Fast delivery of virtual machines and containers : understanding and optimizing the boot operation

Thuy Linh Nguyen

THESE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPERIEURE MINES-TELECOM ATLANTIQUE
BRETAGNE PAYS DE LA LOIRE - IMT ATLANTIQUE
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique et applications

Par
Thuy Linh NGUYEN

Fast delivery of Virtual Machines and Containers: Understanding and optimizing the boot operation

Thèse présentée et soutenue à Nantes, le 24 Septembre 2019
Unité de recherche : Inria Rennes Bretagne Atlantique
Thèse N° : 2019IMTA0147

Rapporteurs avant soutenance :

Maria S. PEREZ	Professeure, Universidad Politecnica de Madrid, Espagne
Daniel HAGIMONT	Professeur, INPT/ENSEEIHT, Toulouse, France

Composition du Jury :

Président :	Mario SUDHOLT	Professeur, IMT Atlantique, France
Rapporteur :	Maria S. PEREZ	Professeure, Universidad Politecnica de Madrid, Espagne
Rapporteur :	Daniel HAGIMONT	Professeur, INPT/ENSEEIHT, Toulouse, France
Examinateurs :	Ramon NOU	Chercheur, Barcelona Supercomputing Center, Espagne
Dir. de thèse :	Adrien LEBRE	Professeur, IMT Atlantique, France

Acknowledgements

The PhD journey is not a long journey but definitely a memorable time in my life. I see myself growing both professionally and personally. I would like to express appreciation to many people for all their advice, support, and inspiration for my progress.

First of all, I would like to express my gratitude to my thesis supervisor Adrien Lebre. I have enjoyed working with him very much and to me, that was my luck to have a very nice supervisor for my PhD life. Not only he has provided me with professional guidance and strict feedback on my research work, but he has also been an inspiration for me. He always brought out the positive aspect in difficult time during my PhD. This was very important to me and it helped me to go through the hard time. I also learned from him how to write scientific articles and how to present my work to others in the most efficient way.

I was having a great opportunity to have my internship at Barcelona Supercomputing Center. I am thankful for all the discussion and efforts that my advisor, Ramon Nou, spent on me. He gave me a lot of encouragement when I got stuck in my research.

I would like to especially thank my thesis committee members for their valuable time and feedback on my manuscript.

I am so grateful for my time at INRIA, I had a wonderful time there. The institute provided me with the best working environment I could ever think of. Thanks to Anne-Claire, she has always given me the best guidance to the administration of INRIA and my doctoral school at IMT Atlantique.

Thank you to all my colleagues at STACK team. We had many interesting discussions either scientific-related matters or social-related ones at lunchtime or over coffee time. Thank you for the gaming nights that you guys tried so hard to persuade me to join the path of playing games, thank you for the hangout time after work in the hot summer days. These things made my PhD life much more enjoyable.

I am also proud and really appreciate the opportunity that I received from taking part in the BigStorage project. The project brought PhD candidates working in institutes and universities around Europe together. I met and worked with other PhD candidates and supervisors to exchange ideas, experiences and culture clashes. With many training programs and meetings, I had opportunities to improve my skills and collaborate in this small research community.

Thank you to my newborn baby. Having you at the end of my PhD brought me

exciting new experiences. It was a big motivation and full of energy as well as stress and challenging at the same time.

Finally, I cannot thank enough my family. Without their support, I cannot imagine how I can walk through all the ups and downs in my journey. They are always, and will always support and take care of me.

Contents

List of Tables

List of Figures

Publications

Virtual Machine Boot Time Model, T. L. Nguyen and A. Lebre. *In Proceeding of IEEE Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on IEEE*, March 2017

Conducting Thousands of Experiments to Analyze VMs, Dockers and Nested Dockers Boot Time, T. L. Nguyen and A. Lebre. *Research Report RR-9221, Inria Rennes Bretagne Atlantique*, November 2018

YOLO: Speeding up VM Boot Time by reducing I/O operations, T. L. Nguyen and A. Lebre. *Research Report RR-9245, Inria Rennes Bretagne Atlantique*, January 2019

YOLO: Speeding up VM and Docker Boot Time by reducing I/O operations, T. L. Nguyen, R. Nou and A. Lebre. *In Proceeding of 25th Europar conference*, August 2019

I

Introduction

In this information era, we witness the emergence of Cloud Computing as the revolutionized force for the IT industry. Taking advantage of the abundant resources combined with the rapid development of web technologies, Cloud Computing has realized the idea of "computing as a utility". Today, users can access different services powered by the cloud literally everywhere from navigating with Google Maps, and watching movies on-demand with Netflix, to having a teleconference with business partners halfway around the world using Skype. All of these services are made possible by having heavy computation handled by the cloud. Moreover, start-ups use the cloud infrastructures to materialize their ideas without the hassle of having to build and manage their own physical infrastructure.

One key technology in the development of Cloud Computing is system virtualization. System virtualization can be thought of as the abstraction of a physical object. Such abstraction allows the split of physical resources into groups of different sizes in order to share them between different "virtualized environments". This way of sharing resources among different tenants is a critical capability to utilize physical resources effectively on the massive scale of a cloud system. In Cloud Computing, different types of virtualization technologies have been proposed but Virtual Machines (VMs) and Containers are the two most important ones. A VM is the combination of different physical resources under a layer of abstraction on which users can perform their tasks. Meanwhile, containers introduce a lighter and more efficient approach to the virtualization of the physical resources [5, 6, 7, 8, 9, 10]. Without diving into details for the moment, all these studies presented the same conclusion: the performance of a container-based application is close to that of the bare metal, while there was a significant performance degradation when running the same application in a VM, especially for VM I/O accesses.

Among key operations of a cloud resource management system, the provisioning process is in charge of deploying and starting a VM (or a container as fast as possible). It is composed of three complex stages: (i) after receiving the provisioning order for a machine, a scheduler identifies an appropriate physical node to host the VM/container; (ii) the image for that machine will be transferred from a repository to the designated node; (iii) and finally, the requested VM/container is booted. To solve the resource scheduling problem of the first stage, several approaches have been proposed over the years with different scheduling policies according to the expected objective (energy saving, QoS guarantee, *etc.*,) and various methodologies in order to reduce as much as possible the computation time [11, 12]. Depending on the properties of the client's request, the availability of physical resources and the scheduling algorithm criteria, the duration of the scheduling operation can vary. For the image retrieval, *i.e.*, the second stage, most cloud solutions leverage a centralized approach where VM/container images are transferred from a centralized repository to the physical host that will be in charge of hosting the "virtualized environment". To deal with performance penalties that raises such a centralized approach, several works have focused on improving the transfer of these images over the network, leveraging techniques such as peer-to-peer image transferring, deduplication or caching method, *etc.*, [13, 14, 15, 16, 17]. The last stage consists of turning on the VM/container itself. It is noteworthy that people usually ignore the time to perform a VM/container boot process because they assume that this duration is negligible with respect to the two first ones and constant when the environment is ready (*i.e.*, the image is already present on the host). However, in reality, users may have to wait several minutes to get a new VM [18] in most public IaaS clouds such as Amazon EC2, Microsoft Azure or RackSpace. Such long startup durations have a negative impact when additional VMs/containers are mandatory to handle a burst of incoming requests: the longer is the boot duration, the bigger is the economic loss. Under resource contentions, the boot time of one VM can take up to a few minutes to finish. There is also a misconception for the boot time of containers. Although containers are said to be 'instantly' ready, they may also suffer from the interference produced by other co-located "virtualized environments". In some cases, the boot time of containers can be as long as that of VMs.

In this thesis, we show how it is critical to limit the interference that can occur when booting a VM or a container. More precisely, we present a complete performance analysis of the VM/container boot process, which shows how co-located VMs/containers impact the boot duration of a new VM or container on the same compute node. Leveraging this study, we propose a novel solution to speed up the boot process of both a VM and a container, which in return improves the efficiency of the provisioning process as a whole.

Our contributions in this thesis are as follows:

— We conducted thousands of experiments on the boot time of VMs, containers and nested containers (*i.e.*, a container running inside a VM). More precisely, we performed, in a software-defined manner, more than 14.400 experiments during 500 hours in order to analyze how does the boot time of VMs and containers react. The gathered results show that the time to perform boot process is not only affected by the co-workloads and the number of simultaneously deployed VMs/containers but also the parameters that are used to configure VMs/containers. This study has been published in a research report [19]. Besides, we leveraged this analysis to propose a VM boot time model [20]. The motivation of this work was to propose an accurate model for VM operations when researchers use cloud simulation tools to evaluate the characteristics of real cloud systems. Because this proposed model is not the main contribution of the thesis, I chose to present in the Appendix section.

— In order to mitigate the cost of the boot process, we designed *YOLO* (*You Only Load Once*), a mechanism that minimizes the number of I/O operations generated during a VM/container boot process. *YOLO* relies on the *boot image* abstraction which contains all the necessary data from a VM/Container image to perform a boot process. These boot images are stored on a fast access storage device such as memory or a dedicated SSD on each compute node. Whenever a VM/container is booted, *YOLO* intercepts all read accesses and serves them directly. The improvements *YOLO* delivers have been presented in details in a research report [21] and a shorter version of this report will be presented during the next Euro-Par conference in August 2019 [22].

The rest of this thesis is organized in 3 parts and 7 chapters. **Part II** focuses on explaining Cloud Computing. In Chapter 1, a brief history of Utility Computing from the time of mainframes to the current cloud is presented. Also, we introduce the virtualization technique as a key element in Cloud Computing and explain further the virtualization concept from the hardware virtualization to the containerization. Chapter 2 focuses on the VM/container provisioning processes. Contradiction to many beliefs, many factors can damage the actual boot process time of VMs or containers. Therefore, it is interesting to understand the current issues and solutions to this process. Furthermore, Chapter 3 presents the architectural detail and the workflow of the two widely used virtualization solutions: QEMU-KVM and Docker which were used in all experiments of our work.

Part III of this thesis describes the contributions related to the boot time optimization. Chapter 4 provides a comprehensive study on the boot time of VMs and containers under high contention scenarios. With this analysis, we understand how different factors in a system affect the boot time. In Chapter 5, we present and evaluate our novel method to improve the boot time, called *YOLO*, by mitigating the I/O contention during the boot process.

In **Part IV**, we conclude the thesis in Chapter 6 and then we give some directions for future research in Chapter 7.

II

Utility Computing and Provisioning Challenges

1

Utility Computing

Contents

In this chapter, we discuss the transformation of utility computing from the early days mainframes to the current day cloud computing era. The emerging of cloud computing is the result of a chain of improvements of technologies in many aspects of computing technologies. The core technology lies in the middle of the cloud is the virtualization, which comes in many shapes and forms. We also discuss the overall overhead of these virtualization techniques on the system.

1.1 Utility Computing: from Mainframes to Cloud Computing Solutions

Since the invention of digital computers, computing technologies have changed dramatically in the past couple of decades. From the 1950s to the 1970s, mainframe computers were the main computing technology. At that time, mainframes were extremely expensive and only a few organizations could access them. Users connected to mainframes through terminals which did not have any real processing power. This sharing scheme allowed multiple people to harness the centralized processing power of mainframes, and conceptually speaking, this could be considered as an ancestor to cloud computing.

With the advancement of networking technologies, computers are interconnected with each others. As the result, in the late 1970s, a new paradigm of computing was born called distributed computing. The idea of connecting and distributing the computation over networks of computers solved many problems in an efficient way, even better than a single supercomputer. The arrival of the Internet truly brought distributed computing to the global scale during the 1990s. Computing powers kept increasing by many orders of magnitude while being available and affordable. This motivated the evolution in the way we provided computing as an utility as predicted by John McCarthy: "computation may someday be organized as a public utility" [23]. His speech at the MIT's centennial celebration in 1961 showed his vision of what we now know as Cloud Computing, as if he had the ability to glimpse the future. The term Grid Computing showed up in the 1990s as an analogy to the electric power grid, showing that computing power is as easy to access as an electric power in the grid. Many efforts have been made in the scientific community to make use of the under-utilized resources from a network of geographically dispersed computers. Grid was originally developed as a solution to provide these resources to researchers from everywhere.

Cloud computing emerged as a solution for making computing power easily accessible to everyone with different needs, at the affordable prices. A graphic designer can request a high specifications "machine" with high-end graphic cards to handle 3D rendering tasks in a straightforward fashion. A researcher can rent multiple GPUs to train on the cloud a deep neural network model on over a million of images to classify images for a small amount of money [24]. We used to perform these tasks with our own custom built physical machines, and now we can simply request the cloud provider for the resources to run them.

Supercomputers and clusters were created to fulfill the requirements to have massive computing power for a specific objective (they were used for climate research, molecular modeling, or studying earthquakes). Then, Cloud Computing evolved out of Grid Computing to deliver abstract resources and services while Grid Computing focuses on an infrastructure to deliver storage and compute resources [1]. Both being varied distributed systems, Cloud Computing indeed relies on the infrastructure of Grid

Computing. We started to see an extension of the cloud model to the HPC area as the convergence of the two models (Figure 1.1).

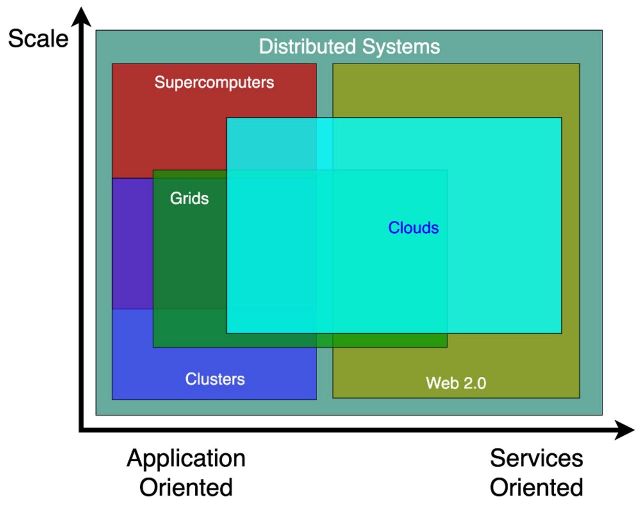


Figure 1.1 – Overview of Grid and Cloud computing, updated version of [1]

There have been many proposed definitions both academically and industrially for cloud computing. An informal definition for cloud computing describes it as a way to deliver computing services over the Internet ("the cloud"). However, a definition proposed by the American National Institute of Standards and Technology (U.S. NIST) in 2009 included major common elements that are widely used in the cloud computing community. The definition given by NIST [25] is as follows:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Like mechanical machines powering the industrial evolution that transforming the human society in the industrial age, cloud computing is the engine for today worldwide information era, providing solid infrastructure with new opportunities to disrupt various industries. Small businesses and young start-ups make use of the capabilities of cloud computing to realize their products. Netflix disrupts the video renting market by providing the online movie streaming service using Amazon Web Services (AWS) for the infrastructure. Google has just announced the new cloud gaming service called Stadia in which the cloud renders the game at the server side, and gamers don't need to own a dedicated gaming console or high-end PC to play [1]. Users can easily stream the

1. https://www.blog.google/products/stadia/stadia-a-new-way-to-play/

game just like they do with the movie or music. We live in the world that almost every modern services are powered by the Cloud.

1.2 Virtualization System Technologies: A Key Element

Virtualization technologies play an essential role in Cloud Computing infrastructures. Conceptually, virtualization is a method to consolidate different types of resources and allow multiple users to access them through a "virtual representation" of those resources. In other words, virtualization provides an abstraction layer over actual underlying resources, it creates a virtual version of a resource (like memory, storage, processor, etc.), a service, or data. A more formal definition is provided by Susanta et. al. [26]:

> "Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others"

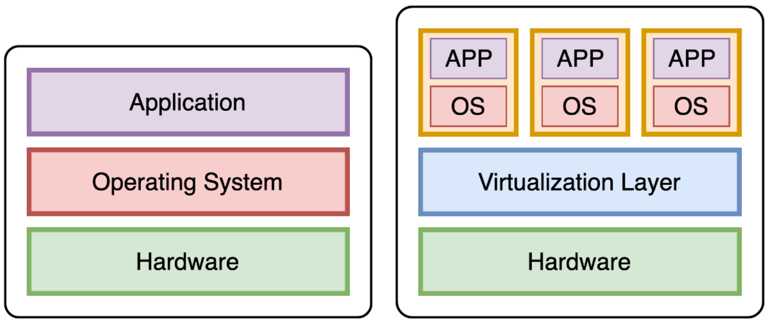
The concept of virtualization dates back to the 1960s, with solutions to allow multiple users to run programs on the same mainframe by using virtual memories and paging techniques. Furthermore, as the cloud emerges in recent years, virtualization has matured rapidly and has been applied to various aspect of computing (CPU, memory, storage, network). There are basically countless usage patterns of users on a cloud system. Some want to have as many memory as possible in which they will use as a cache for the conntent of their website, or to perform data analytics on big data using in-memory computing framework (Spark). Some want to train a machine translation model using a cluster of GPUs in a short amount of time without having to buy and setup the cluster themselves. Some don't even want to have a whole VM but only the capability for running custom functions at scale. Virtualization of those resources is the solution to share and manage all the accesses to the underlying machines in a cloud system in order to satisfy the diversified requests from users.

Virtualization allows abstraction and isolation of lower level functionalities and underlying hardware. This enables portability of higher level functions and sharing and/or aggregation of the physical resources. The different virtualization approaches can be categorized into: application virtualization, desktop virtualization, data Virtualization, network Virtualization, storage Virtualization, hardware Virtualization, OS-level Virtualization.

In this thesis, we study two virtualization technologies that has become widely used: hardware Virtualization (*virtual machines*) and OS-level Virtualization (*containers*). We provide the background of these two technologies in the next sections.

1.3 Hardware Virtualization

Hardware Virtualization enables us to run multiple operating systems on a single physical computer by creating virtual machines that act like real computers with an operating system inside. Software and applications executed on the virtual machines are separated from the underlying hardware resources. Today, hardware virtualization is often called server virtualization, hypervisor-based virtualization or, simply, virtualization.



(a) Traditional architecture (b) Virtual architecture

Figure 1.2 – Traditional and Virtual Architecture

Figure 1.2 shows the model of hardware virtualization, where the essential difference to the traditional one is a virtualization layer. In state-of-the-art virtualization systems, this layer is a software module called a **hypervisor** or also known as Virtual Machine Monitor (VMM), which works as an arbiter between a VM's virtual devices and the underlying hardware. Hypervisor creates a virtual platform on a host computer, where multiple operating systems, which are either multiple instances of the same or different operating systems, can share the hardware resources offered by the host. This virtual environment is not only providing a sharing resources but also performance isolation, and security between running VMs. However, having to consult the hypervisor each time a VM makes a privileged call introduces a high overhead in the VM performance as the hypervisor must be brought online to process each request. This overhead can be mitigated depending on different virtualization mechanisms. Currently, we have two approaches to provide hardware virtualization: *software-emulated virtualization* and *hardware-assisted virtualization*.

1.3.1 Types of Hardware Virtualization

Software-Emulated Virtualization

In this solution, the hypervisor is responsible for the instruction emulation from VMs to physical devices. There are two distinct types of this virtualization scheme: full virtualization and para-virtualization.

Full Virtualization is where the hypervisor holds the responsibility for the emulation of the instruction from VMs to physical devices. In other words, a virtual machine can run with an unmodified guest operating system (using the same instruction set as the host machine) and it is completely isolated. A hypervisor emulates the physical hardware by translating all instructions from the guest OS to the underlying hardware. Full virtualization brings the compatibility and flexibility as a VM can run any OS with corresponding drivers and does not require any specific hardware, it also offers the best isolation and security for the virtual machines. However, this method has a bad performance due to the emulation of the hardware devices.

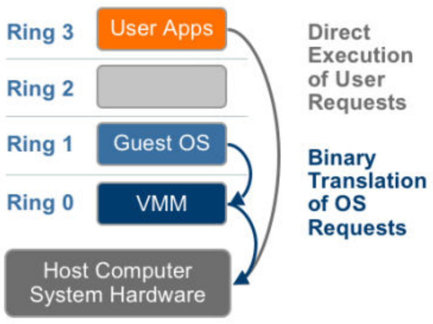

Ring 3
User Apps
Ring 2
Ring 1
Guest OS
Ring 0
VMM
Host Computer System Hardware
Direct Execution of User Requests
Binary Translation of OS Requests

Figure 1.3 – The binary translation approach to x86 virtualization [2]

Figure 1.3 depicts the way a hypervisor combines the binary translation and the direct execution techniques to achieve full virtualization for CPU access. While a normal user level code is directly executed on the processor for high performance, a hypervisor has to translate kernel code to replace non-virtualizable instructions with new sequences of instructions that have the intended effect on the virtual hardware. Each hypervisor provides each VM with all the services of the physical system, including a virtual BIOS, virtual devices and virtualized memory management.

Para-virtualization is different compared to full virtualization because the hypervisor does not need to emulate the hardware for the VM. The VM is aware that it is running in a virtualized environment and it access hardware devices “directly” through special drivers, obtaining a better performance when compared to full virtualization. However, the guest OS kernel must be modified in order to provide new system calls. This modification increases the performance because it reduces the CPU consumption but, at the same time, it reduces the security and increases the management difficulty.

Figure 1.4 describes the Para-virtualization mechanism when a VM want to access to CPU core. Para-virtualization provides hypercalls for guest OS to communicate with the VMM directly. The hypervisor also provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping.

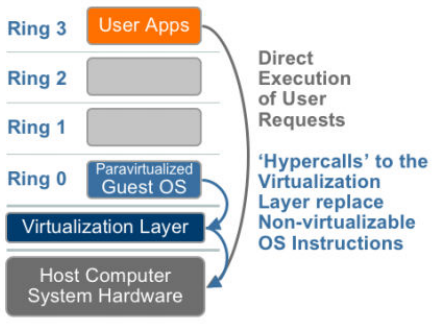

Ring 3
User Apps
Ring 2
Ring 1
Ring 0
Paravirtualized Guest OS
Virtualization Layer
Host Computer System Hardware
Direct Execution of User Requests
'Hypercalls' to the Virtualization Layer replace Non-virtualizable OS Instructions

Figure 1.4 – The Para-virtualization approach to x86 Virtualization [2]

Hardware-Assisted Virtualization

Hardware-assisted virtualization relies on special hardware to allow the instructions generated from guest OS to be directly executed on physical hardware. Because the X86 processor did not have such facilities available in its original design, this type of virtualization was used on the virtualization systems only from the 2000s when Intel and AMD introduced new level to the processor for the first time.

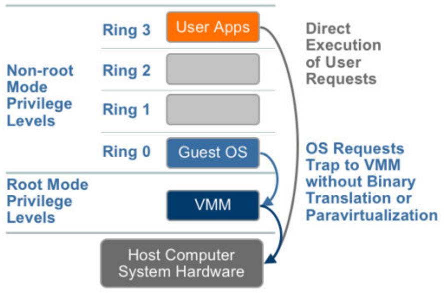

Ring 3
User Apps
Non-root Mode Privilege Levels
Ring 2
Ring 1
Ring 0
Guest OS
Root Mode Privilege Levels
VMM
Host Computer System Hardware
Direct Execution of User Requests
OS Requests Trap to VMM without Binary Translation or Paravirtualization

Figure 1.5 – The hardware assist approach to x86 virtualization [2]

As depicted in Figure 1.5, hardware vendors add a new privilege level to processor. This level is a new root mode, stays below ring 0. In the new privilege frame, when the guest OS attempts to perform the privileged operations, traps will be automatically raised to VMM without any binary translations. The new level lets the VMM safely and transparently uses direct execution for VMs to increase the performance of VMs. Moreover, the guest OS remains unmodified.

1.3.2 Discussion

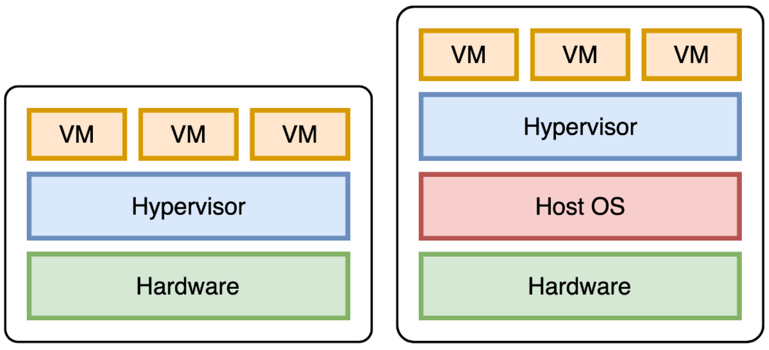
All three above approaches have their own advantages and drawbacks. Full virtualization requires neither the OS modification nor the hardware modification, hence has the best compatibility. As VMWare has declared, full virtualization with binary translation is currently the most established and reliable virtualization technology available [2].

And it will continue to be a useful technique for years to come. But the software implemented binary translation still has its inherence problems, such as memory accessing overhead, host CPU execution overhead. This is the inherence limitation of software approaches. So hardware assisted virtualization is where virtualization is going with pure software virtualization being a performance enhancing stopgap along the way.

Whatever the type of hardware virtualization, we always need a hypervisor to make the communication between the VM and the underlying hardware. The following sections will discuss two main elements inside a hardware virtualization environment: the hypervisor and the virtualized interfaces.

1.3.3 Hypervisor

Hypervisor is commonly classified as one of these two types, as show in Figure 1.6.



(a) Hypervisor Type 1 (b) Hypervisor Type 2

Figure 1.6 – 2 types of hypervisors

Type 1 - Bare-metal Hypervisor is also referred to as a "native" or "embedded" hypervisors in vendor literature. Type 1 hypervisor runs directly on the host's hardware, meaning that the hypervisor has direct communication with the hardware. Consequently, the guest operating system runs on a separate level above the hypervisor. Examples of this classic implementation of virtual machine architecture are Xen, Microsoft Hyper-V, VMWare ESX.

Type 2 - Hosted Hypervisor runs as an application on a host operating system. When the virtualization movement first began to take off, Type 2 hypervisors were most popular used. Administrators could buy the software and install it on a server they already had. Some well-known examples of hosted hypervisor are VMWare Server and Workstation, QEMU, Microsoft Virtual PC, and Oracle VM VirtualBox. Full virtualization uses hosted hypervisor to manage VMs.

Type 1 hypervisors are gaining popularity because building the hypervisor into the firmware has been proved to be more efficient. According to IBM, Type 1 hypervisors provide higher performance, availability, and security than Type 2 hypervisors (IBM recommends that Type 2 hypervisors should be used mainly on client systems where efficiency is less critical or on systems where support for a broad range of I/O devices is important and can be provided by the host operating system). Experts predict that shipping hypervisors on bare metal will impact how organizations purchase servers in the future. Instead of selecting an OS, they will simply have to order a server with an embedded hypervisor and run whatever OS they want.

1.3.4 Virtualized Interfaces

Many important operations in virtualized systems suffer from some degree of virtualization overhead. For example, in both full virtualization and paravirtualiztion, each time a VM encounters a memory page fault the hypervisor must be brought on the CPU to rectify the situation. Each of these page faults consists of several context switches, as the user space process is switched to the guest kernel, the kernel to the hypervisor, and sometimes the hypervisor to the host kernel. Compare this to a bare-metal operating system that generally has only two context switches: the user space process to kernel process and back again. Disk access has similar problems. It is fairly intuitive that the higher number of context switches and their associated operations can impart considerable overhead on privileged calls as each call now consumes considerably more CPU cycles to complete. As stated earlier, the hypervisor is necessary as it is required to operate between running VMs and the hardware for both performance isolation and security reasons.

The virtualized interfaces are concrete implementation of the two types of hardware virtualization techniques mentioned in Section 1.3. There are two types of virtualized interfaces in a virtualized environment: software-based interfaces and hardware-assisted virtual interfaces.

Software Interfaces

The virtual interfaces are generally considered to be in two classes: device emulation (fully virtualized) and paravirtualized devices.

Emulation of hardware devices is performed by the hypervisor. Since guest OS in VM only sees the emulated hardware, the VM can basically run on any hardware. However, emulation comes with a huge performance issue because the hypervisor needs to translate the communication of the VM and its emulated devices to the real physical hardware and back. For example, to emulate the CPU, the hypervisor has to capture all instructions sent to the processor by the VM, then translate them to use the real instruction set of the current physical CPU. After the CPU has finished the task, the hypervisor has to translate the result to the VM. In case of disk I/O from the guest OS, we can emulate the hard disk for a VM by mapping the I/O request addresses from the guest OS to the physical addresses and perform the read/write on the VM disk file on the host machine.

Para-virtualization lets the VM have special access to the hardware by using a modified physical hardware interface provided by the hypervisor. The guest OS has to be modified to make use of the paravirtualized interfaces. One of the most well-known implementation is `virtio` [27], which is used by KVM to provide paravirtualized devices to VM. We have `virtio-blk` (paravirtualized block device) and `virtio-scsi` (paravirtualized controller device) provide efficient transport for guest-host communication which improve the I/O performance of VM to hard disks, while `virtio-ballon` and `virtio-mem` tackle the problem of hot plug/unplug virtual memory for VMs.

Hardware-Assisted Interfaces

These interfaces are support by hardware companies. They add a new privilege level to the physical hardware devices so that a hypervisor can safely and transparently uses direct execution for VMs. The most known hardware-assisted interfaces are:

CPU: Intel VT-x [28], **AMD-V** [29]

They are two independent but very similar technologies by Intel and AMD which are aimed to improve the processor performance for common virtualization challenges like translating instructions and memory addresses between VM and the host. A VM can generate the instructions to change the state of system resources or the instructions executed by a program on a VM reveal that they were executed on a VM since the results differ from those when they are executed on the physical machine (e.g, *htop* command). These instruction can become the serious problem for the hypervisor and guest system. Both Intel VT-x and AMD-V were developed in response to this problem. It allows the hypervisor can execute these kind of instructions on behalf of the program.

Memory: Second Level Address Translation (SLAT)

A VM is allocated with virtual memory of the host system that serves as a physical memory for the guest system. Therefore, the memory address translation has to perform twice – inside the guest system (using software-emulated shadow page table), and inside the host system (using hardware page table). Nested paging or Second Level Address Translation (SLAT) is a hardware-assisted virtualization technology developed to overcome overhead of hypervisor shadow page tables operations. Intel's extended page tables (EPT) [30] and AMD's Rapid Virtualization Indexing (RVI) [31] are implementations of the SLAT technology. Using SLAT, the two levels of address space translations required for each virtual machine is performed in hardware, reducing the complexity of the hypervisor and the context switches needed to manage virtual machine page faults.

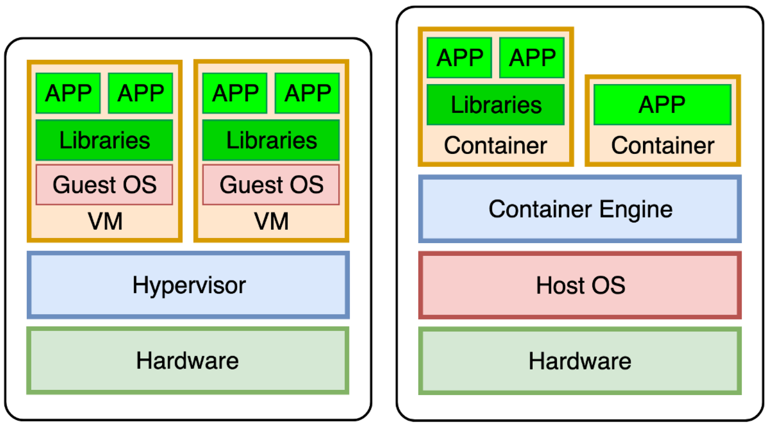
Network: Virtual Machine Device Queues (VMDq) [32] and **Intel Data Direct I/O Technology (Intel DDIO)** [33]

They are devices focus on reducing the interrupt requests and remove the extra packet copy which happen when using a virtual NIC. When a VM transfers data through the network, the hypervisor is responsible for queueing and sorting the packets. VMDq moves packet sorting and queues out of the VMM and into the network controller hardware and allows parallel queues for each virtual NIC (vNIC). Intel DDIO allows the network data to exchange between the CPU and NIC directly without moving those packets to and from memory which help to reduce latency and enhancing bandwidth.

I/O: Single Root I/O Virtualization (SR-IOV) [34] and **IO memory management unit (IOMMU)**

Single Root I/O Virtualization (SR-IOV) [34], developed by the PCI-SIG (PCI Special Interest Group), provides direct access between the devices and the VM. SR-IOV can share a single device to multiple VMs. Also, IO memory management unit (IOMMU) allows guest VM to directly use peripheral devices through Direct Memory Access (DMA) and interrupt remapping.

1.4 OS-level Virtualization (or Containerization)



(a) Hardware Virtualization (b) Containerization

Figure 1.7 – Virtualization Architecture

Containerization, also called container-based virtualization or application containerization, is an OS-level virtualization method for deploying and running distributed applications without launching an entire VM for each application. Therefore, containerization is considered a lightweight alternative to full machine virtualization. Figure 1.7 illustrates the differences between containers and VMs: (1) Containers run on a single control host and access a single kernel (Figure 1.7b), (2) VMs require a separate guest OS (Figure 1.7a). Because containers share the same OS kernel as the host, containers can be more efficient than VMs in term of performance. Essentially, containers are processes in the host OS that can directly call the kernel functions without performing many context switches as in the case of VMs.

The earliest form of a container dates back to 1979 with the development of `chroot` in 1979 in Unix V7, which created an early process isolation solution. There is no advancement in this field until the 2000s when FreeBSD Jails allows a computer system to be divided into multiple independent smaller systems. Then Linux VServer [35] used that jail mechanism to partition resources, which implemented by patching the Linux kernel. Solaris Containers [36], released in 2004, allows system resource controls and boundary separation provided by zones. In 2005, Open VZ [37] patched a Linux kernel to provide virtualization, isolation, resource management and checkpointing, however, it is not merged to the official Linux kernel. A major advancement in container technology happened when Google launched Process Containers [38] in 2006. It was later renamed to Control Groups - `cgroups` - and merged to Linux kernel 2.6.24. `cgroups` can limit and isolate resources of a group of processes. Linux Containers (LXC) [39] was among the first implementation of a Linux container manager when it was introduced in 2008 that uses `cgroups` and `namespaces`. In the 2010s, Cloud Foundry with Warden and Google with Let Me Contain That For You (LMCTFY) are different solutions that tried to improve the adoption rate of the container technology. When Docker [40] emerged in 2013, containers gained huge popularity. At first, Docker also used LXC and later replaced with `libcontainer`. Docker can stand out from the rest dues to the whole container management ecosystem it brings to users.

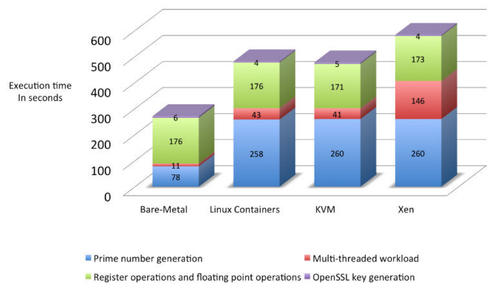
Two most popular container technologies nowadays are **LXC** and **Docker**. Both utilize the Linux `cgroups` and `namespaces` in the Linux kernel to create an isolation environment for the containers. Essentially, Linux containers are just isolated processes with controlled resources running on a Linux machine. `cgroups` is a kernel mechanism for limiting and monitoring the total resources used by a group of processes running on a system. While `namespaces` are a kernel mechanism for limiting the visibility on the system's resources that a group of processes has over the rest of a system. Accordingly, `cgroups` manages resources for a group of processes, whereas `namespaces` manages the resource isolation for a single process.

1.5 Virtualization Overhead

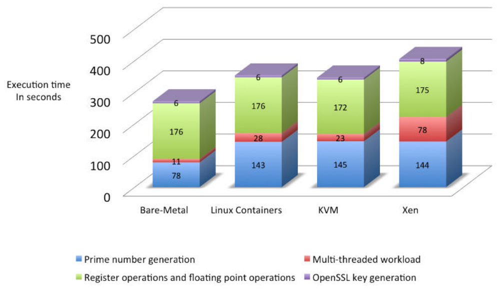
While hypervisor-based technology is the current virtualization solution widely used in a cloud system, the container-based virtualization starts receiving more attention for being a promising alternative. Although container offers near bare metal performance and is a lightweight and faster solution compared to VM, both of these virtualization solutions still rely on sharing the host's resource. They may suffer from performance interference in multi-tenant scenarios and their performance overheads would lead to negative impacts on the quality of cloud services.

To help fundamentally understand the overhead of these two types of virtualization solutions, we do a survey among studies that measure the overhead and compare the performance between VM, container and bare-metal. The results show that although the container-based solution is undoubtedly lightweight, the hypervisor-based technology does not come with higher performance overhead in every case.

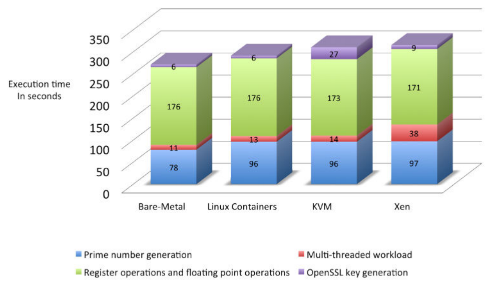
1.5.1 CPU Overhead



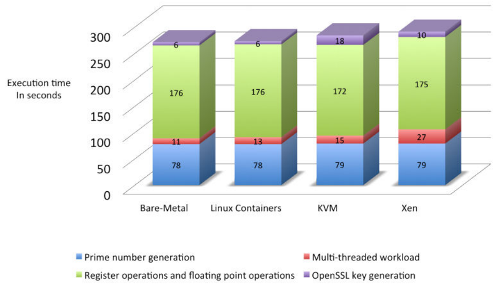
(a) 1 Virtual CPUs



(b) 2 Virtual CPUs



(c) 4 Virtual CPUs



(d) 8 Virtual CPUs

Figure 1.8 – CPU Virtualization Overhead [3]

In Kumar's thesis [3], he compared the CPU overhead between hardware virtualization (using both Xen and QEMU/KVM), containerization (LXC), and bare-metal. The objective of the CPU test suite is to measure the execution time of the sample application and its individual tasks. His result depicts in Figure 1.8, which shows that Linux Containers and QEMU/KVM perform the best for both single-threaded and multi-threaded workloads, exhibiting the least overhead compared to the bare-metal performance. Though Xen performed identically to the others for single-threaded workloads, it exhibited relatively poor performance when scheduling multi-threaded workloads.

Other study [9] shows the difference in performance for CPU intensive workloads when running on VMs vs. LXCs is under 3% (LXC fares slightly better). Thus, the hardware virtualization overhead for CPU intensive workloads is small, which is in part due to virtualization support in the CPU (VMX instructions and two dimensional paging) to reduce the number of traps to the hypervisor in case of privileged instructions.

1.5.2 Memory Overhead

To evaluate the virtualization overhead associated with memory access, Kumar [3] created a sample application to allocate two arrays of a given size, and then copies data from one to the other. The reported "bandwidth" is the amount of data copied, over the time required for the operation to complete. Then, he measured the memory access bandwidth available to the virtual machine and bare-metal host. The results shows that Linux Containers and KVM yielded higher memory bandwidth than Xen.

The author in [9] measured the performance of Redis in-memory key-value store under the YCSB benchmark. For the load, read, and update operations, the VM latency is around 10% higher as compared to LXC.

1.5.3 Network Overhead

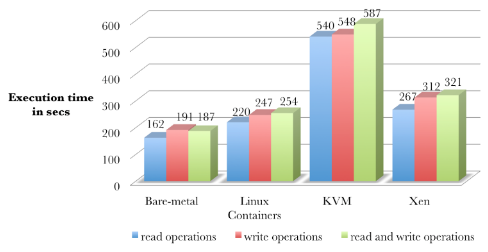
Kumar [3] measured the network bandwidth available to a virtual machine in both NAT and bridged configurations. To ensure a fair comparison, the author set up an `iperf` server on a machine outside of the test network and measured the network bandwidth by running an `iperf` client on each of VM/container. His results shows that there is no observable overhead introduced when virtualizing network interfaces hardware between virtualization (using both Xen and QEMU/KVM), containerization (LXC) and bare-metal.

The authors [9] also have the same conclusion, they do not see a noticeable difference in the performance between the two virtualization techniques when using the RUBiS benchmark to measure network performance of guests.

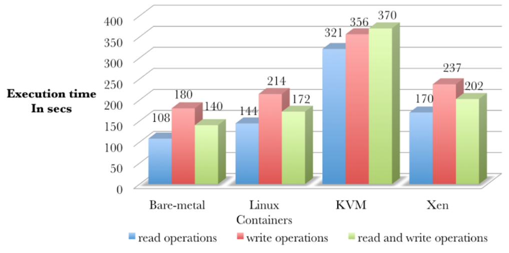
1.5.4 Disk I/O Overhead

Kumar [3] evaluated the overhead introduced when virtualizing disk I/O by performing a set of sequential and random disk I/O. He measured the execution time of the sample application and its individual tasks. The results summarized in Figure 1.9 confirm that KVM exhibiting the highest overhead, and Linux Containers exhibiting the least. Based on these results, the author concludes that Linux Containers perform the best with respect to virtualizing disk I/O operations.

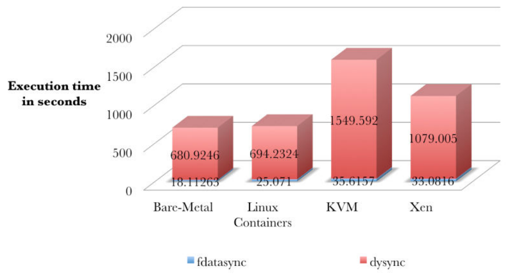
The author [9] uses filebench randomrw workload which issues lots of small reads and writes, and each one of them has to be handled by a single hypervisor thread. VirtIO is used as I/O virtualized interface for VMs. Their results shows that the disk throughput and latency for VMs are 80% worse than Linux containers.



(a) Sequential File I/O



(b) Random File I/O



(c) Disk Copy Performance

Figure 1.9 – Virtualization Overhead - I/O Disk (reads/writes) [3]

1.6 Summary

We have provided an overview of Cloud Computing from its beginning. A key component in the Cloud Computing technology is the virtualization, which comes in many different shapes and sizes. Even though virtualization brings many benefits to the Cloud system, it does not come without any trade-offs. In this chapter, we also discussed various overheads of virtualization. After having a general understanding of the concept of Cloud Computing, we present a crucial operation happens within a cloud system - the provisioning process, and we examine this operation in great detail in the following chapter.

2

IaaS Toolkits: Understanding a VM/Container Provisioning Process

Contents

*After drawing a high level picture about cloud computing and the role of virtualization technology, we focus in this chapter on the provisioning process of a cloud system (*i.e.*, the process to allocate cloud provider's resources and services to a customer). Because the provisioning process consumes resources, other running applications in the system can impact on this process duration. Besides, if we need an additional VM or container for a task, and the time to have that virtual environment ready is longer than the time that task finished, this leads to resource waste and unnecessary cost for the system. Therefore, optimizing the provisioning process is crucial in providing a better experience for users as well as improving the overall operations of the cloud system.*

2.1 Overview

Cloud provisioning is the allocation of a cloud provider's resources or services to a customer on demand and makes it available for use. In a typical cloud system, when a user requests to start a VM with specific resources, these following steps of a provisioning process are performed:

- Step 1: The scheduler identifies a suitable compute node for the VM/container based on the user requirements.
- Step 2: VM/container image is transferred from the repository to the designated compute node.
- Step 3: The VM/container starts its booting process on the compute node.

All of these 3 steps are involved in the deployment duration of a new VM or container in a cloud platform. Depending on the properties of the client's request (i.e., VM/container type, resources, start time, etc.), the availability of physical resources and the scheduling algorithm purpose (i.e., energy saving, resources usage, or QoS guarantee, etc.), the duration of operation in Step 1 can vary. In Step 2, the size of image, the I/O throughput on both compute node and repository and the network bandwidth between the compute node and the repository are important factors. Researchers usually consider that a VM launching process time takes place mostly in this stage and they try to speed it up [41, 42]. In Step 3, because the environment for the VM/container is ready, researchers assumed that the boot process utilise little resources and the time to boot that VM/container is negligible and can be ignored.

Step 1 is a well-known problem in cloud computing and there are thousand solutions according to the different purposes. Because the duration of Step 1 relies on the scheduling algorithm itself, this amount is not considered in total of a VM deployment time, and we do not focus on this step in our thesis. In general, the startup time or also known as launching time of a VM/container is consider as the last two steps. The previous works often skip the duration of step 3 [18, 41] or naively use a constant number to represent Step 3 duration [43]. Meanwhile, Step 3 may have significant effect on the total startup time of a VM as we explain further in part III. In the following sections, we introduce more detail about Step 2 and Step 3, we describe what happens in each Step and the current issues and solutions.

2.2 Step 2: VM/Container Image Retrieval

2.2.1 Retrieving Process

Generally, the images of VM/Container are stored in a centralized repository in most cloud system. Therefore in a provisioning process, we have to transfer the images from the repository to the assigned compute node before start a new VM/Container. Moving the image through out the network when deploying a machine puts a significant pressure on the network capacity. In a shared environment, the degradation in performance of the network can have critical impact on the experience of other users in the system. When serving the images, the repository suffers from the workload on its I/O to retrieve the images from its local storage. In case the images are compressed before sending, the repository has to perform the compression task which is quite CPU heavy. This problem is more severe in case there are simultaneously deploying requests from users where the images need to be transferred to multiple physical nodes at the same time. Moreover, the compute node is also stressed on its CPU and IO.

2.2.2 Retrieving Process Issues and Solutions

The image of a VM/Container is, in fact, heavy in size and abundant. Each user is able to create or upload their own images. As a result, a number of storage nodes are dedicated to store images. When receiving user requests to create a new VM/Container, the image will be transferred from the storage nodes to the compute node, and this process becomes a burden to the cloud infrastructure. A lot of efforts focused on mitigating the penalty of the VM images (VMI) transferring time either by using deduplication, caching and chunking techniques or by avoiding it thanks to remote attached volume approaches [13, 14, 15, 44, 45]. On the contrary, there are only a few studies on improving container image transferring [16, 17, 46]. We give a summary of all techniques that present in these works in Table 2.1.

Table 2.1 – Summary methodologies to transferring images

		Chunking	Deduplication	Caching	Peer-to-Peer	Lazy loading
VM	[13]	x	x			
	[14]			x		
	[44]		x		x	
	[15]			x		
	[45]				x	
Container	[16]					x
	[17]				x	
	[46]				x	

VM Image Transferring

In this work, the authors [13] use deduplication technique on identical parts (chunks) of the images to reduce the required storage for VM disk images. They conducted extensive evaluations on different sets of virtual machine disk images with different chunking strategies. Their results show that the amount of stored data grows very slowly after the first few virtual disk images, which have similar kernel versions and packaging systems. In contrast, when different versions of an operating system or different operating systems are included, the deduplication effectiveness decreases greatly. They also show that fixed length chunks work well compared to variable-length chunks. Finally, by identifying zero-filled blocks in the VM disks, they can achieve significant savings in storage.

Kangjin et al [44] propose a novel approach called the Marvin Image Storage (MIS) has which efficiently stores virtual machine disk images using a content addressable storage. For this purpose, the disk image is split in a manifest file which contains metadata information of each file in the image and the actual file content stored in the MIS data store. By using special Copy-On-Write layers, the MIS can reuse a virtual machine disk image as a shared base image for a number of virtual machines to further reduce the storage requirements. It also offers a fast and flexible way to apply updates. Furthermore, the MIS offers advanced features like hard link detection and algorithms to merge and diff image manifests, directly mount disk images from the store, an efficient way to apply updates to a disk image and the possibility to apply filters to remove sensitive content. The presented evaluation has shown that the storage requirements could be reduced by up to 94% of the original images. However, because they adopted the deduplication at the file level, they have to check the duplicated file content against the data store with each file in every VM image. This is an intensive CPU task which may effect the image server, especially in case the image server is transferring VM image to compute nodes.

Machine image templates are large in size, often ranging in tens of Gigabytes, thus, fetching image templates stored in centralized repositories results in long network delay. A solution to replicate the image repositories across all hosting centers is expensive in terms of storage cost. Therefore, a solution - called DiffCache [14] - which maintains a cache collocated with the hosting center to mitigate such latency issue is proposed. Generally, there is a high percentage of similarity between image templates, and this feature has been exploited in optimizing storage requirement of image repository by storing only common blocks across templates. DiffCache algorithm that populates the cache with patch and template files instead of caching large templates. A patch file is the difference between two templates, and if the templates are highly similar to each other then this patch file is rather small in size. As a result, DiffCache minimizes the network traffic, and leads to significant gain in reducing service time when compared to standard caching technique of storing template files. When the template and the patch file are in cache, then a new template can be generated by using the in-cache template and patch.

The key observation from the tests of the authors in [15] is that VMs actually read only small fractions of the huge VMI during the boot process, with $\approx$200 MB being the biggest size observed from a Windows Server 2012 image. Therefore, they proposed VMI caches, as an extension to QCOW2 format, that can significantly reduce the amount of network traffic for booting a VM. The authors made use of the characteristics of the copy-on-write mechanism to populate the VMI caches during the boot process of VMs. This cache is chained, and positioned between the base image and the COW layer. Whenever a VM needs the boot data from the base image, it can read from the warmed cache, which reduce the IO to the base image and speed up the process.

Nicolae et al. [45] introduced a novel multi-deployment technique based on augmented on demand remote access to the VM disk image template. Since the IO is performed on-demand, it prevents bottlenecks due to concurrent access to the remote repository. The authors organized VMs in a peer-to-peer topology where each VM has a set of neighbors to fetch data chunks from. The VM instances can exchange chunks asynchronously in a collaborative scheme similar to peer-to-peer approaches. The scheme is highly scalable, with an average of 30-40% improvement in read throughput compared to simple on-demand schemes.

Container Image Pulling

Slacker [16] is a new Docker storage driver utilizing lazy cloning and lazy propagation to speed up the container startup time. Docker images are downloaded from a centralized NFS store and only a small amount of data needed for the startup process of the container is retrieved. Other data is fetched when needed. All container data is stored in the NFS server which is shared between all the worker nodes. However, this design tightens the integration between the registry and the Docker client as clients now need to be connected to the registry at all times (via NFS) in case additional image data is required.

CoMICon [17] proposes a cooperative management system of Docker images on a set of physical nodes. In each node, only a part of images is stored. CoMICon uses peer-to-peer (P2P) protocol to transfer layers between nodes. When an image is pulled, CoMICon tries to fetch a missing layer from a closest node before pulling from a remote registry.

FID [46] is a P2P-based large-scale image distribution system, which integrates the Docker daemon and registry with BitTorrent. A Docker image is stored in the Docker Registry as two static files: the manifest and the blobs. Blob is a compressed file of the layer. When images are pulled, the blobs are downloaded using P2P. For each blob, a torrent file is created and seeded to the BitTorrent network. Because BitTorrent is used to distribute images, it exposes Docker clients to other nodes in the network which can become a security issue.

2.3 Step 3: Boot Duration

In this section, we describe a VM and container boot process so that readers can understand clearly the different steps of the boot operation. From that, we can have an idea of the level of influence of these factors on the boot time of a VM or container.

2.3.1 Boot Process

VM Boot Procces

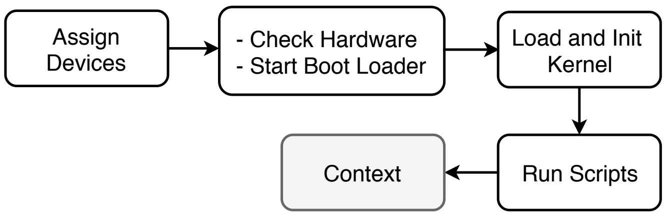


Figure 2.1 – VM boot process

Figure 2.1 illustrates the different stages in a VM boot process. During a VM boot operation, a standard OS boot process happens. First, the hypervisor is invoked to assign resources (*e.g.*, CPU, memory, disk storage) to the VM. After that, BIOS checks all the devices and tests the system, then BIOS loads the boot loader into memory and gives it the control. Boot loader (GRUB, LILO, etc.) is responsible for loading the OS kernel. Finally, the OS kernel starts the configured services such as SSH. The last step is made based on client requirements. A VM boot process generates both read and write operations: it loads the kernel files from the image into memory and writes the data (logs, temporary files, etc.).

Container Boot Process

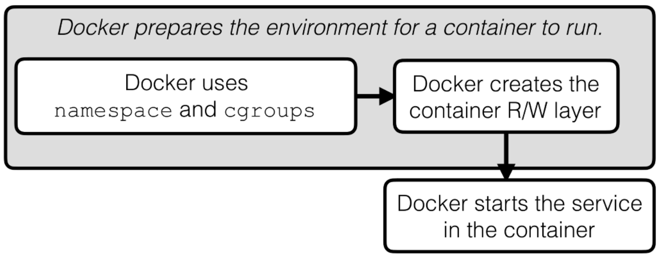

Docker prepares the environment for a container to run.
Docker uses namespace and cgroups
Docker creates the container R/W layer
Docker starts the service in the container

Figure 2.2 – Container boot process

Although we use the words *container boot process* in comparison with the hardware virtualization system terminology, it is noteworthy that a container does not technically boot, but rather start. The overview of the container boot process is depicted in Figure 2.2. Booting a docker starts when the *dockerd* daemon receives the container starting request from the client. After verifying that the associated image is available, *dockerd* utilizes `cgroups` and `namespace` to prepare the container layer structure, initializes the network settings, performs several tasks related to the specification of the container and finally gives the control to the *containerd* daemon. *containerd* is in charge of starting the container and managing its life cycle.

Summary

CPU, memory, and IO resources from the compute node are required to achieve the boot process. Consequently, workloads or VMs/containers that are already executed on the compute node can significantly increase the VM/container boot time and should be considered in a boot operation.

2.3.2 Boot Process Issues and Solutions

The promise of elasticity of cloud computing brings the benefits for clients to add and remove new VMs in a manner of seconds. However, in reality, users must wait several minutes to start a new VM in the public IaaS cloud such as Amazon EC2, Microsoft Azure or RackSpace [18]. Such long startup duration has a strong negative impact on services deployed in a cloud system. For instance, when a web service faces spontaneously increasing workloads in the high sale season, they need to add new VMs temporarily. The websites may be unreachable if the new VMs are only available after a few minutes, leading to unsatisfied clients and a loss of revenue for the site operators. Therefore, the startup time of VMs is also essential in provisioning resources in a cloud infrastructure.

While a lot of efforts focused on speeding up the VMI transferring time, there are only few works that focus on the startup/boot operation. To the best of our knowledge, the solutions that have been investigated rely on the VM cloning technique [47, 48, 49, 50] or the suspend/resume capabilities of VMs [51, 52, 53, 54]. The cloning solutions require to keep a live VM on a host to spawn new identical VMs so that they skip the whole VM boot process. Moreover, after cloning, VMs need to be reconfigured to get specific parameters such as IPs or MAC addresses. With the resuming technique, the entire VM state is suspended and resumed when necessary. This mechanism has to store a significant number of VMs due to the variety of requested applications and configurations. In our discussion, we analysed the works that focus on improving the VM booting phase by using two techniques: *cloning* and *resuming*.

Cloning

SnowFlock [47] and Kaleidoscope [48] are similar systems that can start stateful VMs by cloning them from a parent VM. SnowFlock utilises lazy state replication to fork child VMs which have the same state as a parent VM when started. Kaleidoscope has introduced a novel VM state replication technique that can speed up VM cloning process by identifying semantically related regions of states.

Potemkin [49] uses a process, called flash cloning, which clones a new VM from a reference image in the compute node by copying the memory pages. To create the reference image, Potemkin initiates a new VM then snapshot the VM memory pages. After changing its identity (*i.e.*, IP address, MAC address, etc.), the newly cloned VM is already ready to run without going through the VM boot process. Potemkin presents an optimization by marking the parent VM memory pages as copy-on-write and shares these states to all child VMs without having to physically copying the reference image. On the contrary, Potemkin can only clone VMs within the same compute node. Moreover, the authors restrict their system to have only one combination of operation system and application software, which is not very useful for a real cloud system.

Wu et al. [50] perform live cloning by resuming from the memory state file of the original VM, which is distributed to the compute nodes. The VM is then reconfigured by a daemon inside each cloned VMs that load the VM-metadata from the cloud manager. These systems clones new VMs from a live VM so that they have to keep many VMs alive for the cloning process. This method also suffers from the downside of the cloning technique, as discussed previously.

Resuming

Several works [51, 52, 53, 54] attempt to speed up VM boot time by suspending the entire VM's state and resuming when necessary. To satisfy various VM creation requests, the resumed VMs are required to have various configurations combined with various VMIs, which leads to a storage challenge. If these pre-instantiated VMs are saved in a different compute node or an inventory cache and then they are transferred to the compute nodes when creating VMs, this may place a significant load on the network.

VMThunder+ [55] boots a VM then hibernates it to generate the persistent storage of VM memory data. When a new VM is booted, it can be quickly resumed to the running state by reading the hibernated data file. The authors use hot plug technique to re-assign the resource of VM. However, they have to keep the hibernate file in the SSD devices to accelerate the resume process. Razavi et al. [56] introduce prebaked μVMs, a solution based on lazy resuming technique to start a VM efficiently. To boot a new VM, they restore a snapshot of a booted VM with minimal resources configuration and use their hot-plugging service to add more resources for VMs based on client requirements. The authors only evaluated their solution by booting one VM with μVMs on a SSD device. However, VM boot duration is heavily impacted by the number of VM booted concurrently as well as the workloads running on a system [20], thus, their evaluation is not enough to explore the VM boot time in different environments, especially, under high I/O contention.

A recent development in lightweight virtualization combines the performance aspect of Containers technology with the better isolation capability and security advantage of VMs. One of the advancement is the Kata Containers [57] project, which is managed by the OpenStack Foundation. Kata Containers incorporates two technologies Intel Clear Containers and Hyper.sh runV to introduce the lightweight VMs which run one container inside [58]. In order to reach the boot time of a container, the lightweight VM uses a minimal and optimized guest OS and kernel. Moreover, Kata Containers uses a specific version of QEMU called `qemu-lite` together with some custom machine accelerators [59], including: `nvdimm` to provide the root filesystem as a persistent memory device to the VM; `nofw` to boot an ELF format kernel by skipping the

BIOS/firmware in the guest; and `static-prt` to reduce the interpretation burden for guest ACPI component. VM templating is another technique used by Kata Containers so that new VMs are "forked" from a pre-created template VM. The cloned VMs will share the same initramfs, kernel and agent memory in readonly mode. Because the lightweight VM is stripped down to the minimum VM that can run containers and uses an custom kernel, the techniques of Kata Containers cannot be applied on a general VM.

2.4 Summary

As mentioned in Section 2.1, the provisioning process of a VM/container includes transferring the image from the repository to the local compute node and the actual VM/Container boot/startup process. In fact, most studies have only focused on reducing the image transferring time. They made an assumption that the actual boot process duration is stable and not as significant as the transferring time. This assumption has recently been challenged by some studies [20, 60], which demonstrate that the actual booting up process varies considerably under different scenarios. The boot operation is a key factor in the resources provisioning process in a cloud system. If we allocate a VM/container on a high resources contention compute node it can take up to some minutes to complete the boot process. This situation is critical when the customers need to turn on a VM/container to handle a burst of incoming requests to their systems and potentially causes the economic loss.

There are two main approaches toward improving the boot duration of a VM: resuming and cloning. Resuming techniques allow a VM to be started quickly by using a suspended state of an entire running VM. It essentially skips the VM boot process altogether, thus, improves the boot time. However, the resumed VM is required to have different configurations to align with the requirements of a VM. This will lead to the storage explosion because of many possible combinations of configurations and VMIs. Another approach for this problem is using the cloning technique, in which a new VM is identically cloned from a running VM. As a result, this newly cloned VM is exactly the same to the running VM and its configurations have to be modified to match the VM request's requirements. Moreover, a VM has to be kept running in order to clone new VM from it. Even though these 2 solutions somehow skip the init process when booting VMs, it still generates I/O for copying the files. To the best of our knowledge, there is no study on improving the boot duration of a container in the literature. Prior work only proposed solutions for increase the performance of the container image retrieval process.

There are many virtualization solutions for VMs and Containers. QEMU-KVM and Docker are the most popular among them, they have a wide-scale adoption in both the industry and academia. The background related to the boot process of these two specific solutions is essential before we design experiments and explain the boot behavior of both VMs and containers. In the next chapter, we introduce the technical details of the QEMU-KVM and Docker, which is used to perform all the experiments in this thesis.

3

Technical background on QEMU-KVM VM and Docker Container

Contents

We provided a high-level overview of the provisioning process in the previous chapter. In this chapter, we dive into the technical details related to the boot operation of QEMU-KVM and Docker, the two widespread use virtualization solutions. Understanding the architectural and the workflow of these two techniques is mandatory to understanding the overhead and behaviors of boot duration results.

3.1 QEMU-KVM Virtual Machine

3.1.1 QEMU-KVM Work Flow

It is also worth mentioning a little history, which can make clear to the confusion around QEMU and KVM. QEMU is a type 2 hypervisor for performing full virtualization. It is flexible in that it can emulate CPUs via dynamic binary translation allowing code written for a given processor to be executed on another (i.e ARM on x86, or PPC on ARM). Given that QEMU is a software-based emulator which can run independently, it interprets and executes CPU instructions one at a time in software, which means its performance is limited.

Previously, KVM (Kernel-base Virtual Machine) was a fork of QEMU, named qemu-kvm. The main idea of KVM development is leveraging hardware-assisted virtualization to greatly improve the QEMU performance. KVM cannot by itself create a VM, to do so, it must use QEMU [61]. The KVM was included in mainline QEMU version 1.3 and the kernel component of KVM is merged in the mainline Linux 2.6.20.

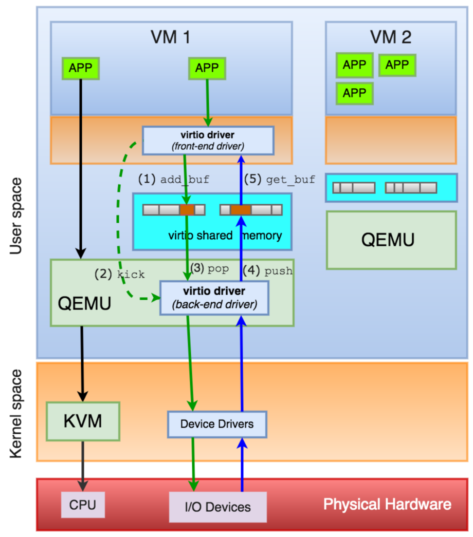


Figure 3.1 – QEMU/KVM with virtio work flow

In our work, we focus on full virtualization using QEMU-KVM, the default Linux hypervisor, and *virtio* [27] as paravirtualization driver for I/O elements. The QEMU-KVM architecture is presented in details in Figure 3.1. From a host point of view, each VM is a QEMU process, each application inside a VM is a thread that belong to a QEMU process. When an application on a guest OS requires an instruction, QEMU conveys this request to KVM. KVM will identifies the instruction, if it is an execution of a sensitive instruction by the CPU, it will be transferred without modification to the CPU for direct execution [61]. If it is an I/O instruction, in case we use QEMU as emulated virtualization drives, KVM will give the control back to the QEMU process, and QEMU executes the task. However, with paravirtualization driver (in Figure 3.1), a I/O instruction is handled by the virtio kernel module on the guest OS (not go through the KVM and get back to QEMU). Virtio creates a shared memory that can be access from both guest OS and QEMU. Using this shared memory, I/O processing for multiple items of data can be perform together, thereby reducing the overhead associated with QEMU emulation [62].

3.1.2 VM Disk Types

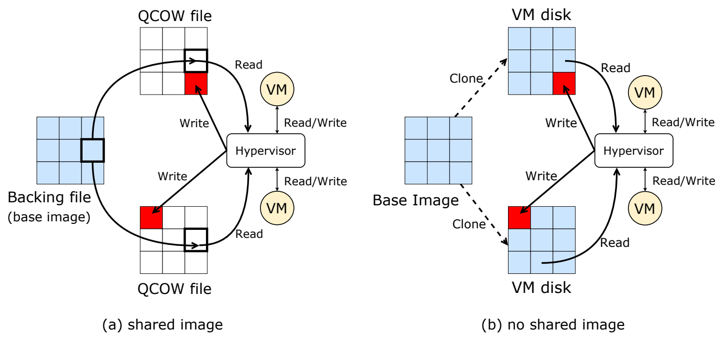
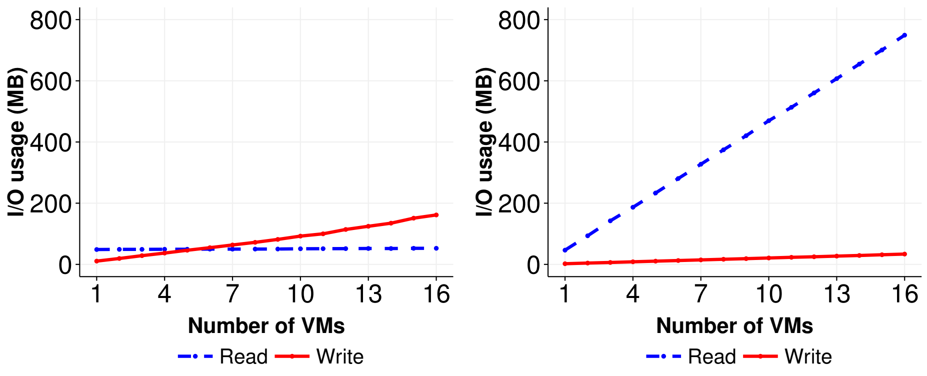


Figure 3.2 – Two types of VM disk

QEMU offers two strategies to create a VM disk image from the VMI (*a.k.a.* the VM base image). Figure 3.2 illustrates these two strategies. For the sake of simplicity, we call them *shared image* and *no shared image* strategies. In the *shared image* strategy, the VM disk is built on top of two files: the backing and the QCOW (QEMU Copy-On-Write) files [63]. The backing file is the base image that can be shared between several VMs while the QCOW is related to a single VM and contains write operations that has been previously performed. When a VM performs read requests, the hypervisor first tries to retrieve the requested data from the QCOW and if not it forwards the access to the backing file. In the *no shared image* strategy, the VM disk image is cloned fully from the base image and all read/writes operations executed from the VM will be performed on this standalone disk.

3.1.3 Amount of Manipulated Data in a VM Boot Process

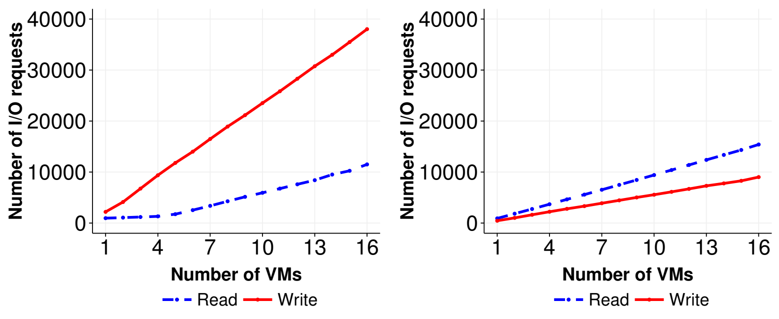
Because a VM boot process implies I/O operations, understanding the difference in terms of the amount of manipulated data between these two strategies is important. To identify the amount of data that is manipulated during VM boot operations in both VM disk strategies, we performed a first experiment that consisted of booting up to 16 VMs simultaneously on the same compute node. We used QEMU-KVM (QEMU-2.1.2) as the hypervisor, VMs are created from the 1.2 GB Debian image (Debian 7, Linux-3.2) with *writethrough* cache mode (at the opposite of the *writeback*, and each write operation is directly propagated to the VM disk image [64]).



(a) *shared image* disk (b) *no shared image* disk

Figure 3.3 – The amount of manipulated data during boot operations (reads/writes)

Figure 3.3 reveals the amount of read/write data when booting up to 16 VMs at the same time. Although the VMs have been created from a 1.2 GB VMI, booting 1 VM only needs to read around 50 MB from kernel files in both cases of *shared image* and *no shared image*. In addition to confirming previous studies regarding the small amount of mandatory data *w.r.t.* the size of the VMI, this experiment shows that booting several instances of the same VM simultaneously leads to a different amount of manipulated data according to the disk strategy used to create the VM disk(s). When the VMs share the same backing file (Figure 3.3a), the different boot process benefits from the cache and the total amount of read data stays approximately around 50 MB no matter how many VMs are started (the mandatory data has to be loaded only once and stays into the cache for later accesses). When VMs rely on different VM disks (Figure 3.3b), the amount of read data grows linearly since each VM has to load 50 MB data for its own boot process. Regarding write accesses, both curves follow the same increasing trend. However, the amount of manipulated data differs: the *shared image* strategy writes 10 MB data when booting one VM and 160 MB for booting 16 VMs while the *no shared image* strategy slightly rises from 2 MB to 32 MB. The reason why the *shared image* strategy writes 5 times more data is due to the *"copy-on-write"* mechanism: when a VM writes less than cluster size of the QCOW file (generally 64 kB), the missing blocks should be read from the backing file, modified with the new data and written into that QCOW file [65]. In addition to reading from the base image, the QEMU-KVM process (*i.e.*, the daemon in charge of handling the boot request) has to load into the memory a total of 23 MB. This amount of data correspond to host libraries and the QEMU binary file. The write operations performed by the QEMU-KVM process are negligible (a few KBytes).



(a) *shared image* disk (b) *no shared image* disk

Figure 3.4 – The number of I/O requests during boot operations (reads/writes)

To summarise, whatever the disk strategy, this experiment shows us that the number of I/O operations that are performed during boot operations are significant (as depicted in Figure 3.4) and should be mitigated as much as possible in order to prevent possible interference with other co-located workloads/vms. Loading mandatory data into the memory before starting the boot process may be an interesting approach to serve read requests faster. We investigate such a strategy can be achieved in the next chapter.

3.2 Docker Container

3.2.1 Docker Container Work Flow

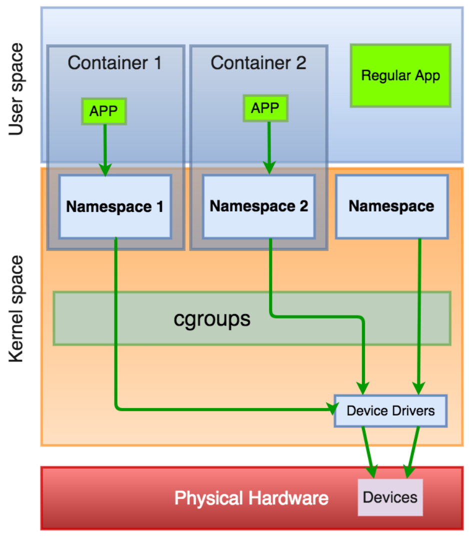


Figure 3.5 – Container work flow

When a container is started, its Docker image is downloaded from the image registry (Docker Hub) if the image is not stored locally. Docker creates a `namespace` for this container and uses `cgroups` to set a limit to the host's resources based on the configuration of the container (Figure 3.5). A thin writable container layer is created with a merged mount point to the Docker image. Docker setups an isolated environment so that the process of the container can be executed with the pre-defined configuration. This process can access to the host resources like a normal process with only some restrictions. Therefore, the I/O performance of a container is very close to that of the host system.

3.2.2 Docker Image

Docker image is the template for creating a container. It is comprised of several layers stacked on top of each other, with each layer represents an instruction in the image's `Dockerfile`. Each layer is the differences from the layer before it. As a result, a layer can be reused in different images, for example, the layer for Redis as well as the layer for Postgres can use the Ubuntu OS as the base layer. When a new container is created, Docker adds a new writable layer - called container layer - on top of the underlying read-only image layers. All changes made to the running container are applied to this container layer. When the container is deleted, only the container layer is removed, the image layers remain intact. By separating the container layer with the image layers, the image layers can be shared among different containers.

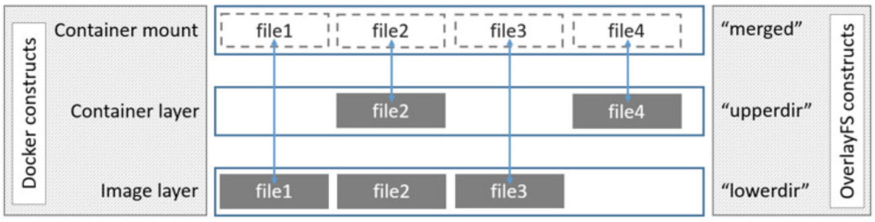


Figure 3.6 – Docker union file system: *overlayfs* [4]

From the storage viewpoint, a layer consists of a set of directories and files that makes up the root file system for a container. Therefore, the image layer is the *lowerdir* files and the container layer is the *upperdir* files (Figure 3.6). Correspondingly, these two layers can be seen as the backing and COW files in the VM terminology.

The unified view of the two directories is exposed as the *merged* union mount that is mounted into the container thanks to the *overlayfs* file system. This file system implements the copy-on-write strategy. When the image layer and the docker container layer contain the same files, the data of those files are read from the container layer instead of the image layer. When an existing file in the docker container is modified, the file is copied to the container layer first and the modification is applied to this version of this file in the container layer.

3.2.3 Amount of Manipulated Data of a Docker Boot Process

Although the order of magnitude differs, the amount of manipulated data when booting several times the same container follows the same trend of VMs sharing the same backing file: thanks to the cache, the amount of read data is constant. However, at the opposite of VMs, we observed that the significant part of read accesses when booting one container is related to the host directories and not the docker image. In other words, loading the docker binaries (*docker*, *docker-containerd-shim* and *docker-runc*), their associated libraries and configuration files represent much more data than the one that is performed on the docker image. Table 3.1 gives the details for different kinds of containers. Regarding the write operations, they are related to the creation of the container layer and the union mount. Although this amount is not significant *w.r.t* read operations, we noticed that the creation of the merge union mount point is a synchronous process: the docker daemon has to wait the completion of this action before progressing in the boot process. This is an important point as the more competition we will have on the I/O path, the longer will be the time to start the container.

Table 3.1 – The amount of read data during a docker boot process

	Host OS	Docker image
debian	62.9 MB	3.7 MB
ubuntu	62.6 MB	4.1 MB
redis	61.8 MB	8.2 MB
postgres	60.1 MB	24.4 MB

3.3 Summary

We presented in this chapter the technical background for QEMU-KVM and Docker with the focus on the running work flow, the VM/Container disks and the amount of manipulated data when booting. QEMU-KVM has two types of disks creation strategies: *shared image* and *no shared image*, while Docker creates the container disks as a separated layer on top of the Docker base image. Our experiment shows the amount of manipulated data on the disks of a VM/Container when booted. VMs have a different pattern of I/O usage when using different type of disks. There are significant number of I/O operations during the boot operation regardless of the disk strategies. A major different between the amount of read between VMs and Docker is that Docker performs many read accesses to the directories of the Docker installation, instead of the docker image. Equipped with this knowledge, we can perform experiments to uncover the boot behavior of VMs/containers in the next chapter.

III

Contribution: Understanding and Improving VM/Container Boot Duration

4

Understanding VM/Container Boot Time and Performance Penalties

Contents

In this chapter, we discuss a large experimental campaign that we performed to understand in more detail the boot behaviour of both virtualization techniques. Particularly, we analyzed thoroughly the boot time of VMs, containers on top of bare-metal servers, and containers inside VMs (or nested container) which is a current trend of public Cloud Computing such as Amazon Web Services or Google Cloud. We conducted more than 14.400 experiments in a software-defined way on top of Grid'5000 testbed for a bit more than 500 hours. This work discusses several experiment scenarios that aim to investigate different factors related to the boot action. As far as we know, our study is the first work that deals with many different resource contention scenarios for the boot operation. After that, we present our preliminary studies which are related to preloading all the mandatory data during a boot process to speed up the boot duration.

4.1 Experiments Setup

4.1.1 Infrastructure

We use the physical nodes of Nantes cluster in Grid'5000 [66]. Each physical node has 2 Intel Xeon E5-2660 CPUs (8 physical cores each) running at 2.2 GHz; 64 GB of memory, a 10 Gbit Ethernet network card and one of two kinds of storage devices: (i) HDD with 10 000 rpm Seagate Savvio 200 GB (150 MB/s throughput) and (ii) SSD with Toshiba PX02SS 186 GB (346 MB/s throughput). Regarding remote attached volume, we used Ceph version 10.2.5 deployed on 5 nodes (1 master and 4 data nodes, using HDD). When needed, Ceph has been used to deliver the remote-attached VM image disks to different VMs (each "compute" node mounted the remote block devices with *ext4* format).

4.1.2 VM and Container Configurations

For container, we used Docker [67] version 17.05.0-ce, build 89658be, with `overlay2` storage driver. Regarding the VMs' configuration, the hypervisor is QEMU-KVM (QEMU-2.1.2 and Linux-3.2) [68], `virtio` [27] is enabled for network and disk device drivers. VM disks are created from the VMI with QCOW2 format. The I/O scheduler of VMs and the compute node is CFQ. We choose QEMU-KVM and Docker for our evaluation because they are the most widely used virtualization solutions. From this point forward, the term "docker" is used interchangeably with "docker container". We set up all VMs and containers with 1 vCPU and 1 GB of memory.

In our experiments, we defined two type of machines:

— $e - machine$ is an experimenting machine (*i.e.*, VM, docker or nested docker) which is used to measure the boot time;

— $co - machine$ is a co-located machine, it is allocated on the same compute node as *e-machines* and runs competitive workloads;

We created the combinations of booting VMs by using the following parameters:

— cpu_policy: whether all VMs/dockers are started on the same physical core or each VM/docker is isolated on a dedicated core.

— $boot\_policy$: defines the way to boot e-VMs

  — (a) ***one then other***: the first e-machine is started. Once the boot operation is completed, the rest of e-machines are booted simultaneously. The goal is to evaluate the impact of the cache. The boot time is calculated as the time to boot the first e-machine plus the time to boot all remaining ones.

  — (b) ***all at once***: all e-machines are booted at the same time. The time we report is the maximum boot time among all VMs.

— $cache\_mode$: we use *writeback*, *writethrough* and *none* when configure a VM. A detail explanation for each cache mode is available at [64].

— $image\_policy$: is the way to create a VM disk. There are two strategies: *shared image* and *no shared image*. The explanation of these two image policies is in section 3.1.2. For container, there is one way to create its disk, so we do not use this parameter to configure a container.

We combined the above parameters to cover a maximum of boot scenarios because the boot time is not only impacted by resource contention on the same compute node but also the way VMs are booted and the way they are created. We underline that we did not consider the configuration of a VM. Indeed, Wu et al [60] showed that the capacity of a VM does not impact the duration of a VM boot process (*i.e.*, a VM with 1 core and 2G memory takes a similar time to boot as a VM with 16 cores and 32GB). We also did not consider the size of a VMI. Although the application files can significantly increase the size of the VMI, only the kernel data is loaded for the boot process. Several studies [15, 55, 69] confirmed that a small portion of a VMI is loaded during a VM boot process. All our experiments have been repeated at least 10 times to get statistically significant results.

4.1.3 Benchmark Tools

LINPACK [1] is used to produce CPU workloads. `LINPACK` estimates a system's floating point computing power by measuring how fast a computer solves a dense n by n system of linear equations $Ax = b$.

CacheBench [2] is a benchmark to evaluate the raw bandwidth in megabytes per second of the memory of computer systems. It includes read, write and modify operations on the memory to fully simulate the effect of memory usage.

1. http://people.sc.fsu.edu/ jburkardt/c_src/linpack_bench/linpack_bench.html
2. http://icl.cs.utk.edu/llcbench/cachebench.html

Stress [1] simulates an I/O stress by spawning a number of workers to continuously write to files and unlink them.

Iperf [2] measures the maximum achievable bandwidth on IP networks. Iperf creates TCP and UDP data streams to measure the throughput of a network that is carrying them.

4.1.4 Boot Time Definition

In our work, the boot time is calculated as the duration to perform only the boot process. We did not take into account the duration of VM/Container placement nor the VMI transferring process. The boot duration is measured as follows:

VM boot time: we assume that a VM is ready to be used when the guest OS is deployed completely and clients can log into the VM, therefore, we calculated the VM boot duration as the time to have the SSH service started. This time can be retrieved by reading the system log, and it is measured with milliseconds precision. In our setup, we configured SSH as the first service to be started.

Docker boot time: The main idea behind a container is running applications in isolation from each other [67]. For this reason, docker boot duration is considered as the time to have a service starts inside a docker. Specifically, by using *Docker CLI* [3], we measured the boot time as the duration for starting the SSH service inside a docker.

Nested Docker boot time: We measured this boot time in the same manner as the Docker boot time. We did not include the host VM boot time in this calculation.

4.1.5 A Software Defined Experiment

All our experiments have been performed on top of the Grid'5000 [66], a large scale and highly reconfigurable experimental grid testbed. We created a dedicated script to create and manage VMs automatically by leveraging the `Execo` framework [70] and `libvirt`. Precisely, our script extends `vm5k` [4] - a python module to perform reproducible experiments of VMs on the Grid'5000 platform. At coarse-grained, `vm5k` relieves researchers of the burden of deploying virtualization stack on bare-metal servers: (1) it deploys and configures servers with all necessary packages, (2) creates the VM templates according to the defined parameters, (3) starts the VMs, and (4) performs specific benchmarks and collects information. By extending `vm5k` and using advanced features of `Execo`, we completely scripted our experimental campaign in order to measure the boot time and monitor the resource usage during the boot process for both VMs and containers [19]. We underlined this script allows any researchers to reproduce all experiments anytime as it performs all scenarios in an isolated environment.

1. http://people.seas.harvard.edu/~apw/stress/
2. https://iperf.fr/
3. https://docs.docker.com/engine/reference/commandline/cli/
4. http://vm5k.readthedocs.io/

In details, our script leverages:

1. **Execo [70]:** Performing experiments consists of running different scenarios over and over again. `Execo` provides a template to run customized experiments with different predefined scenarios. `Execo` takes experiment parameters as inputs and produces combinations of these inputs. Each combination is a scenario for an experiment following a complex workflow that we designed. The engine of `Execo` then distributes these combinations to pre-booked physical nodes to conduct experiments.
2. **OARSUB [71]:** is a batch scheduler for large clusters, based upon an original design that emphasizes on low software complexity by using high level tools.
3. **Kadeploy3 [72]:** is a scalable, efficient and reliable deployment system (cluster provisioning solution) for cluster and grid computing. It provides a set of tools for cloning, configuring (post installation) and managing cluster nodes. We use `Kadeploy3` in `Execo` to deploy operating system images to physical hosts.
4. **Taktuk [73]:** is a tool for deploying efficiently parallel remote executions of commands to a potentially large set of remote nodes.

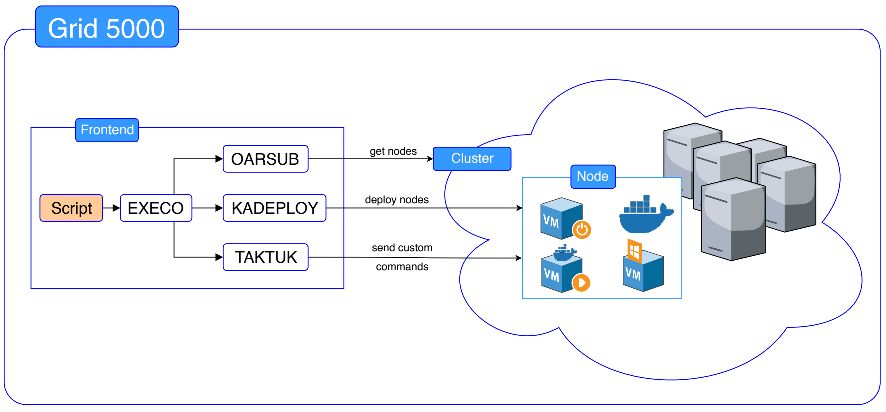


Figure 4.1 – Engine Architecture

Figure 4.1 shows the workflow of our script. First, our script reserves the compute nodes in Grid'5000's clusters by using `OARSUB` software suite. Then `Kadeploy3` is used to deploy the machine with a specific Linux environment and setup the network. Next, we install in those nodes the software and tools required for our experiments by utilizing `Taktuk` to send installation commands. For remote attached volume scenarios, we deploy a dedicated Ceph environment using `dfs5k`[1] - a tool for the deployment of Distributed File System on top of Grid'5000. When the compute nodes are ready, we loop through all the combinations of possible scenarios for our experiment, as explained in Section 4.1. For each combination, we use `Execo` with `Taktuk` to run

1. `https://www.grid5000.fr/mediawiki/index.php/Storage`

the scripted scenario and collect results from the compute nodes. All the results are gathered and saved on the machine where the script is executed.

4.2 Boot Time In No-Workload Environment

This scenario aims to evaluate how the boot time is affected when multiple machines are booted simultaneously on different storage devices. We boot several *e-machines* on a "fresh" compute node (*i.e.*, a compute node without the co-located workload and with an empty page cache). Each *e-machine* is assigned to a single core to avoid CPU contention, the number of *e-machines* has been increased from 1 to 16 because the compute node we used only has 16 physical cores. For VMs, we created them with two types of disk creation strategies, three cache modes and they are booted in two ways: *all at once* and *one then others*.

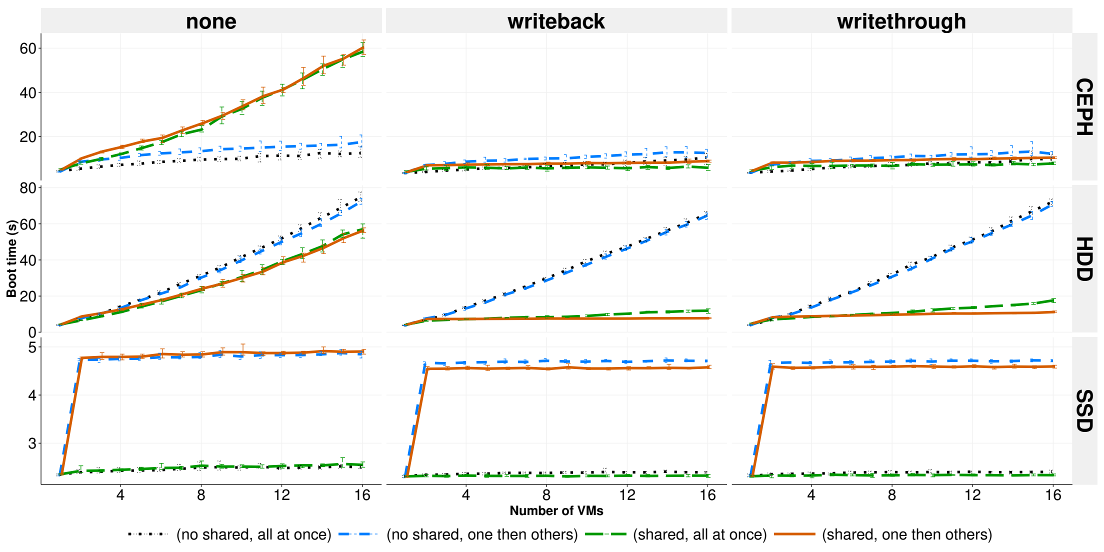


Figure 4.2 – Boot time of VMs with different installations on three storage devices

4.2.1 VM Boot Time

We expect the impact on VM boot time in this experiment is mostly the I/O throughput from loading the kernel files and writing the system log from VMs. Figure 4.2 reveals that even on a "fresh" node when we boot 16 VMs simultaneously, the boot duration can last more than one minute in the worse case. In comparison, it only takes 4.5 seconds to boot one VM.

On HDD, the boot time of VMs with *shared image* disk is always faster than the *no shared* one in three cases of cache mode. When the cache mode is on (*i.e.*, *writeback* and $writethrough$), the boot time of VM with *shared image* disk is much faster. The reason is when the very first VM boots, the mandatory data for VM boot process on the backing file is loaded on the memory, and then, because all VMs are sharing the same backing file, the following VMs can access this data directly from the memory and do not generate read access to the storage device. In the meanwhile, with *no shared image* strategy, because every VM has its standalone disk, the mandatory data has to be read many times even all these VMs are created from the same VMI. In case of *none* cache mode, VMs with *shared image* disk also have to read the boot data from the backing file many times. Moreover, there is the overhead of read access of *shared image* strategy which comes from checking if the requested data is on the QCOW or on the backing file as explained previously in section 3.1.2. Therefore, the boot time is less faster than the *no shared* one.

To compare $one\ then\ others$ and $all\ at\ onces$ boot policy, we should only consider scenarios where VMs get benefit from the cache, specifically, VMs are created by using *shared image* disk with $writeback$ or $writethrough$ cache mode. In this condition, the $one\ then\ others$ policy boots faster than $all\ at\ once$. In case of $one\ then\ others$, only the first VM generates read operations to the disk, the other VMs read the mandatory data from memory so that the I/O contention only comes from write operations (VMs write log files during the boot process). On the other hand, when all VMs are booted at the same time, the I/O contention comes from read and write access of all VMs. The difference in the amount of I/O requests between these two cases leads to the faster boot of $one\ then\ others$ boot policy.

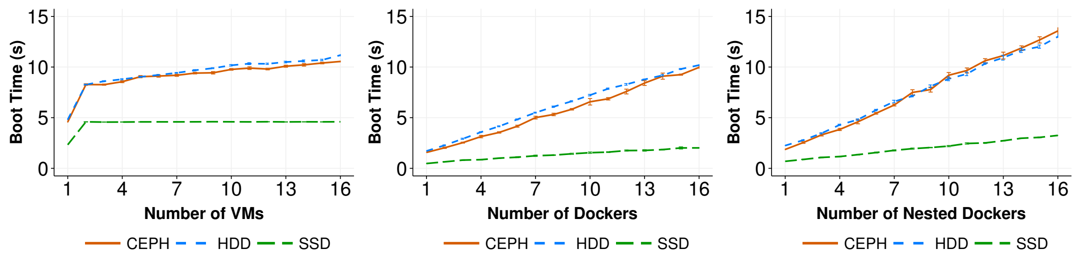
On SSD, the boot time of one VM is around 2.5 seconds, and it is mostly constant when we increase the number of VMs from 1 to 16, and the VM boot time is not impacted by different cache modes nor image policies. The boot duration does not increase because the I/O throughput of the SSD is five times bigger than HDD, all I/O requests generated by VMs are not enough to stress the I/O and they are handled very quickly. The boot time of $one\ then\ others$ is longer because we accumulate the boot time of the first VM.

On Ceph, with $writeback$ and $writethrough$ cache mode, we observe that there is not a big difference in boot time between two image policies as on HDD. The *shared image* disk VMs still gain benefits from the cache, however, the VMs with *no shared image* disk do not directly suffer the bad overhead of the random read as on HDD. With $none$ cache mode, VMs with *no shared image* disk boot much faster than VM with *shared image* disk. There are two reasons. First, with the good I/O performance of Ceph, random read access from different VM disks in case of *no shared image* on Ceph is responded faster. Second, when there is no cache, we have already explained that VMs with *share image* disk have to load mandatory data many times. Plus, the overhead of checking where the needed data is stored to read of the *shared image* mechanism becomes significant in case of the appearance of the latency.

In brief, without co-workloads, VMs with various installations have different boot duration, and even booting multiple VMs with the same installations have diverse behaviors on different storage devices. *Shared image* policy always has better performance on all storage devices if the cache mode of VMs is on. Although the I/O performance of $writeback$ is the best, the data is not protected in a power failure situation and it is recommended only for temporary data where data loss is not a concern. With $one\ then\ others$ boot policy, the boot time is good only on HDD where the random read is costly; on Ceph and SSD, with good I/O performance the *all at once* boot policy is slightly better.

4.2.2 Docker Boot Time

To compare the boot time between VMs and containers, we consider the boot time of VMs with $writethrough$ cache mode, $share\ image$ disk and $one\ then\ others$ boot policy. We illustrate the boot time of those VMs using 3 different storage devices in Figure 4.3a.



(a) VM (b) Docker (c) Nested Docker

Figure 4.3 – Boot time of multiple VMs, Dockers and Nested Dockers on 3 storage devices

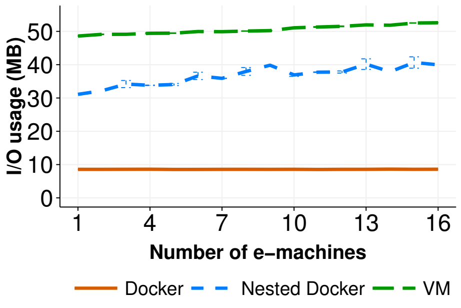
Figure 4.3b depicts that the boot time of docker increases linearly along with the increasing of the number of dockers on three storage devices. The result also shows that dockers boot faster than VMs, as expected, even in the case of the best configured of VMs (Figure 4.3a). On HDD, booting one docker in an idle compute node takes 1.7 seconds and 16 dockers need around 10.5 seconds. On Ceph, the boot duration is slightly better than on HDD because of the higher I/O performance of Ceph. It takes around 1.5 seconds to 10 seconds to boot 1 to 16 dockers simultaneously. On SSD, the boot time also has the upward trend with smaller slope compared to HDD and Ceph. This is the result of having the highest I/O throughput between the 3 types of disks.

4.2.3 Nested Docker Boot Time

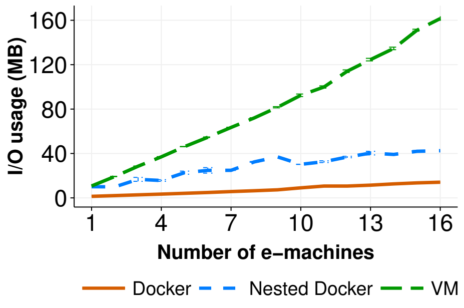
Boot duration of nested docker has the same trend as docker as illustrated in Figure 4.3c. However, the nested docker boot time is a bit longer compared to docker. The explanation for this observation is that the nested docker is located inside a VM placed on the compute node. Therefore, the nested docker also suffers from the reduced performance of the virtualization technology.

4.2.4 Discussion

In general, booting one docker or nested docker is faster than booting a VM in an idle environment. The main reason is the number of I/O operations during their boot processes. When a VM is booted, a normal Linux boot process is in place. VM loads the kernel data from the VM disk, performs some computations, and then writes the log files to VM disk. The full Linux boot process runs inside a VM on restricted pre-assigned resources. On the contrary, when we start a new container, Docker engine initializes the file system for a container and mount the read-write layer for it. Then Docker engine assigns the CPU and memory of the container using resources of the compute node. Finally, the requested service is executed inside that container with limited resources. In brief, containers do not need to load the kernel data or go through the full Linux boot process. Plus, Docker engine prepares the environment to run a service inside a container without any restriction on the physical resources.



(a) I/O read



(b) I/O write

Figure 4.4 – I/O usage during boot process of multiple *machines*

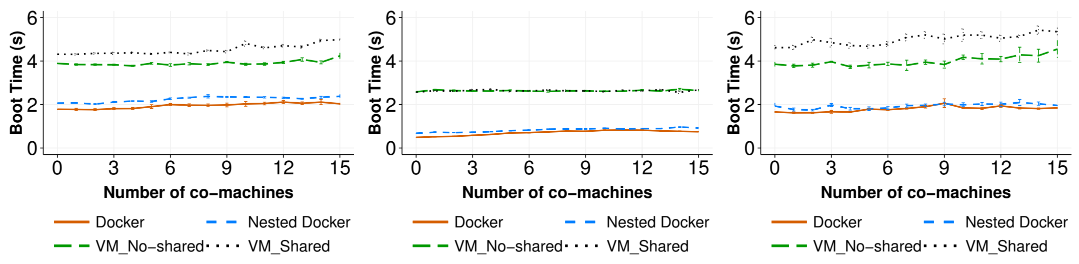
In Figure 4.4, we compared the amount of data that VMs, dockers and nested dockers read from their images. The amount of I/O read shown in Figure 4.4a is steady for VMs and containers when the number of machines increases because they all use the *shared image* disk strategy (*i.e.*, all machine share the same base image, and the mandatory data for their boot processes are read only once). However, data need for a VM boot process is much more than docker one. Figure 4.4b depicts that the amount of written data of VMs and containers increase linearly. VMs also have much higher I/O writes than dockers because of the *"copy-on-write"* mechanism of the VM disk. When a VM writes new data to a COW image, the relevant blocks are read from the backing file, modified with the new data and written into that COW image. This mechanism leads to 2 read and 3 write operations for every single write operation on the COW image [65]. The amount of read and write data of nested docker is higher than that of a docker because a nested docker is booted inside a VM with *shared image* disk so that it is suffered from the COW image mechanism.

4.3 Boot Time Under Workloads Contention

This experiment focuses on understanding the boot time of a single machine in the presence of concurrent workloads. On a compute node, first, we boot n *co-machines* ($n \in [0, 15]$) and then run workload on them. After that, we start one *e-machine*. In the case where we want to generate CPU stress, all *co-machines* are allocated on the same physical core with the *e-machine*. For the other experiments where CPU contention should be avoided (*i.e.*, when we want to stress either only the memory, network or the I/O bus), every machine has to be assigned to different cores. In this scenario, all VMs are configured with $writethrough$ cache mode. Because we only measure the boot duration of one VM, the parameters $cache\_mode$ and $boot\_policy$ do not have an important effect.

4.3.1 Memory Impact

As shown in Figure 4.5, when we increase the number of *co-machines* to stress the memory, the boot duration of the *e-machine* does not change on all three storage devices. The boot time of VM is consistently slower than nested docker and docker, while nested docker boots a little bit slower than docker. In conclusion, the effect of memory competition on boot time is negligible.

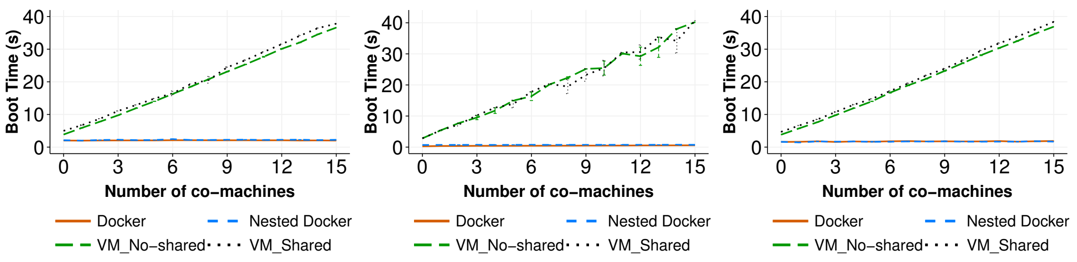


(a) HDD (b) SSD (c) Ceph

Figure 4.5 – Boot time of 1 VM, Docker and Nested Docker on three storage devices under memory contention

On HDD (Figure 4.5a), the boot time of the VM that has *shared image* disk is 4.5 seconds while *no shared image* disk is 4 seconds. The overhead of checking where is the needed data to read of *shared image* strategy make it 0.5 seconds longer than *no shared image* when booting only one VM. On Ceph (Figure 4.5c), we also see 1 second difference between two disk creating strategies. This gap becomes bigger on Ceph because of the latency when accessing data on a remote storage device. We also have differences between two image policies in case of CPU and network contention as we discussed in the next paragraph.

4.3.2 CPU Impact



(a) HDD (b) SSD (c) Ceph

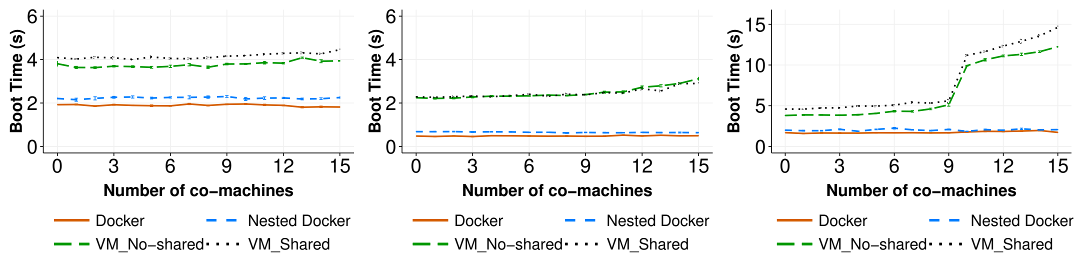
Figure 4.6 – Boot time of 1 VM, Docker and Nested Docker on three storage devices under CPU contention

Figure 4.6 reveals that VM boot time is impacted by CPU contention while docker and nested docker boot time are stable. For VMs, there is a clear linear correlation between the boot time of a VM and the number of co-allocated VMs, and the same increasing trend for three types of storage devices. Because all e-VM and co-VMs are assigned to one core, and the workload on each co-VM is always 100% full capacity of CPU usage, the more co-VMs running on one physical core, the more the VM has to wait until it can access the CPU core to perform the computation for the boot process. This leads to a longer boot time when we increase the number of co-VMs. In general, it takes around 4.5 seconds to boot a VM without CPU contention and increases to 40 seconds when we have 16 co-VMs generating CPU contention. We still see the 0.5 seconds gap between boot time of two disk creation strategies and this is also explained in the same way in case of memory contention.

In case of containers, under CPU stress, the boot time of both docker and nested docker are not affected on any storage devices. In this experiment, to generate CPU contention, we assign all *co-machines* into the same physical core where the *e-machine* is allocated. As explained before, during a docker boot process, when docker engine prepares the environment for a docker (i.e., mounting root file system, limiting resource), the engine may not run in the core that has been stressing. Only the starting process of the containers actually run under the constrained resources and affected by the CPU contention.

4.3.3 Network Impact

As we expected, Figure 4.7 reveals the network contention does not have any effect on the boot time of e-machine on local storage devices (HDD and SSD) and only affects

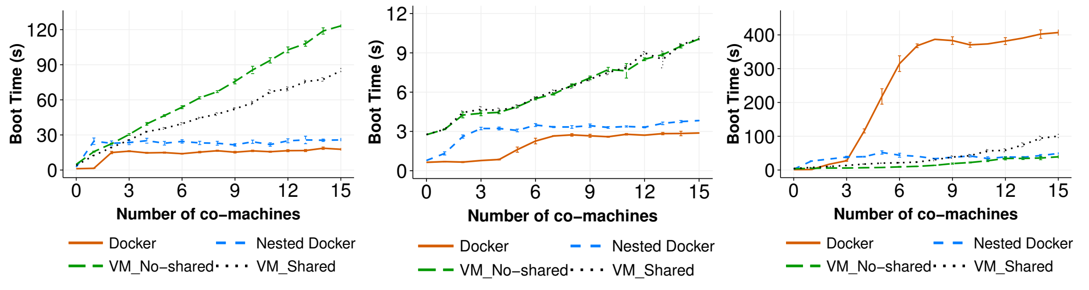


(a) HDD (b) SSD (c) Ceph

Figure 4.7 – Boot time of 1 VM, Docker and Nested Docker on three storage devices under network contention

the remote storage backend. It is reasonable since, on the local storage devices, the boot process of a VM or a docker does not require network exchanges. On Ceph, in the case of VM (Figure 4.8c), the e-VM uses bandwidth to load the kernel files remotely. The boot time of a VM with Ceph is quite stable around 4.5 seconds until we have 10 stressing the 10 Gigabit Ethernet interface of the compute node. When we keep increasing the network usage by growing the number of co-VMs, the remaining network bandwidth for the e-VM is limited, therefore, the boot time rises from 4.5 seconds to around 15 seconds. However, even on remote storage device - Ceph, docker and nested docker boot time are not significantly impacted under network stress.

4.3.4 I/O Disk Impact



(a) HDD (b) SSD (c) Ceph

Figure 4.8 – Boot time of 1 VM, Docker and Nested Docker on three storage devices under I/O contention

Figure 4.8 depicts the boot time of a machine on three storage devices under I/O stress. The VM boot time increases linearly with the increased number of co-VMs for both disk creation strategies. Under CPU, memory and network contention, the boot time of e-VMs with *no shared image* is faster than the *shared image* one and this gap is stable. However, the gap evolves differently under I/O contention. On HDD (Figure 4.8a), boot time of *no shared* image VM increases longer than the *shared image* one (from 4 seconds to over 120 seconds, compared to 4.5 seconds to 85 seconds, respectively), but the boot time between these two disk creating strategies is similar on SSD (Figure 4.8b). The reason is the high I/O performance of SSD. On Ceph, the overhead of checking data to read from *shared image* is again significant, this leads to the boot time of VM with *shared image* disk is longer. The gap between two disk creation strategies evolves differently on HDD can be explained as follows. With *no shared image* disk, a disk is populated with its full size so in case we have many coVMs, the space that is used on the host HDD is larger than the one used for the *shared image* approach. Given the mechanism of the HDD with the seek operation, the probability to have longer seeks is proportional to the space used on the HDD (the HDD's arm needs to seek back and forth often and through a larger distance in order to server all requests coming from all the hosted VMs). We verified such an explanation by conducting additional experiments with *blktrace* tool [74] to extract the I/O access data on the compute node while coVMs run I/O stress.

Docker and nested docker boot time also increase under I/O contention but they are faster than VMs on both HDD and SSD storage devices. The boot time of docker and nested docker increases when there is more stress on I/O, then becomes stable. In case of Ceph, the boot time of a docker is abnormally much longer than that of one VM. Docker boot time increases significantly in a bizarre manner when we stress I/O on Ceph which also leads to the contention in network resources. We do not observe this behavior with nested docker and VMs which shows the isolation limitation of Docker. Therefore, our hypothesis is that because Docker is not fully isolated from the host OS so it may have competitions in kernel system calls under high I/O contention on remote attached volumes. However, to fully explain this, a number of intensive experiments has to be conducted.

4.3.5 Discussion

Workloads that are already running on a compute node obviously increase the time to boot an additional VM/docker and this should be considered. While comparing the workloads together, our results show that VM, docker and nested docker boot time introduce negligible overhead for memory and network bandwidth (except in the case of remote attached volume such as Ceph). I/O disk is the most influence factor for both VM and docker on different types of storage devices. Even though the impact of CPU is rather small as compared to the I/O factor for VM (docker is not impacted by CPU contention), we cannot simply ignore it. Indeed, it is still significant enough to alter the boot time. Finally, as we expected, dockers have better boot time compared to nested dockers.

4.4 Preliminary Studies

In this section, we give additional elements regarding how we can reduce the impact of the I/O accesses during the boot operation. These preliminary studies led us to the *YOLO* proposal that I define in this thesis.

4.4.1 Prefetching *initrd* and *kernel* files

In the kernel stage of a normal Linux boot process, *initrd* [75] is used as a small file system located on RAM disk to run user space programs before the actual root file system is mounted. Because *Libvirt* [76] offers the possibility to boot a VM from specific kernel and *initrd* files, a simple way of speeding up the VM boot operation could be to load these files into the page cache beforehand. Such a strategy looks interesting because most VMIs differ only in the set of installed software. That is, we can use the same kernel and initrd files to serve many VMs that have different VMIs but the same kernel. However, diving into details, we observed that a large part of the I/O operations come after the initrd phase. That is after the kernel has mounted the real file system into the VM disk and has called the */etc/init* and other scripts on this real file system to start the services .

To summarise, the initrd and kernel files only represent a small part of the I/O accesses and another approach is needed.

4.4.2 Prefetching Mandatory Data

Leveraging the *shared-image* disk experiment, we observed that it is possible to mitigate the number of I/O operations by using the cache so that read operations are served from memory rather than from the storage device as long as the page cache is not evicted. To identify which part of a VMI is needed during a VM boot process, we booted one VM on a dedicated compute node with an empty page cache. To determine which pages of the VMI were resident into the cache after the boot operation, we used the Linux *mincore* function [77]. From that information, we extracted the list of logical block addresses of the VMI that a VM accesses during a boot process.

In addition to the accesses list, we collected additional information thanks to the Linux `blktrace`. This tool allowed us to capture this exact read access pattern according to the time as depicted by Figure 4.9.

These results are important. First they confirm that there is a large amount of read accesses and second that there is an alternation between I/O and CPU intensive phases. Leveraging these results, we investigated the most efficient approach to prefetch the mandatory data into the memory. There are two possibilities either by time or offset order. The time order corresponds to the same order a VM reads data during its boot process. This strategy is not optimal because of the small size of the I/O operations and the large number of random accesses. With the offset order, we sort and merge

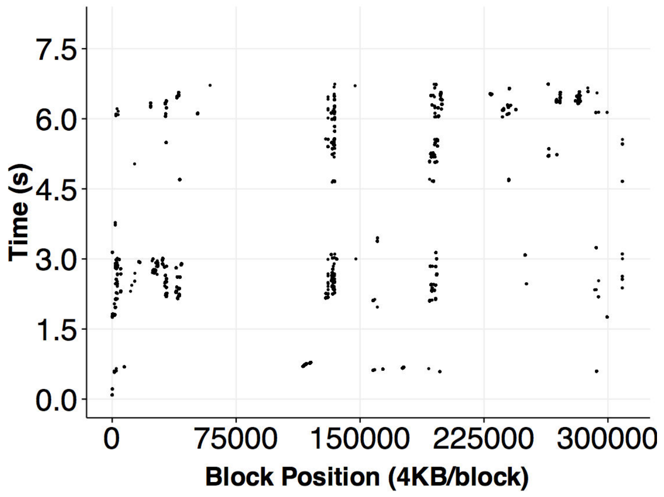
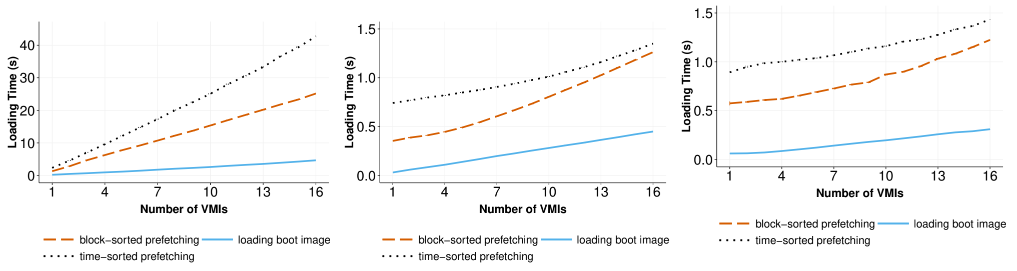


Figure 4.9 – Read accesses during a VM boot process. Each dot corresponds to an access at a certain period of the boot process and a certain offset.

the accesses by the logical block addresses so that we can have a sequential reading of the VM image. This strategy is more efficient. However, the number of accesses is still significant. The best solution would be to extract the mandatory data from each VMI and store it into a single file in the time order. Thanks to this file entitled *boot image*, it would be possible to read the file in a contiguous manner, benefit from the Linux kernel prefetching strategy [78] and thus put mandatory data into the memory in the most efficient manner.



(a) Local volume (HDD) (b) Local volume (SSD) (c) Remote attached volume (Ceph)

Figure 4.10 – Time comparison for prefetching the VM boot mandatory data

To effectively measure the benefit of the different strategies, we developed an ad-hoc script, which uses the *vmtouch* [79] command, to fetch the content of all the mandatory blocks according to the expected order. Figure 4.10 shows the comparison between the three policies on different storage devices emulating respectively locally stored (HDD and SSD) and remote-attached (Ceph [80]) VMIs. We underline that we did not measure the boot time duration but only the time to prefetch the mandatory data while increasing the number of manipulated VMIs (the more VMIs we have to access the more I/O contention we should expect). Hardware and configuration details are discussed in Section 4.1. Results confirm that retrieving the data through the two first prefetching strategies leads to worse performance in comparison the *boot image* approach.

To conclude, it would be interesting to create for each VMI its associated boot image and link it to the VM image disk structure in a similar manner of the share image disk strategy. By this way, the boot process should be modified at the hypervisor level in order to leverage the boot image during the boot process instead of using the VM image disk. However, in addition to requiring modifications at the hypervisor level and the VMI format, this solution has an important shortcoming related to the page cache space that can be claimed by the host OS whenever the memory is needed. In other words, while we expect the mandatory data would be available in the cache, VMs can face corner cases where they have to prefetch the cache once again. Consequently, it is not a practical solution especially in an I/O-intensive environment where the page cache of the host OS would be used intensively. Another approach, less dependent from the kernel and the hypervisor should be designed.

4.5 Summary

Performing reproducible experiments on cloud computing platforms implies complex tasks which required to evaluate a large number of configurations. The analysis of VMs and containers boot times is a such kind of this task. We presented in this chapter how we conducted, in a software-defined manner, more than 14.400 experiments and discussed the collected results.

This study gave us a comprehension view about the VM and container boot time. The VM boot duration is not only affected by the co-workloads and the number of VMs simultaneously deployment but also the parameters that we use to configure VMs. Furthermore, on different storage devices, booting VMs with the same configuration even have varied behaviors. Having the understanding of the VM boot time will allow us to choose the best way to boot new VMs within a cloud infrastructure. When comparing the boot time between containers and VMs, it is obvious that containers boot much faster than VMs. However, the boot duration of containers also has an increasing trend like the VM boot time when we boot multiple VMs/containers at the same time or under a workload contention environment. Especially, I/O throughput is the most significant factor on both VMs and containers boot duration. This behavior of VMs/containers boot time should be taken into account when we want to deploy new VMs/containers. Using the above analysis, we proposed a VM Boot Time model that gives a more accurate estimation of the time a VM needs to boot in a shared resource environment. The model is described in the Appendix Section A.

Besides that, the preliminary studies about the amount of mandatory data for a boot process in this chapter gave some insightful observations. Combining with the understanding of the effect of I/O operations in a boot process, we introduced the new mechanism in the following chapter to reduce the I/O requests by manipulating the mandatory boot data.

5 YOLO: Speed Up VMs and Containers Boot Time

Contents

Based on the detailed analysis for VM and container boot time behavior and some preliminary studies in the previous chapter, we present in this chapter our proposal YOLO *mechanism (*You Only Load Once*). YOLO reduces the number of I/O operations generated during a boot process by relying on a boot image abstraction, a subset of* *the VM/container image that contains all data blocks necessary to complete the boot operation. First, we describe the* YOLO *implementation and workflow and then, we evaluate* YOLO *under several scenarios with different types of storage devices as well as the overhead of* YOLO.

5.1 YOLO Design and Implementation

To leverage the boot image abstraction as well as limiting the cache effect, we designed *YOLO* as a new method to serve the mandatory boot data for a VM effectively. In this section, we give an overview of our proposal and its implementation. First, we explain how boot images are created. Second, we introduce how *yolofs*, our custom file system, intercepts I/O requests to speed up the VM boot process.

5.1.1 Boot Image

In this section we present how we implement the boot image abstraction and we give a few details regarding the storage requirements by analysing the Google Cloud platform as an example with a relevant number of VM images.

Creating a Boot Image

To create boot images, we capture all read requests generated when we boot completely a VM or a container. Each read request has: (i) a *file_descriptor* with *file_path* and *file_name*, (ii) an *offset* which is the beginning logical address to read from, and (iii) a *length* that is the total length of the data to read. For each read request, we calculate the list of all *block_id* to be read by using the *offset* and *length* information and we record the *block_id* along with the data of that block. A boot image contains a dictionary of key-value pairs in which the key is the pair (*file_name*, *block_id*) and the value is the content of that block. Therefore, with every read request on the base image, we can use the pair (*file_name*, *block_id*) to retrieve the data of that block. In a cloud system, we create these boot images for all available VM and container images and store them on each compute node. To avoid generating I/O contentions with other operations when accessing these boot images, we advocate to store them on dedicated devices for *yolofs*, which can be either local storage devices, remote attached volumes, or even memory.

Storage Requirement

The space needed to store 900+ VMIs available from Google Cloud is $1.34\,\text{TB}$. Then, for each VMI, we built a boot image using the method described in Section 5.1.1. In Table 5.1, we can see how the size reduction rate goes from 94% to 99%. Instead of storing all these VM images ($1.34\,\text{TB}$) locally on the physical machines to speed up the

Table 5.1 – The statistics of 900+ Google Cloud VMIs and their boot images. We group the VMIs into image families and calculate the boot images for each image family.

Image	No. of images	Size of images	Size of all boot images	Reducing rate
CentOS	156	223 GB	6.3 GB	97.2 %
Debian	180	216 GB	4.7 GB	97.8 %
Ubuntu	236	272 GB	16 GB	94.1 %
CoreOS	221	173 GB	1.2 GB	99.3 %
RHEL	167	302 GB	7 GB	97.7 %
Windows	15	191 GB	5.2 GB	97.3 %
Total	983	1.34 TB	40.4 GB	97.4 %

VM boot process, we only need to create and store 40 GB of boot images, which is less than 3% of the original size of all VMIs.

5.1.2 *yolofs*

Read/Write Data Flow

We developed *yolofs* using FUSE (**F**ilesystem in **Use**r space) to serve all the read requests executed by VMs during the boot process via the *boot images*. FUSE allows to create a custom file system in userspace without changing the kernel of the host OS. Furthermore, recent analysis [81, 82] confirmed that the performance overhead when using FUSE against read requests is acceptable. However, other solutions might be used if the performance is a critical issue, for example using library interposition.

In Figure 5.1, we illustrate the workflow of *yolofs* along with the read/write data flow for a QEMU-KVM VM which is created with a *shared image* disk and a Docker container. *yolofs* is executed as a daemon on each compute node (that is before any boot operation). When a VM/container issues read operations on its base image, which is linked to our mounted *yolofs* file system, the VFS routes the operation to the FUSE's kernel module, and *yolofs* will process it (*i.e.*, Step 1, 2, 3 of the read flow). *yolofs* then returns the data directly from the boot image which already was in the *yolofs*' memory (Step 4). If the boot image is not already on the memory, *yolofs* will load it from its dedicated storage device (where stores all the boot images of this cloud system) to the memory. Whenever the VM/docker wants to access data that is not available in the boot image, *yolofs* redirects the request to the kernel-based file system to read the data from the disk (Step 5, 6, and 7 of the read flow). Regarding write operations, they are not handled by *yolofs* and are forwarded normally to the corresponding COW file (the write flow in Figure 5.1).

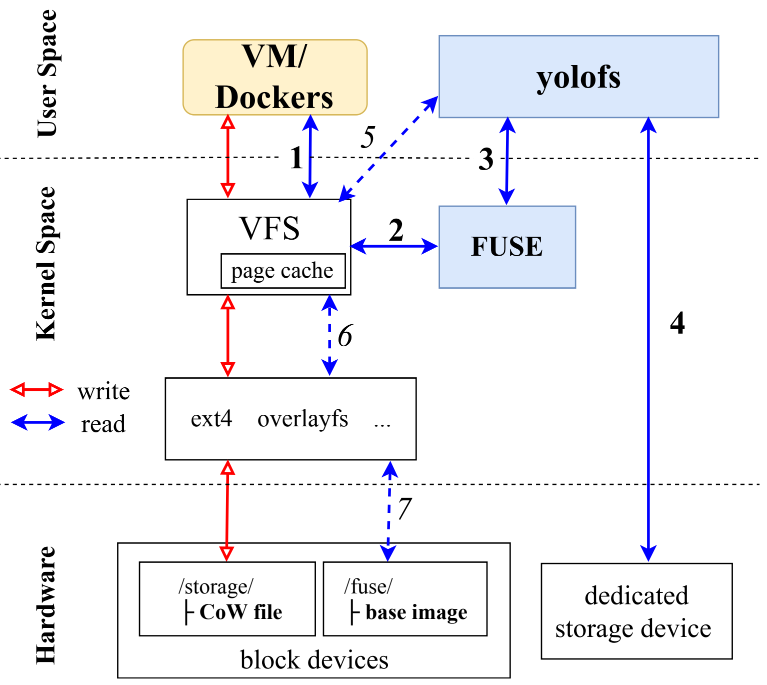


Figure 5.1 – yolofs read/write data flow

Implementation

We implemented *yolofs* to handle the VMs' I/O requests as shown in Algorithm 1. *yolofs* runs as a daemon waiting to handle I/O requests sent to the FUSE mount point. As mentioned, write requests do not go through *yolofs*, the kernel-based file system is used to write this data to the hardware disk (Line 1 of *Algorithm 1*). Otherwise, with every read request, we first extract the *file_descriptor*, *offset*, and *length* of the request (line 5). We calculate the begin *block_id* and the end *block_id* given the system *BLOCK_SIZE* (Line 6 and 7). *yolofs* takes the corresponding boot image B from the local repository and loads it into the memory if needed (Line 9). Next, we iterate over all *block_id* belong to the range [*block_begin, block_end*], with each *block_id* we check the corresponding boot image for the data of that block and return it (Line 12, 13, and 14). After the VM is booted, if the VM needs to read a block which is not in the boot image, that block is read from the kernel-based file system as described in line 17.

YOLO works using any storage backend and is transparent to the VMs, the hypervisor and the kernel of the host/guest OS as well.

5.2 Experimenting Protocol

In this section we discuss the experiments performed on top of Grid'5000 [66]. The code of *YOLO* as well as the set of scripts we used to conduct the experiments

Algorithm 1: VM Boot time speedup with yolofs

```
    input  : boot image $B$, I/O request $R$
    output : data $D$
 1  if $R$ is a write request then
 2      forward to a kernel-based file system to handle $R$
 3  end
 4  else
 5      $offset, length, file\_descriptor \leftarrow R$
 6      $block\_begin \leftarrow \frac{offset}{BLOCK\_SIZE}$
 7      $block\_end \leftarrow \frac{offset+length}{BLOCK\_SIZE}$
 8      $D \leftarrow empty\ list$
 9      if boot image $B$ not loaded then
10          load boot image $B$ from dedicated storage device into yolofs' memory
11      end
12      for $block\_id \leftarrow block\_begin$ to $block\_end$ do
13          if $block\_id$ in boot image $B$ then
14              $D \leftarrow D + B.get(block\_id, file\_descriptor)$
15          end
16          else
17              $D \leftarrow D + ((block\_id, file\_descriptor)$ from a kernel-based file
                system)
18          end
19      end
20      return $D$
21  end
```

are available on public git repositories [1]. We underline that all experiments have been made in a software defined manner so that it is possible to reproduce them on other testbeds (with slight adaptations in order to remove the dependency to Grid'5000). We have three sets of experiments. The first set is aimed to evaluate how *YOLO* behaves compared to the traditional boot process when the base images are either locally stored (HDD and SSD) or remotely attached through a Ceph system [80]. The second set investigates the impact of collocated memory and I/O intensive workloads on the boot process. Finally, we measured the overheads of using the *yolofs* during the execution of the VM with the third set of experiments.

1. `https://github.com/ntlinh16/vm5k`

5.2.1 Experimental Conditions

All experiment setups (including the infrastructure, VM and container configurations, benchmark tools, boot time definition) are similar to the one introduced in Section 4.1 in the previous chapter.

5.2.2 Boot Time Policies

For the first set of experiments, we investigated the time to boot up to 16 VMs using either the same or the different VMI in parallel. Our goal was to observe multiple VM deployment scenarios from the boot operation viewpoint.

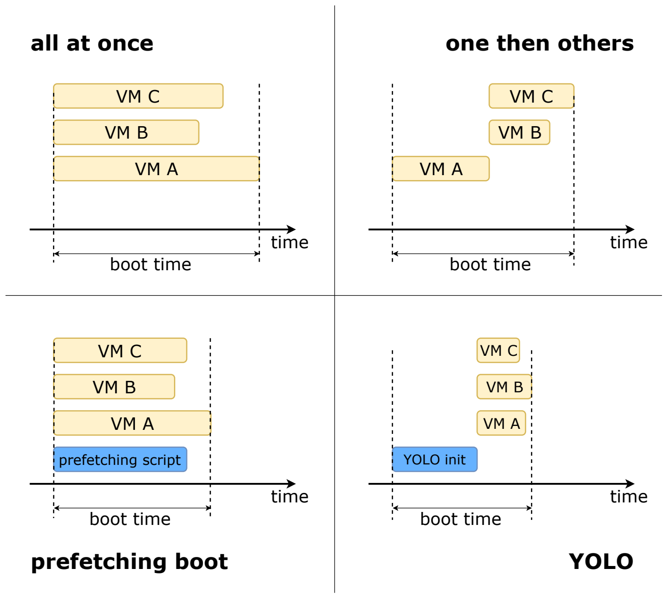


Figure 5.2 – Four investigated boot policies. Each block represents the time it takes to finish. *Prefetching boot* performs prefetching in a parallel fashion to leverage gaps during the booting process of a VMs for faster loading. *YOLO* loads and serves boot images whenever VMs need to access the mandatory data.

We considered four boot policies as depicted in Figure 5.2:

— *all at once*: all VMs are booted at the same time (the time we report is the the maximum boot time among all VMs)

— *one then others*: the first VM is started. Once the boot operation is completed, the rest of VMs are booted simultaneously. The goal is to evaluate the impact of the cache we observed during the preliminary study on the boot time (see Section 4.4). The boot time is calculated as the time to boot the first VM plus the time to boot all remaining ones.

— *Prefetching boot*: We used the *prefetching script* we developed for the preliminary studies (see Section 4.4) to fetch the mandatory data from the VMI in the offset order. As depicted the prefetching script and the boot process of VMs are invoked simultaneously. Figure 4.9 illustrates that there are several time gaps in reading data during the boot process, especially, at the beginning of the boot process and around the fourth second. These non I/O intensive periods, in particular the first one, enables us to start the prefetching script and the boot process of a new VM at the same time. If they were not, the duration needed for the prefetching operation would be almost similar than booting a VM, making this strategy similar to the previous one.

— *YOLO*: All VMs have been started at the same time, and when VM need to access mandatory data, *YOLO* will serve them. We underline that boot images have been preloaded into the *YOLO* memory before starting VMs. This way enabled us to emulate a non volatile device. While we agree that there might be a short overhead to copy from the non volatile device to the *YOLO* memory, we believe that doing so is acceptable as (i) the amount of manipulated boot images in our experiments is less than $800\,\mathrm{MB}$ ($16{*}50\,\mathrm{MB}$) and (ii) the overhead to load simultaneously 16 boot images from a dedicated SSD is negligible as discussed in the preliminary studies and confirmed in Figure 5.3.

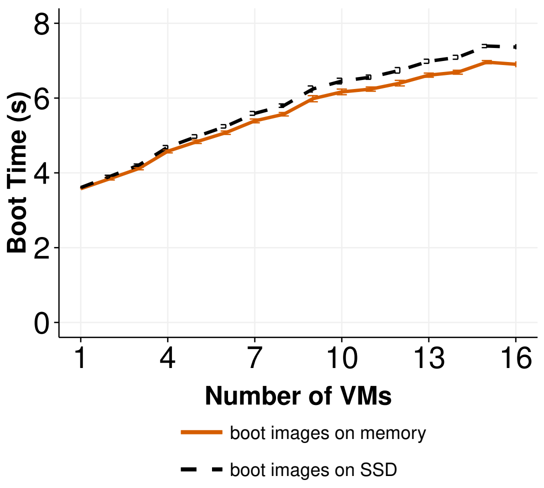


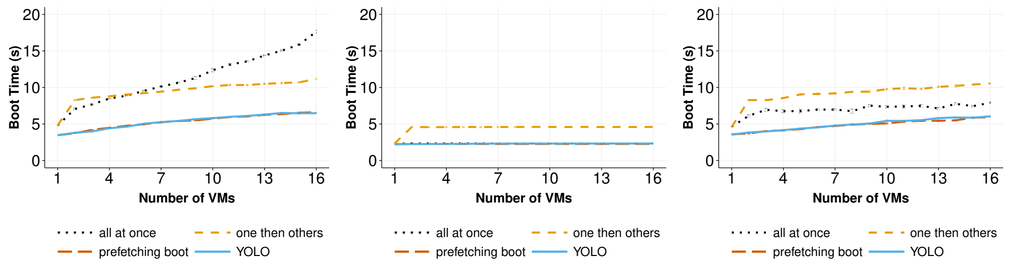
Figure 5.3 – Overhead of serving boot's I/O requests directly from the memory *vs.* a dedicated SSD

5.3 VM Boot Time Evaluation

In this section, we analyze the boot time of VMs with two scenarios. The first one is to boot multiple VMs simultaneously on a idle compute node (*i.e.*, there is no other workload is running on that node). In this experiment, VMs are created with the same or different backing files. The second scenario is booting only one VM under I/O or memory interference.

5.3.1 Deployment multiples VMs

VMs Deployment With the Same VMI



(a) HDD (b) SSD (c) Ceph

Figure 5.4 – Time to boot multiple VMs, which share the same VMI (cold environment: there is no other VMs that are running on the compute node)

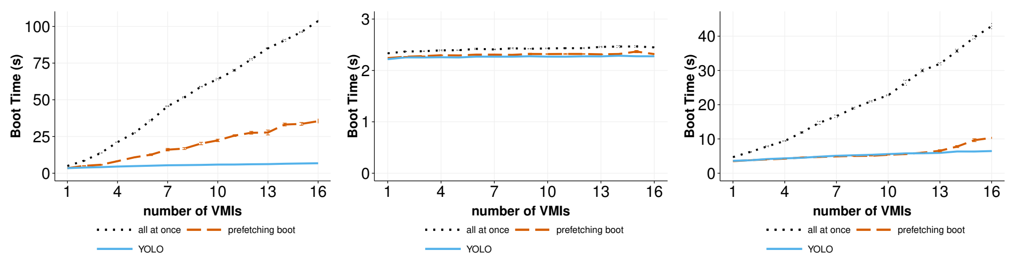
Figure 5.4 shows the time to boot up to 16 VMs leveraging the same VMI (*i.e.*, the same backing file).

On HDD (Figure 5.4a), the *all at once* boot policy has the longest boot duration because VMs perform read and write I/O operations at the same time for their boot processes. This behavior leads to I/O contentions: the more VMs started simultaneously, the less I/O throughput can be allocated to each VM. When we use the *one then others* policy, we can see better performance in comparison to the previous policy. As already explained, this is due to the cache that has been populated during the boot of the first VM. The boot time raises slightly with the number of VMs (from 2 to 16) due to the I/O writes. Regarding the *Prefetching boot* and *YOLO* strategies, they greatly speed up the VM boot time compared to other two boot policies because the VMs always get benefit from the cache for reading mandatory data. It is noteworthy that the performance gap between both strategies is not perceptible in this scenario. This is due to (i) the number of I/O requests that is not significant and (ii) that there is not cache eviction (all VMs are using the same VMI).

On SSD (Figure 5.4b), the boot time of several VMs is mostly constant for all boot policies. The I/O contention generated during the boot process on SSD becomes negligible because the I/O throughput of the SSD is higher than HDD. The I/O requests executed by the VMs can be handled quickly. Therefore, *all at once*, *prefetching boot* and *YOLO* relatively show the same boot duration. The duration of *one then others* boot policy is longer because we accumulated the boot time of the first VM.

Using Ceph (Figure 5.4c), *prefetching boot* and *YOLO* still have the best performance. The boot duration with *one then others*, *prefetching boot* and *YOLO* policy, which are mostly affected by I/O writes contention, follows the same trends when running on HDD. However, on Ceph, *all at once* are faster than *one then others*, because the bottleneck on Ceph is not on I/O disk anymore (all I/O operations go through the 10 Gbit network interface and are served by Ceph).

VMs Deployment With Distinct VMIs



(a) HDD (b) SSD (c) Ceph

Figure 5.5 – Time to boot multiple VMs, which have different VMIs (cold environment: there is no other VMs that are running on the compute node)

We performed the same experiment as the previous one, but each VM had its own VMI (*i.e.*, backing file). In this particular case, there was no interest to evaluate the *one then others* because there was no possible gain from the cache. Therefore, we only compared the results of the three other boot policies. Figure 5.5 depicts the results.

On HDD (Figure 5.5a), the boot time using *YOLO* increases slightly while *all at once* and *prefetching boot* rise sharply. For example, to boot 16 VMs, *prefetching boot* and *all at once* needs 38 s and 107 s, respectively, compared to only 7.2 s by using *YOLO*. The performance for the *prefetching boot* strategy is strongly impacted due to the fact that the script has been invoked several times simultaneously (generating a lot of competitions and a large number of seek operations on the HDD). While *YOLO* would have also suffered from this issue (we remind that the boot images have been preloaded into the memory before booting VMs), the impact would be less important because *YOLO* reads boot images in a contiguous manner. The *all at once* boot policy also suffered I/O contentions from random reads generated by multiple VMs simultaneously as in case of *prefetching boot*. However, the performance is even worse because of : (i) the I/O virtualization overhead and (ii) the I/O access pattern that cannot benefit from the read-ahead strategy of the host OS.

On SSD (Figure 5.5b), it takes less than *3* seconds to boot VMs in three cases of boot policies. This behaviour is again explained by the SSD capability.

On Ceph (Figure 5.5c), *YOLO* and *prefetching boot* rise slightly while *all at once* increases in a linear way. However, it is noteworthy to mention that *prefetching boot* is constant when the number of VMs is less than 13, and then it slightly increases. The reason for this trend is due the number requests sent through the network : when we boot more than 13 VMs at the same time, the traffic is high enough to cause a network bottleneck.

Summary

When simultaneously booting several VMs in a cold environment, the VM boot time is affected by the I/O generated by the boot process of other VMs. On HDD and Ceph, *YOLO* speeds up the VM boot time up to 13 times and 6 times respectively when VMs do not have the same VMI and 2 times when VMs are sharing the same backing file. *YOLO* does not improve the boot time on SSD in both cases with or without sharing VMIs. However, this is not true in high I/O contention scenario. Because SSD has very high I/O throughput, the I/O contention generated by VMs booted simultaneously is negligible. We will see the improvement of *YOLO* for SSD in the next scenario.

5.3.2 Booting One VM Under High Consolidation Ratio

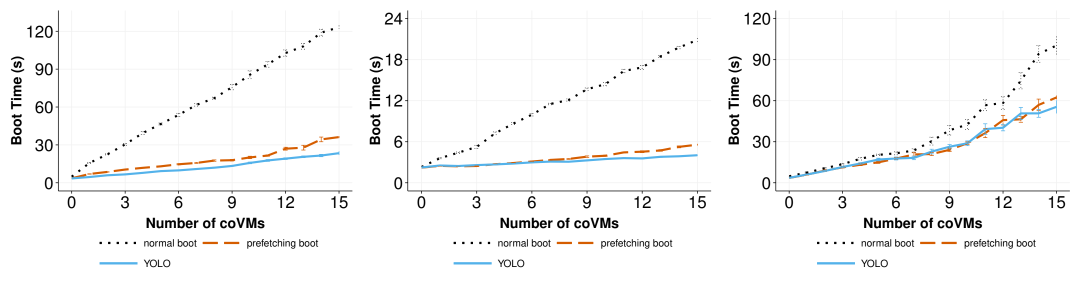
The second set of experiments is aimed to understand the effect of booting a VM in a high-consolidated environment. We defined two kinds of VMs :

— *eVM* (*experimenting* VM), which is used to measure the boot time;

— *coVM* (*collocated* VM), which is collocated on the same compute node to run competitive workloads.

We used the command *Stress* [83] to generate the I/O and memory workloads. We measured the boot time of the eVM while the multiple coVMs run their workloads (generating I/O and memory interferences). First, we started n *coVMs* where $n \in [0, 15]$, and then we start one *eVM* to measure its boot duration. Each *coVM* utilises a separate physical core to avoid CPU contention with the *eVM* while running the *Stress* benchmark. The I/O (and respectively) memory capacity is gradually used up when we increase the number of *coVMs*. Finally, there is no difference between *all at once* and *one then others* boot policy because we measure the boot time of only one VM. Hence, we simply started the *eVM* with the *normal* boot process.

Booting One VM Under I/O Contention



(a) HDD (b) SSD (c) Ceph

Figure 5.6 – Boot time of 1 VM (with *shared image* disk, *write through* cache mode) under I/O contention environment

Figure 5.6 shows the boot time of one VM on the three storage devices under an I/O-intensive scenario. *YOLO* delivers significant improvements in all cases. On HDD, booting only one VM lasts up to 2 minutes by using the *normal* boot policy. Obviously, *prefetching boot* and *YOLO* speed up boot duration much more than the *normal* one because the data is loaded into the cache in a more efficient way. However, the performance, which was almost similar for *YOLO* and the *prefetching boot* when manipulating one VMI (see Figure 5.4a), is now clearly different and in favour of *YOLO*.

The same trend can be found on SSD in Figure 5.6b where the time to boot the eVM increased from 3 to 20 seconds for the *normal* strategy, 3 to 6 seconds for the *prefetching boot*, and from 3 to 4 seconds for *YOLO*. While *YOLO* is faster than *prefetching boot* by a small amount, *YOLO* is up to 4 times faster than *all at once* policy under I/O contention of 15 coVMs. Because we generated much more I/O contention than in the first scenario, there is enough workload on SSD (which has very high I/O throughput). Therefore, we started to see the improvement of *YOLO* on SSD. An interesting point is related to the Ceph scenario. When the *coVMs* are stressing the I/O, they generate a bottleneck on the NIC of the host OS, which impacts the performance of the write requests that are performed by the *eVM* during the boot operation. This leads to worse performance for *YOLO* and the *prefetching boot* strategies than in the HDD and SSD scenarios, ranging from 3 to 58 seconds and from 3 to 61 seconds respectively. However, it is still twice faster than the boot time of *all at once*, which is 107 seconds at 15 coVMs.

To sum up, on a physical node which already had I/O workloads, *YOLO* is the best solution to boot a new VM in a small amount of time. *YOLO* reduces the boot duration 5 times on local storage (HDD and SSD) and 2 times on remote storage (using Ceph).

Booting One VM Under Memory Contention

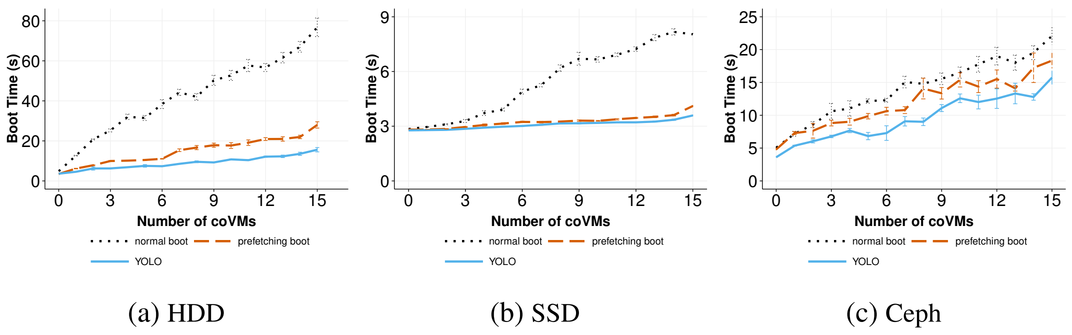


Figure 5.7 – Boot time of 1 VM (with *shared image* disk, *write through* cache mode) under memory usage contention environment

We use this scenario to assess the influence of not having enough space to load and keep the boot images into the memory of both *prefetching boot* and *YOLO* strategies. Figure 5.7 gives the results we measured.

On HDD, the *normal* boot time can reach up to 4 times longer compared to the other two methods. With *prefetching boot*, the prefetched data stays in the memory to reduce the boot time until the page cache space is claimed. In this situation, the hypervisor might have to to read the prefetching data on the storage device once again. While comparing to *YOLO*, the boot data stays in the memory space that has been allocated by *YOLO*, hence, cannot be removed by the kernel. For this reason, under memory-intensive environment, *YOLO* is almost 2 times faster than *prefetching boot* with 15 *coVMs* that stress the whole memory. On SSD, the different between *YOLO* and *prefetching boot* is small thanks to the performance of SSD. It is also true for Ceph, which has high read performance in general.

Summary

Using *YOLO* under I/O and memory intensive scenarios enables faster boot times in comparison to the normal boot approach. We underline that the gain should be even more important under when several VMs would be booted simultaneously under such intensive conditions. Regarding the memory impact, it would be interesting to conduct additional experiments in order to better understand the influence of the SWAP for *YOLO*. Indeed, when there is not enough memory at the host OS level, the *YOLO* daemon should be impacted by the SWAP mechanism. In such a case, it would be probably better to access boot images directly from a dedicated fast efficient storage device instead of putting the boot image into the *YOLO* memory. By such a way, it would be possible to prevent *YOLO* to suffer from SWAP operations. As we already observed in Figure 5.3, accessing directly a SSD device is almost similar in terms of performance than accessing the memory. Such an experiment under a memory intensive environment should be however performed to confirm this assumption.

5.4 Docker Container Boot Time Evaluation

In this section, we evaluate the boot time of dockers with two scenarios which are similar to the VM ones: (i) booting several dockers at the same time and (ii) booting only one docker under I/O contention.

When we discussed the amount of manipulated data during a boot process in Chapter 3, we observed that VMs mainly access the boot data in their images. However, for docker, the majority of I/O accesses is related to the docker binaries together with their associated libraries and configuration files. Thus, we use *vmtouch* to enforce Docker daemon data to stay in the cache before we boot. In this evaluation, we compare 3 different boot policies: (i) normal boot which is booting all dockers at once; (ii) *YOLO*; and (iii) *YOLO* + *vmtouch*.

5.4.1 Booting Multiple Distinct Containers Simultaneously

Similarly to VMs, we discuss in this paragraph the time to boot several different containers simultaneously. Figure 5.8 presents the results. Although *YOLO* reduces the time to boot containers, the time increases more significantly in comparison to VMs. This is due to the write operations that need to be completed as explained in Section 2.3.1. Understanding how such writes can be handled more efficiently is let as future works. Overall, *YOLO* enables the improvement of the boot time by a factor 2 in case of HDD (Figure 5.8a). The trend for SSD is similar to the VM one: there is no enough competition on the I/O path to see an improvement.

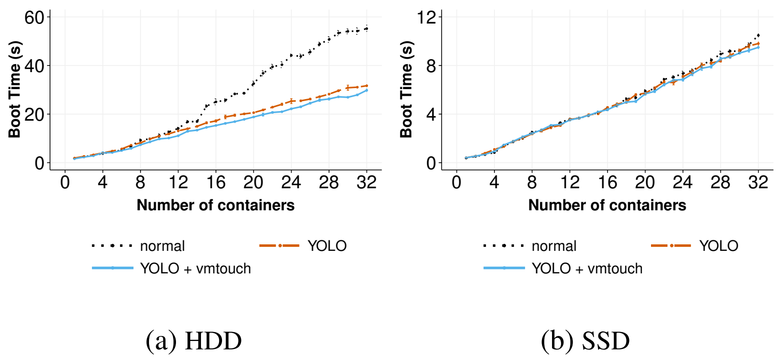


Figure 5.8 – Boot time of different docker containers on different storage devices

5.4.2 Booting One Docker Container Under I/O Contention

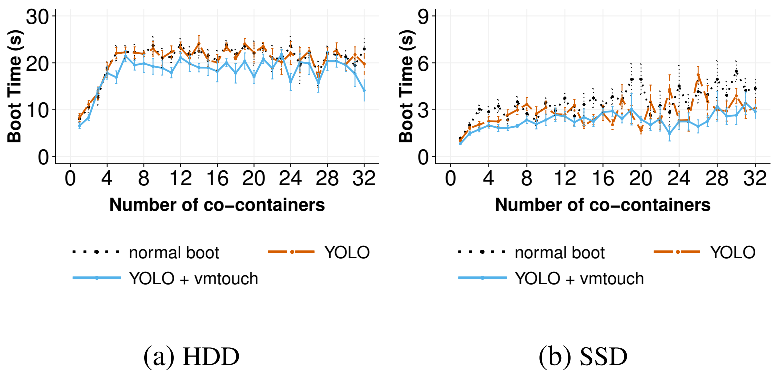


Figure 5.9 – Boot time of one debian docker container under I/O contention

In this paragraph, we discuss the time to boot a container under I/O contention. Figure 5.9 depicts the results: the boot time is increasing until it becomes quite stable. When a container is started, Docker needs to generate the container layer with all the directories structure for that container. As mentioned, this action generates write operations on the host disk, which suffer from the I/O competition. Although *YOLO* and *YOLO + vmtouch* help mitigate the read operations, Docker still waits for the finalization of the container layer to continue its boot process. Therefore, the gain of *YOLO* is much smaller than for VMs.

5.5 *yolofs* Overhead

Although booting VMs as fast as possible is the objective of our study, the performance of applications or services running inside VMs should be taken also into account. To this aim, we performed two different experiments to evaluate the I/O performance a VM can expect once it has been booted using the *YOLO* mechanism.

Table 5.2 – Time (second) to perform sequential and random read access on a backing file of VMs which are booted by normal boot and YOLO on three storage devices.

	HDD		**SSD**		**Ceph**	
	ext4	*yolofs* +ext4	ext4	*yolofs* +ext4	ext4	*yolofs* +ext4
Sequential Read	19.047 s	19.074 s	3.540 s	4.084 s	9.7 s	10.55 s
Random Read	13.405 s	13.553 s	6.408 s	6.692 s	11.27 s	12.25 s

In the first experiment, we compared the read performance when accessing data stored in the backing file with the additional *yolofs* layer (*i.e.*, *yolofs* +ext4) and the straightforward way (*i.e.*, ext4 only). We evaluated both sequential and random access. For sequential read, we measured the time a VM needs to read sequentially a whole 2.5 GB file. For random read, we read randomly 868 MB on a 2.5 GB file. Table 5.2 presents the time we observed. The difference of read performance of a VM booted by the two methods is at worst 10%.

Table 5.3 – The number of transactions per second (tps) when running *pgbench* inside a VM booted using *YOLO* and normal way on 3 types of storage devices.

	HDD	SSD	Ceph
yolofs	139	1205	145
Normal I/O path	140	1226	164

In the second experiment, we used *pgbench* [84] to measure PostgreSQL performance (in transactions per second). The VMI used in this experiment already contained pgbench benchmark (with PostgreSQL 9.4.17). We stored the 1.34 GB test database (with over 5 million rows of data) on a QCOW file of a VM. After booting the VM, we performed pgbench with the default TPC-B test (involving five *SELECT*, *UPDATE*, and *INSERT* commands per transaction). Table 5.3 presents the number of transactions per second when we used pgbench to access the database. The read/write accesses to the database of the VM is not handled by *yolofs* because the database is stored on the QCOW file and it is not located in the *yolofs* mount point. In other words, only I/O reads access to the files related to PostgreSQL application will go through *yolofs*. Consequently, *YOLO* has a similar performance compared to VM boot in a normal way.

To conclude, the overhead caused by FUSE in *YOLO* surfaces when a VM has to read big chunk of data on the backing file. However, in most practical cases, the backing file contains only essential application and system files that are shared with many VMs, other data of those VMs are stored on the QCOW file. This has been confirmed in a study [69], the authors showed that only a small fraction of the VMI is accessed by VMs throughout its run-time. Accordingly, VMs booted by *YOLO* still maintain the same performance for running the applications or services.

5.6 Integrating *YOLO* into cloud systems

The proposed methodology *YOLO* has been applied to both VMs and containers solutions. In fact, *YOLO* uses user-space components that are suitable for both full-virtualization (QEMU-KVM) as well as Containerization (Docker), which shows its flexibility to apply to a number of virtualization solutions. In order to further demonstrate the adaptability of the method, we have integrated *YOLO* to OpenStack. The motivation behind our choice of OpenStack is its large ecosystem with wide adoption by many organizations and businesses. Getting *YOLO* to work on OpenStack and reduce the boot time of VMs/container in this platform brings a great benefit to the huge community of OpenStack users.

We performed a simple test to check if *YOLO* works with OpenStack or not by changing the backing file of a VM to the *yolofs* mount point, then we performed a normal process using OpenStack. Our preliminary results (in Table 5.4) shows that *YOLO* speeds up the VM boot time at least 30% in a cold environment and up to 39% in an I/O contention environment. The early result gives a positive outcome of integrating *YOLO* to OpenStack.

Table 5.4 – Booting VM with OpenStack

	Without YOLO	With YOLO	% Speedup
Boot 1 VM	3.93 s	2.71 s	30 %
Boot 1 VM under I/O stress	8.23 s	5.03 s	39 %

YOLO works using any storage backend and is transparent to the VMs, the hypervisor, the kernel of the host/guest OS and the Container managers. This characteristic eases the integration of *YOLO*, allows *YOLO* to be deployed on a wide range of existing systems.

5.7 Summary

Starting a new VM or container in a cloud infrastructure depends on the time to transfer the base image to the compute node and the time to perform the boot process itself. According to the consolidation rate on the compute node, the time to boot a VM or a container can reach up to one minute and more. Results in the previous chapter showed that booting a VM/container generates a large amount of I/O operations and the I/O access is the most influence factor to the boot time of VMs and containers. To mitigate the overhead of these operations, we proposed *YOLO*. *YOLO* relies on the boot image abstraction which contains all the necessary data from a base image to boot a VM/container. Boot images are stored on a dedicated fast efficient storage device and a dedicated FUSE-based file system is used to load them into memory to serve boot I/O read requests. We discussed several evaluations that show the benefit of *YOLO* in most cases. In particular, we showed that booting a VM with *YOLO* is at least 2 times and in the best case 13 times faster than booting a VM in the normal way. Regarding containers, *YOLO* improvements are limited to 2 times in the best case. Although such a gain is interesting, we claim that there is space for more improvements. Chapter 7 will elaborate more on these improvements.

IV

Conclusions and Perspectives

6

Summary of Contributions

In Cloud Computing era, whenever a user wants to execute an application, the executable binary together with its required dependencies and its related applications are wrapped around by a layer of abstraction, which can be a virtual machine (VM) or a container or a lightweight VM (like kata-container [57]). Users do not really care what types of abstractions are used and only concern about how their applications are running: will they run in a stable behavior? Will they be secured and isolated from other applications and systems? Will they have good performance with different workloads? One issue that is important for both users and infrastructure providers is how fast can these "machines" boot? Deploying a new VM or container in a cloud infrastructure is a time-consuming process. It depends on the time to select the hosting node, the time to transfer the VMI or container image to the compute node and finally the time to boot the "machine". The boot time can affect the users' experience and may cause negative effects on their business. Furthermore, it can also affect the cloud system in case of handling high workloads and requests, or auto scaling systems.

Currently, users may have to wait from tens of seconds to several minutes before they can access their VMs/Containers. Obtaining a deep understanding of the boot operation in a cloud environment helps us identify which factors affect the VM/Container boot time and their influence levels. In this dissertation, we discussed several experiments that cover many related scenarios on the boot operation. We conducted performance analyses in order to investigate in details VM and container boot time [19, 20]. To the best of our knowledge, this is the first study that deals with this question under so many different resources contention scenarios. Specifically, we performed a large combination of experiments in order to understand how resource workloads such as CPU utilization, memory usage, I/O and network bandwidth on several storage devices and with different boot parameters impact the behavior of booting VMs as well as containers. Performing reproducible experiments on cloud computing platforms includes complex tasks with complicated workflows and generally a huge number of configurations to consider. Our experiments (more than 14.000) are performed inside the clusters of Grid5000 where resources for running long time experiments are only available during nights or weekends. Therefore, we followed a software-defined experiment approach, we created a script that allows us to execute all the experiments in an automatized and replicable manner. The study shows that the VM/Container boot duration is not only affected by the co-workloads and the number of VMs/Containers simultaneously deployed but also the parameters that have been used to configure VMs/Containers. Furthermore, on different storage devices, booting VMs/Containers with the same configuration leads to various behaviors.

When comparing the boot time between containers and VMs, it is obvious that containers boot much faster than VMs. However, the boot duration of containers also has an increasing trend like the VM boot time when we boot multiple containers at the same time or under a workload contention environment. In brief, I/O throughput is the most significant factor on both VMs and containers boot duration. This behavior of VMs/containers boot time should be taken into account when we want to deploy new VMs/containers. Lately, a recent trend called nested virtualization surfaces. Specifically, it is the possibility to have a virtualization layer inside another virtualization layer. This is the scenario where we can run a VM inside a VM, or essentially, a container inside a VM. For example, it is possible to run Docker containers inside a Linux VM deployed on top of an Amazon Web Services EC2 instance. Having an understanding of the boot behavior of both VMs and containers helps interpret the behavior of the nested virtualization scenario.

As we mentioned before, I/O throughput is the most significant factor on both VMs and containers boot duration, and mitigating the overhead on I/O path will result in a better boot performance for VMs/Containers. Previous observations have shown that only a small portion of the image is required to complete the VM boot process. To confirm the observation, we made an analysis on the I/O operations that occur during the boot process. It enabled us to conclude that (i) like VMs, containers only require to access a small part of the image to complete the boot process and (ii) unlike VMs, the amount of manipulated data for a container is much smaller in comparison to the I/O operations performed by the container management system itself. Based on these conclusions, we proposed in this thesis a novel method - called *YOLO* (You Only Load Once). We introduced the boot image abstraction which contains all the necessary data from a base image to boot a VM/container. Boot images are stored on a dedicated fast efficient storage device and a dedicated FUSE-based file system is used to load them into memory to serve boot I/O read requests. To reduce the I/O operations related to the VM/container management system itself, we decided to use the Linux program `vmtouch`, which is able to lock specific pages in the Linux system. Our evaluation of *YOLO* shows that booting a VM with *YOLO* is at least 2 times and in the best case 13 times faster than booting a VM in the normal way. Regarding containers, *YOLO* improvements are up to 2 times in the best case. Furthermore, *YOLO* can be easily applied to other types of virtualization (*e.g.*, Xen) and containerization because it does not require intrusive modifications on the virtualization/container management system nor the base image structure. Due to the fact that *YOLO* is running in the user space, it has some overhead on the I/O path. Nevertheless, this overhead has been proved to be acceptable in [81].

We successfully applied *YOLO* into OpenStack, a popular cloud system, in order to demonstrate the adaptability of the proposed solution. In the initial experiment, we achieved at least 30% improvement on the boot time of VM.

In the following and last chapter, we give opportunities for future research.

7

Directions for Future Research

Contents

In this last chapter, we discuss some perspectives of the present works. First, we give some improvements suggestions for YOLO *and second, we present a potential resource allocation solution that takes into account the boot duration.*

7.1 Improving YOLO

In this section, we describe a few improvements that it would be interesting to develop in *YOLO*.

Speeding the start up the applications

It would be interesting to study whether it makes sense to also construct the boot image abstraction with the data that is mandatory to start the application services. Current experiments have been done by using SSH as the signal to the end of the boot operation (in other words, we put in a boot image all the blocks that are mandatory to reach the start of SSH). In many public cloud system, users usually have at least one application or service running on a VM/container. As a result, it is more reasonable to also have the application/service started in a fast manner. The boot image creation process can be extended in order to include the data related to the boot plus the data related to the starting of the expected services. The drawback of this procedure might be the enlargement in the storage of the boot image, as well as the reduction in the amount of duplicated boot data among boot images which might affect the reusability of the boot images.

Reducing the storage space for boot images

First, we are investigating the capacity of deduplication techniques to reduce the size of all boot images. Each image has an underlying operating system and a set of applications installed in it. More specifically, if several images differ only in the installed applications and share a common underlying operating system, it would be interesting to generate only one boot image for these images. We can create a boot image for each family of images that shares the same operating system (*e.g.*, Ubuntu, Debian, Windows, etc.). Yet each operating system can have different versions and it can be inefficient to create a boot image for each version. Different minor versions of a same operating system can share multiple boot data, especially if they use the same kernel, thus, it is possible to create a boot image for each group of images which belong to the same operating system family. This method helps reduce the duplicated boot data to be persisted and served by *YOLO*.

The same methodology can be used for images that have different versions of an application running on the same operating system. As discussed above, a boot image can contain boot data for applications. It is quite common to have a specific image for running an application with all of its dependencies (*e.g.*, a Microsoft SQL Server runs on Windows Server image or a deep learning image with Tensorflow). Then each group of these specific images can share a boot image. Another method for deduplication of boot data is that we can create only one *big boot image* for all images in the system. While the second method offers the greatest reduction in the term of duplicated boot data, it increases the complexity while reduces the manageability and extensibility of the boot image, compared to the first method. Altogether, the deduplication technique is an improvement to the memory footprint and persistent size of *YOLO* overall.

Using deduplication techniques to reduce the size of all boot images seems a promise solution but we need to evaluate carefully the performance when keeping only one big boot image on memory.

Removing the overhead of *YOLO* after booting

A second technical improvement would consist in redirecting all the application's I/O requests to the disk directly, instead of going through *yolofs* after booting. Specifically, after the VM/container is fully booted, *yolofs* has finished its responsibility and we do not want *yolofs* to handle the I/O read requests from the VM/container anymore, hence, minimizing the overhead of *yolofs* on the VM/container performance. QEMU supports a feature to change the link between QCOW file and backing file of a VM disk. Therefore, it should be possible to leverage this mechanism to dismiss the FUSE mount point after the boot operation. However, this mechanism requires to restart the VM. To execute such a change in an online fashion, extensions at the hypervisor level would be required.

7.2 *YOLO* in a FaaS model

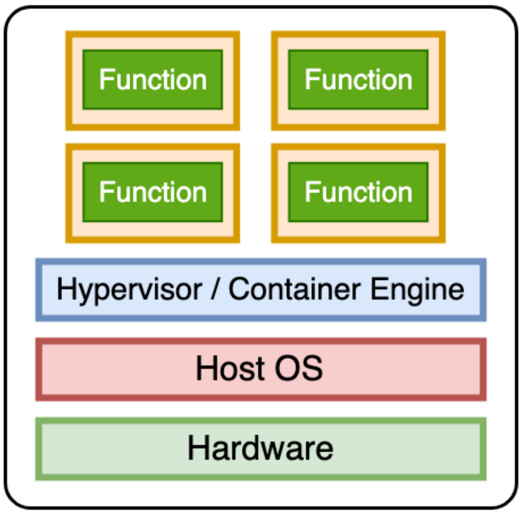


Figure 7.1 – Overview of a Function as a Service (FaaS) model

FaaS, which stands for Function as a Service, is a relatively new concept and recently gains more attention in the Cloud Computing communities. Many cloud giants have heavily invested in this technology and FaaS is now implemented in services such as AWS Lambda, Google Cloud Functions, IBM OpenWhisk, and Microsoft Azure Functions. Essentially, FaaS allows the users to run their backend code (a function) without going through the complexity of building and managing their infrastructure to host that piece of code. A function is encapsulated by a layer of virtualization (container, VM, or nested container) as illustrated in Figure 7.1. FaaS can scale horizontally using automatic resource provisioning and allocation by the provider. The ability to quickly scale out "machines" that host users' functions creates the requirement of booting "machines" in an instant. This is the area where *YOLO* proves to be useful to improve the boot time of the encapsulating layer and it is worth the effort to integrate *YOLO* into the FaaS underlying infrastructure. The transparency of *YOLO* further helps the integration process.

7.3 An allocation strategy with boot time awareness

Virtual Machine Placement is a well-known problem in a cloud system, and it is defined as the process to select the appropriate physical machine (PM) to run a given VM that satisfies a set of constraints. There are thousands proposed VM placement algorithms by previous works, with each of them tries to focus on some specific goals (e.g. optimizing operational cost, energy consumption, security, or SLA, etc.) [85, 86, 87]. Each algorithm performs well under certain conditions.

VMI transferring is usually considered as the most time-consuming stage in the whole VM deployment process. While there are a lot of solutions to speed up the VMI transferring time efficiently, it still takes time to perform the VM boot process. In fact, the duration of a VM boot process by itself varies from a few seconds to several minutes according to the number of concurrent VMs deployment and the system load, as we largely discussed in this thesis. A VM placement strategy that considers the rapid VM booting as one of the key conditions is required to provide dynamic scalability and fast provisioning in an IaaS cloud system. Thanks to *YOLO* which can give some guarantee in boot duration (as all the read accesses are mitigated), it would be possible to create a VM scheduler that takes into account the VM boot time.

For example, OpenStack implements a Filter scheduler service to determine which host a VM should launch. From a pool of available hosts, the scheduler will filter out hosts that do not satisfy the criteria of the filters. Then the designated host is selected from the filtered pool of hosts. We can use a VM boot time model based on *YOLO* as a filter to predict the VM boot time given the current resource utilization of the host. The unsatisfied hosts will be discarded from the pool.

A

Appendix A: VM Boot Time Model

Contents

In this Appendix, we present and evaluate our proposed VM boot time model. Thanks to the detailed analysis of the boot time behavior in Chapter 4, we can identify which factors are the most dominant factors that impact to a boot process in a shared resource environment. From that results, our proposed model can calculate the boot duration of a VM based on the resources utilization percentage of the system. Having an accurate model for the VM boot time helps researchers evaluate the real cloud system more correctly.

A.1 Motivation

Base on the analysis from Chapter 4, it is clearly that VM boot operations are complex processes that involved a significant number of parameters. The time it takes to boot a VM is an important element for many operations of a cloud system. Therefore, estimating the correct boot time of a VM is a necessary step for several studies such as designing scheduling policy or performing cloud benchmarks. In order to study large scale cloud systems, a cloud simulation tool that is capable of simulating as close as possible real cloud behaviors is a huge advantage. It helps researchers to focus only on the parts they are interesting about without facing the aforementioned challenges. To deliver such a framework, VM operations such as booting, migrating, suspending, resuming and turning off should be correctly modeled. While actions such as turning off can be modeled as a constant time activity, other operations such as booting or migrating require advanced models.

A recent survey on cloud simulation tools [88] showed that current models of VM boot time are ignored because they assumed that the VM boot time duration is insignificant. In addition, Costache et al. [43] included a simple VM boot time model by set the duration to a small constant. This inaccurate estimating can impact the system's performance. For instance, because the VM boot process consumes some resources, other applications performance on the host will be impacted if the boot process takes a long time. Besides, if we need an additional VM for a task, and the time to boot that VM is longer than the time that task finished. These lead to resource waste and unnecessary cost due to imprecise VM boot time information.

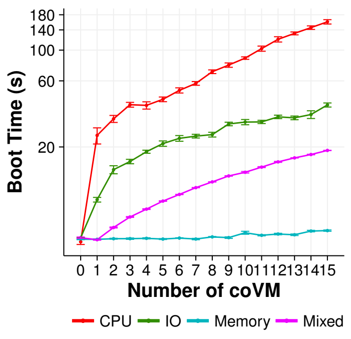
In this Appendix, using the obtained results in Chapter 4, we propose a model that is capable of calculating the VM boot time and taking into account resources contentions caused by other computations and data loading actions within the whole system. Not only resource workloads such as CPU utilization, memory usage, I/O and network bandwidth but also the method we use to boot VMs or the type of storage device can affect the boot time and should be considered in the model to have a correct measurement. Delivering a first model for the VM boot operation is an important contribution as several people might erroneously consider that the time of VM boot process is negligible. Through evaluation, we also confirmed that our model correctly reproduced the boot time of a VM under different resources contention.

A.2 VM Boot Time Model

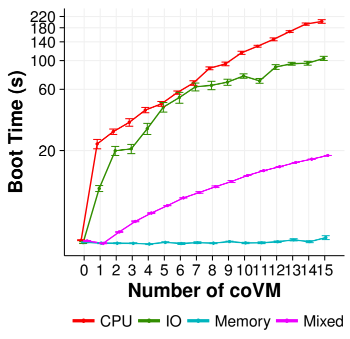
We used the coVMs to generate stress on the shared resources on a compute node so that we can identify the effect of different resources on the VM boot time, as described in Section 4.3. To clearly compare between the impact of different resources, we put those impact factors together in one chart for each of the storage device (in Figure A.1). As concluded in Section 4.3, the VM boot time is mainly influenced by CPU and I/O factors. Moreover, within a VM boot process, CPU has to check devices and set up necessary initialization; while the VM has to load kernel image from disk to its memory, therefore the VM boot time model has to deal primarily with processing time of I/O and CPU. Thus, we can split the boot time of a VM into the time of two components:

$$t = time_{IO} + time_{CPU} \tag{A.1}$$

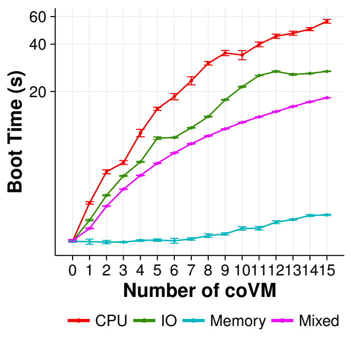
We convert Figure A.1 by transforming the x-axis from number of coVMs into the utilization percentage of I/O throughput and CPU capacity and the y-axis to the boot time of a eVM in seconds. Figure A.2 presents this conversion. The correlation between



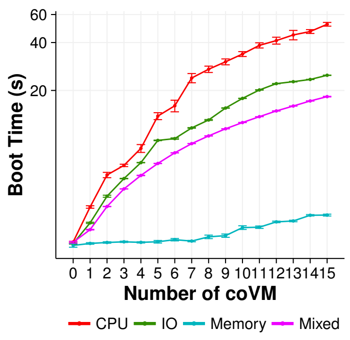
(a) *Shared* image - on HDD



(b) *No Shared* image - on HDD



(c) *Shared* image - on SSD



(d) *No Shared* image - on SSD

Figure A.1 – The comparison of VM boot time with 4 factors: I/O throughput, CPU capacity, memory utilization and the mixed factors, on HDD vs SSD. The y-axis shows VM boot time in seconds on a log scale with base 10

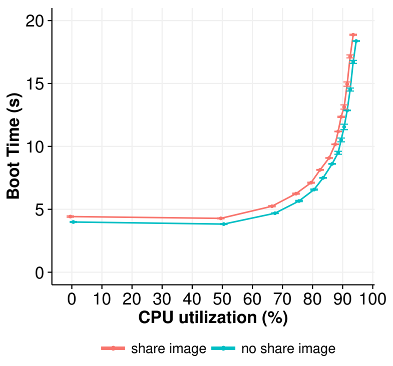
the boot time and the utilization percentage of both CPU capacity and I/O throughput is the upward curve. In fact, when the utilization reaches 100%, theoretically the boot process can last indefinitely. Therefore, the boot time model will have the form $\frac{\gamma}{1-x}$ where x is the resource utilization percentage and γ denotes the constant boot time when there is no resource utilization.

Figure A.2 shows that the boot time for I/O throughput utilization increases exponentially so that we add the exponential growth function to this model. Regarding the CPU capacity utilization, we see no such high exponential trend. Furthermore, when there is no resource contention, which means the utilization percentage is zero, the boot time of each component is a constant, denoted α and β correspondingly. Respectively, we can model them as:

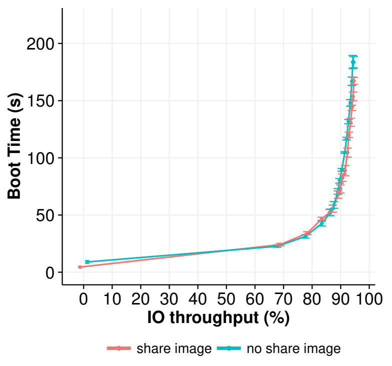
$$time_{IO} = \frac{e^x \times \alpha}{1 - x} \tag{A.2}$$

$$time_{CPU} = \frac{\beta}{1 - y} \tag{A.3}$$

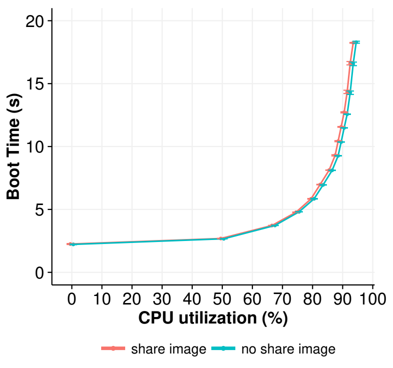
with:
— x is the utilization percentage of I/O throughput
— y is CPU utilization percentage
— α is the time that a VM needs to perform I/O operations in boot process when there is no resources contention
— β is the time that CPU takes to run operations in boot process when there is no resources contention



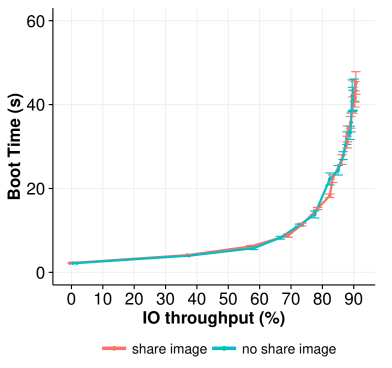
(a) CPU utilization on HDD



(b) I/O utilization on HDD



(c) CPU utilization on SSD



(d) I/O utilization on SSD

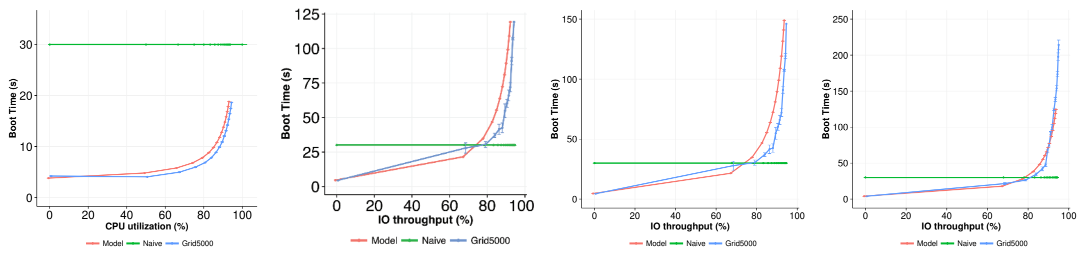
Figure A.2 – The correlation between VM boot time and resources utilization with two disk creation strategies, on HDD and SSD

A.3 Model Evaluation

To validate the correctness and the accuracy of our VM boot time model, we compared the boot time from Grid5000 experiments with the boot time calculated from the model. To perform such a comparison, we needed to identify the α and β parameters of our model.

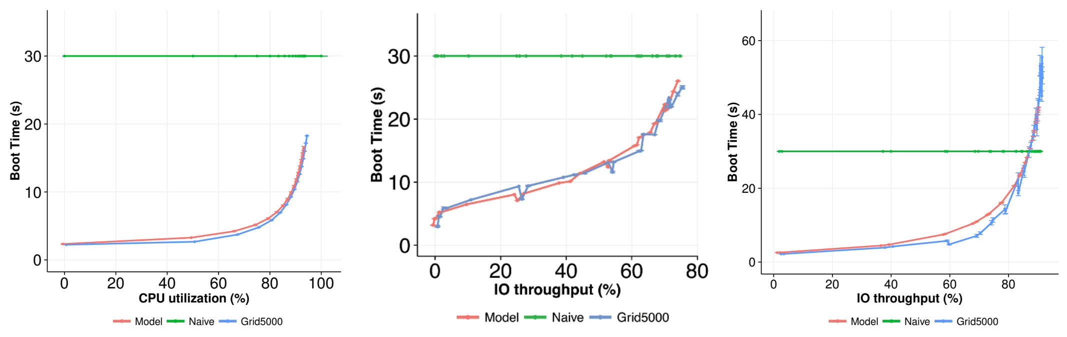
$$t = \frac{e^x \times \alpha}{1-x} + \frac{\beta}{1-y} \tag{A.4}$$

Those parameters vary according to different hardware configurations and guest operating systems. Therefore, to evaluate the model, we run a calibration experiment, *i.e.*, an experiment where one VM is started with no resource contention. Since there is no resource contention ($x = 0$ and $y = 0$), then the boot time model becomes $t = \alpha + \beta$. The CPU time β is the necessary time for calculation to boot a specific operating system and we can retrieve it in the system log. The model variables are the I/O throughput and the CPU utilization percentage. We used these inputs to calculate the boot time with various utilization levels.



(a) CPU (b) Mixed workload (c) IO, *Shared* image (d) IO, *No Shared* image

Figure A.3 – Boot time with different resource contentions on HDD



(a) CPU (b) Mixed workload (c) IO

Figure A.4 – Boot time with different resource contentions on SSD

Figures A.3 and A.4 show the comparison of boot time under resources contentions between the Grid5000 experiments, our model and a naive one (*i.e.*, with a constant boot time of 30 seconds [43]) with HDD and SSD machines respectively. We evaluated the model on three resource contention cases: CPU, mixed workload (using *Pgbench*) and I/O contention. It is clear that the naive model cannot represent the variation of VM boot time under different conditions. The naive model overestimates the boot time in CPU contention while it underestimates by a large margin in high I/O contention.

Figure A.3 shows that our model can keep up with the upward curve of VM boot time under different workloads stress on HDD. The deviation in CPU contention case is within 2 seconds in Figure A.3a. For the mixed workload (Figure A.3b), the average deviation is 15 seconds, but when we get high resource contention, at 90% CPU and I/O throughput utilization, the deviation increases up to 37 seconds, knowing that the boot time can reach up to 130 seconds. Under I/O stress, the boot time can rise as high as 210 seconds. The difference on average between our model and the Grid5000 experiments is 10 seconds for *shared* disk (Figure A.3c) and 25 seconds for *no shared* disk (Figure A.3d). However, when I/O resource utilization reaches over 90%, the deviation can be up to 23 seconds and 89 seconds for each disk type, respectively. In the case of HDD, under I/O contention, the model is less accurate due to the large difference in boot time of the two disk creation strategies. With the *no share* disk, there is a higher probability that the HDD needs to do more seek operations, which leads to the high deviation of the model. However, our VM boot time model does not model such a phenomenon yet.

The graph in Figure A.4a shows that our model also successfully represents the upward curve of VM boot time for SSDs. The deviation is within 1.5 seconds for CPU stress and 4 seconds for mixed worload. The comparison between the Grid5000 experiment and the model under I/O contention using SSDs is illustrated in Figure A.4c. Under high I/O contention, the boot time can rise up to 70 seconds. The model differs from the Grid5000 experiment by 25 seconds maximum when the I/O utilization reaches 90% and 5 seconds on average.

A.4 Conclusion

Based on the results, we can confirm that a simple model based on constant values is definitely not accurate since it cannot handle the variation of the boot time under workloads that may increase up to 50 times under high I/O contentions. On the other hand, our model succeeds to follow such fluctuations with deviations within 2 seconds in case CPU contention, and 10 seconds in average for I/O contention (when resource utilization is lower than 90%). The time to boot a VM grows from a few seconds to several minutes depend on the contention that occurs on the I/O and CPU resources. We introduce a VM boot time model to give a more accurate estimation of the time a VM needs to boot in a shared resource environment.

Regarding the future work, it would be interesting to extend the VM boot time model to capture the HDD seek time when there are I/O competitive workloads. Furthermore, a container boot time model can be obtained based on performing a more detailed analysis on the results of Chapter 4. Finally, both VM and container boot time model can be integrated into existing cloud simulators such as SimGrid [89].

Résumé en Français

Contexte

L'émergence du Cloud Computing a révolutionné le secteur des technologies de l'information et de la communication. Profitant de l'abondance des ressources associée au développement rapide des technologies Web, le Cloud Computing a permis de démocratiser le concept d'"informatique en tant qu'utilitaire". Aujourd'hui, les utilisateurs peuvent accéder à différents services exécutés sur le cloud, depuis la navigation avec Google Maps, la visualisation de films à la demande avec Netflix, ou encore la téléconférence avec des partenaires commerciaux utilisant Skype à l'autre bout du monde. Par ailleurs, de nouvelles entreprises peuvent concrétiser leurs idées sans avoir à déployer et gérer leur propre infrastructures. Tous cela est rendu possible par la puissance de calcul offertes à la demande par les infrastructures de type cloud.

Une technologie clé dans le développement et l'adoption du Cloud Computing est la virtualisation. La virtualisation peut être considérée comme l'abstraction d'un objet physique ou d'un service. L'utilisation de virtualisation permet notamment de diviser les ressources physiques en groupes de taille différente et donc de les partager entre plusieurs "environnements virtualisés". Cela constitue un moyen simple et efficace pour utiliser les ressources physiques offertes par une infrastructure cloud tout en apportant un certain niveau d'isolation entre les utilsateurs. Parmi les différents types de virtualisation qui ont été développés, les deux technologies les plus importantes sont les machines virtuelles (VM) et les conteneurs. Une machine virtuelle est la combinaison de différentes ressources physiques sous une couche d'abstraction sur laquelle les utilisateurs peuvent démarrer un système d'exploitation afin d'effectuer des tâches spécifiques. Les conteneurs peuvent être vus comme une approche plus moderne et surtout plus légère de la virtualisation des ressources. De nombreuses études [5, 6, 7, 8, 9, 10] ont comparé ces deux technologies sur l'axe de la performance. Tous ces travaux ont abouti à la même conclusion: les performances d'une application s'exécutant au dessus d'un conteneur sont proches de celles du système nu, alors qu'il y a une dégradation significative des performances lors de l'exécution de la même application au dessus une VM, en particulier pour les opérations d'entrée/sortie.

Gardant à l'esprit ce concept de virtualisation, un défi important pour les opérateurs de clouds est de pouvoir provisionner aussi rapidement que possible une VM ou un conteneur. Ce processus dit de "provisionnement" peut être d'un durée plus ou moins longue. Il comprend trois étapes: (i) après réception de la demande de provisionnement, le système de gestion de ressources identifie un nœud physique approprié pour héberger la VM / le conteneur, (ii) l'image disque permettant de démarrer cette machine est transférée depuis une base de données vers le nœud désigné, et (iii) la machine virtuelle / le conteneur est démarré. Chacune de ces étapes est complexe et peut impacter le temps de provisionement de l'environement demandé. De nombreuses politiques ont été étudiées pour identifier de manière efficace, la machine physique qui va héberger l'environnement. Elles essaient d'optimiser des objectifs spécifiques tels que l'économie d'énergie, la garantie de qualité de service, l'équilibrage de charge, etc. [11, 12]. Selon les caractéristiques de la demande, la disponibilité des ressources physiques et les critères de l'algorithme de séléction, la durée de l'opération d'identification peut varier. La seconde étape, le transfert de l'image disque associée à l'environnement, a également fait l'objet de nombreuses recherches [13, 14, 15, 16, 17]. De nos jours, dans la plupart des plateformes de cloud, les images de machines virtuelles ou de conteneurs sont stockées dans une base de données distante et sont transférées sur un hôte physique en vue du démarrage d'une nouvelle machine. Diverses techniques ont été proposées afin d'optimiser le transfert de ces images sur le réseau, notamment le transfert d'image par des protocoles "pair-à-pair", la déduplication ou encore des méthodes de mise en cache. La durée de cette étape est généralement considérée comme la plus longue du processus de provisionnement. La dernière étape du processus de provisionnement consiste à démarrer la machine virtuelle / le conteneur lui-même. Le temps associé à cette action est souvent négligé (ou considéré constant dans le meilleur des cas). Cependant, il a été montré que dans la plupart des clouds IaaS publics tels qu'Amazon EC2, Microsoft Azure ou RackSpace, les utilisateurs peuvent attendre plusieurs minutes pour obtenir un nouvel ordinateur virtuel [18]. Une telle durée de démarrage a un impact négatif sur les services de cloud computing. Cette situation est critique lorsque les clients doivent activer des machines virtuelles / conteneurs pour traiter un pique d'activités sur leurs applications. Dans un environnement contraint (i.e., une machine physique surchargée), le temps de démarrage d'une machine virtuelle peut atteindre plusieurs minutes là où il en prend généralement une vingtaine de secondes. De la même manière, de nombreux utilisateurs ont une idée fausse concernant le temps de démarrage des conteneurs. Bien que les conteneurs soient dits «instantanément» prêts au démarrage, le taux de charge éffèctif de la machine physique a un impact sur le temps nécessaire pour mettre à disposition le conteneur.

Objectifs

Dans cette thèse, nous montrons combien il est essentiel de limiter les interférences pouvant se produire lors du démarrage d'une machine virtuelle ou d'un conteneur. Plus précisément, nous présentons une analyse complète des performances du processus de démarrage de la machine virtuelle / du conteneur. Cette analyse montre l'impact des environnements virtualisés co-localisés sur la durée de démarrage d'une nouvelle machine virtuelle ou d'un nouveau conteneur. Afin d'y remedier, nous proposons une approche permettant de réduire au maximum le nombre d'operations d'entrées/sorties afin d'accélérer le processus de démarrage d'une machine virtuelle et d'un conteneur.

Contributions

Comprendre les pénalités de performances et de temps de démarrage des machines virtuelles / conteneurs

Actuellement, les utilisateurs peuvent avoir besoin d'attendre de quelques dizaines de secondes à plusieurs minutes avant de pouvoir accéder à leurs VM / conteneurs. Obtenir une compréhension approfondie de l'opération de démarrage dans un environnement cloud nous aide à identifier les facteurs qui affectent le temps de démarrage de la machine virtuelle / du conteneur et leurs niveaux d'influence. Nous avons effectué un nombre conséquent d'expériences afin de comprendre les temps de démarrage des machines virtuelles, des conteneurs et des conteneurs imbriqués (c'est-à-dire un conteneur s'exécutant dans une machine virtuelle). Plus précisément, nous avons effectué de manière automatisé plus de 14 400 expériences, représentant plus de 500 heures de traitement sur l'infrastructure Grid'5000. À notre connaissance, il s'agit de la première étude traitant de cette question dans le cadre de nombreux scénarios de conflit de ressources.

Cette campagne d'expériences nous a permis d'analyser finement le comportement du temps de démarrage d'une machine virtuelle et d'un conteneur. Les résultats obtenus ont permis de montrer que la durée d'exécution du processus de démarrage est affectée par les environnements co-localisés, le nombre de VM / conteneurs déployés simultanément, et également par les paramètres utilisés pour configurer les machines virtuelles / conteneurs. Lors de la comparaison des charges de travail, nos résultats montrent que le temps de démarrage de la machine virtuelle, introduit une surcharge négligeable en termes de mémoire et de bande passante réseau (sauf dans le cas d'un volume distant). Les entrées/sorties disques sont le facteur le plus déterminant quelquesoit le type de périphériques de stockage.

L'ensemble de cette campagne d'expériences ainsi que les principaux résultats obtenus ont donné lieu à un rapport de recherche [19]. Nous avons également exploiter cette étude pour proposer un modèle de temps de démarrage pour les machines virtuelles. La motivation de ce travail etait de fournir un modèle pouvant être intégrer dans les outils de simulation de type SimGrid. Ces travaux qui ont été menés en parallèle de la problématique d'optimisation du temps de démarrage sont présentés en annexe du présent manuscrit et ont été publiés à la conférence IEEE PDP en 2017 [20].

YOLO: accélérer le temps de démarrage des ordinateurs virtuels et des conteneurs

Afin d'accélérer le processus de démarrage, nous avons conçu *YOLO* (*You Only Load Once*), un mécanisme qui minimise le nombre d'opérations d'Entrées/Sorties générées au cours d'un processus de démarrage. *YOLO* repose sur le concept de *boot image*. Un *boot image* contient toutes les données nécessaires d'une image disque usuel effectuer un processus de démarrage. Ces nouvelles images dédiées au processus de démarrage sont stockées directment sur chacun des noeuds pouvant héberges une VM ou un conteneur, plus précisemment sur un périphérique de stockage à accès rapide, tel qu'une mémoire vive ou un disque SSD dédié. À chaque démarrage d'une machine virtuelle / d'un conteneur, *YOLO* intercepte les accès en lecture et les sert directement.

Notre évaluation de *YOLO* montre que le démarrage d'une machine virtuelle avec *YOLO* est au moins 2 fois et dans le meilleur des cas 13 fois plus rapide que le démarrage normal d'une machine virtuelle. En ce qui concerne les conteneurs, les améliorations *YOLO* vont jusqu'à 2 fois dans le meilleur des cas. De plus, *YOLO* peut être facilement appliqué à d'autres types de virtualisation (*e.g.*, Xen) et de conteneurisation car il ne nécessite pas de modifications intrusives sur le système de gestion des conteneurs / virtualisation ni sur la structure d'image de base.

La méthode proposée a été publiée dans la conférence internationale Europar [22] et des évaluations plus détaillées sont présentées dans un rapport de recherche [21].

Directions pour la recherche future

Malgré l'efficacité de *YOLO*, il y a encore matière à amélioration. Tout d'abord, il serait intéressant d'étudier la pertinance de construire également l'abstraction d'image de démarrage avec les données obligatoires pour démarrer les services d'application. L'inconvénient de cette procédure pourrait être l'élargissement du stockage de l'image de démarrage, ainsi que la réduction de la quantité de données de démarrage dupliquées entre les images de démarrage, ce qui pourrait affecter la possibilité de réutilisation de ces dernières.

Deuxièmement, si plusieurs images diffèrent uniquement par les applications installées et partagent un système d'exploitation sous-jacent commun, il serait intéressant de générer une seule image de démarrage pour ces images. Nous pouvons créer une image de démarrage pour chaque famille d'images partageant le même système d'exploitation. Cette méthode permet de réduire le nombre de données de démarrage dupliquées à conserver et à servir par *YOLO*. Utiliser des techniques de déduplication pour réduire la taille de toutes les images de démarrage semble une solution prometteuse, mais il sera utile d'évaluer avec soin les performances lorsque nous conservons une seule image de démarrage volumineuse en mémoire.

Enfin, une fois que le conteneur/machine virtuelle est complètement initialisé, il n'est plus utile que *yolofs* traite les demandes de lecture d'Entrées/Sorties de la machine virtuelle/du conteneur. Ainsi, il serait pertinent d'étudier comment cela pourrait ils a mis en place et le gain apporté.

Publications

Virtual Machine Boot Time Model, T. L. Nguyen and A. Lebre. *In Proceeding of IEEE Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on IEEE*, March 2017

Conducting Thousands of Experiments to Analyze VMs, Dockers and Nested Dockers Boot Time, T. L. Nguyen and A. Lebre. *Research Report RR-9221, Inria Rennes Bretagne Atlantique*, November 2018

YOLO: Speeding up VM Boot Time by reducing I/O operations, T. L. Nguyen and A. Lebre. *Research Report RR-9245, Inria Rennes Bretagne Atlantique*, January 2019

YOLO: Speeding up VM and Docker Boot Time by reducing I/O operations, T. L. Nguyen, R. Nou and A. Lebre. *In Proceeding of 25th Europar conference*, August 2019

References

[1] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," *arXiv preprint arXiv:0901.0131*, 2008. ix, 10, 11

[2] VMWare, "Understanding full virtualization, paravirtualization, and hardware assist," 2008. [Online]. Available: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf ix, 14, 15

[3] A. Sampathkumar, "Virtualizing intelligent river®: A comparative study of alternative virtualization technologies," 2013. ix, 21, 22, 23

[4] D. Inc., "Use the OverlayFS storage driver," 2019. [Online]. Available: https://docs.docker.com/storage/storagedriver/overlayfs-driver/#how-the-overlay-driver-works ix, 40

[5] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, "A comparison of virtualization technologies for hpc," in *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*. IEEE, 2008, pp. 861–868. 3, 95

[6] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 386–393. 3, 95

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172. 3, 95

[8] A. Sonone, A. Soni, S. Nathan, and U. Bellur, "On exploiting page sharing in a virtualised environment-an empirical study of virtualization versus lightweight containers," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 49–56. 3, 95

[9] P. Sharma, L. Chaufournier, P. J. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study." in *Middleware*, 2016. 3, 21, 22, 95

[10] C. Lin, H. Pai, and J. Chou, "Comparison between bare-metal, container and VM using tensorflow image classification benchmarks for deep learning cloud platform," in *Proceedings of the 8th International Conference on Cloud Computing*

and Services Science, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018., 2018, pp. 376–383. 3, 95

[11] Z.-H. Zhan, X.-F. Liu, Y.-J. Gong, J. Zhang, H. S.-H. Chung, and Y. Li, "Cloud computing resource scheduling and a survey of its evolutionary approaches," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 63, 2015. 3, 96

[12] S. Singh and I. Chana, "A survey on resource scheduling in cloud computing: Issues and challenges," *Journal of grid computing*, vol. 14, no. 2, pp. 217–264, 2016. 3, 96

[13] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009, p. 7. 4, 27, 96

[14] D. Jeswani, M. Gupta, P. De, A. Malani, and U. Bellur, "Minimizing Latency in Serving Requests through Differential Template Caching in a Cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 269–276. 4, 27, 28, 96

[15] K. Razavi and T. Kielmann, "Scalable virtual machine deployment using VM image caches," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 65. 4, 27, 28, 47, 96

[16] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 181–195. 4, 27, 29, 96

[17] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "Comicon: A co-operative management system for docker container images," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 116–126. 4, 27, 29, 96

[18] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 423–430. 4, 26, 31, 96

[19] T. L. Nguyen and A. Lèbre, "Conducting thousands of experiments to analyze vms, dockers and nested dockers boot time," Tech. Rep., 2018. 4, 48, 81, 98

[20] ——, "Virtual Machine Boot Time Model," in *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*. IEEE, 2017, pp. 430–437. 4, 32, 33, 81, 98

[21] T. L. Nguyen, R. Nou, and A. Lèbre, "Yolo: Speeding up vm and docker boot time by reducing i/o operations," Tech. Rep., 2019. 5, 98

[22] ——, "Yolo: Speeding up vm and docker boot time by reducing i/o operations (forthcoming)," 2019. 5, 98

[23] S. Garfinkel, *Architects of the information society: 35 years of the Laboratory for Computer Science at MIT*. MIT press, 1999. 10

[24] "Now anyone can train imagenet in 18 minutes - fast.ai," https://www.fast.ai/2018/08/10/fastai-diu-imagenet, accessed: 2019-03-23. 10

[25] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011. 11

[26] S. N. T.-c. Chiueh and S. Brook, "A survey on virtualization technologies," *Rpe Report*, vol. 142, 2005. 12

[27] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, 2008. 18, 36, 46

[28] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005. 18

[29] AMD, "Amd-v," Jan. 2012. [Online]. Available: https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf 18

[30] R. Singhal, "Inside Intel next generation Nehalem microarchitecture," in *Hot Chips*, vol. 20, 2008, p. 15. 18

[31] AMD, "AMD-V™ Nested Paging," 2008. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf 18

[32] Intel, "Virtual machine device queues," Jan. 2008. [Online]. Available: https://www.intel.com/content/www/us/en/virtualization/vmdq-technology-paper.html 18

[33] ——, "Intel® Data Direct I/O Technology (Intel® DDIO): A primer," 2012. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html 18

[34] ——, "PCI-SIG SR-IOV Primer: An introduction to SR-IOV technology," 2011. [Online]. Available: https://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf 19

[35] H. Pötzl and M. E. Fiuczynski, "Linux-VServer. resource efficient OS-level virtualization," 2007. [Online]. Available: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-151-160.pdf 20

[36] Sun Microsystems, Inc., "Solaris Containers — what they are and how to use them," 2005. [Online]. Available: http://pds11.egloos.com/pds/200808/22/85/819-2679.pdf 20

[37] "OpenVZ: a container-based virtualization for linux." [Online]. Available: http://openvz.org/ 20

[38] P. B. Menage, "Adding generic process containers to the linux kernel," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 45–57. 20

[39] "LXC: Linux container." [Online]. Available: https://linuxcontainers.org/ 20

[40] "Docker." [Online]. Available: https://www.docker.com/ 20

[41] K. Razavi, L. M. Razorea, and T. Kielmann, "Reducing vm startup time and storage costs by vm image content consolidation," in *Euro-Par 2013: Parallel Processing Workshops*. Springer, 2013, pp. 75–84. 26

[42] "Creating an Amazon EBS-Backed Linux AMI," http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/creating-an-ami-ebs.html. 26

[43] S. Costache, N. Parlavantzas, C. Morin, and S. Kortas, "On the use of a proportional-share market for application slo support in clouds," in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 341–352. 26, 90, 93

[44] R. Schwarzkopf, M. Schmidt, M. Rüdiger, and B. Freisleben, "Efficient storage of virtual machine images," in *Proceedings of the 3rd workshop on Scientific Cloud Computing*. ACM, 2012, pp. 51–60. 27, 28

[45] B. Nicolae and M. M. Rafique, "Leveraging collaborative content exchange for on-demand vm multi-deployments in iaas clouds," in *European Conference on Parallel Processing*. Springer, 2013, pp. 305–316. 27, 29

[46] W. Kangjin, Y. Yong, L. Ying, L. Hanmei, and M. Lin, "Fid: A faster image distribution system for docker platform," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2017, pp. 191–198. 27, 29

[47] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock: rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 1–12. 31

[48] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. De Lara, "Kaleidoscope: cloud micro-elasticity via VM state coloring," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 273–286. 31

[49] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 148–162. 31

[50] X. Wu, Z. Shen, R. Wu, and Y. Lin, "Jump-start cloud: efficient deployment framework for large-scale cloud applications," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 17, pp. 2120–2137, 2012. 31, 32

[51] P. De, M. Gupta, M. Soni, and A. Thatte, "Caching VM instances for fast VM provisioning: a comparative evaluation," in *European Conference on Parallel Processing*. Springer, 2012, pp. 325–336. 31, 32

[52] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *ACM SIGPLAN Notices*, vol. 46, no. 7. ACM, 2011, pp. 87–98. 31, 32

[53] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, "Optimizing VM Checkpointing for Restore Performance in VMware ESXi." in *USENIX Annual Technical Conference*, 2013, pp. 1–12. 31, 32

[54] T. Knauth and C. Fetzer, "DreamServer: Truly on-demand cloud services," in *Proceedings of International Conference on Systems and Storage*. ACM, 2014, pp. 1–11. 31, 32

[55] Z. Zhang, D. Li, and K. Wu, "Large-scale virtual machines provisioning in clouds: challenges and approaches," *Frontiers of Computer Science*, vol. 10, no. 1, pp. 2–18, 2016. 32, 47

[56] K. Razavi, G. Van Der Kolk, and T. Kielmann, "Prebaked μvms: Scalable, instant vm startup for iaas clouds," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. IEEE, 2015, pp. 245–255. 32

[57] "Kata Containers." [Online]. Available: https://katacontainers.io/ 32, 81

[58] "Kata Containers - the way to run virtualized containers." [Online]. Available: https://www.openstack.org/assets/presentation-media/Kata-Containers-The-way-to-run-virtualized-containers.pdf 32

[59] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: A performance evaluation," *Sensors*, vol. 19, no. 7, p. 1488, 2019. 32

[60] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, K. Chadwick, and S.-Y. Noh, "A reference model for virtual machine launching overhead," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 250–264, 2016. 33, 47

[61] Y. Goto, "Kernel-based virtual machine technology," *Fujitsu Scientific and Technical Journal*, vol. 47, pp. 362–368, 2011. 36

[62] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 36–47, 2012. 37

[63] The QCOW2 Image. [Online]. Available: https://kashyapc.fedorapeople.org/virt/lc-2012/snapshots-handout.html 37

[64] "Description of Cache Modes," https://www.suse.com/documentation/sles11/book_kvm/data/sect1_1_chapter_book_kvm.html. 38, 47

[65] A. Garcia, "Improving the performance of the qcow2 format," https://events.static.linuxfound.org/sites/events/files/slides/kvm-forum-2017-slides.pdf, 2017. 38, 53

[66] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. Ivanov, M. Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20. 46, 48, 66

[67] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014. 46, 48

[68] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230. 46

[69] B. Nicolae, F. Cappello, and G. Antoniu, "Optimizing multi-deployment on clouds by means of self-adaptive prefetching," in *European Conference on Parallel Processing*. Springer, 2011, pp. 503–513. 47, 77

[70] M. Imbert, L. Pouilloux, J. Rouzaud-Cornabas, A. Lèbre, and T. Hirofuchi, "Using the execo toolbox to perform automatic and reproducible cloud experiments," in *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013*, 2013. 48, 49

[71] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 2. IEEE, 2005, pp. 776–783. 49

[72] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum, "Kadeploy3: Efficient and scalable operating system provisioning for clusters," *USENIX; login:*, vol. 38, no. 1, pp. 38–44, 2013. 49

[73] B. Claudel, G. Huard, and O. Richard, "Taktuk, adaptive deployment of remote executions," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 2009, pp. 91–100. 49

[74] A. D. Brunelle, "Block i/o layer tracing: blktrace," *HP, Gelato-Cupertino, CA, USA*, 2006. 57

[75] Initial ramdisk. [Online]. Available: https://wiki.debian.org/initramfs 58

[76] virt-install. [Online]. Available: https://linux.die.net/man/1/virt-install 58

[77] mincore. [Online]. Available: https://linux.die.net/man/2/mincore 58

[78] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 75–84, 2008. 59

[79] H. Doug, "vmtouch—the Virtual Memory Toucher," https://hoytech.com/vmtouch/, 2012. 59

[80] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320. 59, 67

[81] B. K. R. Vangoor, V. Tarasov, and E. Zadok, “To FUSE or Not to FUSE: Performance of User-Space File Systems.” in *FAST*, 2017, pp. 59–72. 65, 83

[82] A. Rajgarhia and A. Gehani, “Performance and extension of user space file systems,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 206–213. 65

[83] Stress. [Online]. Available: http://people.seas.harvard.edu/~apw/stress/ 72

[84] Pgbench benchmark. [Online]. Available: https://wiki.postgresql.org/wiki/Pgbenchtesting 76

[85] R. K. Gupta and R. Pateriya, “Survey on virtual machine placement techniques in cloud computing environment,” *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, vol. 4, no. 4, pp. 1–7, 2014. 88

[86] R. Ranjana and J. Raja, “A survey on power aware virtual machine placement strategies in a cloud data center,” in *2013 International Conference on Green Computing, Communication and Conservation of Energy (ICGCE)*. IEEE, 2013, pp. 747–752. 88

[87] K. Mills, J. Filliben, and C. Dabrowski, “Comparing vm-placement algorithms for on-demand clouds,” in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE, 2011, pp. 91–98. 88

[88] A. Ahmed and A. S. Sabyasachi, “Cloud computing simulators: A detailed survey and future direction,” in *Advance Computing Conference (IACC), 2014 IEEE International*. IEEE, 2014, pp. 866–872. 90

[89] H. Casanova, A. Legrand, and M. Quinson, “Simgrid: A generic framework for large-scale distributed experiments,” in *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*. IEEE, 2008, pp. 126–131. 94

[90] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, “Performance overhead comparison between hypervisor and container based virtualization,” in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2017, pp. 955–962.

[91] Images for OpenStack. [Online]. Available: https://docs.openstack.org/image-guide/obtain-images.html

[92] C. Peng, M. Kim, Z. Zhang, and H. Lei, “VDN: Virtual machine image distribution network for cloud data centers,” in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 181–189.

[93] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, K. Chadwick, and S.-Y. Noh, "A reference model for virtual machine launching overhead," 2014.

[94] T. Hirofuchi, A. Lèbre, and L. Pouilloux, "Adding a live migration model into simgrid: One more step toward the simulation of infrastructure-as-a-service concerns," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 1. IEEE, 2013, pp. 96–103.

[95] Red Hat, "libvirt: The virtualization api," *http://libvirt.org*, 2012.

[96] T. Knauth and C. Fetzer, "Fast virtual machine resume for agile cloud services," in *Cloud and Green Computing (CGC), 2013 Third International Conference on*. IEEE, 2013, pp. 127–134.

[97] P. J. Mucci, K. London, and J. Thurman, "The cachebench report," *University of Tennessee, Knoxville, TN*, vol. 19, 1998.

[98] "LINPACK_BENCH - the LINPACK benchmark," http://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html.

[99] V. De Maio, R. Prodan, S. Benedict, and G. Kecskemeti, "Modelling energy consumption of network transfers and virtual machine migration," *Future Generation Computer Systems*, vol. 56, pp. 388–406, 2016.

[100] V. Kherbache, F. Hermenier *et al.*, "Scheduling live-migrations for fast, adaptable and energy-efficient relocation operations," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2015, pp. 205–216.

[101] B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 277–280.

Titre : Contributions à l'approvisionnement d'environnements virtualisés : la problématique des temps de démarrage des machines virtuelles et des conteneurs

Mots clés : Informatique en nuage, virtualisation système, conteneurs, temps de démarrage.

Résumé : Le processus d'approvisionnement d'une machine virtuelle (VM) ou d'un conteneur est une succession de trois étapes complexes: (i) la phase d'ordonnancement qui consiste à affecter la VM/le conteneur sur un nœud de calcul ; (ii) le transfert de l'image disque associée vers ce nœud de calcul; (iii) et l'exécution du processus de démarrage (généralement connu sous le terme « boot »).

En fonction des besoins de l'application virtualisée et de l'état de la plate-forme, chacune de ces trois phases peut avoir une durée plus ou moins importante. Si de nombreux travaux se sont concentrés sur l'optimisation des deux premières étapes, la littérature couvre que partiellement les défis liés à la dernière. Cela est surprenant car des études ont montré que le temps de démarrage peut atteindre l'ordre de la minute dans certaines conditions. Durée que nous avons confirmée grâce à une étude préliminaire visant à quantifier le temps de démarrage, notamment dans des scénarios où le ratio de consolidation est élevé. Pour comprendre les principales raisons de ces durées, nous avons effectué en jusqu'à 15000 expériences au dessus de l'infrastructure Grid5000. Chacune de ces expériences a eu pour but d'étudier le processus de démarrage selon différentes conditions environnementales. Les résultats ont montré que les opérations d'entrée/sorties liées au processus de démarrage étaient les plus coûteuses. Afin d'y remédier, nous défendons dans cette thèse la conception d'un mécanisme dédié permettant de limiter le nombre d'entrées/sorties générées lors du processus de démarrage. Nous démontrons la pertinence de notre proposition en évaluant le prototype YOLO (You Only Load Once). Grâce à YOLO, la durée de démarrage peut être accélérée de 2 à 13 fois pour les VM et jusqu'à 2 fois pour les conteneurs. Au delà de l'aspect performance, il convient de noter que la façon dont YOLO a été conçu permet de l'appliquer à d'autres types de technologies de virtualisation/conteneurisation.

Title : Fast delivery of Virtual Machines and Containers: Understanding and optimizing the boot operation

Keywords : Cloud Computing, Virtualization, Containerization, Boot Duration.

Abstract : The provisioning process of a Virtual Machine (VM) or a container is a succession of three complex stages : (i) scheduling the VM/Container to an appropriate compute node; (ii) transferring the VM/Container image to that compute node from a repository; (iii) and finally performing the VM/Container boot process.

Depending on the properties of the client's request and the status of the platform, each of these three phases can impact the total duration of the provisioning operation. While many works focused on optimizing the two first stages, only few works investigated the impact of the boot duration. This comes to us as a surprise as a preliminary study we conducted showed the boot time of a VM/Container can last up to a few minutes in high consolidated scenarios. To understand the major reasons for such overheads, we performed on top of Grid'5000 up to 15k experiments, booting VM/Container under different environmental conditions. The results showed that the most influential factor is the I/O operations. To accelerate the boot process, we defend in this thesis, the design of a dedicated mechanism to mitigate the number of generated I/O operations. We demonstrated the relevance of this proposal by discussing a first prototype entitled YOLO (You Only Load Once). Thanks to YOLO, the boot duration can be faster 2-13 times for VMs and 2 times for containers. Finally, it is noteworthy to mention that the way YOLO has been designed enables it to be easily applied to other types of virtualization (e.g., Xen) and containerization technologies.