Received: 3 August 2018 | Revised: 3 March 2019 | Accepted: 11 April 2019
DOI: 10.1002/cpe.5323

WILEY

SPECIAL ISSUE PAPER

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, Newcastle upon Tyne, UK
[2]University of Tasmania, Hobart, Australia
[3]Swinburne University of Technology, Melbourne,Australia
[4]The University of Melbourne, Melbourne, Australia
[5]University of Concepción, Concepción, Chile

Correspondence
Devki Nandan Jha, School of Computing, Newcastle University, Newcastle upon Tyne NE4 5TG, UK.
Email: D.N.Jha2@newcastle.ac.uk

Summary

Containers are gaining popularity over virtual machines 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 possibility of interference between different microservices hosted within the same container (intracontainer) and different containers (intercontainer) 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 intracontainer and intercontainer 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 behavior of HPC microservices in the interfering containerized environment.

KEYWORDS

container, docker, interference, microservice, 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: (1) it supports heterogeneous applications to run on one physical environment, which is not otherwise possible; (2) it allows multiple tenants to share the physical resources that in turn increases the overall resource utilization; (3) tenants are isolated from each other, promoting the guarantee of quality-of-service (QoS) requirements; (4) it eases the allocation and maintenance of resources for each tenant; (5) it enables resource scaleup or scaledown depending on the dynamically changing application requirements; and (6) it 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 types 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) (eg, KVM[2] and VMWare[3]). Each VM possesses an isolated operating system (OS) allowing heterogeneous consolidation of multiple applications. However, the advantages of virtualization are provided at a cost of additional overhead as compared to the nonvirtualized system. Since there are two levels of abstraction (top level by a VM OS and bottom level by physical host machine) any delay incurred by the VM layer can not be removed. Current research trends concentrate on reducing the degree of performance variation due to such overhead

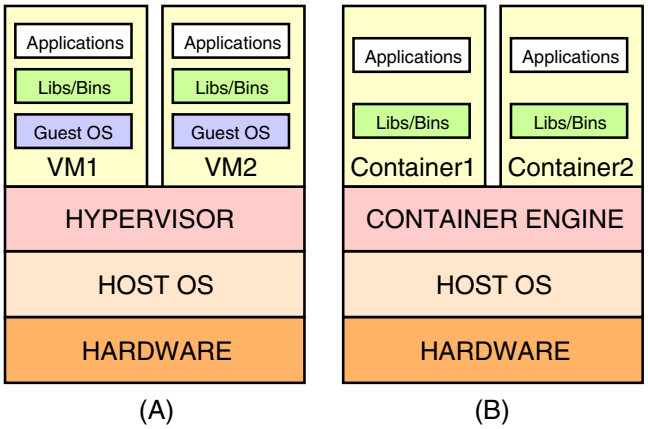

FIGURE 1 Basic architecture of hypervisor-based and container-based virtualizations. A, Hypervisor-based virtualization; B, Container-based virtualization

between virtualized and bare-metal systems.[4] Containers provide virtualization advantages by exploiting the services provided by the host OS (eg, LXC* and Docker†). Except for applications that require strict security requirements, containers becomes a viable alternative for VMs. Figure 1 represents the basic architectural difference between hypervisor-based and container-based virtualization.

Different features provided by the containers (eg, lightweight, self-contained, and fast start-up and shutdown) makes them a popular choice for virtualization. Recent research findings[5-7] 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 OS version, underlying compilers and particular environment variables, thus making 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 lightweight 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 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 reduction in intercontainer data transfer delay, increased utilization of resources, and avoidance of any dependency shortcomings. This scenario is suitable for HPC workloads where the resource requirements for each components/microservice are fixed and known a priori. However, the performance of containerized microservices may be affected by other microservices running inside the same container causing intracontainer interference. The performance of microservices running in separate containers may also be affected because of intercontainer 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 intracontainer and intercontainer 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 microbenchmarks running in containerized environments,[5-7,10] but they normally consider only 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 (RQs):

RQ 1: **How does the performance of Docker containers executing HPC microservices vary while running in the intracontainer or the intercontainer 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

* https://www.linuxcontainers.org
† https://www.docker.com

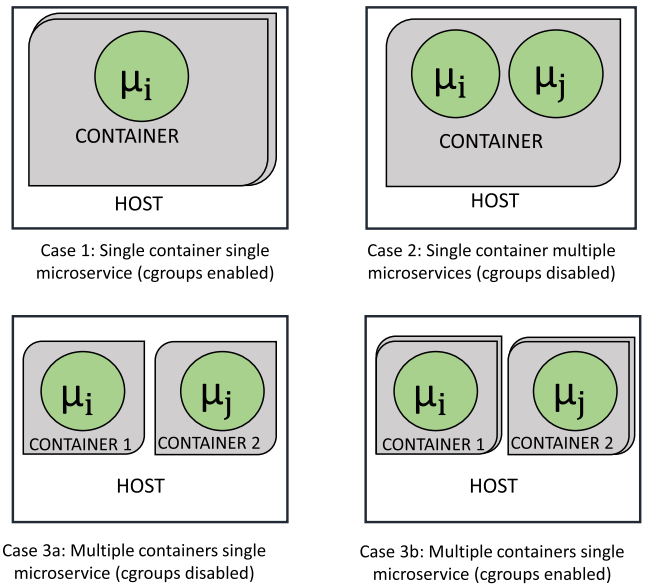

FIGURE 2 Test case scenarios

signify the deployment of multiple containers on one host machine with control group (cgroups) constraints disabled or enabled. More details are presented in Section 4.1. Irrespective of the type of HPC application (eg, message passing interface or MapReduce application), these are the basic deployment scenarios for microservices executing on a single host machine. Each application has different interhost 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. Interhost network communication for a specific HPC application is beyond the scope of this paper.

A set of microbenchmarks is considered to represent the behavior of HPC application components where each microbenchmark is specific for a particular resource type. For our purposes, the microbenchmarks 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) microbenchmarks, respectively. We also consider another microbenchmark Y-Cruncher, which depends on both CPU and memory of the system. All these microbenchmarks 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 intracontainer 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 (intracontainer 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 intercontainer environment. Two containers running in parallel can cause interference and the effect of interference depends on the type of microservice that 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 intracontainer performance. This result can be used for modeling smart container resource provisioning techniques to minimize the interference effect.

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-based and hypervisor-based virtualizations. Different containers can share the same physical resources, but from a hosted application's point of view, each container has their own autonomous OS running independently.

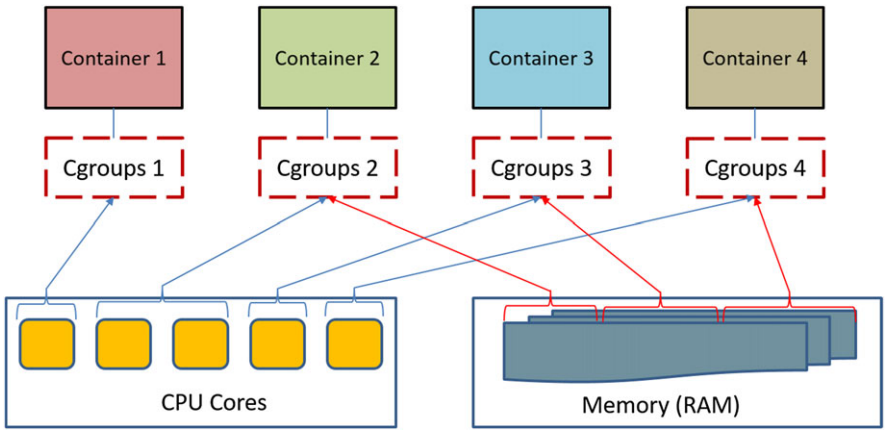

FIGURE 3 Resource restrictions provided by the control groups (cgroups)

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, and IPC are some common namespace features used by containers to provide abstraction for process IDs, file-system mount points, network features, and interprocess communications, respectively, in an intercontainer environment.[13] Each new container uses the clone() system call to create an abstract system of an existing namespace in the OS 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 on 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, Advanced Multi-layered Unification Filesystem (AUFS), 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

To investigate the performance of heterogeneous HPC microservices running in a container (such as Docker), we followed the 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 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 benchmark listing**: List all the metrics and benchmarks that may be used for the proposed evaluation.
4. **Metrics and benchmark selection**: Select suitable metrics and benchmarks for the proposed evaluation.
5. **Experimental factor listing**: List all the factors that may be involved in the evaluation experiments.
6. **Experimental factor 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 RQs 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 setup an intracontainer configuration.

Case 3. *Multiple containers each running one microservice.* We specified two subases:

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 that a container can use is limited by specifying cgroups restrictions.

We call this setup an intercontainer 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 VMs. Following the cloud service evaluation strategy,[17] we examine fundamental resource parameters, ie, CPU computation, memory, block 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 behavior. The selection of benchmarks depends on the chosen metrics and is 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 floating-point operations per seconds (FLOPS), 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 (FLOPS) 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 runtime 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 runtime parameters.
2. *Memory performance*: For memory performance, we considered STREAM[20] microbenchmark that measures the data throughput for different memory operations (eg, copy and 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 are presented in terms of megabytes per second.
3. *Disk I/O performance:* We considered disk throughput and random seeks as the suitable measures for evaluating disk I/O performance. To measure the disk throughput, we used the Bonnie++[21] microbenchmark, 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 performance between two hosts. We

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

TABLE 2 Metrics and benchmarks for selected resource types

Resource Type	Selected Metrics	Selected Benchmarks	Version
CPU	Floating point operations per second (FLOPS)	Linpack	Mklb_p_2018.0.006
	Total computation time	Y-Cruncher	0.7.5
	Total turnaround time		
Memory	Data throughput	STREAM	5.10
Disk I/O	Disk throughput	Bonnie++	1.03e
	Random seeks		
Network	Network throughput	Netperf	2.7.0

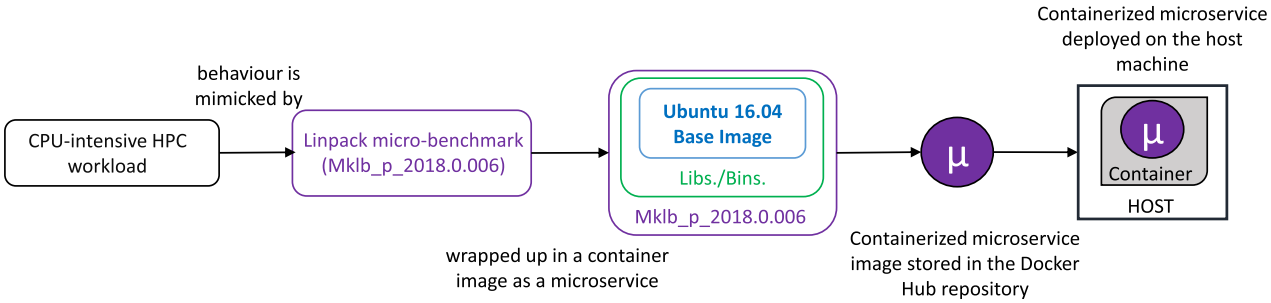

FIGURE 4 Steps for Linpack HPC microservice construction

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 megabits per second.

Table 2 summarizes the selected metrics and benchmarks for different resource types. For deployment, the microbenchmarks 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 microbenchmarks. Finally, the container image is stored in the Docker Hub[‡] repository so that it can be easily downloaded and deployed when required.

4.3 | Experimental factors listings and selection

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

- *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 two cores. For Cases 1 and 3b, each container can use only one 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 are 4-GB DDR3 RAM and 50 GB respectively. Similar to the specified CPU configuration, containers in Cases 1 and 3b can use 2 GB and 25 GB of RAM and storage, respectively, while the configuration is fairly shared for Cases 2 and 3a.
- *OS:* The OS 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 microbenchmark, we specified a particular configuration. For Linpack, the problem size (ie, the number of equations to solve) is 15 000. In addition, the leading dimensions of the array and data alignment value are set to 15 000 and 4 KB, respectively. For Y-Cruncher, the decimal place is set to 100 m. We set the STREAM benchmark by configuring DSTREAM_ARRAY value as 60 M 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.

[‡]https://hub.docker.com/

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 to 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 (eg, 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 Cases 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 (ie, 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 90th percentile and the 10th 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:

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

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

5.1 | 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 values for the performance of Linpack in different scenarios. Other statistics are presented in

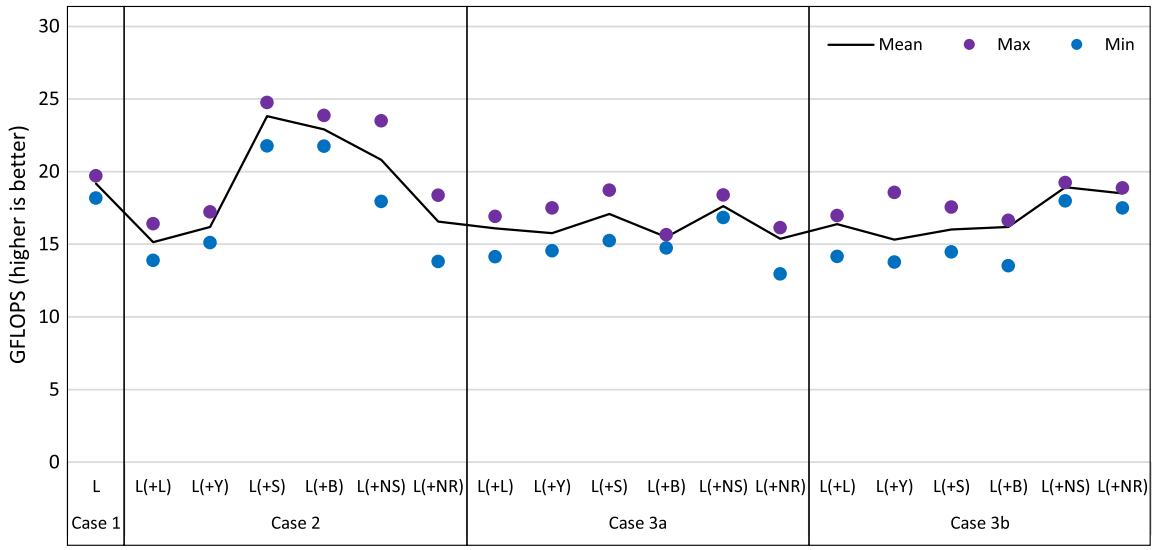

FIGURE 5 Linpack performance results

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

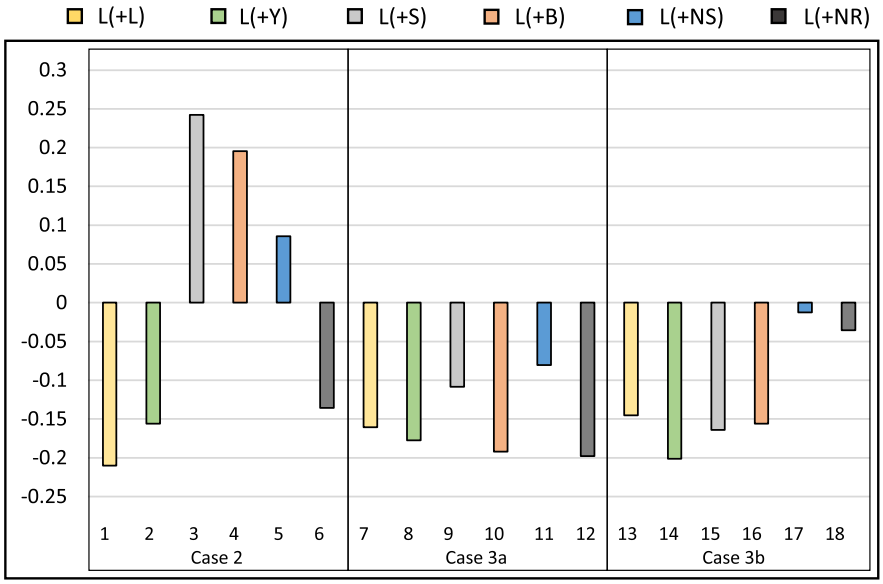

FIGURE 6 Linpack interference ratio values. Horizontal axis labels represent various cases. 1 – 6 represents L(+L), L(+Y), L(+S), L+(B), L(+NS), and L(+NR) for Case 2. Similarly, 7 – 12 and 13 – 18 are used to represent different scenarios for Cases 3a and 3b, respectively

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 the 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 CPU resource 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

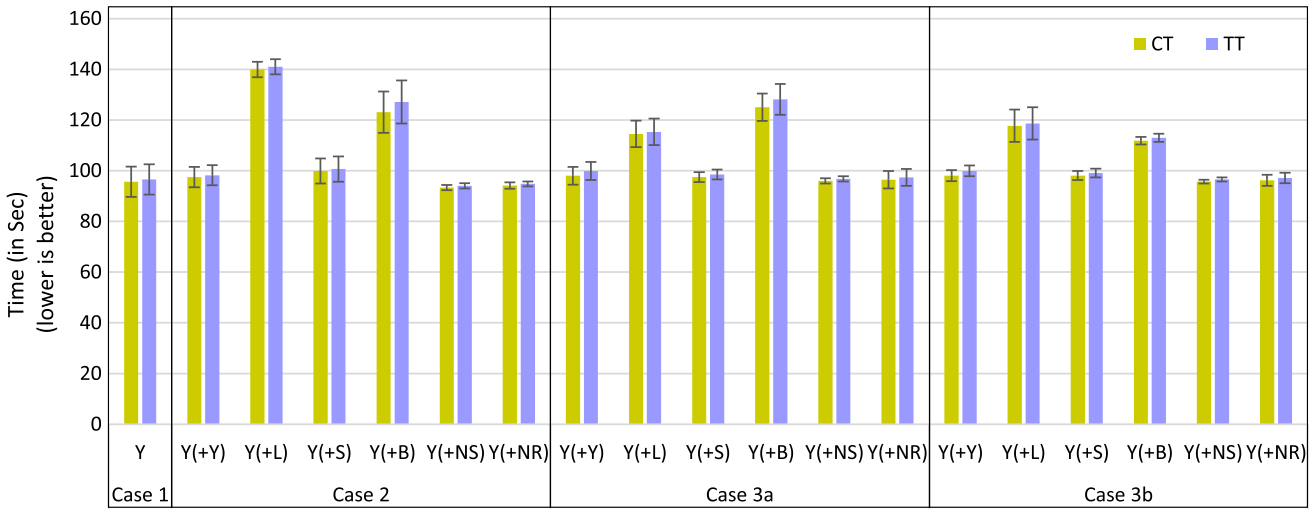

FIGURE 7 Y-Cruncher performance result for computation time (CT) and total time (TT). Black bars represent the standard deviation

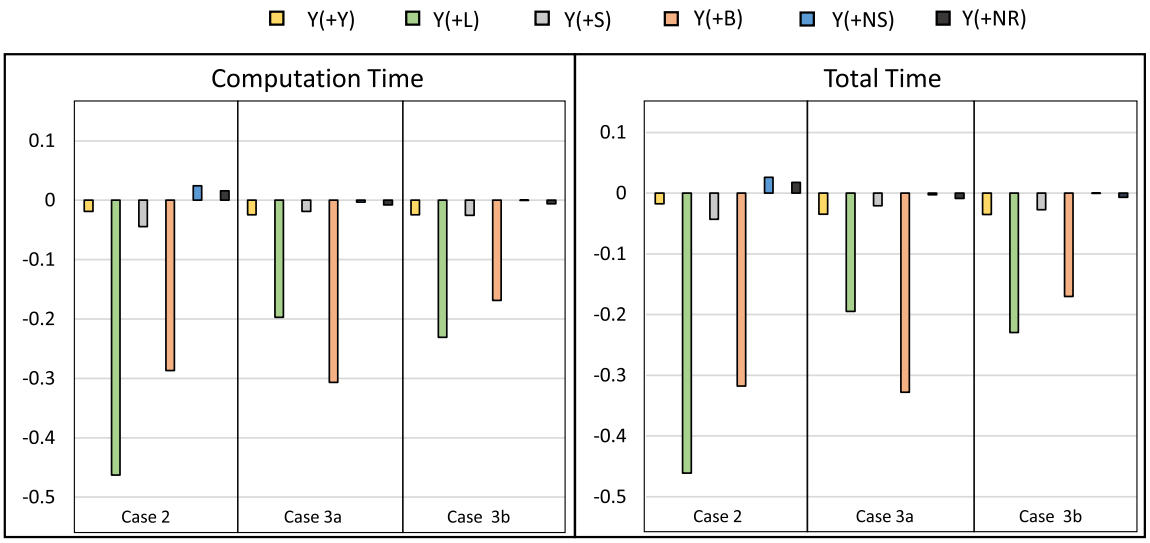

FIGURE 8 Y-Cruncher interference ratio values. Horizontal axis labels represent various cases

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 multithreading, 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).

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 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 intercontainer interference is less than intracontainer 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.

5.2 | 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

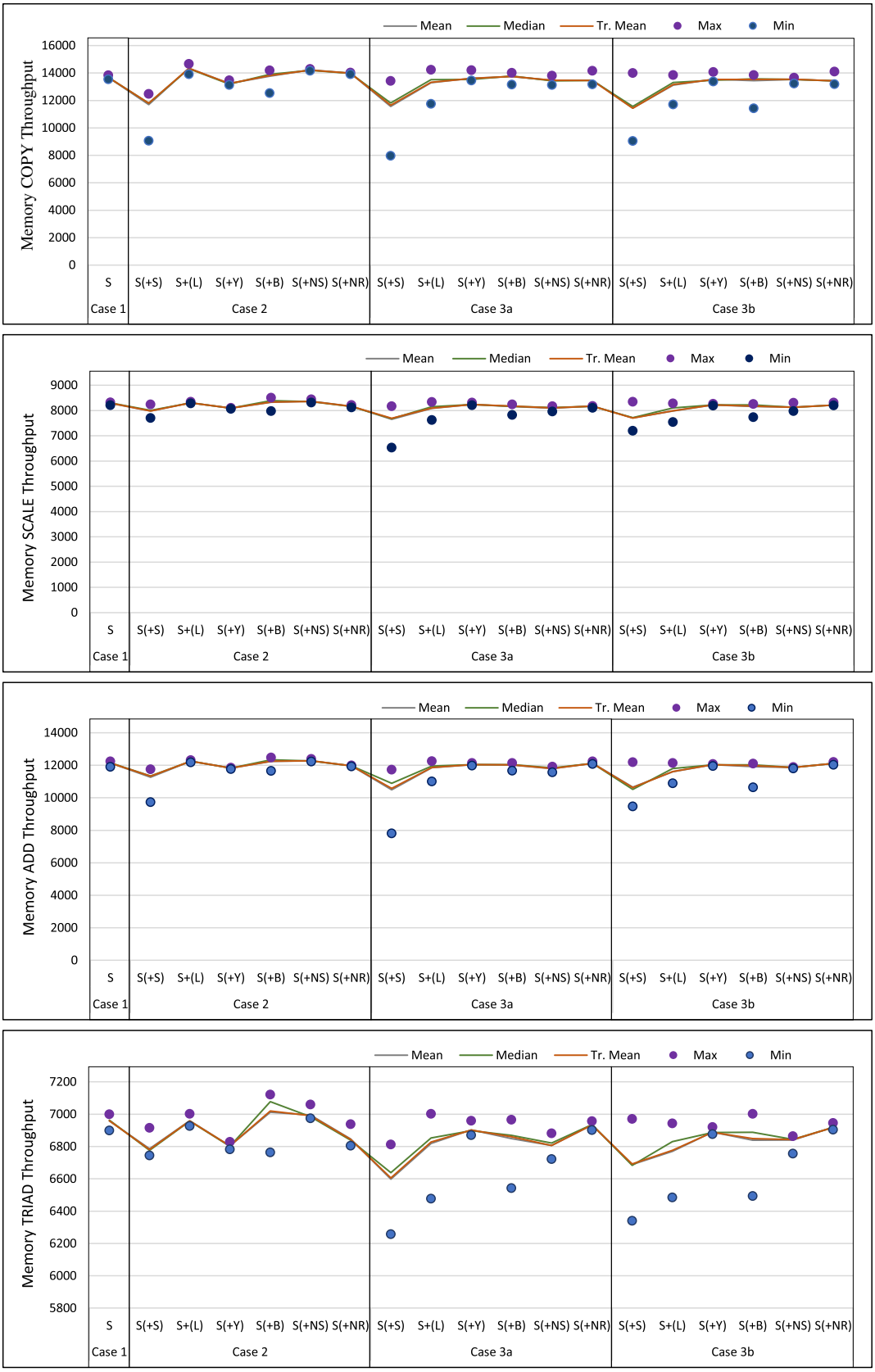

FIGURE 9 STREAM performance result

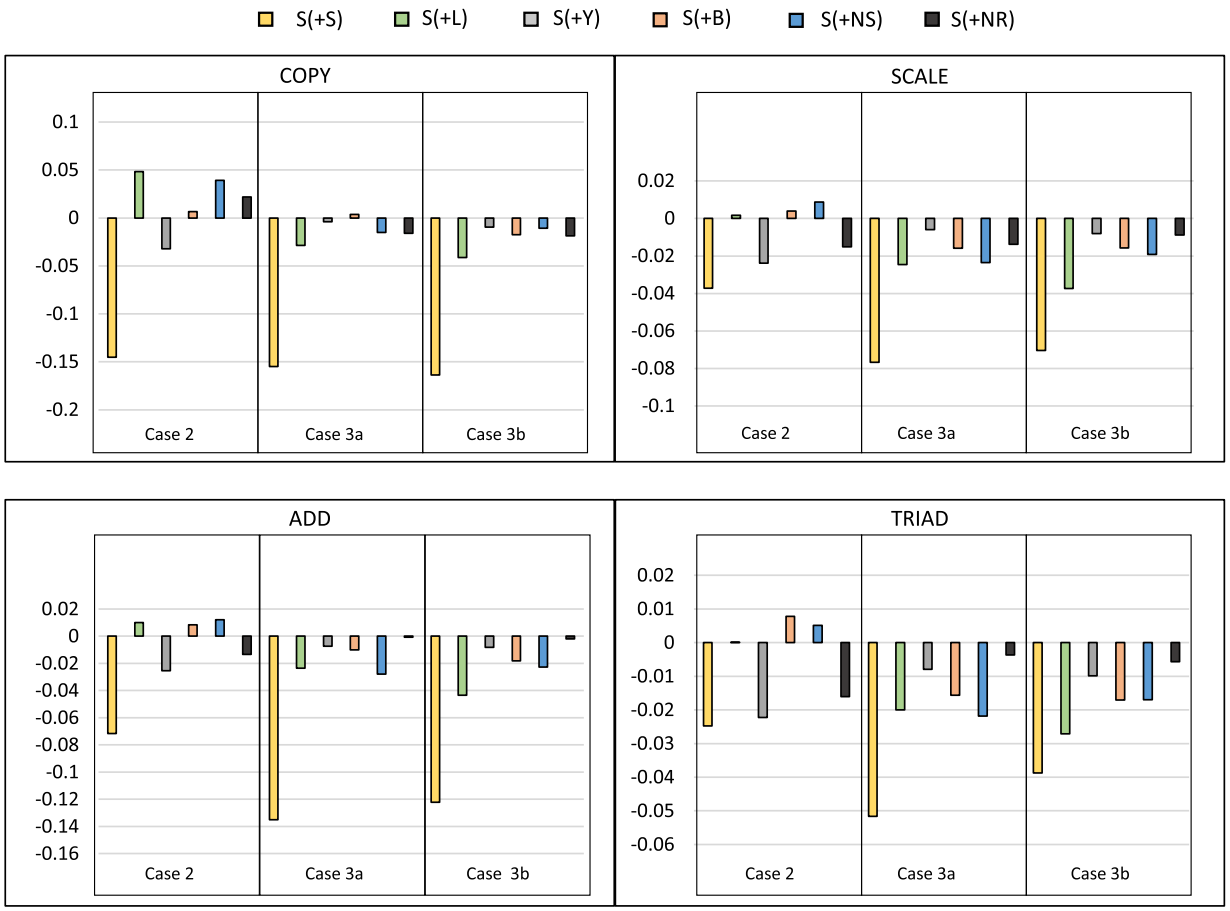

FIGURE 10 STREAM interference ratio values. Horizontal axis labels represent various cases

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 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 instances.

For the SCALE and ADD operations, there is a slight difference for various scenarios. For SCALE operations, the worst performance is S(+S) with degradation of 7.6% and 7% 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. The 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.

5.3 | 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 instances 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,

TABLE 4 Bonnie++ block input result

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case 1	B	348 948.7	346 912.5	349 092.2	366 590	328 725	8816.5	0.025
Case 2	B(+B)	150 318.6	151 020.0	150 499.3	153 749	143 634	2707.9	0.018
	B(+L)	280 510.3	279 708.0	280 276.2	300 307	264 927	9073.6	0.032
	B(+Y)	310 682.1	306 304.0	307 027.5	401 019	286 128	25 319.8	0.081
	B(+S)	293 854.5	294 489.5	294 033.4	304 766	279 722	7891.1	0.027
	B(+NS)	321 789.2	322 759.5	322 253.7	333 090	302 127	8111.1	0.025
	B(+NR)	345 039.6	345 520.0	344 307.0	379 975	323 291	12 765.5	0.037
Case 3a	B(+B)	152 934.3	153 021.5	152 967.7	155 528	149 739	1549.4	0.010
	B(+L)	268 383.2	271 084.0	268 774.8	291 607	238 110	16 004.1	0.060
	B(+Y)	292 321.3	292 458.0	292 357.1	306 960	277 038	7988.0	0.027
	B(+S)	291 840.9	293 848.5	292 551.0	303 054	267 845	8129.0	0.028
	B(+NS)	330 460.5	329 437.0	330 246.0	367 948	296 834	14 219.2	0.043
	B(+NR)	326 684.5	326 248.0	326 754.7	343 819	308 286	11 378.8	0.035
Case 3b	B(+B)	150 259.9	149 811.0	150 222.7	159 286	141 903	5348.0	0.036
	B(+L)	292 410.9	292 421.5	292 703.9	301 705	277 842	6226.1	0.021
	B(+Y)	313 616.1	296 750.5	297 370.1	639 995	279 666	77 770.2	0.248
	B(+S)	294 275.9	295 652.0	295 235.1	302 914	268 373	7984.3	0.027
	B(+NS)	263 074.5	266 438.0	264 483.2	285 272	215 519	17 366.2	0.066
	B(+NR)	288 610.8	293 233.0	289 713.6	310 925	246 446	16 459.5	0.057

TABLE 5 Bonnie++ Block Output Result

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	278 362.4	277 073.0	278 099.4	294 574	266 885	8425.81	0.030
Case 2	B(+B)	159 530.5	152 429.5	152 711.6	294 041	147 760	31 809.07	0.199
	B(+L)	283 667.1	281 342.5	281 533.3	332 479	273 264	12 189.84	0.043
	B(+Y)	281 957.0	283 641.0	281 708.0	303 000	265 396	8224.12	0.029
	B(+S)	289 314.7	289 074.0	289 009.1	305 278	278 851	8192.07	0.028
	B(+NS)	310 772.1	309 600.5	309 897.6	350 216	287 068	14 772.71	0.048
	B(+NR)	283 923.6	285 201.5	283 682.2	297 530	274 662	6981.28	0.025
Case 3a	B(+B)	148 960.7	148 394.0	148 732.7	161 455	140 569	4818.72	0.032
	B(+L)	264 280.2	263 157.0	262 325.1	290 663	243 089	11 910.16	0.045
	B(+Y)	252 390.3	252 859.5	252 346.7	293 046	232 520	4665.51	0.018
	B(+S)	270 180.9	279 268.5	269 561.9	286 893	254 610	8232.98	0.030
	B(+NS)	262 805.1	268 557.0	262 225.2	273 535	242 514	12 189.49	0.046
	B(+NR)	255 982.2	256 025.0	256 512.7	288 801	233 615	10 050.62	0.039
Case 3b	B(+B)	154 354.5	153 829.0	153 333.8	177 870	149 210	6037.06	0.039
	B(+L)	269 236.3	268 707.5	268 775.0	286 873	249 902	13 364.55	0.050
	B(+Y)	252 809.9	251 839.0	252 907.3	264 715	239 152	5977.67	0.024
	B(+S)	270 158.8	268 596.5	269 212.7	291 482	265 866	9354.95	0.035
	B(+NS)	269 616.7	270 105.5	269 896.2	283 752	250 450	9592.73	0.036
	B(+NR)	257 024.6	257 307.5	257 125.4	266 154	246 080	5303.87	0.021

and Case 3b, respectively. Table 4 also shows 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 an SD value of 77 770.2 and 25 319.8 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 instances 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%. A similar performance is witnessed for random seeks with only a small performance gain of 0.4% for Case 2 Y(+NS). For all other 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 (eg, in Case 3a B(+L), the SD (CV) is 1782.48 (19.2%)).

TABLE 6 Bonnie++ Block Rewrite Result

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	149 159.4	149 094.0	149 202.7	153 877	143 663	2841.94	0.019
Case 2	B(+B)	66 082.6	66 379.5	66 575.9	67 674	55 611	2546.87	0.039
	B(+L)	129 841.8	129 942.0	129 935.8	134 415	123 577	3087.65	0.024
	B(+Y)	125 820.3	125 511.0	125 729.9	131 053	122 215	2721.36	0.022
	B(+S)	131 675.6	131 611.0	131 756.6	137 909	123 983	4193.53	0.032
	B(+NS)	131 091.2	131 262.0	131 171.2	136 593	124 149	3128.53	0.024
	B(+NR)	127 168.5	126 673.5	127 022.2	136 821	120 149	4287.56	0.034
Case 3a	B(+B)	66 617.6	66 792.5	66 614.1	68 768	64 529	1223.94	0.018
	B(+L)	132 061.0	132 441.5	132 218.2	139 880	121 412	4558.59	0.035
	B(+Y)	126 750.6	126 659.0	126 806.7	131 635	120 855	2589.28	0.020
	B(+S)	132 399.4	131 666.5	132 319.7	139 712	126 521	3440.37	0.026
	B(+NS)	134 379.0	133 777.5	133 984.6	147 012	128 845	4553.22	0.034
	B(+NR)	111 762.8	123 709.5	116 469.7	127 234	11 566	34 380.04	0.308
Case 3b	B(+B)	67 366.5	67 807.5	67 362.8	70 339	64 459	1627.47	0.024
	B(+L)	137 653.4	137 873.5	137 813.6	142 397	130 027	3186.39	0.023
	B(+Y)	133 628.2	132 893.0	133 397.9	142 822	128 580	3435.94	0.026
	B(+S)	136 741.4	137 559.5	136 850.7	144 602	126 913	3577.91	0.026
	B(+NS)	126 108.8	127 890.5	127 169.7	131 055	102 065	6403.13	0.051
	B(+NR)	128 480.1	129 910.0	128 663.6	134 240	119 417	4381.08	0.034

TABLE 7 Bonnie++ Random Seeks Result

		Mean	Median	Tr. Mean	Max	Min	SD	CV
Case1	B	10 801.3	10 807.4	10 817.4	11 890.7	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	11 877.4	8681.6	813.41	0.082
	B(+Y)	8895.7	8947.2	8915.1	9452.9	7988.8	370.89	0.042
	B(+S)	10 274.4	10 427.5	10 330.5	11 516.4	8022.4	779.58	0.076
	B(+NS)	10 849.3	11 048.3	10 878.1	12 582.4	8596.7	1224.63	0.113
	B(+NR)	9607.7	9925.2	9626.5	10 670.6	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	11 660.3	4096.4	1782.48	0.192
	B(+Y)	8779.4	8803.5	8822.2	9550.6	7239.3	488.66	0.056
	B(+S)	10 401.3	10 583.1	10 409.4	11 882.7	8773.5	766.95	0.074
	B(+NS)	8198.8	8130.5	81 711.2	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	11 232.2	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	10 233.4	9948.3	11 260.3	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

5.4 | 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 coexecution 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 coexecution 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

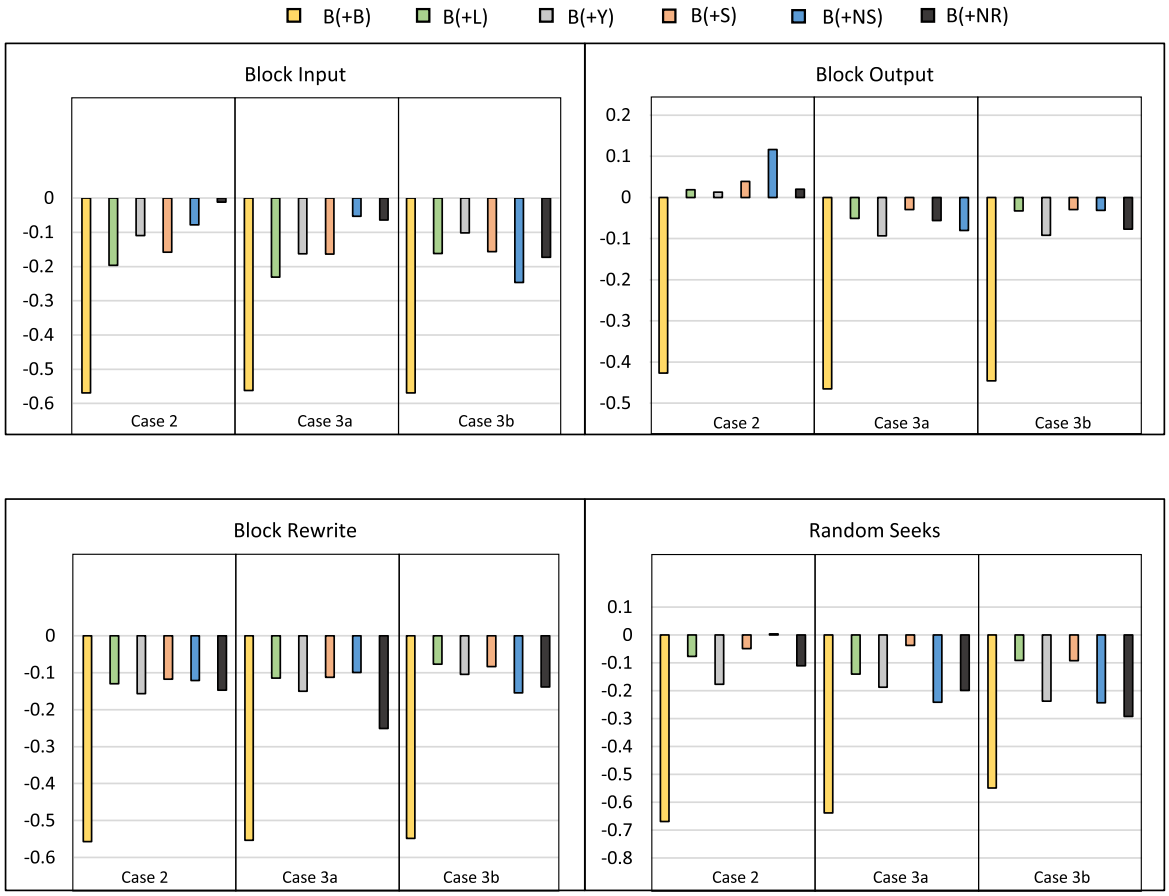

FIGURE 11 Bonnie++ interference ratio values. Horizontal axis labels represent various cases

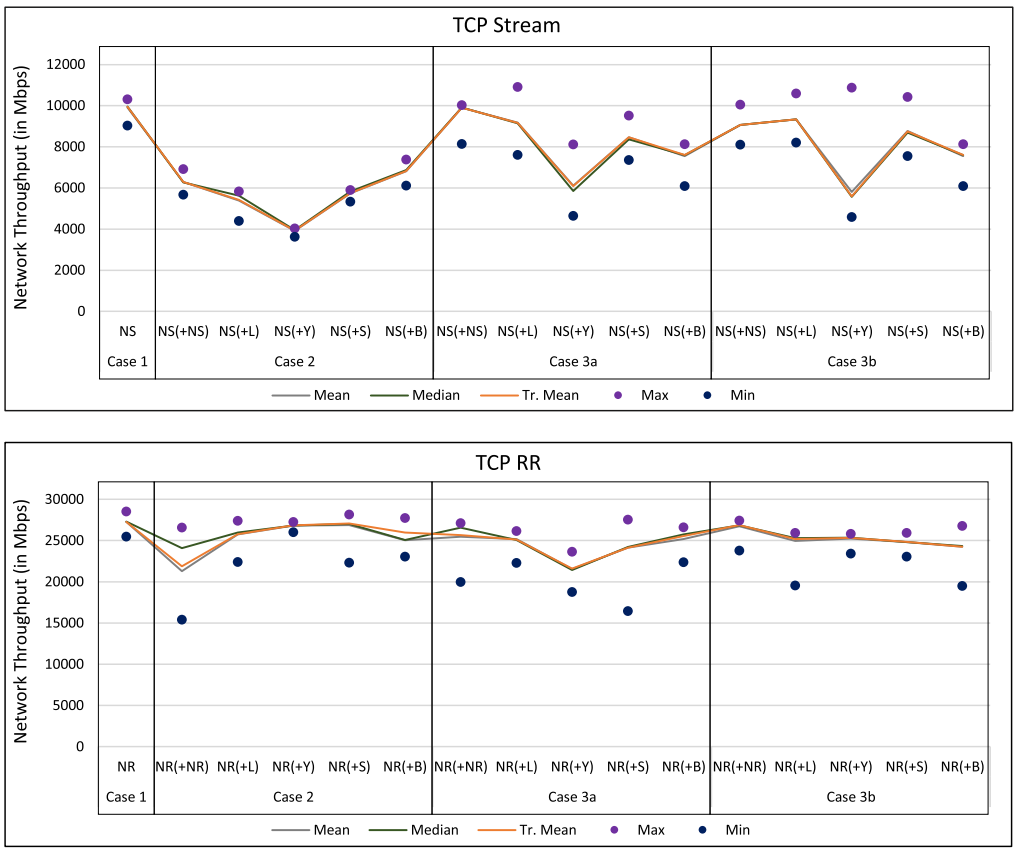

FIGURE 12 Netperf performance result

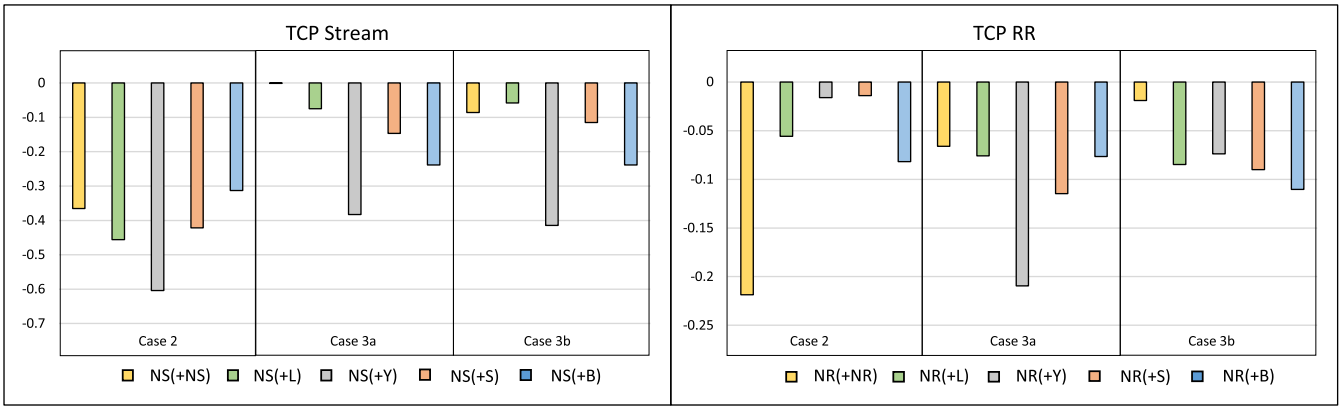

FIGURE 13 Netperf interference ratio result. Horizontal axis labels represent various cases

different containers, the performance is comparable with the baseline performance with only a small degradation of 1% and 8% for Case 3a and Case 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 same. The best performance is noticed for the execution of the two instance of TCP RR for Cases 3a and 3b with a performance loss of only 6% and 2%, respectively. The overall interference effect is presented in Figure 13.

6 | RELATED WORK

The concept of container-based virtualization is not new and has roots going back to FreeBSD Jails[24] and Solaris Zones[25] that use the Unix chroot feature to provide OS virtualization. Containers as a deployment environment are initially introduced by platform-as-a-service (PaaS) providers such as Heroku,[§] DotCloud,[26] CloudFoundry,[¶] and OpenShift[#] 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 infrastructure-as-a-service workloads (eg, HPC workloads) can also take advantage of container technology. Bernstein[28] explained 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 a DoKnowMe evaluation strategy to compare the performance of KVM and Docker and 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 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 the work of Gavriil et al.[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.

[§]https://www.heroku.com
[¶]https://www.cloudfoundry.org/
[#] https://www.openshift.com/

A study by Zhanibek and Sinnott[34] evaluates the performance of Docker and Flockport running different benchmarks and shows that Flockport outperforms Docker in almost all cases. The study by Morabito[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 et al[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 with 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 and Shane[37] advocated the use of containers for HPC environments. The work by Higgins et al[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] evaluated the performance of LXC containers using the NPB. In these experiments, containers in different configurations (ie, isolated intercontainer and multinode intercontainer) are considered for performance evaluation. The results conclude that intercontainer 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.[38-40] Bhimani et al[38] compared the performance of VMWare and Docker for different big data applications using Spark. The experimental results show that Docker achieves a speedup for map-intensive 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 with VMs.

Most of the studies presented in the literature do not consider the effect of interference in containerized environments. Sharma et al[41] considered 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 or a VM. They showed the effects of interference caused by noisy neighbor containers running competing, orthogonal, or adversarial applications. All the experiments are done on LXC containers. Ye and Ji[39] also considered the intercontainer interference for big data applications (Spark). Similar work is done by Zhang 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 (eg, K-means and 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 compare the interference impact with the microservices running in separate containers. In this paper, we have demonstrated the performance evaluation of HPC microbenchmarks 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 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 behavior 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 intercontainer and intracontainer deployments. Our results present a comprehensive study into the performance variation of containerized microservices. The main findings are given as follows:

- 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 is always higher as compared to those of microservices with different resource requirements. This effect of interference is higher for intracontainer scenarios than intercontainer scenarios. The performance in intracontainer 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-intensive operations, network-intensive operations give the best performance, while mixed CPU + memory requirement operations give 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 (eg, LXC, uDocker, Socker, and Singularity) and compare them with the performance of Docker and VM. We will also study the variation of interhost communication performance in a clustered environment (eg, Kubernetes and 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 minimizing the effect of interference.

SOURCE CODE

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

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

FINANCIAL DISCLOSURE

None reported.

ORCID

Devki Nandan Jha https://orcid.org/0000-0003-1322-2588
Rajkumar Buyya https://orcid.org/0000-0001-9754-6496

REFERENCES

1. Xing Y, Zhang X. Virtualization and cloud computing In: Zhang Y, ed. *Future Wireless Networks and Information Systems*. Berlin, Germany:Springer, Berlin, Heidelberg;2012:305-312.
2. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. KVM: the Linux virtual machine monitor. Paper presented at: 2007 Ottawa Linux Symposium; 2007; Ottawa, Canada.
3. Gulati A, Holler A, Ji M, Shanmuganathan G, Waldspurger C, Zhu X. VMware distributed resource management: design, implementation, and lessons learned. *VMware Tech J*. 2012;1(1):45-64.
4. Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. Paper presented at: Second ACM SIGOPS/EuroSys European Conference on Computer Systems; 2007; Lisbon, Portugal.
5. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and Linux containers. Paper presented at: 2015 IEEE International Symposium on Performance Analysis of Systems and Software; 2015; Philadelphia, PA.
6. Higgins J, Holmes V, Venters C. Orchestrating docker containers in the HPC environment. Paper presented at: International Conference on High Performance Computing; 2015; Frankfurt, Germany.
7. Ruiz C, Jeanvoine E, Nussbaum L. Performance evaluation of containers for HPC. Paper presented at: Euro-Par 2015: Parallel Processing Workshops; 2015; Vienna, Austria.
8. Claus P, Jamshidi P. Microservices: a systematic mapping study. Paper presented at: Sixth International Conference on Cloud Computing and Services Science; 2016; Rome, Italy.
9. Fazio M, Celesti A, Ranjan R, Liu C, Chen L, Villari M. Open issues in scheduling microservices in the cloud. *IEEE Cloud Comput*. 2016;3(5):81-88.
10. Morabito R, Kjällman J, Komu M. Hypervisors vs. lightweight virtualization: a performance comparison. Paper presented at: 2015 IEEE International Conference on Cloud Engineering; 2015; Tempe, AZ.
11. Jha DN, Garg S, Jayaraman PP, Buyya R, Li Z, Ranjan R. A holistic evaluation of docker containers for interfering microservices. Paper presented at: 2018 IEEE International Conference on Services Computing; 2018; San Francisco, CA.

12. Li Z, O'Brien L, Zhang H. CEEM: a practical methodology for cloud services evaluation. Paper presented at: IEEE Ninth World Congress on Services; 2013; Santa Clara, CA.
13. Biederman EW. Multiple instances of the global Linux namespaces. Paper presented at: 2006 Linux Symposium; 2006; Ottawa, Canada.
14. Noyes K. Docker: a shipping A 'shipping container' for linux code. *linux.com*. https://www.linux.com/news/docker-shipping-container-linux-code. Published August 1, 2013. Accessed March 24, 2019.
15. Li Z, OBrien L, Ranjan R, Zhang M. Early observations on performance of Google compute engine for scientific computing. Paper presented at: 2013 IEEE Fifth International Conference on Cloud Computing Technology and Science (CloudCom); 2013; Bristol, UK.
16. Li Z, Mitra K, Zhang M, et al. Towards understanding the runtime configuration management of do-it-yourself content delivery network applications over public clouds. *Futur Gener Comput Syst*. 2014;37:297-308.
17. Li Z, OBrien L, Cai R, Zhang H. Towards a taxonomy of performance evaluation of commercial cloud services. Paper presented at: 2012 IEEE Fifth International Conference on Cloud Computing (CLOUD); 2012; Honolulu, HI.
18. Gennady F, Shaojuan Z. Intel math kernel library benchmarks. https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite/. Published November 1, 2016. Updated May 29, 2018. Accessed March 24, 2019.
19. Yee AJ. y-cruncher - A multi-threaded pi-program. http://www.numberworld.org/y-cruncher/. Accessed March 24, 2019.
20. McCalpin JD. STREAM benchmark. http://www.cs.virginia.edu/stream/. Accessed March 24, 2019.
21. Coker R. Bonnie++. https://www.coker.com.au/bonnie++/. Accessed March 24, 2019.
22. The netperf homepage. https://hewlettpackard.github.io/netperf/. Accessed March 24, 2019.
23. Li Z, O'Brien L, Zhang H, Cai R. A factor framework for experimental design for performance evaluation of commercial cloud services. Paper presented at: 2012 IEEE Fourth International Conference on Cloud Computing Technology and Science (CloudCom); 2012; Taipei, Taiwan.
24. Poul-Henning K, Watson RNM. Jails: confining the omnipotent root. Paper presented at: Second International SANE Conference; 2000; Maastricht, The Netherlands.
25. John B, David C, Ozgur L, et al. Virtualization and namespace isolation in the solaris operating system (PSARC/2002/174); 2006.
26. Dua R, Raja AR, Kakadia D. Virtualization vs containerization to support PaaS. Paper presented at: 2014 IEEE International Conference on Cloud Engineering (IC2E); 2014; Boston, MA.
27. Hindman B, Konwinski A, Zaharia M, et al. Mesos: a platform for fine-grained resource sharing in the data center. Paper presented at: Eighth USENIX Conference on Networked Systems Design and Implementation (NSDI); 2011; Boston, MA.
28. Bernstein D. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Comput*. 2014;1(3):81-84.
29. Kolla. https://wiki.openstack.org/wiki/Kolla. Accessed March 24, 2019.
30. Bankole K, Krook D, Murakami S, Silveyra M. A practical approach to dockerizing OpenStack high availability. Paper presented at: 2014 OpenStack Paris Summit; 2014; Paris, France.
31. Xavier MG, Neves MV, Rossi FD, Ferreto TC, Lange T, De Rose CAF. 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); 2013; Belfast, UK.
32. Li Z, Kihl M, Lu Q, Andersson JA. Performance overhead comparison between hypervisor and container based virtualization. Paper presented at: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA); 2017; Taipei, Taiwan.
33. Gavriil KC, Nicolas S, Vandikas K. Bare-metal, virtual machines and containers in OpenStack. Paper presented at: 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN); 2017; Paris, France.
34. Zhanibek K, Sinnott RO. A performance comparison of container-based technologies for the cloud. *Futur Gener Comput Syst*. 2017;68:175-182.
35. Morabito R. Power consumption of virtualization technologies an empirical investigation. Paper presented at: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC); 2015; Limassol, Cyprus.
36. Cuadrado-Cordero I, Orgerie A-C, Menaud J-M. 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; 2017; Split, Croatia.
37. Jacobsen DM, Shane CR. Contain this, unleashing Docker for HPC; Paper presented at: Cray User Group 2015; 2015; Chicago, IL.
38. Bhimani J, Yang Z, Leeser M, Mi N. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. Paper presented at: 2017 IEEE High Performance Extreme Computing Conference (HPEC); 2017; Waltham, MA.
39. Ye K, Ji Y. Performance tuning modeling for big data applications in docker containers. Paper presented at: 2017 international conference on networking, architecture, and storage (NAS); 2017; Shenzhen, China.
40. Zhang Q, Liu L, Pu C, Dou Q, Wu L, Zhou W. A comparative study of containers and virtual machines in big data environment. Paper presented at: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD); 2018; San Francisco, CA.
41. Prateek S, Lucas C, Prashant S, Tay YC. Containers and virtual machines at scale: a comparative study. Paper presented at: Proceedings of the 17th International Middleware Conference (Middleware '16); 2016; Trento, Italy.

How to cite this article: Jha DN, Garg S, Jayaraman PP, et al. A study on the evaluation of HPC microservices in containerized environment. *Concurrency Computat Pract Exper*. 2021;33:e5323. https://doi.org/10.1002/cpe.5323