A Performance Study of Containers in Cloud Environment

Bowen Ruan, Hang Huang(✉), Song Wu(✉), and Hai Jin

Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology,
Huazhong University of Science and Technology, Wuhan 430074, China
{huanghang,wusong}@hust.edu.cn

Abstract. Container technology has gained great popularity since containers could provide near-native performance in cloud environment. According to different design purposes and underlying implementations, containers could be classified into application containers (e.g., Docker) and system containers (e.g., LXC). The diversity of containers may lead to a confusing choice about which kind of container is suitable for different usage scenarios. Meanwhile, the architectures of public container services are quite controversial because cloud platforms tend to run containers in virtual machines. From the perspective of performance, an extra virtual machine layer between the bare metal and containers probably brings in unnecessary performance overhead. In this paper, we carry out a performance study to explore the appropriate way to use containers from different perspectives. We first conduct a series of experiments to measure performance differences between application containers and system containers, then evaluate the overhead of extra virtual machine layer between the bare metal and containers, and finally inspect the service quality of ECS (*Amazon EC2 Container Service*) and GKE (*Google Container Engine*). Our results show that system containers are more suitable to sustain I/O-bound workload than application containers, because application containers will suffer high I/O latency due to layered filesystem. Running containers in virtual machine would result in severe disk I/O performance degradation up to 42.7 % and network latency up to 233 %. We also find out ECS offers better performance than GKE, and cloud platforms could acquire better performance by running containers directly on the bare metal.

Keywords: Container technology · Cloud platform · Performance comparison · Service quality · Virtualization overhead

1 Introduction

With lightweight design and near-native performance, container technology is emerging as a promising virtualization solution for developers to deploy applications, and has gained great popularity in the industry. Container technology

G. Wang et al. (Eds.): APSCC 2016, LNCS 10065, pp. 343–356, 2016.
DOI: 10.1007/978-3-319-49178-3_27

is also called operating-system-level virtualization, which allows multiple isolated user-space instances sharing the same operating system kernel and applies CGroups to take control of resources in the host. So far, there have been a number of container products released to the market, which include LXC (Linux Container), Docker, rkt (Rocket), and OpenVZ etc. Docker is the most prevalent one among them and being widely used in startup companies like Uber and Groupon. Moreover, major cloud platforms include Amazon Web Service [3] and Google Compute Engine [7] are also beginning to provide public container services for developers to deploy applications in the cloud. Undoubtedly, the emergence of container technology has virtually changed the trend of cloud computing market.

According to different design purpose and underlying implementation, we can classify container products into application containers and system containers. Application containers (e.g., Docker and rkt) are designed to encapsulate a single task into a standard image to effectively distribute applications. To be more specific, application containers simplify a container as much as possible to a single process to run micro service. However, system containers (e.g., LXC and OpenVZ) are designed to provide fully functional operating system with the most frequently-used services. In a sense, system containers are like a virtual machine but with more lightweight design. Besides, another significant difference between application containers and system containers is filesystem. Application containers introduce a layered stack of filesystems [9], which allows different containers reusing these layers to diminish disk usage and simplify application deployment. But system containers originally support all sorts of filesystems and are not limited to one filesystem. In default, system containers directly bind the mount to the host. Because the diversity of containers may lead to a potential misuse in the cloud environment, the differences between application containers and system containers should be more clearly clarified.

At present, public container services, such as ECS (*Amazon EC2 Container Service*) and GKE (*Google Container Engine*), have a controversial issue that they tend to run containers in the virtual machines to acquire technical support from existing management tools [4]. It is obviously to understand that an extra virtual machine layer between the bare metal and containers probably brings in unnecessary performance overhead. In principle, the essence of public container service is to provide a generic running environment for developers, no matter what underlying infrastructure it is. Hence, it is worth a comprehensive inspection for cloud platforms to evaluate the service quality. Then we are able to explore the most appropriate architecture to provide container services.

In this paper, we make the following contributions:

– We conduct a series of experiments to measure performance differences between application containers (Docker) and system containers (LXC). We find out Docker, compare to LXC, will suffer higher I/O latency due to AUFS's implementation. Besides, Docker's network latency is slightly higher than LXC because of port mapping.

- We evaluate the impact of adding an extra virtual machine layer between the bare metal and containers. By comparing the performance gap between Docker and Docker-Machine, we reveal that running containers in virtual machine will result in severe performance degradation in all aspects.
- We conduct an inspection of service quality of ECS and GKE. Our results show that ECS offers better performance than GKE, and cloud platforms could acquire better performance by running containers directly on the bare metal.

The rest of the paper is organized as follows. Section 2 provides necessary backgrounds for container technology. Section 3 describes experimental methodology, and we conduct the evaluation and analyze the experiment results in Sect. 4. We review related works in Sect. 5, and finally, Sect. 6 concludes the paper.

2 Background and Motivation

2.1 Container Background

Container technology is experiencing a rapidly development with the support from industry and being widely used in large scale production environment. Two outstanding features, speedy launching time and tiny memory footprint, make containers launch an application in less than a second and consume a very small amount of resources [2]. Relative to virtual machines, using containers not only improves the performance of applications, but it also allows the host to sustain multiple times more applications simultaneously.

Technically, we can classify containers as application containers and system containers. Application containers only contain a single process, and stop the container after this process finished. However, system containers contain a complete runtime environment, and run services like *init*, *sshd*, and *syslog* in the background. The idea behind application containers is to reduce a container as much as possible to a single process to provide micro service. Thus, an application is able to be deconstructed into many small parts, and every part will be executed in a container separately. On the contrary, the idea behind system containers is to provide fully functional operating system in a container. They are more like a lightweight virtual machine and mainly used for providing underlying infrastructure. To sum up, Table 1 demonstrates the comparisons between application containers and system containers.

One major feature of application containers is layered filesystem, which allows different containers reusing image layers to diminish disk usage and simplify application distribution. For instance, both MySQL image and Redis image could be built on top of Ubuntu image. They can share underlying system image and only store their own separate programs. Image registry is introduced as a database for developers to download existing images or submit their customized images. The ecosystem of application containers provides a convenient framework to build, ship, and run applications. In contrast, system containers support all sorts of filesystems and are not limited to one filesystem. Thus, system containers

Table 1. Comparison between application containers and system containers

	Application containers	System containers
Content	Contain a single process	Contain a complete runtime environment
Filesystem	Layered filesystem	Filesystem neutral
Design purpose	Run micro services	Provide a lightweight virtual machine
Usage scenario	Used for distributing applications	Used for providing underlying infrastructure

can not share images because they probably adopt completely different filesystems. Besides, operation system images like Ubuntu or CentOS are the only ones that originally supported by system containers. As a consequence, developers need to clone a container and then migrate to the other host to accomplish application distribution. With so many distinctions of implementation, we consider that application containers and system containers should be clarified more clearly and applied to different usage scenarios in the cloud environment.

2.2 Motivation

Due to the convenience of deploying applications, container technology triggers an overwhelming revolution for cloud platforms. Figure 1 demonstrates the architecture of mainstream container service. The CaaS (*Container as a Service*) layer is based on IaaS (*Infrastructure as a Service*) layer and provides container's running environment for developers to deploy their applications. At present, ECS and GKE have won the most shares of public container services in the industry. They allow developers to purchase virtual machines with pre-installed Docker running environment, and define their tasks and submit them to the cloud platform for execution. In contrast to the past, developers no longer need to take a long time to install softwares and tweak configurations. They could simply pull images from Docker hub and launch containers. Thus, public container service is a new solution for developers to build, deploy, and run their applications in an efficient method.

But one major deficiency of existing public container services is cloud platforms tend to run containers in the virtual machines to acquire mature support from existing management tools. Evidently, adding an extra virtual machine layer between the bare metal and container service probably generates unnecessary performance overhead. In principle, the essence of container service is to provide container running environment for developers, no matter what kind of underlying infrastructure it is. With the coming mature of container technology, the IaaS layer could be merged into CaaS layer. Nowadays container technology uses namespaces to isolate users, processes, and network between containers in the same host, and empower CGroups to control CPU, memory, and I/O usage for each container in fine-grained measure. Thus, the enhanced resource isolation and management mechanisms in containers could provide a

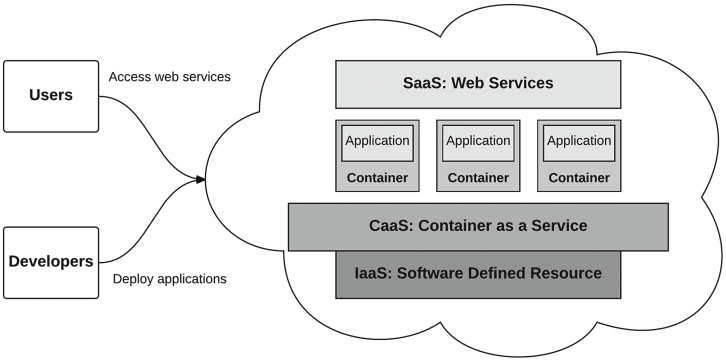

Fig. 1. The architecture of mainstream container service

sufficient virtual environment for tenants in shared resource environment. Based on current industry status, it is worth a comprehensive study to measure the exact overhead of the virtual machine layer and discuss how to provide container service appropriately.

3 Experimental Methodology

To systematically evaluate container performance in cloud environment, our experiments consist of two parts: evaluation for container technologies and evaluation for public container services.

3.1 Evaluation for Container Technologies

In this part, we will conduct a series of experiments on Docker and LXC to evaluate the performance differences between application container and system container. We use micro benchmarks to measure CPU performance, memory bandwidth, disk I/O, and network latency for our experimental objects. We also analyze the underlying implementation of different containers to figure out the fundamental reasons that lead to performance differences.

Meanwhile, in order to investigate the exact overhead introduced by extra virtual machine layer between the bare metal and container, we also measure the performance of Docker-Machine which is a virtualization tool to provide Docker environment by installing Docker in a virtual machine. We compare these three different forms of container to make an evaluation for container technologies.

3.2 Evaluation for Public Container Services

In this part, we will inspect the service quality of public container services. We choose ECS and GKE as our experimental objects because these two are most

influential cloud platforms in the industry. So far, these two container services have gained a certain level of popularity and created several successful user cases.

Although both ECS and GKE provide highly scalable and high performance container management services, there also exist several differences between ECS and GKE. First, ECS is based on Amazon EC2, and GKE is based on Google Compute Engine. Amazon EC2 offers more options for developers to purchase instances with different hardware architectures for different usages, even for graphical calculation. In contrast, Google Compute Engine offers one unified hardware architecture that allows developers to customize the number of vCPUs and the capacity of memories. Second, ECS is tightly integrated with Amazon Web Services. Developers need to store their data in S3 (*Simple Storage Service*), and depend on web services including RDS (*Relational Database Service*) or EMR (*Elastic MapReduce*) to acquire a full range of support from AWS. However, GKE is more flexible because Kubernetes, the underlying management framework, permits GKE to access web services in other cloud platforms.

4 Performance Evaluation

4.1 Platform Setup

Local Platform. Our local testbed is a server with 32 cores Intel X5650 CPU and 64 GB memories. The operating system is Ubuntu 15.10, running with Linux 4.2 kernel. LXC is version of 1.1.3 and Docker is version of 1.9.1.

For different containers, we adjust CGroups control parameters to limit containers resource consumption to a same level. In other words, we only allow a container to occupy 2 vCPUs and 8 GB memories. For Docker-Machine, we create the virtual machine with 2 vCPUs and 8 GB memories as well. We conduct a series of experiments on a single container to measure performance differences between application containers and system containers. We also establish a container cluster that contains 8 computing nodes to evaluate the performance of distributed applications.

Cloud Platform. In order to unify the hardware specification, we choose m4.large instance on EC2 as standard computing node, which has 2 vCPUs and 8 GB memories. We purchase 8 m4.large instances to compose the container cluster. For GKE, we customize the instance for equivalent specification with EC2 for fair comparison.

4.2 Evaluation for Container Technologies

CPU Performance. In order to evaluate CPU performance [15], we adopt 473.astar and 450.soplex in SPEC CPU 2006 to test integer and floating computing capacity. SPEC CPU 2006 is an industry-standardized benchmark suite that test CPU performance. To be more specific, 473.astar is a path finding

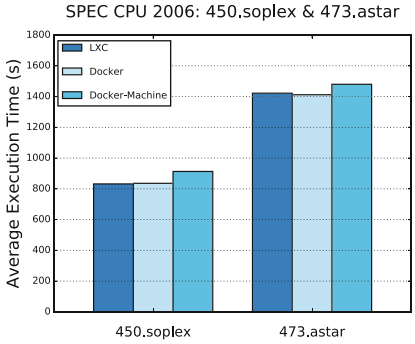

Fig. 2. Execution time of 450.soplex and 473.astar in SPEC CPU 2006. Lower is better

algorithm that derived from a portable game AI library, and 450.soplex solves a linear program using simplex algorithm and sparse linear algebra.

Figure 2 shows the result of SPEC CPU 2006. LXC and Docker have equivalent performance on CPU, and beat Docker-Machine by 8.4 % to 8.8 % in 450.soplex, 3.9 % to 4.6 % in 473.astar respectively. We can conclude that both LXC and Docker could utilize the CPU computing resource in a relatively high level, but extra virtualization layer will encumber the performance of containers. Although current hardware assisted virtualization technology allows virtual machine executing commands directly on CPU, a slightly delay is still exist for virtualization overhead.

Memory Bandwidth. We adopt STREAM [16] as memory benchmark, which is designed to measure sustainable memory bandwidth in high performance computers. At first, STREAM would allocate an array that is bigger than the machine's cache, then executes Copy, Scale, Add, and Triad operations in the memory. Since the program accesses memory with regular pattern, memory bandwidth is the main determinant of performance. At last, we collect the speed of each operation as the results. The version of STREAM is 5.10.

Table 2 shows the results of STREAM. No matter LXC, Docker, or Docker-Machine, they have similar memory bandwidth. We can conclude that there is no

Table 2. Memory bandwidth result

Stream operations	LXC	Docker	Docker-Machine
Copy (MB/s)	8420	8503	8311
Scale (MB/s)	8362	8564	8319
Add (MB/s)	9159	9085	8819
Triad (MB/s)	9199	9042	8964

significant difference on memory bandwidth for LXC and Docker. Even adding an extra virtualization layer only causes negligible overhead.

Disk I/O Performance. We adopt FIO [5] as the benchmark to test disk I/O performance. We collect IOPS of disk as the metric for evaluation. In FIO configuration file, we set *ioengine* to *libaio* (a Linux native asynchronous I/O library) in *O_DIRECT*, *iodepth* equals to 16 (number of I/O units to keep in flight), and *numjobs* equals to 8 (number of processes performing the same workload of this job). Besides, buffer size is 4KB and test file size is 1 GB. The version of FIO is 2.1.3.

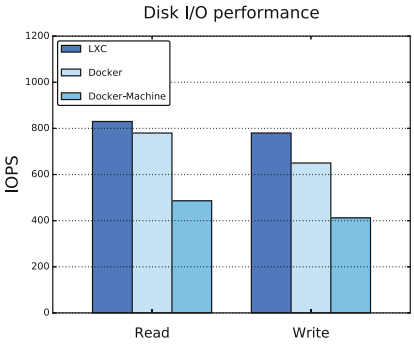

Fig. 3. Disk I/O performance of random read and random write. Higher is better

Figure 3 shows the results of FIO. We can observe that LXC has better disk I/O performance than Docker, and Docker-Machine suffers significant virtualization overhead and results in poor disk I/O performance. To be more specific, LXC advances Docker by 6.1 % in random read, and 16.6 % in random write. Docker-Machine falls behind Docker for 40.1 % in random read and 42.7 % in random write.

The disk I/O performance gap between LXC and Docker is caused by AUFS, the default filesystem in Docker container. Figure 4 illustrates the architecture of AUFS. AUFS is consists of image layers and container layer. Image layers are composed of multiple read-only AUFS branches. For each AUFS branch, it only saves differences relative to underlying branches to maximally support image reuse. Container layer is the writable layer to store modifications of a container. Eventually, a union mount point is introduced to provide a composite view of the filesystem for developers. In practice, AUFS could generate significant latency for write performance because the first time a container writes to any file, the file has to be located and copied into the container's top writable layer [1]. Latency will increase when file size is large or this file is saved in lower AUFS branch. Thus, file searches in AUFS branches and the requirement to copy files into top writable layer result in the extra disk I/O latency for Docker.

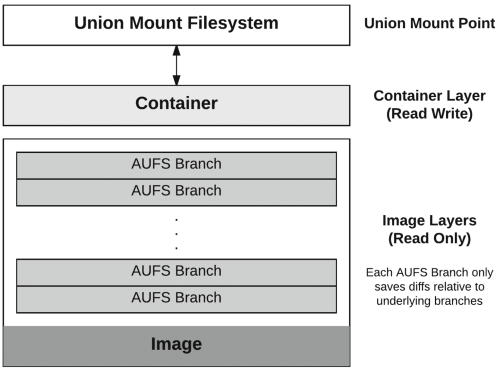

Fig. 4. The architecture of AUFS

For the reasons given above, we suggest developers prefer LXC to Docker when executing massive disk I/O requests. As for Docker-Machine, device emulation is the major source for poor disk I/O performance because every I/O operation in virtual machines must go through QEMU. Thus, we can conclude that adding an extra virtualization layer between the bare metal and container service will cause severe disk I/O latency.

Network Latency. For purpose of measuring network latency, we use Netperf's [12] request-response mode to test round trip latency. In request-response mode, client will send a 100 bytes packet to server, and server will reply it immediately after receiving the packet. This request-response action will repeat over and over again until being manually stopped. Thus, we can calculate network latency by counting the number of request/response in a specified period time. To avoid network congestion or other issues, we set up Netperf client in the host and communicate with Netperf server in the container.

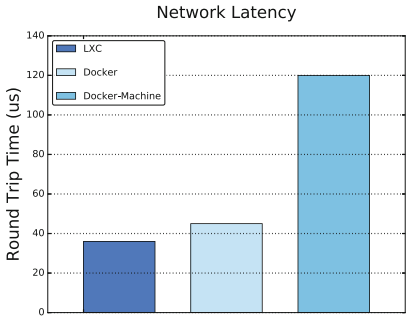

Fig. 5. Network latency of LXC, Docker and Docker-Machine. Lower is better

Figure 5 shows the result of Netperf. Docker's network latency is 1.25 times of LXC, and Docker-Machine greatly increases the network latency to 120 μs for each round trip. Considering propagation delay could be neglected in local network, the main source of latency is processing delay.

Both LXC and Docker use network namespace to create a virtual ethernet pair between the host and container, and every network packet should go through the bridge network. Comparing to LXC, Docker adds a NAT (*Network Address Translation*) mechanism to expose specific network ports. Only by port mapping, services in Docker containers could be accessed normally from the outside. As a result, the NAT mechanism increases the network latency and makes it difficult for service discovery across different hosts. However, LXC is much easier to use the full stack of Linux capabilities to manage container's network with no limitations.

In the case of Docker-Machine, network packets go through a virtual device created by virtio from guest OS to host OS. The combination of hardware emulation and NAT mechanism causes the severe network latency of Docker-Machine. Therefore, we can conclude that virtualization overhead for network latency is serious, and developers could prefer LXC to Docker for network-intensive applications in the perspective of low latency.

4.3 Evaluation for Public Container Services

In this part, we will inspect the service quality of public container services by using HiBench [10] to evaluate the performance of distributed data processing systems, Hadoop and Spark, on different container platforms. Nowadays, distributed data processing applications have been used extensively in cloud environment. We choose several typical workloads, including WordCount, TeraSort, PageRank, and Kmeans, as CPU-bound and I/O-bound workloads respectively to evaluate the performance. We briefly introduce these four workloads as follows:

- WordCount: WordCount is a classical MapReduce workload, which counts the number of occurrences for each word in input text. WordCount is a CPU-Bound workload. In our test, the input data is 10 GB and generated by RandomWriter and RandomTextWriter in Hadoop distribution.
- TeraSort: TeraSort is a classical workload, which sorts massive data as fast as possible. TeraSort is a CPU-Bound workload in map stage, but it turns into an I/O-Bound workload in reduce stage. In our test, the input data is 10 GB and generated by TeraGen in Hadoop distribution.
- PageRank: PageRank is a link analysis algorithm used widely in web search engines, which calculates the ranks of web pages according to the number of reference links. PageRank is a CPU-Bound workload. In our test, the input data is 0.5 GB.
- Kmeans: Kmeans is well-known clustering algorithm for data mining to partition input data into k clusters. In map stage, Kmeans is a CPU-Bound workload for data training. In reduce stage, Kmeans becomes an I/O-bound workload for data clustering. The input data is 4 GB and generated by DenseKmeans.

To carry out the experiment, we need to set up a cluster of 8 computing nodes. At first, we install required softwares and configure them properly in all computing nodes. Then we run HiBench, a benchmark developed by Intel to test Hadoop and Spark system, in the cluster to test aforementioned four workloads. For each workload, we run five times to eliminate performance deviation. At last, we collect each workload's execution time as the experiment results. The version of Hadoop is 2.7.2, and the version of Spark is 1.6.0.

Figures 6 and 7 show the results of Hadoop and Spark system in different container testbeds respectively. Due to the performance gap between underlying instances, ECS has better service quality than GKE in all workloads. To be more accurate, ECS takes the lead of GKE in a range from 1.4 % to 12.2 %. Although Docker-Machine in the local testbed is far surpassed by ECS and GKE, we find out both ECS and GKE have a certain degree of performance gap comparing to

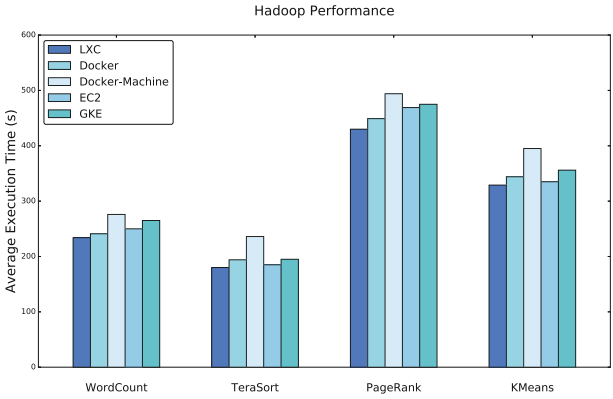

Fig. 6. Execution time of Hadoop in different container testbeds. Lower is better

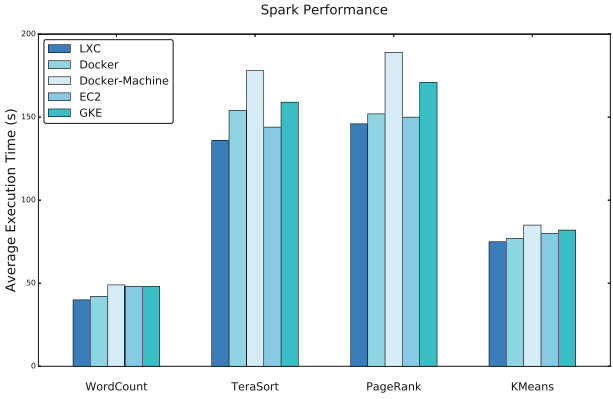

Fig. 7. Execution time of Spark in different container testbeds. Lower is better

LXC and Docker in the local testbed. That means running containers directly on the bare metal could gain more efficacy than running containers in virtual machines. Public container services still have enough room for improving their performance.

The experiment results also reveal that LXC has better performance than Docker in these scenarios in a range from 2.5 % to 11.6 %. According to previous experiment results, Docker has equivalent CPU performance comparing to LXC, but Docker would suffer severe network and disk I/O latencies when massive I/O requests arrive. Therefore, we recommend that users should carefully choose suitable container to execute I/O-Bound workload.

5 Related Works

Container technology is gradually gaining more and more attention from the research community, and comparisons between containers and virtual machines have been extensively conducted in many research works. Felter et al. [4] use a suite of workloads to measure the performance differences between Docker and KVM. They conclude that containers result in equal or better performance than virtual machines in almost all cases. Agarwal et al. [2] evaluate the density and start-up latency of LXC and KVM. They conclude that the overall density is highly dependent on the most demanded resource and the small memory footprint of containers could raise the density to a higher level. Morabito [11] measures power consumption of virtual machines and containers in different applications. Xavier et al. [17] measure the performance of containers when applying them into HPC environment. They conclude that containers could obtain a very low overhead leading to near-native performance, but performance isolation in containers is immature. However, these studies do not discuss the distinctions between different containers and the impact of adding an extra virtualization layer between the bare metal and containers.

Evaluation for cloud platform could provide a valuable guidance for developers to choose appropriate platform to deploy applications and save cost. Schad et al. [14] use established micro benchmarks to measure performance variance on EC2, and find out EC2's performance varies a lot and often falls into two bands having a large performance gap in-between. Taking performance variation and performance isolation [8,19] into account, studies [13] aim to propose QoS-aware frameworks to improve resource utilization in the cloud through diverse ideas and methods. Xu et al. [18] summarize the performance overhead of virtual machines in the cloud, and analyze the challenges of cloud platforms. Latest studies [6] review the benefits and requirements of container services and discuss the fitness of containers to facilitate applications in the cloud. On the basis of these studies, we make an inspection of the service quality of public container services including ECS and GKE.

6 Conclusion

Motivated by the increasing popularity of container technology in the industry, in this paper we conduct a performance study of containers in cloud environment. Our study focuses on two points: one is performance differences between application containers and system containers, and the other is performance overhead caused by extra virtual machine layer between the bare metal and containers. What's more, we make an inspection of service quality of public container services. To carry out experiments, we first conduct a series of experiments to measure CPU performance, memory bandwidth, disk I/O performance, and network latency among LXC, Docker, and Docker-Machine. Then we evaluate public container services by testing the performance of typical distributed data processing systems, Hadoop and Spark, on different container platforms.

Our experiments distinguish the differences between applications containers and system containers. We conclude that system containers have performance advantage on executing I/O-Bound workload and are more suitable to provide underlying infrastructure service. Our experiments also prove that running containers in virtual machine would result in severe disk I/O performance degradation up to 42.7 % and network latency up to 233 %. Our inspection for public container services reveal that the service quality of public container services are competitive, but their infrastructure architectures are quite controversial. Although both ECS and GKE keep updating their hardwares constantly, the performance of LXC and Docker in the local testbed could surpass these two public container services in most cases. Therefor, we learn that the performance overhead caused by extra virtual machine layer can not be neglected in public container services, and cloud platforms could acquire better performance by running containers directly on the bare metal.

To summarize, the performance study we present in this paper provides several suggestions for developers to choose appropriate containers in different usage scenarios. It also proposes several suggestions for the establishment of container services. With the coming mature of container technology, we believe containers could be extensively used and play a very significant role in the cloud environment.

Acknowledgments. This research is supported by National Science Foundation of China under grant No. 61232008, National Key Research and Development Program under grant 2016YFB1000500, National 863 Hi-Tech Research and Development Program under grant No. 2015AA01A203.

References

1. Docker and aufs in practice. https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/
2. Agarwal, K., Jain, B., Porter, D.E.: Containing the hype. In: Proceedings of the 6th Asia-Pacific Workshop on Systems, pp. 8–16. ACM (2015)
3. Amazon web service. https://aws.amazon.com/

4. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 171–172. IEEE (2015)
5. Fio. https://github.com/axboe/fio
6. Fu, S., Liu, J., Chu, X., Hu, Y.: Toward a standard interface for cloud providers: the container as the narrow waist. IEEE Internet Comput. **20**(2), 66–71 (2016)
7. Google compute engine. https://cloud.google.com/
8. Govindan, S., Liu, J., Kansal, A., Sivasubramaniam, A.: Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, pp. 22–33. ACM (2011)
9. Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Slacker: fast distribution with lazy docker containers. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies, pp. 181–195. USENIX (2016)
10. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The hibench benchmark suite: characterization of the mapreduce-based data analysis. In: Proceedings of the 26th IEEE International Conference on Data Engineering Workshops, pp. 41–51. IEEE (2010)
11. Morabito, R.: Power consumption of virtualization technologies: an empirical investigation. In: Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing, pp. 522–527. IEEE (2015)
12. Netperf. http://www.netperf.org/netperf/
13. Novaković, D., Vasić, N., Novaković, S., Kostić, D., Bianchini, R.: Deepdive: transparently identifying and managing performance interference in virtualized environments. In: Proceedings of the 2013 USENIX Annual Technical Conference, pp. 219–230. USENIX (2013)
14. Schad, J., Dittrich, J., Quiané-Ruiz, J.A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proc. VLDB Endow. **3**(1–2), 460–471 (2010)
15. SPEC CPU 2006. http://www.spec.org/cpu2006/
16. Stream. https://www.cs.virginia.edu/stream/
17. Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A.F.: Performance evaluation of container-based virtualization for high performance computing environments. In: Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 233–240. IEEE (2013)
18. Xu, F., Liu, F., Jin, H., Vasilakos, A.V.: Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions. Proc. IEEE **102**(1), 11–31 (2014)
19. Zhang, X., Tune, E., Hagmann, R., Jnagal, R., Gokhale, V., Wilkes, J.: Cpi 2: CPU performance isolation for shared compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp. 379–391. ACM (2013)