A Holistic Evaluation of Docker Containers for Interfering Microservices

Devki Nandan Jha[1], Saurabh Garg[2], Prem Prakash Jayaraman[3], Rajkumar Buyya[4], Zheng Li[5], Rajiv Ranjan[1]
[1]Newcastle University, UK
[2]University of Tasmania, Australia
[3]Swinburne University of Technology, Australia
[4]The University of Melbourne, Australia
[5]Nanjing University of Science and Technology, China
{d.n.jha2, raj.ranjan}@newcastle.ac.uk

***Abstract*— Advancement of container technology (e.g. Docker, LXC, etc.) transformed the virtualization concept by providing a lightweight alternative to hypervisors. Docker has emerged as the most popular container management tool. Recent research regarding the comparison of container with hypervisor and bare-metal demonstrates that the container can accomplish bare-metal performance in almost all case. However, the current literature lacks an in-depth study on the experimental evaluation for understanding the performance interference between microservices that are hosted within a single or across multiple containers. In this paper, we have presented the experimental study on the performance evaluation of Docker containers running heterogeneous set of microservices concurrently. We have conducted a comprehensive set of experiments following CEEM (Cloud Evaluation Experiment Methodology) to measure the interference between containers running either competing or independent microservices. We have also considered the effects of constraining the resources of a container by explicitly specifying the cgroups. We have evaluated the performance of containers in terms of inter-container (caused by two concurrent executing containers) and intra-container (caused between two microservices executing inside a container) interference which is almost neglected in the current literature. The evaluation results can be utilized to model the interference effect for smart resource provisioning of microservices in the containerized environment.**

Keywords- Microservice, Container, Docker, Interference, Cloud Service Evaluation

I. INTRODUCTION

Virtualization is the core component of cloud computing that allows multiple tenants to run their heterogeneous applications in an isolated environment. It provides numerous advantages including heterogeneous consolidation, easy allocation, reduced failure probability, increased availability, etc. that makes virtualization user amenable whilst increasing hardware utilization. Modern enterprise applications are usually hosted in virtualized cloud datacenter utilizing the services in the form of infrastructure, platform or software.

Different virtualization techniques are developed for cloud environment. The two common methods are hypervisor-based virtualization and container-based virtualization. In hypervisor-based virtualization, each virtual machine (VM) has its own operating system irrespective of the host machine running on a hypervisor, whereas in container-based virtualization, the container utilizes the services provided by the host operating system using container engine. The architectural difference between hypervisor-based virtualization and container-based virtualization is shown in Fig. 1. Xen [1], VMWare [2], KVM [3], etc. are typical examples of hypervisor-based virtualization while Docker [4], LXC [5], Rkt [6], etc. are typical examples of container-based virtualization.

Virtual machines are considered to be the default method of virtualization. They provide many advantages but at the cost of high overhead as compared to the bare-metal performance. Recent research focuses on optimizing the degree of performance gap between virtualized and non-virtualized state of solutions. Containers are becoming an attractive choice for providing near bare-metal performance with the advantage of virtualization. It becomes a viable alternative solution for applications that do not require extreme security or strict isolation. A set of research findings [7]–[10] shows that containers are a suitable alternative to VMs for HPC workloads. These applications usually have complex software dependencies involving several libraries and support software that can be embedded together inside a container image without worrying about the host platform configurations. Any additional functionality can easily be added or removed from the existing image that makes it more flexible and customizable. The light-weight feature makes it convenient to share the container with different users. These features also allow performing repeatable and reproducible experiments on any underlying host environment.

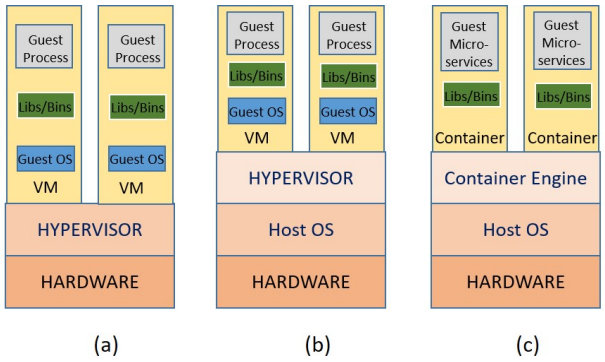

Figure 1. (a) Type-1 hypervisor, (b) Type-2 hypervisor and (c) Container-based virtualisation

2474-2473/18/$31.00 ©2018 IEEE
DOI 10.1109/SCC.2018.00012

Recent study [11] shows that we can easily run multiple microservices inside a container that also makes it suitable for IaaS workloads in addition to PaaS workloads e.g. Heroku [12], OpenShift [13], etc., which has been considered as the classical applications for the containers. Although there are several advantages of container-based virtualization, multiple containers sharing the common hardware and OS is not well investigated. The performance of a container not only depends on its own characteristics but is also affected by the interference created by other containers running on the same host (inter-container interference). The effect of interference may be more if both the containers are running microservices having competing resource requirements. The interference may also be caused by microservices running inside a container (intra-container interference). It is important to visualize the performance variation while running multiple microservices inside a container to make a decision about whether to deploy multiple microservices inside a container or to use separate containers for each microservice.

Although a set of research findings available for running HPC micro-benchmarks on containers [7]–[10], they usually consider individual microservice running in an isolated environment, which is not the usual case in cloud environment. Our work is constructed on the previous works by investigating the container performance variation in case of interference. Based on the existing trends, this paper is aimed at answering the following key questions:

- **How does the performance of Docker container vary from the baseline while running heterogeneous microservices (competing or independent) either inside a container or in separate containers?** The heterogeneity is specified towards particular resource type (CPU, memory, disk and network) of the microservices. This gives the measure of interference caused by multiple microservices running in a containerized environment.
- **Is it suitable to deploy multiple microservices inside a container? If yes, which type of microservices can be deployed together with minimal interference effect?** As different microservices may have similar resource requirements that may lead to interference and resource contention. It is important to recognize the performance variation of microservices running together so that the interference can be minimized and resources can be utilized in an optimal way.

The motivation of this paper is based on the above two research questions. This paper answers these questions and provides an understanding about the performance variation of HPC microservices. HPC microservices mimics the behavior of HPC workloads using a set of micro-benchmarks, where each micro-benchmarks has an affinity towards particular resource type. In this paper, we have adopted four popular micro-benchmarks namely Linpack, STREAM, Bonnie++ and Netperf for evaluating CPU, memory, disk I/O and network resource performance respectively. To evaluate the performance, we choose the most popular open source container technology, Docker. Each Docker container can run single or multiple microservice(s) in different possible combinations inside a container.

In this paper, we have utilized CEEM (Cloud Evaluation Experiment Methodology) approach to evaluate the performance of containers. We perform a quantitative experimental evaluation of containers under real-world conditions to analyze the performance variation. In particular, the main contributions of this paper are as follows:

- We evaluate the performance of containers running collocated microservices and compare it with the baseline container that runs only one microservice in an isolated environment. This helps us to identify the interference effect of varying microservices, each intended towards specific resource type, running inside a container (intra-container interference). This also gives an idea about mixing different microservices inside a container with minimal performance degradation.
- We also evaluate the performance of containers running in a clustered environment. Two containers running in parallel can cause interference (inter-container interference) and the effect of interference depends on the type of microservice the containers are executing. If both the containers are executing microservices having similar resource requirement, the interference effect may be higher. Our result compares the performance of this interference with the baseline performance and intra-container performance. The result can also be used for modeling smart container resource provisioning techniques to minimize the interference effect.

The rest of this paper is organized as follows. Section II discusses some relevant related work. CEEM methodology is explained briefly in Section III while application of CEEM to evaluate the performance of Docker containers is presented in section IV. Experimental results along with the inferences are discussed in Section V. Finally, Section VI concludes the paper giving some future work suggestions.

II. Related Work

Recently, containers received a lot of hype as a virtualization technology due to several features such as lightweight, packaged, self-contained, etc. Containers as a deployment environment are initially introduced by PaaS providers such as CloudFoundry [14], Heroku [12], OpenShift [13] and DotCloud [15] for workload deployment and isolation. Here, the containers are used as overlays hosted on the top of VMs running on cloud servers [16]. The PaaS workloads are mostly elastic and stateless applications, but IaaS workloads (e.g. HPC workloads) can also take advantage of the container technology.

Numerous efforts [7], [17], [18] show that containerizing the cloud infrastructure leads to highly efficient and agile solutions. It is evident from the previous work that containers can reduce the overhead while increasing the overall performance. Multiple studies support the value of containers

with respect to VMs. These studies compare the performance of containers with respect to VMs for different benchmarks and show that the performance of container is better than or almost equal to the performance of VM. Felter et al. [7] measured the CPU, memory, disk and network performance of Docker with KVM and concludes that Docker performs better than KVM in all case. Morabito et al. [8] perform similar study but they consider LXC and OSv along with Docker and KVM. They conclude that LXC outperforms KVM and Docker in almost all case. A similar study is given in [19] that uses DoKnowMe evaluation strategy to compare the performance of KVM and Docker. Kozhirbayev et al. [20] evaluates the performance of two-container technology Docker and Flockport and shows that Flockport outperforms Docker in almost all case.

Few of the works also specify running HPC workloads in Docker containers. The work in [9] shows how to orchestrate multiple containers on a physical node. The study is validated by running Linpack inside the container. Ruiz et al. [10] evaluate the performance of LXC container using NAS parallel benchmark. However, none of these works considers running multiple microservices either in same or different containers.

Sharma et al. [21] compare the performance of collocated applications on a common host but only one application is running in a container/VM. They show the effects of interference caused by noisy neighbor containers running competing, orthogonal or adversarial applications. All the experiments are done on LXC container. Similar work is done by Ye et al. [22] for big data application (Spark). They also consider the interference caused by multiple containers each running different big data applications. From the best of our knowledge, none of the existing works consider the performance evaluation of heterogeneous microservices executing inside a container and compares the interference impact with the microservices running in separate containers. In this paper, we have demonstrated the performance evaluation of HPC micro-benchmarks intended towards specific resource type (CPU, memory, disk and network) in the form of microservices running inside the Docker container. The obtained result presents the performance variation of containers while running single or multiple co-allocated (competing or independent) microservices. The output gives an understanding of interference effect caused by microservices running either in the same container or in separate containers. The output also gives a suggestion about the microservices to be mixed together with minimal interference for better resource utilization.

III. Evaluation Methodology

In order to investigate the performance of heterogeneous HPC microservices running in a container (such as Docker), we followed the Cloud Evaluation Experiment Methodology [23]. CEEM is a well-established performance evaluation methodology for cloud service evaluation and provides a systematic framework to perform evaluation studies that can easily be reproduced or extended for any environment. Due to similar guiding principles of VMs and containers, we argue by using CEEM, we will achieve rational and accurate experimental results. The steps of CEEM is briefly illustrated as follows:

1. **Requirement Recognition:** Identify the problem, and state the purpose of the proposed evaluation.
2. **Service Feature Identification:** Identify Cloud services and their features to be evaluated.
3. **Metrics and Benchmarks Listing**: List all the metrics and benchmarks that may be used for the proposed evaluation.
4. **Metrics and Benchmarks Selection:** Select suitable metrics and benchmarks for the proposed evaluation.
5. **Experimental Factors Listing:** List all the factors that may be involved in the evaluation experiments.
6. **Experimental Factors Selection:** Select limited factors to study, and also choose levels/ranges of these factors.
7. **Experimental Design:** Design experiments based on the above work. Pilot experiments may also be done in advance to facilitate the experimental design.
8. **Experimental Implementation:** Prepare experimental environment and perform the designed experiments.
9. **Experimental Analysis:** Statistically analyze and interpret the experimental results.
10. **Conclusion and Reporting:** Draw conclusions and report the overall evaluation procedure and results.

In the next section, we present the experimental design and outcomes of the experiments.

IV. Performance evaluation – Experiment Design

A. Requirement Recognition and Service Feature Identification

In this paper, our problem is mainly focused towards evaluating the performance variation of HPC microservices executing in a container with the following scenarios:

Case 1. Single container running one microservice. The resources are limited by specifying runtime constraints (cgroups) for different resource types. It provides the baseline performance for further comparison.

Case 2. Single container running multiple microservices (either independent or competing). No cgroups restrictions are enforced. Hence, the container can use all the resources provided by the host machine in a fair-share manner. For the sake of experimental validity, the number of microservices is limited by the host machine size i.e. for deploying two microservices inside a container, the host size must be double the size as defined in Case 1. We call this setup intra-container.

Case 3. Multiple containers each running one microservice. We specify two sub-case:

a) *No cgroups*: container compete for host resources on a fair-share basis.

b) *With cgroups*: each container is limited by resources specified via a configuration.

We call this setup as inter-container. The different experimental scenarios are given in Fig. 2.

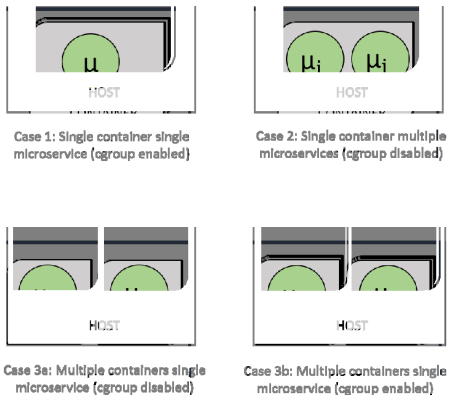

Figure 2. Schematic diagram showing different experimental scenarios

B. *Metrics/Benchmarks Listing and Selection:*

To measure the performance of individual resources of the container (e.g. CPU, memory, I/O and network), specific micro-benchmarks are employed. Table I summarizes the resource type with relevant metrics and benchmarks used for our experimental evaluation.

i. *CPU performance*: To evaluate the performance of CPU, we used Linpack [24]. Linpack measures the system's computing performance by solving a system of linear algebra equations of order *N* using lower upper factorization and partial pivoting. This benchmark is highly adaptive and provides the expected peak CPU performance. The performance is measured in terms of number of Giga Floating Point Operations performed Per Seconds (GFLOPS).

ii. *Memory performance:* To test the memory performance, we used STREAM [25] that measures the system performance by using simple operation on vectors. This benchmark shows the best possible memory bandwidth achieved by the system. There are four operations in STREAM namely COPY, SCALE, ADD and TRIAD. A description of these operations and the number of FLOPS required per iteration is shown in Table II. The result of memory performance is measured in terms of GB/sec.

iii. *Disk I/O performance:* For evaluating disk I/O performance, an open-source micro-benchmark Bonnie++ [26] is used. The size of data set must be double the size of available system memory. The output represents multiple performance parameters in terms of block output and block input for read and write respectively. The other parameters to be measured are rewrite and random seeks. The result for block input, block output and block rewrite is represented in Mb/sec while the output for random seeks is represented in /sec.

iv. *Network performance:* For measuring network performance, we used Netperf [27]. Netperf is a request-response benchmark that measures the round trip network performance between two hosts. Two identical machines are considered for the test, one running the netperf and other running the netserver. We analyze the bidirectional network traffic using TCP Stream test. No traffic is placed on the system's control connection while performing the test. The result is measured in Mbps.

For deployment, the micro-benchmarks are wrapped up as a microservices in the form of a container image. The whole process of constructing a containerized microservice image from Linpack micro-benchmarks and deployment on a host machine is shown in Fig. 3. A similar process is followed for other micro-benchmarks. The container image can be stored in a shared repository and can be easily downloaded and deployed whenever required.

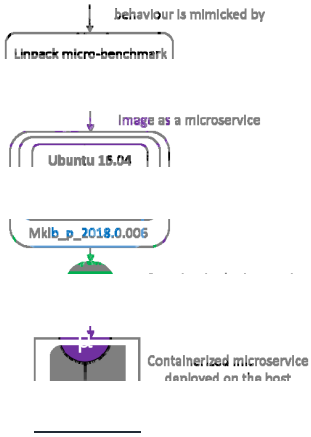

Figure 3. Steps for Linpack HPC microservice construction

TABLE I. METRICS AND BENCHMARKS FOR SELECTED SERVICE FEATURES

Resource Type	Selected Metrics	Selected Benchmarks	Version
CPU	FLOPS (Floating Point Operations Per Sec)	Linpack	Mklb_p_2018.0.006
Memory	Data Throughput	STREAM	5.10
Disk I/O	Disk Throughput, Random Seeks	Bonnie++	1.03e
Network	Network Throughput	Netperf	2.7.0

TABLE II. STREAM BENCHMARK OPERATIONS

Operation	Kernel	FLOPS per iteration
COPY	A[i] = B[i]	0
SCALE	A[i] = n × B[i]	1
ADD	A[i] = B[i] + C[i]	2
TRIAD	A[i] = B[i] + n × C[i]	3

C. Experimental Factors Listings and Selection

The performance of designed experiment is completely influenced by the selection of experimental factors. In this paper, we followed the experimental factor framework for the evaluation of cloud services to identify the various factors as discussed in [28].

- *Deployment Environment:* For our evaluation, we considered Docker container. The selection of Docker container is due to its popularity and uniformity among the cloud environment. Docker wraps up the application with all its dependencies into a container so that it can easily be executed on any Linux server (on-premise, bare metal, public or private cloud) [29]. It uses layered file system images along with the other Linux kernel features (namespace and cgroups) for the management of containers. This feature allows Docker to create any number of containers from a base image, which are copies of the base image wrapped up with additional features. This also reduces the overall memory and storage requirements that eases fast startup. Another important feature of AUFS is that each update in the base image is saved as a new image that contains all the updated information that makes it easy to track any changes.
- *CPU Index:* The CPU configuration of the virtual machine running Docker is X64 bit CPU @ 2.30GHz processor with 2 cores. We have not specified any cgroups for Case 2 and Case 3a so the container can access both cores in a fair share manner. For Case 1 and Case 3b, we specified the cgroups so that one container can use only 1 core.
- *Memory and Storage Size:* The memory configuration is confined to 4 GB DDR3 RAM while the storage is limited to 50 GB. Here also, we specified the memory limit to use only 2 GB of RAM for Case 1 and Case 3b while Case 2 and Case 3a can access the whole memory in a fair share manner.
- *Operating system:* The operating system employed in all the experiments are Ubuntu 16.04. The Docker containers also use Ubuntu 16.04 as their base image.
- *Workload size and configuration:* For each microservice, we provide a specific configuration. For Linpack, we consider the matrix of size N as 15000. We also considered the problem size (number of equations to solve) as 15000. Finally, we considered the data alignment value as 4 Kbytes. We configure the STREAM by setting the DSTREAM_ARRAY_SIZE as 60M and DNTIMES as 200. The total memory requirement for this configuration is 1373.3 MiB. For Bonnie++, we considered the file size as 8192 MB and set the uid to use as root. The network protocol for Netperf is set to TCP along with the testlen (-l) as 120 seconds.

D. Experimental Design

For evaluating the performance of a single microservice, we run our container with each microservice and collect the results. We repeat our experiments for 30 iterations to validate the results and factors for any variations introduced by the delay in accessing the resources.

For running multiple microservices together, we consider all the possible combinations as discussed in Section IV (A) with competing and independent case (e.g. CPU intensive with another CPU intensive microservice or with a Memory intensive microservice and so on). Since the isolated running time of different microservices are not identical (Linpack: 121 sec, STREAM: 122 sec, Bonnie++: 108 sec and Netperf: 120 sec), running the experiments for Case 2, 3a and 3b for some fixed number of iterations is not suitable. Hence, for these case, we repeat the benchmarking experiments for an interval of 90 minutes and compute the average performance. For Case 3a and 3b, both the containers are running concurrently while for Case 2, different microservices are running parallely in an infinite loop.

V. Performance evaluation - Outcomes and Analysis

In this section, we present the outcomes of experimental evaluations conducted to benchmark the performance of microservices running in a Docker container for the various case described in Section IV (A). For the ease of presentation of the results, we used the following abbreviations for the microservices namely Linpack (L), STREAM (S), Bonnie++ (B) and Netperf (N).

For each experimental outcome, we compute the mean and standard deviation (SD). The mean indicates the overall outcome of the experiments in terms of resource performance. To visualize the effect of interference, we also calculate interference ratio (IR). IR is calculated using particular mean value (μ_i) and the baseline mean value (μ) as shown in equation 1. Negative IR value shows the performance degradation while positive IR value shows the performance increment.

$$IR = (\mu_i - \mu)/\mu \quad (1)$$

A. CPU Performance (CPU) Evaluation and Analysis

For evaluating the computation performance, we implemented Linpack microservice in Docker. We ran experiments to evaluate the performance of Linpack microservice as per the various case described in Section IV(A). Fig. 3 shows the CPU performance variation of Linpack in terms of GFLOPS.

From Fig. 3, we can see that except for Case 2 with heterogeneous microservices, remaining combinations have a significant impact on the performance. The worst performance is shown by Case 2(L+L) with 21% performance degradation. The reason behind this is the lack of resource pinning that causes both microservices to

contend for the same core even though more cores are available. The effect of interference is clearly visible from Table III that shows the combination (L+S) have minimum interference for all the case. The performance of two Linpack instance is best when they are running in separate containers with cgroups constraints enabled with only 14% performance degradation. The remaining performances are comparable with the baseline performance.

One more point to notice from the result of Fig. 4 and Table III is that there is not much variation in the performance of containers caused due to constraining the resources as seen from Case 3a and 3b, but the performance is always better for Case 3a where one container can use extra resources not used by the other containers. The result also infers the interference effect between two containers running Linpack instance is much lesser than the interference caused by two Linpack instance running inside a single container. This shows that inter-container interference is lesser than intra-container interference while considering similar type of microservices.

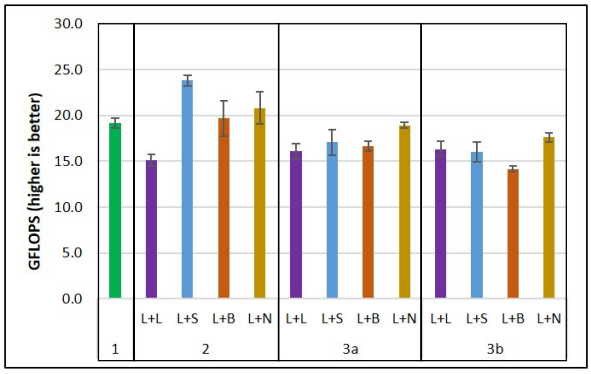

Figure 4. Average CPU Performance of Linpack. Black bars on top show the SD.

TABLE III. IR VALUE FOR LINPACK

	L+L	L+S	L+B	L+N
Case 2	-0.2102	0.2421	0.0267	0.0856
Case 3a	-0.1613	-0.1085	-0.1310	-0.0125
Case 3b	-0.1487	-0.1642	-0.1752	-0.0805

B. *Memory Evaluation and Analysis*

For memory performance evaluation, we implemented STREAM microservice deployed inside a Docker container. The simplicity of STREAM makes it suitable for sustainable memory evaluation. The average result of all the four vector operations (COPY, SCALE, ADD and TRIAD) for different case of STREAM expressed in GB/sec are shown in Table IV. The result shows that a performance gain is achieved when STREAM is deployed with different microservices collocated inside a container (Case2) with an average of 2% performance gain. Also, the performance of two instances of STREAM is better for Case 2. For Case 3a and 3b, the best performance is achieved when STREAM is deployed with Bonnie++ followed by Netperf and Linpack. The performance is worst with high standard deviation when two similar instances are deployed in separate containers as shown in Case 3a and 3b.

Fig. 5 shows the IR values for all the case. A performance gain for Case 2 (S+L), (S+B), (S+N) for all operations can be easily visualized from the positive IR values. The result also shows that the effect of interference is very less for memory intensive microservices as the performance is nearly same as the baseline performance for all the case except multiple instances of STREAM.

C. *Disk I/O Evaluation and Analysis*

To evaluate the I/O capacity of the storage disk, we used Bonnie++ microservice that creates a large dataset atleast twice the size of inbuilt memory. It evaluates multiple performance parameters but we are concerned only with sequential block output and input, sequential block rewrite and random seeks. The average performance result is shown in Table V while the IR variation is shown in Fig. 6.

The result shows the similar interference pattern indicating, running multiple instances of Bonnie++ creates higher contention while executing either in a single container or separate containers. The performance is worst for random seek with a maximum degradation of 66 % for Case 2 (B+B). The best performance is achieved by the combination of Bonnie++ with Netperf followed by Linpack and STREAM for all the case. One more point to notice from Table V is that the performance of collocated microservices inside a single container is comparable to the performance of microservices running in multiple containers.

TABLE IV. AVERAGE MEMORY PERFORMANCE OF STREAM. THE STANDARD DEVIATION IS SHOWN WITH SQUARE BRACKETS "[]".

		COPY	SCALE	ADD	TRIAD
1		13.36 [±0.07]	8.10 [±0.03]	11.85 [±0.08]	6.79 [±0.03]
2	***S+S***	11.42 [±0.78]	7.80 [±0.12]	11.00 [±0.47]	6.63 [±0.04]
	S+L	14.01 [±0.20]	8.11 [±0.02]	11.97 [±0.04]	6.80 [±0.02]
	S+B	13.45 [±0.40]	8.13 [±0.16]	11.95 [±0.24]	6.85 [±0.10]
	S+N	13.89 [±0.04]	8.17 [±0.04]	11.99 [±0.04]	6.83 [±0.03]
3a	***S+S***	11.29 [±1.51]	7.48 [±0.39]	10.25 [±1.00]	6.44 [±0.17]
	S+L	12.98 [±0.74]	7.90 [±0.25]	11.57 [±0.42]	6.66 [±0.17]
	S+B	13.42 [±0.20]	7.97 [±0.09]	11.73 [±0.11]	6.69 [±0.11]
	S+N	13.16 [±0.20]	7.91 [±0.07]	11.52 [±0.11]	6.65 [±0.05]
3b	***S+S***	11.18 [±1.33]	7.53 [±0.28]	10.40 [±0.78]	6.53 [±0.17]
	S+L	12.81 [±0.50]	7.79 [±0.27]	11.33 [±0.44]	6.61 [±0.16]
	S+B	13.13 [±0.50]	7.97 [±0.14]	11.64 [±0.32]	6.68 [±0.13]
	S+N	13.22 [±0.09]	7.94 [±0.05]	11.58 [±0.02]	6.68 [±0.02]

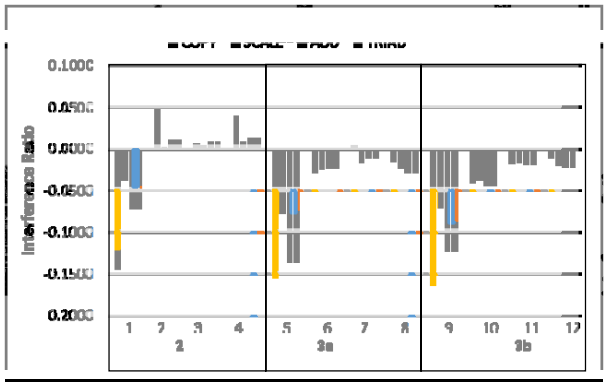

Figure 5. IR value for STREAM. Horizontal axis labels represent various case. 1-4 represents (S+S), (S+L), (S+B) and (S+N) for Case 2. Similarly, 4-8 and 8-12 represent different scenarios for Case 3a and 3b respectively.

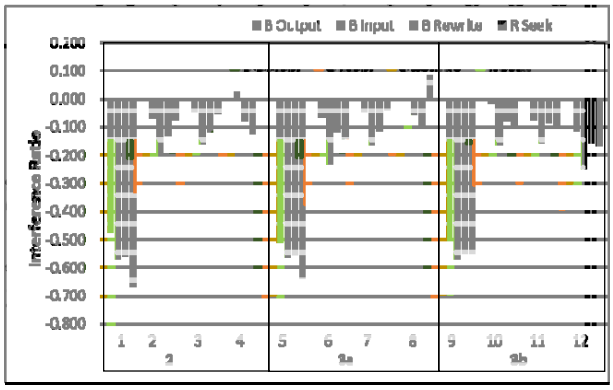

Figure 6. IR value for Bonnie++. Horizontal axis labels represent various case. 1-4 represents (B+B), (B+L), (B+S) and (B+N) for Case 2. Similarly, 5-8 and 9-12 represent different scenarios of Case 3a and 3b respectively.

The result also shows that, except for the case of multiple instances of Bonnie++ (B+B), the performance is nearly equal to the baseline performance. Very small deviation from the baseline performance for all the case except for (B+B) as shown in Fig. 6 exhibits that the performance of Bonnie++ is less affected by other microservice execution.

D. Network Performance Analysis:

To analyze the network performance, we used Netperf microservice operating one container as a server and another container as a client. The server is running the "netserver" application while the client is running "netperf" application. A stream of data is sent from client to server for an interval of 120 seconds following TCP protocol and the network throughput is calculated. The experimental outcomes for the various case are shown in Fig. 7. Table VI shows the variation of IR values.

The result shows that the performance of network intensive microservices (Netperf) is not much affected by any other microservices when deployed in separate containers. The best performance is achieved for Netperf instance deployed with other Netperf instance followed by Linpack deployed in separate containers. A large performance degradation is noticed (up to 23% for Case 3a and 3b and 31% for Case 2) for the execution of Netperf with Bonnie++. In Case 2, for both competing and independent microservices, a large performance degradation is observed (up to 45% for Case 2 (N+L)). One more observation is that the performance of Case 3a and 3b are similar showing that the cgroups resource constraint does not have a significant impact on network intensive microservices.

TABLE V. AVERAGE I/O PERFORMANCE OF BONNIE++. THE STANDARD DEVIATION IS SHOWN WITH SQUARE BRACKETS "[]".

		B Output	B Input	B Rewrite	R Seek
1		296.3 [±8.23]	340.8 [±8.61]	145.7 [±2.78]	10.5 [±0.61]
2	***B+B***	156.1 [±31.88]	146.8 [±2.71]	64.5 [±2.54]	3.5 [±0.15]
	B+L	277.0 [±12.23]	274.0 [±9.10]	126.9 [±3.09]	9.7 [±0.82]
	B+S	283.0 [±7.98]	286.6 [±7.75]	128.8 [±4.12]	10.0 [±0.77]
	B+N	303.4 [±14.81]	314.2 [±8.13]	128.0 [±3.13]	10.6 [±1.22]
3a	***B+B***	145.5 [±4.71]	149.3 [±1.51]	65.1 [±1.20]	3.8 [±0.13]
	B+L	277.6 [±11.63]	262.1 [±15.63]	129.0 [±4.45]	9.1 [±1.74]
	B+S	283.4 [±8.04]	285.0 [±7.94]	129.3 [±3.36]	10.2 [±0.75]
	B+N	295.7 [±11.90]	322.7 [±13.89]	131.2 [±4.45]	11.4 [±0.77]
3b	***B+B***	150.7 [±5.90]	146.7 [±5.22]	65.8 [±1.59]	4.8 [±0.90]
	B+L	292.2 [±13.05]	285.6 [±6.08]	134.4 [±3.11]	9.6 [±0.60]
	B+S	273.6 [±9.14]	287.4 [±7.80]	133.5 [±3.49]	9.6 [±1.36]
	B+N	293.3 [±9.37]	286.9 [±16.96]	133.2 [±6.25]	10.0 [±1.26]

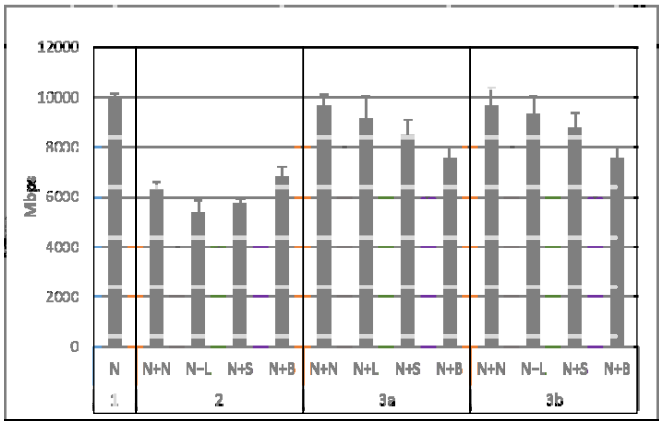

Figure 7. Average Network Performance of Netperf TCP Stream. Black bars on top show the SD.

TABLE VI. IR VALUE FOR NETPERF TCP STREAM

	N+N	N+L	N+S	N+B
Case 2	-0.3652	-0.4560	-0.4215	-0.3127
Case 3a	-0.0269	-0.0747	-0.1464	-0.2381
Case 3b	-0.0251	-0.0581	-0.1150	-0.2382

VI. Conclusions And Future Work

Containers are now considered as a viable alternative for VM in cloud infrastructure services as they provide virtualization advantages with bare metal performance. They bind all the necessary software in the form of image and can easily be deployed in any environment. These advantages make them a suitable choice for microservices with complex hardware or software requirements e.g. HPC applications. To get the full benefit from container-based virtualization, it is important to understand how multiple microservices running in same or different containers interfere with the performance of other microservices.

In this paper, we benchmarked the performance of containerized microservices in different scenarios. In particular, we conducted experimental evaluations using HPC-based microservices to study the interference issues caused by co-location of microservices in a single container and across multiple containers both running on a single host. The result provides a detailed insight about the performance variation of the microservices. The findings are as follows.

- Execution of multiple microservices inside a container is also a feasible deployment option as it gives comparable (sometimes better) performance than the baseline except for multiple execution of similar type of microservices.
- CPU intensive microservices can give better performance when running with either memory or disk intensive microservices. Memory and disk intensive microservices are not affected by other microservices running in either same container or multiple containers. The performance of network intensive microservices impacts any other microservices that are running within the same container.

In the future, we aim to conduct further research to develop a framework that takes into consideration such interference effects while provisioning microservices-based application on containers.

References

[1] P. Barham et al., "Xen and the Art of Virtualization," ACM SIGOPS Oper. Syst. Rev., vol. 37, no. 5, pp. 164–177, 2003.

[2] "VMware hypervisor." [Online]. Available: https://www.vmware.com/products/vsphere-hypervisor.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in Proceedings of the Linux symposium, 2007, pp. 225–230.

[4] "Docker." [Online]. Available: https://www.docker.com.

[5] "LXC." [Online]. Available: https://linuxcontainers.org.

[6] "CoreOS is building a container runtime, rkt." [Online]. Available: https://coreos.com/blog/rocket.html.

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171–172.

[8] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs . Lightweight Virtualization : a Performance Comparison," in Proceedings of the 2015 IEEE International Conference on Cloud Engineering Hypervisors, 2015, pp. 386–393.

[9] J. Higgins, V. Holmes, and C. Venters, "Orchestrating Docker Containers in the HPC Environment," in Proceedings of the International Conference on High Performance Computing, 2015, pp. 506–513.

[10] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance Evaluation of Containers for HPC," in Proceedings of the European Conference on Parallel Processing, 2015, pp. 813–824.

[11] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open Issues in Scheduling Microservices in the Cloud," IEEE Cloud Computing, vol. 3, no. 5, pp. 81–88, 2016.

[12] "Heroku." [Online]. Available: https://www.heroku.com.

[13] "OpenShift." [Online]. Available: https://www.openshift.com.

[14] "CloudFoundry." [Online]. Available: http://www.cloudfoundry.org.

[15] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to support PaaS," in Proceedings of the 2014 IEEE International Conference on Cloud Engineering Virtualization, 2014, pp. 610–614.

[16] B. Hindman et al., "Mesos : A Platform for Fine-Grained Resource Sharing in the Data Center," in NSDI, 2011.

[17] "Kolla." [Online]. Available: https://github.com/stackforge/kolla.

[18] K. Bankole, D. Krook, S. Murakami, and M. Silveyra, "A practical approach to dockerizing OpenStack high availability," OpenStack Paris Summit, 2014. .

[19] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance Overhead Comparison between Hypervisor and Container based Virtualization," in Proceedings of the 2017 IEEE 31st International Conference on Advanced Information Networking and Applications Performance, 2017, pp. 955–962.

[20] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the Cloud," Futur. Gener. Comput. Syst., vol. 68, pp. 175–182, 2017.

[21] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in Proceedings of the 17th International Middleware Conference on - Middleware '16, 2016, pp. 1–13.

[22] K. Ye and Y. Ji, "Performance Tuning and Modeling for Big Data Applications in Docker Containers," in Proceedings of the 2017 IEEE International Conference on Networking, Architecture, and Storage, (NAS 2017), 2017.

[23] Z. Li, L. O. Brien, and H. Zhang, "CEEM : A Practical Methodology for Cloud Services Evaluation," in Proceedings of the 2013 IEEE Ninth World Congress on Services, 2013, pp. 44–51.

[24] "Intel Math Kernel Library- Linpack." [Online]. Available: https://software.intel.com/en-us/articles/intel-math-kernel-librarylinpackdownload.

[25] J. McCalpin, "STREAM: Sustainable Memory Bandwidth in High-Performance Computers," pp. 1–4, 1995.

[26] "Bonnie++." [Online]. Available: http://www.coker.com.au/bonnie++.

[27] "The Netperf Homepage." [Online]. Available: http://www.netperf.org.

[28] Z. Li, L. O. Brien, H. Zhang, and R. Cai, "A Factor Framework for Experimental Design for Performance Evaluation of Commercial Cloud Services," in Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science, 2012, pp. 169–176.

[29] "Docker: A 'Shipping Container' for Linux code." [Online]. Available: https://www.linux.com/news/docker-shipping-container-linux-code.