Author/s:
Jha, DN;Garg, S;Jayaraman, PP;Buyya, R;Li, Z;Morgan, G;Ranjan, R

Title:
A study on the evaluation of HPC microservices in containerized environment

Date:
2019-01-01

Citation:
Jha, D. N., Garg, S., Jayaraman, P. P., Buyya, R., Li, Z., Morgan, G. & Ranjan, R. (2019). A study on the evaluation of HPC microservices in containerized environment. Concurrency Computation, 33, (7), pp.1-. Wiley. https://doi.org/10.1002/cpe.5323.

Received XXXXXXXX; | Revised XXXXXXXX; | Accepted XXXXXXXX

DOI: xxx/xxxx

ARTICLE TYPE

A Study on the Evaluation of HPC Microservices in Containerized Environment

Devki Nandan Jha*[1] | Saurabh Garg[2] | Prem Prakash Jayaraman[3] | Rajkumar Buyya[4] | Zheng Li[5] | Graham Morgan[1] | Rajiv Ranjan[1]

[1]School Of Computing, Newcastle University, UK
[2]University of Tasmania, Australia,
[3]Swinburne University of Technology, Australia
[4]The University of Melbourne, Australia
[5]University in Concepción, Chile

Correspondence
*Devki Nandan Jha Email:
D.N.Jha2@newcastle.ac.uk

Summary

Containers are gaining popularity over virtual machines (VMs) as they provide the advantages of virtualization with the performance of near bare-metal. The uniformity of support provided by Docker containers across different cloud providers makes them a popular choice for developers. Evolution of microservice architecture allows complex applications to be structured into independent modular components making them easier to manage. High performance computing (HPC) applications are one such application to be deployed as microservices, placing significant resource requirements on the container framework. However, there is a possibilty of interference between different microservices hosted within the same container (intra-container) and different containers (inter-container) on the same physical host. In this paper we describe an extensive experimental investigation to determine the performance evaluation of Docker containers executing heterogeneous HPC microservices. We are particularly concerned with how intra-container and inter-container interference influences the performance. Moreover, we investigate the performance variations in Docker containers when control groups (cgroups) are used for resource limitation. For ease of presentation and reproducibility, we use Cloud Evaluation Experiment Methodology (CEEM) to conduct our comprehensive set of experiments. We expect that the results of evaluation can be used in understanding the behaviour of HPC microservices in the interfering containerized environment.

KEYWORDS:
Container, Docker, Microservice, Interference, Performance Evaluation

1 | INTRODUCTION

Virtualization is the key concept of cloud computing that separates the computation infrastructure from the core physical infrastructure. There are numerous benefits of virtualization such as: (a) supports heterogeneous applications to run on one physical environment which is not otherwise possible; (b) allows multiple tenants to share the physical resources that in turn increases the overall resource utilization; (c) tenants are isolated from each other promoting the guarantee of QoS requirements; (d) eases the allocation and maintenance of resources for each tenant; e) enables resource scale up or scale down depending on the dynamically changing application requirements; (f) increases service availability and reduces failure probability. Applications leverage the advantages of virtualization for cloud services in the form of software platform or infrastructure [1].

There are two type of virtualization practices common in cloud environments namely, hypervisor-based virtualization and container-based virtualization. Hypervisor-based virtualization represents the de facto method of virtualization, partitioning the computing resources in terms of virtual machines (VMs) (e.g., KVM [2], VMWare [3]). Each VM possesses an isolated operating system allowing heterogeneous consolidation of multiple

applications. However, the advantages of virtualization are provided at a cost of additional overhead as compared to the non-virtualized system. Since there are two levels of abstraction (top level by VM operating system and bottom level by physical host machine) any delay incurred by the VM layer may not be removed. Current research trends concentrate on reducing the degree of performance variation due to such overhead between virtualized and bare-metal systems [4]. Containers provide virtualization advantages by exploiting the services provided by the host operating system (e.g. LXC[1], Docker[2].) Except for applications that require strict security requirements, containers becomes a viable alternative for VMs. Figure 1 represents the basic architectural difference between hypervisor and container-based virtualization.

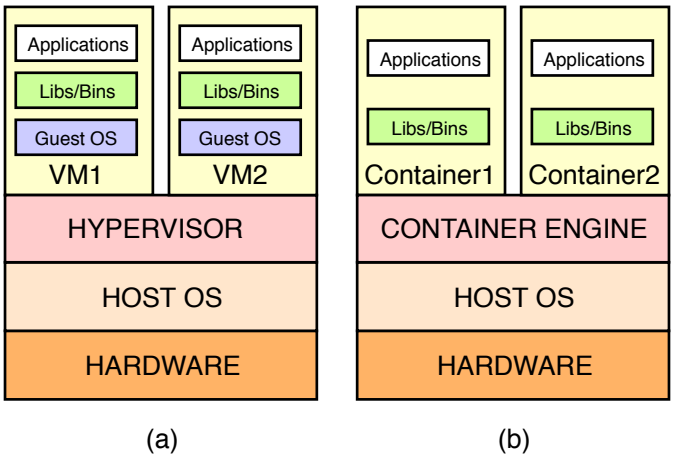

FIGURE 1 (a) Hypervisor-based Virtualization and (b) Container-based Virtualization

Different features provided by the containers (e.g., light-weight, self contained, fast start-up and shut-down) makes them a popular choice for virtualization. Recent research findings [5,6,7] verify the suitability of the container as an alternative deployment infrastructure for high performance computing (HPC) applications. Many HPC applications are highly complex as they are constructed from a variety of components, each having strict software requirements including system libraries and supporting software. This makes them closely dependent on the supporting operating system version, underlying compilers and particular environment variables, thus, makes them difficult to upgrade, debug or scale. Microservice architectures provide the complex HPC application with a more modular component-based approach, where each component can execute independently while communicating through light-weight REST-based APIs. As such, containers are considered a suitable deployment environment for microservices [8]. A container can embed the complex requirements of the HPC microservice into an image that can be deployed on heterogeneous host platforms. These features allow containers to perform repeatable and reproducible experiments on different host environments without considering the system heterogeneity issues and platform configurations. The flexibility of container images also allows customization, permitting the addition or removal of functionality. A recent study [9] shows that multiple microservices can be executed inside a single container.

Executing different microservices together can have many benefits, such as reducing inter-container data transfer delay, increased utilization of resources, and avoiding any dependency shortcomings. This scenario is suitable for HPC workloads where the resource requirements for each components/microservice are fixed and known apriori. However, the performance of containerized microservices may be affected by other microservices running inside the same container causing intra-container interference. The performance of microservices running in separate containers may also be affected because of inter-container interference as containers share the same host machine. The effect of interference is higher if both the microservices have similar resource requirements (and thus considered to be competing). For this reason, making an optimal decision about the deployment of microservices requires an extensive performance evaluation of both intra-container and inter-container interference.

Despite the increased interest in container technology there is a lack of detailed study evaluating the performance of containerized microservices executing on a host machine considering different types of interference. Many research studies are available for HPC micro-benchmarks running in containerized environments (e.g., [5,6,7,10]), but they normally only consider isolated environments. Our work is to build on the existing works by evaluating the performance variation of containerized microservices while considering the effect of interference. In a nutshell, this paper is intended to answer the following research questions:

[1]https://www.linuxcontainers.org
[2]https://www.docker.com

RQ 1: **How does the performance of Docker containers executing HPC microservices vary while running in the intra-container or the inter-container environment?**

RQ 2: **Is it suitable to deploy multiple HPC microservices inside a container? If yes, which type of microservices should be deployed together?**

The most common way to evaluate the performance of a system is to benchmark the system parameters. In the preliminary version of this paper[11], we try to answer these questions by performing a set of experiments. In this paper, we extended our previous work by providing an extensive set of experiments and discussions regarding the performance variation of HPC microservices running in the containerized environment. Figure 2 exhibits the different deployment options for HPC components/microservices on a host machine. Case 1 describes the default deployment option while Case 2 describes the scenario where multiple microservices are deployed inside a container. Case 3a and Case 3b signify the deployment of multiple containers on one host machine with cgroups constraints disabled or enabled. More details are presented in Section 4.1. Irrespective of the type of HPC application (e.g., MPI or MapReduce application), these are the basic deployment scenarios for microservices executing on a single host machine. Each application has different inter-host network communications that depend on their modular composition and architecture. The main aim of this paper is to show the performance variation caused by different types of interference on a single host machine. Inter-host network communication for specific HPC application is out of scope of this paper.

A set of micro-benchmark is considered to represent the behaviour of HPC application components where each micro-benchmark is specific for a particular resource type. For our purposes, the micro-benchmarks are considered as microservices. For evaluating the performance of common system parameters namely CPU, memory, disk and network, we consider Linpack, STREAM, Bonnie++ and Netperf (TCP Stream and TCP RR) micro-benchmarks respectively. We also consider another micro-benchmark Y-Cruncher which depends on both CPU and memory of the system. All these micro-benchmarks are evaluated in the Docker container environment under real world conditions. To ease the performance evaluation of containerized microservices, we employed Cloud Evaluation Experiment Methodology (CEEM)[12]. In particular, the main contributions of this paper are as follows:

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

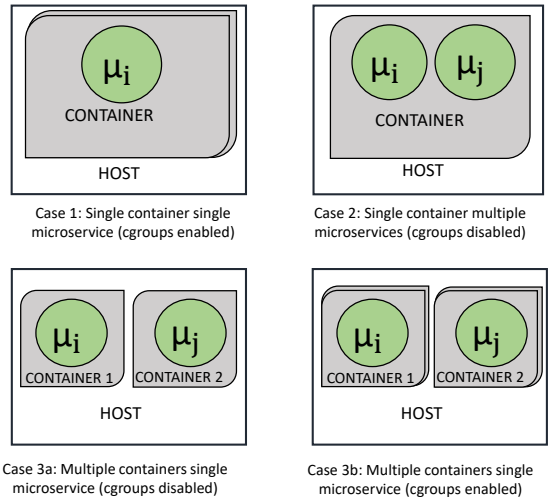

FIGURE 2 Test case scenarios

The rest of this paper is organized as follows. Section 2 gives a background of container-based virtualization sufficient for understanding the contribution of this paper. The basic concepts of the evaluation methodology CEEM is presented in Section 3 followed by the application of CEEM for the evaluation of the Docker container in Section 4. Section 5 presents the experimental results with descriptions highlighting observed interference. Section 6 presents relevant related work. A detailed discussion along with the conclusion is presented in Section 7. Finally, Section 8 provides future work suggestions.

2 | CONTAINER-BASED VIRTUALIZATION

Container-based virtualization enables multiple user-spaces (containers) to run on a physical machine by virtualizing the OS kernel rather than the physical hardware as in hypervisor-based virtualization. Figure 1 shows the main difference between container and hypervisor based virtualization. Different containers can share the same physical resources but from a hosted application's point of view, each container has their own autonomous operating system running independently.

Each container can abstract a microservice with all its dependent libraries to execute in an isolated environment. The isolation and abstraction is provided by the Linux feature, namespace and cgroups[4]. The namespace feature restricts the visibility of a container so that it can only access the resources allocated to it. PID, MNT, NET, IPC are some common namespace features used by containers to provide abstraction for process ids, file system mount points, network features and inter-process communications respectively in an inter-container environment[13]. Each new container uses the clone() system call to create an abstract system of an existing namespace in the operating system kernel. Linux cgroups are additional kernel mechanisms that control the resource allocation by restricting the system resource consumption in terms of CPU, memory, network and disk I/O for each process group. cgroups also determines the priority of resource usage by a process group. Namespace and cgroups together make the container approach an ideal choice for implementation and testing in cloud environments. Figure 3 shows the resource limitations provided by cgroups.

Docker is the most popular container management framework. An application with all its dependencies can be wrapped inside a Docker container, allowing unconstrained execution of any Linux server (bare-metal or private/public cloud)[14]. In addition to Linux kernel features namespace and cgroups, Docker also uses a layered file system, AUFS (advanced multi-layered unification file-system), for the complete management of containers. Using AUFS, a union mount for different layers of the file system is provided. This enables Docker to build multiple containers from a single base image which reduces memory and storage requirements. Additional features can easily be added to the base container and the resulting container can be saved as a new container. Each update in the container is saved as a new image that facilitates easy change tracking.

3 | 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 (CEEM)[12]. CEEM is an established performance evaluation methodology for cloud services and provides a systematic framework to perform evaluative studies that can easily be reproduced or extended for any environment. Due to similar guiding

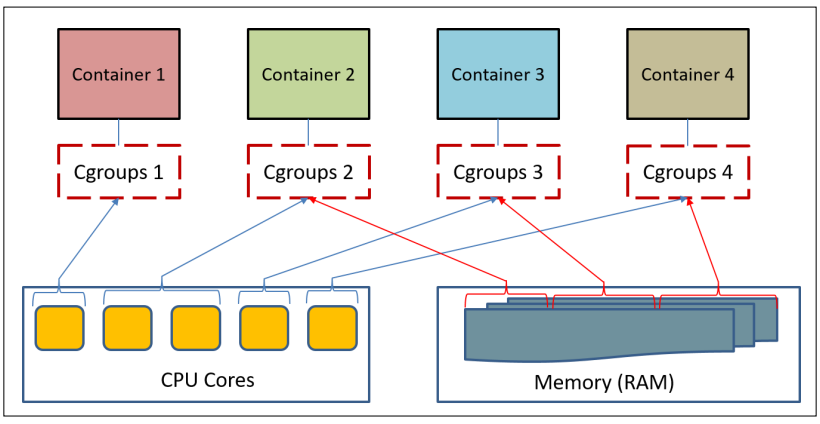

FIGURE 3 Resource restrictions provided by the cgroups

principles of VMs and containers, we argue by using CEEM, we will achieve rational and accurate experimental results[15]. The steps of CEEM are briefly illustrated as follows[16]:

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 with the option of provisioning pilot experiments 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.

To represent our evaluation in a better structured way, we divide the CEEM methodology into two major steps; namely Experimental Design and Experimental Evaluation as given in Section 4 and Section 5 respectively.

4 | PERFORMANCE EVALUATION: EXPERIMENTAL DESIGN

4.1 | Requirement Recognition and Service Feature Identification

Following the CEEM methodology, this study is based on explicitly defined requirements. In this paper our main aim is to evaluate the performance variation of containerized microservices executing in environments that may raise interference issues and compare this with a baseline performance. The requirement is defined in terms of two research questions as given in Section 1. The evaluation is driven by following three scenarios:

Case 1. *Single container running one microservice.* The resources are constrained by defining strict cgroups for different resource types. This performance acts as a baseline for remaining experimental comparisons.

Case 2. *Single container running multiple microservices (either competing or independent).* No cgroups restrictions are enforced so containers can share host machine resources in a fair-share manner. We call this set-up an intra-container configuration.

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

a. *No cgroups:* no cgroups restrictions are defined so containers can compete for resources in a fair-share manner.
b. *With cgroups:* the maximum resource a container can use is limited by specifying cgroups restrictions.

We call this set-up an inter-container configuration.

For the sake of experimental validity and fair performance comparison, the resources allocated per-container depends on the number of microservices executed by a container. For instance, the resource allocated to a container deploying two microservices is double the resource allocated to a container running only one microservice. Figure 2 depicts the different scenarios explained here.

In this study, we view containers as an alternative to virtual machines. Following the cloud service evaluation strategy[17], we examine fundamental resource parameters i.e., CPU computation, memory, I/O and network.

4.2 | Metrics and Benchmarks Listings and Selection

For measuring the performance of containerized microservices we need to consider the metrics that represent exact system behaviour. The selection of benchmarks depends on the chosen metrics and are required to be configurable and customizable for different system configurations. The metrics and benchmarks selected for the fundamental resource parameters are discussed below:

1. *CPU Computation Performance:* CPU is the system component responsible for all the processing operations happening in the system. To measure the CPU computation performance we considered the measurement of FLOPS (Floating Point operations Per Seconds), Total Computation Time and Total Turnaround Time. To check the FLOPS, we used HPC benchmark Linpack [18]. This allows us to measure the CPU computation performance by solving a set of linear algebra equations of defined order (N) using partial pivoting and Lower-Upper (LU) factorization and estimates the highest CPU performance.

   To evaluate the Total Computation and Turnaround Time we considered Y-Cruncher [19]. Y-Cruncher is a CPU+memory benchmark that stresses CPU resources by computing the value of Pi to a large number of digits. This is also dependent on the memory for swapping content at run time when available memory is insufficient. This evaluates the performance of single-core as well as multi-core systems. Y-cruncher is flexible as it allows the setting of different run-time parameters.

2. *Memory Performance:* For memory performance we considered STREAM [20] micro-benchmark that measures the data throughput for different memory operations (e.g. copy, scale). Performance is measured via different operations (COPY, SCALE, ADD and TRIAD) enacted on the memory system. Table 1 explains the kernel operations and FLOPS used by the STREAM operations. The results of STREAM is presented in terms of MB/sec.

3. *Disk I/O Performance:* We considered disk throughput and random seeks a suitable measure for disk I/O performance. To measure the disk throughput we used the Bonnie++ [21] micro-benchmark which allows us to measure the I/O file system performance with respect to data read/write speed. The output represents different performance parameters in terms of data read/write, data rewrite and random seeks per second.

4. *Network Performance:* To measure the network performance we considered round-trip network throughput. We chose Netperf [22] for measuring network throughput. Netperf is a request-response benchmark that measures network perforamance between two hosts. We identified the bidirectional network traffic using TCPStream test. We show the round trip network performance by using the TCP-RR test. To maintain the integrity, no external traffic is present during the test duration. The results are given in terms of Mbps.

Table 2 summarizes the selected metrics and benchmarks for different resource types. For deployment, the micro-benchmarks are first containerized by wrapping them up in the form of container images and then initialized for performance evaluation. Figure 4 shows the whole process of composing a Linpack microservice benchmark and deploying it onto a host machine. A similar process is used for the other micro-benchmarks. Finally, the container image is stored in the Docker Hub[3] repository so that it can be easily downloaded and deployed when required.

4.3 | Experimental Factors Listings and Selection

The performance of the experiments are entirely driven by the experimental factor selection. Following the experimental factor framework for cloud service evaluation [23] we identify the various factors:

TABLE 1 STREAM Benchmark Operations.

Operation	Kernel	FLOPS per iteration	Bytes per Iteration
COPY	A[i] = B[i]	0	16
SCALE	A[i] = n x B[i]	1	16
ADD	A[i] = B[i] + C[i]	1	24
TRIAD	A[i] = B[i] + n x C[i]	2	24

[3]https://hub.docker.com/

TABLE 2 Metrics and Benchmarks for Selected Resource Types.

Resource Type	Selected Metrics	Selected Benchmarks	Version
CPU	FLOPS (Floating Point Operations Per Sec)	Linpack	Mklb_p_2018.0.006
	Total Computation Time Total Turnaround Time	Y-Cruncher	0.7.5
Memory	Data Throughput	STREAM	5.10
Disk I/O	Disk Throughput, Random Seeks	Bonnie++	1.03e
Network	Network Throughput	Netperf	2.7.0

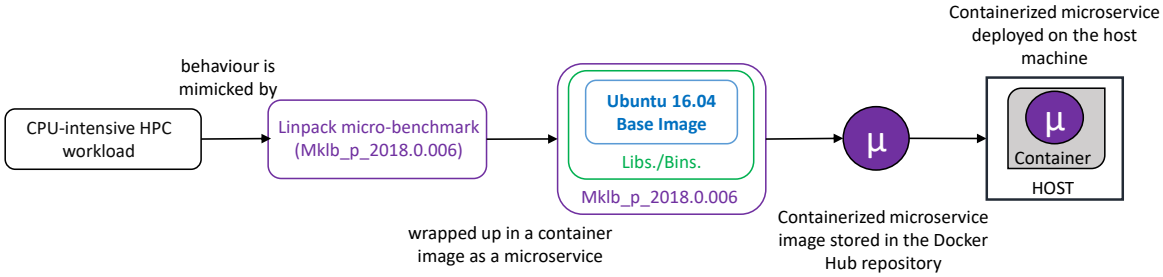

FIGURE 4 Steps for Linpack HPC microservice construction

- *Resource Type:* We considered Docker container (version: 17.05.0-ce, API version: 1.29, Go version: go1.7.5) for our evaluation. The reason for selecting the Docker container has been described previously in detail (Section 2). Application along with its dependencies can be packed inside a Docker image that can be deployed on different environments without having any prior knowledge of underlying infrastructure.
- *CPU Index:* The CPU configuration of a host machine running Docker containers is X64 bit CPU @ 2.30 GHz processor with 2 cores. For Cases 1 and 3b each container can use only 1 CPU core as specified by cgroups while for Cases 2 and 3a both available cores are shared by the containers in a fair share manner.
- *Memory and Storage Size:* The host memory and storage configuration is 4 GB DDR3 RAM and 50 GB respectively. Similar to the specified CPU configuration, containers in Cases 1 and 3b can use 2GB and 25 GB of RAM and storage respectively while the configuration is fairly shared for Cases 2 and 3a.
- *Operating System:* The operating system employed for all the experiments is Ubuntu 16.04. Docker uses Ubuntu 16.04 as a base image for all the containers.
- *Workload Size and Configuration:* For each micro-benchmark we specified a particular configuration. For Linpack the problem size (i.e., the number of equations to solve) is 15000. In addition, the leading dimensions of the array and data alignment value is set to 15000 and 4 Kbytes respectively. For Y-Cruncher the decimal place is set to 100m. We set the STREAM benchmark by configuring DSTREAM_ARRAY value as 60M and DNTIMES value as 200. The file size for Bonnie++ is set to 8192 MB while the uid is set as root. Finally for Netperf we specified TCP as the selected protocol. To check the network streaming and round-trip performance we chose TCP-STREAM and TCP-RR benchmarks and set the testlen to 120 seconds.

4.4 | Experimental Design

Our aim is to evaluate the performance of individual microservices running in the containerized environment. We used **docker run** command to start a new container instance. The container is removed (using **—rm** instruction) after finishing the execution and a new container instance is started. For Case 1, where only one microservice performance is evaluated in a single instance, we simply run the container and collect the results. To validate the results and normalize for any variations we repeated our experiments for 50 iterations.

For running multiple microservices together we considered different combinations as discussed in Section 4.1 with different cases of independent and competing microservices (e.g., CPU-intensive with other CPU-intensive or with memory-intensive). Since the average running time of different microservices are not identical, running the experiments for Case 2 and 3 for a particular number of iterations only is not suitable. Therefore, we repeat the experiments with an interval of two hours and compute the average performance. For Case 2, both microservices are executing in parallel in an infinite loop while for Case 3 both the containers are running in parallel.

5 | PERFORMANCE EVALUATION: EXPERIMENTAL RESULTS

This section describes the experimental evaluations illustrating the effect of interference for containerized microservices executing in different scenarios as given in Section 4.1. For ease of representation the following abbreviations are used for the microservices - Bonnie++: B, Linpack: L, Netperf TCP-Stream: NS, Netperf TCP-RR: NR, STREAM: S and Y-Cruncher: Y.

For each experimental outcome we compute different statistics (i.e., Mean, Trimmed Mean, Median, Maximum, Minimum, Standard Deviation (SD), Coefficient of Variance (CV) and Interference Ratio (IR)). These statistics are categorized into three types. The first category consists of Mean, Trimmed Mean and Median and represent the average result. Mean is the most commonly used parameter to represent the average, however, where there is a large variation in the result the mean does not provide an indicative average. Hence, we also selected Trimmed Mean and Median values. Trimmed Mean simply retains values between the 90^{th} percentile and the 10^{th} percentile (removing values at the extremities that may represent error spikes) while calculating the total average. The second category of statistics consists of Maximum, Minimum, SD and CV. Maximum, minimum and SD represent the variations of the result but may not give a clear comparison for different ranges of values. To compare the degree of variation between different ranges of values we chose CV. Finally, IR presents our third category of statistics which explains the effect of interference as compared to the baseline performance. IR is calculated using the following equation:

$$IR = \begin{cases} (\mu_i - \mu)/\mu, & \text{if higher is better} \\ (\mu - \mu_i)/\mu, & \text{if lower is better} \end{cases} \quad (1)$$

where, μ_i is the mean value for the particular set of microservices and μ is the baseline mean. Positive value of IR represents the performance enhancement while negative IR values represent the performance degradation.

A. *CPU Computation Performance Evaluation and Analysis*

To evaluate the CPU performance we implemented Linpack and Y-Cruncher microservices using Docker containers. Figure 5 shows the arithmetic Mean with Maximum and Minimum value for the performance of Linpack in different scenarios. Other statistics are presented in Table 3. The results show that the performance of Linpack is highest in Case 2 L(+S) with a value of 23.81 GFLOPS, which is 24% higher than baseline performance. The next highest performance is for Case 2 L(+B) followed by Case 2 L(+NS) with a performance gain of 19% and 8% respectively. The performance gain is achieved because of the availability of extra computational resources not used by other microservices (non-CPU intensive) thus increasing the performance of Linpack.

For all the other cases, a considerable performance interference is noticed. The worst performance is observed in Case 2 L(+L) where two instances of Linpack are competing in the same container resulting in a performance degradation of 21%. This is due to a lack of resources (CPU pinning) which cause both microservices to compete for the same core at the same time even though multiple cores are available. For two Linpack instances the best performance is observed for Case 3b where microservices are running in separate containers with cgroups enabled resulting in performance degradation of only 14%. The remaining performances are comparable with the baseline performance. The effect of interference is clearly observed in Figure 6.

The result in Figure 5 and Table 3 also show that the results do not deviate significantly from the Mean value. The maximum deviation is noticed in Case 2 L+(NS) followed by Case 3a L+(S) with the SD of 1.753 and 1.403 and CV of 8.4% and 8.2% respectively. Also, the difference between the Mean and Median is insignificant with the highest difference of 0.41 for Case 3a L(+S) which is smaller than the SD value (1.403).

Y-Cruncher is a CPU + memory-intensive microservice. The average performance of Computation Time (CT) and Total Time (TT) evaluated by Y-Cruncher in different scenarios is presented in Figure 7. The results show that the performance of Y-Cruncher is worst for Case 2 Y(+L) with a performance degradation of almost 46% compared to the baseline performance. This is due to the fact that both Linpack and Y-Cruncher are CPU-intensive microservices and they both compete for CPU resources inside a container. Since the operations in Y-Cruncher are highly parallelized using multi-threading, only a small performance degradation (< 2%) is noticed for Case 2 Y(+Y) as there are two cores available for the execution of two instances of Y-Cruncher. For similar reasons, the performance degradation for Case 3a and Case 3b Y(+Y) is also very less (2.3% and 2.4% respectively).

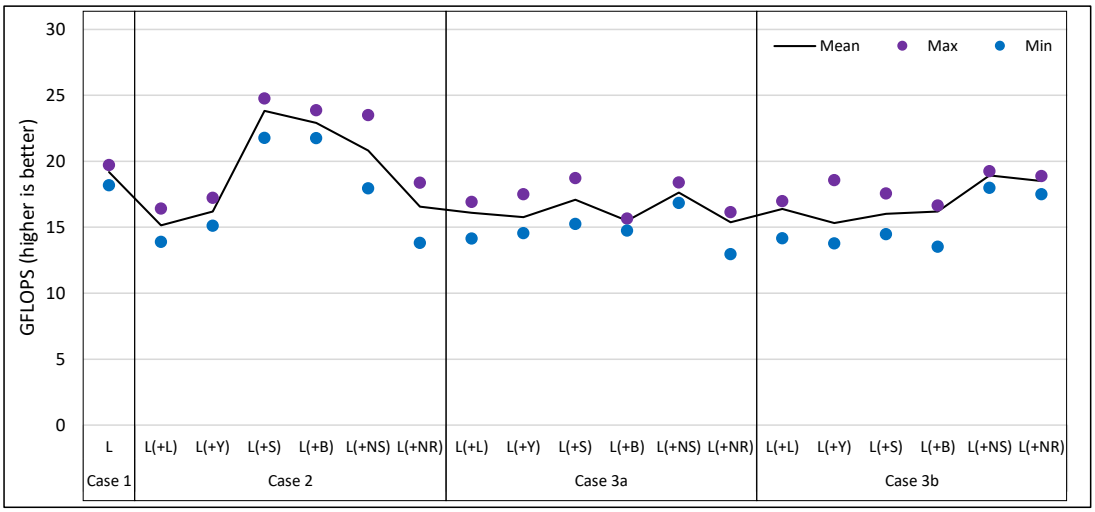

FIGURE 5 Linpack Performance Results

TABLE 3 Linpack Result.

		Mean	Median	Tr. Mean	SD	CV
Case1	L	19.17	19.37	19.19	0.5	0.026
Case 2	L(+L)	15.14	15.14	15.14	0.641	0.042
	L(+Y)	16.18	16.48	16.43	0.94	0.058
	L(+S)	23.81	23.99	23.87	0.6	0.025
	L(+B)	22.92	23.06	22.93	0.586	0.026
	L(+NS)	20.81	20.55	20.82	1.753	0.084
	L(+NR)	16.57	16.88	16.62	1.193	0.072
Case 3a	L(+L)	16.09	16.47	16.16	0.871	0.054
	L(+Y)	15.76	15.68	15.73	0.961	0.061
	L(+S)	17.09	17.5	17.1	1.403	0.082
	L(+B)	15.49	15.81	15.58	1.177	0.057
	L(+NS)	17.63	17.71	17.63	0.496	0.028
	L(+NR)	15.38	15.68	15.47	0.874	0.057
Case 3b	L(+L)	16.39	16.75	16.48	0.824	0.05
	L(+Y)	15.32	15.4	15.22	1.03	0.067
	L(+S)	16.02	15.75	16.02	1.104	0.069
	L(+B)	14.18	14.24	14.19	0.314	0.022
	L(+NS)	18.93	19.01	18.97	0.284	0.015
	L(+NR)	18.49	18.61	18.53	0.359	0.019

The next worst performance is observed for the collocated execution of Y-Cruncher and Bonnie++ with a performance degradation of 28.7%, 30.6% and 21.4% for Case 2, Case 3a and Case 3b respectively. This is due to constrained disk size. Since Y-Cruncher uses continuous swapping from main memory to disk while Bonnie++ also accesses the disk for different operations, only one process at a time can access the disk memory to perform the I/O resulting in the higher completion time.

Even though both Y-Cruncher and STREAM are memory-intensive microservices, for the collocated execution of Y-Cruncher and STREAM there is only a slight degradation of 4% for Case 2 and 1.9% and 2.5% for Case 3a and Case 3b respectively. The reason behind this is the sufficient availability of memory to run the experiment with minimal performance degradation. The best performance is observed for Case 2

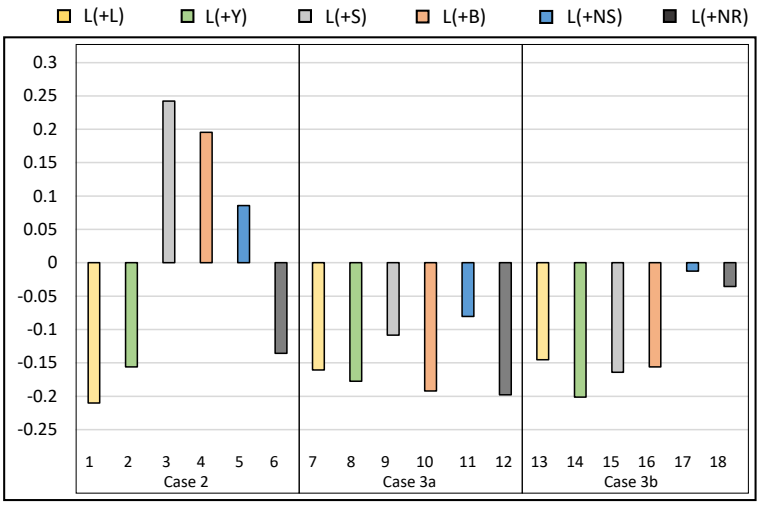

FIGURE 6 Linpack Interference Ratio (IR) Values. Horizontal axis labels represent various cases. 1 – 6 represents L(+L), L(+Y), L(+S), L+(B), L(+NS) and L(+NR) for Case 2. Similarly, 7 – 12 and 13 – 18 is used to represent different scenarios for Case 3a and 3b respectively.

Y(+NS) followed again by Case 2 Y(+NR) with a performance gain of 2.4% and 1.5% respectively as they are not directly interfering and so not competing for resources. The effect of interference can be observed in Figure 8.

The result indicates that inter-container interference is less than intra-container interference while considering similar types of CPU-intensive microservices. Another important point to note is that the performance of microservices is comparable for the cases when cgroups is enabled or disabled in our scenarios.

B. *Memory Performance Evaluation and Analysis*

To evaluate memory performance we use the STREAM microservice benchmark. Statistics for the four vector operations (COPY, SCALE, ADD and TRAID) are presented in Figure 9. For the COPY operation a degradation of 14%, 15% and 16% is observed for collocated execution of two STREAM microservices in Case 2, Case 3a and Case 3b respectively. This is because of the interference caused by memory-intensive operations executing together. The next worst case performance is observed for Case 2 S(+Y), as Y-Cruncher shares the available memory with a degradation of 3%. For other combinations in Case 2 a slight performance gain is noticed with a maximum 4.8% gain for S(+L) followed by 3.9% for S(+NS) due to the nature of their dependencies on memory. The results also show that there is very slight deviation from the Mean

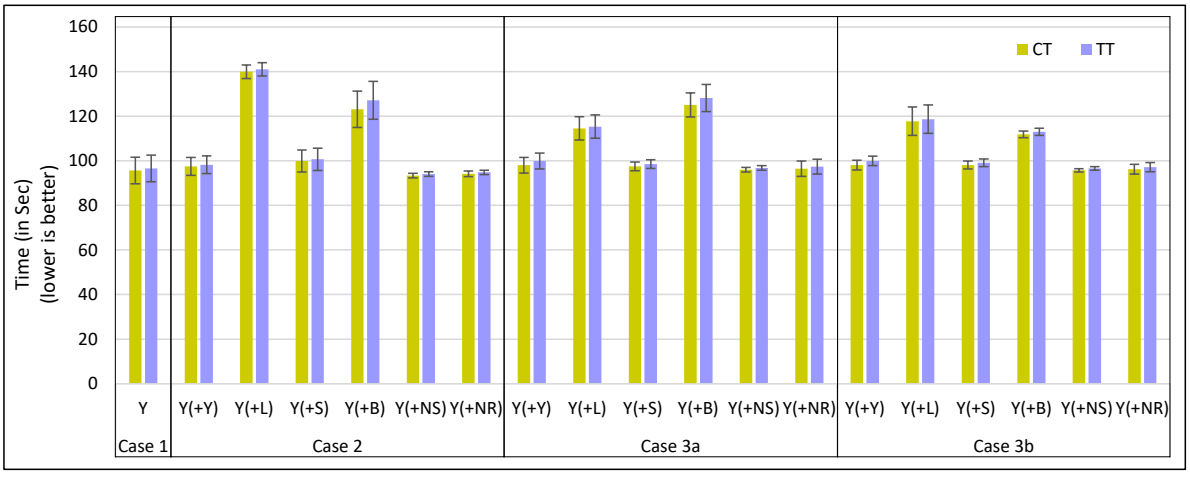

FIGURE 7 Y-Cruncher Performance Result for Computation Time (CT) and Total Time (TT). Black Bars represent the SD.

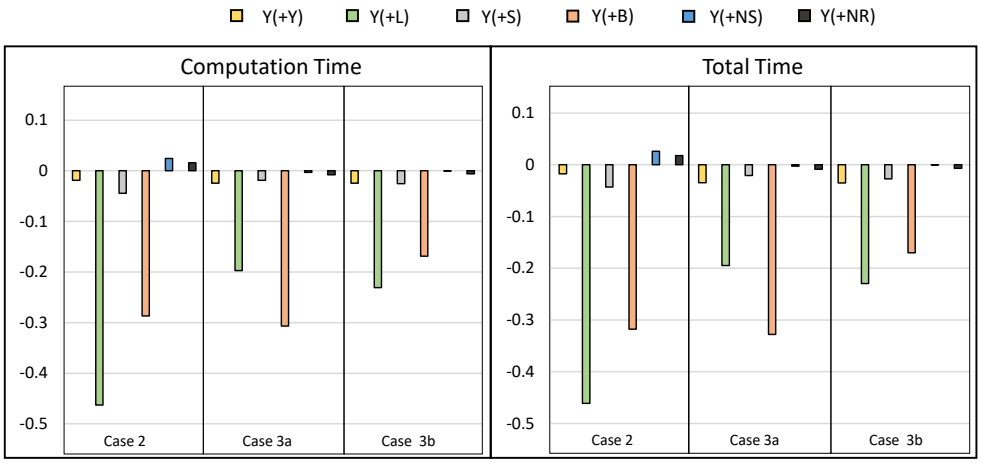

FIGURE 8 Y-Cruncher Interference Ratio (IR) Values. Horizontal axis labels represent various cases.

value as the Median and Trimmed Mean are almost same as the Mean. The Maximum and Minimum values are also similar to the Mean except for the case of the collocated execution of two STREAM instance.

For the SCALE and ADD operation, there is a slight difference for various scenarios. For SCALE operations the worst performance is S(+S) with 7.6% and 7% degradation for Case 3a and Case 3b respectively followed by 3.7% for Case 2. Other performances are comparable with a maximum gain of 1% for Case 2 S(+L). Similarly, for the ADD operation the worst performance is noticed for collocated execution of STREAM with a degradation of 13%, 12% and 7% for Case 3a, Case 3b and Case 2 respectively. Maximum performance gain is observed for S(+NS) followed by S(+L) with an increment of 12% and 10% respectively. However, for TRIAD operations a significant performance deviation is observed but follows the same trend of performance degradation for collocated execution of the same type of microservice. The maximum performance degradation is observed in S(+S) for Case 3a (5%) followed by Case 3b (4%) and Case 2 (2.5%). The effect of interference in terms of IR is given in Figure 10.

Overall, the execution of STREAM microservices in different scenarios does not show a significant variation from the baseline performance. A small performance gain is achieved when STREAM is collocated with different microservices inside a container. The performances are comparable in Cases 3a and 3b for different scenarios.

C. *Disk I/O Performance Evaluation and Analysis*

The I/O performance is represented using the Bonnie++ microservice which generates a dataset of at least twice the size of available memory (RAM). The performance for Sequential Block Input, Block Output, Block Rewrite and Random Seeks is presented in Table 4, Table 5, Table 6 and Table 7 respectively. For Block Input the performance is affected by the collocated execution of other microservices. The maximum performance degradation is observed for two instance of Bonnie++ with a loss of 56.92%, 56.17% and 56.13% for Case 2, 3a and 3b respectively. This high degradation has occurred because of the common disk which is shared by all the microservices. The least interference is noticed for the collocated execution of Bonnie++ with Netperf (NS, NR) with performance loss of only (7%, 1%), (5%, 6%) and (24%, 17%) for Case 2, Case 3a and Case 3b respectively. Table 4 also show that the results are consistent as there is a only a minimal difference between Mean, Median and Trimmed Mean values except for Case 3b B(+Y) and Case 2 B(+Y) with SD value of 77770.2 and 25319.8 and with CV of 24.8% and 8.1% respectively. In these situations, Median and Trimmed Mean are more appropriate measures to represent the average values. The interference effect is presented in Figure 11.

For Block Output operations, a slight performance gain is observed for heterogeneous execution of microservices for Case 2 with a maximum performance gain of 11.6% for Y(+NS) followed by 3.9% for B(+S). The performance of multiple instance of Bonnie++ is worst with a maximum loss of 46.48% for Case 3a followed by 44.5% and 42.6% for Case 3b and Case 2 respectively. The remaining performances are comparable to the baseline performance.

The result of Block Rewrite follows the trend of Block Input and is given in Table 6. The worst performance is observed for Case 2 B(+B) with 55.7% followed by Case 3a B(+B) with 55.3% performance loss. The least performance loss is noticed for Case 3a B(+L) with a degradation of only 7.7%. Similar performance is witnessed for Random Seeks with only a small performance gain of 0.4% for Case 2 Y(+NS). For all other

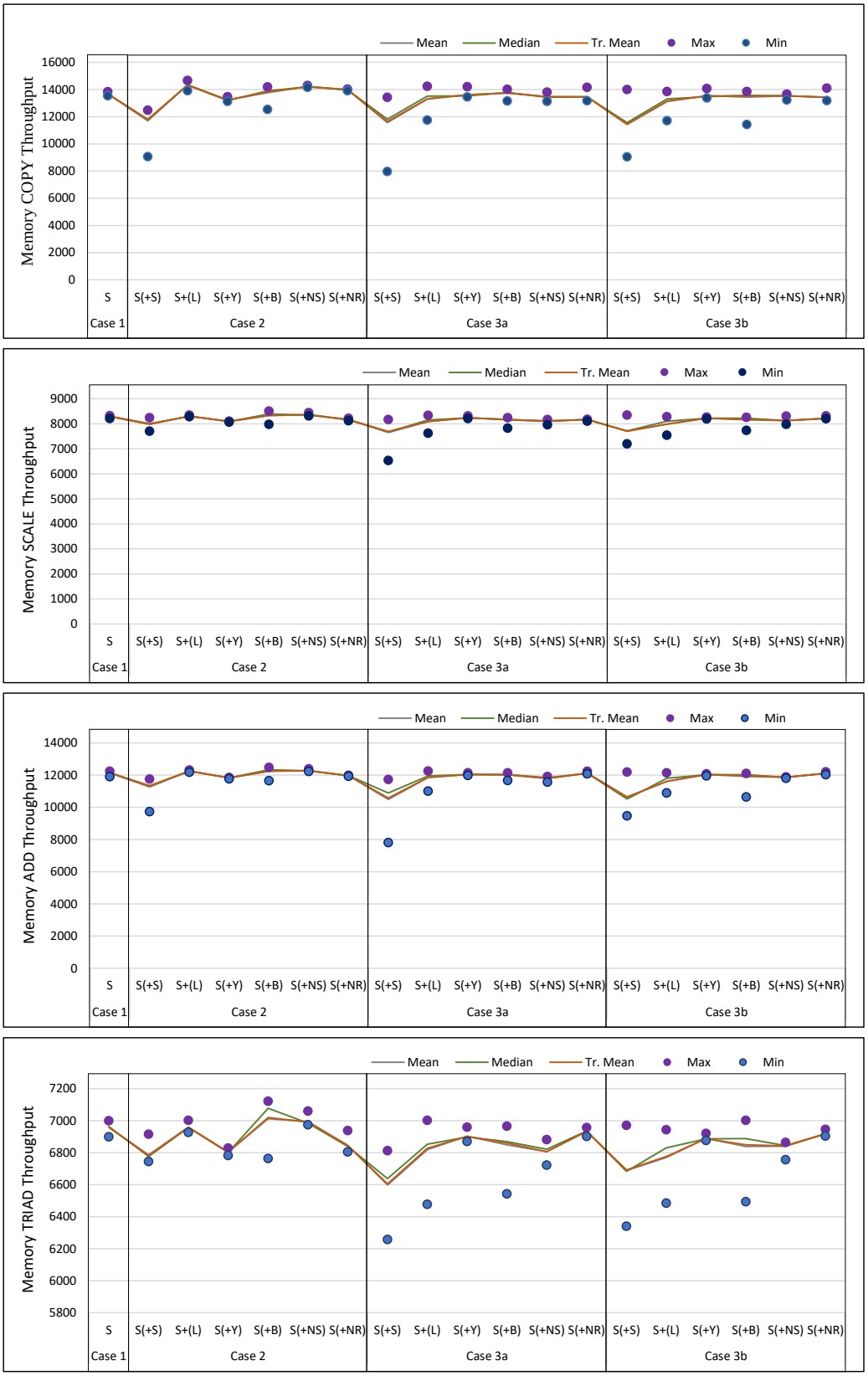

FIGURE 9 STREAM Performance Result

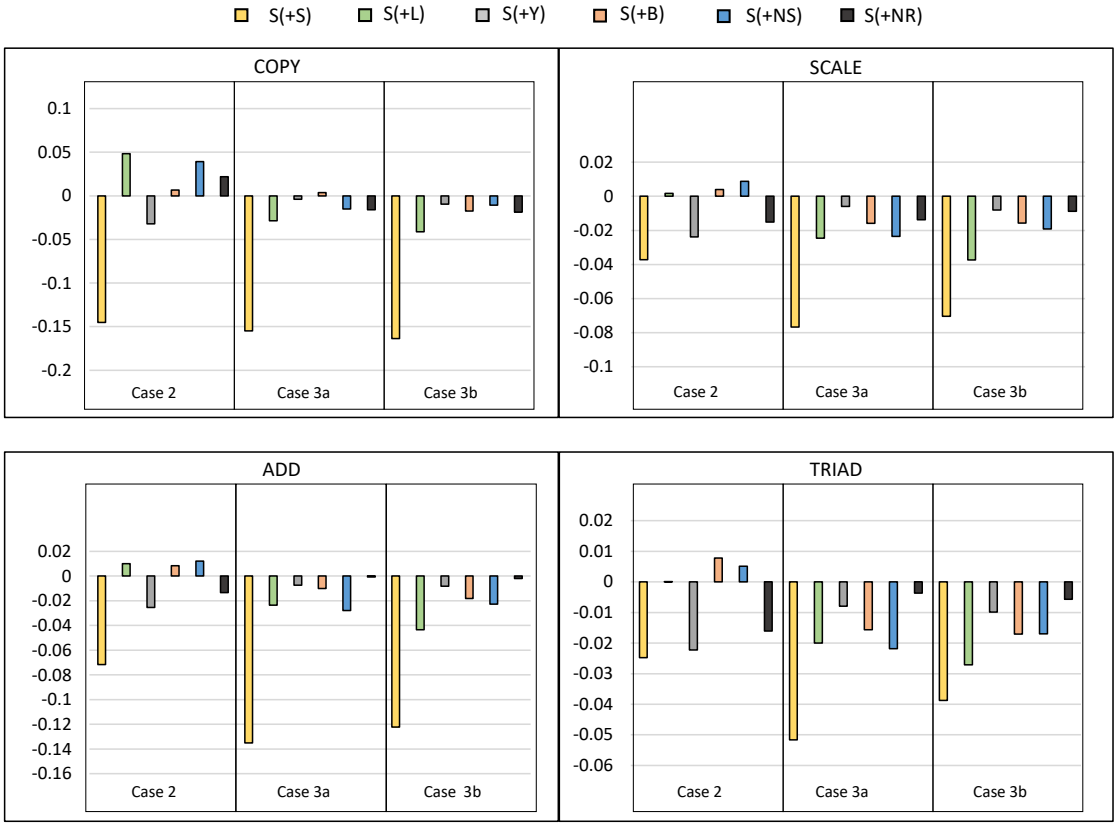

FIGURE 10 STREAM Interference Ratio (IR) Values. Horizontal axis labels represent various cases.

scenarios there is a performance loss with a maximum of 66.9% for Case 2 B(+B). There is one important point to note here in that there is a large variation in the results as shown by the SD (CV) values (e.g., in Case 3a B(+L), the SD (CV) is 1782.48 (19.2%)).

D. *Network Performance Evaluation and Analysis*

The Netperf microservice is used to analyze the system network performance. Neperf uses client/server architecture for data transfer and in our test case one container acts as a server that runs the *netserver* application of Netperf while another container acts as a client running the *netperf* application. A data stream is transferred from client to server for a defined duration of 120 seconds using TCP and the network performance is analyzed. The throughput of Request Response is also analyzed for the defined configuration. The experimental results showing the performance of TCP Stream and TCP RR are presented in Figure 12.

The results in Figure 12 show that the average throughput for Netperf TCP-Stream is always affected by the co-execution of other microservices. On average the maximum degradation is observed for multiple microservices executing inside a container (Case 2) with an average performance loss of 42.8%. The worst performance is noticed for NS(+Y) with a degradation of 60.4%. For other cases also, there is a large performance degradation for the co-execution of TCP Stream with Y-Cruncher with an average loss of 38.3% and 41.4% for Case 3a and Case 3b respectively. This is due to the fact that Y-Cruncher stresses both CPU and memory together while TCP Stream also accesses memory and CPU resources for transferring continuous data streams, thus leading to strong performance interference. For the execution of two instances of TCP Stream in different containers the performance is comparable with the baseline performance with only a small degradation of 1% and 8% for Case 3a and 3b respectively. However, a large degradation of 36% is observed for collocated execution inside a container (Case 2). The result also shows a significant performance difference for other scenarios in Case 2.

For TCP RR, the result is different from that of TCP Stream. Most of the performances are comparable with the baseline with the exception of two instances of TCP Stream executing inside a container (Case 2 NR(+NR)) with a performance degradation of 22% from the baseline. For this scenario the result shows a large variation with Mean and Median values significantly different. For most other cases these values are almost

TABLE 4 Bonnie++ Block Input Result.

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	348948.7	346912.5	349092.2	366590	328725	8816.5	0.025
Case 2	B(+B)	150318.6	151020.0	150499.3	153749	143634	2707.9	0.018
	B(+L)	280510.3	279708.0	280276.2	300307	264927	9073.6	0.032
	B(+Y)	310682.1	306304.0	307027.5	401019	286128	25319.8	0.081
	B(+S)	293854.5	294489.5	294033.4	304766	279722	7891.1	0.027
	B(+NS)	321789.2	322759.5	322253.7	333090	302127	8111.1	0.025
	B(+NR)	345039.6	345520.0	344307.0	379975	323291	12765.5	0.037
Case 3a	B(+B)	152934.3	153021.5	152967.7	155528	149739	1549.4	0.010
	B(+L)	268383.2	271084.0	268774.8	291607	238110	16004.1	0.060
	B(+Y)	292321.3	292458.0	292357.1	306960	277038	7988.0	0.027
	B(+S)	291840.9	293848.5	292551.0	303054	267845	8129.0	0.028
	B(+NS)	330460.5	329437.0	330246.0	367948	296834	14219.2	0.043
	B(+NR)	326684.5	326248.0	326754.7	343819	308286	11378.8	0.035
Case 3b	B(+B)	150259.9	149811.0	150222.7	159286	141903	5348.0	0.036
	B(+L)	292410.9	292421.5	292703.9	301705	277842	6226.1	0.021
	B(+Y)	313616.1	296750.5	297370.1	639995	279666	77770.2	0.248
	B(+S)	294275.9	295652.0	295235.1	302914	268373	7984.3	0.027
	B(+NS)	263074.5	266438.0	264483.2	285272	215519	17366.2	0.066
	B(+NR)	288610.8	293233.0	289713.6	310925	246446	16459.5	0.057

TABLE 5 Bonnie++ Block Output Result.

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	278362.4	277073.0	278099.4	294574	266885	8425.81	0.030
Case 2	B(+B)	159530.5	152429.5	152711.6	294041	147760	31809.07	0.199
	B(+L)	283667.1	281342.5	281533.3	332479	273264	12189.84	0.043
	B(+Y)	281957.0	283641.0	281708.0	303000	265396	8224.12	0.029
	B(+S)	289314.7	289074.0	289009.1	305278	278851	8192.07	0.028
	B(+NS)	310772.1	309600.5	309897.6	350216	287068	14772.71	0.048
	B(+NR)	283923.6	285201.5	283682.2	297530	274662	6981.28	0.025
Case 3a	B(+B)	148960.7	148394.0	148732.7	161455	140569	4818.72	0.032
	B(+L)	264280.2	263157.0	262325.1	290663	243089	11910.16	0.045
	B(+Y)	252390.3	252859.5	252346.7	293046	232520	4665.51	0.018
	B(+S)	270180.9	279268.5	269561.9	286893	254610	8232.98	0.030
	B(+NS)	262805.1	268557.0	262225.2	273535	242514	12189.49	0.046
	B(+NR)	255982.2	256025.0	256512.7	288801	233615	10050.62	0.039
Case 3b	B(+B)	154354.5	153829.0	153333.8	177870	149210	6037.06	0.039
	B(+L)	269236.3	268707.5	268775.0	286873	249902	13364.55	0.050
	B(+Y)	252809.9	251839.0	252907.3	264715	239152	5977.67	0.024
	B(+S)	270158.8	268596.5	269212.7	291482	265866	9354.95	0.035
	B(+NS)	269616.7	270105.5	269896.2	283752	250450	9592.73	0.036
	B(+NR)	257024.6	257307.5	257125.4	266154	246080	5303.87	0.021

same. The best performance is noticed for the execution of the two instance of TCP RR for Case 3a and 3b with a performance loss of only 6% and 2% respectively. The overall interference effect is presented in Figure 13.

TABLE 6 Bonnie++ Block Rewrite Result.

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	149159.4	149094.0	149202.7	153877	143663	2841.94	0.019
Case 2	B(+B)	66082.6	66379.5	66575.9	67674	55611	2546.87	0.039
	B(+L)	129841.8	129942.0	129935.8	134415	123577	3087.65	0.024
	B(+Y)	125820.3	125511.0	125729.9	131053	122215	2721.36	0.022
	B(+S)	131675.6	131611.0	131756.6	137909	123983	4193.53	0.032
	B(+NS)	131091.2	131262.0	131171.2	136593	124149	3128.53	0.024
	B(+NR)	127168.5	126673.5	127022.2	136821	120149	4287.56	0.034
Case 3a	B(+B)	66617.6	66792.5	66614.1	68768	64529	1223.94	0.018
	B(+L)	132061.0	132441.5	132218.2	139880	121412	4558.59	0.035
	B(+Y)	126750.6	126659.0	126806.7	131635	120855	2589.28	0.020
	B(+S)	132399.4	131666.5	132319.7	139712	126521	3440.37	0.026
	B(+NS)	134379.0	133777.5	133984.6	147012	128845	4553.22	0.034
	B(+NR)	111762.8	123709.5	116469.7	127234	11566	34380.04	0.308
Case 3b	B(+B)	67366.5	67807.5	67362.8	70339	64459	1627.47	0.024
	B(+L)	137653.4	137873.5	137813.6	142397	130027	3186.39	0.023
	B(+Y)	133628.2	132893.0	133397.9	142822	128580	3435.94	0.026
	B(+S)	136741.4	137559.5	136850.7	144602	126913	3577.91	0.026
	B(+NS)	126108.8	127890.5	127169.7	131055	102065	6403.13	0.051
	B(+NR)	128480.1	129910.0	128663.6	134240	119417	4381.08	0.034

TABLE 7 Bonnie++ Random Seeks Result.

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	10801.3	10807.4	10817.4	11890.7	9422.4	628.77	0.058
Case 2	B(+B)	3576.1	3538.0	3572.6	3901.1	3314.5	156.69	0.044
	B(+L)	9974.5	9918.3	9940.6	11877.4	8681.6	813.41	0.082
	B(+Y)	8895.7	8947.2	8915.1	9452.9	7988.8	370.89	0.042
	B(+S)	10274.4	10427.5	10330.5	11516.4	8022.4	779.58	0.076
	B(+NS)	10849.3	11048.3	10878.1	12582.4	8596.7	1224.63	0.113
	B(+NR)	9607.7	9925.2	9626.5	10670.6	8205.9	715.84	0.075
Case 3a	B(+B)	3912.7	3866.1	3905.5	4191.9	3763.3	131.87	0.034
	B(+L)	9290.7	9480.5	9447.6	11660.3	4096.4	1782.48	0.192
	B(+Y)	8779.4	8803.5	8822.2	9550.6	7239.3	488.66	0.056
	B(+S)	10401.3	10583.1	10409.4	11882.7	8773.5	766.95	0.074
	B(+NS)	8198.8	8130.5	81711.2	9231.5	7243.3	785.97	0.096
	B(+NR)	8652.5	8335.0	8720.6	9861.5	6218.2	844.55	0.098
Case 3b	B(+B)	4877.5	4745.0	4896.1	6149.2	3270.8	926.19	0.190
	B(+L)	9821.2	9619.9	9785.0	11232.2	9061.9	614.84	0.063
	B(+Y)	8234.6	8245.3	8249.7	8713.6	7483.6	293.17	0.036
	B(+S)	9803.9	10233.4	9948.3	11260.3	5748.7	1397.07	0.143
	B(+NS)	8180.1	8592.0	8282.0	9531.7	4994.1	1290.04	0.158
	B(+NR)	7642.7	7671.3	7664.9	8224.5	6660.5	451.41	0.059

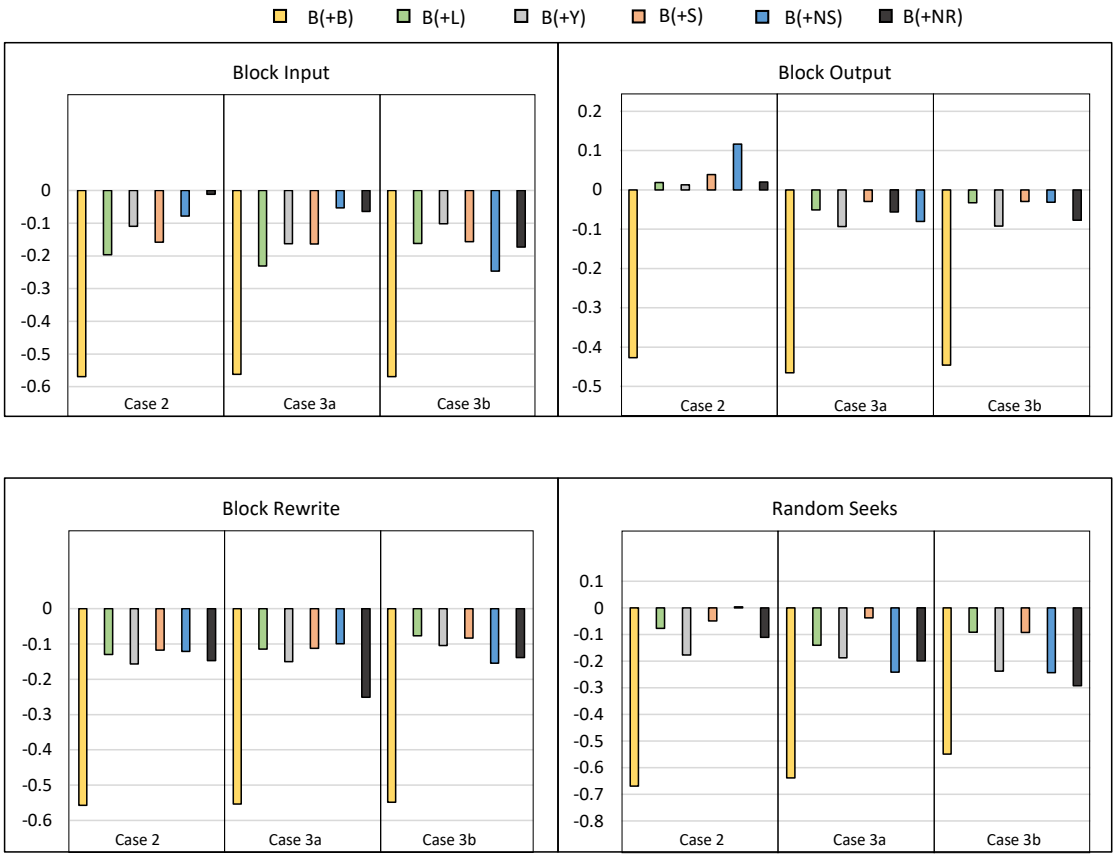

FIGURE 11 Bonnie++ Interference Ratio (IR) Values. Horizontal axis labels represent various cases.

6 | RELATED WORK

The concept of Container-based virtulization is not new and has roots going back to FreeBSD Jails[24] and Solaris Zones[25] that use the Unix chroot feature to provide operating system virtualization. Containers as a deployment environment are initially introduced by PaaS providers such as Heroku [4], DotCloud[26], CloudFoundry [5] and OpenShift [6] for deployment and isolation of different workloads. Here, containers are mainly used as overlays hosted on the top of VMs running on cloud servers[27]. The containers are simply treated as a process rather than a virtual server. The PaaS workloads (mostly elastic and stateless applications) are considered the classical applications for containers, but IaaS workloads (e.g., HPC workloads) can also take advantage of container technology. Bernstein[28] explains the benefits of containers for PaaS applications.

Numerous efforts[5,10,29,30] show that containerizing the cloud infrastructure leads to highly efficient and agile solutions. Evident from the previous work is that containers can reduce the overall resource overhead while increasing the overall performance. The value of containers with respect to VMs is supported by different studies. These studies compare the performance of containers with respect to VMs for different benchmarks and show that the performance of the container is better than, or almost equal to, the performance of the VM. Xavier et al.[31] compared the performance of VM with container-based virtualization for HPC environment. The experiments are performed on Linux VServer, OpenVZ and LXC comparing Xen and bare-metal performance using NAS Parallel Benchmark (NPB). The results show that container-based virtualization has near native performance for different fundamental components (CPU, memory, disk and network). Felter et al.[5] perform similar experiments with Docker in comparison to KVM using different benchmarks. These results show that for CPU and memory the performance of a Docker container is comparable to VM but for I/O and network intensive applications Docker's performance is better than VM. Similar studies are performed by Morabito et al.[10], but here LXC and OSv is also compared with Docker and KVM. They conclude that LXC outperforms KVM and Docker in almost all cases. A similar study is given by Li et al.[32] that uses DoKnowMe evaluation strategy to compare the performance of KVM and Docker and

[4]https://www.heroku.com
[5]https://www.cloudfoundry.org/
[6]https://www.openshift.com/

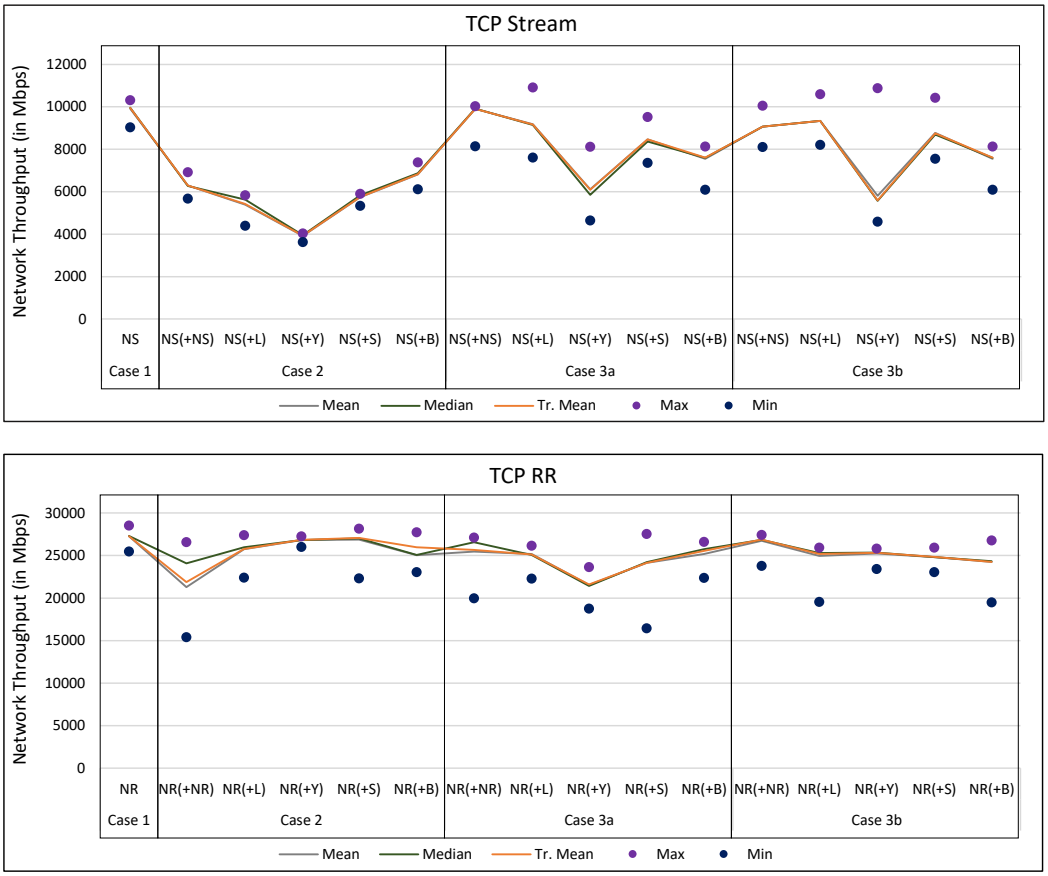

FIGURE 12 Netperf Performance Result

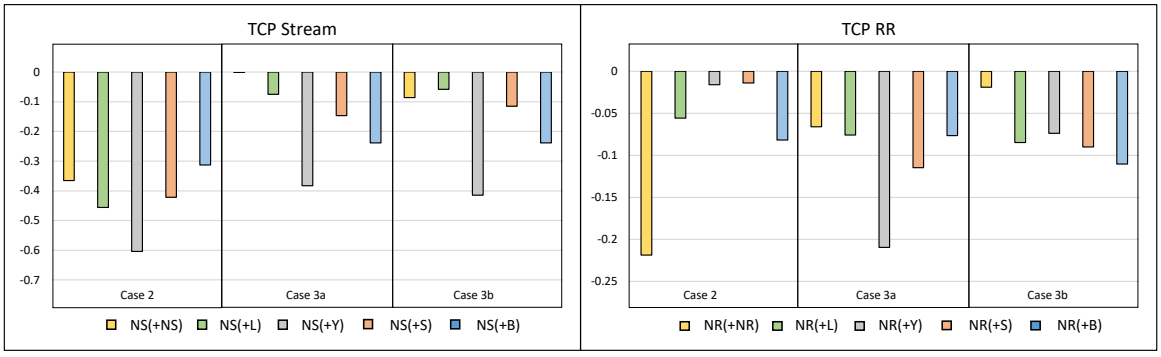

FIGURE 13 Netperf Interference Ratio (IR) Result. Horizontal axis labels represents various cases.

illustrates that the effect of virtualization depends not only on features but also on job types. These results show that the average performance of a container is similar and sometimes better than the VM and also shows a significant degree of performance variation in the case of containers given varying job types. Similar comparative studies between bare-metal, VM and container performed on OpenStack is presented in [33]. These results show that Docker has fastest boot-up time and the performance is comparable with bare-metal except for network evaluation. The VM has a high overhead that increases with the workload size and assigned resources.

A study by Kozhirbayev et al.[34] evaluates the performance of Docker and Flockport running different benchmarks and shows that Flockport outperforms Docker in almost all cases. The study in [35] compares the power consumption of container and VM and shows that both types of

virtualization have similar power consumption for idle situations or for CPU/memory operations but containers consume less power for network intensive operations. Cuadrado-Cordero [36] compared the QoS and energy performance of Docker containers and KVM for different services. These experimental results show that Docker allows more services to run compared to KVM. These results also show that Docker consumes less energy than KVM promoting energy savings.

Few of the works consider the running of HPC workloads in Docker containers. Jacobsen et al. [37] advocate the use of containers for HPC environments. The work in [6] shows how to orchestrate multiple containers on a physical node. This study confirms that a job can be transparently executed inside a Docker container without having any knowledge about the underlying host configuration. The study is validated by running Linpack inside the container. Ruiz et al. [7] evaluate the performance of LXC containers using the NAS parallel benchmark. In these experiments containers in different configurations (i.e., isolated inter-container and multi-node inter-container) are considered for performance evaluation. The results conclude that inter-container communication is faster than physical machine communication but there is a degradation of CPU performance for memory-intensive operations.

Few of the studies consider big data applications for comparing the performance of containers [38 39 40]. Bhimani et al. [38] compare the performance of VMWare and Docker for different big data applications using Spark. The experimental results show that Docker achieves a speed-up for map- and reduce-intensive applications but not for shuffle-intensive applications. Zhang et al. [40] also presented a similar study where an extensive comparison between VM and Docker is presented for different big data applications. The results show that Docker containers are more convenient, highly scalable, and achieve higher system utilization as compared to VMs.

Most of the studies presented in the literature do not consider the effect of interference in containerized environments. Sharma et al. [41] considers the interference for performance evaluation. Their study compares the performance of collocated applications on a common host but only when 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 containers. Ye et al. [39] also consider the inter-container interference for big data applications (Spark). Similar work is done by Kejiang et al. [40] to evaluate the performance of big data applications by changing the cgroups system configuration while considering the interference between containers using different Spark applications (e.g., K-Means, Page rank).

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 executing inside the Docker container. The obtained results present the performance variation while running single or multiple collocated (competing or independent) microservices. Our evaluation gives an understanding of interference effects caused by microservices running either in the same container or in separate containers. This also gives a suggestion about how microservices may be combined with minimal interference for gaining better resource utilization.

7 | DISCUSSION AND CONCLUSIONS

With the combination of virtualization advantages and bare-metal performance, containers are treated as a feasible alternative to VMs in cloud environments. They bind all the required supporting software required for an application along with the application itself into a container image that can easily be deployed and executed in different environments. These advantages can be easily utilized to package HPC microservices, which usually have complex software and hardware requirements. However, execution of microservices in containerized environments may cause interference that lead to performance degradation. Therefore, it is necessary to understand the behaviour of microservices executing in containerized environments.

In this paper, we investigated the performance of HPC microservices in Docker container environments. Our main focus is to analyze the effect of interference on HPC microservices executing within inter-container and intra-container deployments. Our results present a comprehensive study into the performance variation of containerized microservices. The main findings are given below:

- Executing multiple microservices inside a container is a feasible deployment option as the result shows that the performance is better than the baseline performance for some cases.
- The interference caused by microservices with similar resource requirements are always higher as compared to those microservices with different resource requirements. This effect of interference is higher for intra-container scenarios than inter-container scenarios. The performance in intra-container scenario is worst for network-intensive stream operations.
- The results show that core CPU intensive operations can cause least interference with memory, disk and stream network operations. For memory intense operations, network intensive operations give best performance while mixed CPU + memory requirement operations gives the worst performance. For the combined CPU + memory intensive operations, network intensive operations cause the least performance

interference while the core CPU intensive operations cause the highest performance degradation. I/O intensive operations are minimally affected by other types of microservices and have comparable performance for all cases. Finally, for network-intensive operations the worst performance is noticed with combined CPU + memory operations. One important point to note for network intensive operations is that they are less affected when executed in separate containers.

- The results show that the performance of containerized microservices are comparable with either cgroups enabled or disabled if the system resources in both cases are exactly same.

8 | FUTURE DIRECTIONS

In our future work we will compare the performance of Docker containers with VM for HPC microservices. We will also investigate the performance of other container technology (e.g. LXC, uDocker, Socker, Singularity) and compare them with the performance of Docker and VM. We will also study the variation of inter-host communication performance in a clustered environment (e.g. Kubernetes, Docker Swarm) for HPC applications. Additionally, we will aim to benchmark the power consumption of containers for different interference conditions. Further research to develop a framework will be explored that takes into consideration interference effects while making provisioning decisions for microservice-based application in containerized environment. Ultimately, we will map microservices into a containerized environment to find eligible deployment plans considering different microservice requirements while minimising the effect of interference.

Source Code

All the scripts for running our benchmark experiments are available at https://github.com/DNJha/CCPE-DockerBenchmarkCode.git.

Financial disclosure

None reported.

Conflict of interest

The authors declare no potential conflict of interests.

References

1. Xing Yuping, Zhan Yongzhao. Virtualization and cloud computing. Paper presented at: Future Wireless Networks and Information Systems; 305–312; 2012.
2. Kivity Avi, Kamay Yaniv, Laor Dor, Lublin Uri, Liguori Anthony. kvm: the Linux virtual machine monitor. Paper presented at: Proceedings of the Linux symposium; 225–230; 2007.
3. Gulati Ajay, Holler Anne, Ji Minwen, Shanmuganathan Ganesha, Waldspurger Carl, Zhu Xiaoyun. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*. 2012; 1(1); 45–64.
4. Soltesz Stephen, Pötzl Herbert, Fiuczynski Marc E, Bavier Andy, Peterson Larry. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. Paper presented at: ACM SIGOPS Operating Systems Review; 275–287; 2007.
5. Felter Wes, Ferreira Alexandre, Rajamony Ram, Rubio Juan. An updated performance comparison of virtual machines and linux containers. Paper presented at: 2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS); 171–172; 2015.
6. Higgins Joshua, Holmes Violeta, Venters Colin. Orchestrating docker containers in the HPC environment. Paper presented at: International Conference on High Performance Computing; 506–513; 2015.
7. Ruiz Cristian, Jeanvoine Emmanuel, Nussbaum Lucas. Performance evaluation of containers for HPC. Paper presented at: European Conference on Parallel Processing; 813–824; 2015.

8. Pahl Claus, Jamshidi Pooyan. Microservices: A Systematic Mapping Study.. Paper presented at: CLOSER (1); 137–146; 2016.
9. Fazio Maria, Celesti Antonio, Ranjan Rajiv, Liu Chang, Chen Lydia, Villari Massimo. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing.* 2016; 3(5); 81–88.
10. Morabito Roberto, Kjällman Jimmy, Komu Miika. Hypervisors vs. lightweight virtualization: a performance comparison. Paper presented at: 2015 IEEE International Conference on Cloud Engineering (IC2E); 386–393; 2015.
11. Jha Devki Nandan, Garg Saurabh, Jayaraman Prem Prakash, Buyya Rajkumar, Li Zheng, Ranjan Rajiv. A Holistic Evaluation of Docker Containers for Interfering Microservices. Paper presented at: 2018 IEEE International Conference on Services Computing (SCC); 33–40; 2018.
12. Li Zheng, O'Brien Liam, Zhang He. CEEM: A practical methodology for Cloud services evaluation. Paper presented at: 2013 IEEE Ninth World Congress on Services (SERVICES); 44–51; 2013.
13. Biederman Eric W, Networx Linux. Multiple instances of the global linux namespaces. Paper presented at: Proceedings of the Linux Symposium; 101–112; 2006.
14. Docker: A 'Shipping Container' for Linux code https://www.linux.com/news/docker-shipping-container-linux-code.
15. Li Zheng, OBrien Liam, Ranjan Rajiv, Zhang Miranda. Early observations on performance of Google compute engine for scientific computing. Paper presented at: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom); 1–8; 2013.
16. Li Zheng, Mitra Karan, Zhang Miranda, et al. Towards understanding the runtime configuration management of do-it-yourself content delivery network applications over public clouds. *Future Generation Computer Systems.* 2014; 37; 297–308.
17. Li Zheng, OBrien Liam, Cai Rainbow, Zhang He. Towards a taxonomy of performance evaluation of commercial Cloud services. Paper presented at: 2012 IEEE 5th International Conference on Cloud Computing (CLOUD); 344–351; 2012.
18. Intel Math Kernel Library Benchmarks https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite/.
19. Y-Cruncher - A Multi-Threaded Pi-Program http://www.numberworld.org/y-cruncher/.
20. STREAM benchmark http://www.cs.virginia.edu/stream/.
21. Bonnie++ https://www.coker.com.au/bonnie++/.
22. The Netperf Homepage https://hewlettpackard.github.io/netperf/.
23. Li Zheng, O'Brien Liam, Zhang He, Cai Rainbow. A factor framework for experimental design for performance evaluation of commercial cloud services. Paper presented at: 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom); 169–176; 2012.
24. Kamp Poul-Henning, Watson Robert NM. Jails: Confining the omnipotent root. Paper presented at: Proceedings of the 2nd International SANE Conference; 2000.
25. Beck John, Comay David, Ozgur L, et al. *Virtualization and Namespace Isolation in the Solaris Operating System (PSARC/2002/174).* 2006.
26. Dua Rajdeep, Raja A Reddy, Kakadia Dharmesh. Virtualization vs containerization to support paas. Paper presented at: 2014 IEEE International Conference on Cloud Engineering (IC2E); 610–614; 2014.
27. Hindman Benjamin, Konwinski Andy, Zaharia Matei, et al. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. Paper presented at: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI); 295–308; 2011.
28. Bernstein David. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing.* 2014; 1(3); 81–84.
29. Kolla https://wiki.openstack.org/wiki/Kolla.
30. Bankole Kalonji, Krook Daniel, Murakami Shaun, Silveyra Manuel. *A practical approach to dockerizing OpenStack high availability.* 2014.

31. Xavier Miguel G, Neves Marcelo V, Rossi Fabio D, Ferreto Tiago C, Lange Timoteo, De Rose Cesar AF. Performance evaluation of container-based virtualization for high performance computing environments. Paper presented at: 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 233–240; 2013.

32. Li Zheng, Kihl Maria, Lu Qinghua, Andersson Jens A. Performance overhead comparison between hypervisor and container based virtualization. Paper presented at: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA); 955–962; 2017.

33. Kominos Charalampos Gavriil, Seyvet Nicolas, Vandikas Konstantinos. Bare-metal, virtual machines and containers in OpenStack. Paper presented at: 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN); 36–43; 2017.

34. Kozhirbayev Zhanibek, Sinnott Richard O. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems.* 2017; 68; 175–182.

35. Morabito Roberto. Power Consumption of Virtualization Technologies: An Empirical Investigation. Paper presented at: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC); 522–527; 2015.

36. Cuadrado-Cordero Ismael, Orgerie Anne-Cécile, Menaud Jean-Marc. Comparative experimental analysis of the quality-of-service and energy-efficiency of VMs and containers' consolidation for cloud applications. Paper presented at: International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2017); 1-6; 2017.

37. Jacobsen Douglas M, Canon Richard Shane. *Contain this, unleashing docker for hpc.* 2015.

38. Bhimani Janki, Yang Zhengyu, Leeser Miriam, Mi Ningfang. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. Paper presented at: 2017 IEEE High Performance Extreme Computing Conference (HPEC); 1–7; 2017.

39. Ye Kejiang, Ji Yunjie. Performance tuning and modeling for big data applications in Docker containers. Paper presented at: 2017 International Conference on Networking, Architecture, and Storage (NAS); 1–6; 2017.

40. Zhang Qi, Liu Ling, Pu Calton, Dou Qiwei, Wu Liren, Zhou Wei. A Comparative Study of Containers and Virtual Machines in Big Data Environment. Paper presented at: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD); 178-185; 2018.

41. Sharma Prateek, Chaufournier Lucas, Shenoy Prashant, Tay Y. C.. Containers and virtual machines at scale: A comparative study. Paper presented at: Proceedings of the 17th International Middleware Conference (Middleware '16); 1–13; 2016.

How to cite this article: D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, G. Morgan, and R. Ranjan (xxxx), A Study on the Evaluation of HPC Microservices in Containerized Environment, *xxxxxxxxx*, *XXXX;XX:XX–XX*.