Comparative evaluation of Virtual Environments:

Virtual Machines and Containers

by

Ranjan Dhar

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

August 2016

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Ranjan Dhar

August 29, 2016

Permission to Lend and/or Copy

Acknowledgments

I would like to express my sincere gratitude to my supervisor Prof. Stefan Weber for the continuous support throughout my time as his student, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me during the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my research work.

Last but not the least, I would like to thank my parents and friends for supporting me throughout my life.

Ranjan Dhar

University of Dublin, Trinity College

August 2016

Comparative evaluation of Virtual Environments:

Virtual Machines and Containers

Ranjan Dhar
University of Dublin, Trinity College, 2016

Supervisor: Stefan Weber

Cloud computing is a major component of the IT industry. Current cloud computing solutions make extensive use of virtual machines because they offer a high degree of isolation as well as an opportunity to optimize for effective utilization of available infrastructure. However, virtual machines also rely on a degree of abstraction which may result in performance degradation thereby affecting the users or customers. New advancement in container-based virtualization techniques simplifies deployment and offers similar opportunities in the control and optimization of resources.

The availability of these two alternative technologies has resulted in a number contrasting opinions and studies. However, the advantages and disadvantages of each technology when compared against its alternative are not clear, which leads to difficulties for infrastructure providers who aim to optimize the use of their resources and the service provided to their customers.

In this study, we explore the performance profile of traditional virtual machine deployments and contrast it with containers. We use a set of workloads that stresses CPU, File I/O and MySQL server to evaluate the performance while scaling up the deployments incrementally. We also evaluate the migration performance of the two on the same network. We use KVM as hypervisor for virtual machines and Docker as container manager. Our results point out that containers perform better in terms of both density and boot latency; however, the results are reversed in terms of migration performance due to the maturity of migration mechanisms for virtual machines which result in comparatively low service downtime. We also discuss the potential performance inhibiting factors as well as future optimizations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing often referred as “Cloud”, is the delivery of on demand, scalable, elastic and location independent computing resources —be it an application or a whole datacenter over an Internet on a pay –for –use basis. Based on the services offered, the cloud is usually classified into following models:

1. Software as a Service (SAAS): Cloud based applications are run on distant computers “in the cloud” that is either a third party infrastructure or a personally owned one. The users usually connect to the application via Internet, usually a web browser (HTTP based)

2. Platform as a Service (PAAS): It provides a cloud based environment with all computing resources available within the ecosystem to support the complete lifecycle of building and delivering cloud application without bothering about the cost and complexity of buying and managing the hardware, software, provisioning and hosting.

3. Infrastructure as a Service (IAAS): It provides companies/organizations with the computing resources ranging from servers to networking to storage.

Besides these three fundamental models, cloud service taxonomy includes two additional services termed as:

i) Database as a service (DBaaS)

ii) Storage as a service (SaaS)

Figure 1.1 describes the hierarchical taxonomy of a traditional cloud service.

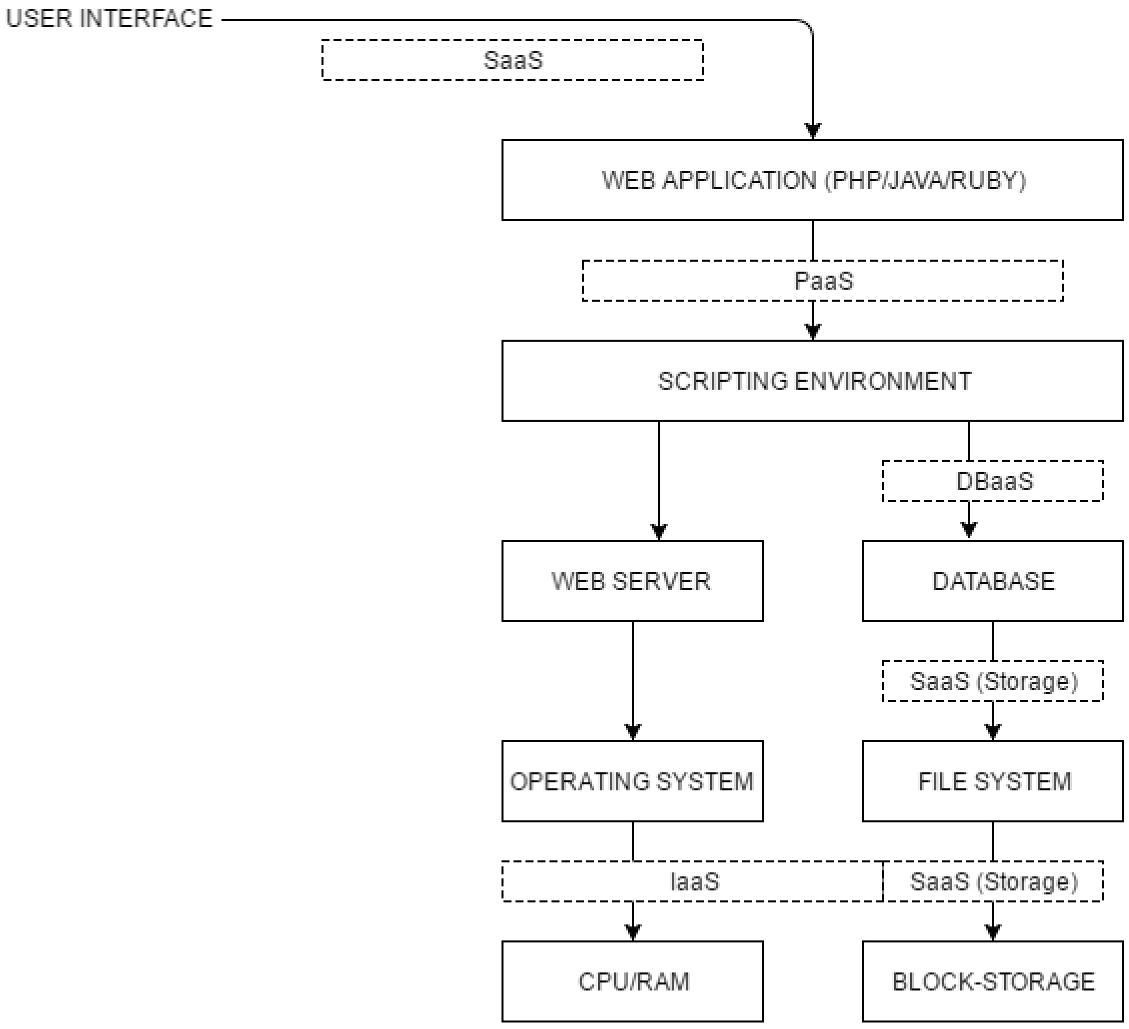


Figure 1.1: Cloud Service Taxonomy.

The deployment model for cloud computing are as under:

i) Public cloud: Owned, operated and managed by organizations that offer rapid access over a public network towards the vision of affordable computing resource. By using public cloud services, the users do not need to invest in hardware, software or supporting infrastructure, as the same is owned and maintained by a third party cloud provider.

ii) Private cloud: The infrastructure in a private cloud is operated specifically for a single organization whether by deploying an in-house data center or outsourcing to a third party. Private cloud infrastructure can be tuned to maximum efficiency with more refined control of resources and avoiding any multi-tenancy isolation issues.

iii) Hybrid cloud: Uses a private cloud infrastructure combined with the strategic integration and use of public cloud service. Although the public and private segments of hybrid cloud are bound together, they remain unique entities. This allows it to offer the benefits of multiple deployment models at once.

Data centers being the core of cloud technologies, their importance in terms of management and evolution can not be ignored. The topology of a data center plays an important role in determining the service level agreements (SLAs) from a data center service provider. Todays data centers follow a 3-layer topology. It comprises a core of data center switches that connect to each other and to the external network providers, users or access layer and the aggregation layer between the two moves information primarily from north to south.

With the advancement in technologies and introduction of some new optimized algorithms, leaf-spine network topology is an upcoming popular model that experiences more east west network traffic because most of the traffic is internal due to the use of virtualization. This topology adds the spine-layer with more switches to handle traffic with the data center, such as storage area network data traffic. Figure 1.2 and 1.3 differentiates the architectural difference between the two topologies. There are other architectures that govern the networking in data centers like Clos topology, fat tree or the 4 post architecture.

Cloud computing has become increasingly important because it makes computing economical and flexible. Today, almost every enterprise runs their applications in data centers which provide them with scalable and on demand computational and storage resources. Cloud computing provides a new avenue for both small and large enterprises to host their applications by providing features like on demand provisioning and payment based on metered usage. The cost of licensing software and use of hardware is reduced considerably as users pay a monthly or hourly fee to use services which include hardware, software and maintenance cost. Considering the mammoth infrastructure involved in

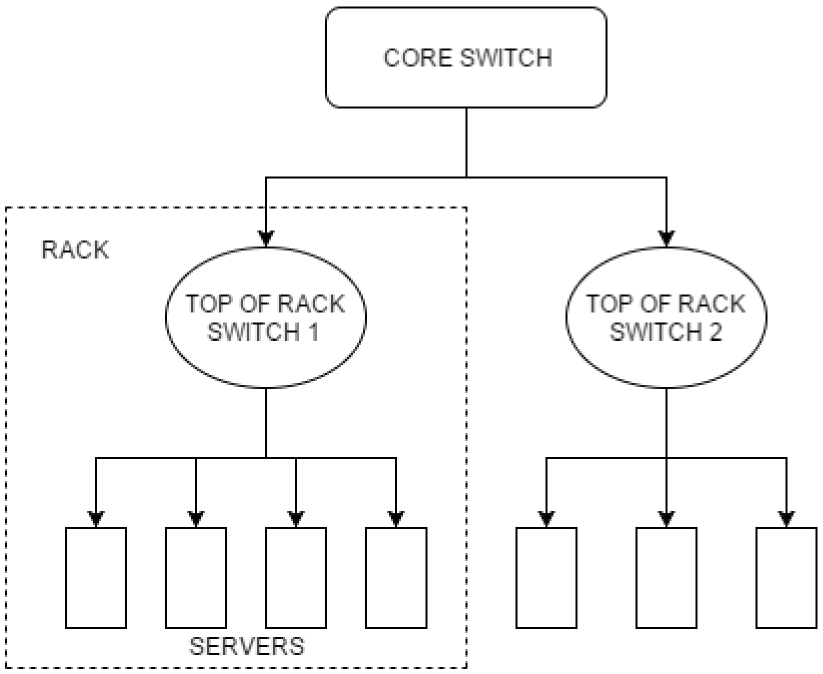


Figure 1.2: Three-layer topology

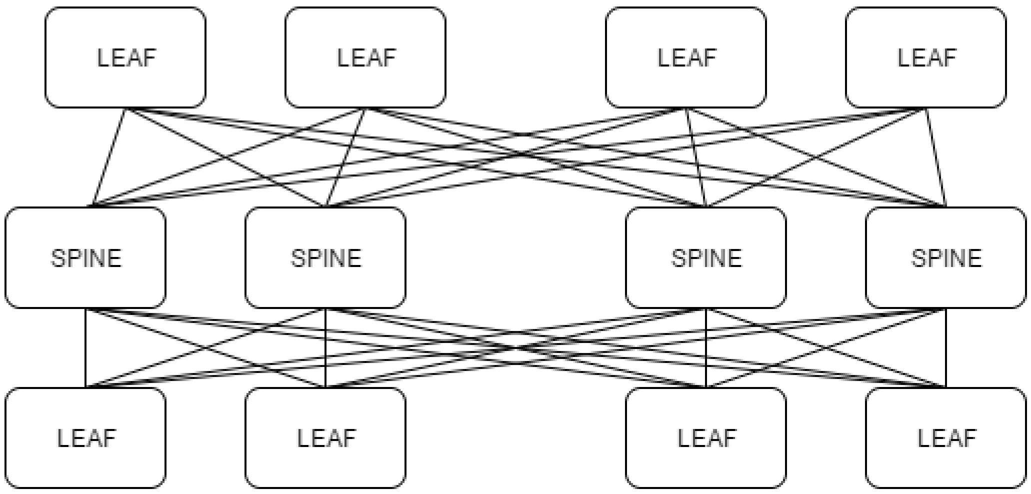


Figure 1.3: Leaf Spine Architecture.

building a data center, the cloud computing would not have been the most sort and trending computing model with a software technology termed as Virtualization.

Vitualization as the name suggests is the process of creating a virtual version of a resource logically build on top of a physical infrastructure. In computing terms, it simply means provisioning device or resources such as a server, storage device, network or even a fully-fledged operating system, where the virtualization framework divides the resource into one or more execution environments. Users are able to interact with virtual resources as if it were a real single logical resource. In simple terms, virtualization describes the separation of a resource or request for a service from an underlying physical infrastructure. For example, with virtual memory, a system gains access to more memory than physically present by swapping of data to the disk.

Some of the leading business challenges in todays IT organizations are: cost effective utilization of infrastructure, responsiveness in supporting new business initiatives and flexibility in adapting to organizational changes. Virtualization techniques can be applied to other IT infrastructure layers like networks, operating system and applications. Virtualization technologies or virtual infrastructure provides a layer of abstraction between computing, storage and networking hardware. Virtualization gives data center operators the advantage of managing pooled resources across different tenants allowing data center owners to be more responsive to dynamic organization needs and to better leverage infrastructure investments.

The key benefit of virtualization is the ability to run multiple operating systems on a single physical system and share the underlying hardware resources —partitioning. Virtualization can be applied to a range of systems layers like: i) hardware level ii) operating system level and iii) paravirtualization.

For UNIX and standard X86 systems, the two approaches are used:

i)Hosted architecture and

ii)Bare metal (hypervisor) architecture

Hosted architectures provide partitioning services on top of a standard operating systems and supports various available hardware configurations. In contrast, a hypervisor (virtual machine manager) architecture is the first layer of software installed on a clean x86 system often termed as bare metal virtualization. Since, this architecture enables direct access to hardware resources, this approach is much more efficient than the hosted architecture enabling greater scalability, performance and robustness.

Operating system level virtualization serves a specific function. With this architecture, the virtualization software layer runs on top of operating systems. All guest systems run on top of this layer using the same operating system, but each of the guests has its own resources and runs in complete isolation from other machines. Main identifying difference for Operating system virtualization is the fact that it does not support other operating systems other than the one the host is running.

Paravirtualization allows guest operating systems to gain direct access to the underlying hardware. Instead of using a simulated hardware resources, paravirtualization acts as a thin layer which ensures all guest operating system share the same resources and coordinate among themselves. This method is generally more efficient than traditional hardware emulation virtualization.

The primary aim of containers and virtual machines is to isolate an application and its dependencies in a lightweight self-contained unit that can run anywhere. They eliminate the need of physical hardware, allowing more efficient use of resources. The difference between the two is in terms of their architecture. Virtual machines (VMs) essentially is an emulation of a real computer that performs all tasks as if it is a real physical entity. VMs run on top of physical machine using a “hypervisor” or “virtual machine manager” which in turn runs on either a host or a bare metal. VM has a virtual operating system of its own and hypervisor acts as an orchestrator that manages the sharing of resources among its peers.

Unlike a Virtual machine which provides hardware virtualization, a container provides operating system level virtualization by abstracting processes from each other. Containers like a virtual machine has its own private space for processing, can execute commands as root, has a private network interface and an IP address.

The biggest difference between containers and VM is that containers share host systems Kernel with other running containers. Each container gets its own isolated user space to run processes independent of each other. Since all containers share same operating system level architecture, the only part that is created and varies from container to container is their respective namespaces. This makes containers lightweight as compared to its counterpart i.e. Virtual Machines

By using appropriate virtualization technology, the computing becomes economical. The cost involved of licensing software and using hardware infrastructure is reduced as users pay a monthly/hourly fee to use resources that include hardware, software and maintenance costs. One of the important characteristic of virtualization is the number of guests per physical machine. i.e. density of the machine. Higher the density, less is the cost involved as more virtual machines can be provisioned on the same underlying physical infrastructure. Another vital characteristic of virtualization is Migration. It allows a VM/Container to be transferred from one host to another host which can be passive (cold migration) or active (live migration). Lastly another characteristic that governs effective virtualization is boot latency i.e. time required to provision a guest to respond to a service requirement. Lower latency helps a data center operator to scale up instances during peak load, thereby complying to the SLAs. For establishing a cloud infrastructure, it is vital for a data center operator to choose a guest technology (VMs or Containers) that has high density, low latency and ensures low migration time and minimum service downtime.

One vital consideration for a cloud provider is to respond to peak demand (load spikes) without any considerable lag. If initial boot up latency is high, the data center operator will always provision for expected peak load rather than the expected load to minimize request latency. Less initial boot up time is an important factor for cloud technology as the provider can provision resources dynamically without any delays. Presently by using optimizations like check pointing, the latency can be reduced considerably. Similarly, high density of a machine ensures that by increasing the density as per the service requirements does not downgrade the CPU and I/O performance of the previously provisioned guests. Lastly, migration of machines or containers ensures optimized load balancing and disaster provisioning. Migration of machines is common as more and more distributed applications are deployed on the cloud. Therefore, selecting an appropriate technology that ensures minimum downtime and migration time for a service is of utmost priority.

This thesis contributes updated density, latency and migration time measurements of virtual machines and containers by using latest kernel versions, new technologies like Docker and available optimizations.

1.1 Motivation

Virtualization techniques like VMs and containers experience overheads with respect to the density, latency and migration time. It is vital to understand the overheads before committing to a technology for deploying guests in cloud infrastructure. There are few reports/studies that contrast the performance of virtual machines and containers but they are either dated in an evolving technology or are performed keeping in mind the brand value of a particular virtualization product and sometimes confined to a very narrow application scope. In the interest of obtaining fair experimental data to the scientific discourse in this domain, this thesis performs qualitative and quantitative analysis of virtual machines and containers, considering various optimization's and latest available technologies.

1.2 Scope

This thesis work describes the performance analysis of two virtualization techniques –VMs and Containers. It evaluates the performance of the two based on metrics, which include density, boot latency and migration time. Experiments were performed using workloads that specifically stress system parameters such as CPU and IO operations. The performance of containers and virtual machines are further analysed by scaling the number of guests incrementally and capturing the performance dip.

1.3 Outline

This thesis is organized as follows:

- Chapter 2 gives background details on virtualization, its types, containers, parameters and migration methodologies.
- Chapter 3 surveys related work in terms of performance and migration mechanisms.
- Chapter 4 analyzes impact of increasing the density of virtual machines and containers on the CPU, File I/O and MySQL performance of the guests.
- Chapter 5 analyzes boot up and shutdown latency for both virtual machines and containers. It also analyzes the impact of checkpoint and restore optimization on their respective boot latency.
- Chapter 6 analyzes the migration performance in both virtual machines and containers. We use experimental build packages to perform container migration
- Chapter 7 concludes and summarizes the whole research work.

Chapter 2

Background

The widespread use of virtual machines has brought many positive effects on computing environments and internet businesses. The most prominent advantage of a virtual machine is the capital cost involved are reduced. With such cost merit, virtual machines also provide many advantages in terms of operation. They allow us to provision infrastructure fast and dynamically manage computing resources, such as memory size remotely.

The term "virtualization" was coined in 1960 to refer to a virtual machine or "pseudo machine". The creation and management of virtual machines has been termed as "platform virtualization" or "server virtualization". Figure 2.1 shows the classification of virtualization techniques.

For understanding a virtual machine functionality, it is vital to know what a hypervisor is. A hypervisor is a program that controls and manages the operating system. Hypervisor method is one of the most sort out and widely used method among virtualization technologies because of its simplicity and open platform support.

As discussed previously, a hypervisor is a program that enables us to run one or more virtual machines on a single physical host. In virtualization, each virtual machine is called as guest system (guestOS) and the physical server on which the guest operating systems run are called as host system. Some of the commercial products of hypervisors present in the market are Xen, Virtual box, VMWare etc.

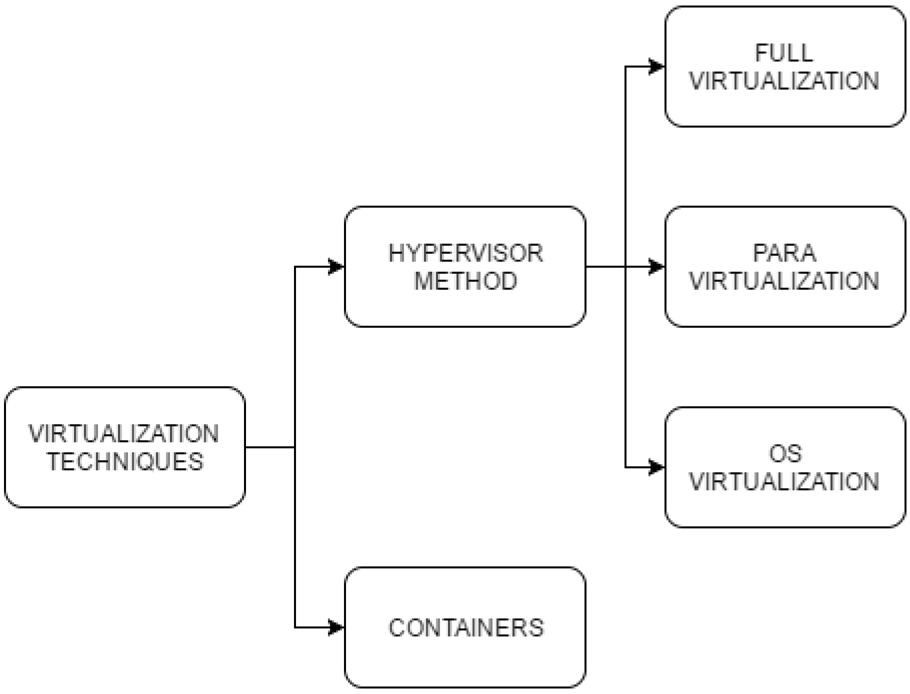


Figure 2.1: Classification of virtualization techniques.

Further a hypervisor is classified into two types (figure 2.2 and 2.3):

I. Type 1: Hypervisor runs on host system without an operating system

II. Type 2: Hypervisor runs on host operating system installed on the host system.

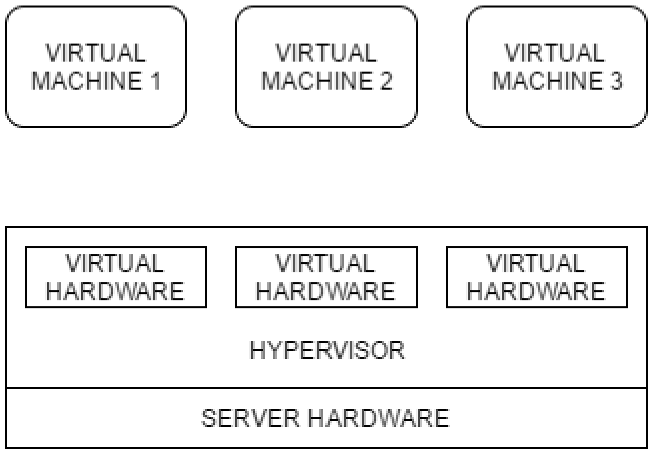


Figure 2.2: Type 1 hypervisor.

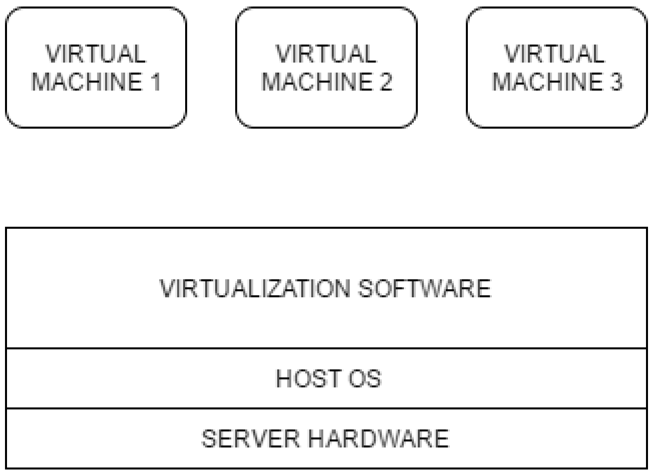


Figure 2.3: Type 2 hypervisor.

2.1 Types of Virtualization

2.1.1 CPU Virtualization

Since the physical host has only one CPU but all the running virtual machines require their own separated CPU, this gives rise to the need of CPU Virtualization. This CPU virtualization is enabled by hypervisor, it converts the set of CPU commands by guest operating system and passes it on to the host CPU and collects the processing results and delivers it back to the guest system. There are several ways in which converting and delivering a set of CPU commands are performed. For better understanding, fig 2.4 shows a privileged architecture, also called as "Protection ring" of X86 architecture.

2.1.2 Full Virtualization

In full virtualization, figure 2.5, a hypervisor has ring 0 authority while a guest operating system has ring 1. In full virtualization, machine code of guest operating system is converted to machine code of host OS through binary translation. For privileged commands such as device driver access, a trap for device access is executed by the hypervisor. Because of this hierarchy, a variety of operating systems can run on the hypervisor, as a result of an indirect access method, the speed in full virtualization is also effected.

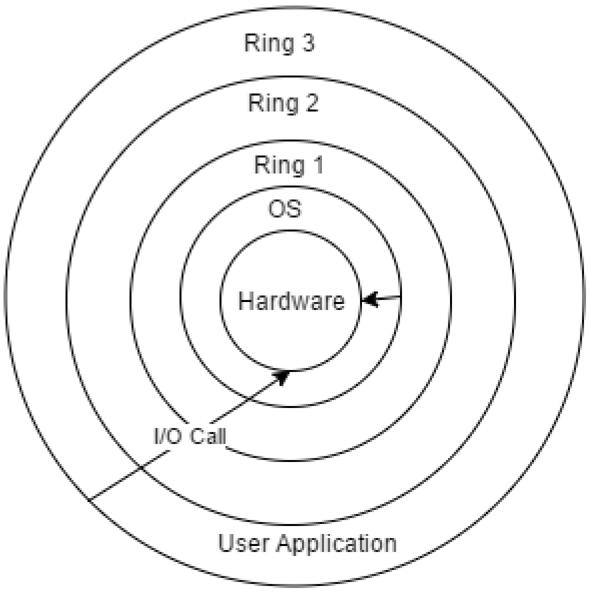


Figure 2.4: x86 architecture.

2.1.3 Para virtualization

As per figure 2.6, in para virtualization whenever a privileged command is executed on the guest operating system, it is delivered to the hypervisor using a hypercall, a system call, instead of operating system, the hypervisor receives this hypercall directly, accesses the hardware and returns the result as guest operating system has direct authority of resources like CPU and memory. This method may be faster than full virtualization method but the disadvantage being the operating system kernel has to be modified for compatibility to hypercalls. Unlike full virtualization, para virtualization provides limited number of guest operating systems the hypervisor can support. E.g. In Linux, approximately 20 % of entire Kernel code has been modified for para virtualization.

2.1.4 Host OS virtualization

In this virtualization method the operating system itself provides the hypervisor functionality. The whole virtualization environment is supported on the host operating system. The only weakness being its inability in intra virtual machine resource management, performance and security. Any security issues on host operating system endangers the

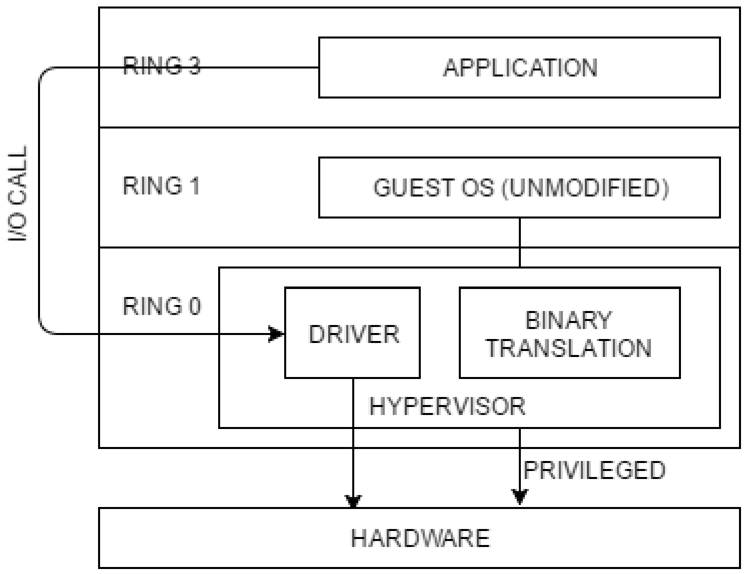


Figure 2.5: Full virtualization.

reliability of the entire guest operating system running on a hypervisor. Figure 2.7 points out the architecture of a Host OS Virtualization.

2.2 OS level virtualization

Lightweight virtualization also called as OS level virtualization is not new. On a Linux platform it evolved from commercial products like VServer to OpenVZ, and, more recently, to Linux Containers (LXC). OS level virtualization is not Linux specific, on free BSD it is called as “Jails”, while on Solaris its “Zones”. Some of these techniques have been available for a decade now and can be seen widely deployed in services like VPS, cheaper and effective alternate to Virtual or physical machines. With this Containers have now been popularized as the core components of public and private platform as a service. Just like virtual machines, a Linux Container can run almost anywhere. Theoretically, containers have many advantages: They are lightweight and easier to manage than virtual machines.

Operating system level virtualization is a server virtualization method in which kernel of an operating system allows the existence of multiple isolated user space instances,

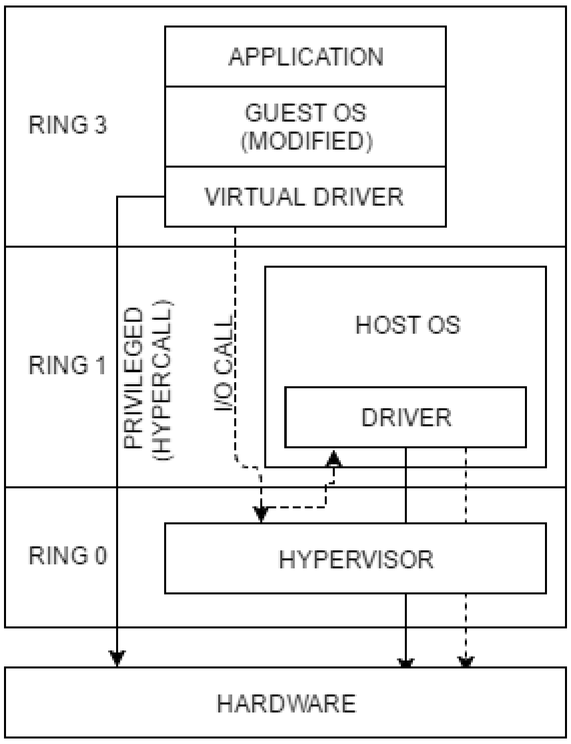


Figure 2.6: Para virtualization.

instead of just one. Such instances, are often termed as Containers, Software Containers, Virtualization engine or jails (Free BSD jail or chroot jail), it may give the illusion of a real server from the point of view of its owners and users. This type of virtualization usually imposes little to no overhead because applications in virtual partitions use the operating systems normal system call interface and do not emulate. This form of virtualization is not flexible as other virtualization approaches because it cant host a different guest operating system or kernel as compared to its host. Fig 2.8, shows the architecture of an operating system virtualization method with a number of virtual servers running on a shared kernel.

The operating system level virtualization can be briefly explained based on the evolution of the virtualization products in the virtualization market: Solaris, Free BSD, LXC and Docker.

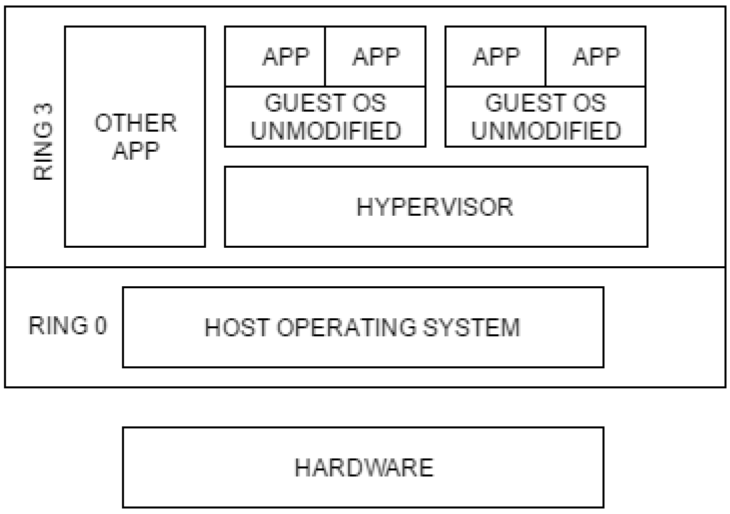


Figure 2.7: Host OS Virtualization.

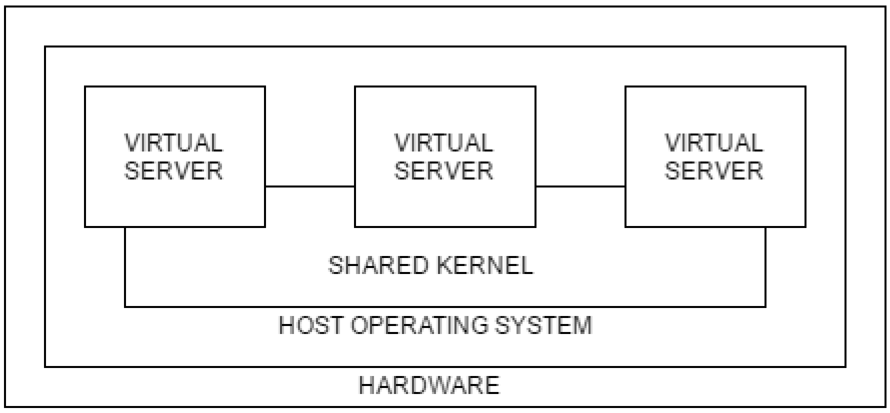


Figure 2.8: OS level Virtualization.

2.3 Evolution of Containers

2.3.1 Solaris Containers

It is an implementation of operating system level virtualization technology for X86 and SPARC (Scalable Processor Architecture) systems. It can be understood as the combination of system resource control and boundary separation provided by zones. Zones are like an isolated entity (virtual servers) within a single operating system instance. By consolidating multiple services onto one system and by isolating them into separate zones, system administrators can reduce cost and provide most of the same protections of separate machines on a single machine. Solaris containers with resource controls are termed as Zones. There are two types of zones: Global and Non global zones, the traditional view of a Solaris OS is a global zone with process ID as 0. It is the default zone and is used for system wide configuration, each of the non-global zone shares the same base kernel with the main global zone. All the non-global zones have their own process ID, file system, network namespace and are isolated from all other zones running on base system. All of them have their separate boot environments, the zones also have many states and can be categorized in following states: Configured, Incomplete, Installed, Ready, Running, Shutting down.

2.3.2 BSD Jails

Jails are built on the chroot concept, which changes the root directory of a set of processes. It in turn creates a safe and an isolated system, process created in the chrooted environment cannot access resources outside of its boundary. This creates a security framework where a compromised chrooted environment wont affect the entire system. In a traditional chroot environment, processes have their separate file system they can access independently. Rest of the system resources, users running processes and networking subsystem Jail is characterized by four elements:

I. A directory subtree: It is the entry point of a jail. Once process is inside the jail, it is not allowed to escape outside of this subtree.

II. Hostname: Used by the jail

III. IP Address: Assigned to a jail. It is usually an alias address for an existing network interface.

IV. A command: The path name of an executable inside the jail. This path is relative to the root directory of the jail environment

Jails have their own set of users and root account which are limited to jail environment. The root account of a jail is not allowed to perform operations outside of the associative jail environment. Thus we can say that there are various operating system level virtualization techniques on different operating system that are more or less extension to chroot environments.

2.4 Containers

2.4.1 Linux Containers

Linux containers is an operating system level virtualization method for running multiple isolated Linux systems on a single host. LXC doesnt provide a virtual machine but rather a virtual environment that has its own CPU, memory, file system and network. It can also be defined as a lightweight Linux environment which is hermetically sealed, introspectable, running artefact and helps in reliable deployments, high portability and loose coupling. It generally supports only homogeneous environment. The initial release of LXC was in 2008 but it came into mass deployment only in 2014 when kernel 3.8 was released, as this kernel release supported features like Namespaces and Cgroups. The credit for the existence of LXC can be primarily given to Linux features: Namespaces and Cgroups

Namespaces : A concept originally developed by IBM, a Linux namespace wraps a set of system resources and presents them to processes within the namespace, making it look as if they are dedicated to the processes. Linux currently has six different types of namespaces: Mount, IPC, UTS, PID, User Namespace and network. All these namespaces together form the basis of containers.

Cgroups: Originally contributed by google, Cgroups is a Linux kernel concept that governs the isolation and usage of system resources, such as CPU and memory, for a group of processes. For example, if we have an application that takes up a lot of CPU cycles and memory, such as a scientific computing application, we can put the application in a cgroup to limit its CPU and memory usage. An example of a cgroup is given in figure 2.9.

Figure 2.10 describes the architecture of a Linux container, which has a host operating

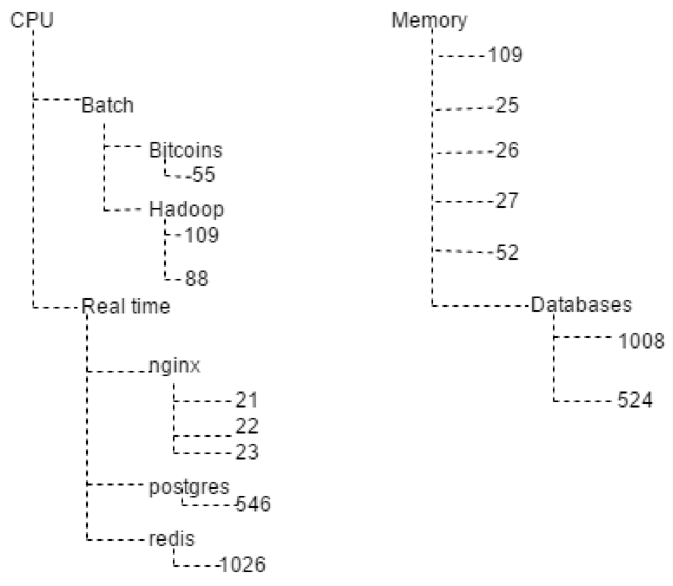


Figure 2.9: cgroup example.

system (usually a Linux OS) running on a physical hardware. The host operating system in turn hosts number of different containers with a minimal operating system installation (Bins/Libs) which in turn runs in complete isolation with respect to each other but share the host operating system as well as the underlying infrastructure.

Containers supposedly have near native performance as compared to virtual machines as they do not emulate hardware. These isolated environments run on a single host and share the same kernel and do not allow running different operating systems or kernel versions.

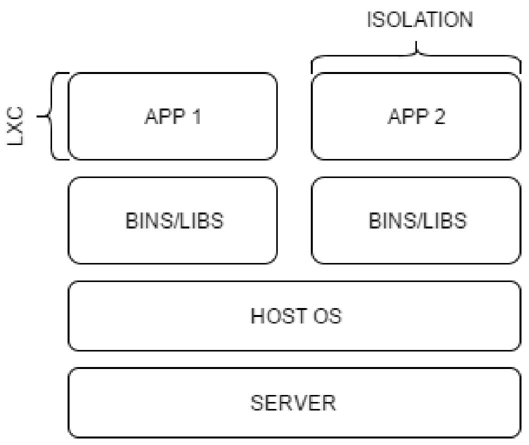


Figure 2.10: Linux Containers.

2.4.2 Docker

With time, virtualization methodologies have evolved and various new techniques have been practiced, out of this evolving field, application containers have come forward as one of the most widely deployed virtualization technique presently. In the era of micro services, application containers are meant to run a single service, they have a layered filesystem and are built on top of operating system container technology. The evolution of virtualization can be summarized from the figure 2.11 below with hypervisor technology at top of the triangle followed by operating system containers and application containers.

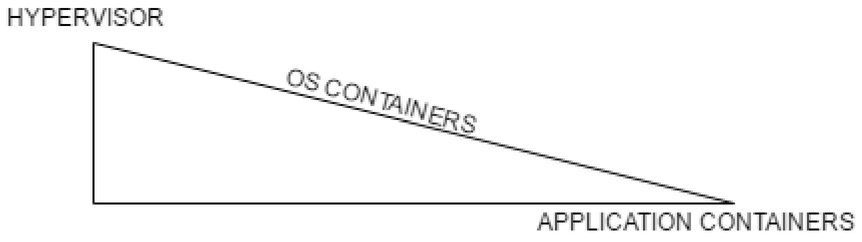


Figure 2.11: Evolution of Virtualization.

Docker is an open source project that makes packaging of applications inside containers by providing an additional layer of abstraction and automation of operating system virtualization on Linux. Containers themselves are just an abstraction over Linux cgroups, which in turn are lower level kernel construct for jailing and limiting the resources of a process and its children.

Docker initially used LXC (Linux containers) as its base but subsequently switched to runC also known as libcontainer as shown in figure 2.12. Like LXC, runC also runs in the same operating system as its host allowing it to share host operating system resources like RAM, CPU, networking etc. In short, Docker adds an application deployment engine on top of a virtualized environment. It is a lightweight and powerful open source container virtualization technology combined with a workflow for building and containerizing applications.

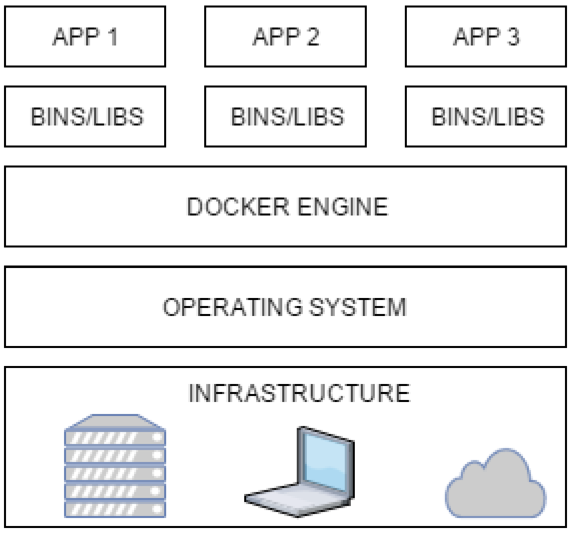


Figure 2.12: Docker low level architecture.

2.4.3 Docker Components

I. **Docker Client and Server**: Docker can be understood as a client server application as shown in figure 2.13. Docker client talks to the Docker server(daemon) which is responsible for managing all the requests. Docker presently does not support other operating systems as it requires a Linux kernel.

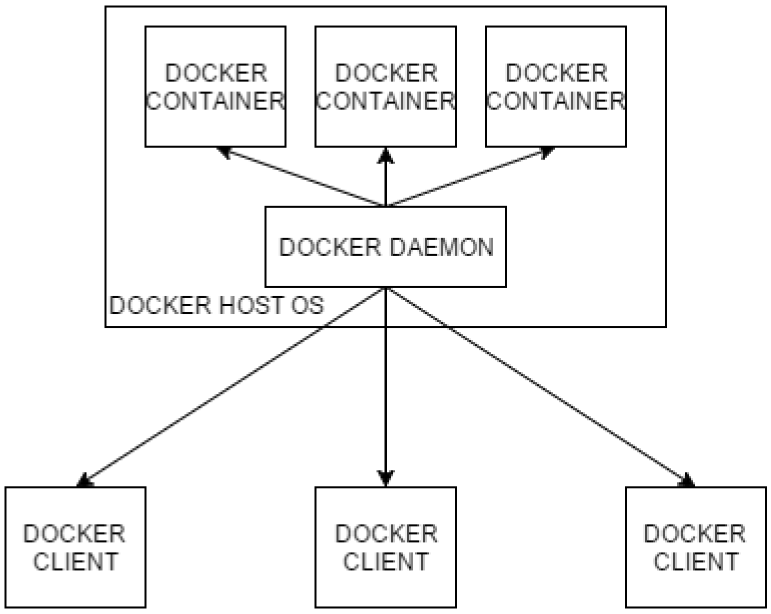


Figure 2.13: Docker client-server view.

II. **Docker Images**: Images are the building blocks of Docker. They are the "build" part of Dockers life cycle and are in a layered format, using union file systems that are build step by step using series of instructions. *Docker file* contains all the necessary commands a client can run to assemble an image. An analogy can be understood as Docker file to be the "source code" and images be the "compiled code" of the containers, which are the running code. Docker files are highly portable and can be shared, updated or stored easily.

III. **Docker Registry**: Once an image is build it is stored to a registry either in a native Docker hosted registry *Docker hub* or any user defined registry.

Figure 2.14 and 2.15 illustrates the key differences between operating system containers and application containers with respect to their supported commercial products LXC and Docker. As we can see, all the services are bundled together in OS containers. It has a neutral filesystem and behave like virtual machines with a fully fledged operating system. Data can be stored inside or outside the container depending on the nature of the running services. Operating system containers are loosely coupled and are built in composite stacks. On the other hand, application containers mimic the architecture of micro services and are made up of read only layers using AUFS (Unification File System) or device mapper. Instances are ephemeral and persistent data is stored in bind mounts to host or data volume containers. Application containers are specifically designed to support a single application.

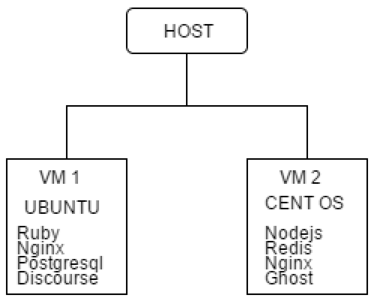


Figure 2.14: Services hosted in OS Containers.

Figure 2.16 summarizes the important commercial products available in the market for each virtualization technique.

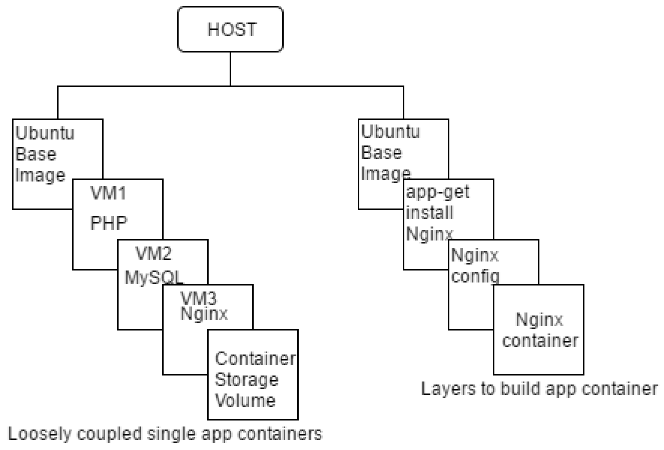


Figure 2.15: Services hosted in application containers.

2.5 Virtualization Parameters

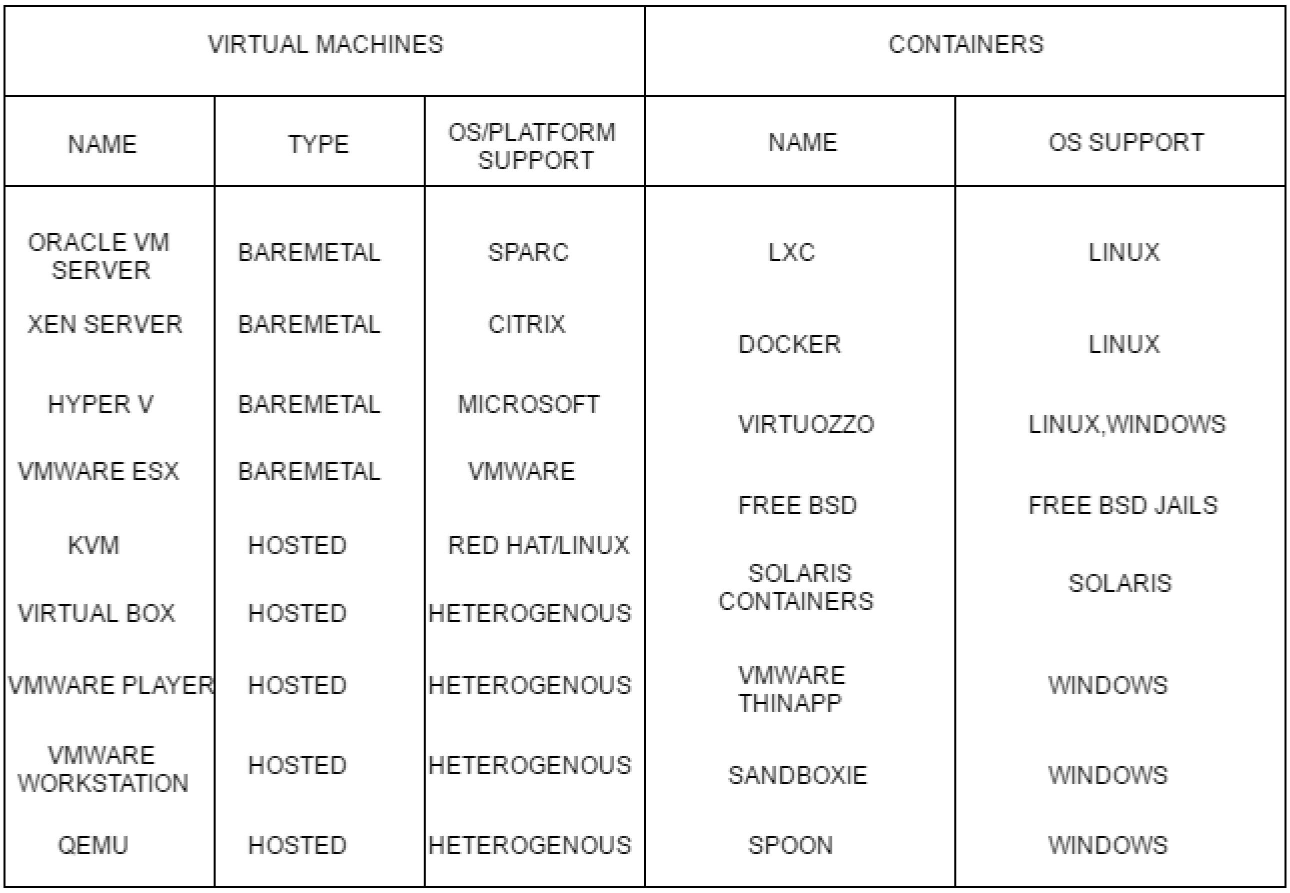
2.5.1 Security

Unlike virtual machines, containers expose host system call table to each guest and relies on pointer hooks to redirect system calls to isolated data structures called as *namespaces.* One of the important security concern in containers is the presence of an exploitable vulnerabilities in pointer indirection code which eventually leads to privilege escalation or data leakage. All the system calls on a guest operate in the same kernel address space as data structure of other guests therefore disabling capabilities like loading kernel extensions. Another reason, being any vulnerability in system API of host kernel are shared, a bug exploited through system call is a shared vulnerability among peer container but not in the case of co resident virtual machines.

2.5.2 Density

Density can be defined as a characteristic which tells us how many virtual machines or containers can run successfully on a given underlying physical hardware which in turn depends on:

I. Virtualization technology employed (VMs or Containers).



VIRTUAL MACHINES			CONTAINERS	
NAME	TYPE	OS/PLATFORM SUPPORT	NAME	OS SUPPORT
ORACLE VM SERVER	BAREMETAL	SPARC	LXC	LINUX
XEN SERVER	BAREMETAL	CITRIX	DOCKER	LINUX
HYPER V	BAREMETAL	MICROSOFT	VIRTUOZZO	LINUX,WINDOWS
VMWARE ESX	BAREMETAL	VMWARE	FREE BSD	FREE BSD JAILS
KVM	HOSTED	RED HAT/LINUX	SOLARIS CONTAINERS	SOLARIS
VIRTUAL BOX	HOSTED	HETEROGENOUS		
VMWARE PLAYER	HOSTED	HETEROGENOUS	VMWARE THINAPP	WINDOWS
VMWARE WORKSTATION	HOSTED	HETEROGENOUS	SANDBOXIE	WINDOWS
QEMU	HOSTED	HETEROGENOUS	SPOON	WINDOWS

Figure 2.16: Summary of commercial virtualization products.

II. Optimizations and features provided by virtualization technology. E.g. Page sharing, ballooning etc.

III. The workload involved.

Some hypervisors reserve the whole guest virtual machine memory on its startup and does not allow memory over commitment. Theoretically it helps to achieve maximum performance but usually there is a bit of tradeoff between high performance and density.

QOS (Quality of service) is also a vital parameter for any virtualization technology e.g. response time. When an underlying hardware can handle the load, the metrics either do not grow while increasing the number of virtual machines and containers or grow linearly. When hardware becomes over utilized, these metrics start to degrade exponentially.Theoretically, containers possess higher density than containers because:

I. No memory reservations as containers are usually a free running flexible processes.

II. Containers memory management is system wide wise. If one container needs more physical RAM and hardware does not have more memory available, kernel automatically reclaims last recently used caches of other containers.

III. Container CPU scheduler is an interactive system wide. If an external event triggers a process, it quickly preempts current CPU hog task and schedules the interactive one.

2.5.3 Latency

Latency can be defined as the time required to provision a guest to service an incoming request. The boot up latency directly influences the efficiency of a cloud infrastructure. A low latency overhead ensures on demand provisioning of guests during break load as specified in their SLAs. Resources consumed during virtualization can be a significant factor of overhead equation. High boot up latency is a major source of overhead. Unlike conventional applications that are brought up once and kept running, emerging environments often run short lived tasks. If an average task runs for a few couple of minutes, it is unacceptable to spend a major fraction of that time booting the same virtual machine or container. Theoretically, containers have a lower boot up latency than virtual machines, as only API and *lib* directory related data structures need to be initialized, while in virtual machines the delay is because of booting an entire operating system, as containers share the same kernel and bootloader, other initialization tasks are skipped. Boot up latency is important and for applications that scale out to manage peak loads. Other optimizations like check pointing a booted virtual machine and restoring it further decreases the boot up latency of virtual machines. Theoretically even after all the optimizations, containers possess a lower boot up latency than the virtual machines.

2.6 Migration

With reference to virtualization, where a guest simulation of an entire computer is actually a software virtual machine under a hypervisor or a running container. Migration or teleportation can be defined as the process in which a running virtual machine or a container is moved from one physical host to another, with little or no disruption in service. The process of migration should be completely transparent with no downtime. Practically there is always a minor pause in availability but the magnitude is low enough that only hard real time systems are affected. The migration in terms of virtualization can be described as below.

2.6.1 Use cases

Migration becomes mandatory in the following conditions:

I. The running host has encountered a failure / Catastrophic failure

II. The host needs maintenance

III. Load balancing

IV. Optimal resource placement with respect to efficiency and topology

2.6.2 Assumptions

Some of the common assumptions during a migration process are:

I. Migration is LIVE (zero downtime)

II. Consistency is maintained

III. Transparency

IV. Minimal service disruption

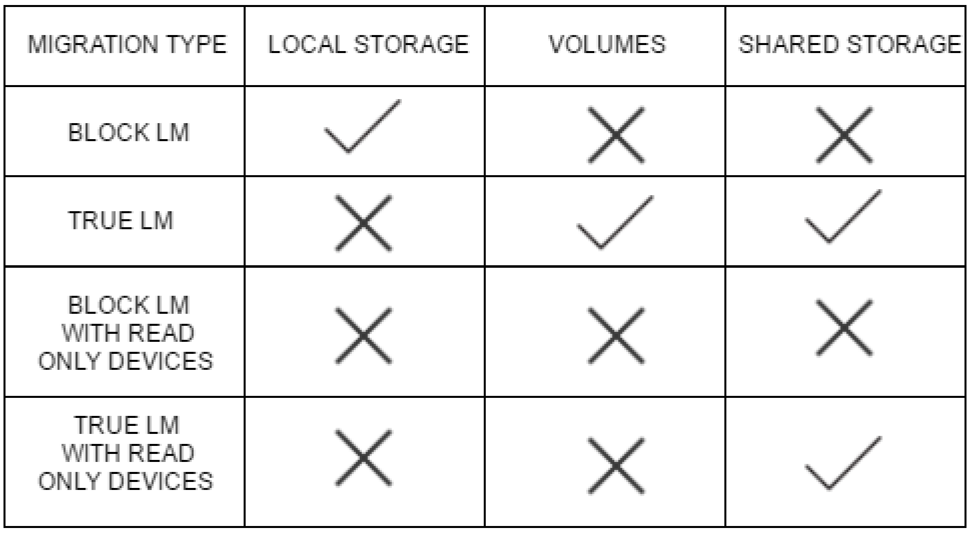
2.6.3 Types of Migration

Migration can be classified as:

I. Non live migration (cold migration)

II. True live migration (shared storage based)

III. Block live migration (volume based): No shared storage required, downtime expected.



MIGRATION TYPE	LOCAL STORAGE	VOLUMES	SHARED STORAGE
BLOCK LM	✓	✗	✗
TRUE LM	✗	✓	✓
BLOCK LM WITH READ ONLY DEVICES	✗	✗	✗
TRUE LM WITH READ ONLY DEVICES	✗	✗	✓

Figure 2.17: Characteristics of different types of migration.

Figure 2.17 demonstrates the characteristic of each migration type

2.6.4 Live Migration

Live migration is usually performed using these two techniques:

I. **Pre Copy**: Memory state is copied first followed by state transfer

II. **Post Copy**: Processor state is sent first followed by memory contents

Pre copy is one of the most widely used technique in majority of the deployment configurations.

The Pre copy Live migration process can be explained from fig 2.18.

The pre copy live migration begins with a daemon process reserving a particular resource on the destination machine. Numerous algorithms and optimizations are employed to identify the target host for migration. After finalizing the target host, the memory is transferred from source to destination/target host. Once the whole memory is copied, the

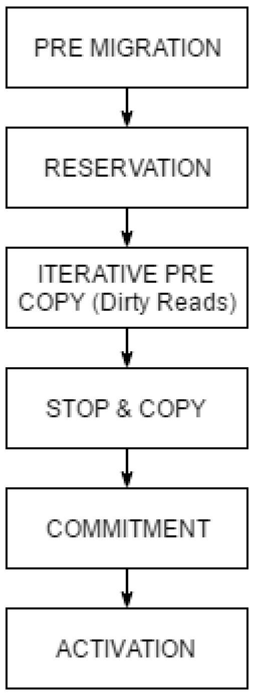


Figure 2.18: Pre Copy Live VM Migration Process flow.

processor starts to copy the chunks of memory written during the round trip time also known as *Dirty reads*. This process continues till a minimum threshold of memory is left after which the source host is stopped and the destination host/ target is switched on, all the new incoming requests are now serviced by the new physical host and in case of any missing memory bytes a page fault is usually created and the old host acknowledges the fault with required memory transfer. After successful transfer, the service at the earlier host is discontinued and the new physical location now services all the clients without any considerable downtime or delay. Various new algorithms and optimizations have been introduced to enable a hassle free migration with minimum downtime and high efficiency and are usually a bundled part of the virtualization product in the market.

Chapter 3

State of Art

This thesis analyses the virtual machine and container overheads with respect to consolidation density, latency and studies the migration process of both the virtualization techniques. It also quantifies the observances with latest experimental data and lays a metric from a data center operator point of view. Further various optimization's are suggested with respect to the observed pattern and a transparent view of virtualization profile is obtained. Thus, the first section of the chapter outlines various studies which compare virtual machines and Containers with respect to density, latency and other influential parameters, and the second section outlines the studies that explore virtualization migration methodologies with respect to virtual machines and containers.

3.1 Virtual Machines Vs Containers Performance

Few previous studies have evaluated the performance of containers with respect to its counterpart virtual machines in specific contexts, this piece of work bridges the gaps left in previous evaluations with respect to both the technologies.

Felter et al. [24] evaluates memory throughput, CPU, storage and networking performance of Docker with respect to KVM (Kernel Virtual Machine). This report claims the performance of Docker either equals or exceeds KVM performance in every case. The experimental analysis were done using the linear equation solving package *Linpack*, the *Stream* benchmark of memory bandwidth, network bandwidth using *nuttcp*, latency using *netperf*, block I/O speeds with *fio* and *Redis* respectively. This study reports KVMs start-up time to be double than that of Docker. Also, the performance of MySQL is evaluated in both KVM and Docker. As per the experiments conducted, Docker has similar performance to native, with the difference asymptotically approaching 2% at higher concurrency. KVM experienced higher overhead, more than 40% in all measured cases. The study was concluded with a claim that when both KVM and Docker are tuned for performance, Docker equals or exceeds KVM performance in every case. For I/O intensive workloads, both forms of virtualization should be used carefully. The only concern shown with Docker in this report is the extra traffic generated during network address translation for the networks. While the results corroborate the findings of this study with different benchmarks, this study ignores the role of memory footprint or scalability of either system. This study also overlooks the checkpoint and restore optimization to start a virtual machine, which considerably decreases the boot up time of a KVM. Further they conclude the report with claims of virtual machines being appropriate for IaaS (Infrastructure as a Service) and containers dominating the PaaS (Platform as a Service) model. The gradual optimization's happening over the years with respect to KVM is compared to native performance delivering containers and evolution of the latter is questioned with respect to its counterfeit. Lastly, the practice of deploying containers inside a virtual machine is questioned as this imposes the performance overheads of virtual machines while giving no benefit compared to deploying containers directly on non-virtualized Linux.

With reference to these studies, Canonical conducted some similar series of experiments with respect to LXD and KVM running a complete Ubuntu 14.04 guest [1]. These experiments take into account the density, speed and latency and evaluate the two products / technologies similarly. The results conclude that for idle or low workloads, LXD gives a density improvement of 1450%, or nearly 15 times more density than KVM. With respect to the speed, on an average, LXD guest takes a mere 1.5 seconds to start while

KVM guests nearly took 25 seconds to boot. Lastly for latency, using a sample 0MQ workload, testing resulted in 57% less latency for guests under LXD in comparison to KVM. Again all important optimization's like check pointing were ignored in case of KVM.

Kyoung-Taek Seo et al. [45] constructs cloud environment under same two servers. One for KVM as a virtualization tool in Openstack platform and the other one for Docker. The reason of choosing KVM as a hypervisor technology was that KVM delivers near native performance and is provided by the native Ubuntu server. This paper evaluated the size, average boot time and operational speed with respect to the two technologies. For size, the paper concludes that not more than 50 virtual machines can be provisioned on a 500 GB hard disk with an average resource requirement exceeding 8 GB while in case of Docker containers, it only used half of the same 500 GB HDD and 177 Mb of resources to generate more than 100 images. For average boot time, 20 images were generated on each server with same hardware and software configurations, NMON (Nigels Monitor) tool was used to monitor the performance. As per the results, in case of Docker, average boot-time was 1.53 seconds and standard deviation as 0.03, while in case of virtual machines, average boot-time was 11.48s and standard deviation as 3.4s. The dependency on CPU resources was concluded as the deciding factor for such diverse numbers. Lastly for operational speed measurement, a python code for calculating 100000! was used as a benchmark and the process was repeated 100 times to eliminate vague readings. With Docker environment, operational speed was noted as 4.546s and standard deviation as 0.02s. While in case of virtual machine environment, operational speed was 4.793s and standard deviation as 0.05. To conclude, CPU resource dependency and storage were considered as prime parameters for such diverse numbers. Again in this paper, industrial workloads were ignored with minimal focus on I/O and memory optimization's.

Another study measures the memory dependency of Docker and KVM using OpenStack [4] and concludes that while running a real application workload, KVM requires six times more memory than containers. This study ignores the degree to which these measurements are fundamental.

Clear containers [11] is a Linux project which aims to reduce the overheads of virtual machines as compared to containers. The project focuses on optimizing away a few start boot CPU initialization delays, which in turn reduces start-up time of a virtual machine. Some of the optimization's used are as under:

- Introduction of a fast and light weight hypervisor. Instead on finding optimization's with respect to QEMU, this project uses *kernel virtual machine tool*
- Optimization's in the kernel
- Optimization's in system
- Utilization of the DAX (Direct Access) feature of the 4.0 kernel which enables the page cache and virtual machine subsystems to be bypassed entirely, allowing for faster file system accesses and lower per-container memory usage
- KSM (Kernel Same-page Merging) on the host. This allows virtual machines to share memory pages in a secure manner for memory that is not already shared via DAX
- Optimization of core user space for minimal memory consumption

To conclude, clear containers is a system where one can use the isolation of virtual-machine technology along with the deployment benefits of containers. It is an ongoing work and some of the optimization's introduced along with some improvements by this can help in reducing the overheads of virtual machines to a large extent.

Soltesz et al. [46] evaluates the scalability of Linux V Server (container technology) with Xen. According to the experiments conducted, Linux V server performs twice better than virtual machines for server workloads and scales further while preserving the performance parameters. In case of I/O related benchmarks, Xen performs worse as compared to V Server. This paper focuses on aggregate throughput and avoids the impact specification on a single host/guest which from a data center operator point of view is vital in achieving quality of service in a multi-tenancy environment.

Estrada et al. [23] ran synthetic benchmarks and measured the run-times of two short-read alignment applications on cloud-like virtualization environments. The environments were implemented utilizing the KVM hypervisor, the Xen hypervisor and Linux Containers. The runtime in each environment is compared against a physical server and the conclusions are derived. For experimentation, two synthetic benchmarks were used to quantify different aspects of each virtual machine manager technology. The paper concluded the following observations:

i) CPU pinning can be effective for managing resource contention

ii) Despite having a similar objective, alignment programs use system resources in different ways like periodic or one time read access

iii) Different workloads with same input and processing produce contrasting execution time as one uses bursty read write patterns, fills caches and causes the application to block an I/O bandwidth while the other makes many small writes

iv) Performance for a given virtual machine manager is dependent on the application being executed

v) CPU bound applications run at near bare metal speeds in virtualized environments

vi) Virtual machine manager scheduling overheard is low when virtual machines are not over provisioned

vii) Linux Containers have low performance overhead

In order to provide meaningful comparisons and keep the work tractable, the vast parameter space of this study was reduced to a few important cases thereby not allowing us to apply the conclusion to a wide range of application domains.

Regola and Ducom [43] did a similar study regarding the applicability of container for high performance computing environment, like Open MP and MPI (Message Passing Interface). The conclusion of the study was that the input/output performance of virtual machines is usually the limiting factor in adopting a particular virtualization technology and that only containers offer a near native CPU and I/O performance. As a result of this, several works/optimization's have been introduced to reduce I/O overheads.

Parameter	LXC	Warden	Docker	OpenVZ
Process Isolation	Uses PID namespaces	Uses PID namespaces	Uses PID namespaces	Uses PID namespaces
Resource Isolation	Uses cgroups	Uses cgroups	Uses cgroups	Uses cgroups
Network Isolation	Uses net namespaces	Uses net namespaces	Uses net namespaces	Uses net namespaces
Filesystem Isolation	Uses chroot	Overlay filesystem	Uses chroot	Uses chroot
Container Lifecycle	Tools lxc-create lxc-stop, lxc-start to create, stop and start a container	Client Server interaction in Warden to manage containers	Uses Docker daemon and a client to manage containers	Uses vzct1 to manage container lifecycle

Table 3.1: Container Implementation Comparison

[17][26][35][47]

Dua et al. [22] explores the use of containers in PaaS. This paper explores various container implementations: Linux Containers, Docker, Warden Container, Imctfy and OpenVZ. The parameters used for implementation differentiation were process handling, file system and namespace isolation. Table 3.1 summarizes the outcomes of this paper while they mention the following factors in choosing a right container technology:

i) Ecosystem for pre build containers

ii) Hardened layer for isolation of Process, Network, CPU and Filesystem

iii) Tools to manage lifecycle of a container

iv) Ability to migrate containers between hosts

v) Support for multiple operating systems and Kernels.

Like before, this paper further questions the use of containers inside a virtual machine and encourages direct container deployment by using management tools like *libvirt* for near native performance, containers can be seen as a part of IaaS layer rather than just a PaaS service. The paper concludes with the claims of containers having an inherent advantage over virtual machines because of their performance improvements and reduced start up time. To add to the conclusion, all the commercial container products use the same common Linux containment principles like *chroot* and *namespaces* be it Docker or OpenVZ.

Xavier et al. [52] performs the comparative evaluation in terms of performance isolation of Xen (Virtual machine) to various container implementations including Linux VServer, Open VZ and Linux Containers (LXC). This paper concludes Xens performance isolation is considerably better than other container implementations, it is only for CPU isolation where Xens performance is not comparable to its container counterparts. An another study [36] also concludes the same claims but considering the pace of innovation in virtualization field, we note that these results may be dated as the experimental setup used technologies dated back in 2009.

Chang et al. [19] performs the analysis of virtualized server and shared storage accessing performance with the estimation of consolidation ratio and TCO (Total cost of ownership)/ROI (Return on Investment) in server virtualization. The setup implements five heterogeneous virtualized cloud computing systems:

i) vSphere ESX/ESXi Server

ii) Hyper-V R2 Server

iii) Proxmox Virtual environment server

iv) KVM at Ubuntu Enterprise Server

v) Cent OS based Xen server

All these heterogeneous virtualized clouds use a shared storage system. They conducted a series of three experiments with each focusing on virtual machine performance isolation, Performance evaluation based on shared storage and estimation of consolidation ratio and TCO/ROI. The paper concludes that virtual machine performance achieves nearly the same level for all of the virtualized servers, but the estimation of virtual machine density and TCO/ROI totally differ among them. We may choose a virtualization product such as ESX server if we need a scheme with the highest ROI and the lowest TCO in server virtualization. Alternatively, Proxmox VE would be another best choice if we like to save the initial investment at first and own a high-performed virtualized infrastructure in server virtualization.

Morabito et al. [41] also presents an extensive comparison of traditional hypervisor based virtualization and new lightweight solutions like containers. Several benchmarks were evaluated with respect to parameters like processing speed, storage, memory and network. This paper uses KVM, LXC, Docker and OSv for its evaluation. Paper concludes for CPU performance, both container based solutions perform better than KVM. Considering only the computation time, containers display performance almost similar to the native environment. In terms of memory indexing, KVM introduced roughly 30 percent performance degradation. For disk I/O performance, the two container-based platforms offer very similar performance in both cases, which are quite close to the native one. KVM write throughput is roughly a 3rd and read throughput almost a 5th of the native one. For memory performance, KVM, Docker and LXC all reach performance similar to the native execution. The performance of OSv is approximately half of the others. For network I/O performance, using TCP, LXC and Docker, it achieves almost equal performance compared to the native one, and KVM is 28.41% slower. OSv performs better than KVM and it introduces a gap equal to 26.46% compared to the native system. All platforms offer lower throughput with UDP (User Datagram Protocol). LXC and Docker offer comparable performance between them (42.14% and 42.97% lower than native); KVM overhead is the largest (54.35%). OSv ranks in the middle (46.88% worse than native). The paper concludes on the performance of KVM being improved over past couple of years with appropriate optimization's. Disk I/O efficiency represents the only bottleneck for some types of applications in case of KVM.

Moolenbrook et al. [48] argues that the two virtualization techniques: hardware level and operating system level virtualization are extremes in a continuum and that boundaries in between the extremes may combine the several properties of both. This paper proposes a mid level abstraction to form a new virtualization boundary, where the virtualizing infrastructure provides object-based storage, page caching, mapping, memory management and scheduling, whereas any higher level abstractions are implemented within each virtual environment. The prime goal in this paper was to establish a new set of abstractions implemented in the host system and exposed to the domains. Each domains system layer may use those abstractions to construct an interface for its applications. As a supporting infrastructure, the paper proposes exposing a number of micro kernel inspired abstractions: address spaces, threads of execution and IPC (Inter Process Communication). The paper further claims their prototype to be better in terms of flexibility and security as it imposes no internal structure on the system layers on either side of virtualization boundary and manages proper resource accounting and isolation. The initial prototype is built on the *MINIX 3* micro-kernel operating system. A virtualization boundary was drawn between the system services such that the host system consists of the micro kernel, the virtual machine and SCHED services, the lower three Loris layers and all hardware driver services. Each domain has a system layer consisting of a private instance of PM, VFS, the Loris naming layers and any of the other POSIX (Portable Operating System Interface) services as needed by its applications. The user application layer of the domain consists of a copy of the *init* user process and any actual application processes. To conclude the paper, the research work towards this new virtualization technology is still fresh and the prototype has not been tuned / optimized for low level performance thus cannot be taken into consideration by data center operators.

Li et al. [34] presents a brief comparison of virtualization technologies from a high availability perspective. The following parameters were considered vital from a high availability characteristics of virtualization:

i) the ability to retrieve and save the state of a virtual machines

ii) the ability to perform virtual machine live migration with continuous failure detection

iii) automatic state synchronization between the active and standby

iv) recovery management by dynamically failing over to the standby when a failure is detected and terminating the active

In this paper, series of experiments were carried on virtual machines and containers, rising to a conclusion: HA (High Availability) /FT (Fault Tolerance) solutions already exist in hypervisor based platforms and are commonly achieved by fail over clustering.

However, these solutions have limitations, e.g., either on SMP (Symmetric Multiprocessing) support or on guest OS. On the other hand, in container-based platforms, there are large amount of missing pieces. In particular, the checkpoint/restore features in container-based environments are far from complete. Despite the efforts of clustering containers, there is no mature feature available for continuous monitoring to detect failure of the container and automatic fail over management, which are mandatory for a complete high availability solution. Furthermore, future optimization's such as using compression techniques like *qzip*, *bzip2* and *rar* to reduce the data volume from source to destination thereby speeding up the migration process is also proposed. The conclusion clearly drives and proposes the use of virtual machines in a high availability requirement and according to the author, containers are still under maturation and would need some time to cope up with the high availability demand.

Antunes et al. [18] focuses on the use of jail environments, provided by the FreeBSD operating System, which present a relevant set of features that can enhance the performance. A set of data collection tests that allows measuring the degree of optimization obtainable in current models of cloud computing, based on the use of hypervisors tests is also presented. Two different sets of tests were developed, the first focused on the consumption of resources required by each of the respective solutions and applications, while the second set aimed at obtaining access times to resources available for each of these applications. For tests of consumption, the focus was on consumption of CPU, RAM and disk space required for installing the environment and respective applications were also evaluated. Along with this, start up time and availability of services was also measured using system tools like *top* and *du.* The access to resource tests were divided into 3 test groups for 3 different protocols SSH, HTTP & NetBIOS, each protocol was further tested in 4 different scenarios. Each test was performed 20 times and these tests aimed at determining ability of each environment to provide access to resources. It was noted that with the use of ambient jail these resources are mainly channeled to the service and the number of instances has a negligible impact on the computer system. Also the resources required to launch a jail environment are insignificant when compared with the requisites needed to launch a virtualized environment. Jails were more effective than the virtual machines, having a minimum release time accompanied by an irrelevant hardware consumption that does not interfere with performance of the equipment. During data access performance test, it was evident that the number of virtual machine instances negatively influences the overall performance of virtualized systems, being a limiting factor for the scalability of the solution. With jail environment, the impact was greatly reduced, reinforcing the concept of using jail environments in high availability solutions based on open sources services. To conclude, the paper states that the use of virtualized environments translates into a loss of performance against the base machine, which tends to worsen with the increasing number of instances running. A similar trend was observed in the consumption of hardware resources required by the running instances. It can also be stated that the performance of a virtualized system will suffer a loss of performance greater than 50%, and that the number of virtualized instances running, negatively influences the performance of the computer system and can easily compromise the viability of the solution. As opposite, it is observed that when using jails environments, there is no performance influence to the system, and in certain cases, it even increases the performance of the solution. It was also observed that the number of instances of jail environments does not influence the computer system negatively, allowing flexibility of high availability solutions.

3.2 Virtual Machine Migration

Migration of resources and processes from one host to another is very vital for effective cloud management platform. Provisioning and load balancing are the pillars of SLAs laid down by a cloud provider. This sections explains the state of art related to all virtual machine migration followed by container/application migration. The aim of this section is to lay down methodologies and current technologies being used in this domain and finding a combined optimal study of the overheads involved in both the virtualization techniques.

Clark et al. [20] introduces a writable working set, the paper uses Xen hypervisor to demonstrate virtual machine migration. The author targets migration of active OSes hosting live services. As described before, pre copy approach of migration is used for migration, the migration is examined with respect to two vital parameters:

i) downtime

ii) total migration time

This paper also introduces mechanism of *pure stop and copy* and pure demand migration. Author also lays down issues in migrating local dependencies e.g. In a cluster environment, network interfaces of the source and destination machines typically reside on single switched LAN. Also generating an unsolicited ARP (Address Resolution Protocol) reply from migrated host, adverting the IP has been moved to a new location is considered as an overhead. The author also lays down few implementation issues such as inconsistency in shadow page tables and irregularity in check pointing in case of self-migration. In the end, the paper concludes that realistic server workload such as SPECweb99 can be migrated with 210ms downtime while the Quake 3 game server is migrated with an imperceptible 60ms outage. However, the paper ignores WAN (Wide Area Network) migration and environments that cannot rely completely on network shared storage. The paper also questions the iteration level in pre copy phase i.e. the pages that are modified often (writable working set) and concludes the algorithm purely relying on heuristics.

Mishra et al. [40] describes virtual machine migration as the transferring of whole state in order to enable maintenance and load balancing. The paper emphasizes on identifying the virtual machine type and managed resource allocation rather than using good old bin packaging approach but for workloads like a web application, finding an initial resource pool is usually hard as predicting traffic density and pattern is non trivial. The paper further points out various migration schemes like suspend and copy, pre copy and post copy. The paper also details each process with respect to the context.

Nelson et al. [42] describes a migration system named **vMotion**, a VMware product. The paper lays down three kind of states that are mandatory during migration:

i) Virtual device state. E.g. CPU, Motherboard, networking, storage adapters, graphic adapters etc

ii) External connections with devices including networks, USB devices and CD ROMS

iii) Virtual machines physical memory

For networking, VMware ESX Server architecture provides a VNIC (Virtual Network Interface Card), it has a MAC address that uniquely identifies it on local network. Each VNIC is associated with one or more physical NICs (Network Interface Cards) managed by virtual machine kernel, as each VNIC has its own MAC address that is independent of physical NICs MAC address, the mobility is easy. For storage, it relies on SAN (Storage Area Network) or NAS (Network Attached Storage) to allow us to migrate connections to SCSI devices. Paper assumes that all physical machines involved in migration are attached to the same SAN or NAS server. The author acknowledges, physical memory as largest piece of state that needs to be migrated, also a level of indirection is introduced to iteratively pre copy the memory while virtual machine continues to run on source machine. During migration, before each page is copied, it is marked as read only so that any modifications can be deduced by virtual machines manager. This process is repeated till the number of modified pages is small enough or there is insufficient forward progress. According to current VMware product, it terminates the pre copy phase when there are less than 16 Mbs of modified pages left and there is a reduction in changed pages of less than 1 Mb. This paper focuses on calculating the time required to migrate a virtual machine and the time period during which virtual machine is unavailable. The total downtime was less than one second for all the workloads except *memtest86* as it rose minimally with increasing memory sizes. memtest86 was a pathological case where all the memory was modified during the pre-copy so that the virtual machine downtime equals the time necessary to send the virtual machines entire memory. End to end time strictly depends on the size of the virtual machines memory, and confirms the need to keep the virtual machine running during most of this time. With pre-copying, the virtual machine continues to run while memory is being transferred to the destination. The number of pre-copy iterations required to migrate each workload was small. All workloads except for memtest86 took 1 or 2 rounds before the number of modified pages was small enough to terminate the pre copy. It took 2 or 3 rounds before the pre-copy was aborted because of lack of progress for memtest86. To conclude, the results differ from nature of workloads and the measurements point out that virtual machine normally experiences less than one second of down time. Also, the end-to-end time of the migration and the impact on other virtual machines running on the machine involved in migration can be controlled by proper management of CPU resources.

Clark et al. [21] also broadens the pre copy approach and focuses on iterative page transfer. This paper further claims that the size of virtual machine directly influences the total transfer time.

Wu, Zhao et al. [51] measures the performance of live virtual machine migration under different level of resource availability. This paper focuses on deducing performance measurements like downtime and total migration time. This paper considers only CPU usage of virtual machine migration and the modeling of migration performance under different CPU allocations on source and destination. The fundamental goal in this paper is to build a performance model for live virtual machine migration which can accurately estimate the migration time based on resource allocation. The paper also models the relationship between resource allocation and migration time by profiling the migration of virtual machines running different types of resource intensive benchmarks. Xen was used to perform all the experiments with a DomU (Domain Unprivileged) virtual machine running different resource intensive applications while Dom0 (Domain Zero) was allocated different CPU shares for processing the migration. The results from the experiment showed that the virtual machine migration time is substantially impacted by Dom0s CPU allocation whereas the performance model can accurately capture the relationship with the coefficient of determination generally higher than 90%. The paper concludes with Table 3.2 which shows the difference between the predicted and measured migration times was typically within a few seconds. The prediction accuracy proved to be vital

VM	Average	Median
CPU-Intensive	1.58	1.74
Memory read intensive	7.63	3.33
Memory write intensive	9.25	6.79
Disk I/O intensive	4.01	2.97
Network send intensive	3.53	3.16
Network receive intensive	4.94	2.43

Table 3.2: Model Prediction Error

for migrating a CPU intensive virtual machine, the prediction error was higher when the CPU allocation to Dom0 is low for migrating a memory, disk or network intensive virtual machine. It also concludes that when Dom0 is under heavier contention from the DomU activities, its behavior becomes less predictable.

He, Guo et al. [28] preferred freezing and transferring of the state rather than using other methods. The paper points out resource management as cumbersome because of guest operating system using considerable amount of cloud resources. Further, author also points out the need of an approach for better resource provisioning and management in hybrid cloud environment.

Zhao et al. [53] implements a simple model that decreases the migration time of virtual machines by shared storage and fulfills zero downtime of relocation of virtual machines by transforming them as Red hat cluster services. It also proposes a distributed load balancing algorithm **COMPARE_AND_BALANCE** based on sampling to reach an equilibrium solution. The paper uses OpenVZ and evaluates the live migration time of virtual machine and the convergence. In the experiment, *EUCALYPTUS* provides support for VLAN across multiple physical hosts via VDE (Virtual Distributed Ethernet) virtual switches and there is a VDE switch on each physical host. In the implementation, the physical host on which algorithm is running randomly uses the loads of other physical hosts that have been stored on shared storage periodically. Computation executed at each physical host should not cost too many local resources. Physical hosts may also join the cloud because of scalability and depart from it because of invalidation, so simplicity and statelessness are necessary to the algorithm. During evaluation, for migration time, the data pointed out that the average time of live migration is smallest in a scenario where the whole data of a virtual machine is not considered for copying, all it does is to sum the contents of a VE to a file on shared storage and restoring it via the file on the target host. The worst migration time occurs when there is a direct operation of data on the shared storage and the process includes a lot of locking operations and synchronization cost. For convergence, during the running process of system, each host logs the processor usage every minute on the shared storage and the algorithm executes once every twenty minutes. It then uses the standard deviation of average processor usage of all hosts in past twenty minutes as a measure of the balance of the system. If the number of system hosts were very large, it would select some hosts uniformly at random and then compute the standard deviation of the random variable. The paper concludes that the **COMPARE_AND_BALANCE** algorithm converges very fast, but the migration time for an OpenVZ virtual machine was still high.

Milojii et al. [38] prefers process migration to state migration as only current running process is involved in migration. The paper further describes process migration a step forward towards application migration. Process migration needs continuous monitoring of memory pages and copying them iteratively creates an overhead.

Kalim et al. [31] focuses on virtual machine migration across subnets as it proposes a challenge for the purpose of load balancing, moving computation close to data sources or connectivity recovery during natural disasters. The paper points out that conventional approaches use methods like tunneling, routing and layer 2 expansion methods to extend the network to geographically disparate locations, thereby transforming the problem of migration between subnets to migration within a subnet, further increasing complexity and involves human intervention. This paper enables virtual machine migration across subnets with uninterrupted network connectivity. The author proposes decoupling IP address from the notion of transport endpoints as the key for solving problems like mobility. The mechanics involved in such an approach can be defined into following processes:

i) Connection setup: 3 way handshake

ii) Suspension

iii) Resumption

iv) Synchronization: SYN message validation occurs

v) Continued Operation

The paper further evaluates the proposed approach with the existing available solutions and presents a case study in terms of following sub characteristics a) Layer 3 vs Layer 2 Migration b) Downtime and Latency c) Virtual machines coupling with previous state d) Correctness e) Pause and copy migration f) Backward compatibility g) Deployment h) Compatibility with live migration i) Middle boxes and TCP Options j) Network performance and scalability and k) Security considerations. The paper finally concludes that the use of IP addresses as part of flow labels –to identify the transport connection –overloads the notion of network naming and that this is the fundamental reason that inhibits a clean approach for virtual machine migration beyond a subnet. It suggests that decoupling the transport end point naming from IP addresses is not only possible, but is also efficient.

Fischer et al. [25] provides an analysis of the potential failures of services running within virtual machines and how virtual machine migration or replication can be used to address these failures. It further addresses the problem of re-establishing connectivity between a service and its clients upon successful migration, by leveraging results from mobility specially focused on WAN migration. The paper lays down potential issues for virtual machine migration across WAN as the bandwidth between source and destination site is severely limited in comparison to a local connection and with the new location of the virtual machine, the implication of different addresses at the new location, the routing of network traffic for the virtual machine has to be adapted –i.e. a network recovery has to be performed. The author points at the result of wide area migration may result into a mobility problem which may further render the service unreachable unless network recovery is performed. The paper further explains techniques like Indirection where at the application layer, the traffic is forwarded to virtual machine upon successful migration via an application layer proxy remaining at the source network. Second approach at layer 3 is knows as MIP (Mobile IP) that leverages IP mobility. Another approach pointed out was hiding the virtual machine behind a NAT (Network Address Translation) router. The paper points the drawback of using these techniques as the indirection point must remain available in the upcoming communication. The problem can be alleviated by giving DNS record of a server running within the virtual machine a short TTL (Time to Live). After the particular time, all new requests to the virtual machine should arrive at the new location of the virtual machine, there are certain drawbacks in this approach as there are still clients which are connected to the old location and considering the TTL to be short, the DNS cache can still persist longer in several locations in the network. The paper further implements two variants of network recovery after WAN migration

i) **Link Tunneling**: A tunnel between the original access router of a service and its new location is created. To facilitate the migration, several steps have to be performed. First, an additional virtual bridge, named *sshbr*, is created on destination host. Next, a link layer tunnel is created between virtual machine br1 on the access router and the created sshbr. Using this bridged approach, all network traffic can be transparently redirected to the virtual machine. This is verified by the client, who accesses the virtual machine during migration.

ii) **SIP (Session Initiation Protocol) Signaling**: In this approach, the author uses SIP Specific Event Notification (RFC 3265). The SIP server acts as a publish/subscribe system where server mobility events are propagated. Each UA (User Agent) which has subscribed to that kind of event receives a notification when such an event occurs.

For the implementation, the experiment uses the SIP stack in the Python programming language. The paper concludes by analyzing the respective advantages and disadvantages of the mentioned approaches. The link tunneling approach keeps the network configuration of the virtual machine unchanged. This is in particular beneficial for legacy services that cannot be changed to take topological mobility into account. However, the approach requires access to the access router of the virtual machine at its original location. This can be a problem if the access router is affected by the network challenge itself (e.g. in the case of a natural disaster). In contrast, the SIP signaling approach indicates how network recovery on the application layer can mask an IP address change. This is beneficial, as traffic is routed more directly, and does not require a cooperative access router. However, the migration is not transparent to the application anymore. Both approaches achieve a downtime below one second –which is significantly better than no action and should be acceptable for most services.

Considering the WAN migration, Wood et al. [49] states that virtual machine migration in WAN gets complicated when the size of the virtual machine exceeds 50 GBs, the paper also points out I/O processing as a bottleneck during virtual machine migration.

Jin et al. [30] presents the design and implementation of a novel memory compression based virtual machine migration approach (MECOM) that first uses memory compression to provide fast, stable virtual machine migration, while guaranteeing the virtual machine services to be slightly affected. Based on memory page characteristics, the paper introduces an adaptive zero aware compression algorithm for balancing the performance and the cost of virtual machine migration. The paper first exploits memory compression technique to minimize the downtime and total migration time during live migration and also analyses memory page characteristics. It uses adaptive memory compression approach to tune migration performance of virtual machines. The author states compression as a significant factor in improving the performance of live migration as it indirectly augments network bandwidth for migration. Compressed dirty pages take shorter time to fly on the network. Also the network traffic drops dramatically when less amount of data is transferred between source and target nodes. The compression algorithm introduced needs to be lossless as the compressed data requires to be exactly reconstructed and the overhead of memory compression should be as small as possible so that it does not affect the live migration. The CBC (Characteristics based Compression) algorithm introduced in the paper uses multithreading techniques to parallelize compression tasks and the number of threads is tuned to provide best performance. All the experiments performed are based on Xen 3.1.0, the paper concludes that by using the described system, the process can get a better average performance than Xen: up to 27% on virtual machine downtime, up to 32% on total migration time and up to 68.8% data cut down that must be transferred.

Ibrahim et al. [29] states that continuous page monitoring for modification causes an increase in transitional look aside buffer (TLD) flusher and soft page faults on write operation which further affects the performance of the application. Furthermore, continuous copying of pages to I/O buffer may rise to memory contention. In fault tolerant scenarios, process migration is not considered a viable option.

Laadan and Nich et al. [32] underlined the process isolation of container type virtualization as a reason for application independence. i.e. running same applications of different versions on same physical machine. The paper further explains private virtual namespace which is primarily responsible for achieving isolation.

Mirkin et al. [39] presents the check pointing and restart feature for containers in OpenVZ. This feature allows one to checkpoint the state of a running container and restart it later on the same or a different host, in a way transparent for running applications and network connections. Check pointing and restart are implemented as a loadable kernel modules in addition to a set of user space utilities. Since a container is an isolated entity, its complete state can be saved into a disk file-the procedure known as check pointing. The same container can then be restarted back from that file. Check point and restart also make it possible to move a running container from one server to another without a reboot. The author describes the check pointing in containers as a three stage process:

i) Freeze process: Move processes to previously known state and disable network

ii) Dumping: Collect and save the complete state of all the containers processes and the container itself to a dump file

iii) Stop the container: Killing all the processes and unmount containers file system

Like checkpointing, author also describes restart procedure being vice versa of check pointing:

i) Restarting the container: Creating a container with same state as previously saved in dump file

ii) Restart process: Creating all the processes inside the container in frozen state, and restore all of the dependent resources from the dump file

iii) Resume the container: Resuming processes execution and enable the network.

Post this process, the container continues its normal execution. Paper states that by using check pointing and restart feature, the live migration process becomes easier, it implements a simple algorithm that has no external dependency on any additional hardware like SAN. The algorithm can be summarized in a process flow as:

i) Containers file system synchronization: Transferring containers file system to destination server using rsync utility

ii) Freezing the container: Freezing all processes and disabling networking

iii) Dump the container: Collecting all resources and saving them to a file on disk

iv) Second containers file system synchronization: During the first synchronization, a container is still running, so some files on the destination server can become outdated. So second synchronization should be performed only post container freezing

v) Copy the dump file: Transfer the dump file to the destination server

vi) Restart the container on the destination server: Create a container on the destination server and creating processes as saved in dump file

vii) Resume the container: Resuming containers execution on destination

viii) Stopping the container on the source server: Killing the containers processes and unmounting its file system

ix) Destroying the container on source server: Removing containers file system and config files on source server

The paper further identifies stages iii-vi responsible for the most delay in service, the experiments also point out that second file system sync time and dump file copying time are responsible for about 95% of all the delay in service. The paper further lays down possible optimization's for this problem as i) Second file system sync optimization: Decreasing the number of data being compared during the second sync ii) Decreasing the size of a dump file by using either lazy migration or iterative migration. The paper concludes by using check pointing and restore optimization's, a considerable decrease in the delay of a service has been achieved.

Romero and Hacker et al. [44] also use Open VZ to explain the three step process of live migration:

i) Container suspension followed by memory dumping

ii) The dumped memory is copied to destination host

iii) The migrated container is resumed and source container is shut down followed by memory cleansing

The paper also compares live migration approaches for parallel applications and concludes by using optimization's like check pointing can affect the application performance, contrasting to the view of Mirkin [39]. Various literature's are available which implement check pointing for efficient application portability. E.g. The Condor Project [2] and CRIU [3]. The CRIU check pointing method is also explained in Chapter 6.

Hadley et al. [27] presents Multi-Box, a lightweight container technology that facilitates flexible vendor-independent migration. The framework introduced in the paper defines workload fluidity as the capability to migrate one or more applications from one CSP (Cloud Service Provider) to another at short notice with little or no human intervention. The paper introduces a method of achieving workload fluidity by utilizing extremely lightweight containers based on control groups (cgroups). The author describes Multibox container as a versatile execution environment supporting various processes, which are unaware of the remainder of the system. By using cgroups, a container is created as a namespace within the host to isolate its own processes, system, network devices and the file system. Multibox containers are managed by a container manager that creates the namespace, routes network traffic to the host and its containers and allocates resources to the container to support running applications. It also facilitates user interaction with the namespace. During implementation, kernel support is needed to run control groups within a virtual machine. The lightweight cgroups package provides this functionality that can be compiled into a standard 3.x kernel. To facilitate easy deployments, the author compiled Kernel 3.14.22 with support for control groups. The support for para virtualised environments is enabled to ensure compatibility with a larger number of CSPs. The built kernel was compiled into an RPM package to enable easy installation on RedHat Linux systems such as those running RHEL and CentOS. The container manager is also implemented as an *init* script supporting the commands like *prep*, *create*, *boot*, it sync and *resync*. The Multibox work flow can be explained as shown in figure 3.1. The paper evaluates its framework in terms of migration of stateful and stateless applications. In the case of stateful application, a client connected to the game server was able to reconnect within a few seconds without changes to the connection settings or the game's state. In the case of the stateless application, a client browsing the hosted website did not notice any interruption. The paper concludes that the performance overhead of deploying within a lightweight container is 4.90% of the resources available to an average virtual machine and downtime during a migration is less than the time needed to scale a server using provider-centric tools. The author finally remarks MultiBox containers of great advantage as they are transferable to any CSP infrastructure that supports any Linux variant. Meaning no more CSP cooperation is required, which is a ground breaking advancement in the area of cross-cloud computing. Furthermore, MultiBox containers are lightweight by design and migrating them is significantly more resource efficient than other cloud workload migration approaches.

Wood et al. [50] points out in all current implementations, detecting a workload and initiating a migration is handled manually, this paper introduces automated **Black box** and **Gray box** strategies for virtual machine migration in large data centers. Black box technique automates the task of monitoring system resource usage, hotspot detection, determining a new mapping and initiating necessary migrations. All these necessary decisions are made entirely based on observing each virtual machine from outside and without any knowledge of application resident within each virtual machine. On the other hand, gray box technique assumes access to small amount of operating system level statistics in

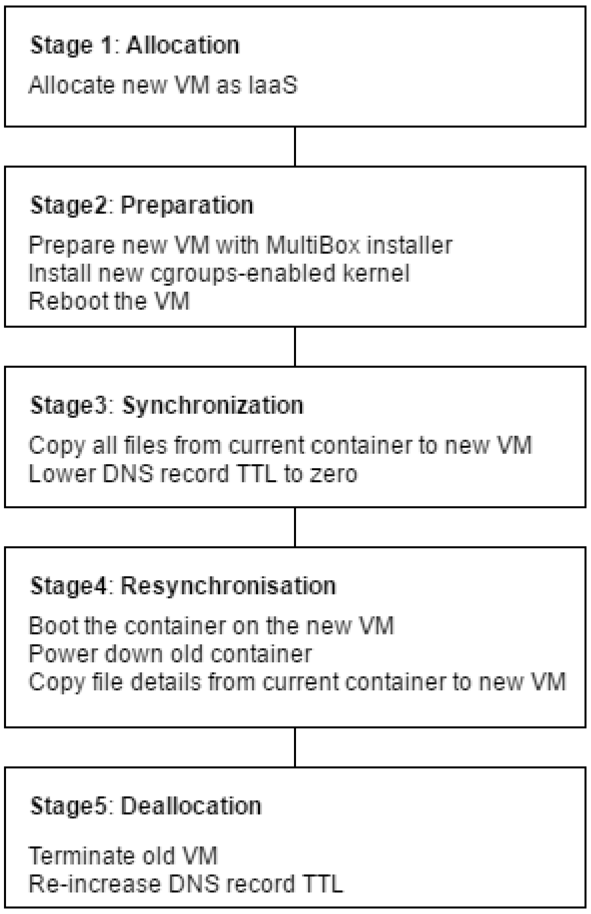

Stage 1: Allocation
Allocate new VM as IaaS
Stage2: Preparation
Prepare new VM with MultiBox installer
Install new cgroups-enabled kernel
Reboot the VM
Stage3: Synchronization
Copy all files from current container to new VM
Lower DNS record TTL to zero
Stage4: Resynchronisation
Boot the container on the new VM
Power down old container
Copy file details from current container to new VM
Stage5: Deallocation
Terminate old VM
Re-increase DNS record TTL

Figure 3.1: The MultiBox work flow.

addition to external observation to better optimize the migration algorithm. This paper introduces **Sandpiper** which implements a hotspot algorithm that determines when to migrate a virtual machine and a hotspot mitigation algorithm that determines what and where to migrate and how much to allocate after the migration. The detection component employs a monitoring and profiling engine that gathers usage statistics on various virtual and physical servers and constructs profiles of resource usage. Upon detection, Sandpipers migration manager is invoked for hotspot mitigation. The migration manager employs provisioning techniques to determine the resource needs of overloaded virtual machines and uses a greedy algorithm to determine a sequence of moves or swaps to migrate over-loaded virtual machines to under loaded servers. Results point out that Sandpiper can alleviate single server hotspots in less than 20s and more complex multi-server hotspots in few minutes. Results show that Sandpiper imposes negligible overheads and that gray-box statistics enable Sandpiper to make better migration decisions when alleviating memory hotspots. Sandpiper runs a component called the *nucleus* on physical server; the nucleus runs inside a special virtual server and is responsible for gathering resource usage statistics on that server. It also employs a monitoring engine that gathers processor, network interface and memory swap statistics for each virtual server. For gray-box approaches, it implements a daemon within each virtual server to gather OS-level statistics and application logs. The nuclei periodically relay these statistics to the Sandpiper control plane. The control plane runs on a distinguished node and implements much of the intelligence in Sandpiper. It comprises three components: a profiling engine, a hotspot detector and migration manager. The monitoring engine is responsible for tracking the processor, network and memory usage of each virtual server. It also tracks the total resource usage on each physical server by aggregating the usages of resident virtual machines. It maintains a usage history for each server, which is then used to compute a profile for each virtual and physical server. Three black box profiles are introduced on per virtual server basis: CPU utilization, network bandwidth utilization and swap rate. To ensure that a small transient spike does not trigger needless migration, a hotspot is flagged only if thresholds or SLAs are exceeded for a sustained time. Sandpiper employs a time series prediction technique to predict future values and uses Xen to implement it. In short, Blackbox is responsible for estimating peak CPU, memory and network bandwidth needs while Gray box is responsible for peak CPU & network needs. The migration process can be explained in following phases:

i) Hotspot Mitigation: Once a hotspot has been detected and new allocations have been determined for overloaded virtual machines, the migration manager invokes its hotspot mitigation algorithm. This algorithm determines which virtual servers to migrate and where in order to dissipate the hotspot

ii) Capturing multi-dimensional loads: Involves calculating volume of a virtual machine

iii) Migration Phase: Involves querying the physical machine with high volume servers and moving the machines with least volume where the VSR (Volume to Size Ratio) is satisfied, the algorithm prepares a list of overloaded virtual machines and a new destination server for each

iv) Swap Phase: Deals with the situation when there are not sufficient idle resources on less loaded server. Algorithm exchanges a high VSR virtual machine from a loaded server with one or more low VSR virtual machines from an under loaded server

The Sandpiper nucleus is a Python application that extends the XenMon CPU monitor to acquire network and memory statistics for each virtual machine. The monitoring engine in the nucleus collects and reports measurements once every 10 seconds –the default measurement interval. The nucleus uses Xens Python management API to trigger migrations and adjust resource allocations as directed by the control plane. While black-box monitoring only requires access to domain 0 events, gray—box monitoring employs two additional components: A Linux OS daemon and an Apache module. The paper concludes that Sandpiper is able to resolve single server hotspots within 20 seconds and scales well to larger data center environments thereby proving a breakthrough in dynamic migration across various data centers.

Chapter 4

Density

In the previous chapter, we summarized the state of art for virtualization technology, including all modern and legacy virtualization techniques. In this chapter, we will describe the design, implementation and evaluation of virtual machines and containers by considering density as a parameter and taking into account their performances in terms of scalability. This chapter further gives us a brief description of our approach and motivations behind the architecture design, benchmark characteristics and configurations. The later section evaluates the density for both virtual machines and containers, and deduces a conclusion based on their respective performances under same test conditions.

4.1 Introduction

In terms of virtualization, density can be referred as the total number of virtual machines or containers that can be run on an underlying host. In recent times, one of the primary parameters of a virtualization technology is the virtual CPU to physical CPU ratio. VMWare for example puts forward the notion that total TCO of virtualization technology must consider virtual machine density. The general measure in this context is: The more virtual machines per physical server, the better. Virtual machine density directly influences the IT efficiency whether you measure it in terms of *cost/application* or *cost/user*. Another aspect of density is that the management cost increases linearly with respect to density, which further results in decrease of efficiency. Such increase in density results in higher management cost because of increase in complexity imposed by increase in the number of virtual machines and containers especially at the network layer. DHCP (Dynamic Host Configuration Protocol), DNS etc. are all impacted and not necessarily favorably by the increase in density.

One of the important objective of achieving high density is achieving it without affecting the performance of other guests already running on the same physical host. A data center operator benefits only if the guest technology is scalable, does not impact peer guest performances and is easy to manage. As a result, it is vital to analyse how the increase in number of guests affects the performance of other guests running on the same physical infrastructure.

This chapter quantifies the deterioration of performance for containers and virtual machines by running them simultaneously on the same physical host. Running more guests simultaneously causes a lot of resource contention and swapping among the guests running on the same physical host. In order to avoid high overheads, an optimal trade off should be maintained between density and complexity. To analyse the impact of density among the peers we measure the performance in terms of:

i) CPU Performance

ii) File I/O Performance

iii) MySQL Server Performance

We gather these performance parameters as we simultaneously increase the number of guests per host. All guests run the same workload under same configurations to avoid any ambiguities. We use this methodology based on the need of spinning up multiple instances of an application at a particular site during a data center outage.

4.2 Benchmarking

Sysbench [5] is a modular cross platform and multithreaded benchmark tool for evaluating operating system parameters that are vital for a system running database related intensive loads. Sysbench currently provides features that help us to measure the following system parameters:

- File I/O performance
- Scheduler performance
- Memory allocation and transfer speed
- POSIX threads implementation performance
- Database server performance

For our experiments, we will perform the CPU stress testing which in our case performs an intensive prime number calculation. All calculations are performed using 64 bit integers. Each thread executes the requests concurrently until the total number of requests or total execution time exceeds the limit as specified in the *run* command.

Secondly, we will use the file I/O test mode, which is used to produce various kinds of I/O workloads. The test consists of two stages:

i) *Prepare*: At this stage, sysbench creates dummy test files with a specified size. It is very vital to ensure size of test files should always be higher than the RAM of the test machine, in order to avoid memory caching affecting our result sets.

ii) *Run*: During this stage, test execution initiates with respect to the number of threads specified. Each thread performs specified I/O operations on the set of files.

Several I/O operations are supported such as *seqwr* (sequential write), *rndrd* (random read), *rndrw*(random read/write) etc.

Lastly, we use the *oltp* test mode, which is used to benchmark a real database server performance. The motivation behind using this test scenario is that majority of the guests running in a modern data center facility are usually used to host heavy databases such as MySQL and Cassandra. Just like file I/O test mode, this test also has two stages:

i) *Prepare*: In this phase, we create a test table after MySQL installation with a specified number of rows

ii) *Run*: During this stage, we simply execute the test by explicitly specifying parameters such as mode, time, request filtering etc

For all our experiments, we will be using sysbench version 0.4.12.

4.3 Experimental Setup and Configuration

Table 4.1 lists the configuration of the machine on which all our experiments were performed.

4.4 Technologies

With respect to our experiment, choosing an appropriate guest technology was vital. Considering all our experiments were performed on a host running Linux operating system, it was relevant to choose a virtualization product that natively supports Linux operating system and mimics its underlying architecture as well. For virtual machines, we use KVM (Kernel Virtual Machine) with QEMU (Quick Emulator) and libvirt while for containers we use Docker which is implemented using Linux features such a cgroups and namespaces.

4.4.1 KVM,QEMU & LIBVIRT

For virtual machines, hypervisor serves as the core of virtualization, in case of our experiment, QEMU and KVM both act as a hypervisor. Since QEMUs performance tends to get slower in case of systems that lack hardware virtualization, KVM thereby helps QEMU to access hardware virtualization features efficiently on different architectures by adding an accelerating feature. KVM is a Linux kernel module, that enables the mapping

CPU Name	Intel (R) Core (TM) i7-4770 CPU@3.40 GHz
Type	Hyper Threaded Processor
Architecture	X86-64
Cores per socket	4
Threads per core	2
Total number of CPUs	8
Machine Name	Dell-Optiplex 9020
RAM	8 GB
Operating System type	64 bit
HDD Capacity	500 GB
Disk Type & RPM	ATA - 5400
Model Number	ST500LM000-1EJ162
Host Operating System	Linux 16.04
Guest Operating System	Linux 14.04.4
Kernel Version	4.4.0-28-generic

Table 4.1: System Configuration

of physical CPU to virtual CPU. This mapping provides the hardware acceleration of virtual machines and boosts its performance. In our case, QEMU uses this accelerating agent when we set the *virttype* feature as KVM. When used together, QEMU behaves like a hypervisor and KVM as an accelerating agent. We also use libvirt which is a virtualization management library that manages both KVM & QEMU. Libvirt consist of three utilities:

i) API library

ii) Daemon (libvirtd)

iii) Command line tool (-virsh)

To minimize the performance overhead, modern processors support virtualization extensions e.g. Intels VT–X. Using such technologies, a slice of physical CPU can be directly mapped to the virtual CPU. Hence the instructions meant for virtual CPU can be directly executed by physical CPU slice therefore improving the execution speed. If server CPU does not support such virtualization extensions, then the hypervisor is responsible for executing the virtual CPU instructions using translation. QEMU uses TCG (Tiny Code Generator) to optimally translate and execute the virtual CPU instructions on physical CPU. Also, KVM only runs on x86 CPU architectures. The kernel component of KVM is included in all mainline Linux distributions while we use the following versions of packages for our experiments:

QEMU Version: 2.7.0

libvirt Version: 2.1.0

The configurations of our guest virtual machines are as under:

RAM allocated: 1024 MB

CPUs allocated: 1 out of 8 available

HDD space allocated: 25 GB

4.4.2 Docker

As explained already in chapter 2, we use Docker for performing all experiments with containers. We will be using Docker version 1.12.0 for all our tests. We choose Docker over all other container products because it is built on top of LXC (Linux Containers) and provides a clean, manageable and portable form of containerization. Due to the darn popularity of Docker over the last couple of years, it is now almost used by every large IT infrastructure organization, so evaluating a technology of today would further add a value to our final analysis.

4.5 Test Parameters

To evaluate the density of both virtual machines and containers, we evaluate the following parameters to establish a fair comparison between the two. All the specified experiments were performed 10 times to avoid any errors in the experimental numbers. All values were averaged for the resultant evaluation. Further graphs for 12 or more running containers are also plotted along with the evaluation graphs only to analyse the performance trend individually and has no role in direct comparison to that of virtual machines.

4.5.1 CPU Performance

We measure the CPU performance of virtual machines and docker containers with the configuration as depicted previously. The benchmark runs a single thread to calculate the specified prime numbers. The test verifies the prime numbers by dividing the numbers with sequentially increasing numbers and verifying the remainder for zero value. In our current analysis, we calculate 30000 prime numbers and the test scenario remains same in all the cases. The test is initially done on a native Linux host with no guest technology running. The same test is repeated on a virtual machine followed by incrementally increasing the number of virtual machines up to 8 on the same physical host. The scaling quotient of our experiment was restricted to 8 virtual machines and 12 Docker containers. By incrementing the number of guests, we measure the total time required by each guest to perform the operation. With total time, we also evaluate the per request statistics of the running guests such as minimum, average and maximum time required to process a single request. The thread fairness is also considered as a parameter for evaluation. In our iterations for both virtual machines and containers we use a single thread for all our calculations. A test image file was created with all the configurations and desired dependencies installed [6].

Evaluation and Results

The CPU performance of virtual machines and containers primarily depend upon the following two parameters:

i) Total time required to finish a particular task

ii) Per request statistics of the process where we evaluate the minimum, maximum, average and 95 percentile of the time required to process a single request

We plot the graphs with respect to the native performance and virtual machine performance (increasing the numbers incrementally) as well as native and Docker container performance separately. Since the calculation of prime numbers in both the cases is same (i.e. 30000). The total time required therefore directly depends upon the availability of CPU to the process. By incrementing the number of guests on the underlying host we find a pattern in terms of process time.

As we can clearly infer from figure 4.1, the time required to finish the process on a native Linux machine was timed around 35.94 seconds. As far as the virtual machines are concerned, the performance of CPU deteriorates slowly as we incrementally increase the number of guests. The delay is still not very high considering the number of guests running, we can still observe an increase of around 7.06 seconds when running a single instance to running 8 guests simultaneously. In case of containers, the performance is comparable to its counterpart. For 8 running containers, the process takes 42.05 seconds

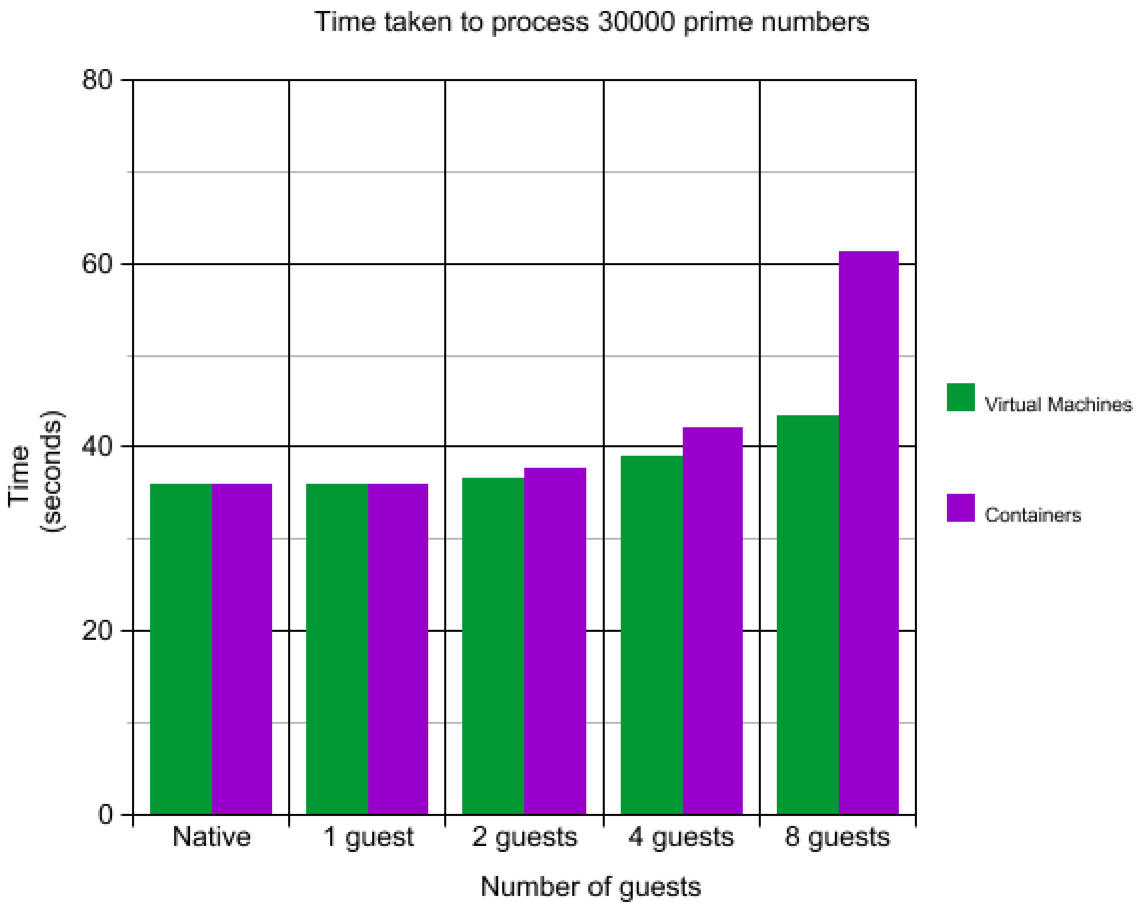


Figure 4.1: Total Processing Time.

to finish which is approximately equal to the performance delivered by virtual machines.

In case of per request statistics, we evaluate the performance of virtual machines and containers with respect to table 4.2 to establish a clear performance degradation trend between the two.

As seen from figure 4.2 & 4.3, the minimum and average time required to process a request is fairly same in both virtual machines and containers i.e. clocked at 3.67 ms and 4.45 ms for 8 active running guests. The 95 percentile value also suggests towards the fact of the performance being nearly same. The only parameter that differs between the two is the maximum time, there is a significant performance boost in case of Docker, where the time required is almost half of what is required in Linux virtual machine, this particular parameter may not contribute to the overall aspect of the performance but in case of

Minimum Time	3.58 ms
Average Time	3.59 ms
Maximum Time	9.07 ms
Approximating to 95 percentile	3.62 ms

Table 4.2: Native Performance Profile

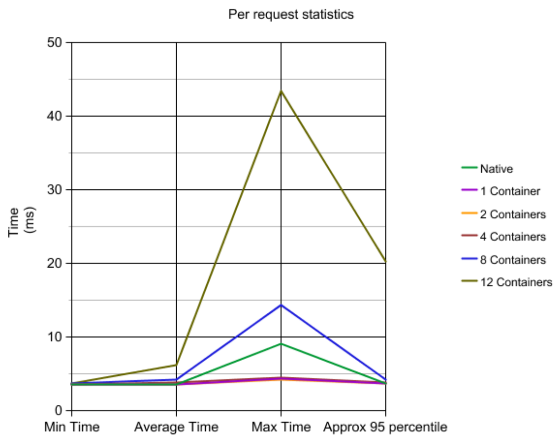


Figure 4.2: Per request statistics of Containers.

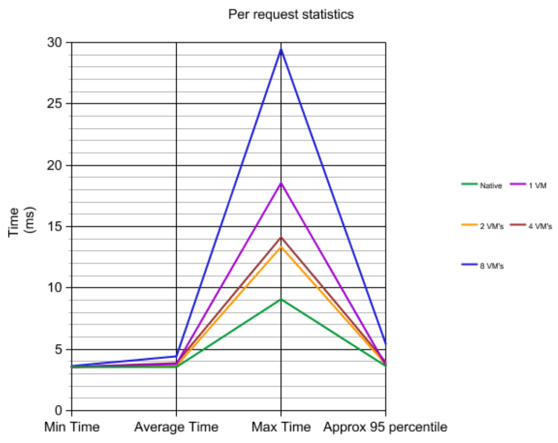


Figure 4.3: Per request statistics of Virtual Machines.

workloads which involve processing huge numbers and pattern matching, containers may just win the bout over its counterpart.

The evaluation of CPU performances of both Docker containers and virtual machines leads to the conclusion that the CPU performance is comparable in both the technologies. The performance degradation factor is also same when it comes to scaling up the number of guests. From a data center operator point of view, there is no huge performance incentive for choosing one of them and the choice should entirely depend on the functional and non-functional requirements of the workload to be processed.

4.5.2 File I/O Performance

We measure the file I/O performance of both Linux machines and Docker containers by preparing appropriate test files. On a native system, we prepare 80 GB test files to calculate the file I/O performance, we maintain a 1:10 ratio between the test file and RAM. For all the tests on Linux virtual machines, we create 10 GB test files as the RAM allocated to each virtual machine is 1 GB. In case of Docker containers, as there is no explicit allocation of RAM to the container, we again use 10 GB test files to set a fair comparison with respect to its counterpart. We configure and set up all test files on a single Linux virtual machine and clone it for experimenting with multiple virtual machines using libvirt utility, thereby decreasing manual work involved in the setup. Vagrant [9] was also an option to spin up multiple virtual machines but we ignore it as vagrant only supports Virtual Box [10] natively, in order to use it for KVM we have to download a separate plugin that uses a command *mutate* which converts the boxes from virtual box format to KVM compatible format. This conversion will affect the performance analysis of our experiment, libvirt further uses full cloning opposed to that of linked cloning, for qualitative output from our benchmark. In case of Docker containers, we prepare an image with all the dependencies and test files and push it to the repository, the image can be found here [7].

Considering the time involved in file I/O operations, we set the maximum time parameter as 300 seconds in all our respective cases. We also use random read & write and synchronous mode for our analysis. The file I/O operations are performed initially on a native Linux machine followed by linear increase in number of guests (1 to 8 for KVM and 1-12 for Docker containers). Using this test, we evaluate parameters such as total number of events performed, number of requests processed per second, speed of execution and per request statistics such as average time, maximum time and approximating request parameters to 95 percentiles. We also deduce specific file I/O parameters such as number of reads and writes.

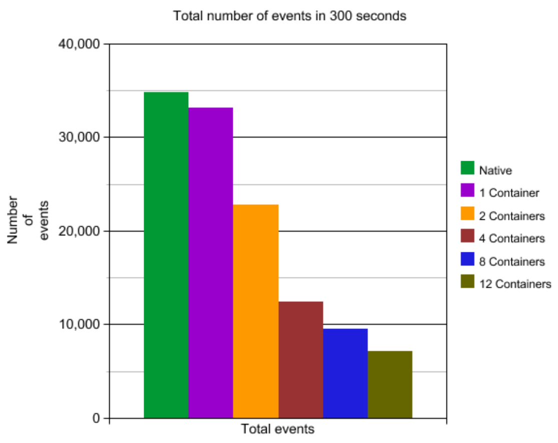


Figure 4.4: Number of events in Containers.

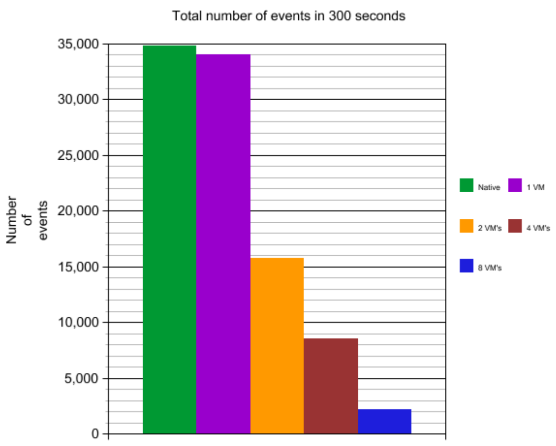


Figure 4.5: Number of events in Virtual Machines.

Evaluation and Results

To evaluate the file input output performance, we consider the following parameters for our experiment:

i) Total number of events executed in a given time interval

ii) Number of reads and writes performed

iii) Speed of execution

iv) Per request profile

v) Number of requests processed per second

We consider these 5 parameters to be vital in a fair evaluation of the two technologies considering the variety of file I/O workloads a system can be exposed to; these parameters are almost the driving factor in all of them.

As seen from figure 4.4 & 4.5, the total number of events processed in case of native was 34854, all our further analysis is in reference to this number. When performing the same operation in both containers and virtual machines, we find out that the performance remains same when a single guest instance is running but as we increase the number of guests, the events executed in case of virtual machines decreases drastically. As from figure 4.4, we can clearly see the events executed almost decreases to half when we incrementally increase the number of guests from 1,2,4 to 8. While in case of Docker containers, the event

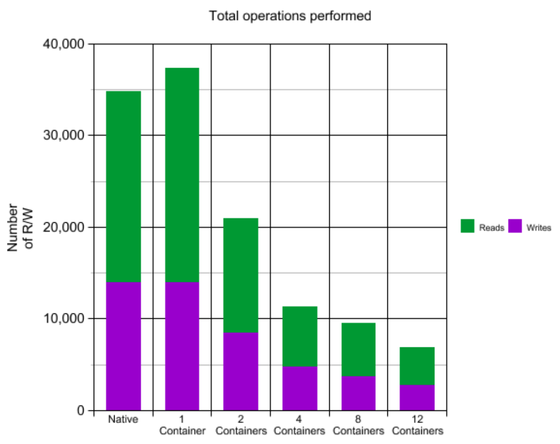


Figure 4.6: Total R/W operations in Containers.

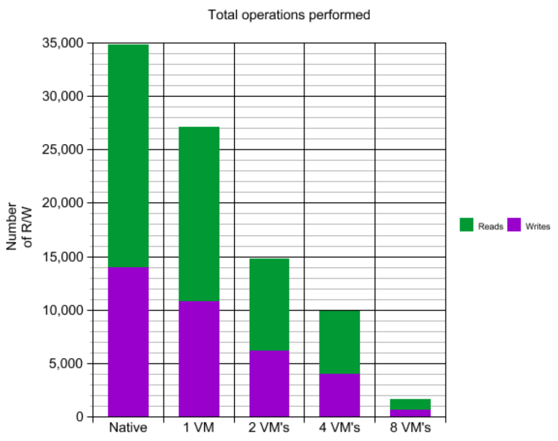


Figure 4.7: Total R/W operations in VMs.

execution decreases by a factor of 0.25% on an initial increase in guests but almost drops marginally when increasing the number of guests further. The degradation of performance in virtual machines is huge when scaling is a requirement.

The total number of reads and writes performed is a driving factor for the eventual conclusion on the performance parameters as we can accordingly use a virtualization technology based on workload characteristics whether it requires high reads or write transactions. Again for our evaluation, the native performance is noted as: Reads = 20912 and Writes = 13941. As we can see from figures 4.6 & 4.7 above, the factor between the difference of reads and writes remains constant throughout our iterations but as suggested earlier the performance of virtual machines degrades drastically as the number of guests are increased similar to what we inferred in last section. For 8 simultaneous running guests, we observe the performance of both reads and writes decreases with a ratio of 1:6, which is surely a big variation considering the scaling and vast data sets involved.

Speed of execution or the speed at which it performs the file I/O operation is one of the most important factor in determining file I/O performance. As we can clearly infer from figure 4.8 and 4.9, the speed of execution on native is approximately around 2.01 Mbps. The speed curve also shows a similar trend like the previous two sections, it decreases linearly as the number of guests increase. The curve in case of virtual machines

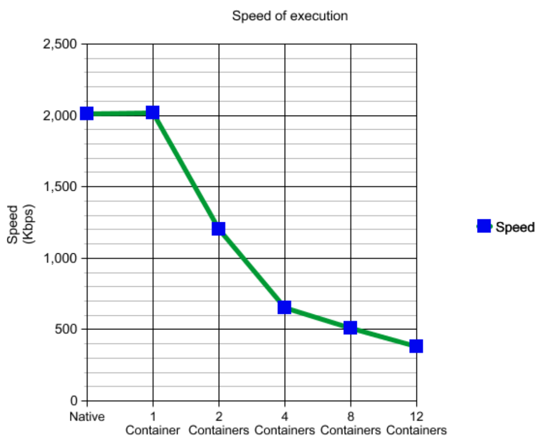


Figure 4.8: File I/O speed variation in Containers.

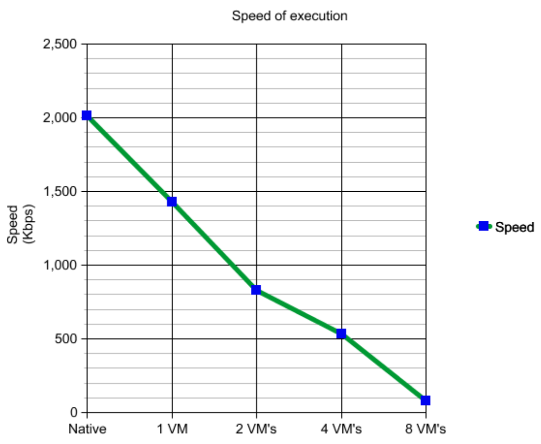


Figure 4.9: File I/O speed variation in VMs.

is sharp and performance degradation is directly proportional to the number of guests added. In case of Docker containers, after an initial linear performance degradation, the curve almost approaches a constant value. E.g. The speed with 8 guests running in case of virtual machines is 77.42 Kbps while on the other hand Docker containers deliver a speed of 512 Kbps for similar number of guests. The difference between the two is relatively high and Docker certainly proves to provide better performance than KVM. The difference can be accredited to the dynamic allocation of memory resources in containers to that of fixed allocation in virtual machines. Sharing the same system resources with the host operating system helps containers to achieve high performance for reads & writes. There is a similar graph trend for the request per second parameter as well. Figure 4.10 & 4.11, in this case the Docker containers almost process 6 times more requests than its counterpart in a second which certainly contributes to the superior performance of Docker containers over KVM.

At the end, the per request profiles are referenced again with respect to the native performance as shown in table 4.3.

Figure 4.12 & 4.13 point out to the conclusion of Docker containers delivering better performance than virtual machines. The distribution of lines on the time axis for maximum time, average time and approx. 95 percentile value clearly points out the superior

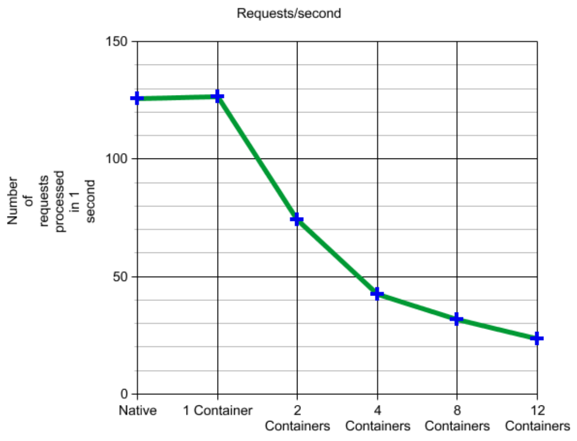


Figure 4.10: Number of Reqs. processed per second in Containers.

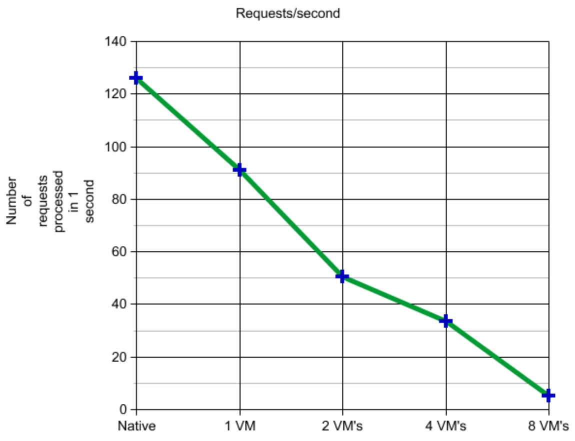


Figure 4.11: Number of Reqs. processed per second in VMs.

Minimum Time	3.58 ms
Average Time	3.59 ms
Maximum Time	9.07 ms
Approximating to 95 percentile	3.62 ms

Table 4.3: Native Performance Profile for File I/O

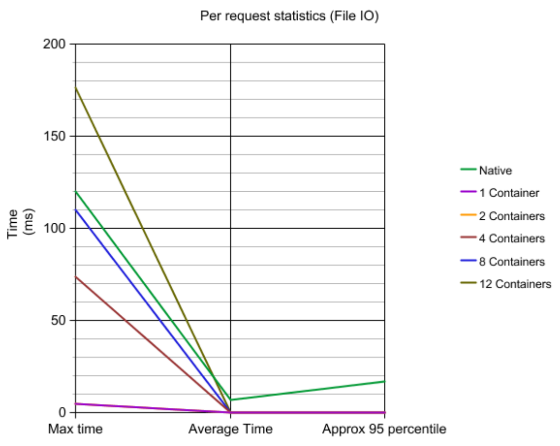


Figure 4.12: Per Request stats of Containers.

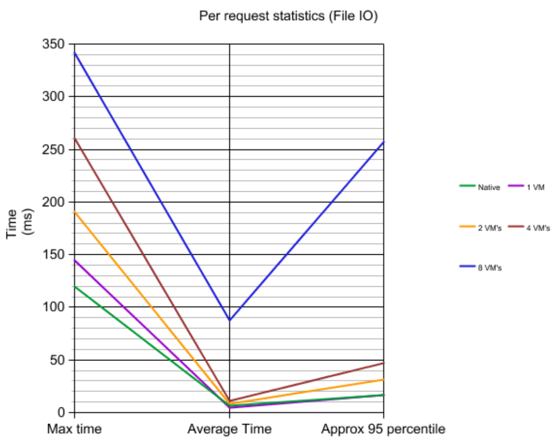


Figure 4.13: Per Request stats of Virtual Machines.

performance of Docker containers. As we see, the maximum time is evenly distributed and varies as we increase the number of guests, the distribution may be similar but the number involved almost differs by a factor of 3, the performance in terms of timings is also noted to be 3 times less than that of virtual machines. Also it is interesting to observe the blue line (8 VMs) drifting so sharply from the mean average time while for the same number of guests, the average time for Docker containers remains around the mean mark. This clearly indicates the performance of containers is efficient when the number of guests are increased.

From evaluating all the five vital parameters, its fair to conclude that Docker containers offer better performance in all aspects of file I/O operations but taking into account the fact that Docker containers almost offer near native performance, there is no room for further future optimization's while in the case of virtual machines, the I/O can be improved by using many optimization's such as by enabling hardware assisted virtualization or using optimal memory swapping and paging techniques. However, the vast difference in performance parameters can still not be ignored and virtual machines would have to come a long way in matching the performance of containers. At present, containers surely seem to be the appropriate choice for all data center operators running heavy file I/O tasks. E.g. Hadoop workloads.

4.5.3 MySQL Performance

We measure the database server performance by using MySQL. We install all the dependencies alongside creating a dummy test table with 1,000,000 rows. The same methodology of cloning and Docker image push is used to automate configuration and setup across all the guests. The MySQL Docker image can be found here [8]. In this experiment, we use 6 threads to perform the computations concurrently, which again remains constant throughout the whole process. The login credentials to MySQL server are passed as an argument in the command line terminal for process automation but is not encouraged in production environment because of data privacy regulations. The explicit parameters used in all our tests are:

Read only mode = OFF

Maximum Time = 60 seconds

Maximum request = 0

Using this test, we evaluate relational database specific statistics such as number of transactions performed, number of reads and writes, total number of events and per request statistics like in all our earlier cases. We use MySQL version 5.7.14 for all our analysis.

Evaluation & Results

To evaluate the MySQL server performances in both virtual machines and containers we evaluate the following parameters to propose a final conclusion:

i) Number of Reads, Writes & Transactions performed

ii) Number of individual events executed

iii) Per request profile

Establishing an evaluation profile in terms of reads and writes was important for our analysis as most of the servers in data centers usually perform database transactions and depending on the particular workload, number of reads and writes to a database greatly quantifies our results.

As seen from figure 4.14 & 4.15, it is evident that the performance of Docker containers quite easily surpasses to that of virtual machines. The steep linearity in the curve points out at the degrading performance of virtual machines as the numbers goes high but in case of Docker containers, the first linear decrease is followed by a smooth curve which hints at its efficiency in terms of scaling the MySQL server. The container running 8 guests performs 27 times more reads and 14 times more writes than its counterpart. Same trend is followed for the number of transactions executed, where Linux virtual machine processes almost 2 and a half transactions per second compared to that Docker executes transactions at the speed of 30.5 transactions/second which is quite a large number considering the overall performance balance.

To add to the above analysis, the number of events executing under different number of active running guests also follow the same trend. From figure 4.16 and 4.17, we observe that the number of events significantly reduce to almost half as the number of guests are increased incrementally. The Docker almost turns out to have equal bar length as the number of containers are increased. The contrast in the numbers can be clearly identified when both of them run 8 guests simultaneously, virtual machine executes 176 events

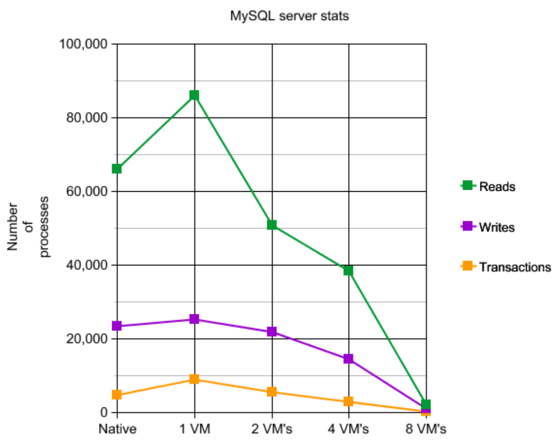


Figure 4.14: R/W and Transactions performed by Containers.

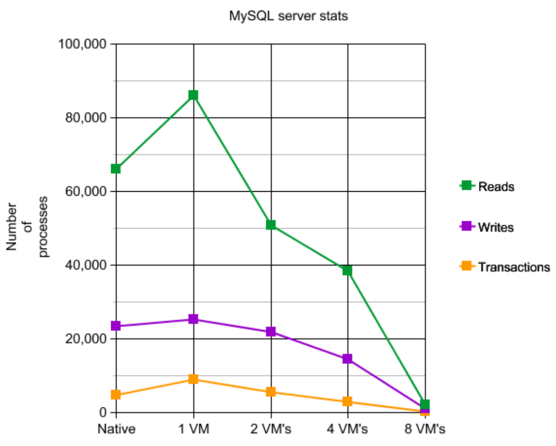


Figure 4.15: R/W and Transactions performed by VMs.

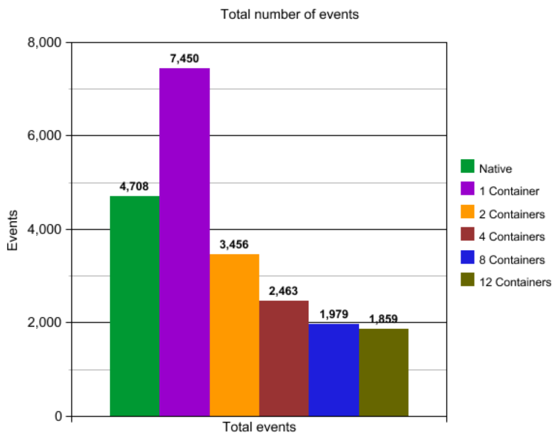


Figure 4.16: Total number of events executed in Containers.

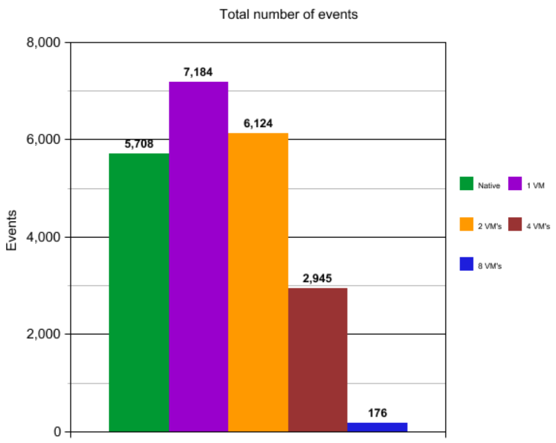


Figure 4.17: Total number of events executed in VMs.

while container executes 1979 which further reduces to 1859 when running 12, from these result sets we can infer that scaling up containers does not affect the performance as compared to virtual machines. This may be primarily because of the design architecture of the containers with respect to virtual machines where resources and access rights are dynamically allocated with respect to process requests.

Lastly, analysing the per request statistics of both containers and virtual machines, we see that the minimum and average time are almost concentrated at a particular time interval which is comparable to both virtual machines and containers but as soon as the virtual machine deployments is increased to 8, the minimum time, average time,

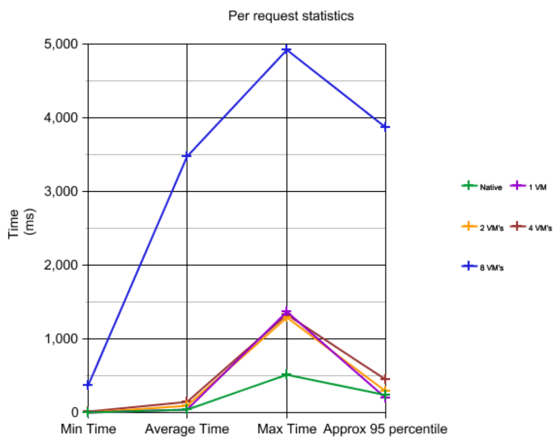


Figure 4.18: Per req. stats of Containers running MySQL server.

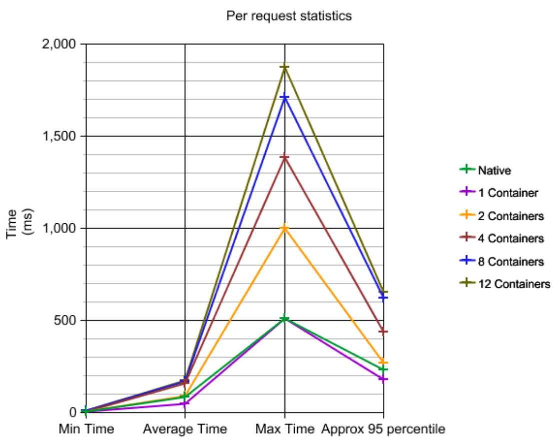


Figure 4.19: Per req. stats of VMs running MySQL server.

Minimum time	369.76 ms
Average time	3472.17 ms
Maximum time	4918.05 ms
Approx. 95 percentiles value	3872.24 ms

Table 4.4: Per req. stats of 8 running VMs

maximum time and approx. 95 percentiles value line shoots away drastically from the mean indicating a sharp performance degradation. This performance dip is experienced because of over provisioning the underlying resources, as the memory swapping process between virtual machines introduces additional latency. Although the maximum time required to process a single request increases gradually in containers but the factor of increment is not even comparable to that of virtual machines. While running 8 guests, the performance delivered by virtual machines and Docker containers are tabulated in tables 4.4 & 4.5 respectively

Clearly the performance of MySQL server in Docker containers is much better to that of virtual machines. As we are using Ubuntu 14.04 as our base image for implementing MySQL server, but in case of Docker containers, the performance can be enhanced by using MySQL image directly as our base image. We have used Ubuntu 14.04 as our base

Minimum time	369.76 ms
Average time	3472.17 ms
Maximum time	4918.05 ms
Approx. 95 percentiles value	3872.24 ms

Table 4.5: Per req. stats of 8 running Containers

image to set a fair comparison profile for both containers and virtual machines. From evaluating all the parameters mandatory from a MySQL server performance point of view, the analysis leads to the conclusion that containers offer better performance over virtual machines plus managing, configuring and provisioning them is trivial and less time consuming than its counterpart.

At the end, this chapter concludes that depending on the type of workload a data center or a server in a data center is about to run, the appropriate technology should be used. For CPU intensive workloads, both containers and virtual machines deliver same performances while for file I/O intensive and MySQL server tasks, containers offer better performance and the results are not even comparable to suggest further optimization's might match the performance throughput of containers. The effort involved in spinning up a container instance is also much lesser to that of the virtual machines. There is no surprise that Docker containers are now the favourite virtualization technology for all the infrastructure dependent organizations as well as cloud providers running heavy transactions as speed and minimum performance degradation govern the efficiency of a running application inside a data center. Also by using containers, cloud providers will be able to offer a more attractive and flexible SLA options to the clients thereby providing both performance and economic incentives.

Chapter 5

Latency

Boot up latency and Shut down time of a virtualization technology are one of the important characteristics of an efficient cloud infrastructure. Boot up latency can be defined as the time required to provision a guest to service a particular request while shut down latency can be defined as the time required to suspend a running guest completely enabling less resource contention of the underlying host. These two parameters directly or indirectly influence the efficiency of cloud computing model. Lower boot up time of a guest helps the data center operator to provision guests for the peak load and ensures higher probability of meeting response deadlines as promised in SLAs. Similarly, a small shut down time for guests ensures high availability of computing resources and flexibility in terms of switching workloads across the servers. In this chapter, we will study the boot up and shut down time of two guest technologies –Virtual machines and Containers. Further we will analyse the experimental results from both of them and propose some optimizations that can help in decreasing the boot up time and shut down latency.

For conducting all the experiments, we will use the same system configurations as specified in chapter 4.

5.0.1 Introduction

Considering the term "Boot up time" is quite vast in its meaning and context, it can be defined in different ways:

i) Time taken to complete the full BIOS (Basic Input/Output System) run

ii) Time taken to get to a login prompt

iii) Time taken to first see the desktop after a successful login

iv) Time taken for the user to efficiently interact with the machine after a successful login

With such a generality, usually for servers, it is measured as the time taken to get to a login prompt while for desktop, it is the time taken where a user can effectively interact with the machine after a login. In our case, since we are using a desktop machine for our experiments, we simply disable the password prompt option and measure the boot up time as "the time taken by the system to display an interactive home screen".

Some of the clients require immediate resources such as CPU, memory, disk space etc for computing resource intensive applications such as game hosting, DNA pattern analysis and weather forecasts etc. In many scenarios, the client submits requests for thousands of virtual machines and containers while the cloud provider has to provision the resources in an acceptable time frame as specified in their service level agreement (SLAs). Once these services are hosted, the clients generally release the resources immediately to save cost. In such cases, a large delay in provisioning may result in customers turning away to other providers and therefore fast provisioning of large amount of resources is a need to stay competent in the market. On demand provisioning is directly influenced by the boot up and shut down time respectively, the lesser the boot and shut down time, faster the provisioning of guests for all concurrent requests.

Theoretically, containers should have a lower boot up and shut down time than virtual machines, because for containers only API –relevant data structures are required to be initialized, but in case of virtual machines, it has to retrieve the whole operating system from the storage. The workload in container uses the host servers operating system kernel, therefore avoiding the unnecessary delay experienced in virtual machines. Higher speeds allow any development projects to get project code activated fast, test code in different ways and to launch additional features such as e-commerce capacity on a website –all very quickly. As specified earlier, the boot up time for a virtual machine is the time required till a user interactive desktop appears while in case of containers it can be measured as the time required to execute a *run* command. Similarly, the shut down time for virtual machines is noted as the time taken from the selection of the shutdown prompt to the time the hypervisor prompts an inactive message for the same machine. In case of containers, it is simply time taken to execute the *exit* command. We account for the boot up and shutdown time to get an estimate of the latency involved while these guest technologies are used in data center deployments.

For all other experiments, the host runs an Ubuntu 16.04 operating system while the guests run Ubuntu 14.04.4. Both are freshly installed and has no additional running processes at boot time. We use default Ubuntu configuration to test the boot up and shutdown time in both virtual machines and containers. System time was used to measure intervals across all iterations.

5.0.2 Experiment, Analysis & Conclusion

For all our iterations, we use the same technologies as in the previous chapter. i.e. Linux Virtual Machines (KVM) and Docker containers. Not only do we calculate the individual boot and shut down time for a particular guest but like before we compute the pattern of boot time change while increasing the number of guests on the same physical host. Number of active running guests also play a significant role in the boot up and shut down timings of a guest and we study the same as a part of our evaluation. The experiment for each scenario was performed 10 times to avoid any ambiguity in the results. Table 5.1 & 5.2 points out the boot up and shut down timings of Linux virtual machines and

Number of Virtual Machines	Boot up time (seconds)	Shutdown time (seconds)
1	6.12	3.24
2	6.30	3.40
4	7.86	3.53
8	9.00	4.03
12	30.77	11.35

Table 5.1: Boot up & Shut down time of VMs

containers in these scenarios:

i) 1 active running virtual machine and container

ii) 2 active running virtual machines and containers

iii) 4 active running virtual machines and containers

iv) 8 active running virtual machines and containers

v) 12 active running virtual machines and 16 containers

Measuring the latency in terms of scaling up the guest deployments is an essential parameter to evaluate the performance of the two, as in realistic data center environments, the number of virtual machines and containers range from few hundreds to thousands. Considering the configuration of our system, our maximum analysis is confined to 12 active running virtual machines and 16 containers.

Table 5.2 summarizes the boot up and shutdown time for Docker containers varying from one to 16 running Docker containers.

As per tables 5.1 and 5.2, we observe that it takes 6.12 seconds to boot a kernel virtual machine while it just takes around 0.44 seconds to start a Docker container. Note that the time measured is in regard to the fact that both run 14.04 base image.

As seen from figure 5.1, the boot up time of a virtual machine almost differs by a factor of 6 for a single running instance. As we increase the number of guests, we observe a sharp trend in the KVM line, the boot up time increases linearly and almost triples when we run 12 guests simultaneously, the reason for this sudden increase can be accounted to the more resource contention activity on our host. As our host can support 8 CPUs,

Number of Virtual Machines	Boot up time (seconds)	Shutdown time (seconds)
1	0.44	0.44
2	0.45	0.44
4	0.43	0.46
8	0.44	0.46
16	0.45	0.45

Table 5.2: Boot up & Shut down time of Containers

the boot up time rises slowly with respect to the increase in guests but as soon as we over provision the resources, the boot time shoots up drastically. To establish a fair comparison, we observe that under 8 active running guests the difference in between the two is almost 8.5 seconds which is significant enough from a data center operator point of view where it is expected to provision a huge number of guests at a promised time interval (based on the SLA). In contrast to virtual machines, we see that the scalability is well handled in case of containers where the boot up time remains constant up to 16 active running containers which is very impressive compared to its counterpart. A similar trend is observed for the shut down time, where the difference between the two is 3.57 seconds for 8 active guests. The shutdown time again increases as we increase the guests in case of KVM but Docker containers offer similar performance throughout, even at 16 running containers, the shut down time value remains constant with respect to the earlier readings and overlaps with the boot up latency line. It is fair to conclude that containers easily win the bout in terms of speed of spinning up and shutting down a guest as the time for provisioning and shutting off is fairly balanced and remains constant throughout all guest iterations. Further it completely depends on the nature of the workload, where other functional and non-functional requirements do not consider provisioning delay a barrier in their performance.

The delay in the boot up time of virtual machines can be accounted to the booting process of an entire operating system, which is not the case with containers. Containers share the same kernel with the host, ending up saving the time used by *bootloader* and other *init* tasks. If we could negate the operating system boot process in virtual machines, significant efficiency can be achieved. Some of the cloud providers create offline virtual machines as a template. E.g. AWS (Amazon Web Services) state saves these virtual machines so the new virtual machines can be provisioned in a short duration.

Boot up and shutdown time of a virtual machine or a container is important for applications that scale out to handle peak workload by spawning more instances of the applications. In such cases, employing containers is more profitable and efficient than using virtual machines. As a result, cloud providers use virtual machines and containers to handle peak loads.

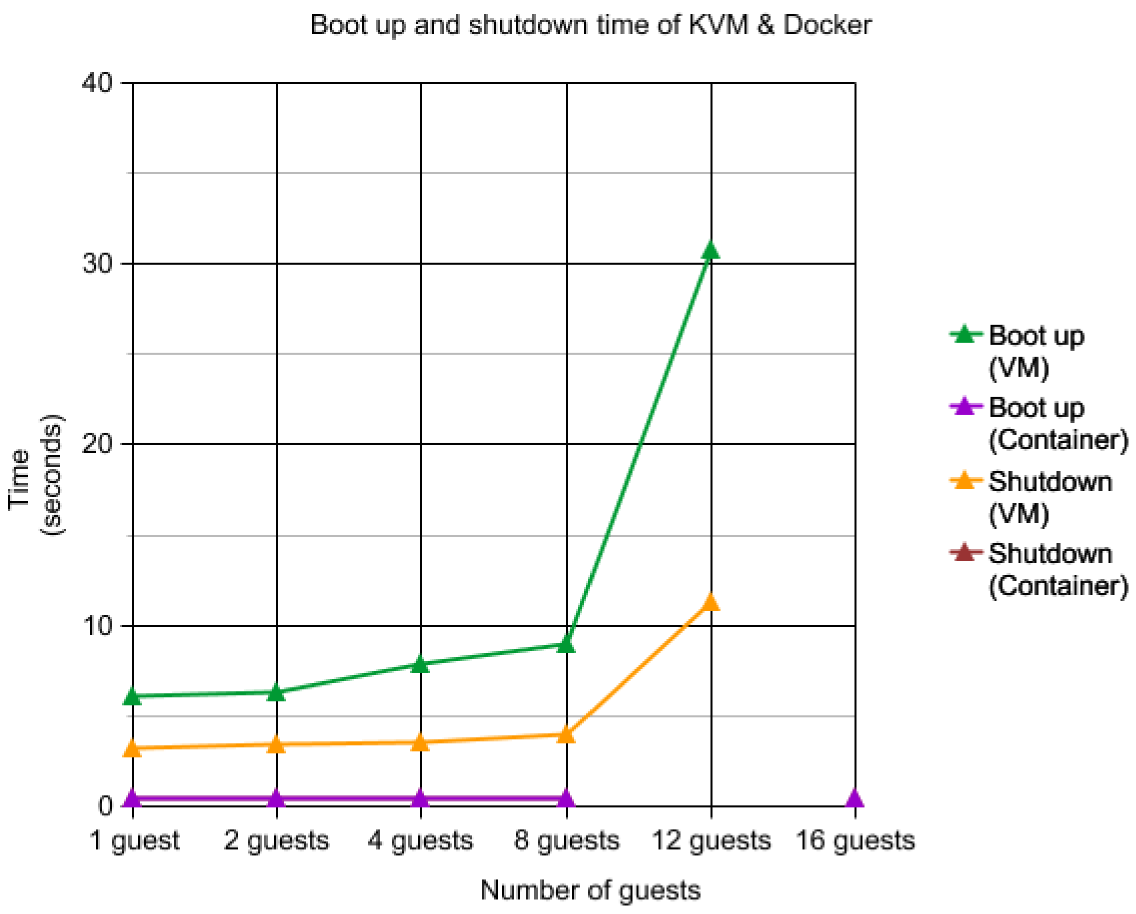


Figure 5.1: Boot up and shutdown latency for VMs and Containers.

5.0.3 Optimizations

From the results, we can clearly see that containers have better defined and faster boot up and shut down times. As far as the results are concerned, there are certain ways in which we can improve the boot up and shutdown time of virtual machines and containers. As containers already provide us a very low boot up and shutdown time, there are not many optimization techniques that can improve our results, except one which we will evaluate towards the end of this chapter. In this section, our primary focus will be on optimizing the boot up and shutdown time of virtual machines in comparison to that of the containers.

Some of the basic performance checklist of improving the boot up and shutdown times in virtual machines are:

i) Making sure that Operating system disk is defragmented

ii) Using optimal application specially at boot time

iii) Enabling hardware assisted virtualization

iv) Giving virtual machines less or required memory. Additional memory puts additional pressure on host operating system

v) Optimizing unwanted performance parameters such as "Responsiveness" control

While these steps improve the boot up and shutdown timings but there are two important methods that can further improve the boot timings considerably:

i) Enabling booting up of virtual machines at host boot time

ii) Using Snapshots (Checkpoint & Restore)

By enabling the run time boot option in Linux virtual machines, we can reduce the additional overhead of spinning up a virtual machine. As we boot the host operating system, the KVM module parallelly boots all the virtual machines that have this feature enabled therefore reducing individual boot timings across all virtual machines. Though this method does negate the boot up and shutdown time respectively but it lacks the flexibility that is required in high scale deployments such as in multi –tenant data centers, not all virtual machines should be booted as we boot the underlying server, continuous dynamic provisioning also poses a serious threat to this method. So we use this approach for a strict set of basic nodes used in cloud infrastructure that should always be up and running such as management consoles.

The second and the most important method that can optimize the boot up time is by using virtual machine snapshots. A snapshot is a copy of a VMDK (Virtual Machine Disk File) at a given point in time. Snapshots generally are used as means of check pointing and restoring a virtual machine to a particular event when a failure or system error occurs. Note that snapshots alone do not provide back up for virtual machines. In modern day data centers, snapshots are extensively used as templates. E.g. In Amazon EC2, there are numerous templates available with a pre installed operating system and required dependencies. By using snapshots, we skip the boot up time required and can in turn return directly to a particular process or application runtime. This elides majority of the overheads such as *init* processes etc., therefore reducing the start-up time by an order of magnitude.

The same optimization can be used by containers by using Checkpoint and restore in User space CRIU [3], which is explained in the next chapter as well. Checkpoint and restore in user space is a software tool for Linux operating system, which freezes a running application and checkpoints it to a disk as a collection of files. Afterwards, the same files can be restored and run from the check pointed state.

From figure 5.2, we can see that check pointing a virtual machine results in considerable improvement. The virtual machine boots almost three times faster than using normal boot up process, which provides a huge performance bonus to the data center operator. In contrast, for a container, the time saved by using a check pointing and restore approach is around 0.03 seconds. Therefore, starting a container is almost as fast as restoring it from a check pointed state. Check pointing does not give the same benefits to container as that to a virtual machine.

Even by using the check pointing and restore optimization, the container is still 4.5

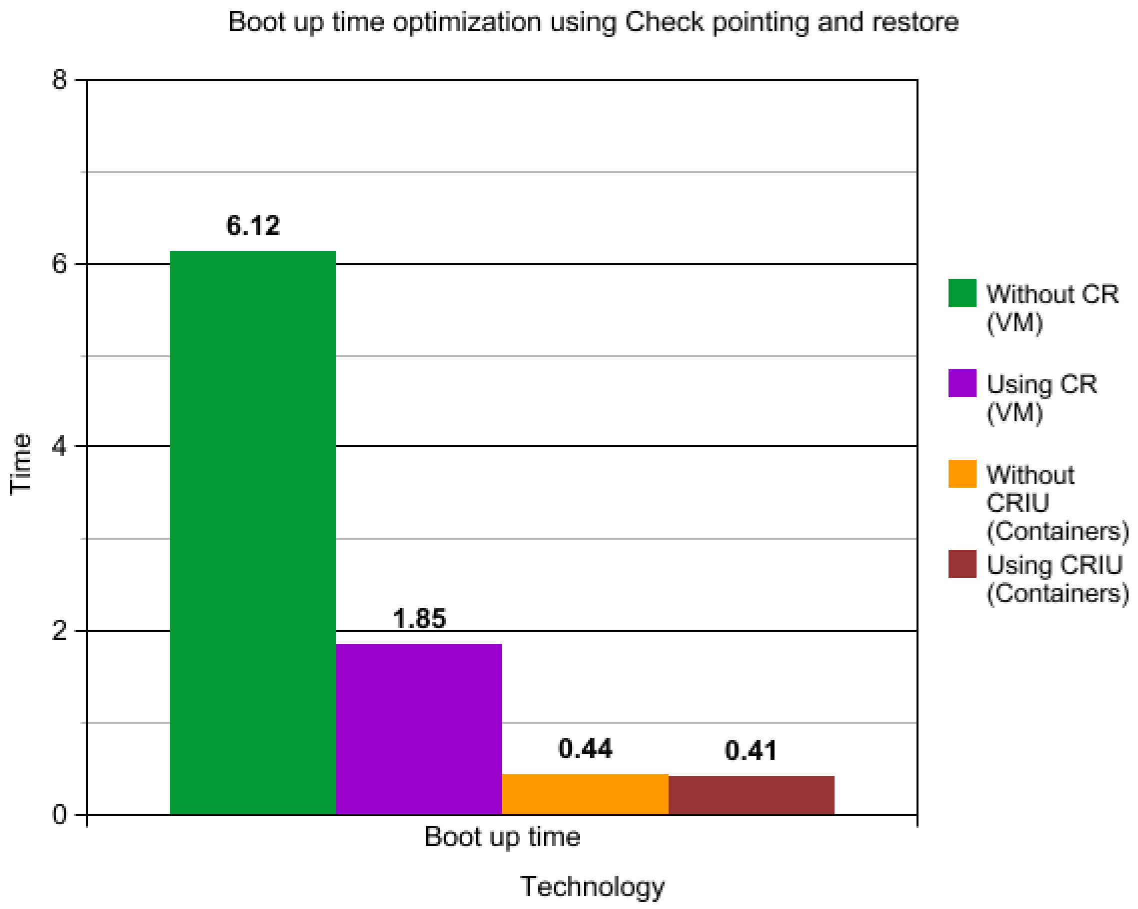


Figure 5.2: Boot up time optimization using checkpointing and restore.

times faster than virtual machines. This difference may further reduce in future with new technologies and optimization specially towards the check pointing and restoring feature. E.g. SnowFlock [33] virtual machine fork system reduces the boot up time of a virtual machine by 3–4 times by duplicating only the virtual machine descriptor and guest kernel memory from an already running virtual machine and then duplicating the rest on demand. Snowflock uses XEN hypervisor for this.

It is usually a non–trivial process to analyse the boot up study from outside a data center and on different hardware architectures but in recent performance analysis studies [37], we note that adding a 100 –200 ms delay to a request is unlikely to be satisfactory considering the orders of magnitude of concurrent requests. We suspect that further changes in the operating system such as using more aggressive para virtualization techniques will help in meeting the *ms* strict demands of latency sensitive services.

Therefore to conclude, containers perform better than virtual machines but the gap is considerably reduced to that of earlier boot times by using checkpointing and restore optimization. The start–up time for a virtual machine which was almost 14 times to that of containers could be reduced by almost a factor of 10, which results in a huge performance incentive. Similar approach does not benefit containers as such and the difference between the boot up latency of containers with or without using check pointing and restore is almost negligible. Overall containers still prove to deliver better performance over virtual machines but with continuous optimizations in check pointing and restore methods for virtual machines, we might suspect that these numbers will almost be comparable to that of containers in near future.

Chapter 6

Migration

Migrating operating system instances across different physical hosts is a useful methodology for data center administrators. It allows a clean distinction between the hardware, software and facilitates fault management, load balancing and system maintenance. It also significantly boosts an organizations disaster recovery efforts and improves business agility. In this chapter we will discuss the migration scenarios in both Linux virtual machines and Docker containers as well as evaluate its performance in terms of migration.

6.1 Introduction

In chapter 2, we have already specified the various use cases, assumptions, types and the process of virtual machine live migration. Migration is one of the important characteristics of a flexible cloud infrastructure. Migration not only provides performance enhancing features such as load balancing and proportional server distribution across the geography but also helps during catastrophic cases like data center crisis or disaster recovery. It also enables efficient performance management from a data center operator point of view. Demand based flexibility is vital in terms of cloud infrastructure. E.g. A game server instances have to be migrated from one location to another depending on the peak load time in a particular time zone. Spinning up additional instances often results in high cost and complex management process. Migration of guests is always a viable option because not only does it present economic incentives in terms of less active instances but also incurs negligible to almost zero downtime. Choosing a particular type of migration methodology usually depends on the type of workload to be processed. Cold migration is a cheaper alternate but involves high processing and considerable downtime, on the other hand live migration is comparatively expensive and has negligible impact on the performance of an active virtual machine and the downtime almost approaches zero.

There are many security and compliance concerns around the migration process. If we do not follow the virtualization best practices for migration, our infrastructure may be susceptible to security risks and sometimes violate the compliance regulations. E.g. Migrating a virtual machine with customer credit card data to a host that in turn is running a public web server, for example, violates the payment card industry data security standard. All the corporate policies and compliances can be fulfilled by using a migration suite consisting of management software, provisioning software and integrated process management tools.

Although the migration of virtual machines is a much matured technology with dedicated products such as VMotion from VMware which governs smooth mobility of virtual machines across same or different domains. Because of the wide applicability of migration in virtual machines, these products are designed to have negligible impact on the performance and enable easy management platforms.

The same cannot be said about containers. Since containers are usually lightweight and stateless, migrating them is not beneficial. The cost of spinning up a new container is almost equal to migrating the same. However, if a container is used to host applications that are required to maintain a state, there is always a volume associated with the container which has to be migrated as well but as of today Docker (Docker version 1.12.0) does not support check pointing and restoring a container which is required for migrating the volume of a container. The present version uses *pause* and *unpause* to suspend and restore the container state on the same host which in turn only checkpoints and restores the instances rather than the volume. This pause and unpause feature is based on cgroups *freeze* feature.

In this chapter, we will use an experimental version of Docker (1.9.0-dev) that enables check pointing and restoring along with its volume. We will further analyse the following parameters to establish a clear conclusion of their respective migration profiles:

I. Total Migration Time

II. Service downtime

These two parameters govern the performance of migration process. Total migration time can be defined as the time taken to complete the whole migration process starting from executing the migration command from the source to the time the target host services its first request.

The downtime during migration can be defined as the time during which the services are unavailable to the end users, the switch over from source to target causes the downtime of a service.

We evaluate these parameters in both virtual machines and containers and also establish a clear view of the performance degradation and its possible countermeasures.

6.2 Benchmark

We perform the migration of virtual machines and containers in both clean and stress testing state. It is important to test the migration during a state where a virtual machine or a container is running an application or service that performs heavy CPU and file I/O transactions. Using such a scenario is vital for our analysis so as to evaluate the impact of a prolonged pre copy phase on the overall migration performance. For all our experiments, we will be using *mprime* as our benchmark.

Mprime is the Linux command line interface version of Prime 95 [12] to be run in a text terminal or in a terminal emulator window as a remote shell client. It is a freeware application written by George Woltman that is used by GIMPS, a distributed computing project dedicated to find new Mersenne prime numbers. In simple words, mprime, processes through a giant set of numbers as fast as possible to determine if they are prime.

Mprime makes heavy use of the processors integer and floating point instructions, it feeds the processor a consistent and verifiable workload to test the stability of the CPU and the L1/L2/L3 processor cache. It also uses all cores of a multi CPU system to ensure high load stress test environments. An additional hybrid mode is provided that tests both CPU and RAM. The thread specification enables multi core testing.

Depending upon the type of processor used for testing, the mprime gives us an option to use the same amount of threads during our testing phase. The mprime supports 3 types of tests:

i) Type1: Performs small fast Fourier transforms which involves maximum heat and power consumption. This mode does not involve any memory testing.

ii) Type2: Performs large fast Fourier transforms (with much higher heat and power consumption). In this mode, some of the RAM is also tested

iii) Type3: Also known as Blend which almost tests every aspect of the system. This mode involves heavy RAM testing.

Type 3 is the extreme level of stress testing in mprime and failure of the same indicates bad memory or a bad memory controller.

Since the primary aim of our experiment is not focused around stress testing of a system therefore for all our tests we will use Type 2 mode as its stresses both CPU and RAM to a considerable limit that may apply to a practical scenario of active services running on a guest to be migrated.

6.3 Experimental setup and Configurations

The nodes used for the experiment have identical system configurations as specified in chapter 4. We use the same system configuration to avoid any mismatch in terms of system architecture and operating system compatibility. From figure 6.1, we can see that

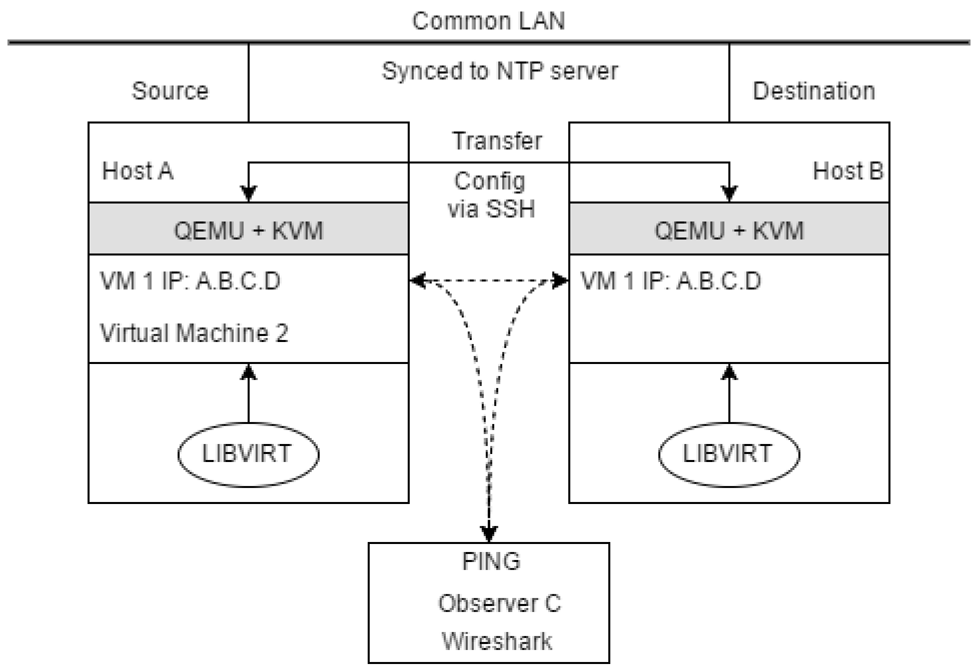


Figure 6.1: Experimental setup for VM Migration.

the experimental setup consists of 3 nodes:

i. Host A (Source node)

ii. Host B (Target node for migration)

iii. Observer C

The nodes use public key authentication to establish an SSH session between them. The observer node C is explicitly used to capture the events on Host A and Host B. The events captured are the timings with respect to total migration time and service downtime. All the nodes exist on the same network and are interconnected by a high speed 1 Gbps LAN. Since the hypervisors in both Host A and Host B are connected, each time Host A boots up, it can remotely manage all the virtual machines on Host B. Port forwarding of virtual machines area enabled to enable virtual machine access from the same or a different source. Further we test the virtual machine under following configurations:

i. Clean Linux machine with 1 CPU and 512 MB RAM

ii. Clean Linux machine with 1 CPU and 1 GB RAM

iii. Clean Linux machine with 1 CPU and 6 GB RAM

iv. Virtual machine running mprime with 1,2,4 CPUs and 512 MB RAM

v. Virtual machine running mprime with 1,2,4 CPUs and 1 GB RAM

vi. Virtual machine running mprime with 1,2,4 CPUs and 2 GB RAM

The following configurations are used to analyse if the migration process is affected during a particular time period where a workload is performing heavy CPU and memory related transactions. The migration process was reiterated 5 times for each of the configurations listed above.

In order to capture the events across the source and the destination, we use a script file which continuously pings (using default 1 second time interval) both the machines and also clears any cached entries for the address resolution. Along with continuous pinging the system, we capture the data packets using Wireshark [13]. The Wireshark intercepts the ping responses on the host A and on host B. The Wireshark is configured to listen to a particular interface/bridge (virbr0) with machines IP address and type of packet filtered as "ICMP". The Wireshark instance on the destination host B waits for the acknowledgment of the ping requests during migration. It is interesting to note that during the process of migration, the IP address of a virtual machine does not change, which enables us to pre configure the Wireshark instance on host B. By using such a setup, we compute the total migration time and downtime respectively.

- Total Migration Time = Time at which Migration command is executed –Time at which the first ping acknowledgment is received on host B.

- Service Downtime = Time at which the last packet on host A failed a ping –The first ping acknowledgment at target host.

To avoid any time synchronization problem between host A and host B, both the systems were synchronized to an external NTP server. For all our experiments, we use Linux 14.04.4 as our guest operating system.

As mentioned earlier, the migration methodology in case of containers is relatively immature as compared to its counterpart, so instead of migrating a container from one

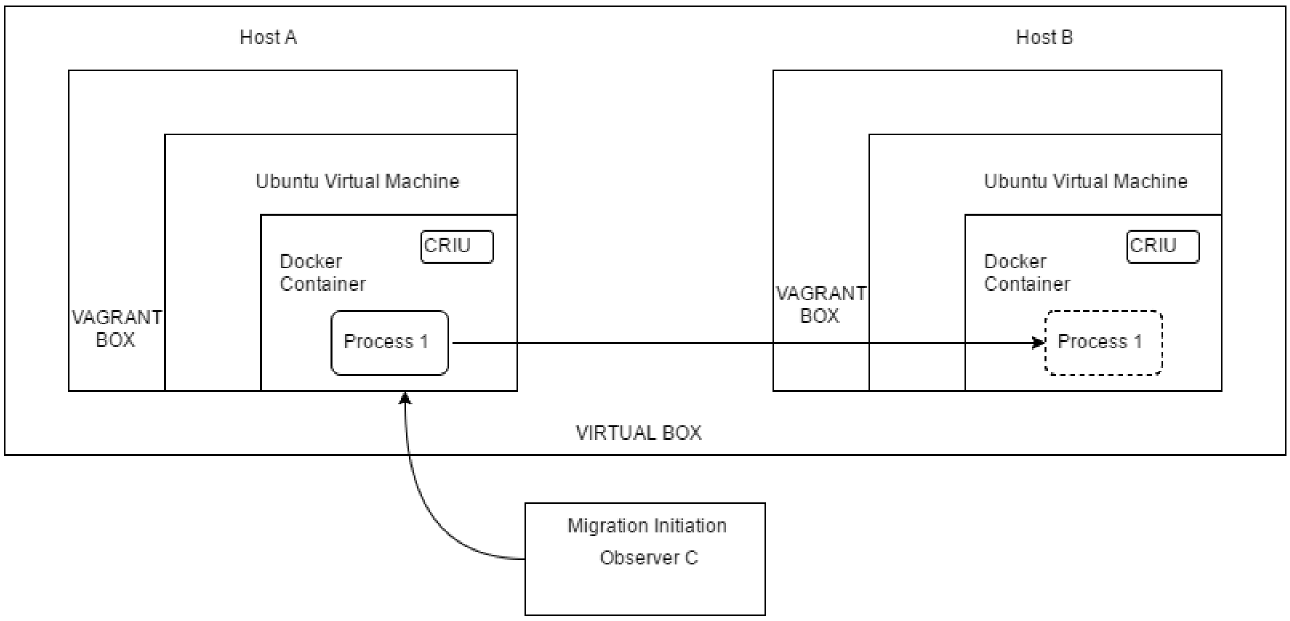


Figure 6.2: Experimental setup for Docker Migration.

physical machine to another, we migrate the containers from one virtual machine to another using Vagrant [9]. The migration process involves moving both active running instances and the volume associated with it. From fig. 6.2, we can infer that Vagrant in turn uses Virtual box [10] to manage these virtual machines. In our test scenario, we perform all the experiments on a single physical host and spin up two virtual machines using Vagrant to enable container migration. Considering the container migration feature is still under experimental phase, we expect the future implementations of containers to support migration natively. The container migration also requires compiling Kernel with specific Kernel modules enabled as well as using an experimental version of Docker (1.9.0-dev), which supports the check pointing and restoring of Docker containers by saving their state to files. The complexity involved in the process made Vagrant the right technology choice for our analysis. We use Ubuntu 14.04 cloud image for all our container experiments. The host system uses Ubuntu 16.04 in both virtual machines and container experimental setup.

6.4 Technologies

6.4.1 KVM Migration

We use the same technologies as specified in chapter 4, i.e. KVM + QEMU + Libvirt for virtual machines and Docker for containers.

KVM supports different types of migrations such as offline and live migration. We can also migrate between an AMD host to an Intel host, but a 64-bit guest can only be migrated to a 64-bit host. In our experiment, we perform live migration of virtual machines on a 64-bit Intel machine. Though it is recommended to use shared storage (NFS) to perform virtual machine migration in KVM but considering the additional infrastructure involved, all our experiments are performed directly without involving any shared storage pool. Using shared storage improves the performance of migration in data center deployments as the numerous systems have access to the same storage pool thereby decreasing network latency involved in fetching a particular resource from one host to another. In our experiments, we will use a pre hook and use the option of *–copy -storage -all* to enable virtual machine migration. The guest machine is migrated from source to destination by using the *virsh* command. The general syntax of the virsh command is:

```
# virsh migrate –live guest name destination URL
```

where, guest name is the name of the guest to be migrated

and, destination URL is the connection URL of the destination host physical machine

The destination system should run same Linux version, be using same hypervisor and has libvirt running. An example of the migration command used in our experiment is:

```
# virsh  migrate  –live –copy –storage –all testVM qemu+ssh: 134.226.34.148
```

The successful migration of virtual machine can be further verified by using the *virsh list* command, which displays the names of current virtual machines running on the system.

It is also possible to perform multiple, concurrent live migrations, where each migration process runs in a different command shell. The migration process can also be performed using virt-manager which is a graphical management console for Linux virtual machines. The live migration process can be verified by checking the newly transferred guest running mprime stress testing. Our analysis further quantifies the effect of RAM size and variable CPU cores during live migration process

6.4.2 Docker Migration

We will use suspend and resume process to migrate a Docker container from one host to another. We will be using CRIU [3] with Docker to enable container migration. CRIU as already mentioned in previous chapter is a software tool for Linux operating system which enables freezing a running application and check pointing it to a persistent storage as a collection of files. The same can be resumed later with all its volume dependencies. The CRIU is implemented in user space rather than in Kernel. Since the current Docker version does not support suspending and resuming of containers, we will therefore use CRIU to enable it.

CRIU was originally build to support only Linux containers but as we know Docker is capable of running a Linux container, we should be able to use the same features in Docker. In order to use CRIU, we need to build a Kernel with certain features enabled. The following features can be enabled by preparing a Kernel *.config* file. The features to be enabled for Kernel can be found out on CRIU official website. Since check pointing and restoring Docker containers is still under experimental phase, we need to build an experimental version of Docker. As CRIU till date is not fully merged into Docker, therefore we will use a fork of Docker [14]. All the alterations and dependency setup were performed inside a Vagrant virtual machine and the box can be downloaded from the repository [15]. After configuring all the dependencies, we spin up a virtual machine and ssh into it to run a Docker container. We can check the checkpoint and restore feature description by using the –help option. The syntax for checkpoint and restore is:

```
# docker checkpoint /*name of container*/
# docker restore /*name of container*/
```

To perform container migration, we spin up two vagrant virtual machines. We execute a script that prints consecutive odd numbers with system time. System time is printed up to millisecond precision, in order to capture the migration time. Another helper shell script is written to perform migration container migration [16]. The script needs 3 arguments to run:

i. Name of container to be migrated

ii. Path of host A vagrant box

iii. Path of host B (target) vagrant box

The migration process involved in containers also uses a live pre copy approach. The total migration time and service downtime is calculated by using a script which prints a stream of odd numbers along with the system time on the console. This system time is used to calculate the total time and service time. Further Docker logs can also be used to verify the timestamps. The success of migration process is confirmed by checking if the process prints the next odd number on the console of host B following the last odd number printed on the console of host A. The total migration time and service downtime are calculated as:

- Total migration time = Time at which migration command is executed –Time at which the first number is printed on the target host

- Service downtime = Time at which the first number was printed on host B –Time of the last printed number on host A

Both of these parameters are analysed with respect to virtual machines to verify the performance of migration and to understand if migrating a container is beneficial than directly running a new container and suspending the old one.

6.5 Evaluation and Conclusion

Migration using the setup as specified in the previous section was performed and the different time period under different workloads and configurations were captured. The evaluation of migration performance was entirely based on the two parameters

i) Total migration time

ii) Service downtime

6.5.1 Total Migration Time

As mentioned previously, total migration time can be computed as the difference between the time the migration command is executed to the time when the target host of migration processes its first request. Migration time in case of other migrations such as offline mode is comparatively less as compared to that of live migration because unlike live migration, other modes of migration do not involve iterative pre copy phase. Iterative pre copy phase directly influences the migration time which in turn depends on the nature of the workload under execution. If an application is generating high number of writes, it will eventually increase the dirty paging frequency, thereby increasing the total migration time as a substantial amount of threshold is required to proceed to the next phase, otherwise the iterative pre copying processes in a continuous loop. Several algorithms [50] have been designed to efficiently manage the pre copy iterations. Because of the high accuracy of iterative copying, the service downtime is directly influenced.

In our experiment, we perform migrations under different configurations of virtual machines. We perform the migrations under different virtual machine configurations to analyse if system parameters (RAM, CPUs) and active processes affect the migration performance.

As seen from figure 6.3, for clean Linux virtual machines with 1CPU each and varying RAM allocations (512MB, 1GB, 6GB), we observe that irrespective of the memory size, the total time remains fairly constant with the mean clocked at 244.2 seconds.

Number of Virtual Machines	Boot up time (seconds)	Shutdown time (seconds)
1	6.12	3.24
2	6.30	3.40
4	7.86	3.53
8	9.00	4.03
12	30.77	11.35

Table 6.1: Migration Profile for VMs running mprime

For further analysis, table 6.1, we perform virtual machine migration by not only allocating a variable memory but variable CPU as well. All these virtual machines also run mprime benchmark. We perform these iterations under different system configurations to emulate the nature of servers and workloads inside a data center. We gradually increase the RAM size with CPU allocation, we also use threads for mprime testing, to establish a fair comparison for our evaluation in case of multi core systems. Each thread stress tests each core of the processor. The mean time completion for 512 MB RAM with variable CPU allocation was clocked at 215.6 seconds while the time increased by 245.6 seconds and 274.2 seconds respectively for 1GB and 2GB RAM sizes. We can see that the RAM size did not affect the overall performance of migration except a minimal rise in time due to varying memory allocations. In case of data centers, where heavy workloads are processed, this minimal fraction of time increase for various memory sizes can sum up to a large value resulting in performance degradation. As we can see from figure 6.4, the time almost remains constant and the slight deviation among the numbers can be accounted to variable memory allocations and network delays.

As we know, containers use host Kernel to perform all process execution, so the check pointing and restoring in case of containers experiences less migration time, as only a process is migrated from one system to another to that of migrating a fully-fledged operating system. The migration time though should not be compared directly to virtual machines as we are using experimentally build packages to perform process migration, which at this point of time is not encouraged in production environment. As per table 6.2, the Docker

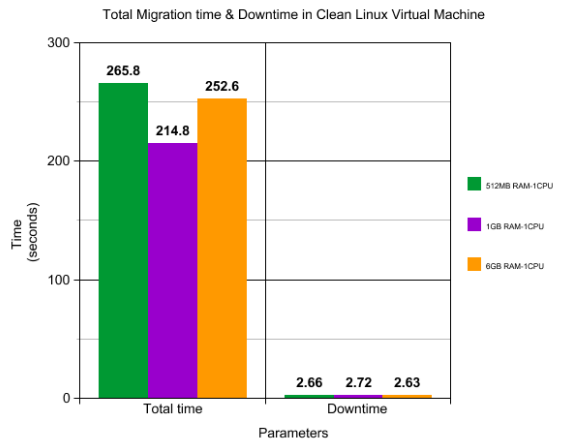


Figure 6.3: Migration profile for clean VM.

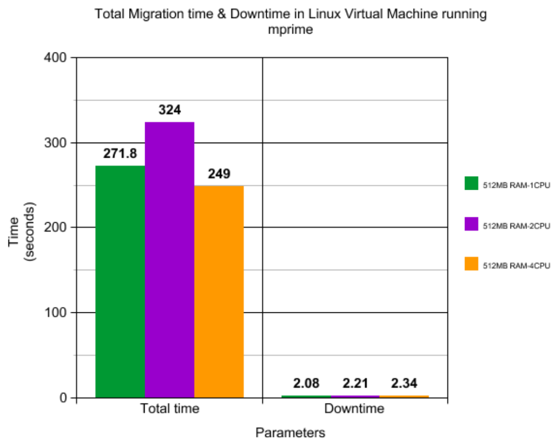


Figure 6.4: Migration profile for VM running mprime.

container migration process requires 11.79 seconds which is almost 20 times better to that of virtual machine.

6.5.2 Service Downtime

From an application performance point of view, total migration time is not an important factor in determining migration efficiency as it is invisible to the end users and the application itself. Total migration time only proves to be beneficial for a cloud provider, where instances should be moved quickly thereby providing flexible resource provisioning. Service downtime can be defined as the time during which a service becomes unavailable due to migration. A large service downtime has a direct effect on the end users and is one of the important parameter of a SLA for a cloud infrastructure. In order to abide by the SLA and offer high performance, a service should have a zero downtime. Practically it is impossible to experience zero downtime as the switching delay will always tend to make services unavailable for a fraction of a time, however it is mandatory to reduce the delay to the smallest number possible.

In our experiments, we compute the downtime as the difference between the time the new host services its first request to that of the last request serviced by the primary host. We use the same system and software configurations as used in the previous section. In case of a clean Linux virtual machine migration, the downtime experienced is marked at a mean of 2.67 seconds. Further we test the downtime under different configurations and memory sizes. It is interesting to note that the downtime remains constant in all scenarios. i.e. for 512 MB RAM at different CPU allocations, the mean downtime is 2.21 seconds. We can see from figure 6.4 and table 6.1, that the downtime is scattered across all configurations with 1.81 & 2.51 seconds respectively. From this we can infer that the memory size and workload do not affect the service downtime as the values approach a constant value primarily because of same threshold for iterative pre copy phase across all configurations and iterations.

In case of containers, the service downtime is measured around 5.70 seconds (table 6.2) which is almost double to that of virtual machines. This downtime can be accounted to the immature version of CRIU package for Docker containers and therefore cannot be considered as an accurate estimation of the performance.

From figure 6.5, it is fairly easy to conclude that a virtual machine has a considerable large migration time (237.2 sec) as compared to containers (11.70 sec) but the trend is reversed for service downtime (2.20 sec for VMs and 5.70 sec for Container). Virtual machines offer high performance to that of containers during migration as per present state of art because of the less service downtime. As described earlier, service downtime directly affects the efficiency of the running application as well as the SLAs. Low or negligible downtime should be key to every migration technique. As seen from our analysis, it should be noted that virtual machines offer better performance during migration than containers although containers offer quick migration than virtual machines by a factor that is even not comparable but considering the service downtime is almost double in containers than virtual machine, it should make virtual machine a favorable option in all data center deployments. Moreover because of the maturity in migration techniques in virtual machine, it is considered as a viable option to that of containers.

In future, we might witness some developments towards the container migration especially in Docker because of its wide spread use across industries. But at this moment, it

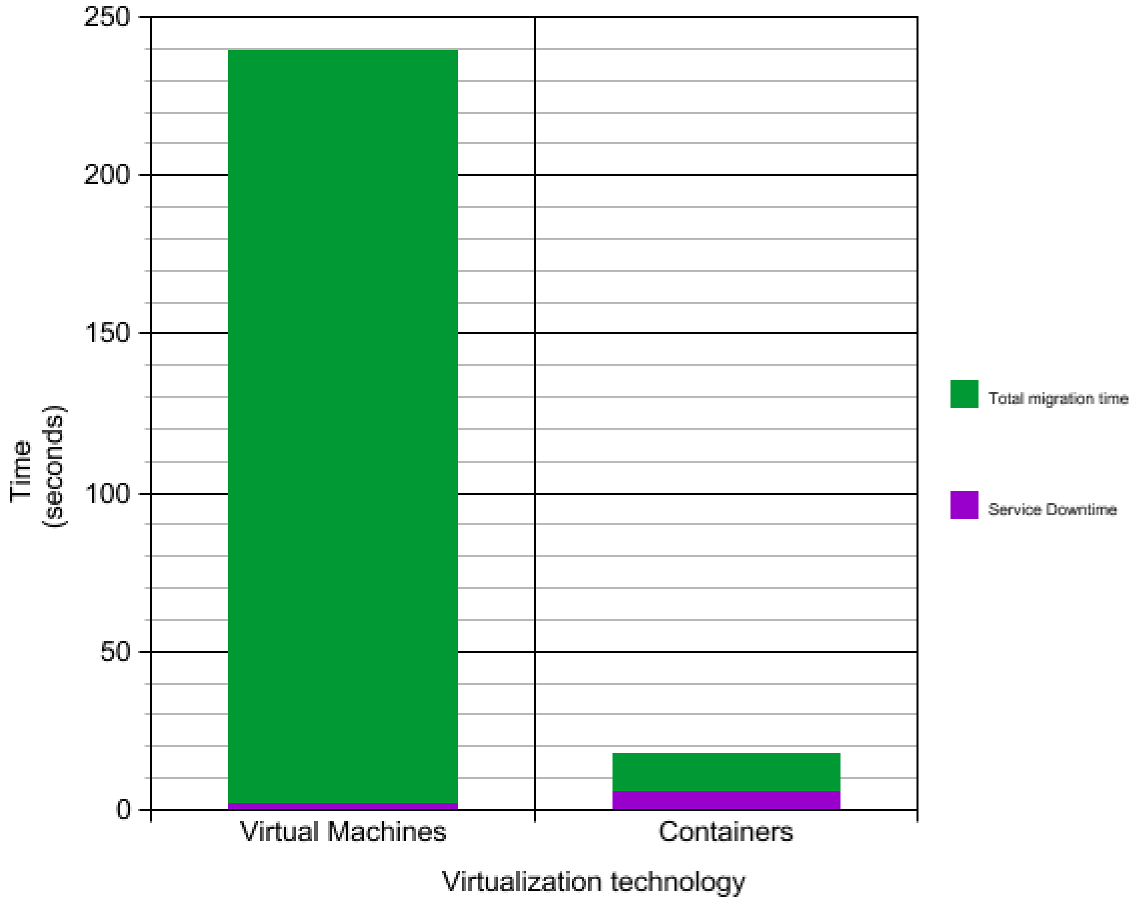


Figure 6.5: Evaluation of Migration Performance.

is fair to conclude that virtual machines perform better than containers when migration is a mandatory requirement.

We may experience minor setbacks when the tools or technology we are using fails to work as expected during migration. This may arise because of locked files or network and site outages. It is important that the technology we use is able to perform under adverse conditions without impacting production and losing data. Considering these risks, virtual machines should be preferred for migration as there are dedicated migration packages which not only optimize but orchestrate the whole migration process keeping potential risks under consideration. On the other hand, same cannot be said about containers as the migration technology under containers is still immature and under constant development.

Chapter 7

Conclusion

In the previous chapter, we evaluated the migration performance which is one of the important characteristics of effective virtualization. In this chapter, we will summarize the project, its findings and discuss some future work.

7.1 Overview

Virtualization projects have been the focus of many IT organizations where primary aim is to consolidate servers or data centers, decrease Capex (Capital expenditure) and Opex (Operating expenses) and launch successful green conservation initiatives. Virtualization in simple terms can be defined as consolidating processing power, memory, network bandwidth and storage capacity onto the smallest number of hardware platforms possible and then apportioning those resources to servers or system based on time slicing. The main aim of virtualization is to make the most efficient use of available IT resources. Virtualization decouples all computing resources making them available to a common pool. As a result of this, new or old services can be added, removed or altered without any complex process.

Virtualization benefits the IT infrastructure in terms of:

i. Lower expenses

ii. Business continuity

iii. High availability

iv. Fast provisioning

v. Corporate governance

Virtualization can be enabled using many technologies, out of which virtual machines and containerization are the most preferred methods. Hypervisor and virtual machines have been the most used approach to virtual workload deployment because of its maturity and massive open community support.

Container virtualization is now quickly emerging as an efficient and reliable alternate to traditional virtualization, thereby providing new flexible features as well as problems to data center operators. The difference between the two can be can be characterized by resource usage management (operating system level) and location of virtualization layer. Virtual machines rely on a hypervisor which is either installed on top of a host operating system or on top of the actual bare metal. Once the hypervisor is installed, virtual machines can be provisioned from systems available computing resources. Each virtual machine in turn possesses its own operating system and running applications. Virtual machines offer full isolation from its other peers running on the same physical host.

On the other hand, in containers a host operating system is installed first followed by a container layer e.g. LXC or libcontainer. Once the virtualized container layer is installed, containers can be provisioned from underlying systems available resources and various applications/services can be deployed inside them. These containers offer isolation but share the same operating system with the host machine.

Containers are considered as more resource efficient than virtual machines because the need of additional resources for each operating system is eliminated. Therefore, the resulting instances are smaller and faster to provision and manage. As a result of this, a single physical system can host more containers to that of virtual machines. It directly affects the economics towards a cloud infrastructure by providing a considerable cost cut but also presents a single point of failure for all running containers. However, containers and virtual machines can co-exist in the same environment, complementing each other thereby expanding the available toolset for data center operators to provide resource optimization based on the nature of the workload.

In this project, we evaluate the performance of virtual machines and containers to narrow down the gap purported between the two. We evaluate the performance considering the economics from a cloud provider point of view. We use Linux virtual machines and Docker containers for our experiments, as both of them are open source technologies and offer near native performance. We evaluate 3 important parameters

- Density
- Boot latency
- Migration

Density is further evaluated using the following factors

i) CPU performance

ii) File I/O performance

iii) MySQL server performance.

The performance is evaluated by scaling up the deployments incrementally on the same physical host. From the results, we observe, the CPU performance is almost equivalent in both the cases as the number of guests scale up. The performance degradation factor also remains constant.

In case of file I/O performance, containers offer better performance than virtual machines in terms of execution speed and number of transactions. The performance degrades considerably when the virtual machines are scaled up to 8, thereby proving to be less optimal as compared to containers, where the performance remains fairly constant. The containers again offer better performance in case of MySQL server implementation by executing high number of reads and writes.

Secondly, we evaluate the boot up and shutdown times for both the technologies. The boot up and shutdown time in case of containers fairly remains constant irrespective of the number of guests. While for virtual machines, the time increases linearly and shoots up when the memory allocation for virtual machine exceeds the actual RAM size of the underlying host leading to provisioning bottleneck. The checkpoint and restore optimization can be used to improve the boot latency in virtual machines while the same optimization technique is not effective in containers.

Lastly, we evaluate the migration performance in both virtual machines and containers considering total migration time and service downtime as our primary factors. We note a contrasting behavior in between the two, virtual machines require more time to migrate a virtual machine than a container. i.e. almost 20 times higher than that of containers. On the other hand, service downtime which directly impacts the end user and the application performance is almost double in containers to that of virtual machines. High service downtime in containers can be accounted to the technology immaturity for migration process. We still use experimental build tools to evaluate migration in containers which cannot be considered as a standard basis of performance analysis in production based environment.

7.2 Future Work

This work quantifies the performance profile of both virtual machines and containers. Further research could express the evaluation more precisely using wider system parameters. Since the experiments were performed in a research laboratory, it would be interesting to evaluate the same in a data center environment with actual industrial workloads. The performance prediction can be improved by using higher degree of scalability such as by spinning up 1000s of virtual machines and containers in a data center environment. Also the migration testing could be improved by using a SAN and performing migration iterations inside or outside a particular network. Further establishing migration performance analysis across different cloud infrastructure providers as well as across variable machine architectures would improve the prediction model. Lastly, due to lack of development in container migration methods, it would be interesting to perform the migration tests when container technologies such as Docker supports the migration process natively rather than using a third party module/package. Further research could capture the role of Kernel data structures to the cost of virtual machines and containers. Theoretically, containers should improve higher costs on host Kernel opposed to that of virtual machines.

7.3 Contribution

In summary, this work contributes to the comparative analysis of virtualization environments. The evaluation quantifies overall performance parameters between virtual machines and containers. Our evaluation results demonstrate that containers offer better performance in terms of density and provisioning (boot latency) when deployments are scaled incrementally, while the virtual machine performance shoots up when the resource allocation exceeds the available resources, because of high swap activity.

Furthermore, this study also focuses on the migration performance, even though it is not officially supported in any of the available container technologies. The study acknowledges the high migration time in virtual machines over containers but at the same time providing minimum service delay over virtual machines.

7.4 Final Remarks

Virtualization today has become a major component of IT infrastructure and has been widely used to effectively provision and manage computing resources. Virtualization has governed the IT policies as well as economics. Virtual machines have been used since the dawn of virtualization and have proved to be efficient. The resurgence of containers has been late and has gained popularity during the last couple of years. As the interest in containers continue to grow, hopefully this will lead into the research of new and better ways to optimize its performance and additional features such as migration, so that IT industries and cloud providers can easily benefit from the performance incentives that modern day containers can offer us.

Appendix A

Abbreviations

Short Term	**Expanded Term**
SPARC	Scalable Processor Architecture
LXC	Linux containers
AUFS	Unification File System
QOS	Quality of service
KVM	Kernel Virtual Machine
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
NMON	Nigels Monitor
DAX	Direct Access
KSM	Kernel Same-page Merging
MPI	Message Passing Interface
TCO	Total cost of ownership
DNS	Domain Name System
ROI	Return on Investment
UDP	User Datagram Protocol
IPC	Inter Process Communication
POSIX	Portable Operating System Interface

HA	High Availability
FT	Fault Tolerance
SMP	Symmetric Multiprocessing
ARP	Address Resolution Protocol
WAN	Wide Area Network
VNIC	Virtual Network Interface Card
NICs	Network Interface Cards
SAN	Storage Area Network
NAS	Network Attached Storage
DomU	Domain Unprivileged
Dom0	Domain Zero
VDE	Virtual Distributed Ethernet
MIP	Mobile IP
NAT	Network Address Translation
TTL	Time to Live
SIP	Session Initiation Protocol
UA	User Agent
VMDK	Virtual Machine Disk File
CBC	Characteristics Based Compression
CSP	Cloud Service Provider
VSR	Volume to Size Ratio
DHCP	Dynamic Host Configuration Protocol
QEMU	Quick Emulator
TCG	Tiny Code Generator
BIOS	Basic Input/Output System

Bibliography

[1] Canonical. lxd crushes kvm in density and speed. https://insights.ubuntu.com/2015/05/18/lxd-crushes-kvm-in-density-and-speed/, 2016. [online; accessed 18:33, 7 july 2016].

[2] Condor team. condor. http://research.cs.wisc.edu/htcondor/, 2016. [online; accessed 18:33, 7 july 2016].

[3] Criu team. criu. http://criu.org/, 2016. [online; accessed 18:33, 7 july 2016].

[4] Kvm, docker & lxc benchmarking with openstack. http://bodenr.blogspot.com/2014/05/kvm-and-docker-lxc-benchmarking-with.html, 2016. [online; accessed 18:33, 7 july 2016].

[5] Sysbench official github repository. https://github.com/akopytov/sysbench, 2016. [online; accessed 18:33, 7 july 2016].

[6] Sysbench cpu test image (ubuntu 14.04). https://hub.docker.com/r/rnjndhr/sysbench-cpu/, 2016. [online; accessed 19:52, 25 august 2016].

[7] Sysbench file i/o test image (ubuntu 14.04). https://hub.docker.com/r/rnjndhr/sysbench-fileio/, 2016. [online; accessed 19:52, 25 august 2016].

[8] Sysbench mysql test image (ubuntu 14.04). https://hub.docker.com/r/rnjndhr/mysql-sysbench/, 2016. [online; accessed 19:52, 25 august 2016].

[9] Vagrant home page. https://www.vagrantup.com/, 2016. [online; accessed 19:52, 25 august 2016].

[10] Virtualbox home page. https://www.virtualbox.org/, 2016. [online; accessed 19:52, 25 august 2016].

[11] Clear containers. https://clearlinux.org/features/clear-containers. 2016. [online; accessed 18:33, 7 july 2016].

[12] Prime95 home page. http://www.mersenne.org/, 2016. [online; accessed 19:52, 25 august 2016].

[13] Wireshark home page. https://www.wireshark.org/, 2016. [online; accessed 19:52, 25 august 2016].

[14] Criu compatible docker build. https://github.com/boucher/docker/tree/docker-checkpoint-restore, 2016. [online; accessed 19:52, 25 august 2016].

[15] Pre configured vagrant box. https://atlas.hashicorp.com/rnjndhr/boxes/criudocker, 2016. [online; accessed 19:52, 26 august 2016].

[16] Container migration helper script. https://gist.github.com/rnjn09/96763e0de315ac53cecf188a84
2016. [online; accessed 19:52, 26 august 2016].

[17] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[18] Carlos Antunes and Ricardo Vardasca. Performance of jails versus virtualization for cloud computing solutions. *Procedia Technology*, 16:649–658, 2014.

[19] Bao Rong Chang, Hsiu-Fen Tsai, and Chi-Ming Chen. Evaluation of virtual machine performance and virtualized consolidation ratio in cloud computing system. *Journal of Information Hiding and Multimedia Signal Processing*, 4(3):192–200, 2013.

[20] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[21] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[22] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[23] Zachary J Estrada, Zachary Stephens, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar K Iyer. A performance evaluation of sequence alignment software in virtualized environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 730–737. IEEE, 2014.

[24] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[25] Andreas Fischer, Ali Fessi, Georg Carle, and Hermann de Meer. Wide-area virtual machine migration as resilience mechanism. In *Reliable Distributed Systems Workshops (SRDSW), 2011 30th IEEE Symposium on*, pages 72–77. IEEE, 2011.

[26] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[27] James Hadley, Yehia Elkhatib, Gordon Blair, and Utz Roedig. Multibox: lightweight containers for vendor-independent multi-cloud deployments. In *Workshop on Embracing Global Computing in Emerging Economies*, pages 79–90. Springer, 2015.

[28] Sijin He, Li Guo, and Yike Guo. Real time elastic cloud management for limited resources. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 622–629. IEEE, 2011.

[29] Khaled Z Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 40. ACM, 2011.

[30] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[31] Umar Kalim, Mark K Gardner, Eric J Brown, and Wu-chun Feng. Seamless migration of virtual machines across networks. In *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–7. IEEE, 2013.

[32] Oren Laadan and Jason Nieh. Operating system virtualization: practice and experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, page 17. ACM, 2010.

[33] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[34] Wubin Li and Ali Kanso. Comparing containers versus virtual machines for achieving

high availability. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 353–358. IEEE, 2015.

[35] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar K Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 29–42, 2006.

[36] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, page 6. ACM, 2007.

[37] David Meisner, Junjie Wu, and Thomas F Wenisch. Bighouse: A simulation infrastructure for data center systems. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 35–45. IEEE, 2012.

[38] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[39] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[40] Mayank Mishra, Anwesha Das, Purushottam Kulkarni, and Anirudha Sahoo. Dynamic resource management using virtual machine migrations. *IEEE Communications Magazine*, 50(9):34–40, 2012.

[41] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 386–393. IEEE, 2015.

[42] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX Annual technical conference, general track*, pages 391–394, 2005.

[43] Nathan Regola and Jean-Christophe Ducom. Recommendations for virtualization technologies in high performance computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 409–416. IEEE, 2010.

[44] Fabian Romero and Thomas J Hacker. Live migration of parallel applications with openvz. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 526–531. IEEE, 2011.

[45] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66:105–111, 2014.

[46] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[47] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *ACM SIGPLAN Notices*, volume 50, pages 1–15. ACM, 2015.

[48] David C Van Moolenbroek, Raja Appuswamy, and Andrew S Tanenbaum. Towards a flexible, lightweight virtualization alternative. In *Proceedings of International Conference on Systems and Storage*, pages 1–7. ACM, 2014.

[49] Timothy Wood, K Ramakrishnan, J Van Der Merwe, and P Shenoy. Cloudnet: A platform for optimized wan migration of virtual machines. *University of Massachusetts Technical Report TR-2010-002*, 2010.

[50] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-

box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007.

[51] Yangyang Wu and Ming Zhao. Performance modeling of virtual machine live migration. In *Cloud computing (CLOUD), 2011 IEEE international conference on*, pages 492–499. IEEE, 2011.

[52] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[53] Yi Zhao and Wenlong Huang. Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 170–175. IEEE, 2009.