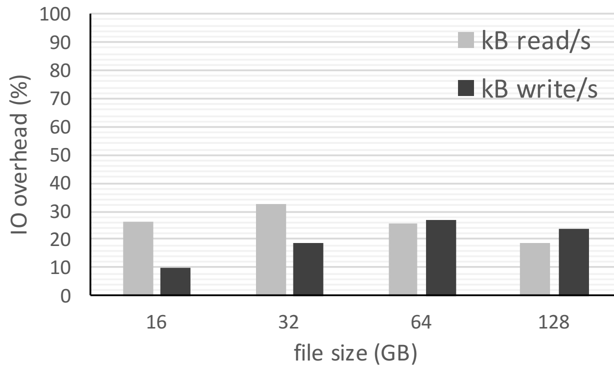

Figure 7: Disk IO Overhead for read and write requests. The Docker throughput is measured with `iostat`

overhead of the container is around the 10% with respect the native CPU load. When the CPU_{req} is over the 80% the overhead is less than the 5%.

5. There are no tools dedicated to the monitoring of disk I/O for dockerized environments.

6. The disk I/O overhead ranges from 10% to 30% but we didn't find any correlation between the overhead and the size of the input or the composition of the disk workload.

To conclude, we have not provided an exhaustive answer to the proposed research questions, but with our study we have contributed to tidying up the mess in Docker performance evaluation. The correlation between quotas and overhead need further analysis, and in general, the obtained result left space to further investigations that will be covered by our future works.

7. REFERENCES

[1] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014.

[2] E. W. Biederman. Multiple instances of the global Linux namespaces. In *2006 Ottawa Linux Symposium*, 2006.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[4] E. Casalicchio. Autonomic orchestration of containers: Problem definition and research challenges,. In *10th EAI International Conference on Performance Evaluation Methodologies and Tools*. EAI, 2016.

[5] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support PaaS. In *Proc. of 2014 IEEE Int'l Conf. on Cloud Engineering*, IC2E '14, pages 610–614, March 2014.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. Technical Report RC25482(AUS1407-001), IBM, IBM Research Division, Austin Research Laboratory, July 2014.

[7] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai, and F. Meyer. Skyport: Container-based execution environment management for multi-cloud scientific workflows. In *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*, DataCloud '14, pages 25–32, Piscataway, NJ, USA, 2014. IEEE Press.

[8] M. Helsley. Lxc: Linux container tools. *IBM devloperWorks Technical Library*, page 11, 2009.

[9] Z. Kozhirbayev and R. O. Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175 – 182, 2017.

[10] Linux Containers. Linux Containers - LXC. https://linuxcontainers.org/lxc/introduction, 2016.

[11] S. McDaniel, S. Herbein, and M. Taufer. A two-tiered approach to i/o quality of service in docker containers. In *2015 IEEE International Conference on Cluster Computing*, pages 490–491, Sept 2015.

[12] D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[13] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, March 2015.

[14] S. Natarajan, A. Ghanwani, D. Krishnaswamy, R. Krishnan, P. Willis, and A. Chaudhary. An analysis of container-based platforms for nfv. Technical report, IETF, April 2016.

[15] D.-T. Nguyen, C. H. Yong, X.-Q. Pham, H.-Q. Nguyen, T. T. K. Loan, and E.-N. Huh. An index scheme for similarity search on cloud computing using mapreduce over docker container. In *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication*, IMCOM '16, pages 60:1–60:6, New York, NY, USA, 2016. ACM.

[16] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, Apr. 1993.

[17] E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen. Towards a container-based architecture for multi-tenant saas applications. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, ARM 2016, pages 6:1–6:6, New York, NY, USA, 2016. ACM.

[18] R. Zhang, M. Li, and D. Hildebrand. Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers. In *2015 IEEE International Conference on Cloud Engineering*, pages 365–368, March 2015.

[19] J. A. Zounmevo, S. Perarnau, K. Iskra, K. Yoshii, R. Gioiosa, B. C. V. Essen, M. B. Gokhale, and E. A. Leon. A container-based approach to os specialization for exascale computing. In *First Workshop on Containers 2015 (WoC)*, 03/2015 2015.