Future Generation Computer Systems 68 (2017) 175–182

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# A performance comparison of container-based technologies for the Cloud

Zhanibek Kozhirbayev [a,*], Richard O. Sinnott [b]

[a] Faculty of Information Technologies, L.N.Gumilyov Eurasian National University, Kazakhstan
[b] Department of Computing and Information Systems, The University of Melbourne, Australia

## HIGHLIGHTS

- The key features of the micro-service hosting technologies for the Cloud were identified.
- We perform test cases to evaluate virtualization performance of these technologies.
- There were roughly no overheads on memory utilization or CPU by the examined technologies.
- I/O and operating system interactions incurred some overheads.

## ARTICLE INFO

*Article history:*
Received 27 August 2015
Received in revised form
19 July 2016
Accepted 27 August 2016
Available online 22 September 2016

*Keywords:*
Container-based virtualization technologies
Cloud computing
Performance comparison

## ABSTRACT

Cloud computing allows to utilize servers in efficient and scalable ways through exploitation of virtualization technology. In the Infrastructure-as-a-Server (IaaS) Cloud model, many virtualized servers (instances) can be created on a single physical machine. There are many such Cloud providers that are now in widespread use offering such capabilities. However, Cloud computing has overheads and can constrain the scalability and flexibility, especially when diverse users with different needs wish to use the Cloud resources. To accommodate such communities, an alternative to Cloud computing and virtualization of whole servers that is gaining widespread adoption is micro-hosting services and container-based solutions. Container-based technologies such as Docker allow hosting of micro-services on Cloud infrastructures. These enable bundling of applications and data in a manner that allows their easy deployment and subsequent utilization. Docker is just one of the many such solutions that have been put forward. The purpose of this paper is to compare and contrast a range of existing container-based technologies for the Cloud and evaluate their pros and cons and overall performances. The OpenStack-based Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR) Research Cloud (www.nectar.org.au) was used for this purpose. We describe the design of the experiments and benchmarks that were chosen and relate these to literature review findings.



## 1. Introduction

Nowadays Cloud platforms and associated virtualization technologies are in great demand. Many software companies such as VMware (VMware), Citrix (Xen), Microsoft (Hyper-V) dominate the virtualization market with their solutions and companies which are oriented to hardware such as Intel and AMD now offer advanced processors to support virtualization. Collectively these technologies are utilized for server consolidation—typically in data centers that offer large collections of servers for external communities in a flexible manner through for example elastic scaling.

There has been much research related to virtualization performance. Some of these works concentrate on HPC facilities [1,2] whilst others focus on Cloud environments. Previous research identified that technologies which utilize hypervisor-based virtualization, face high performance overheads. In addition they suffer from I/O limitations and hence are normally avoided in HPC environments. Recently container-based virtualization and support for microhosting services has gained significant acceptance since it provides a lightweight solution that allows bundling applications and data in a simpler and more performance-oriented manner that can run on different Cloud infrastructures. This way of dealing with virtualization offers horizontally scalable, deployable systems

* Corresponding author.
*E-mail address:* zhanibekkm@gmail.com (Z. Kozhirbayev).

http://dx.doi.org/10.1016/j.future.2016.08.025
0167-739X/

without the difficulty of high-performance challenges of traditional hypervisors and the overheads of managing large scale Cloud infrastructures [3]. In this work, we undertake a review of micro-hosting services and perform a number of experiments to provide a comprehensive performance evaluation of container-based virtualization technologies for the Cloud. We focus in particular on representative systems: Docker [4–6] and Flockport (LXC) [7,8] as leading offerings.

The purpose of this work is to compare the performance of container-based virtualization technologies on the Cloud. This work focused specifically on CPU, memory as well as I/O devices capacities. In order to meet these criteria four objectives were defined:

- Critically review performance experiments of related works to measure the performance of different existing virtualization technologies on different environments;
- Identify performance-oriented case studies in order to evaluate the performances of virtualization technologies on the Cloud;
- Implement several case studies to evaluate the performances of the microhosting technologies; and
- Compare the results obtained from the performed experiments and identify the pros and cons of microhosting technologies under these experimental conditions.

This paper is organized as follows: Section 2 presents an overview of various technologies for virtualization and related work on benchmarking applications. More precisely, it describes container-based virtualization and hypervisor-based virtualization as well as representative examples of these solutions including Docker, LXC (Flockport) and CoreOs Rocket. Section 3 compares the key features of the container-based technologies that may influence to their performance. The design of the experiments executed to evaluate virtualization performance is described in Section 4 including the Cloud environment, the system architecture and benchmarking tools that were used to perform the benchmarking case studies. The implementation details of the experiments and the results of the performance comparison are presented in Section 4. The summary of the performance comparison and areas of further research are given in Section 5.

## 2. Background and related work

Virtualization of resources typically includes utilizing an additional software layer above the host operating system with an eye to handle multiple resources. Such virtual machines (VMs) can be considered as a separate execution environment. Several approaches are used for virtualization purposes [9]. One popular technique is hypervisor-based virtualization. Well-known solutions based on hypervisor-based virtualization are: KVM and VMware. In order to use this kind of technology there should be a virtual machine monitor above the underlying physical system. Each virtual machine also has support for (isolated) guest operating systems. It is quite possible that one host operating system may support many guest operating systems within this virtualization approach [9].

Container-based virtualization technology represents another approach. In this model, the hardware resources are divided by implementing many instances with (secure) isolation properties [9]. The difference between the two technologies can be seen in Fig. 1. Here, guest processes obtain abstractions immediately with container-based technologies as they operate through the virtualization layer directly at the operating system (OS) level. In hypervisor-based approaches however there is typically one virtual machine per guest OS [9]. One OS kernel is typically shared among virtual instances in container-based solutions. Therefore, there is an assumption that the security of this kind of approach

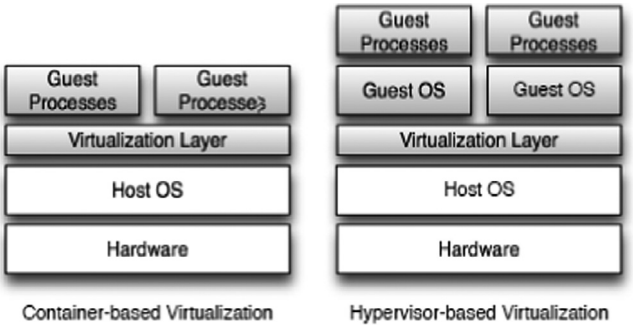


**Fig. 1.** Comparison of container-based and hypervisor-based approaches.

is weaker than with hypervisors. From a users perspective, containers operate as autonomous OSs, which appear able to run independently of hardware and software [10].

Biederman [11] argues that kernel namespaces are responsible for handling the isolation property of containers. This is considered as a Linux kernel characteristic attribute, which allows processes to obtain the necessary levels of abstraction. Despite the fact that there is no interaction between containers and outside of a namespace layer, there is isolation between the Host OS and Guest Processes and each container has its own operating system. According to Biederman [11], file systems, process identifiers, networks as well as inter-process communication are considered to be isolated over namespaces. However, there is a restriction of the resource utilization in accordance with process groups in container-based virtualization technologies. This procedure is managed by *cgroups* [12]. To be precise, cgroups are responsible for determining the priority for CPU, memory as well as I/O utilization in container-based virtualization. However certain technologies, which use containers implement their management of the resources in accordance with the consistency of cgroups.

Using such container-based solutions allows for the dynamic deployment and use of micro-services in bundled hosting environments. Micro-service patterns are not a new idea in software architecture. Nowadays they are widely recognized as an efficient solution to develop applications. Prior to the micro-services architecture, the general approach for service development was to create largely monolithic applications. From a functional point of view, this required a single environment that handled all the things. In Cloud environments, many of these issues can be overcome through scripting approaches supporting Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). However, such solutions are challenged when there is many *ad hoc* projects and communities involved with their own diverse software and data demands, e.g. where community-specific virtual machine images are not available for the Cloud. In such environments, light-weight Cloud-enabled solutions are beneficial. Micro-services are one such model.

The main concept of the micro-services architecture is "divide and conquer". Fundamentally, micro-services replace a single larger code base with multiple small-scale code foundations controlled by small, agile groups. These code foundations have a single API relationship with one another. The benefits behind this concept are that every group may work in isolation and be protected/disconnected from one another—so called exemption cycles. However, these exemption cycles may be connected in certain cases, e.g. when there is dependency between services in different groups [13].

A range of container-based micro-hosting services now exists. The most established of these are Docker, CoreOS and LXC. Docker provides a less complicated way to wrap an application into a container including the accessories it requires for execution.

This action is conducted through a targeted set of tools and integrated Application Programming Interface Guiding technologies with kernel-level structure, e.g. Linux containers, control groups as well as a copy-on-write file system. Acting as a file system for containers Docker is dependent on Advanced Multi-Layered Unification File system (AuFS). AuFS is able to explicitly superimpose single or multiple available file systems. It enables Docker to utilize images required for the container's foundation. For instance, a person may use a Ubuntu image that in turn might provide the foundation for multiple other containers. With the help of the Advanced Multi-Layered Unification File system Docker utilizes a single copy of Ubuntu. This dramatically saves storage and reduces the use of memory in line with prompt launching of containers. AuFS has one more advantage which enables it to create image versions. Every latest version is marked as diff, which is a file collation benefit that identifies the difference between two files. This allows for example for image files to be reduced in volume. Another efficiency of AuFS is that every modification made on image versions can be tracked—much like software development code versioning systems [4].

CoreOS is a comparatively recent distribution of Linux, which has been designed to provide characteristics required to operate stacks of software systems. This technology provides a reduced Linux kernel to decrease overheads to the maximum. Moreover, CoreOS supports a cluster with tools in order to ensure redundancy as well as methods of protecting systems from failure.

Recently, CoreOS introduced a new product—rocket container runtime (rkt) [14], which runs the application container specification. The main purpose of this technology is to build a specified model for a container. This approach also supports Amazon Machine Images. The rocket container runtime is an option to Docker, which includes advanced security as well as other demands necessary for production activities on servers. The rocket container runtime is aligned with the Application Container specification, which offers a novel set of formats for containers that allows them to be easily carried or moved. In Docker, every process runs via a daemon and from security point of view it does not provide as much assurance as Rocket. In order to correct this phenomenon it has been suggested that Docker should be rewritten completely.

LXC is maintained in the standard Linux kernel and the project enables instruments to manage container and OS images. Containers can be considered as lightweight OS facilities which execute together with the host OS. Containers do not imitate the hardware layer and therefore they can execute at almost native speed in the absence of other performance overheads. In their standard utilization, applications as well as web stacks are established and put in a particular form in bare-metal servers for testing purposes. For example, PHP, MySQL, Nginx and Drupal can be installed and configured to run in parallel. However, currently applications are dedicated to the machine where they have been established and cannot simply be migrated. A virtual machine can be installed and migrated and it may give some sense of being easily moved but this compromises performance.

The LXC container can give almost bare-metal throughput and the capability to simply migrate throughout systems by establishing similar stack into containers. An LXC container has improved performance and flexibility, leading to the illusion that there is a separate server. The containers can be cloned, backed up and snapshotted. LXC makes it easy to manage containers and introduces new degrees of flexibility in executing and launching apps. Flockport ensures web stacks as well as software in LXC containers can be launched on any Linux-based machines. LXC containers directly support flexibility and throughput, whereas Flockport is a tool to distribute LXC containers and simplify their utilization.

Several researchers have concentrated on reducing the difference between the virtualization technologies and non-virtualized approaches in terms of performance and optimization. The methodology and tools used are typically different in each of these researches. Moreover, the examined and compared compositions of techniques vary also. For instance, the gap between native systems and visualized versions of them and the performance differences between container-based and hypervisor-based virtualization technologies are examined and compared in recent research papers. We note that this is a fast moving field and hence some of the earlier papers are based on out of date software. Moreover, they do not perform analysis on recent visualization technologies.

Four different hypervisor-based virtualization technologies were compared by Hwang et al. in [15]. They did not discover a hypervisor with any noticeably higher performance. Accordingly, the suggestion offered by them was to use various software as well as hardware platforms in Cloud facilities in order to satisfy customer requirements.

Abdellatief et al. [16] performed a performance comparison of technologies such as VMware, Microsoft Hyper-V, and Citrix Xen in different scenarios. The methodology of evaluation they used was to apply customized SQL instances. With the help of this method, they simulated millions of products, customers and orders. The same kind of analysis was performed by Varrette et al. [17], but with different technologies and associated testbed environment. They utilized kernel-based virtual machines in place of Microsoft Hyper-V, with experiments related to high performance computing. Their experiments were focused on the consumption of power, energy efficiency as well as scalability.

Despite the fact that there was an inconsistent demonstration of virtualization overheads, [17] identified that the virtualization layer for almost every hypervisor provided significant influence on the performance of the virtualized environments, especially for high performance computing domains.

Recent publications consider the similarity or dissimilarity between hypervisors with container approaches. According to Dua et al. [18], containers are becoming popular in supporting PaaS facilities. The representatives of both types of virtualization namely KVM, Xen, and LXC were benchmarked by Estrada et al. [19]. The main methodology of their research was to measure the similarities as well as the differences of the runtime performance of each mentioned technology. The basis of their experiments and benchmarking was supporting sequence-based applications.

Felter et al. [5] also compared technologies for hypervisor and container-based system namely KVM and Docker respectively. They conducted a comprehensive analysis in terms of CPU, memory, storage as well as networking bandwidth and latencies. According to their benchmark results, the performance of the Docker container was almost the same as the "bare metal" system. Their benchmarks were based on memory transfers, floating point handling, network resources, block I/O as well as database capacities. However, the authors work did not examine methodically the influence of containers on traditional hypervisors.

Utilizing containers to deploy applications in an efficient and repeatable manner was introduced in the Heroku PaaS provider [20]. Heroku offers a container as a process with additional isolation properties instead of considering it as a virtual server. As a consequence application deployment containers provide a lightweight technology with insignificant overheads and almost the same isolation as virtual machines. It also has resource sharing properties as standard processes. Such containers were heavily utilized by Google in their infrastructure. Moreover, a standard format for images as well as management tools for application containers was offered by Docker.

The main distinction compared to previous publications and this work is that the analysis made in this work is specifically related to benchmarking the performance of open source container-based technologies and hence identifying their associated advantages and disadvantages.

## 3. Comparison of micro-hosting environments

Each container-based technology has its own features. This section presents some key characteristics of the current leading container-based technologies that may have impact on the performance. We note that even though CoreOs Rocket was identified in the previous section, the performance evaluation for Rocket was not conducted since CoreOS have not yet released an official version of their product.

### 3.1. Docker

Docker utilizes several Linux kernel features in order to run containers in an isolated way [9].

- **namespaces**: Docker employs namespaces to deploy containers. There are several types of namespaces utilized by Docker to perform the tasks of creating isolated containers [4]:
  - Docker uses *pid* as a base for containers which ensures that all processes in the container are not allowed to affect processes in other containers;
  - It uses *net* in order to manage network interfaces or more precisely, it provides isolation of the system resources regarding networking;
  - *ipc* is used to provide isolation to specific inter-process communication (IPC) resources, namely System V IPC objects and POSIX message queues. This means that each IPC namespace has its own inter-process communication resources.
  - In order to allow processes to have their own view of a filesystem and of their mount points Docker uses *mnt* namespace.
  - Isolation of kernel and version identifiers is performed through *uts*.
- **control groups**: Docker executes *cgroups* so that existing containers can share available hardware resources and it is possible that there might be a limitation in use of these resources at a given time.
- **union file system** is a file system that functions by establishing layers which are utilized to provide the building blocks for containers.
- **container format** is considered as a wrapper that integrates all of the fore-mentioned mechanisms.

### 3.2. LXC

Linux Container is a container-based virtualization technology that enables the building of lightweight Linux containers without difficulty by use of a common and flexible API and associated implementations [21]. On the other hand, Docker is an application-centric technology based on containers. They share some common features but have numerous differences. Firstly, LXC is an operating-system-level virtualization technique for executing several isolated Linux containers on a single LXC host. It does not make use of a virtual machine, but rather allows utilizing a virtual environment which has its own CPU, memory, blocking I/O, network as well as the resource control mechanism. This is offered through the *namespaces* and *cgroups* features in the Linux kernel on the LXC host. It is similar to a *chroot*, but provides much more isolation. Various virtual network types and devices are supported by LXC. Secondly Docker utilizes fixed layers to enable reuse of well-structured designs, but this can be at the cost of complexity as well as throughput. The restriction of one application per container reduces the utilization possibilities. With LXC, single and/or multiple applications may be created. Furthermore LXC allows multiple system containers to be created, which may be clones of just one sub-volume that can be run by utilizing a *btrfs* file. This feature of LXC can tackle sophisticated issues of file system levels. Thirdly, LXC offers an extensive list of facilities and privileges to design as well as execute containers. Finally LXC enables the creation of unprivileged containers that ensure non-root users can build containers. Docker does not yet support this feature.

### 3.3. CoreOS rocket

Rocket [14] is a container-based technology introduced by CoreOS, which provides an alternative to Docker. Both Rocket and Docker introduce automation of the application deployment in the form of virtual containers that can execute independently based on the server's characteristics. However, whilst Docker has developed into a sophisticated environment, which supports a diversity of requirements as well as operations, Rocket is structured to perform simple functions but in a secure manner targeted to application deployment. Rocket functions as a command-line instrument for executing application containers that are descriptions of image designs. Rocket is focused on the application container specification introduced by CoreOS as a composition of descriptions that allows for a container to be easily migrated.

As recognized by Polvi [14], Rocket may be more difficult to use compared to Docker since Docker simplifies the whole process of constructing a container through its descriptive interface. [14] argues that Rocket should stay as a command-line based environment and be less likely to change.

## 4. Evaluation methodology and benchmarking

There are many perspectives that can be used to compare technologies, especially from a performance perspective. In order to evaluate container-based technologies from the perspective of their overheads, it was necessary to understand (measure) the overheads incurred based upon non-virtualized environments. The analysis undertaken here focused upon a range of performance criteria: the performance of CPU, memory, network bandwidth and latency and storage overheads. In all of the benchmarking multiple experiments were repeated 15 times to assess the accuracy and consistency of the various results. Average timing and standard deviation was recorded.

The Cloud environment that was used for these activities was the Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR) Research Cloud (www.nectar.org.au). NeCTAR provides a Cloud environment for all researchers across Australia. It offers access to 30,000 servers across eight availability zones—typically located in the State capitals (Melbourne, Canberra, Hobart etc.). The NeCTAR project is led by the University of Melbourne and funded by the Department of Education. NeCTAR utilizes the OpenStack middleware to realize the Cloud infrastructure.

The performance studies are all executed on NeCTAR Research Cloud instances. The following instance configurations were used for performing the experiments: Model: Processor: AMD Opteron 62xx class @ 2.60 GHz; Processor ID: AuthenticAMD Family 21 Model 1 Stepping 2; Memory: 3955 MB; OS: Ubuntu 12.04 (64-bit).

The virtualization technologies and their versions are given in Table 1.

For conformity, all Docker and Flockport (LXC) containers utilized a Ubuntu 64 bit system image. Moreover, both of them were running on the host OS which itself was based on Ubuntu 12.04 64-bit.

**Table 1**
The virtualization technologies and their versions.

| Virtualization technologies | Version |
|---|---|
| Docker | 1.4.0 |
| Flockport (LXC) | 1.1.2 |

**Table 2**
pbzip2 compressor results.

| Platforms | Wall clock (s) |
|---|---|
| Native | 13.7 |
| Docker | 14.8 |
| Flockport (LXC) | 14.9 |

**Table 3**
Multi-core efficiency results from Y-cruncher.

| Platform | Multi-core efficiency |
|---|---|
| Native | 99.2% |
| Docker | 99.3% |
| Flockport | 99.4% |

The outcomes of the performed experiments are demonstrated in this section. As mentioned previously, the chosen benchmark tools evaluate CPU, memory, network bandwidth and latency, storage overhead performances.

### 4.1. CPU performance

The first scenario to evaluate the CPU performance was based on use of a compressor. Compression is a regularly utilized module of Cloud environments processing. PBZIP2 [22] is a parallel realization of the bzip2 block-sorting data compression utility. It utilizes $p$ number of threads and can reach an almost linear acceleration. pbzip2-1.1.12 version was used in order to compress a file. An input file (enwik8) [23], which is 100 MB dump data and frequently applied for compression testing purposes. In order to concentrate on compression 900 kB BWT Block Size and 900 kB File Block Size was used.

The performance of the pbzip2 compressor is presented in Table 2. From the perspective of CPU evaluation, Docker performs slightly better than LXC. In case of Flockport, average elapsed time is 14.9 s and standard deviation is $\pm 0.03$ s, whilst average time by Docker is 14.8 s and standard deviation is $\pm 0.01$ s. However, the size of input file should be taken into consideration. If the size increases, the difference of results can be significant.

The second CPU benchmarking tool is Y-cruncher [24], which is used to compute Pi. It is typically executed as a stress-testing tool for CPUs and frequently used as a test for multi-threaded tools running in multi-core systems. Besides calculating the value of Pi, Y-cruncher can also compute a range of other constants. Y-cruncher performs various outcomes such as multi-core efficiency, computation time, and total execution time. The total time is applied to confirm the outputs, and it includes the total computation time added to the time requested to exploit and perform the outcome.

The performance results of the Y-cruncher benchmarking tool can be seen in Fig. 2. In terms of computation time, Docker performs similarly to the native (non-virtualized) system, whereas Flockport takes on average about 2 s longer. The multi-core efficiency results of these systems are presented in Table 3. This assessment describes how the CPU is effectively utilized in calculation of Pi. This pattern also shows that Docker shows marginally better performance than Flockport.

The next benchmarking tool explored was the standard HPC benchmarking tool: Linpack. There are two options for this tool. The first one is an optimized version by Intel [25], whereas the

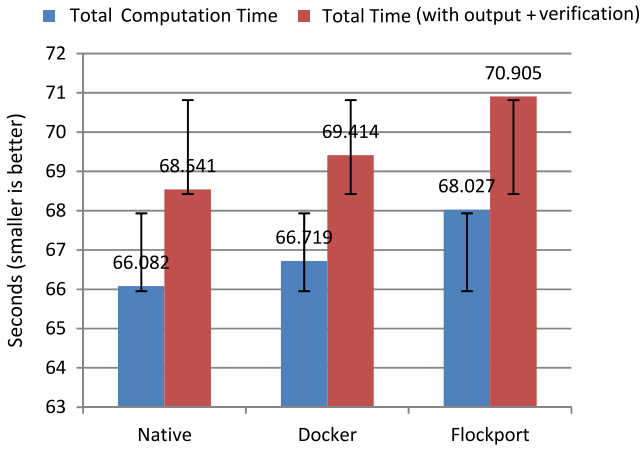


**Fig. 2.** Performance results of the Y-cruncher benchmark.

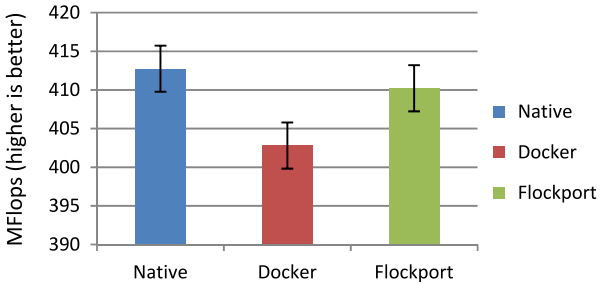


**Fig. 3.** Linpack results (where $N = 1000$).

second one [26] allows to operate on all machines and not only Intel machines. Linpack finds the solution for a system of linear correspondences utilizing an approach that performs 'lower upper' decomposition of numerical analysis with partial pivoting. A great number of computational operations consist of multiplying a process of a scalar with a vector in double-precision floating-point format as well as processes for adding the outcomes to different vectors. The benchmarking tool is built on the basis of linear algebra functions which are modified for the chosen computer architecture. The main Linpack function utilizes a random matrix $M$ of size $N$ and a vector $V$ which is determined as $M * X = V$. The Linpack benchmark tool performs two steps as follows: 'Lower Upper' decomposition of $M$, and subsequently the 'Lower Upper' decomposition applied to solve the linear problem $M * X = V$. The outcomes of Linpack are typically provided in MegaFLOPS—floating-point operations per second. Executing this tool and increasing the value of N, it can be seen from experiment that the behavior of the CPU usage changes and various phases can be identified: rising zone, where no challenge occurs in local memory or processor; flat zone, where challenges occur in processor functionality, and the decaying zone, where challenges occur in the local cache memory.

The results of Linpack run in the micro-hosting environments can be seen in Fig. 3. These outcomes are obtained by applying a specific scenario with $N = 1000$. As seen, even though the gap between them is relatively small, the performance of Flockport in executing the Linpack benchmark is slightly better than Docker.

The final tool used to evaluate the performance of the CPU was Geekbench [27]. This tool is useful to test performance of the Floating Point Unit as well as the throughput of memory systems. An upper bound for the above-mentioned throughput features was assessed by this application. In comparison with Y-cruncher, Geekbench supports the evaluation of single as well as multi-core architectures. It can execute various workloads generating multiple indexes such as Integer Performance, Floating

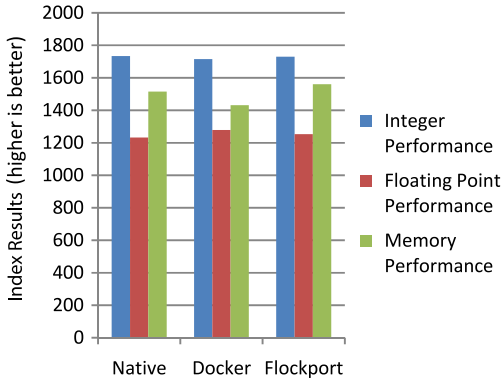


**Fig. 4.** Geekbench results for single-core testing.

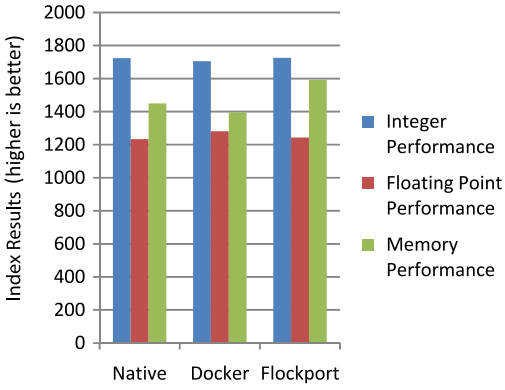


**Fig. 5.** Geekbench results for multi-core testing.

Point Performance, and Memory Performance. Moreover, the index of the complete system can be generated.

As can be seen from Figs. 4 and 5, there is no significant difference in either single-core or multi-core testing of results. However, regarding memory performance, Flockport produces approximately 100 points and 200 points more than Docker performance in single-core and multi-core testing respectively.

### 4.2. Disk I/O performance

A key aspect of performance is the evaluation of disk I/O performance, specifically volumes given as a non-ephemeral storage, were attached to instances. These volumes were created in the same availability zone as the associated instances. The first applied benchmarking tool used to evaluate the micro-hosting environments was Bonnie++ [28]. Bonnie++ is an open-source application that describes disk throughput. Bonnie++ has to be configured for different cases so a common test file was used. A 4 Gb dataset was used for this purpose. Fig. 6 presents the Bonnie++ outcomes for Block Output as well as Block Input based upon sequential write and read respectively. All three systems produce almost same results in terms of sequential write. However Flockport shows a slightly better performance compared to Docker regarding sequential reading of files.

Through the Bonnie++ benchmarking application, the speed can be assessed when reading, writing and flushing processes take place in a file. Table 4 shows the Random Write Speed as well as Random Seeks for each system.

There is relative alignment between random write speed outcomes at which a file is read and then written and flushed to the disk as depicted in Fig. 7. However, the arrangement of the systems is different in their random seek evaluation which shows the number of blocks which Bonnie++ can seek to per second. In

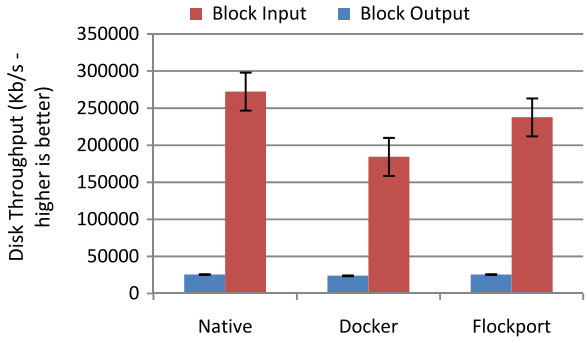


**Fig. 6.** Bonnie++ results for sequential write and sequential read.

**Table 4**
Random write speed and random seeks.

| Platform | Random write speed (kb/s) | | Random seeks | |
|---|---|---|---|---|
| Native | 24829 | % | 3741 | % |
| Docker | 22494 | −9.4% | 389.2 | −89.6% |
| Flockport | 24524 | −1.2% | 3961 | +5.9 |

this case, Flockport has 100% better results than Docker and is almost 6% better than the native platform.

To further evaluate the disk I/O throughput, the Sysbench benchmark tool [29] was utilized. This tool involves many modules as the basis of design. It also can be used as a cross-platform as well as multi-threaded measurement software for measuring operating system characteristics. The main concept of Sysbench is to gain a representation at a fast speed about system throughput without deploying database systems. Actual versions of the suite allow evaluating the system characteristics such as file I/O throughput, memory allocation and transferring speed scheduler throughput. In performing this scenario, the tool was used to assess the platforms performance to read as well as write from specified files. In order to evaluate file IO performance, a test file should be created that should be bigger in size than the available RAM. The size of the test file used here was 35 GB—specifically 128 files of 270 Mb each.

The results of this measurement are given in Table 5. Total time taken by event execution on Docker and Flockport is 27.488 s with standard deviation of ±0.0003 s and 27.491 s with standard deviation of ±0.0001 s respectively. There is no significant difference.

The results achieved by Sysbench reflect those obtained through Bonnie++. Native as well as Flockport showed approximately the same outcome without any discernible difference. However, Docker performed better than both platforms in some runs.

For the purpose of thoroughness, it should be noted that a default file format was utilized for disk images for Native as well as Flockport, whereas Docker applies the advanced multi layered unification file system that ensures layering as well as image versioning. As such, other tests should be conducted in this experiment.

### 4.3. Memory performance

The evaluation of Memory I/O performance is presented in this subsection. The benchmark tool used to test the micro-hosting environments was the STREAM software [30]. STREAM assesses memory throughput utilizing straightforward vector kernel procedures. The outcomes of four procedures namely Copy, Scale, Add as well as Triad are generated by this tool. These procedures and how they are calculated are shown in Table 6.

According to the STREAM software, there is a hard relation between the evaluated throughput and the size of the CPU cache.

**Table 5**
Sysbench results.

| Platform | Read (Gb) | Written (Mb) | Total transferred (Gb) | Throughput (Mb/s) | Elapsed time (s) |
|---|---|---|---|---|---|
| Native | 1.22 | 834.17 | 2.04 | 20.85 | 27.568 |
| Docker | 1.21 | 829.38 | 2.02 | 20.73 | 27.488 |
| Flockport | 1.17 | 796.88 | 1.95 | 19.92 | 27.491 |

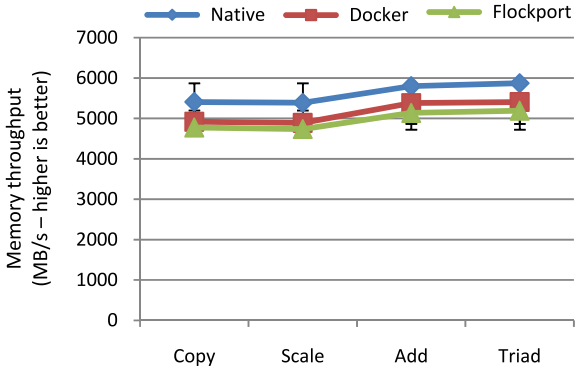


**Fig. 7.** STREAM results.

**Table 6**
Stream procedures.

| Procedure | Kernel |
|---|---|
| Copy | $x[i] = y[i]$ |
| Scale | $x[i] = q * y[i]$ |
| Add | $x[i] = y[i] + z[i]$ |
| Triad | $x[i] = y[i] + q * z[i]$ |

Moreover, there is rule that every stream array has to be at least four times the accessible cache memory size. Therefore, the size of the stream array in the software must be established correctly. As seen in Fig. 7, the difference between the results is not significantly high, but Docker produces slightly better results than Flockport and is nearly the same as the native platform.

### 4.4. Network I/O performance

This subsection describes the performance of container-based environments with regards to their Network I/O. Any containers running on the same host, more precisely on the same host bridge, can contact each other through IP. Network address translation (NAT) networking principles was used for all container traffic. This makes possible for all containers to contact outside world including other Docker containers on different hosts, but it does not allow the outside network to communicate with the containers. This regulation might be avoided by mapping container ports to ports on the host network interface. Linux containers have the same networking. The test cases were conducted on two Docker containers located on two different hosts. The same condition was applied to Flockport containers also. In both cases, one acted as a server and the other as a client. For native conditions, two identical NeCTAR Research Cloud instances were used.

The Netperf benchmark tool [31] was used to measure Network I/O. This tool has many predetermined subtests to evaluate network throughput between a server and clients. It enables execution of data transfers operating in a single direction either with TCP or UDP protocol. Time spent to establish connections between the netperf client and the netserver is not comprised in these evaluations. The assessment outcomes present the throughput of arriving packets is shown in Table 7.

Table 7 shows the outputs for both Docker and Flockport and for both TCP_STREAM and UDP_STREAM. For the TCP_STREAM test case, Docker performed almost 200 Mbps better than Flockport, whereas for the UDP_STREAM test case, Docker again offers approximately 150 Mbps better performance. Moreover, as mentioned, the Netperf benchmark tool has assessment cases including request and response, which measure the amount of TCP as well as UDP transactions. The following establishment connection occurs during these cases: the netperf client dispatches requests to the netserver and the netperf server dispatches a response to the netperf client.

**Table 7**
Netperf TCP_STREAM and UDP_STREAM results.

| Platform | TCP_STREAM (Mbps) | UDP_STREAM (Mbps) |
|---|---|---|
| Docker | 1308.38 | 721.32 |
| Flockport | 1080.48 | 586.16 |
| Native | 1455.45 | 780.24 |

As depicted in Table 8, Flockport again produces the worst results for both TCP_RR as well as UDP_RR test cases.

In order to get a thorough understanding of Network I/O performance, a second tool was applied to test Network throughput: the Iperf suite [32]. Iperf offers a complete suite for evaluating connection performances using either TCP or UDP protocols. Table 9 shows the results of Iperf tests for both TCP and UDP traffic.

It can be seen from Table 9 that Docker is slower when compared to the previous test case on I/O performance. The possible reason is the TCP window size and the UDP buffer size. That is, the quantity of data that can be buffered in the time window of a given connection without verification differs. The sizes of data can be between the range of 2 and 65,535 bytes, but they were small in *Iperf* by default (60.0 kB) and were not tuned.

## 5. Conclusions and future work

Container-based technologies are challenging hypervisor-based approaches as the basis for Clouds. Modern container-based approaches are considered to be lightweight. In this paper, a thorough performance assessment of leading micro-hosting virtualization approaches was presented with specific focus on Docker and Flockport and their comparison with native platforms. As noted, CoreOS was not considered due to the unavailability of the systems at the time of writing.

Regarding the results of performed experiments, many common patterns can be seen. As shown there were roughly no overheads on memory utilization or CPU by either Docker or Flockport, whilst I/O and operating system interactions incurred some overheads. The overheads in these cases appear by way of additional cycles for every input–output operation. Therefore, applications with increased input–output operations have more disadvantages compared to applications with lower input–output demands. As a consequence, input–output latency is exacerbated by these overheads. The CPU cycles required for utility tasks can also cause performance degradation.

Docker includes several other capabilities such as Network Address Translation, which helps to reduce some of the difficulties of Docker container utilization. However, these capabilities

**Table 8**
Netperf TCP_RR and UDP_RR results.

| Platform | TCP_RR (Transfer rate per second) | UDP_RR (Transfer rate per second) |
|---|---|---|
| Docker | 44363.03 | 45093.28 |
| Flockport | 39321.02 | 40625.07 |
| Native | 48451.11 | 49221.17 |

**Table 9**
Iperf results.

| Platform | | TCP | UDP |
|---|---|---|---|
| Docker | Interval | 0.0–10.0 s | 0.0–10.0 s |
| | Transfer | 966 MB | 11.9 MB |
| | Bandwidth | 810 Mb/s | 10.0 Mb/s |
| Flockport | Interval | 0.0–10.0 s | 0.0–10.0 s |
| | Transfer | 1.34 GB | 11.9 MB |
| | Bandwidth | 1.15 Gb/s | 10.0 Mb/s |
| Native | Interval | 0.0–10.0 s | 0.0–10.0 s |
| | Transfer | 1.64 GB | 1.19 MB |
| | Bandwidth | 1.41 Gb/s | 1.00 Mb/s |

directly impact input on throughput quality. As a result, Docker containers utilizing no extra features can be no quicker compared to Flockport in some cases. Software that is either file system or disk intensive must use the advanced multi layered unification file system by applying volumes. The impact of Network Address Translation may be removed by utilizing nethost. However, this negates the advantages of network namespaces. Eventually, the design of one IP address for each Docker container as suggested by the Kubernetes [33] may support quality assurance as well as throughput.

The generated results here may provide some direction of how the architecture of Cloud environments should be designed. Thus current considerations are that IaaS is more to utilization of virtual machines and PaaS developed to utilize containers. If IaaS is designed to use containers instead, they can provide better throughput as well as simpler deployment opportunities. Moreover, containers can reduce the difference between IaaS and "bare metal" systems because they support the management and give almost the same performance as native systems. Another question here is launching containers within virtual machines, thus these can conflict with the throughput overheads of virtual machines while providing no advantages compared to launching containers immediately on native hosts. Such pragmatic considerations should be factored in to the choices of the technologies used to build and manage IaaS and PaaS systems for performance demanding communities. Multi-tenancy is a further very important problem in Clouds, and containers or micro-service applications typically consist of multiple services running in separate containers sharing the same resources. The implications and performance impact on shared resources will be conducted in future work.

## Acknowledgment

The authors would like to thank the NeCTAR Research Cloud (www.nectar.org.au) for the resources used to perform these investigations.

## References

[1] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. Shin, Performance Evaluation of Virtualization Technologies for Server Consolidation, Enterprise Systems and Software Laboratory HP Laboratories Palo Alto HPL-2007-59, 2007.
[2] N. Regola, J. Ducom, Recommendations for virtualization technologies in high performance computing, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CloudCom, 30 2010-dec. 3 2010, pp. 409–416.
[3] N. Slater, Using Containers to Build a Microservices Architecture, viewed 1 April 2015, URL https://medium.com/aws-activate-startup-blog/using-containers-to-build-a-microservices-architecture.
[4] Docker – Build, Ship, and Run Any App, Anywhere, viewed 1 April 2015, URL http://www.docker.com.
[5] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An Updated Performance Comparison of Virtual Machines and Linux Containers, viewed 1 April 2015, URL http://www.research.ibm.com/.
[6] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment, Linux J. 2014 (239) (2014).
[7] Flockport, viewed 1 April 2015, URL http://www.flockport.com.
[8] P. Rubens, Docker Not the Only Container Option in 2015, IT Business Edge, 2015.
[9] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, C. De Rose, Performance evaluation of container-based virtualization for high performance computing environments, in: 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, (PDP), IEEE, 2013.
[10] S. Soltesz, H. Potzl, M. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, SIGOPS Oper. Syst. Rev. 41 (3) (2007) 275–287.
[11] E.W. Biederman, Multiple instances of the global linux namespaces, in: Proceedings of the Linux, 2006.
[12] P. Menage, Control groups definition, implementation details, examples and api, viewed 1 April 2015, URL http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.
[13] L. Marsden, The Microservice Revolution: Containerized Applications, Data and All, viewed 1 April 2015, URL http://www.infoq.com/articles/microservices-revolution.
[14] A. Polvi, CoreOS is building a container runtime, Rocket, viewed 1 April 2015, URL http://coreos.com/blog/rocket.
[15] J. Hwang, S. Zeng, T. Wood, A component-based performance comparison of four hypervisors, in: 2013 IFIP/IEEE International Symposium on Integrated Network Management, (IM 2013), IEEE, 2013.
[16] E. Abdellatief, N. Abdelbaki, Performance evaluation and comparison of the top market virtualization hypervisors, in: 2013 8th International Conference on Computer Engineering & Systems, (ICCES), IEEE, 2013.
[17] S. Varrette, M. Guzek, V. Plugaru, V. Besseron, P. Bouvry, HPC performance and energy-efficiency of Xen, KVM and VMware hypervisors, in: 25th International Symposium on Computer Architecture and High Performance Computing, (SBAC-PAD), IEEE, 2013.
[18] R. Dua, A.R. Raja, D. Kakadia, Virtualization vs containerization to support PaaS, in: 2014 IEEE International Conference on Cloud Engineering, (IC2E), IEEE, 2014.
[19] Z. Estrada, Z. Stephens, C. Pham, Z. Kalbarczyk, R. Iyer, A performance evaluation of sequence alignment software in virtualized environments, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (CCGrid), IEEE, 2014.
[20] Heroku, viewed 1 April 2015, URL https://heroku.com.
[21] Linux Containers, viewed 1 April 2015, URL http://linuxcontainers.org.
[22] PBZIP2, viewed 1 April 2015, URL http://www.compression.ca/pbzip2/.
[23] enwik8, viewed 1 April 2015, URL http://mattmahoney.net/dc/textdata.
[24] Y-cruncher – A Multi-Threaded Pi-Program, viewed 1 April 2015, URL http://www.numberworld.org/Y-cruncher.
[25] Intel® Math Kernel Library – LINPACK Download, viewed 1 April 2015, URL https://software.intel.com/en-us/articles/intel-math-kernel-librarylinpack-download.
[26] LINPACK_BENCH – The LINPACK Benchmark, viewed 1 April 2015, URL http://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html.
[27] Geekbench 3 – Cross-Platform Processor Benchmark, viewed 1 April 2015, URL http://www.primatelabs.com/geekbench.
[28] Bonnie++, viewed 1 April 2015, URL http://www.coker.com.au/bonnie++.
[29] Sysbench in Launchpad - SysBench: a system performance benchmark, viewed 1 April 2015, URL https://launchpad.net/sysbench.
[30] J. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, a continually updated technical report (1991-2007), 2015, viewed 1 April URL http://www.cs.virginia.edu/stream.
[31] The Netperf Homepage, viewed 1 April 2015, URL http://www.netperf.org.
[32] Iperf, viewed 1 April 2015, URL http://www.iperf.fr.
[33] Kubernetes, viewed 1 April 2015, URL http://kubernetes.io.