A Comparative Study on the Performance Isolation of Virtualization Technologies

by

Zige Huang

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved August 2019 by the
Graduate Supervisory Committee:

Ming Zhao, Chair
Ruoyu Wang
Mohamed Sarwat

ARIZONA STATE UNIVERSITY

December 2019

ABSTARCT

Virtualization technologies are widely used in modern computing systems to deliver shared resources to heterogeneous applications. Virtual Machines (VMs) are the basic building blocks for Infrastructure as a Service (IaaS), and containers are widely used to provide Platform as a Service (PaaS). Although it is generally believed that containers have less overhead than VMs, an important tradeoff which has not been thoroughly studied is the effectiveness of performance isolation, i.e., to what extent the virtualization technology prevents the applications from affecting each others performance when they share the resources using separate VMs or containers. Such isolation is critical to provide performance guarantees for applications consolidated using VMs or containers. This paper provides a comprehensive study on the performance isolation for three widely used virtualization technologies, full virtualization, para-virtualization, and operating system level virtualization, using Kernel-based Virtual Machine (KVM), Xen, and Docker containers as the representative implementations of these technologies. The results show that containers generally have less performance loss (up to 69% and 41% compared to KVM and Xen in network latency experiments, respectively) and better scalability (up to 83.3% and 64.6% faster compared to KVM and Xen when increasing number of VMs/containers to 64, respectively), but they also suffer from much worse isolation (up to 111.8% and 104.92% slowdown compared to KVM and Xen when adding disk stress test in TeraSort experiments under full usage (FU) scenario, respectively). The resource reservation tools help virtualization technologies achieve better performance (up to 85.9% better disk performance in TeraSort under FU scenario), but cannot help them avoid all impacts.

ACKNOWLEDGMENTS

First of all, I would like to thank my thesis advisor Dr. Ming Zhao, Associate Professor of ASU School of Computing, Informatics, and Decision Systems Engineering, who provided careful guidance in the past two and half years. Every time I talked with Prof. Zhao, I got interesting ideas about my research and life. He consistently inspired me to think and guided me in the right direction. Moreover, I was impressed by Zhao's questioning spirit and often surprised by his unique and sharp views during the group meeting.

I also thank my two committees for their patient guidance. Prof. Mohamed Sarwat provided useful professional advice in related work, and Prof. Ruoyu Wang provided indispensable guidance in preparing my thesis presentation and CPU research part.

I would also like to thank two talents in the VISA lab who provided help for my research and thesis validation: Qirui Yang and Wenji Li. Without their patient help and enthusiastic input, my research could not go smoothly.

I spent a really happy time in the VISA lab. I hope this thesis is not the end, but the beginning.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . vi

LIST OF FIGURES . . . viii

CHAPTER

1 INTRODUCTION . . . 1

2 BACKGROUND . . . 4

2.1 Full Virtualization . . . 4

2.2 Para-Virtualization . . . 6

2.3 OS-level Virtualization . . . 7

3 EXPERIMENT SETTING . . . 9

4 PERFORMANCE STUDY . . . 11

4.1 Methodology . . . 12

4.2 CPU-LINPACK . . . 12

4.3 Memory-STREAM . . . 13

4.4 Network-Netperf . . . 14

4.5 Disk-FIO . . . 15

4.6 Summary . . . 15

5 ISOLATION STUDY . . . 17

5.1 Methodology . . . 17

5.1.1 SPECweb . . . 21

5.1.2 TPC-DS . . . 24

5.1.3 Hadoop TeraSort . . . 25

5.2 Summary of Results . . . 26

5.2.1 CPU Isolation . . . 28

5.2.2 Disk I/O Isolation . . . 29

Chapter Page

5.2.3 Network Isolation 30
5.2.4 Memory Isolation 31
5.2.5 Fork Isolation 31
6 SCALABILITY STUDY 33
6.1 Methodology 33
6.1.1 Increasing the Number of Threads 34
6.1.2 Increasing the Number of VMs/Containers 35
7 RELATED WORK 37
7.1 Performance Study 37
7.2 Scalability Study 38
7.3 Isolation Study 38
8 CONCLUSIONS 41
REFERENCES 43
APPENDIX
A COMMAND LINES FOR PERFORMANCE STUDY 45
B COMMAND LINES FOR ISOLATION STUDY 47
C COMMAND LINES FOR SCALABILITY STUDY 51

LIST OF TABLES

Table Page

4.1 Performance Degradation of KVM, Xen, and Docker (%) 11

4.2 Performance Difference Compared with Che *et al.* (2010) (%) (Difference = Our Results - Che *et al.* (2010)'s Results) 12

4.3 Performance Difference Compared with Felter *et al.* (2015) (%) (Difference = Our Results - Felter *et al.* (2015)'s Results)................. 13

5.1 SUT (VMs/Containers) Configuration 17

5.2 Resource Reservation Setting for Stress VMs/Containers 19

5.3 Average Resource Profile for Isolation Benchmarks 20

5.4 Stress Test Suit Description.. 21

5.5 Stress Test Suit Setting 22

5.6 CPU Isolation Degradation (%) 26

5.7 Disk Isolation Degradation (%) 27

5.8 Network Transmit Isolation Degradation (%)........................ 27

5.9 Network Receive Isolation Degradation (%) 27

5.10 Memory Isolation Degradation (%) 28

5.11 Fork Isolation Degradation (%) 28

6.1 Containers and VMs Configuration for Scalability Study 33

A.1 Command Lines of Benchmarks (LINPACK, STREAM, Netperf) Used in Performance Study 46

A.2 Command Lines of Benchmarks (FIO) Used in Performance Study 46

B.1 Command Lines of SPECweb Used in Isolation Study 48

B.2 Command Lines of TPC-DS Used in Isolation Study 48

B.3 Command Lines of Hadoop Used in Isolation Study 48

B.4 Command Lines of Stress Tests Used in Isolation Study 49

Table | Page

B.5 Command Lines of CPU Reservation Used in Isolation Study 49

B.6 Command Lines of Disk Reservation Used in Isolation Study 50

B.7 Command Lines of Network Reservation Used in Isolation Study 50

C.1 Command Lines of Apache HTTP Server Compilation Used in Scalability Study 52

LIST OF FIGURES

Figure | Page

2.1 KVM Architecture 4

2.2 Xen Architecture 5

2.3 Docker Architecture 5

5.1 Isolation Experiment Methodology 18

5.2 Resource Reservation Methodology 18

5.3 SPECweb Benchmark Setting. 20

5.4 TPC-DS Benchmark Setting. 22

5.5 TeraSort Benchmark Setting. 23

6.1 Setting of Increasing the Number of Threads 34

6.2 Setting of Increasing the Number of VMs/Containers 34

6.3 Performance of Increasing the Number of Threads 35

6.4 Performance of Increasing the Number of VMs/Containers 36

Chapter 1

INTRODUCTION

Nowadays, scholars, politicians, and even kids take advantage of cloud services every day without realizing it. When you watch videos via Netflix, you are using cloud services because many entertainment services of Netflix are based on Amazon Web Service (AWS) ama (2006c), which is the most widely used cloud platform. With various cloud services, you can access information and resources where the network connection is available. Moreover, the increasing demand from cloud users has led the cloud providers to adopt the resource sharing mechanism by taking advantage of virtualization. Virtualization technologies enable cloud computing's elasticity and become a driving factor behind the success of the emerging commercial cloud computing paradigms (e.g., EC2 ama (2006a), ECS ama (2006b), GCE Krishnan and Gonzalez (2015)).

Various virtualization technologies provide support to cloud computing in different aspects due to their unique features. Full virtualization provides an interface that is identical to the underlying hardware. Thus, full virtualization supports unmodified operating systems (OSes) and applications (e.g., VMware vmw (1998), KVM Kivity Qumranet *et al.* (2007)). Para-virtualization provides a software interface which is similar to the underlying hardware-software interface. Compared with full virtualization, para-virtualization makes modifications on guest OSes to reduce virtualization overheads. OS-level virtualization provides an environment that is identical to the underlying OS, and it can run applications that the corresponding OS supports (e.g., Docker Fink (2014)).

Understanding how different virtualization technologies perform in various scenarios helps cloud users better leverage them to provide diversified features. Emerging lightweight virtualizations are known for their less performance loss and good scalability, but they may have some hidden issues in isolation. We discussed isolation performance in-depth in our work by setting different system utilization and utilizing resource reservation tools. Apart from that, we also verified that our virtualization performance study is consistent with related works' performance study. Furthermore, we studied the scalability by increasing the number of benchmark threads and VMs/containers. Our results provide important performance references for future research.

Since performance, isolation, and scalability are crucial to cloud computing performance, many papers have contributed to this study. Although Pu *et al.* (2010), Mardan and Kono (2016), Xavier *et al.* (2015) discussed isolation problem of different virtualization technologies, they have some limitations in the experimental methods. First, system utilization was not addressed. We found that the system resource occupancy of some previous works is lower than 60% after reproducing their experiments. Since low system utilization cannot create serious resource competition environments, the isolation results from previous works fail to reflect the real isolation of related virtualization technologies. We adjusted the stress test's parameter (e.g., threads number) to create 90% system utilization and full system utilization scenarios. Second, there are many resource reservation tools which can limit virtualization interference, but none of them were evaluated before. We discussed the efficiency of resource reservation tools by using them in 90% system utilization and full system utilization scenarios.

Although Soriga and Barbulescu (2013), Li and Xia (2016) studied scalability of different virtualization technologies, the experiments about increasing number of

benchmark threads were not addressed. We evaluated virtualization scalability by increasing the number of benchmark threads and VMs/containers because this experiment can reflect the performance of virtualization technology to meet users' growing needs of threads.

The contributions of our work are reflected in the following points:

- We conducted an extensive performance study considering performance, isolation, and scalability for KVM, Xen, and Docker.
- We designed experiments under different system utilization (90% usage and full usage scenarios). KVM's performance is relatively stable as the resource contention increases, whereas Docker's performance loss increases dramatically.
- We evaluated the effectiveness of resource reservation tools under different system utilizations. Cgroups, and Traffic Controller help Docker gain 11.25% better performance on average and up to 87.88% better performance when the disk bandwidth is under contention. But resource reservation tools are not good enough to free Docker from any influence.
- We found interesting isolation problems which were ignored before. We noticed that Docker shows 25.13%, 113.8%, and 21.14% performance loss in CPU, disk, and memory under full system utilization, respectively.

The rest of this paper is organized as follow: Section II introduces the background of different virtualization technologies, and Section III describes the environment setting. Section IV, V, VI present performance, isolation and scalability evaluation, respectively. Section VII reviews the related studies and Section VIII concludes our findings.

Chapter 2

BACKGROUND

2.1 Full Virtualization

Full virtualization means that different guest machines co-exist in a host machine through a virtual imitation of the hardware layer. The hardware imitation layer provides an interface that is identical to the underlying hardware, and thus it can run unmodified operating systems (OSes). Full virtualization allows the system to create guests with various OSes which have no knowledge of the host. Take the hardware support for example, full virtualization can build it in the CPU which helps trap and virtualize hardware-specific operations or commands without guest awareness.

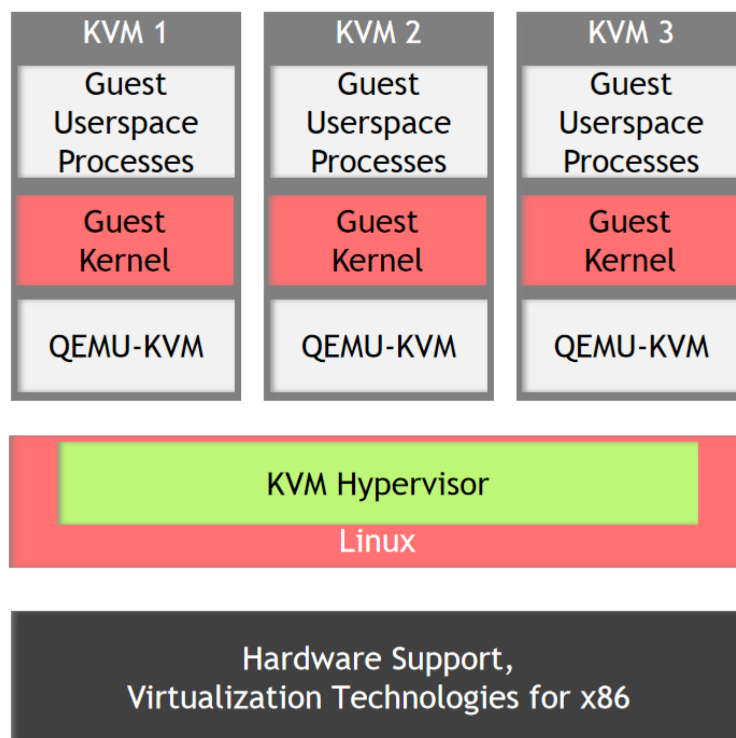

Figure 2.1: KVM Architecture

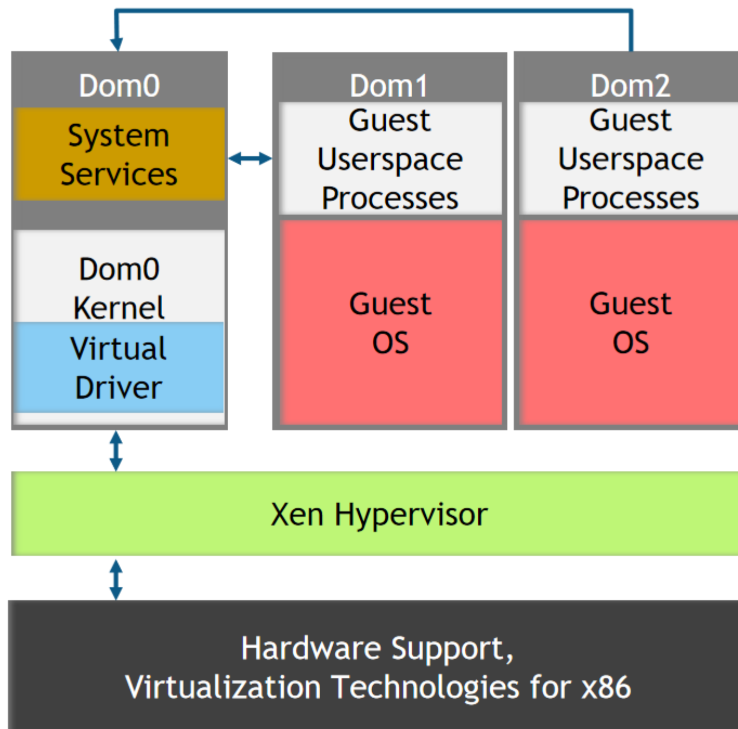

Figure 2.2: Xen Architecture

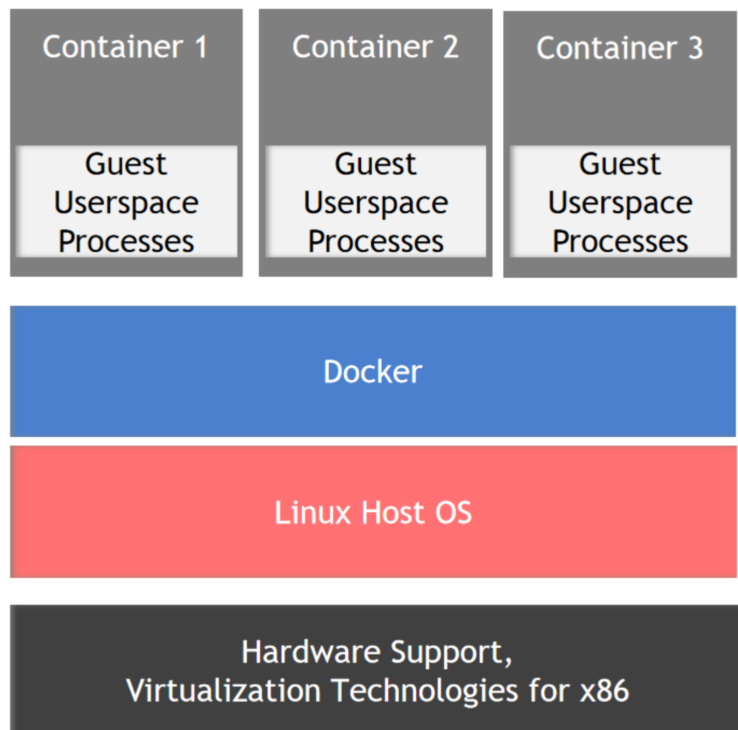

Figure 2.3: Docker Architecture

Machines of full virtualization are widely-used in cloud computing as the infrastructure as a service (IaaS). Google applies KVM on its IaaS cloud management. Amazon Web Service uses the KVM as the main infrastructural tool when implementing their new product Amazon EC2 C5. Moreover, the latest Alibaba Cloud application adopts KVM to its backend management. Full virtualization challenges resource saving due to its fully virtualized layer; however, this layer also delivers a promising isolated performance for each guest.

In our experiments, we evaluate the full virtualization via investigating the KVM. Figure 2.1 shows the details of KVM architecture. For the hypervisor communication, KVM calls the memory management and the process management subsystem by converting the linux kernel as a hypervisor. KVM currently uses virtio, a device-driven framework for Linux to provide an I/O framework for host and guest interaction. Virtio installs front-end drivers in the guest OS kernel and back-end drivers in QEMU Bellard (2005).

2.2 Para-Virtualization

Para-virtualization provides a software interface to guests which is not identical to the underlying hardware-software interface. Thus, the difference between para-virtualization and full virtualization is the para-virtualization VMM has to run modified guest OS's code and takes the advantages of cooperation between host and guests. Moreover, the guests have the knowledge of their virtualization environments because para-virtualization management needs the cooperation from guests. Furthermore, para-virtualization is also based on the host and guest mode with a VMM that supports multiple operating systems.

Generally, para-virtualization shows a good performance on the cloud environment by taking advantage of the collaboration between guest processes and host processes.

The largest infrastructure cloud, Amazon EC2, originally took Xen as its hypervisor. Para-virtualization guarantees high efficiency by allowing the modified guests OS to collaborate with the hypervisor. But this design also greatly limits the popularity of para-virtualization because only limited operating systems are modifiable.

This paper evaluates the para-virtualization via examing Xen. Figure 2.2 shows the details of Xen architecture. Xen contains three major parts. Xen hypervisor is the lowest layer with the highest privilege on top of which running many guest operating systems. The Xen hypervisor manages the guests' activities across the physical resources. Domain 0 is a unique guest operating system running on the Xen Hypervisor with direct access to the hardware and privileges to manage other guest operating systems (Domain U). Domain U is a guest operating system that has no direct access to the hardware resources, but can exist independently and in parallel. Xen's resource allocation and isolation all rely on Domain 0.

2.3 OS-level Virtualization

OS-level virtualization is a lighter weight choice virtualization comparing with the hypervisors. It partitions the host's physical resources and builds multiple isolated guest resource instances on the same OS. Each guest occupies the entire resources exclusively. This design reduces many overheads comparing with the hypervisor-based virtualization on the fact that there is no need to translate and pass the instructions across layers.

OS-level virtualization has the advantage that it occupies less server resources whereas can create thousands of containers in seconds, which makes OS-level virtualization increasingly important in Platform as a Service (PaaS). IBM's private cloud is based on Docker and Kubernates. Oracle's cloud and tutum are also based on Docker. Although OS-level virtualization has many different advantages, it still

causes the security and isolation concerns which are the open problems in academic and industry.

Docker is one of the most popular container frameworks. Figure 2.3 shows the details of Docker architecture. It is increasingly incorporated on platforms that require high scalability and low performance overhead. Docker can build an entire container with Dockerfile easily, whereas the traditional virtualization technologies are more complex in creating the whole system. Moreover, Docker shows promising flexibility of instantaneous replication. (e.g., a Dockerfile can create a new container based on the built container image in seconds.) Docker creates new instances of six namespaces (i.e., Mount, UTS, IPC, PID, Network, and User namespaces). When it creates a container, it helps Docker build its isolation. Although Docker does brings numerous benefits as an OS-level virtualization, it also brings typical isolation issues due to its ineffective resource management.

Chapter 3

EXPERIMENT SETTING

We conducted a thorough comparison of three virtualization techniques and adjusted the system settings to provide complete fairness in all the aspects that may affect the SUT performance. All of our experiments run on the ASU cloud, and the following experimental settings are consistent with the information below.

- Native/ASU cloud node: ASU cloud node is based on Ubuntu Server 14.04.01 (Linux kernel 3.13) with 2.4 GHz Intel (R) Xeon (R) CPU processors for a total of 16 hyperthreaded cores, 60 GB memory, and EXT4 file system on 2 TB HDD. ASU cloud nodes are connected with each other through Mellanox 10 Gbps Ethernet. When we ran experiments directly on the ASU cloud node, we strictly restricted the processes' resources as the same as the SUT by setting Cgroups' limits (Cgroups version 1).

- KVM: We used KVM to create VMs each with 4 virtual CPUs (vCPUs), 10 GB virtual memory, 100 GB virtual disk, bridged network mode, and EXT4 file system. KVM's images are stored as qcow2 format files on host file systems.

- Xen: We used Xen 4.8.1 to create Xen domains with 4 vCPUs, 10 GB virtual memory, 100 GB virtual disk, bridged network mode, and EXT4 file system. The Xen's images are stored as files located on host file systems.

- Docker: We used Docker 17.06.1-ce with 4 vCPUs, 10 GB virtual memory, 100 GB virtual disk, bridged network mode, and EXT4 file system. We chose Device

Mapper as the storage device which mappped different layers of Docker to the physical disk.

Chapter 4

PERFORMANCE STUDY

Table 4.1: Performance Degradation of KVM, Xen, and Docker (%)

Benchmark	Measured Operation	KVM	Xen	Docker
LINPACK	Numerical linear algebra	29	19	1
STREAM	Copy	2	2	0
	Add	3	3	2
	Scale	3	2	1
	Traid	3	2	2
Netperf	TCP RR latency	72	49	8
	UDP RR latency	78	42	9
	TCP send throughput	0	0	0
	TCP receive throughput	0	0	0
	UDP send throughput	22	19	0
	UDP receive throughput	21	19	2
FIO	Random read	40	22	0
	Random write	45	28	0
	Random mix	47	27	2
	Seq read	4	2	0
	Seq write	4	3	0

Table 4.2: Performance Difference Compared with Che *et al.* (2010) (%) (Difference = Our Results - Che *et al.* (2010)'s Results)

Benchmark	Measured Operation	KVM	Xen
LINPACK	Numerical linear algebra	+16	+19
STREAM	Copy	-5	-6
	Add	-1	-1
	Scale	-3	-5
	Traid	-1	-3
Netperf	TCP send throughput	-14	-13
	TCP receive throughput	0	0
	UDP send throughput	+1	+6
	UDP receive throughput	+18	+18
FIO	Seq read	+2	+2
	Seq write	-16	+28

4.1 Methodology

This section investigates the performance loss introduced by full virtualization (KVM), para-virtualization (Xen), and OS-level virtualization (Docker). All the results are normalized by the results from the native machine (degradation = (Native-System Under Test (SUT))/Native)). Each result is averaged over 5 runs.

4.2 CPU-LINPACK

LINPACK benchmark measures how fast the computers solve dense linear equations with floating-point operations. We used Intel Optimized LINPACK Benchmark with Intel Math Kernel Library 2018.0.006 in our experiments.

Table 4.3: Performance Difference Compared with Felter *et al.* (2015) (%) (Difference = Our Results - Felter *et al.* (2015)'s Results)

Benchmark	Measured Operation	KVM	Docker
LINPACK	Numerical linear algebra	-26	+1
STREAM	Copy	+1	0
	Add	+2	+2
	Scale	+2	+1
	Traid	+2	+2
Netperf	TCP send throughput	-1	0
	TCP receive throughput	+21	0
FIO	Random read	-5	0
	Random write	-2	0
	Random mix	-15	+2
	Seq read	+4	0
	Seq write	+3	0

Table 4.1 shows the LINPACK performance on KVM, Xen, and Docker. Xen outperforms KVM by 10% and is slower than Docker by 18%. KVM's performance is the worst with 29% performance loss, whereas Docker can achieve almost identical performance as the native with 1% performance loss.

4.3 Memory-STREAM

As the available CPU cores increase in a processor, memory resource becomes more and more important for the whole system. If a CPU core cannot fetch data from memory on time, it reduces the system efficiency. STREAM uses four basic operations to measure sustainable memory bandwidth and corresponding computing speed. The four operations are as follows:

- COPY: $a[i] = b[i]$. This operation reads the value from a memory cell and then writes the value to another memory cell;
- ADD: $a[i] = b[i] + c[i]$. This operation reads two values from the memory cells, adds them, and writes the result to another memory cell;
- SCALE: $a[i] = factor * b[i]$. This operation reads two values from the memory cells, multiplies them, and writes the results to another memory cell;
- TRIAD: $a[i] = b[i] + factor * c[i]$. This operation reads two values from two memory cells. After multiplication and addition, it writes the result to another memory cell.

Table 4.1 shows memory performance loss of KVM, Xen, and Docker comparing with the native system. For all operations, the virtual machines introduce up to 3% overhead, and Docker introduces at most 2% overhead. The STREAM performance among three virtualization technologies are similar with no more than 2% difference.

4.4 Network-Netperf

Netperf measures the network throughput and latency of SUT for TCP and UDP communications. Netperf works in a client/server mode and our SUT runs as the Netperf client/server. In the network throughput experiments, the client sends a 100-byte request and the server sends a 200-byte response. We set the sending socket size to 16 KB and the running time as 60 seconds. In network latency experiments, Netperf calculates latency statistics on runtime from client sending a package to receiving a reply.

Table 4.1 shows the normalized network performance loss of KVM, Xen, and Docker. For network latency experiments, Xen shows 41% higher latency in TCP

transmission and 33% higher latency in UDP transmission comparing with Docker. The worst case is the KVM which shows 23% and 36% overhead for TCP and UDP transmissions, respectively comparing with Xen. For the throughput experiments, all the virtualization technologies show identical performance to the native system in TCP mode. In UDP mode, Docker also shows almost identical network throughput (up to 2% degradation) to the native system, whereas Xen's perfermance is up to 19% lower than Docker. The worst two cases come from KVM. KVM shows 22% disadvantage in UDP send mode and 19% disadvantage in UDP receive mode than Docker.

4.5 Disk-FIO

We used FIO benchmark to measure the IOPS performance. During the experiments, we set the ioengine to psync and the block size to 32 KB. We started 20 threads together and each thread had a 2 GB workload. All FIO experiments were under DIRECT mode to avoid the influence from the page cache.

Table 4.1 shows the normalized degradation of disk performance of KVM, Xen, and Docker. In the sequential access mode, all the virtualization technologies bring little overhead which is lower than 4%. In the random access mode, Docker has almost identical disk I/O performance as the native system. In ramdom FIO experiments, KVM shows 17% to 22% worse performance than Xen, and Xen brings 22% to 28% overhead than the Docker.

4.6 Summary

Our performance experiment results are generally consistent with the conclusions made by the studies from Che *et al.* (2010) and Felter *et al.* (2015). Table 4.2 and Table 4.3 show the difference between our results and related works about same mea-

sured operations. By adjusting specific parameters (e.g., threads number and block size) for FIO experiments, we noticed a 28% higher difference in XEN's random_write mode comparing withChe *et al.* (2010). Also, by using 10 Gbps network through ASU cloud, we noticed up to 14% lower and 21% higher performance comparing with Che *et al.* (2010) and Felter *et al.* (2015), respectively. Compared with Che *et al.* (2010), our performance study shows 16% and 19% better CPU performance for KVM and Xen, respectively. This may because we strictly restrict the CPU resources for the experimental processes (run directly on Native for baseline runtime).

Chapter 5

ISOLATION STUDY

5.1 Methodology

Performance isolation is a important factor for Quality of Service (QoS) in cloud computing where information and system resources are shared. It is critical to provide performance guarantees for applications consolidated using VMs or containers. If the runtime behavior of an application is affected by other running applications, it is difficult to predict its completion time. To quantify the performance isolation of the virtualization systems, we used CPU, memory, fork, and network bomb from the Isolation Benchmark Suit (IBS) Matthews *et al.* (2007) as stress tests to compete system resources with our SUTs. For disk bomb, we used FIO with the psync backend in O_DIRECT mode to bypass the operating system cache. Table 5.5 introduces different stress tests. In summary, the stress suit effectively models intensive resource competing scenarios in production conditions.

Figure 5.1 shows the basic methodology for isolation experiments. The SUT and the stress test machine are on the one cloud node. We designed four scenarios to test the performance isolation about CPU, disk, and network of virtualization technologies.

Table 5.1: SUT (VMs/Containers) Configuration

Containers and VMs	VCPUs	Virtual Memory	Virtual Disk
SPECweb Web Server	4	10 GB	100 GB
TPC-DS Server	4	10 GB	100 GB
Hadoop Master/Slave	1	1 GB	15 GB

Figure 5.1: Isolation Experiment Methodology

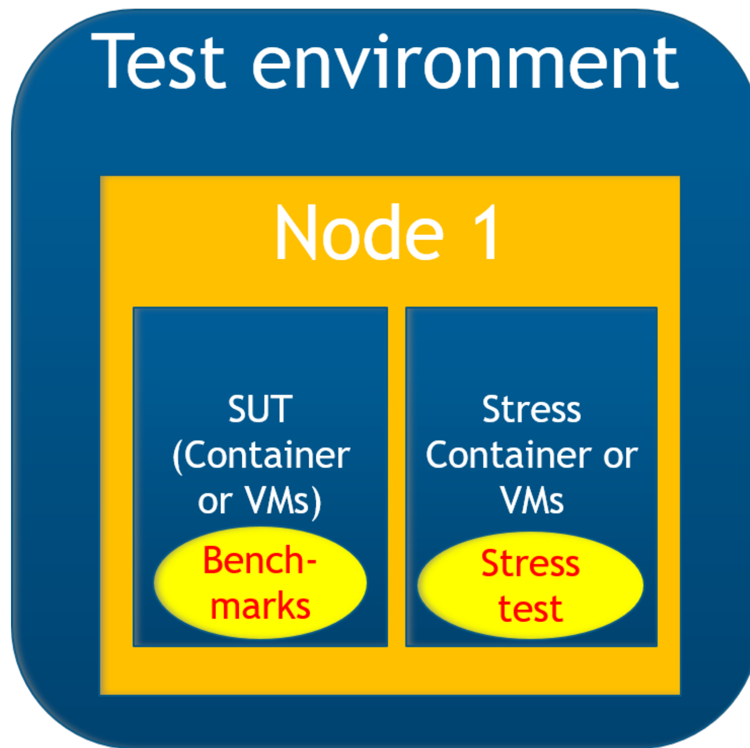

Figure 5.2: Resource Reservation Methodology

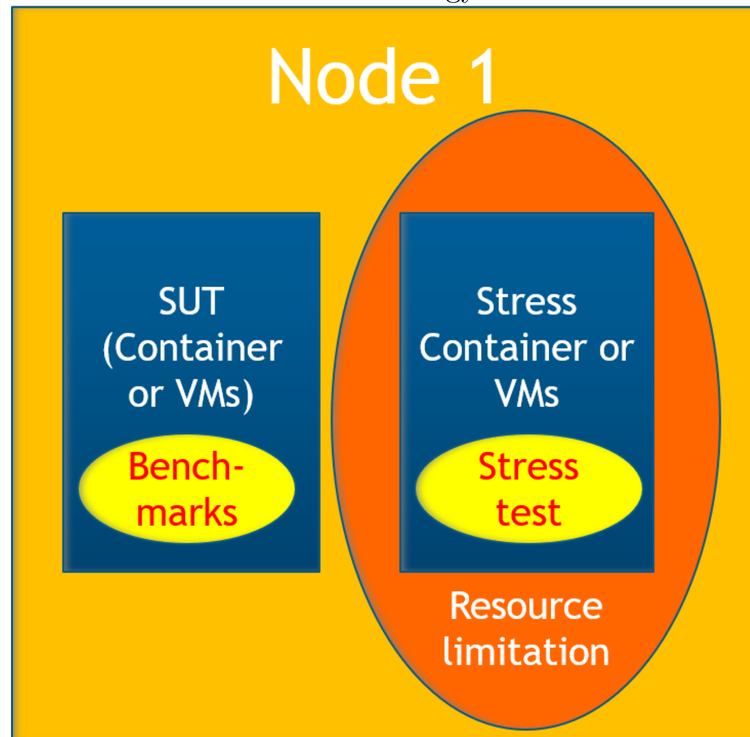

Table 5.2: Resource Reservation Setting for Stress VMs/Containers

Benchmark	CPU	Disk Read	Disk Write	Network Send	Network Receive
SPECweb	bound with 4 CPU cores	272.10 MB/S	275.80 MB/S	5.14 Gbps	5.32 Gbps
TPC-DS	bound with 4 CPU cores	126.23 MB/S	109.10 MB/S	9.71 Gbps/S	9.62 Gbps
Hadoop	bound with 4 CPU cores	57.16 MB/S	58.00 MB/S	9.41 Gbps/S	9.55 Gbps

We designed the first two scenarios based on the total system utilization (benchmark resource usage + stress test VM/container usage) and applied resource reservation into them to build the latter two scenarios. The first two scenarios provide details about how different system utilizations (90% usage and full usage) affect SUTs' performance isolation, and the last two scenarios are designed for testing how good resource protection the resource reservation tools provide under different system utilizations. Table 5.5 shows the details about how we created different system utilization.

Figure 5.2 illustrates how we implemented the resource reservation method. As seen in Table 5.2, we set resource limitation on the stress test VMs/containers to leave enough resource (peak resource usage when corresponding benchmarks running alone) for SUTs. For CPU part, we revised the Cgroups file to bind physical CPU cores for KVM and Docker, whereas used xl command to do the same thing for Xen. For disk part, we modified blkio file in Cgroups to set disk read and write limits for KVM and Docker, and used dm-ioband to control disk I/O in Xen. For network part, we adopted Traffic Controller tct (2003) to classify the network packages from different processes, and set the network limit on the specific class which the stress tests belong to. For the memory and fork parts, we designed separate experiments by increasing the number of memory bomb threads and fork bomb VMs/containers because the resources that these two stress tests consume are difficult to limit directly.

Table 5.3: Average Resource Profile for Isolation Benchmarks

Benchmark	CPU	Disk Read	Disk Write	Network Send	Network Receive
SPECweb	5.86%	9.87 MB/S	8.79 MB/S	1.62 Gbps	1.5699Gbps
TPC-DS	18.99%	51.31 MB/S	60.06 MB/S	57.24 MB/S	88.01 MB/S
Hadoop	26.01%	75.44 MB/S	67.81 MB/S	103.72 MB/S	98.9 MB/S

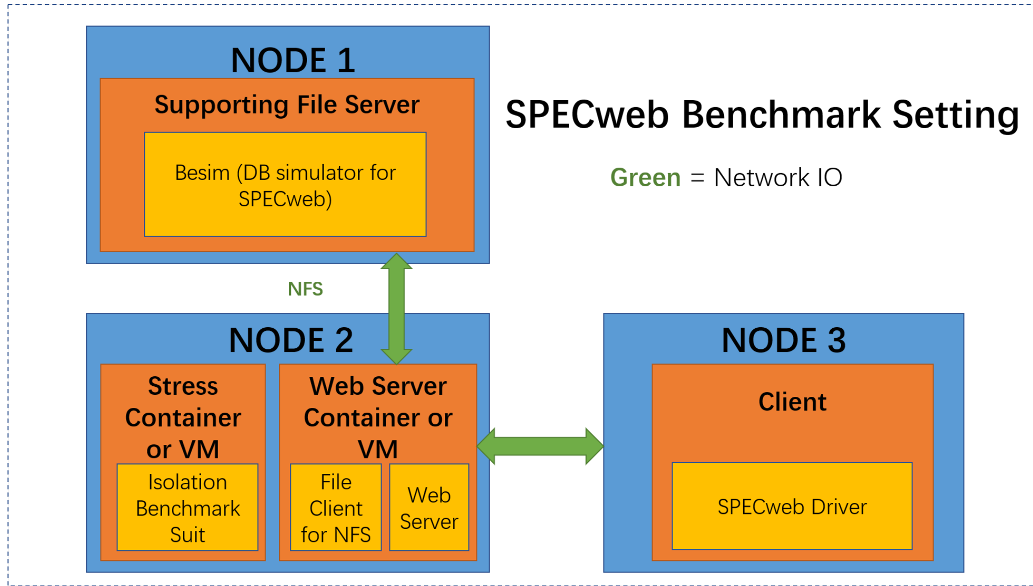

Figure 5.3: SPECweb Benchmark Setting.

In this section, we evaluated different virtualization technologies' isolation by calculating performance degradation. Every result was averaged over 3 stable runs.

- Baseline Result (BR): We recorded the benchmark runtime in SUT when there was no stress test running.
- Results under Stress Tests (RST): We introduced stress tests into containers or VMs individually and recorded the benchmark runtime.
- The SUT degradation results: we calculated the the normalized degradation (degradation $= (BR - RST)/BR)$) to evaluate performance isolation. The lower degradation is, the better performance isolation is.

Table 5.4: Stress Test Suit Description

Stress Test	Description
CPU Bomb	Uninterrupted run interger and floating-point calculations
FIO Bomb	Uninterrupted run 20 threads and each thread runs in random mixed mode (50% read and 50% write) with 32 KB block
Network Bomb	Uninterrupted run 4 threads and each thread sends or reads 60 KB packets to or from external receivers or senders
Memory Bomb	Uninterrupted allocate and touch memory without releasing it
Fork Bomb	Uninterrupted creation of new child processes

We used three virtualization technologies, including Docker, KVM, and Xen. Table 5.1 shows our containers and VMs configurations in different scenarios. We used three real-world benchmarks which intensively use different resources, including SPECweb, TPC-DS, and Hadoop TeraSort. Table 5.3 shows the characteristics of their use of different resources. SPECweb is more network-intensive than TPC-DS and Hadoop TeraSort, whereas Hadoop TeraSort uses more CPU and disk resources.

5.1.1 SPECweb

SPECweb 2005 V1.30 is a web server benchmark developed by the Standard Performance Evaluation Organization (SPEC). It evaluates a web server performance by measuring the maximum amount of concurrent connections that meet specific

Table 5.5: Stress Test Suit Setting

Stress Test	90% Usage Scenario	Full Usage Scenario
CPU Bomb	SPECweb 25 CPU Bombs	SPECweb 28 CPU Bombs
	TPC-DS 21 CPU Bombs	TPC-DS 24 CPU Bombs
	TeraSort 19 CPU Bombs	TeraSort 22 CPU Bombs
FIO Bomb	SPECweb 95 FIO Bombs	SPECweb 120 FIO Bombs
	TPC-DS 80 FIO Bombs	TPC-DS 106 FIO Bombs
	TeraSort 74 FIO Bombs	TeraSort 99 FIO Bombs
Network Bomb	SPECweb 11 Network Bombs	SPECweb 20 Network Bombs
	TPC-DS 13 Network Bombs	TPC-DS 22 Network Bombs
	TeraSort 13 Network Bombs	TeraSort 22 Network Bombs
Memory Bomb	Increase the number of Memory Bomb from 1 to 64	
Fork Bomb	Increase the number of Fork Bomb VM/container from 1 to 32	

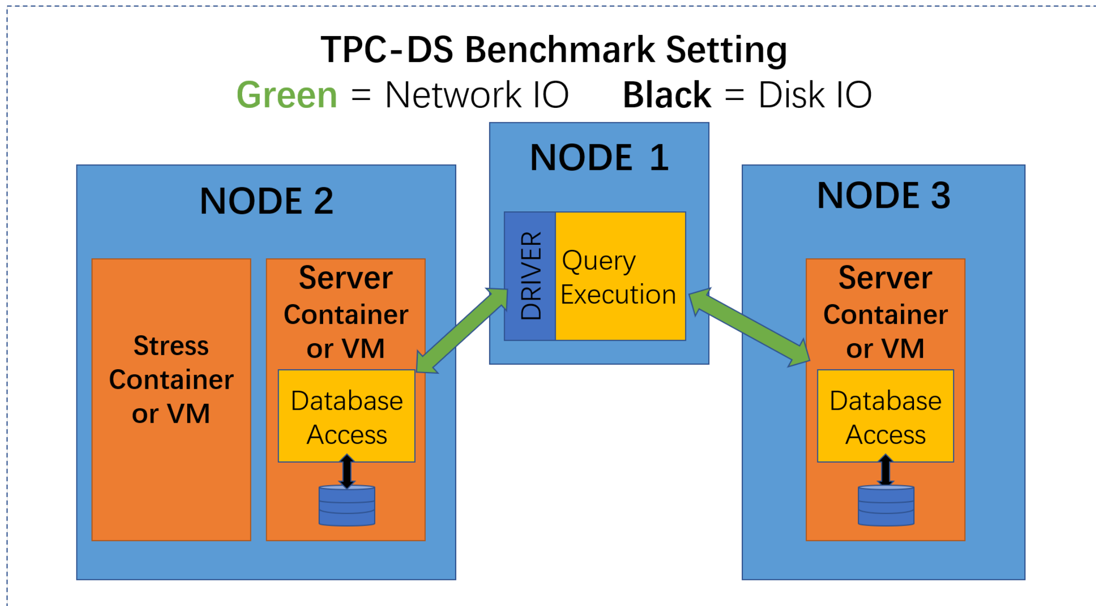

Figure 5.4: TPC-DS Benchmark Setting.

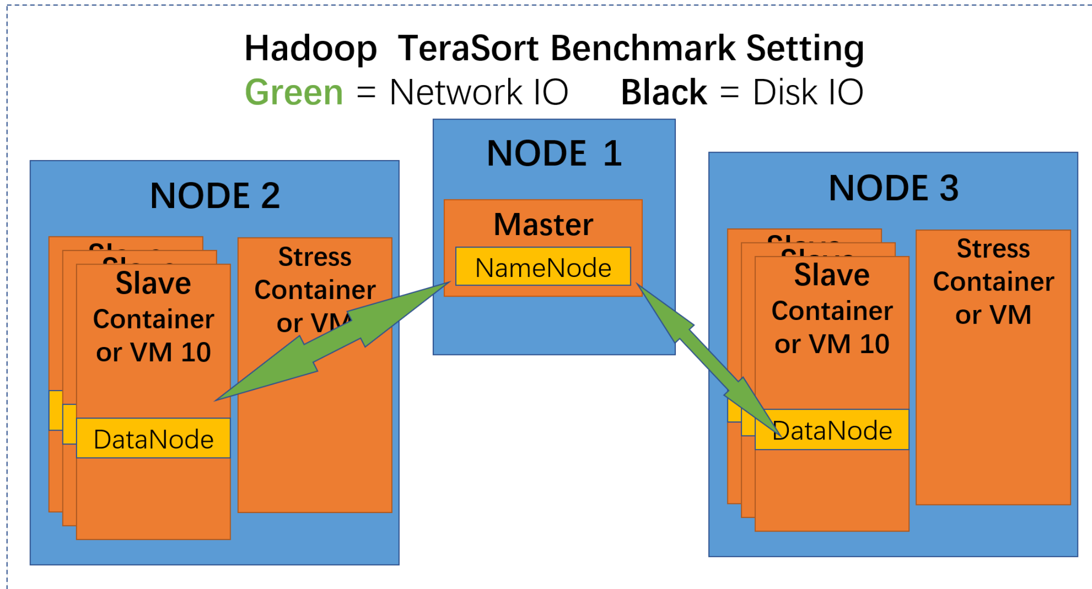

Figure 5.5: TeraSort Benchmark Setting.

throughput, customer request rates, and customer response rates.

Figure 5.3 shows our setting for SPECweb benchmark. SPECweb consists of three parts: the client, the web server, and the back-end simulator (Besim). The client keeps sending HTTP Get requests to the Web server. Those requested web page files range from several kilobytes to several megabytes. The web server stores all the necessary data in the Besim part and accesses the stored information through NFS Arthursson (2012). SPECweb experiments ran with JDK 1.8.0_162 and started every single test from cold cache.

Figure 5.3 shows our SPECweb benchmark setting. SPECweb consists of three parts: client, web server, and Besim.

- Client: The clients work as load drivers for the whole system. In the client system, there is one or more special client that acts as the prime client which manages other clients' activities. Every time before the experiment starts, the prime client starts the user-designed management scripts and establishes a TCP/IP connection with a listening thread on every client. Then the prime client passes

the workflow to the other clients. The clients set up a group of processes which generate workload and send HTTP requests to the web server. After all the processes have finished sending requests and receiving responses, the prime client collects the response information. In our experiments, we set the number of the prime client as one and put the prime client and the other eight clients in one dedicated cluster node.

- Web Server: Before the experiment starts, Web server generates web files in a remote folder that is physically located in the Besim. Then it handles the clients' requests and retrieves information from Besim during the experiment. The web server runs together with the stress test in separate containers/VMs and it has the same resource allocations of SUT that are mentioned in Table 5.1.

- Besim: The Besim simulates the back-end application server which provides the database service to the web server to complete related HTTP requests. The Besim node's memory needs to be sufficiently large to provide at least 34 GB remote folder for the web server. Otherwise, it may affect the performance of the web server. Besim run on a dedicated node.

5.1.2 TPC-DS

The TPC-DS benchmark builds different aspects of a decision support system through various multidimensional data patterns and tests user response time and system maintenance performance through query and data maintenance. TPC-DS has seven fact tables and 15 latitude tables. Each table contains 18 columns on average. The workload used in our experiment contains 99 SQL queries which covers the core parts of SQL 99, SQL 2003, and Online Analytical Processing (OLAP). Moreover,

the TPC-DS benchmark includes complex applications which are consistent with real data such as statistics, report generation, online query, and data mining.

Figure 5.4 shows our experiment setting in which the TPC-DS benchmark consists of two parts: client and server.

- Client: The client contains the driver and the query execution parts. It sends the SQL queries to the server and receives the responses from the server.
- Server: The Server executes the queries from the client and replies to the client. In TPC-DS experiments, the server VM/container inherits the resource allocation scheme of SUT in Table 5.1.

We used Microsoft SQL Server 2017 as our database software for the SUT. We generated TPC-DS table data files with scale factor equals 50GB then loaded them into server SQL database. We ran 99 queries as our workloads and used the elapsed time as our results.

5.1.3 Hadoop TeraSort

Hadoop TeraSort is a divide and conquer model which runs many sub-tasks of a large scale task in parallel on multiple slave nodes under the management of one master. Hadoop TeraSort paradigm consists of two phases: mapping and reducing. The mapping part divides a task into multiple sub-tasks, and the reducing part combines the multiple partial results together to obtain a final solution.

Figure 5.5 shows our Hadoop TeraSort benchmark setting. There are two components in Hadoop TeraSort paradigm. One is the JobTracker for dispatching tasks. The whole Hadoop cluster has only one JobTracker which is on the master machine. The other is the TaskTracker for executing tasks, located on each slave machine. We used 21 VMs/containers for this benchmark where one was used as the master

Table 5.6: CPU Isolation Degradation (%)

Scenarios	SPECweb			TPC-DS			TeraSort		
	KVM	Xen	Docker	KVM	Xen	Docker	KVM	Xen	Docker
90% U	0	0.12	10.1	0.01	1.15	11.27	0.09	1.23	14.53
FU	0.91	3.11	18.1	0.04	3.12	22.9	0.97	3.95	25.13
90%URR	0	0.03	1.2	0	1.21	0.62	0.01	2.09	3.97
FURR	0	1.25	2.1	0	1.71	2.01	0.02	2.97	7.8

machine on one node, and 20 were used as slave machines evenly distributed on the other two nodes. We set a separate node on each slave node for stress tests. For every machine in this experiment, we allocated 1 virtual CPU and 15G virtual Disk. Other specifications follow the resource allocation of SUT in Table 5.1. We used Hadoop 3.0.0 and java 1.8.0_162 to build the experimental environment. We ran 30G workloads and measured the elapsed time to evaluate performance isolation.

5.2 Summary of Results

Our isolation experiments used the stress tests in Table 5.4 to compete resources with the SUT. We tested performance isolation about CPU, disk, and network in four scenarios. According to the different characteristics of the four scenarios, we abbreviate them as 90% usage scenario (90%U), full usage scenario (FU), 90% usage with resource reservation scenario (90%URR), full usage with resource reservation scenario (FURR) in the result table. In the latter two scenarios, we explored how well the different resource reservation methods (e.g., Cgroups, dmioband iob (2009)) helped to improve performance isolation. Based on the resource profile we got for each benchmark, we reserved the average resource each benchmark needed for the SUT. Furthermore, we ran memory and fork experiments through multiple threads scenarios and multiple machines scenarios, respectively.

Table 5.7: Disk Isolation Degradation (%)

Scenarios	SPECweb			TPC-DS			TeraSort		
	KVM	Xen	Docker	KVM	Xen	Docker	KVM	Xen	Docker
90% U	0.07	2.91	17.29	1.01	4.27	46.17	1.02	2.03	73.4
FU	0.27	6.62	48.13	2	8.7	88.12	1.3	8.88	113.8
90%URR	0.01	2.62	1.98	0.15	3.61	8.11	0.17	3.83	21.29
FURR	0.01	4.17	2.06	0.21	6.94	14.58	0.55	7.01	27.9

Table 5.8: Network Transmit Isolation Degradation (%)

Scenarios	SPECweb			TPC-DS			TeraSort		
	KVM	Xen	Docker	KVM	Xen	Docker	KVM	Xen	Docker
90% U	0.13	2.8	34.57	0.05	2.01	21.9	0	0.51	31.19
FU	1.09	6.01	48.91	0.87	5.12	25.29	0.05	5.4	47.12
90%URR	0	0.75	6.12	0	0.81	6.54	0	1.92	6.27
FURR	0	4.24	17.62	0	3.79	8.2	0	5.08	9.43

Table 5.9: Network Receive Isolation Degradation (%)

Scenarios	SPECweb			TPC-DS			TeraSort		
	KVM	Xen	Docker	KVM	Xen	Docker	KVM	Xen	Docker
90% U	0.01	1.54	90.81	0.01	1.92	18.01	0.01	1.01	21.23
FU	0.72	1.99	91.88	0.51	3.92	24.32	0.07	2.08	29.11
90%URR	0.01	0.06	28.19	0	0.75	2.42	0	0.63	5.17
FURR	0.07	1.13	39.21	0	2.39	4.91	0	2.31	8.22

Table 5.10: Memory Isolation Degradation (%)

Scenarios	SPECweb			TPC-DS			TeraSort		
	KVM	Xen	Docker	KVM	Xen	Docker	KVM	Xen	Docker
1 Thread	0	0.01	0.92	0	0.01	0.18	0	0.01	0.55
4 Threads	0.47	1.21	3.77	0.21	0.01	2.31	0.27	0.82	3.42
16 Threads	0.87	4.08	9.92	0.25	3.76	7.28	0.88	3.66	11.21
32 Threads	0.21	4.12	14.43	0.99	4.35	14	1.01	4.64	15.81
64 Threads	1.43	6.57	19.18	1.14	4.98	20.98	1.56	5.43	21.14

Table 5.11: Fork Isolation Degradation (%)

Scenarios	SPECweb			TPC-DS			TeraSort		
	KVM	Xen	Docker	KVM	Xen	Docker	KVM	Xen	Docker
1 Machine	0	0	0.15	0	0.01	0.02	0	0	0.21
4 Machines	0	0	0.18	0.01	0.01	0.12	0	0	0.43
16 Machines	0	0	1	0.01	0.42	0.99	0.01	0.02	2.51
32 Machines	0	0.04	2.03	0.01	1.89	6.98	0.02	3.03	8.15
64 Machines	0.51	7.53	DNR	0.01	5.46	DNR	0.02	8.03	DNR

5.2.1 CPU Isolation

The IBS Matthews *et al.* (2007) adds burden to CPU by running a tight loop of integer arithmetic operations. We verified that the CPU stress test enables the corresponding container or VM to occupy 100% of assigned vCPU. In the resource reservation scenarios, we physically bound CPU cores with corresponding SUT container/VM by using Cgroups and XL (for Xen).

Table 5.6 shows performance isolation of KVM, Xen, and Docker. In 90%U scenario, KVM's CPU isolation keeps the leading places with up to 0.09% degradation in all benchmarks. Xen shows 1.14% performance loss compared with KVM with TPC-DS, and Docker shows 13.3% more overhead than Xen under TeraSort. In FU scenarios, Xen shows 3.08% worse performance than KVM at FU scenario with TPC-

DS. Docker has the worst CPU isolation (up to 25.13%) among the three virtualization technologies in all benchmarks.

Resource reservation tools help the three virtualization technologies achieve better CPU isolation, especially for Docker. Docker benefits 8.9%, 10.65%, and 10.56% better performance from Cgroups at 90% resource usage under SPECweb, TPC-DS, and TeraSort, respectively. Moreover, Docker also reduces 16%, 20.89%, and 17.33% performance overhead at FURR scenario compared with FU scenario. Cgroups helps Docker achieve better CPU isolation by setting processor affinity. After binding processes to specific CPU cores, the CPU scheduler ran the specified process on the "bound" CPU based on the CPU affinity setting which means that processes are usually not migrated frequently between processors and they have much fewer chances to affect the other processes.

5.2.2 Disk I/O Isolation

We used the multiple FIO bombs to consume disk bandwidth resource continuously for performance isolation analysis. For one FIO bombs, we ran 20 threads together and each thread ran mixed random accessing (50% read and 50% write) with 32 KB sized block. In the resource reservation scenarios, we used dmioband iob (2009) to set resource limitation for Xen, whereas revised the Cgroups files to limit the disk bandwidth for KVM and Docker.

Table 5.7 shows the disk isolation degradation of KVM, Xen, and Docker in different scenarios. In 90%U scenario, KVM shows up to 3.26% better performance than Xen with TPC-DS, and 71.37% less performance loss than Docker with TeraSort. In FU scenario, we noticed that Xen shows up to 7.58% worse performance isolation than KVM, but up to 104.92% better performance isolation than Docker with TeraSort.

Resource reservation tools help KVM, Xen, and Docker in various degree. Xen generally gets slight improvement (less than 2.45%) in performance with dm-ioband. However, Cgroups helps Docker regain 15.31%, 38.06%, 52.11% performance in SPECweb, TPC-DS, and TeraSort in 90%U scenario, and it also helps Docker reduce 46.07%, 73.54%, and 85.9% overhead when running those three benchmarks respectively in the FU scenario. Through Cgroups helps Docker achieve much better performance, it cannot protect Docker from all influence.

5.2.3 Network Isolation

We used the IBS Matthews *et al.* (2007) network stress test to evaluate network isolation in terms of transmit mode and receive mode. The network stress test introduces the intensive network I/O to the system through the communication between the server and the client. We used the server/client as our SUT and put the client/server on another cluster node. The stress machine starts 4 threads together and each thread constantly sends or receives packages of size 60 KB through UDP to or from external receiver or sender. In resource reservation scenarios, We used Linux Traffic Control tct (2003) to limit the total network bandwidth for stress tests. The Linux Traffic Control establishes priority queues for processing packets and defines how the packets in the specific queue are sent, thereby enabling control of traffic and reducing interference between processes.

Table 5.8 and Table 5.9 show the network isolation for KVM, Xen, and Docker. In 90%U scenario, KVM shows up to 2.67% better performance than Xen in SPECweb transmit mode, and gains up to 90.8% less overhead than Docker in SPECweb receive mode. In FU scenario, Xen shows up to 5.35% less performance than KVM in TeraSort transmit mode, and has 89.89% better performance than Docker in SPECweb receive mode. To briefly sum up, KVM shows the best network isolation among

the three in both transmit and receive mode which benefits from KVM's network interface.

Traffic Controller helps Docker get better network isolation in both network transmit and receive mode. For network transmit mode, Docker regains up to 28.45% performance in SPECweb under 90%URR scenario, and 37.69% performance in TeraSort under FURR scenario. Furthermore, for network receive mode, Docker achieves 62.62% better performance in SPECweb under 90%URR scenario, and 52% better performance in SPECweb under FURR scenario.

5.2.4 Memory Isolation

For the memory part, the IBS Matthews *et al.* (2007) exhaustively consumes memory by repeatedly allocating and accessing memory. We used multiple threads scenarios (up to 64 threads) to simulate the environment of different system utilizations.

Table 5.10 shows the memory isolation performance for KVM, Xen, and Docker. KVM offers the best memory isolation performance among the three virtualization technologies with up to 1.43% degradation. Xen has up to 5.14% worse performance than KVM when the number of threads reaches 64. Docker shows the highest degradation in Hadoop TeraSort experiment with 64 threads (21.14%). Therefore, Docker is less suitable to be the host of memory-intensive applications than hypervisor-based virtualizations.

5.2.5 Fork Isolation

We used the IBS's Matthews *et al.* (2007) fork bomb to compete for system resources with SUT by repeatedly invoking system call to create new child processes. Moreover, we designed experiments of multiple stress containers/VMs to burden the

target SUT. The stress VM/container number is from 1 to 32 and each VM/container owns only one CPU core which guarantees that the CPU resource is enough for the SUT.

As we can see in Table 5.10, KVM is the most promising one in fork experiments with up to 0.51% degradation. Xen is in the second place with up to 8.03% performance degradation. Docker fails to response for the 64 threads experiments.

Chapter 6

SCALABILITY STUDY

6.1 Methodology

Scalability is the ability of a process, resource, or application to grow and manage the increased demands of its users. Good scalability of virtualization technologies guarantees high performance in the process of cloud services' expansion and growth.

We adopted two ways to evaluate scalability performance for KVM, Xen, and Docker. In the first case, we scaled the system by adding more benchmark threads to one existing VM/container. In the second case, we scaled the system by adding more VMs/containers into one ASU cloud node with one benchmark running in each VM/container. We chose Apache compilation as the test benchmark, and related software is Apache HTTP Server 2.4. Table 6.1 shows our VMs' and containers' configuration in the following experiments. In this experiment, we used the runtime as our results. The shorter the runtime is, the better the scalability is. Every result in this section is averaged over 3 stable results.

Table 6.1: Containers and VMs Configuration for Scalability Study

Scenarios	VCPUs	Virtual Memory	Virtual Disk
Increasing the number of threads	4	10 GB	100 GB
Increasing the number of VMs/containers	1	1 GB	15 GB

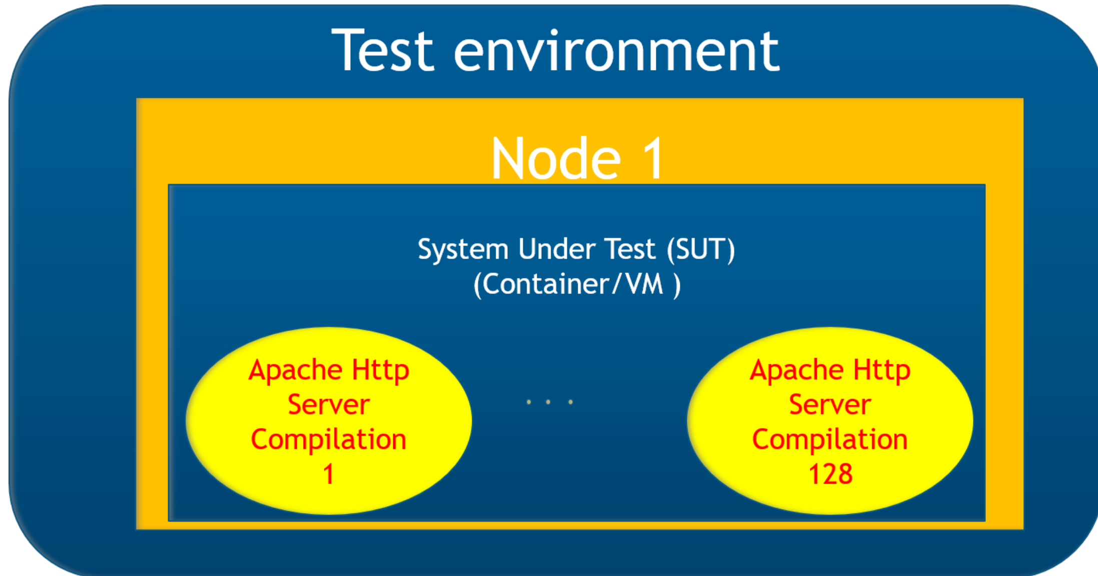

Figure 6.1: Setting of Increasing the Number of Threads

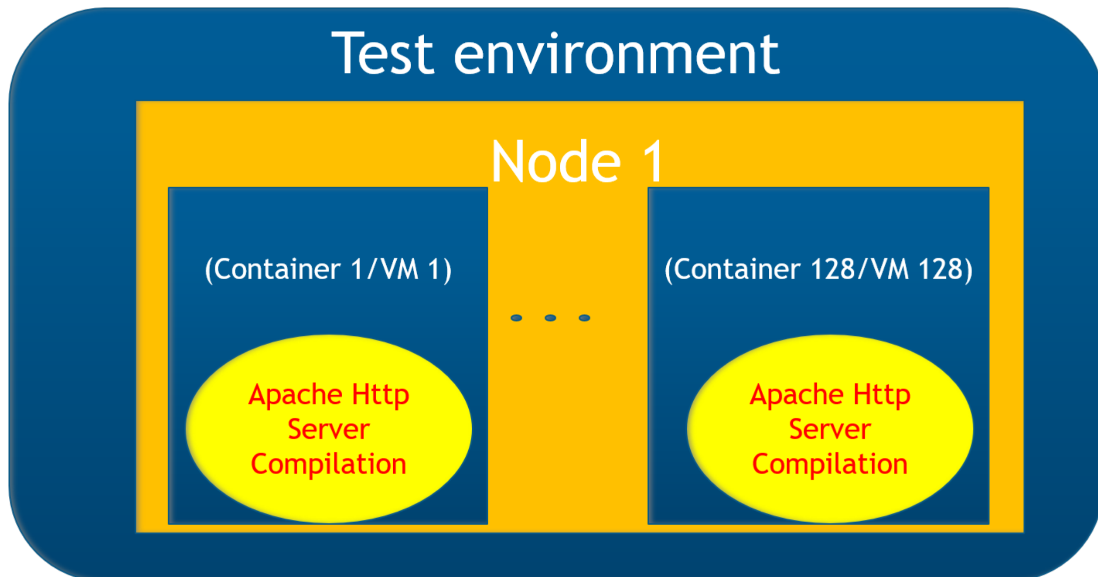

Figure 6.2: Setting of Increasing the Number of VMs/Containers

6.1.1 Increasing the Number of Threads

This experiment is to run multiple threads instead of a single thread to increase the utilization of system resource. Figure 6.1 shows our experiment setting. We increased the number of threads up to 128, and each thread ran an instance of Apache HTTP Server compilation. All the Apache HTTP Server Compilation started at the same

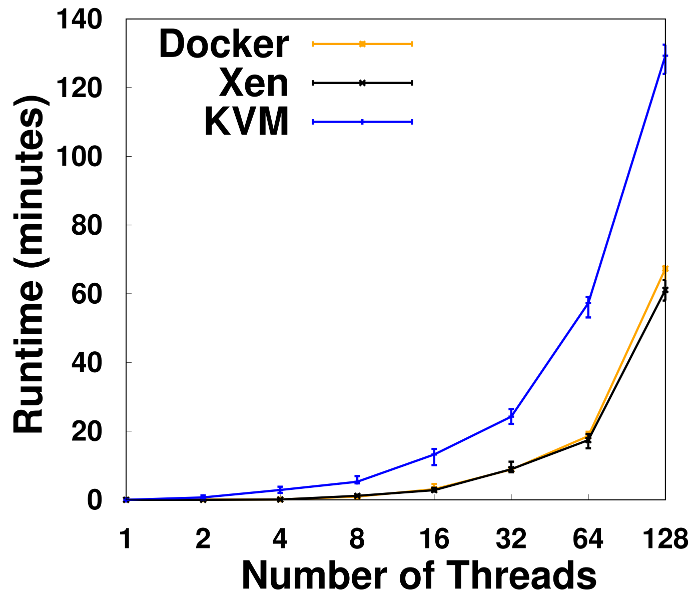

Figure 6.3: Performance of Increasing the Number of Threads

time.

Figure 6.3 shows the average runtime of all VMs/container when increasing the number of threads in one container/VM. KVM shows the worst scalability performance among the three different virtualization technologies when increasing the number of benchmark threads. Xen shows the best scalability performance with 6.1 minutes and 68.2 minutes shorter runtime than Docker and KVM when 128 threads are running. Docker shows the most stable scalability with up to 2.18 minutes and 5.28 minutes smaller fluctuation than Xen and KVM.

6.1.2 Increasing the Number of VMs/Containers

This experiment replicates our single SUT and runs multiple SUTs in one ASU cloud node to increase resource utilization. In the ideal environment with unlimited resources, the runtime would not be affected as the number of VMs/containers increasing. Figure 6.2 shows our experiment setting. We increased the number of VMs/containers up to 128, and each of them ran an Apache HTTP Server compila-

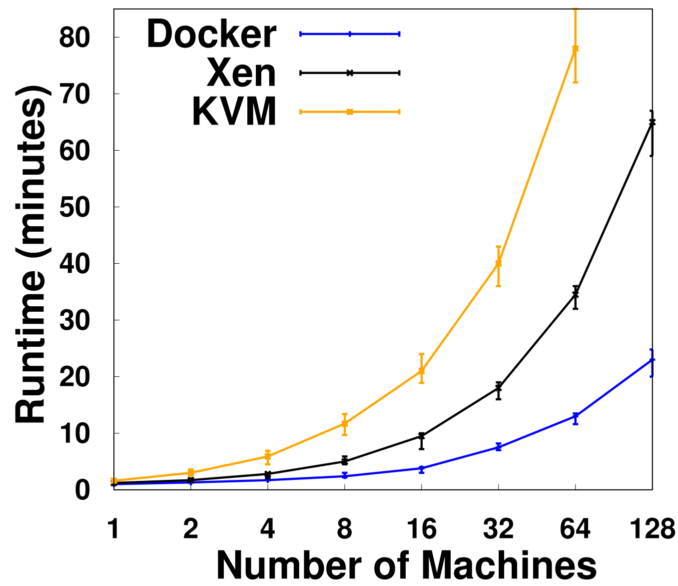

Figure 6.4: Performance of Increasing the Number of VMs/Containers

tion thread.

Figure 6.4 shows the average runtime of all VMs/containers when increasing the number of test containers/VMs. KVM shows the worst scalability performance with no response in 128 VMs case. Docker shows the best scalability with up to 42.37 minutes shorter runtime than Xen when the number of containers/VMs grows to 128.

Chapter 7

RELATED WORK

7.1 Performance Study

Emerging virtualization technologies have brought a lot of new energies and flexibility to cloud computing. However, the performance overhead is still an issue which is widely discussed by the literature. Soltesz *et al.* (2007) presented the performance comparison between VServer and Xen, focusing on disk bandwidth, network communication, CPU usage, and memory access. This paper also concluded that OS-level virtualization (e.g., VServer) can achieve two times better performance than hypervisor-based virtualizations (e.g., Xen).

Felter *et al.* (2015) explored the performance trade-off of LXC, KVM, and Docker. It identified the primary impact of these virtualizations for HPC and server workloads. Specifically, this paper pointed out both KVM and Docker should be used carefully in I/O intensive workloads.

Morabito *et al.* (2015) evaluated the performance difference between hypervisors with lightweight virtualization. It revealed the KVM performance bottleneck in disk I/O for different types of applications.

Compared with related works, our paper studied performance loss for full virtualization, para-virtualization, and OS-level virtualization together by using five performance metrics (e.g., network throughput, network latency, disk, CPU, and memory). We built experimental environments on ASU cloud which provide 10 Gbps network and 10 GB memory for every SUT than related works, which ensured the resource availability cannot be the performance bottleneck. Our results are consistent with

the related works, which supports the universality of performance comparison results of different virtualization technologies.

7.2 Scalability Study

New lightweight virtualizations are known for their scalability benefits which can help them grow and manage the increased demands of their users. Many recent papers evaluated the scalability performance of different virtualization technologies. Soriga and Barbulescu (2013) studied the scalability of hypervisor-based virtualizations, and found that Xen has a small advantage than KVM.

Li and Xia (2016) studied Docker's scalability by adopting container technology to deploy web applications on hybrid cloud. They demonstrated that Docker provides great scalability guarantee for the web application when increasing the number of containers.

Quétier *et al.* (2007) evaluated the scalability performance of Vserver, Xen, UML, and VMware on different scalability metrics (e.g, overhead, linearity). This paper found that VMware provides bad scalability with high overhead and poor linearity.

Though related literature generally believe lightweight virtulization shows good scalability, experiments about increasing number of benchmark threads were not addressed. Those ignored experiments are the important basis to verify whether virtualization technologies can manage the growing user demand or not. Our scalability experiments evaluated hypervisor-based and OS-level virtualization by increasing the number of VMs/containers and benchmark threads.

7.3 Isolation Study

Although there are great advantages of various virtualizations, we cannot ignore their pitfalls in performance isolation. Cloud system takes advantage of resource

sharing by leveraging virtualization technologies. Though this approach saves cloud management cost, it also brings inevitable interference issues. At present, many scholars studied these problems.

Pu *et al.* (2010) explored virtual machines' isolation problems of I/O intensive applications. This article showed there are performance losses when running benchmarks on isolated environments due to the competition of resources.

Matthews *et al.* (2007) compared the performance isolation between VMware, Xen, Solaris containers, and OpenVZ through SPECweb. This paper found that container-based virtualization shows good CPU isolation, whereas suffering from poor I/O and memory isolation.

Xavier *et al.* (2015) evaluated performance isolation by using TPC-C on LXC and KVM. This paper revealed LXC suffers much more interference than KVM when running disk I/O intensive workloads.

Mardan and Kono (2016) evaluated performance isolation of the LXC and KVM through proportional weight policy. It used Cgroups to maintain the resource usage rate between the SUT and stress VM/container. This paper found that KVM outperforms LXC in I/O isolation in DBMS.

Although the related works Pu *et al.* (2010), Xavier *et al.* (2015), Matthews *et al.* (2007) claimed that their SUTs were under stress during the experiments, the details of system utilization were not addressed. If the system utilization is low, the system cannot provide competitive resource environment for its SUT. Our isolation experiments provide 90% and 100% system utilization scenarios for different resources (e.g., CPU, memory, disk, and network) by adjusting stress tests' parameters (e.g., the number of threads). Apart from that, the isolation experiments from Pu *et al.* (2010), Xavier *et al.* (2015), and Mardan and Kono (2016) got the result from single SUT. We used Hadoop TeraSort to test mutiple SUTs at the same time under dis-

tributed systems. Moreover, we evaluated the resource reservation tool's efficiency in isolation experiments which was not thoroughly discussed by related works.

Chapter 8

CONCLUSIONS

Although KVM, Xen, and Docker are excellent virtualizations for deploying cloud services, they provide various benefits and a few drawbacks. Understanding their performance losses in different scenarios can help us better manage them in cloud computing. We evaluated full virtualization, para-virtualization, and OS-level virtualization from three aspects, including performance, isolation, and scalability study.

For performance study, through measuring and analyzing KVM, Xen, and Docker with LINPACK, STREAM, Netperf, and FIO, we found that Docker has the best performance and Xen follows Docker with a slight overhead in most experiments, whereas KVM shows much worse performance than Docker and Xen. The results indicate the hypervisor-based virtualization brings more overheads than OS-level virtualization due to its extra layers.

For scalability study, we evaluated the performance of KVM, Xen, and Docker through Apache HTTP Server compilation. Docker excels in experiments of increasing the number of containers, and Xen works well in experiments of increasing the number of compilation threads in one VM. KVM shows the worst performance among the three in both experiments.

For isolation study, we designed four scenarios to evaluate performance isolation of different virtualization technologies under 90% and full system utilization. Unlike KVM, Docker shows bad performance isolation in most experiments. We also found important disk and network isolation problems under full system utilization. Furthermore, we evaluated the resource reservation tools' efficiency under different system utilizations. By compare the performance degradation before and after using the re-

source reservation tools, we found that they can help virtualization technologies, but cannot help them avoid all influences.

REFERENCES

"Vmware", https://www.vmware.com/ (1998).

"The linux traffic control", http://tldp.org/howto/Traffic-Control-howto/intro.html (2003).

"Amazon elastic compute cloud", https://aws.amazon.com/ec2/ (2006a).

"Amazon elastic container service", https://aws.amazon.com/ecs/ (2006b).

"Amazon web services", https://aws.amazon.com/ (2006c).

"The dm-ioband bandwidth controller", https://sourceforge.net/apps/trac/ioband (2009).

Arthursson, D., "Network file system", US Patent 8,156,146 (2012).

Bellard, F., "Qemu, a fast and portable dynamic translator.", in "USENIX Annual Technical Conference, FREENIX Track", vol. 41, p. 46 (2005).

Che, J., C. Shi, Y. Yu and W. Lin, "A synthetical performance evaluation of openvz, xen and kvm", in "Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific", pp. 587–594 (IEEE, 2010).

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

Fink, J., "Docker: a software as a service, operating system-level virtualization framework", Code4Lib Journal pp. 25, 29 (2014).

Kivity Qumranet, A., Y. Kamay Qumranet, D. Laor Qumranet, U. Lublin Qumranet and A. Liguori, "Kvm: The linux virtual machine monitor", Proceedings Linux Symposium p. 15 (2007).

Krishnan, S. and J. L. U. Gonzalez, "Google compute engine", in "Building your next big thing with Google cloud platform", pp. 53–81 (Springer, 2015).

Li, Y. and Y. Xia, "Auto-scaling web applications in hybrid cloud based on docker", in "2016 5th International Conference on Computer Science and Network Technology (ICCSNT)", pp. 75–79 (IEEE, 2016).

Mardan, A. A. A. and K. Kono, "Containers or hypervisors: Which is better for database consolidation?", in "Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on", pp. 564–571 (IEEE, 2016).

Matthews, J. N., W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe and J. Owens, "Quantifying the performance isolation properties of virtualization systems", in "Proceedings of the 2007 workshop on Experimental computer science", p. 6 (ACM, 2007).

Morabito, R., J. Kjällman and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison", in "2015 IEEE International Conference on Cloud Engineering", pp. 386–393 (IEEE, 2015).

Pu, X., L. Liu, Y. Mei, S. Sivathanu, Y. Koh and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments", in "2010 IEEE 3rd International Conference on Cloud Computing", pp. 51–58 (IEEE, 2010).

Quétier, B., V. Neri and F. Cappello, "Scalability comparison of four host virtualization tools", Journal of Grid Computing **5**, 1, 83–98 (2007).

Soltesz, S., H. Pötzl, M. E. Fiuczynski, A. Bavier and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors", in "ACM SIGOPS Operating Systems Review", vol. 41, pp. 275–287 (ACM, 2007).

Soriga, S. G. and M. Barbulescu, "A comparison of the performance and scalability of xen and kvm hypervisors", in "2013 RoEduNet International Conference 12th Edition: Networking in Education and Research", pp. 1–6 (IEEE, 2013).

Xavier, M. G., I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi and C. A. De Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds", in "2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing", pp. 253–260 (IEEE, 2015).

APPENDIX A

COMMAND LINES FOR PERFORMANCE STUDY

Table A.1: Command Lines of Benchmarks (LINPACK, STREAM, Netperf) Used in Performance Study

Benchmark	Measured Operation	Command Lines
LINPACK	Numerical linear algebra	./linpack/benchmarks/linpack/xlinpack_xeon64
STREAM	Copy, add, Scale, Traid	cd stream gcc -O stream.c -o stream ./stream
Netperf	TCP RR latency	netperf -P 0 -t TCP_RR -H 10.107.30.25 – -r 1,1 -o P90_LATENCY
	UDP RR latency	netperf -P 0 -t UDP_RR -H 10.107.30.25 – -r 1,1 -o P90_LATENCY
	TCP send throughput	netperf -H 10.107.30.25 -l 60
	TCP receive throughput	netserver -D -4 -L 10.107.30.25 -p 9991
	UDP send throughput	netperf -t UDP_STREAM -H 10.107.30.25 -l 60
	UDP receive throughput	netserver -D -4 -L 10.107.30.25 -p 9991

Table A.2: Command Lines of Benchmarks (FIO) Used in Performance Study

Benchmark	Measured Operation	Command Lines
FIO	Random read	fio -filename=/dev/sda4 -direct=1 -ioengine=psync -rw=randread -bs=32k -numjobs=20 -size =2
	Random write	fio -filename=/dev/sda4 -direct=1 -ioengine=psync -rw=randwrite -bs=32k -numjobs=20 -size =2
	Random mix	fio -filename=/dev/sda4 -direct=1 -ioengine=psync -rw=randrw -bs=32k -numjobs=20 -size =2
	Seq read	fio -filename=/dev/sda4 -direct=1 -ioengine=psync -rw=read -bs=32k -numjobs=20 -size =2
	Seq write	fio -filename=/dev/sda4 -direct=1 -ioengine=psync -rw=write -bs=32k -numjobs=20 -size =2

APPENDIX B

COMMAND LINES FOR ISOLATION STUDY

Table B.1: Command Lines of SPECweb Used in Isolation Study

Benchmark	Command Lines
SPECweb	Besim: perl test_besim_bank.pl http://"BESIM_HOST"/ fcgi-bin/besim_fcgi.fcgi; Web server: service httpd restart; Client: java -Xms512m -Xmx512m -jar specwebclient.jar; Prime Client: java -Xms512m -Xmx512m -jar specweb.jar;

Table B.2: Command Lines of TPC-DS Used in Isolation Study

Benchmark	Command Lines
TPC-DS	Server: ./dsdgen -DIR /part2/tpcds/v2.6.0/datas/ -SCALE 30; copy "TABLE_NAME" from '/part2/tpcds/v2.6.0/datas/handled /"TABLE_NAME".dat' with delimiter as '|' NULL "; "TABLE_NAME" contains call_center, catalog_page, catalog_returns, catalog_sales, customer, customer_address, customer_demographics, date_dim, dbgen_version, household_demographics, income_band, inventory, item, promotion, reason, ship_mode, store, store_returns, store_sales, time_dim, warehouse, web_page, web_returns, web_sales, web_site; Client: ./99_query.sh (run query 1-99);

Table B.3: Command Lines of Hadoop Used in Isolation Study

Benchmark	Command Lines
Hadoop TeraSort	Master: /hadoop/hadoop-3.2.0/hadoop-3.2.0/bin/hadoop jar hadoop-3.2.0-examples.jar teragen 300000000 terasort/30G-input; bin/hadoop jar hadoop-3.2.0-examples.jar terasort 30G-input 30G-output;

Table B.4: Command Lines of Stress Tests Used in Isolation Study

Benchmark	Command Lines
CPU Bomb	cd stress_suit/cpu; make; ./cpu_bomb;
Memory Bomb	cd stress_suit/memory; make; ./memory_bomb;
Fork Bomb	cd stress_suit/fork; ./fork_bomb;
FIO Bomb	cd stress_suit/fio; ./fio_bomb;(loop FIO random mix test)
Network Bomb	Network server: cd stress_suit/network make ./stress.sh x.txt INFINITY perludp-multithread Network receiver: cd stress_suit/network make ./stress.sh no perludp-multithread

Table B.5: Command Lines of CPU Reservation Used in Isolation Study

Resource	Reservation Tools	Command line
CPU	XL	Xen: xl vcpu-set dom103- 0vcpu 0pcpu; xl vcpu-set dom103- 1vcpu 1pcpu; xl vcpu-set dom103- 2vcpu 2pcpu; xl vcpu-set dom103- 3vcpu 3pcpu;
	Cgroups	Docker: docker run -itd –name docker_stress –cpuset-cpus 0-4 Ubuntu; KVM: virsh kvm_stress 0vcpu 0pcpu; virsh kvm_stress 1vcpu 1pcpu; virsh kvm_stress 2vcpu 2pcpu; virsh kvm_stress 3vcpu 3pcpu;

Table B.6: Command Lines of Disk Reservation Used in Isolation Study

Resource	Reservation Tools	Command line
Disk	Dm_ioband	Xen: SPECweb: dmsetup message dom103 0 io_limit 286317530; TPC-DS: dmsetup message dom103 0 io_limit 114399642; TeraSort: dmsetup message dom103 0 io_limit 59936604;
	Cgroups:	Docker and KVM: SPECweb: echo “8:0 286317530” > /sys/fs/cgroup/blkio/mygroup/blkio.throttle.read_bps_device; echo “8:0 289197261 ” > /sys/fs/cgroup/blkio/mygroup/blkio.throttle.write_bps_device; TPC-DS: echo “8:0 132466606” > /sys/fs/cgroup/blkio/mygroup/blkio.throttle.read_bps_device; echo “8:0 114399642 ” > /sys/fs/cgroup/blkio/mygroup/blkio.throttle.write_bps_device; TeraSort: echo “8:0 59936604” > /sys/fs/cgroup/blkio/mygroup/blkio.throttle.read_bps_device; echo “8:0 60817408 ” > /sys/fs/cgroup/blkio/mygroup/blkio.throttle.write_bps_device;

Table B.7: Command Lines of Network Reservation Used in Isolation Study

Resource	Reservation Tools	Command line
Network	Traffic Controller	tc qdisc add dev eth0 root handle 1: htb stressed_package 1; SPECweb: tc class add dev eth0 parent 1: classid 1:1 htb rate 5140mbps burst 0k; tc class add dev eth0 parent 1:1 classid 1:10 htb rate 5140mbps ceil 5140mbps burst 0k; TPC-DS: tc class add dev eth0 parent 1: classid 1:1 htb rate 9620mbps burst 0k; tc class add dev eth0 parent 1:1 classid 1:10 htb rate 9620mbps ceil 9620mbps burst 0k; TeraSort: tc class add dev eth0 parent 1: classid 1:1 htb rate 9410mbps burst 0k; tc class add dev eth0 parent 1:1 classid 1:10 htb rate 9410mbps ceil 9410mbps burst 0k;

APPENDIX C

COMMAND LINES FOR SCALABILITY STUDY

Table C.1: Command Lines of Apache HTTP Server Compilation Used in Scalability Study

Benchmark	Command line
Apache HTTP Server compilation	cd httpd-NN; ./configure –prefix=PREFIX; make; make install;