Horizon Project

ANR call for proposals number ANR-08-VERS-010
FINEP settlement number 1655/08

Horizon - A New Horizon for Internet

WP2 - TASK 2.1: Identification and Comparison of Appropriate Virtualisation Solutions

Institutions

Brazil	**France**
GTA-COPPE/UFRJ	LIP6 Université Pierre et Marie Curie
PUC-Rio	Telecom SudParis
UNICAMP	Devoteam
Netcenter Informática ltda.	Ginkgo Networks
	VirtuOR

Contents

List of Figures

Chapter 1

Introduction

The Internet is a great success. Since its conception, it has expanded from a small testbed to the greatest network in the world uniting people through the entire globe. Although the Internet is a great success, it is currently "ossified", because it is not possible to easily create new services and innovate in the network core [1, 2, 3]. In order to keep developing new services and deal with the limitations of the current Internet, the scientific community is developing several efforts [4, 5, 6, 7] to build a new Internet architecture able to support the needs of the current and future services. There are two kinds of approaches in the development of new Internet architectures, the purist [8, 9] and the pluralist [10, 11]. The purist approaches propose new Internet architectures with a single network able to deal with all the requirements of all the services. The pluralist approaches propose new Internet architectures with multiple coexistent networks, each network with different protocol stacks with characteristics that fulfill an application or kind of applications requirements.

The Horizon Project has the objective of building a new Internet architecture based in the pluralist approach. An example of a pluralist architecture is shown in figure 1.1. In order to build Horizon Project pluralist architecture we propose to use virtualization, a technique that allows sharing computational resources [12]. Virtualization slices a real computational environment into virtual computational environments that are isolated from each other and interact with the upper computational layer as it would be expected from the non-virtualized environment. A comparison between a virtualized and a non-virtualized environment is shown in figure 1.2. In Horizon Project, we consider router resources (processor, memory, hard disk, queues, bandwidth, etc.) as the computational environment to be virtualized. A set of virtual routers and links is called a *virtual network*. Therefore, using the virtualization technique, the architecture developed in Horizon Project is able

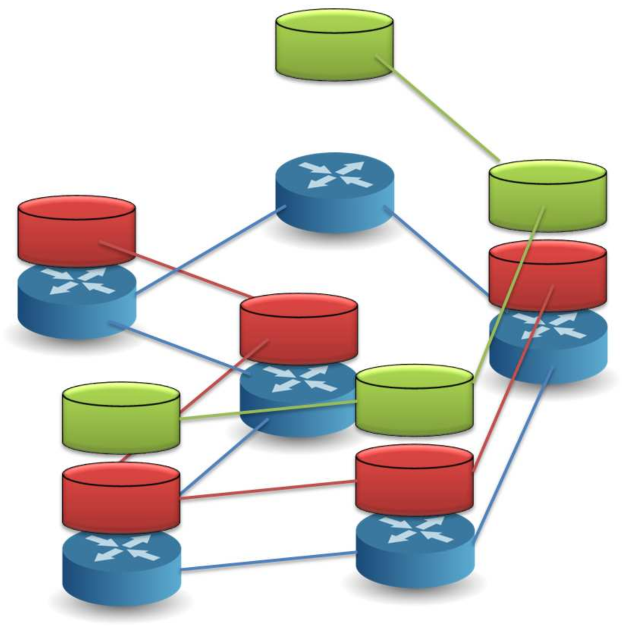

Figure 1.1: Pluralist architecture example. Each color represents a different network with independent protocol stack that share the resources from the underlying network infrastructure, shown in blue.

to have multiple concurrent virtual networks, each with a particular network protocol stack, sharing a single physical network infrastructure, as shown in Fig. 1.1.

Virtualization is commonly implemented by a software layer called hypervisor, which is responsible of multiplexing the computational resources between the multiple virtual environments, which are also known as virtual machines. Each virtual environment runs over the hypervisor, which controls the access of the physical resources. Accordingly, one of the first tasks of Horizon Project is to evaluate the overhead introduced by the hypervisor of different virtualization tools available today. In this report we will present our study on the virtualization tools we choose as candidates for usage in Horizon Project: Xen [13, 14], VMWare [15], and OpenVZ [16]. The study compares virtualization tools regarding their performance of virtualizing the main computer resources of interest to a virtual router: CPU, RAM memory, hard disk, and network. CPU is used by the virtual routers to process incoming packets and decide where to route them based on the forwarding tables. RAM memory is used to store the forwarding tables. Hard disk primary use is to store the virtual routers images. Network is used to receive and transmit the packets. For normal operation of virtual routers, CPU, RAM memory

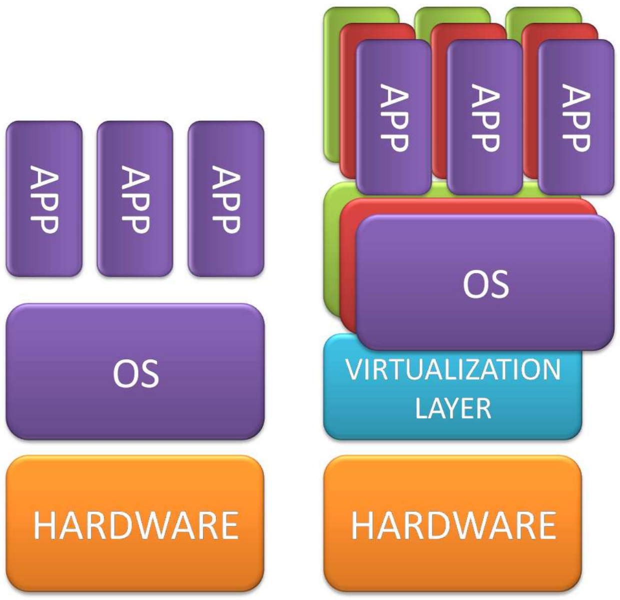

Figure 1.2: Virtualized environment example. On the left side a traditional computational environment is shown with applications executing over an operating system that uses an underlying hardware. On the right side a virtualized environment is shown in which it was added a virtualization layer that allows multiple operating systems to run concurrently each with its own applications.

and network are the most sensitive resources to virtualization overhead. Disk performance overhead is of interest because it impacts on the instantiation of new routers and in virtual router migration. In order to better understand the overhead introduced by such tools, native performance is also presented whenever applicable.

Since virtualization can cause malfunctions in time sensitive applications[17], we used a different technique than the one used in related work [18, 19]. In our tests we based our results in the time taken for a virtualized system to accomplish a task measured from an external non-virtualized computer.

We have conducted two types of experiments. The first experiments aim at analyzing the performance loss incurred by the extra layer - the hypervisor - introduced by virtualization tools. To achieve this goal, in the first type of experiment there is only one virtual machine (VM) running over the virtualization tool software. We compare the performance of native Linux with Xen, VMWare, and OpenVZ virtualization software. Our results justify the use of Xen as virtualization tool for the Horizon Project because it presents acceptable virtualization overhead , as demonstrated in Section 6.1, provides the possibility of hypervisor modification, for being open source, and provides virtual router flexibility, since it provides a virtual hardware interface allowing us to use different operating systems in different virtual routers. Additionally, one-VM experiments provide a baseline for the second type of experiments, which deal with multiple virtual machines (VMs). The second set of experiments investigates how the chosen virtualization tool scales with the number of VMs running in parallel. This kind of tests has the objective of clarifying how instantiating multiple VMs, consuming the same resources, affect the overall performance and how is fairness handled by the virtualization tool. In this kind of test we also verify how different schemes of CPU cores allocation among VMs influence the overall performance.

In Chapter 2 we present the virtualization techniques used by the virtualization tools we tested. In Chapter 3 we describe in higher details the virtualization tools. Chapter 4 describes the methodology of the tests and the testbed. Chapter 5 presents the benchmarks used in the virtualization tools comparison and in Chapter 6 we present the performance comparison results. Finally we present our conclusions in Chapter 7.

Chapter 2

Virtualization Techniques

2.1 Concepts

In order to better understand the compared virtualization tools it is important to understand the different virtualization techniques. This chapter describes deeply virtualization concepts and the virtualization techniques used by the tools we have chosen to compare.

The first concept that must be further explained is the virtualization concept. There are many different definitions for virtualization but they all agree that virtualization is a technique for sharing computational resources granting some level of isolation between the virtual environments. The classical definition according to Popek and Goldberg [12] is that virtualization is a technique that provides virtual machines, which are efficient isolated copies of the underlying hardware. Today, this concept can be generalized not only to hardware but to any computational resources layer as an OS kernel or the virtual machine abstraction used by programming languages like Java and C#.

Several challenges arise from the virtualization objectives. The first issue is scheduling all the virtual environments to access the same underlying computational resources. For hardware virtualization the shared resources are CPU, RAM, storage and network. RAM sharing can be controlled in several different ways. The virtual environment can be given access to a virtual RAM memory space that can be translated into physical RAM in a similar fashion that is done with processes by multi-task OSes. Another approach is to let the virtual environments be aware of their virtualized nature and allow them to directly access RAM after being assigned an area by the hypervisor. CPU sharing can be made in several ways and can be achieved using mechanisms like round-robin, weighted round-robin, allocation under demand and others. I/O in general can be handled in a unified fashion using buffers for storing exchanged data multiplexed and demultiplexed between the physical peripherals and the virtual peripherals from the virtual environments.

Horizon project uses commodity x86-based hardware, which imposes additional challenges to virtualization. In the beginning of virtualization development, in the 70's decade, hardware was designed to be virtualization-enabled. Mainframes in the 70's had instruction sets in which all the instructions that handled resources allocation or usage were privileged, that is, required to be in a certain CPU privilege execution level. In these CPU architectures, hardware virtualization could be reached using a technique called "de-privileging" [20], in which the virtualized OS is executed in an unprivileged context in order to generate traps whenever resources allocation or usage instructions would be executed. In that way, the hypervisor would intercept the traps and emulate the allocation or usage of the required resource in a way that is safe for the other virtual machines. According to Popek and Goldberg [12], there are three requirements for building a hypervisor for hardware virtualization: (i) efficiency, which states that a large subgroup of the instruction set from the virtual CPU should be directly executed into the real CPU without any kind of intervention from the hypervisor; (ii) resource control, which states that the hypervisor must have complete control of all the resources; and (iii) equivalency, which states that the hypervisor must provide a virtual interface to the virtual environment equivalent to the original one. The 70's mainframes would facilitate building hypervisors, since virtual machines isolation could be reached using the "de-privileging" technique. For x86-based hardware, this is not true, since for optimization purposes x86-based hardware instruction sets have instructions that handle with shared resources and do not require a privileged context, and, moreover, the x86-based instruction sets contain a group of instructions classified sensitive to privilege level, in which the instruction is executed in a different manner depending upon the current privilege level. If a de-privileged OS would execute a sensitive to privilege instruction it would fail silently, since it would not generate a trap neither would execute in the way it was intended by the OS. To handle these x86-based hardware problems there are several turnarounds that will be presented in the next sections describing the different virtualization techniques.

2.2 Full Virtualization

Full virtualization is a virtualization technique in which all the original interfaces are virtualized and the interfaces exposed to the virtual environment are identical to the original ones. In this approach, the guest OS, that is, the OS residing inside the virtual machine, does not need to be modified and can be directly executed inside the virtual machine. In order to deal with the problematic instructions from the x86-based hardware platforms, different techniques can be used. A well-known technique is binary-translation. Binary-translation checks the code to be executed searching for problematic instructions and replaces them for instructions that emulate the desired behavior. The advantage of using binary-translation is that it allows full virtualization to be used, allowing applications and OSes to be used without modifications. To obtain this advantage there is a cost tough, binary-translation imbues a high CPU overhead since all executed code must be checked and problematic detected instructions must be replaced during runtime. Recently, there has been effort from the major hardware manufactures to optimize virtualization since consolidating several servers with great idle capacity into a server with low idle capacity and several virtual servers executing has become a very common practice to cut down equipment and maintenance cost in major companies. For that matter, both AMD and Intel have developed technologies for better supporting virtualization in modern CPUs. Intel initiative, Intel Virtualization Technology (IVT), and AMD initiative, AMD Virtualization (AMD-V), provide better performance for full virtualization by introducing two new operation modes: root and non-root. The root operation mode is meant for hypervisor usage and is similar to regular CPU operation, providing full CPU control and the traditional four rings of privilege levels. The non-root mode is meant for the execution of the virtual machines. In this mode the CPU also provides four rings of privilege level and the guest OS no longer executes in a de-privileged ring but in ring 0 as it was designed for. Whenever the guest OS executes a problematic instruction, the CPU generates a trap and returns control to the hypervisor to deal with this trap. With that CPU support, binary-translation has no longer become necessary and hypervisors implementing full virtualization have greatly increased their performance since the overhead for monitoring problematic instructions is now handled by the CPU.

2.3 Paravirtualization

Paravirtualization is a virtualization technique in which the guest OS cooperates with the hypervisor for obtaining better performance. In paravirtualization, the guest OS is modified to call the hypervisor whenever a problematic instruction should be executed. In that way, the hypervisor does not need to monitor the VM execution for problematic instructions, which greatly reduces overhead in comparison to full virtualization using binary-translation. The cost for this performance improvement is that only special paravirtualized OS images modified to make hypervisor calls can be used. This is a blocking issue for virtualizing legacy OSes and requires the cooperation of the OS developer for proprietary OSes, but it is not an issue for Horizon project since the project will be developed over Linux, which is open source and already has paravirtualization support.

Chapter 3

Virtualization Tools

3.1 Xen

Xen is an open-source hypervisor proposed to run on commodity hardware platforms that uses paravirtualization technique [13]. Xen allows to simultaneously run multiple virtual machines on a single physical machine. Xen architecture is composed of one hypervisor located above the physical hardware and several virtual machines over the hypervisor, as shown in Fig. 3.1. Each virtual machine can have its own operating system and applications. The hypervisor controls the access to the hardware and also manages the available resources shared by virtual machines. In addition, device drivers are kept in an isolated virtual machine, called *domain 0* (dom0), in order to provide reliable and efficient hardware support [21]. Because it has total access to the hardware of the physical machine, dom0 has special privileges compared with other virtual machines, referred to as user domains (domUs). On the other hand, user domains have virtual drivers, called front-end drivers, which communicate with dom0's back-end drivers to access the physical hardware. Next, we briefly explain how Xen virtualizes each machine resource (processor, memory, and I/O devices).

Xen virtualizes the processor (Central Processing Unit - CPU) by assigning virtual CPUs (vCPUs) to virtual machines. Virtual CPUs correspond to the CPUs that the running processes within the virtual machine can see. The hypervisor maps vCPUs to physical CPUs. Xen hypervisor implements a CPU scheduler that dynamically maps a physical CPU to each vCPU under a certain period, based on a scheduling algorithm. The default scheduler of the used version (3.2) of Xen is the Credit scheduler, which makes a proportional CPU share. This means that Credit scheduler allocates CPU resources to each virtual machine (or, more specifically, to each its vCPU) according

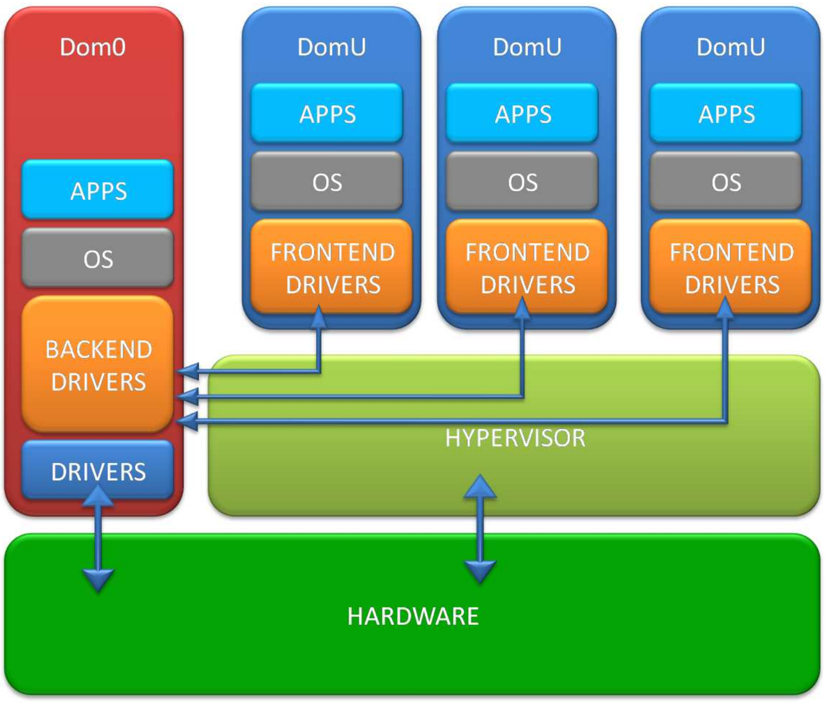

Figure 3.1: Xen architecture.

to weights assigned to virtual machines. Credit scheduler can also be work conserving on SMP (Symmetric Multi-Processing) hosts. This means that the scheduler permits the physical CPUs to run at 100% if any virtual machine has work to do. In a work-conserving scheduler there is no limit on the amount of CPU resources that a virtual machine can use.

Memory virtualization in Xen is currently done in a static way. Random-Access Memory (RAM) is divided among virtual machines, with each machine receiving a fixed amount of memory space, specified at the time of its creation. In addition, to require a minimal involvement from hypervisor, virtual machines are responsible for allocating and managing the hardware page tables. Each time a virtual machine requires a new page table, it allocates and initializes a page from its own memory space and registers it with Xen hypervisor, which is responsible to ensure isolation.

In Xen, I/O data is transferred to and from each virtual machine using shared-memory asynchronous buffer descriptor rings. The task of Xen hypervisor is performing validation checks, e.g., checking that buffers are contained within a virtual machine memory space. Dom0 access I/O devices directly by using its native device drivers and also performs I/O operations on behalf of user domains (domUs). By their turn, user domains employ their back-end drivers to request dom0 for device access [22]. A special case of I/O virtualization is network I/O virtualization, which is responsible for demultiplexing incoming packets from physical interfaces to virtual machines and also for multiplexing outgoing packets generated by virtual machines. Fig. 3.2 illustrates the default network architecture used by Xen. For each user domain, Xen creates the virtual network interfaces required by this user domain. These interfaces are called front-end interfaces and are used by domUs for all its network communications. Furthermore, back-end interfaces are created in dom0 corresponding to each front-end interface in a user domain. In order to exchange data between back-end and the front-end interfaces, Xen provides an I/O channel, which employs a zero-copy mechanism. This mechanism remaps the physical page containing the data into the target domain [22]. Back-end interfaces act as the proxy for the virtual interfaces in dom0. Front-end and back-end interfaces are connected to each other through the I/O channel. In Fig. 3.2, back-end interfaces in dom0 are connected to the physical interface and also to each other through a virtual network bridge. This is the default architecture used by Xen and it is called bridged mode. Thus, both the I/O channel and the network bridge establish a communication path between the virtual interfaces created in user domains and the physical interface.

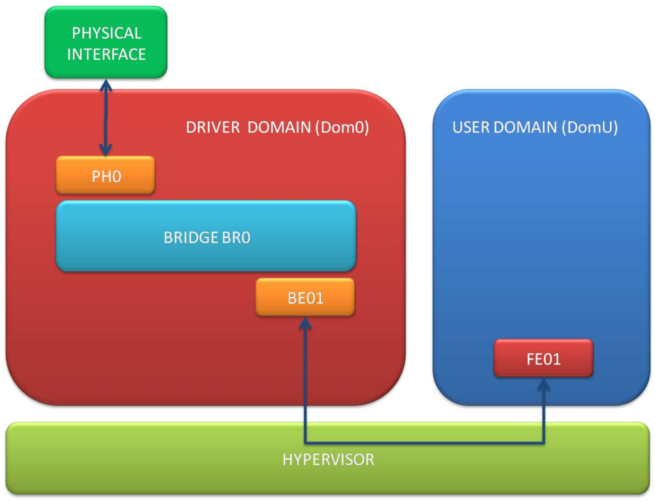

Figure 3.2: Xen network architecture.

3.2 VMware

VMware is a company that provides machine virtualization platforms for end-user and datacenter customers. VMware virtualization platforms are based on the full virtualization concept. This work evaluates a VMware datacenter-class virtualization platform called VMware ESX Server. It is mainly used for server consolidation and it is one of most used enterprise virtualization software. VMware ESX Server aims at guaranteeing virtual-machine isolation and resource-sharing fairness based on resource-allocation policies set by the system administrator. Resource sharing is dynamic, because resources can be allocated and re-allocated to virtual machines on demand [23]. Although this work evaluates the VMware ESX Server 3.5, the pieces of information reported here are from VMware ESX Server 2.5 due to the fact that VMware ESX Server is a proprietary product and there is few information about its implementation. The description below takes into account the version 2.5, considering that there are no significant changes between versions 2.5 and 3.5.

VMware architecture, as shown in Fig. 3.3, is composed of the Hardware Interface Components, the Virtual Machine Monitor, the VMkernel, the Resource Manager, and the Service Console. The Hardware Interface Components are responsible for implementing hardware specific function and create a hardware abstraction that is provided for virtual machines. It makes Virtual Machines hardware independent. The Virtual Machine Monitor (VMM) is responsible for CPU virtualization, providing a virtual CPU for each virtual machine. The VMkernel controls and manages the hardware substrate. VMM and VMkernel together implement the Virtualization Layer. The Resource Manager is implemented by VMkernel and it partitions the underlying physical resources among the virtual machines, allocating the resources for each one. VMkernel also implements the hardware interface components. Eventually, the Service Console implements a variety of services such as bootstrapping, initiating execution of virtualization layer and resource manager, and runs applications that implements supporting, managing, and administrative functions.

VMware ESX Server, as others virtualization tools, virtualizes four main resources: CPU, memory, disk, and network device. We will further detail bellow how VMware does each resource virtualization.

CPU virtualization is done by setting a virtual CPU for each virtual machine. The virtual machine does not realize that it is running over a virtual CPU, because of the fact that virtual CPUs seem to have their own registers and control structures [23]. A virtual machine can have one or two virtual CPUs. When it has more than one CPU, it is called a SMP

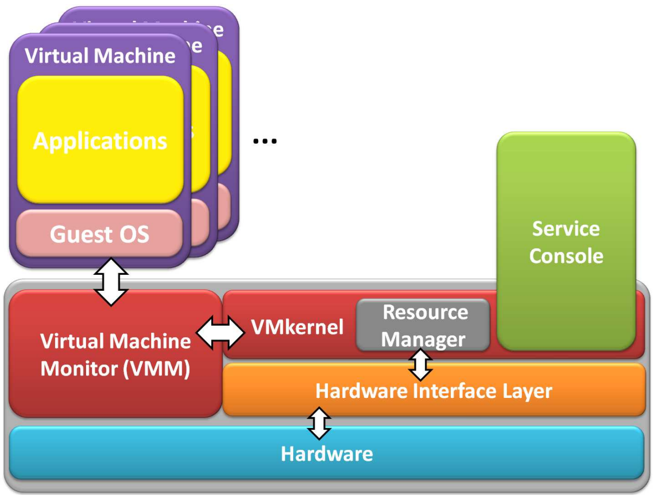

Figure 3.3: VMware architecture.

(symmetric multiprocessing) virtual machine. VMM is the responsible for CPU virtualization, by setting system state and executing instructions issued by the virtual machine.

In a virtualized environment, the guest operating system runs in a lower privilege level than it was designed to run. As mentioned in section 2.1, a classical approach to virtualize CPU resources is to trap-and-emulate, which is a technique in which the virtual machine tries executing an instruction, and, if it cannot be executed in a lower privilege level, the CPU generates a trap that is treated by the VMM, which emulates the instruction execution to the guest operating system. However, as mentioned in section 2.1, this technique does not work for the x86 architecture, since it has instructions that are sensitive to privilege level and would execute in a different way than meant by the OS without generating a trap. In order to solve this issue and keep a satisfactory performance, VMware combines two CPU virtualization techniques: direct execution and CPU emulation. The instructions from the user-space of a virtual machine are executed directly on the physical CPU, a technique known as direct execution. Guest operating system instructions that are sensitive to privilege level are trapped by the VMM, which emulates the instructions execution, adding performance overhead. Combining both techniques allows CPU intensive user-space applications to have near-native performance. The performance loss depends on the number of sensitive instructions that had to be replaced.

CPU scheduling is made by the Resource Manager. CPU scheduling is based on shares, which are units used to measure how much time is given to each virtual machine. CPU scheduling is proportional-share, meaning that

CPU time given to each virtual machine is proportional to the amount of shares it has in comparison to the total amount of shares in the system. In a SMP virtual machine, CPU allocation is different. Resource Manager schedules the two virtual CPUs one-to-one onto physical CPUs, and tries executing both virtual CPUs at the same time. VMware CPU scheduling tries to keep fairness between virtual machines CPU allocation. When a virtual machine is halted, or idle, Resource Manager schedules its CPU time to other virtual machines that are running.

VMware memory virtualization approach is to create a new level of memory address translation. It is done by providing each guest operating system a virtual page table that is not visible to the memory-management unit (MMU) [13]. Within a VMware virtualization environment, the guest operating system accesses a virtual memory space provided to the virtual machine. The guest operating system page table maintains the consistence between guest virtual pages and guest virtual "physical" pages. Guest virtual pages are virtual memory pages within a virtual machine, as in a native operating system virtual memory. However, guest virtual paging mechanism cannot access directly the physical memory, it accesses guest virtual "physical" memory. Guest virtual "physical" memory is an abstraction of the physical memory. When a guest operating system tries to execute an instruction to access physical memory, this instruction is trapped by the VMM and its address is translated to real physical address. Guest virtual "physical" memory is always contiguous, but can be mapped into not contiguous real physical memory. VMware memory sharing obeys administration policies. It can be defined a minimum and a maximum amount of physical memory to be accessed by a virtual machine. It is also possible to have virtual machines consuming more than the total amount of physical memory available on the physical machine. This is possible because the host system can also do swap as in a traditional virtual memory mechanism used in modern OSes. Memory sharing scheduler acts like CPU scheduler, but takes into account memory shares instead of CPU shares.

VMware I/O virtualization approach is to emulate the performance-critical devices, such as disk and network interface cards. The device emulation consists of presenting a device abstraction to the virtual machine which is not equal to the physical underlying hardware. As it is emulation, device accesses are not direct, but emulated by the VMkernel. The VMkernel calls the hardware interface layer, which is responsible for accessing the device driver and executing the operation on the physical hardware device. For storage virtualization, a SCSI driver is presented to the virtual machine. Virtual machine accesses this driver, VMkernel traps driver access instructions and implements virtual machine disks as files in the host filesystem.

In order to do network I/O virtualization, VMware implements the *vmxnet* [23] device driver, which is an abstraction of the underlying physical device. When an application wants to send data throw the network, the guest operating system processes the request and calls the vmxnet device driver. The device driver I/O request is intercepted by the VMM and control is transferred to VMkernel. VMkernel is independent of the physical device, so it processes the request, manages the various virtual machine requests, and calls the hardware interface layer, which implements the specific device driver. When data arrives to the physical interface, the mechanism for sending it to the specific virtual machine is the same, but in reversed order. The main overhead introduced by this mechanism is the context changing required to switch between virtual machine and VMkernel contexts. In order to decrease the overhead caused by context transitions, VMware ESX Server collects cluster of sending or receiving network packets before doing a context transition. This mechanism is only used when packet rate is high enough to avoid increasing packet delay too much.

Summing up, VMware ESX Server is a full virtualization tool which stands for providing a great number of management and administrative tools. VMware ESX Server is focused on datacenter virtualization. It provides a flexible and high performance CPU and memory virtualization. However, I/O virtualization is still an issue, since it is done by emulating the physical devices and involves some context changes.

3.3 OpenVZ

OpenVZ is an open-source operating system-level virtualization tool. OpenVZ allows to have multiple isolated execution environments over a single operating system kernel. Each isolated execution environment is called *Virtual Private Server* (VPS). A VPS looks like a physical server; it has its own processes, users, files, IP addresses, system configuration, and provides full root shell access. OpenVZ claims to be the virtualization tool which introduces less overhead, because each VPS shares the same operating system kernel, providing a high-level virtualization abstraction The main usages for this virtualization technology are in hosting, providing every customer a complete Linux environment, and in information technology (IT) education institutions, providing every student a Linux server that can be monitored and managed remotely [24]. Despite the small overhead introduced by OpenVZ, it is less flexible than other virtualization tools, like VMware or Xen, because OpenVZ execution environments have to be a Linux distribution, based on the same operating system kernel of the physical server.

OpenVZ architecture, as shown in Fig. 3.4, is composed of a modified Linux kernel that runs above the hardware. The OpenVZ modified kernel implements virtualization and isolation of several subsystems, resource management and capability to do checkpoints [16]. In addition, I/O virtualization mechanisms are provided by the OpenVZ modified kernel, which has a device driver for each I/O device. This modified kernel implements also a two level process scheduler that is responsible for in a first level define which VPS will run and, in a second level, which VPS process will run. The 2-level scheduler and some features that provide isolation between VPSs make up the OpenVZ Virtualization Layer. VPSs run above the OpenVZ Virtualization Layer. Each VPS has its own set of applications and Packages, which are segmentations of certain Linux Distributions that contains applications or services. It means that a VPS can have its own services and applications independent of the others.

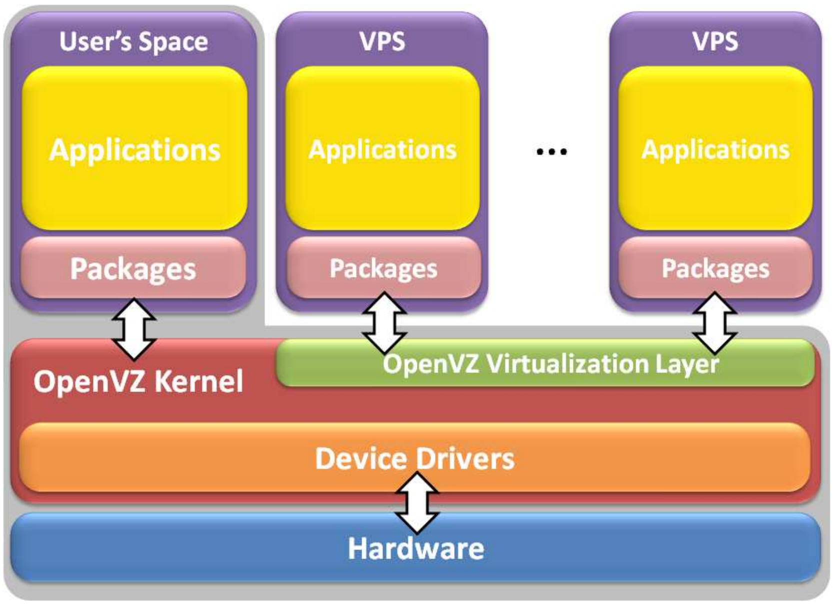

Figure 3.4: OpenVZ architecture.

Resource virtualization in OpenVZ is done by allowing or prohibiting a VPS to access a resource on the physical server. In general, resources on OpenVZ are not emulated, they are shared among VPSs. In order to define the amount of each resource that is guaranteed for each VPS, some counters (about 20) are defined in VPS configuration file. Next, we explain how OpenVZ virtualizes the processor, memory, and disk and network devices.

For processor virtualization, OpenVZ implements a two-level CPU scheduler [16]. On the first level, the OpenVZ virtualization layer decides which

VPS will execute for each time slice, taking into account the VPS CPU priority, measured in /it cpuunits. On the level-2 scheduler, which runs inside the VPS, the standard Linux scheduler defines which process will execute for each time slice, taking into account the standard process priority parameters.

OpenVZ allows VPSs to directly access the memory. However, it is less strict to memory access than other virtualization technologies, such as Xen. During a VPS execution, the memory amount dedicated for one VPS can be dynamically changed by the host administrator. It is possible by changing the virtual memory space of each VPS. OpenVZ kernel manages VPSs memory space to keep in physical memory a block of the virtual memory corresponding to the running VPS.

OpenVZ disk is a partition of the host file system. Similarly to CPU scheduling, OpenVZ disk usage is determined by a two-level disk quota. On the first level, OpenVZ virtualization layer defines a disk quota for each VPS, e.g. limiting the maximum size of folder in the host file system. On the second level, it is possible to define disks quotas for users and groups in a VPS, using standard Linux quota mechanisms.

Network virtualization layer isolates VPSs from each other and from the physical network [24]. OpenVZ default network virtualization mechanism creates a virtual network interface for a VPS and assigns an IP address to it in the host system. When a packet arrives to the host system with an IP address designed for a VPS, the host system routes the packet to the corresponding VPS. This approach of network virtualization allows VPS packets to be received and sent using the host system routing module. This simplifies network virtualization, but introduces a new hop in packets path.

Summing up, OpenVZ provides a high-level virtualization abstraction, and hence introduces less overhead than other virtualization tools. On the other hand, it seems to be more restrictive as it restricts the virtual environments to use the host system kernel.

Chapter 4

Scenario and Methodology of the Experiments

4.1 Scenario and Methodology Description

This chapter presents the methodology chosen for our tests and describes the experiments performed. For evaluating the virtualization tools, we decided to run several different tests to evaluate CPU, RAM, storage and networking performance of the virtualized environments. In order to measure the performance we chose to count the time taken for each tool to perform a task.

When virtualization is implemented, several issues arise from the fact that hardware is being divided between virtual machines, as previously discussed in section 2.1. One of the problems that can have an effect in our tests is how the hypervisor provides the timekeeping mechanisms to the virtual machine [17]. There are several ways to keep time in a computer like reading BIOS clock, using CPU registers like the Time Stamp Counter (TSC), requesting system time from the OS and others. For the system time timekeeping mechanism, Xen addresses this issue by constantly synchronizing the virtual machine clock with Dom0 clock by sending correct time information through shared memory between Dom U and the hypervisor. With that solution, system time timekeeping is correct for most of the applications, but, for applications that samples time in a higher frequency than the one that the virtual machine clock is being synchronized, there can be a cumulative time measurement error. For that matter, we will make our tests in a way that timekeeping from the virtual machine will not be considered. The main idea of the performed tests is to have an external computer responsible for measuring how long does it take for the VMs to perform a specific task in the test computer. In the single-VM tests, the external computer runs a control script, commanding the VM through SSH to start the experiment and do a specific task several times. Each time a round of the task is initiated and completed, the VM notifies the external computer using a probe packet. After the completion of all the rounds, the external computer calculates the mean value and variance of the time taken to complete a round of the specified task using a 95% confidence interval. The number of rounds was chosen in order to have small variance and it is task dependent. When presented variance in results is high, it is because of the nature of the experiment. In the multiple-VM tests, the procedure is basically the same, the only difference is that the external computer tracks multiple tasks executions at the same time. In order to make the amount of time taken to notify the external machine negligible, the tasks of all the tests were configured to make the rounds of the test last a much greater amount of time. Typical notification delays are in the order of milliseconds whereas rounds executions are in the order of minutes. Figure 4.1 illustrates the basic scenario of the single-VM tests considering a test with only one round.

For the network tests, a different approach was taken. The network tests are concerned with the throughput achieved by the VMs both when receiving and sending. For this kind of tests, it is necessary to have at least two computers, one for generating/receiving network traffic and another one for performing the complementary action. For these tests, an extra computer was used for performing the complementary action of the virtualized system, this extra computer is referred to as Network Tests Computer. For these tests, measurements were made by the network benchmarking tool itself and the used scheme is shown in figure 4.2.

4.2 Hardware/Software description

For the performed tests, as described in section 4.1, we used 3 computers. In this section we will describe the hardware and software configurations of these computers.

The first computer is the computer in which the tests are executed, it is referred to as Tests Computer in figures 4.1 and 4.2. For that role, we used a HP Proliant DL380 Generation 5 server equipped with 2x Intel Xeon 5440 CPUs, 10GB of DDR2 RAM @667MHz, an integrated 2-ports Broadcom Nextreme II Gigabit network interface and 4x 146GB SAS hard drives @10k rpm. The HP server can be configured with an independent hard-drive configuration or with a RAID configuration. We used the independent hard-drive configuration because each hard drive was installed with a different

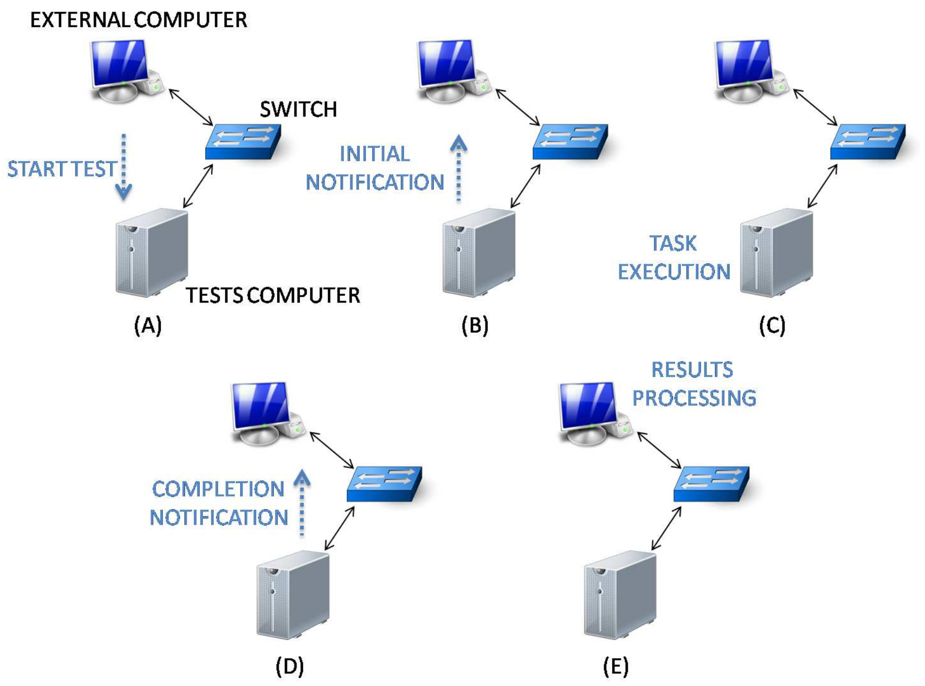

Figure 4.1: Single-VM tests default configuration. The start test (A) command is sent from the external computer to the tests computer. The tests computer sends an initial notification (B) to the external computer before the task execution (C) and sends a completion notification (D) when task execution is done. Finally the external computer processes the results (E).

software configuration for our tests. In the first hard drive, a standard Debian Linux AMD 64 install was made for usage in the native performance tests. In the second hard drive, a standard Xen 3.2-1 installation was made using a paravirtualized Debian Linux AMD 64 for domain 0 and domain U. The domain U VMs were configured with one virtual CPU, 4GB of RAM and 10GB of storage. In the third hard drive, a standard VMware ESX 3.5 installation was made and the VM was configured with one virtual CPU, 4GB of RAM, 10GB of storage and Debian Linux AMD 64. In the fourth hard drive, a standard OpenVZ over Debian AMD 64 installation was made and a container, OpenVZ's virtual environment, was created with quotas similar to the Xen and VMware VMs.

The second computer is the external computer used for controlling the experiments and processing the results, it is referred to as External Computer in the figures. For that role, we used a desktop equipped with an Intel Q6600 CPU, 4GB of DDR2 RAM @667MHz, an integrated Intel e1000e

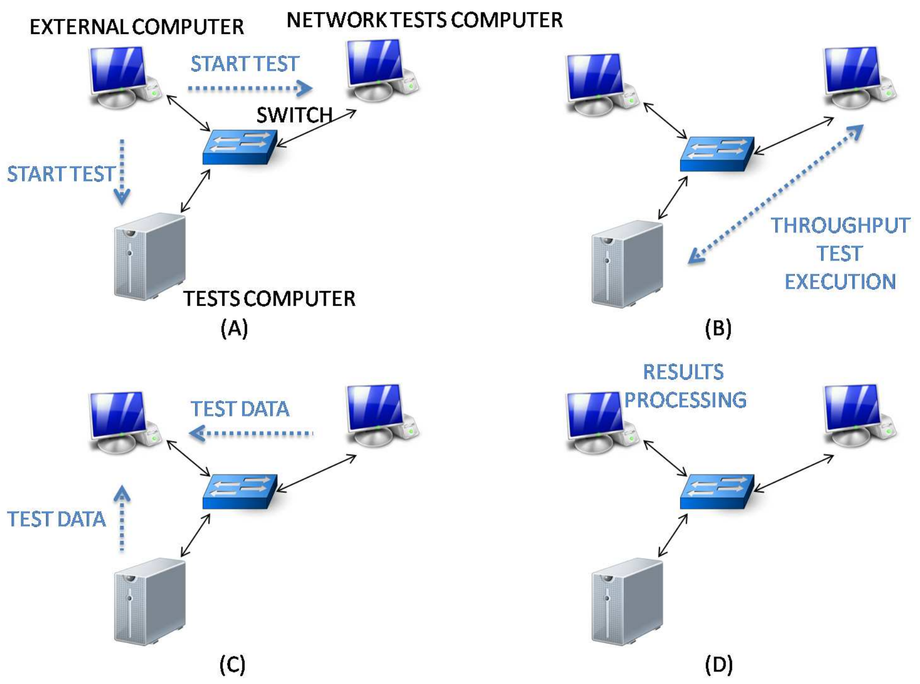

Figure 4.2: Single-VM network tests configuration. The external computer initially triggers the test (A). After receiving the command, the network benchmark is executed and test packets go through the network (B). After the test completion, both the tests computer and the network tests computer send the results to the external computer (C) to process the results (D).

Gigabit network interface and a 500 GB SATA2 hard drive @7k2 rpm. The computer was configured with a default Debian Linux AMD64 install and has all the scripts that control the tasks and processes the results. The last computer is the one used in network tests for establishing a connection to or receiving a connection from the Tests Computer, it is referred to as Network Tests Computer in the figures. This computer is a desktop equipped with an Intel E6750 CPU, 2GB of DDR2 RAM @667MHz, an integrated Intel e1000e Gigabit network interface and a 320GB SATA2 hard drive @ 7k2 rpm. The computer was configured with a default Debian Linux AMD64 install and with the network benchmarking tool. All installed Linuxes are using the 2.6.26 kernel version.

Chapter 5

Performance Evaluation - Benchmarks description

We first present the results of the experiments with one VM, which compare the performance obtained with Xen, VMware and OpenVZ against native Linux. We use benchmarks for CPU, RAM, hard disk, and network. We also show the results for Linux kernel compilation benchmark, an application which demands a mix of CPU, RAM and hard disk resources.

5.1 CPU Performance - Super Pi benchmark

The Super Pi test is a CPU-intensive task based on the Gauss-Legendre algorithm to compute the Pi value. The Gauss-Legendre algorithm is iterative and is based on many arithmetic operations. The most used arithmetic operations are sum, division, square root, potentiating, subtraction and multiplication. For this test, a shell script was developed in which the Pi value is computed with 2^{22} digits (4,194,304 digits) for ten rounds. The performance metric is the time taken to compute Pi, in seconds.

5.2 Memory Performance - MASR

MASR (Memory Allocation, Set and Read) is a memory benchmarking tool developed by the Teleinformatic and Automation Group (GTA) from Federal University of Rio de Janeiro. MASR benchmarks memory by allocating 2GB of memory, setting sequentially all memory positions to a fixed value and reading sequentially all the memory positions. MASR was developed for benchmarking memory with a deterministic number of operations, independent of the performance of the computer. Since no Linux memory benchmarking tools were found with this explicit characteristic, MASR was developed. For this test, a shell script was developed in which MASR is executed for ten rounds. The evaluated parameter is the time taken to execute MASR benchmark in each round.

5.3 Hard disk and file system Performance - Bonnie ++ and ZFG benchmarks

5.3.1 Bonnie++

Bonnie ++ is an open-source disk benchmark with the objectives to evaluate hard drive and file system performance. The main program of bonnie++ tests a single temporary file of 1GB, or multiple temporary files of 1GB for greater amounts of data, with database type access, simulating operations like creating, reading and deleting large amounts of small files in the used temporary files. The second program tests the performance of the different regions of the hard drive, reading data from blocks located at the beginning, in the middle and at the end sectors of the hard drive. For this test, a shell script executes bonnie++ for ten rounds with a parameter for using 2GB of disk space. The performance metric is the time taken to execute bonnie++ benchmark each time.

5.3.2 ZFG

ZFG (Zero File Generator) is a disk benchmarking tool developed by GTA/UFRJ. ZFG benchmarks disk continuous writing speed by writing ten times a 2GB binary file filled with zeros in each round. ZFG was developed for benchmarking disk with a deterministic number of operations, independent of the performance of the computer. Since no Linux disk benchmarking tools were found with this explicit characteristic, ZFG was developed. For this test, a shell script was developed in which ZFG is executed for ten rounds. The evaluated parameter is the time taken to execute ZFG benchmark each round.

5.4 Network Performance - Iperf

Iperf is an open-source network benchmark that allows benchmarking networks using both unidirectional and bidirectional data flows over TCP or UDP. Iperf has several parameters that can be configured, such as packet size, bandwidth that Iperf should try to achieve or test duration. In our test we used Iperf configured with unidirectional UDP data flows using 1472 bytes of payload to avoid IP fragmentation. For this test, a shell script was developed in which a unidirectional UDP data flow goes either from the tests computer to the network tests computer or the opposite direction.

5.5 Overall performance - Linux Kernel Compilation

Linux kernel compilation is a frequently used benchmarking tool [25] to evaluate overall performance because it intensively uses different parts of the computer system. The Linux kernel is composed by thousands of small source-code files, and its compilation demands intensive CPU usage, RAM read/write accesses, and short duration non-sequential disk accesses, since the files to be compiled depend on the target kernel configuration For this test, a shell script was developed in which the kernel is compiled ten times. The evaluated parameter is the time taken to make a default configuration file and compile the kernel.

Chapter 6

Performance Evaluation - Results

6.1 Single-VM tests

We present the results of the executed benchmarks for the single-VM tests we conducted. All the following graphs regarding single-VM tests will show the performance of native Linux, VMware VM, Xen VM and OpenVZ container.

6.1.1 Super-Pi

The results for the Super-Pi benchmark executed for 10 rounds are shown in figure 6.1.

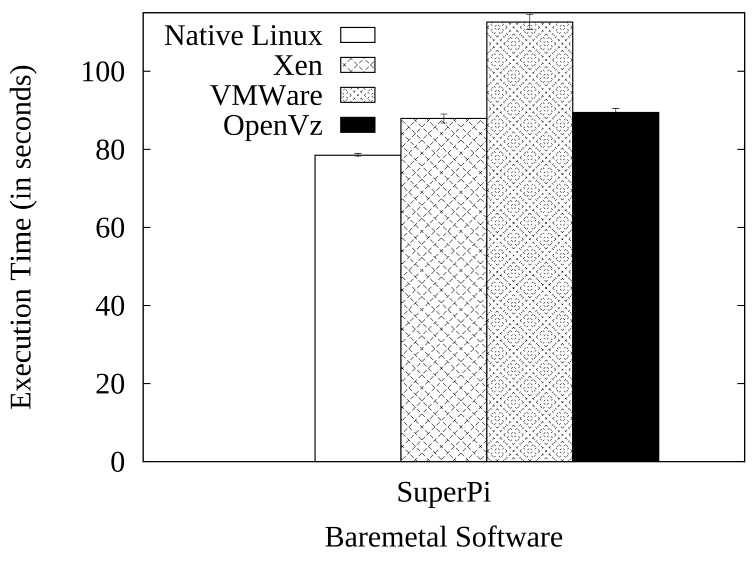

Figure 6.1: Super-Pi test. The mean execution time for a round of the test is shown in the vertical axis, smaller values are better. The virtualization tools and native Linux are spread along the horizontal axis.

The results are consistent with the expected results of having the non-virtualized system outperforming the virtualized systems. Xen has a near native performance which was expected since Xen paravirtualized VMs can directly access the memory after hypervisor approval for using a memory area and we associate most of the presented overhead to the VM CPU scheduler. VMware has the worst performance and we associate it to the extra address translation between the virtual RAM offered to the VM and the physical RAM and also to the VM CPU scheduler. OpenVZ has presented overhead a little higher than Xen due, in our opinion, to the scheduling mechanism to share CPU between the containers.

6.1.2 MASR

The results for the MASR benchmark executed for 10 rounds are shown in figure 6.2.

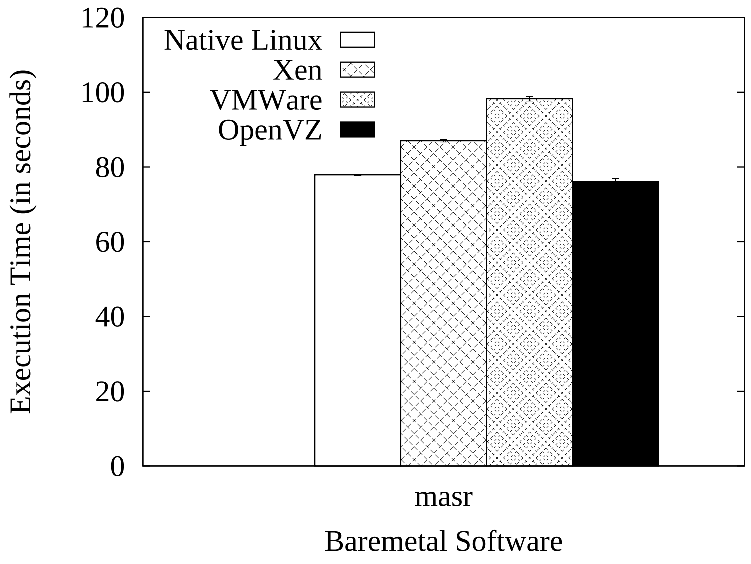

Figure 6.2: MASR test. The mean execution time for a round of the test is shown in the vertical axis, smaller values are better. The virtualization tools and native Linux are spread along the horizontal axis.

The results show that native Linux has the best performance as expected. OpenVZ achieves similar performance to native linux which was expected since OpenVZ shares memory dynamically with all the containers with a low isolation level that only guarantees a minimal amount of private memory for each container. Xen presents some overhead, which we associate with the VM scheduling mechanism. VMware has the worst performance, we attribute it to the extra memory address translation and VM scheduler. All virtualization tools presented acceptable overhead which is a very important result since memory access operations are very common in forwarding table lookup and other routing tasks which are our final objective.

6.1.3 Bonnie++

The results for the Bonnie++ disk benchmark executed for 10 rounds are shown in figure 6.3.

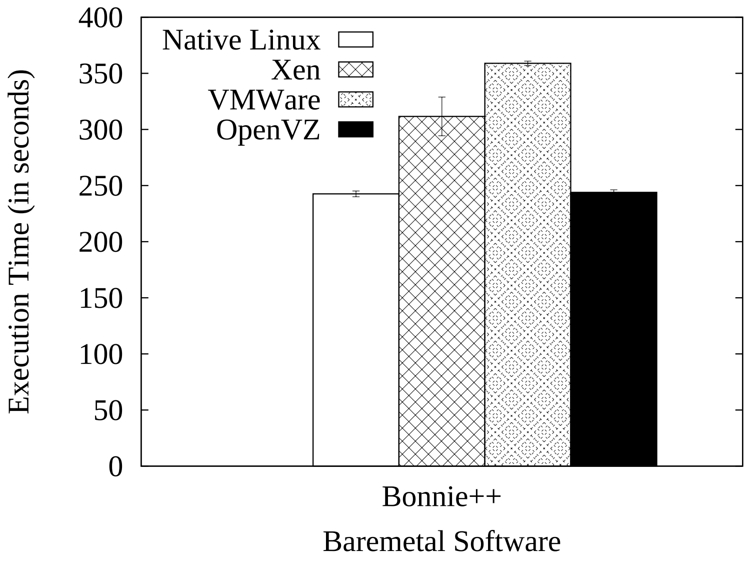

Figure 6.3: Bonnie++ test. The mean execution time for a round of the test is shown in the vertical axis, smaller values are better. The virtualization tools and native Linux are spread along the horizontal axis.

Once more, the results obtained are within what would be expected. Native Linux has the best performance and OpenVZ achieves similar results since disk access in OpenVZ is very similar to disk access in native Linux. Xen presented some overhead which we credit mostly to the extra buffer between dom0 and the VM, the I/O ring, that adds latency when reading from disk. VMware presents the worst results and we attribute the presented overhead mostly to the extra buffers in the hypervisor used to transfer data between the physical disk and the virtual disk. Disk performance is to be considered of minor importance since it does not impact much in normal router forwarding operation.

6.1.4 ZFG

The results for the ZFG test are shown in figure 6.4. Both VMware, Linux and OpenVZ results were obtained from 10 rounds. Xen results were obtained from 100 rounds in order to have an acceptable error bar.

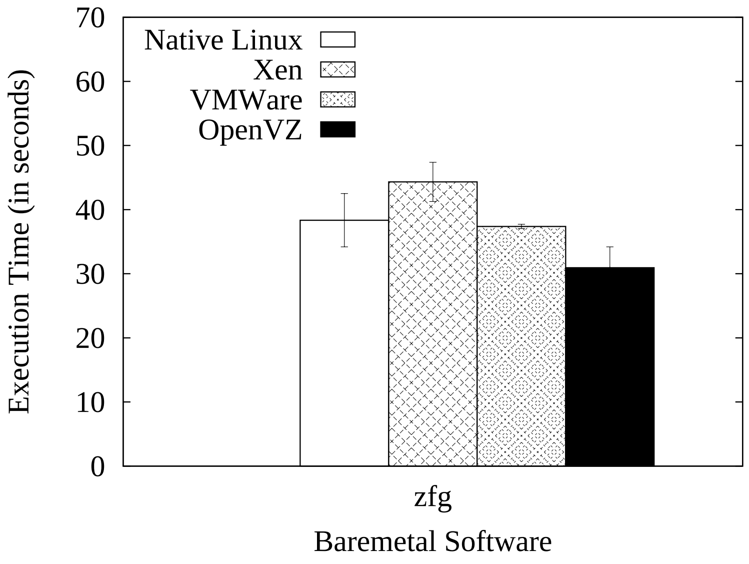

Figure 6.4: ZFG test. The mean execution time for a round of the test is shown in the vertical axis, smaller values are better. The virtualization tools and native Linux are spread along the horizontal axis.

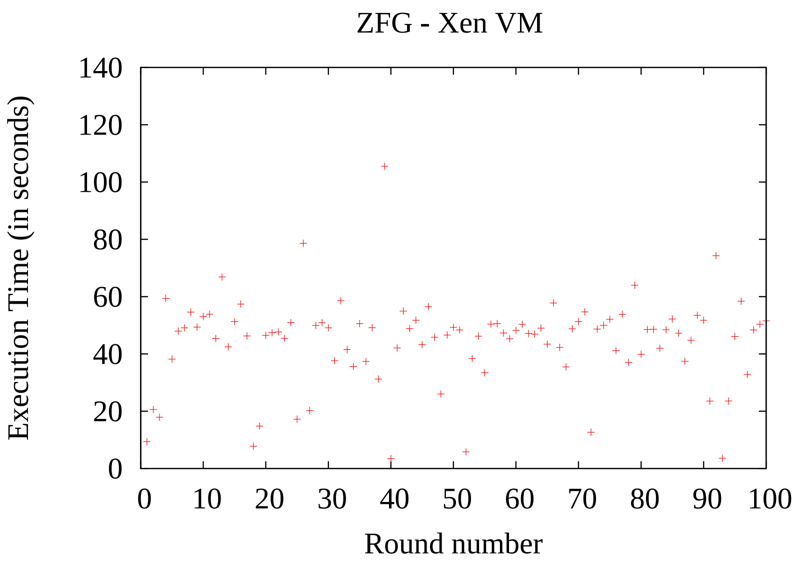

Figure 6.5: ZFG test Xen VM rounds. It is noticeable the presence of several rounds that took less than 10 seconds to execute and a few that took more than 60 seconds to execute.

As observed in figure 6.4, VMware VM performed a little faster than native Linux and we credit this to the buffer that transfers data between the virtual disk and the real disk, since it gives VM the impression that data has already been written to disk when it is actually being transferred from the buffer to the physical disk. OpenVZ unexpectedly outperformed native Linux and further investigation must be performed to discover the reason. Xen VM performed slower than native Linux but an irregular execution time was observed. Looking at figure 6.5, which shows all the rounds durations for Xen VM tests, we can also observe that there are several rounds in which Xen obtains a very high writing speed, completing a round in ten seconds or less. In native Linux, when there is data to be written in an I/O device, the data is sent to a memory buffer and then it is transferred to the device using DMA. In Xen and VMware the extra step in which the virtual I/O device driver writes the data to a memory buffer used to communicate with the real I/O device driver gives the impression that data was written in the hard drive when it is actually in the buffer from the VM do the real I/O device driver. Further details on this buffer for the Xen implementation can be found in [14]. The great round execution time difference presented by Xen results corroborate our hypothesis since the time needed to each round would vary according to the state of the buffer in the beginning of each round. To confirm our suspicion, we made further tests of two kinds. The first kind of tests is for handicapping the VMs buffer. In this kind of tests, we modified the test scripts to write large amounts of data before each round of the test in order to dirty the buffer from the VM to the real I/O device driver. In this kind of tests it is expected that the buffer from the VM to the real IO driver is filled with pending requests at the beginning of each round. The second kind of tests is for favoring the VMs buffer. In this kind of tests, we modified the script to wait a long period before each round, giving Dom 0 enough time to clear the buffer. In this kind of tests it is expected that the buffer from the VM to the real IO driver is empty, that is, without pending requests, at the beginning of each round. The results of the additional test are in subsections 6.1.5 and 6.1.6.

6.1.5 ZFG with buffer dirtying

The results for the modified ZFG test in which we dirty the buffer with large amounts of data before each round is shown in figure 6.6.

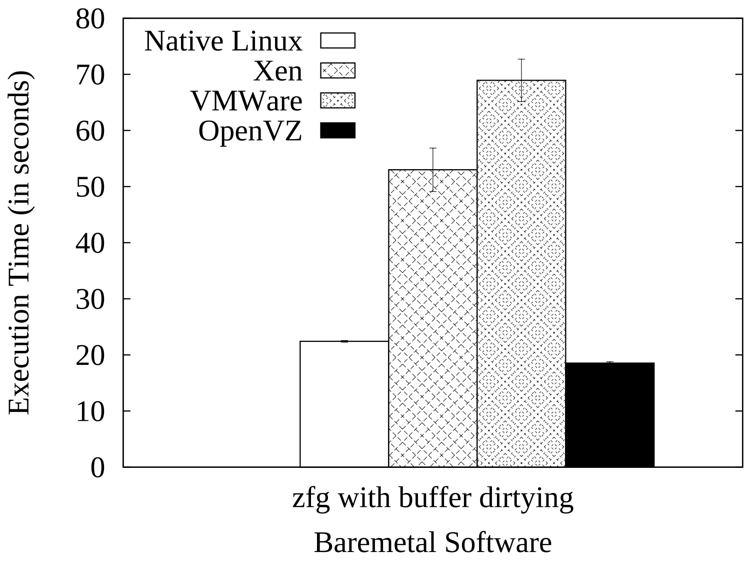

Figure 6.6: ZFG with buffer dirtying test. The mean execution time for a round of the test is shown in the vertical axis, smaller values are better. The virtualization tools and native Linux are spread along the horizontal axis.

The test was made with 100 rounds and, as expected, when we dirty the buffer that transfers data between the virtual and real I/O device driver, the VMs performance drops significantly and native Linux outperforms with great advantage both Xen and VMware virtualized Linuxes. Once again OpenVZ outperforms native Linux, indicating that the disk mechanism adopted by OpenVZ is not similar to the one adopted by Xen and VMware VMs.

6.1.6 ZFG with resting period

The results for the modified ZFG test in which we wait Dom0 to clear the buffers before each round are shown in figure 6.7.

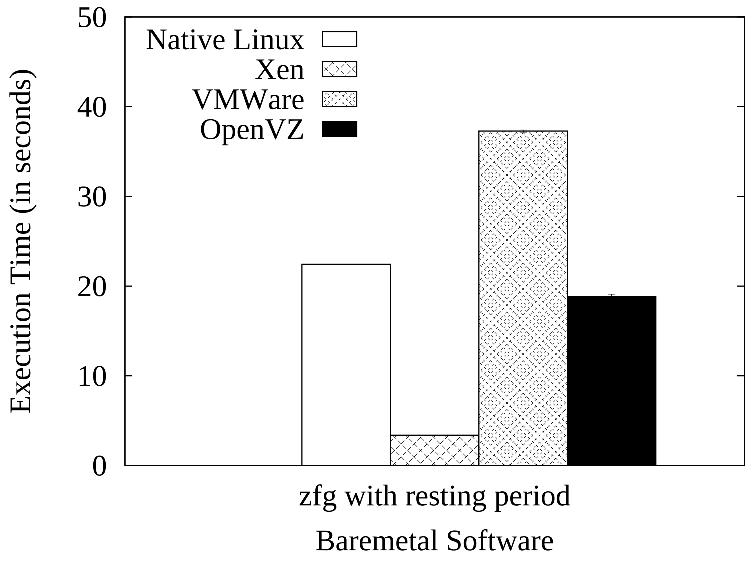

Figure 6.7: ZFG with resting period test. The mean execution time for a round of the test is shown in the vertical axis, smaller values are better. The virtualization tools and native Linux are spread along the horizontal axis.

As expected, when we wait a long period of time between the rounds, Xen has enough time to write all the pending data that is in the buffer and when the next round of the test is executed the buffer is clear, giving the impression that Xen VM writes data to the disk extremely fast. As we suspected, when the amount of written data is not much bigger than the buffer, Xen VM has such a great performance because it is writing on the RAM memory buffer that connects the virtual disk driver and the real disk driver outperforming native Linux since it is actually writing data to the hard drive. We allowed two minutes between each round for Xen to clear the buffers. VMware did not perform well and we attribute this to the VM scheduler. OpenVZ once more outperformed Linux as in the previous ZFG disk writing tests.

6.1.7 Iperf: UDP data flow from VM to Network Tests Computer

In this test UDP packet generation from the VM is being benchmarked. Results for packet reception are shown in the next test. The results for the Iperf network benchmark generating traffic from the VM to the Network Tests Computer for 100 rounds are shown in figure 6.8.

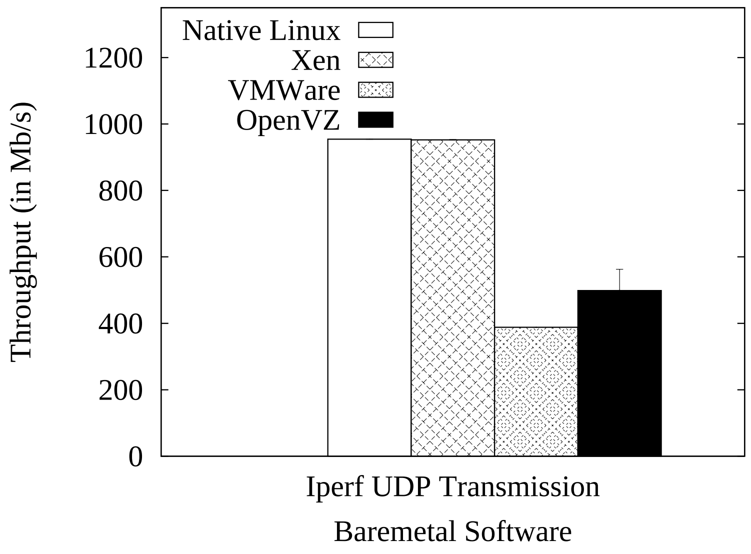

Figure 6.8: Iperf UDP packet generation test. The mean throughput achieved is shown in the vertical axis, greater values are better. The virtualization tools and native Linux are spread along the horizontal axis.

As expected, native Linux reaches the best results, near the theoretical limit of the used Gigabit Ethernet Network Adapter. Xen achieves near native performance showing that the I/O ring mechanism is capable of delivering data in a fast enough rate to use all the Gigabit interface. VMware presents great performance degradation which we credit to inefficient implementation of the default networking mechanism. OpenVZ default networking mechanism based on IP layer virtualization performed poorly and could not use the full Gigabit interface bandwidth. This is a very important result since the forwarding capability of a router directly depends on its capability of sending packets, and the results show that it is possible for a virtualized environment to send large packets at near native speed.

6.1.8 Iperf: UDP data flow from network tests computer to VM

In this test UDP packet reception from the VM is being benchmarked. The results for the Iperf network benchmark generating traffic from the Network Tests Computer to the VM for 100 rounds are shown in figure 6.9.

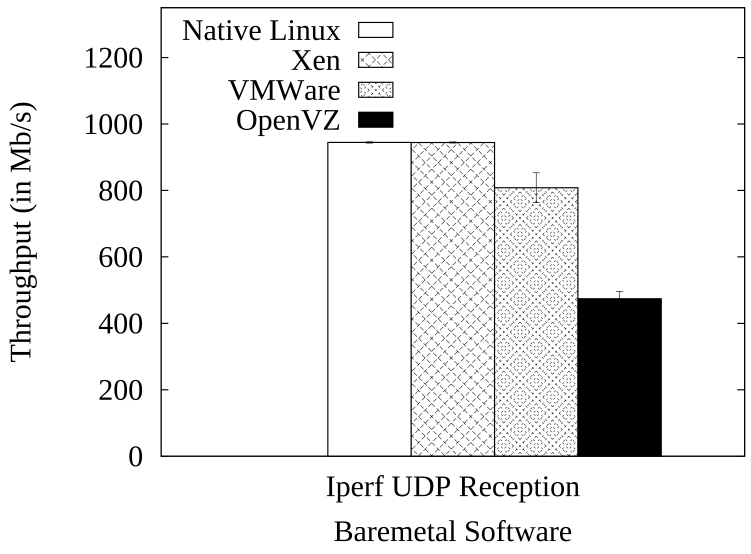

Figure 6.9: Iperf UDP packet reception test. The mean throughput achieved is shown in the vertical axis, greater values are better. The virtualization tools and native Linux are spread along the horizontal axis.

The results show once again that native Linux reaches near theoretical results. Xen achieves near native performance once more. VMware performed much better in reception than in transmission but still worse than native Linux and Xen. OpenVZ default networking virtualization mechanism performed poorly. This is also a very important result since the forwarding capability of a router is directly related to its capability of receiving packets. The results show that Xen can handle large packets reception in near native speed with a single VM, multiple-VM tests in the next section will confirm Xen capability.

6.1.9 Linux Kernel Compilation

The results for the Linux Kernel Compilation test executed for 10 rounds are shown in figure 6.10.

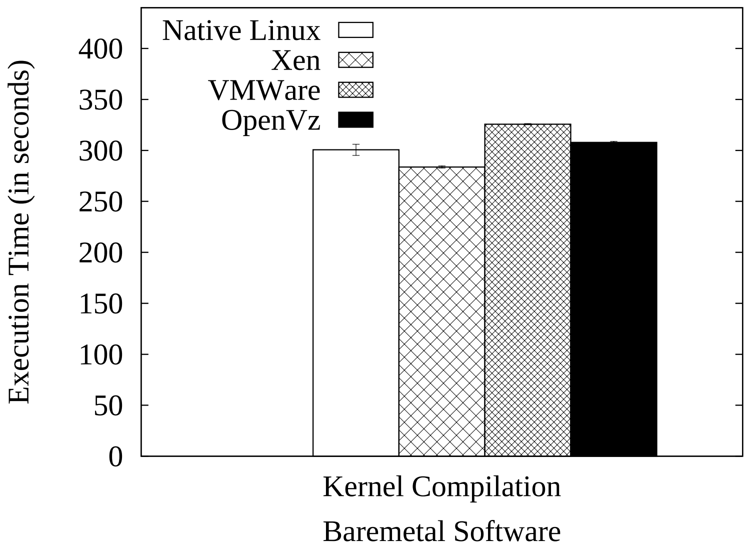

Figure 6.10: Kernel Compilation test. The mean execution time for a round of the test is shown in the vertical axis, smaller values are better. The virtualization tools and native Linux are spread along the horizontal axis.

Once again Xen achieves a better result than native Linux. We relate this slightly better result to the disk writing effect explained in the previous disk writing tests. VMware achieves the worst results for this task which we credit to the poor CPU and memory performance presented in the previous tests. OpenVZ performance was similar to native Linux as expected by the performance presented in the previous CPU, memory and bonnie++ disk tests.

6.1.10 Conclusions

An important point to comment is the disk performance of VMs. The buffer that communicates the real disk driver and the virtual disk driver can have both a positive or negative effect depending on the application. For applications which make numerous reading disk accesses, the extra stage between the hard drive and the VM disk driver introduces longer delays for recovering information, therefore decreasing performance. If writing to disk accesses are most frequent than reading and the application does not write amounts of data sufficiently large to fill the buffer, then the application should benefit from the buffer as ZFG had when executed with an interval between the rounds. Applications that would lose performance include website hosts for sites that have to frequently recover information from the disk or data mining applications that intensively access disk reading information. Applications that would benefit from the buffer include logging applications and other applications that mostly access disk for writing data. Considering the presented results, specially the CPU, memory access and network related, we conclude that, at first, virtualization tools introduce acceptable overhead and that building virtual routers over the tested tools could be feasible. Based on the CPU, memory and network results, which are the main used resources in router normal operation we conclude that VMware is not a good option for usage in Horizon project since it had the worst performance results and also present no flexibility for customization since its source code is proprietary and it is not publicly available for modifications. OpenVZ performed very well in most of the tests, but presented poor network performance which is a major issue for network virtualization. In addition to poor network performance, OpenVZ also restricts too much the virtual environments with the limitation of using, not only the same OS, but also the same kernel version for all the virtual environments which is a major issue for building virtual networks with different levels of routers complexity. For presenting good CPU and memory performance, having near native network performance, being open source allowing customization with no restrictions and for providing virtual environments with a high degree of flexibility, we chose Xen as the virtualization tool to be adopted in Horizon project. In the next section the second kind of tests are presented, the Multiple-VM tests, in which we will evaluate how Xen scales with multiple VMs sharing the same computer resources, giving more complete results and allowing us to conclude if we are able to build multiple virtual networks over a computer.

6.2 Multiple-VM tests

We present the results of the executed benchmarks for the multiple-VM tests we conducted. The objective of these tests is to observe how Xen scales with multiple VMs regarding performance degradation, justice in scarce resource sharing and how different vCPU to CPU allocation schemes affects overall and individual performance. All the following graphs regarding multiple-VM tests will show the performance of Xen VMs varying the number of VMs performing different tasks and varying the CPU allocation scheme for the same task. Xen allows configuring how the vCPU presented from the VMs to the virtualized OSes share the physical CPUs. It allows both dynamic CPU allocation schemes and static allocation schemes. For the multiple-VM tests three different CPU allocation schemes were used in order to study its influence in the scalability of the virtualized environments. As previously mentioned, the Tests Computer is equipped with two Xeon E5400 quad-core CPUs which gives eight physical cores to distribute between the hypervisor and VMs needs. Another relevant point is that each CPU has its own Front Side Bus, allowing them to access memory independently. Inside each CPU, each core has its own L1 and L2 cache, but L3 cache is shared for each 2 cores. The first core allocation scheme used in the tests is Xen Default scheme, in this scheme, the hypervisor dynamically allocates the CPU physical cores to the Xen vCPUs based on its default mechanism. In this scheme there can be unused physical cores, cores being allocated for more than one vCPU and cores being allocated to exactly one vCPU. An example of the core allocations at a certain moment is shown in figure 6.11.

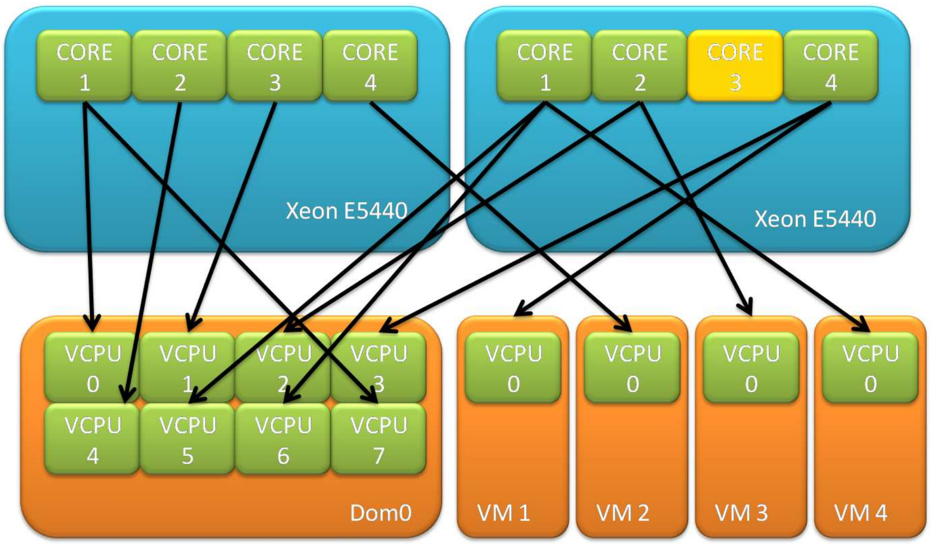

Figure 6.11: Example of the physical cores allocation using Xen Default scheme. At a certain moment there can be unused physical cores, cores allocated to more than one vCPU and cores allocated to a single vCPU.

The second scheme we used is a static physical cores allocation scheme in which we set a fixed exclusive core for each VM vCPU and a fixed exclusive core for four of domain 0 vCPUs, disabling 4 of its vCPUs. The objective of this scheme is to observe if there is a considerable gain of performance when fixing which physical core will handle a vCPU request, especially for leading to a reduced number of CPU cache faults, decreasing RAM memory accesses which have much greater latency than cache. The scheme is shown in figure 6.12.

The third scheme we used is a static physical cores allocation scheme in

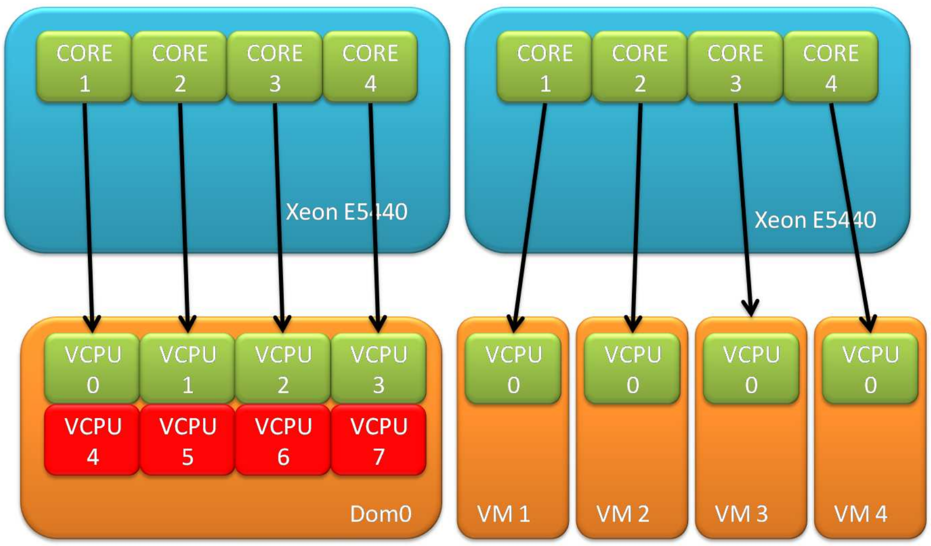

Figure 6.12: Second physical cores allocation scheme. Each VM has a fixed exclusive physical core. Dom 0 has a fixed exclusive physical core for 4 of its vCPUs and the rest of them are disabled.

which we set all the vCPUs to use the same physical core. The objective of this scheme is to overload the physical CPU, seeing how performance decreases and observing if resource sharing between the VMs continues fair. The scheme is shown in figure 6.13.

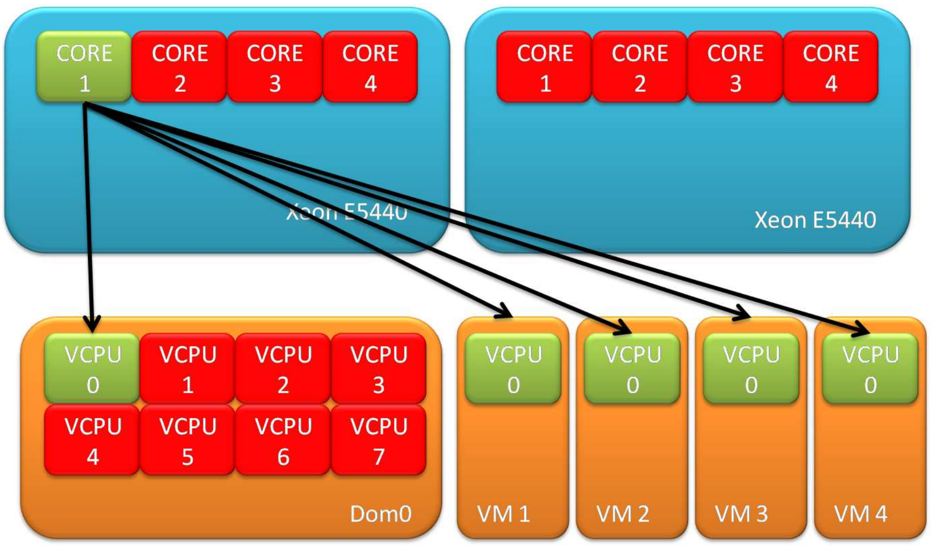

Figure 6.13: Third physical cores allocation scheme. Each VM and Dom0 uses the same fixed physical core. Dom0 has only once active vCPU, all the other seven are disabled.

6.2.1 Super-Pi

The results for the scalability tests using the Super-Pi benchmark with three different physical cores allocation schemes executed for 10 rounds are shown in figures 6.14, 6.15 and 6.16.

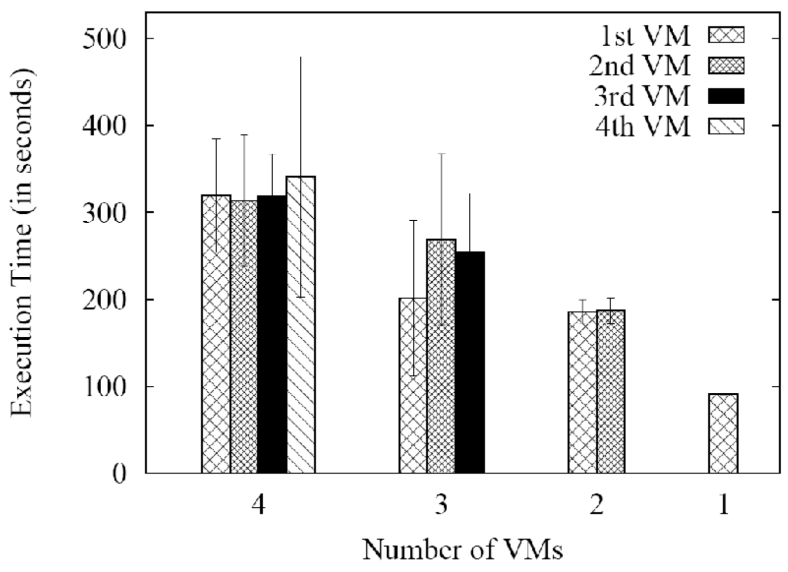

Figure 6.14: Super-Pi benchmark with multiple-VM using one physical core for all the vCPUs.

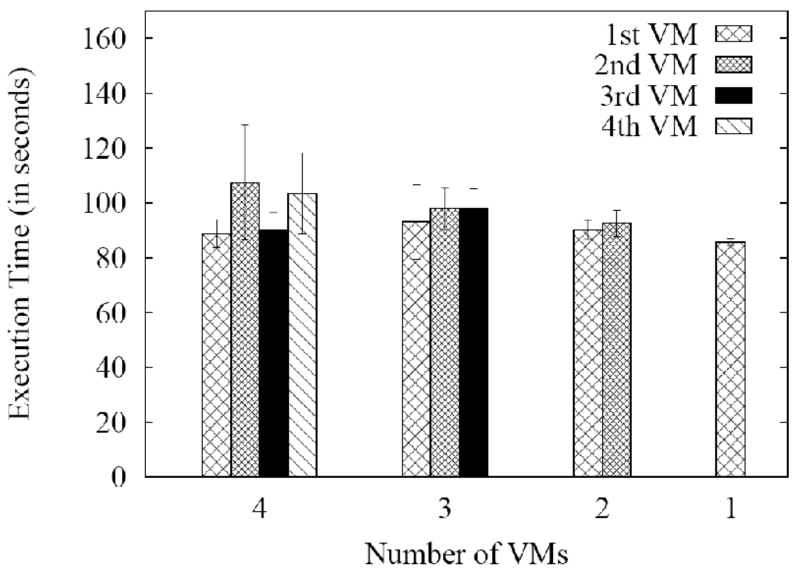

Figure 6.15: Super-Pi benchmark with multiple-VM using one exclusive physical core for each VM vCPU and for each Dom0 vCPU.

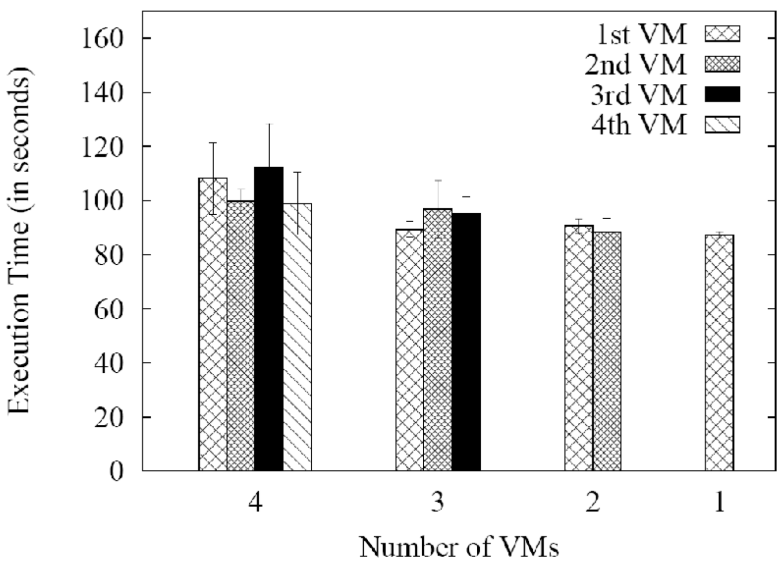

Figure 6.16: Super-PI benchmark with multiple-VM using Xen Default physical core allocation scheme.

The results show that all the three CPU schemes present much greater variance as the number of VMs is increased showing that for this task Xen had difficulty to schedule the VMs in a fair way. For the scheme with one physical core serving all the vCPUs, it is also noticeable that there was the greatest observed variance and, since CPU was a scarce resource, the more VMs there were, the more a round of the Super-Pi benchmark would take to be executed.

6.2.2 MASR

The results for the scalability tests using the MASR with three different physical cores allocation schemes executed for 10 rounds are shown in figures 6.17, 6.18 and 6.19.

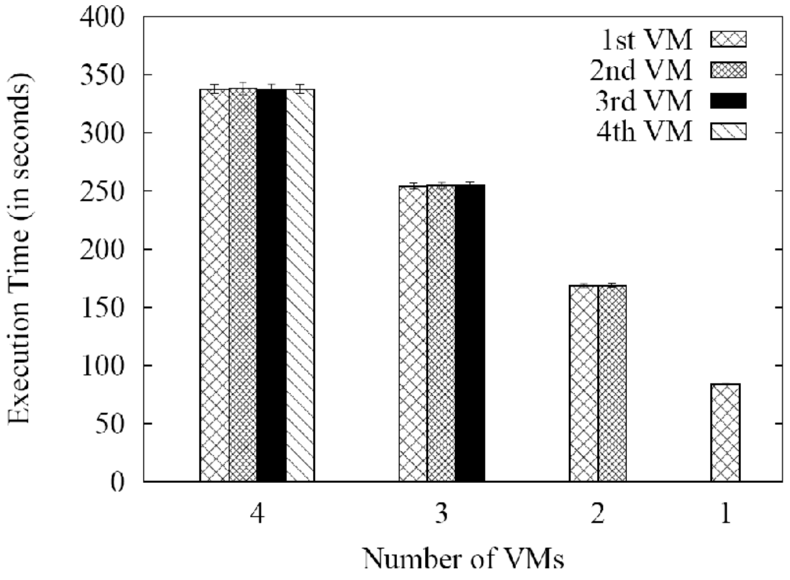

Figure 6.17: MASR benchmark with multiple-VM using one physical core for all the vCPUs.

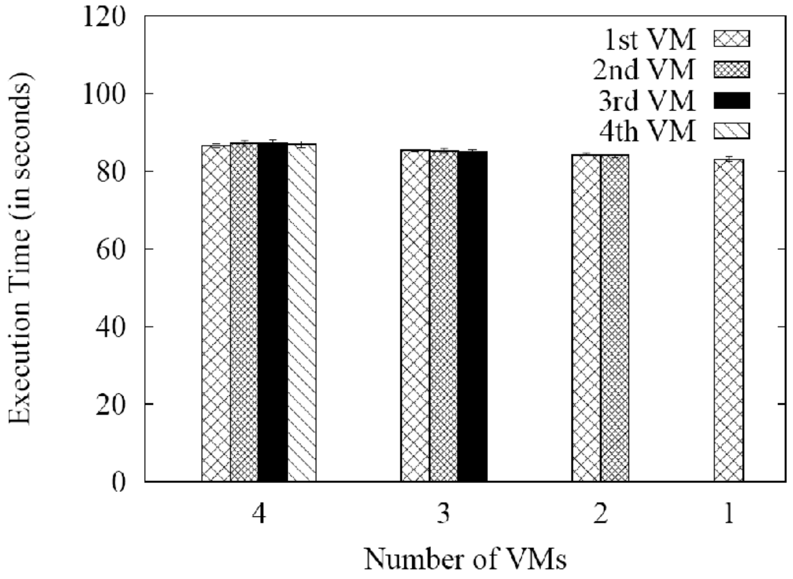

Figure 6.18: MASR benchmark with multiple-VM using one exclusive physical core for each VM vCPU and for each Dom0 vCPU.

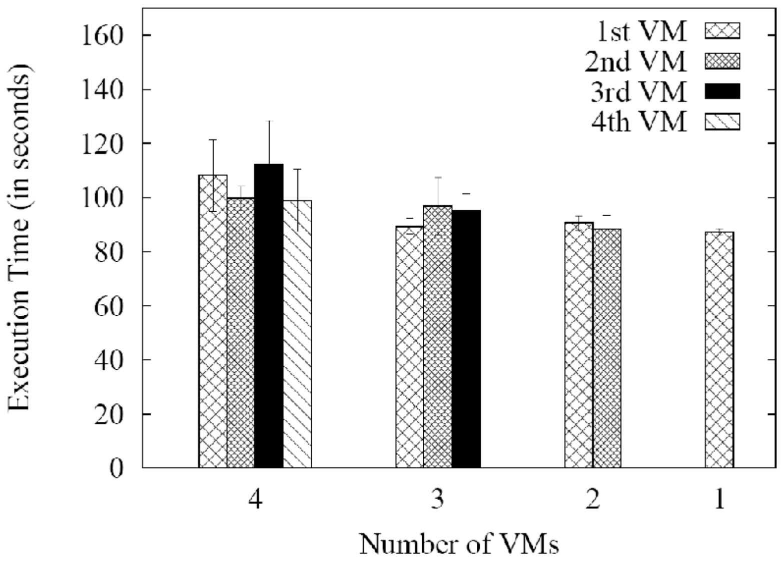

Figure 6.19: MASR benchmark with multiple-VM using Xen Default physical core allocation scheme.

The results show that both Xen scheme with one exclusive physical core for each VM and for each Dom0 vCPU scaled well from one to four VMs, showing no degradation of performance as the VM number was increased. Xen default scheme has not lost much performance when scaling from one to four VMs, but presented worse performance then the scheme with one exclusive physical core for each VM and for each active Dom0 vCPU. We credit this loss of performance to the fact that Xen default scheme reassigns vCPUs over the physical cores dynamically which increases cache faults when a vCPU is reassigned to a different core. The scheme using one physical core for all the VMs and Dom0 has shown that for this test CPU was a bottleneck and that Xen Hypervisor was fair regarding the division of the CPU among the VMs.

6.2.3 ZFG with buffer dirtying

Since from the Single-VM tests we discovered that Xen VMs do not actually write data to the hard drive synchronously, we chose to execute only the disk test that was as close as possible from writing to the hard drive in a synchronous manner which would be the test in which we dirty the buffer from the VM to the real hard drive driver in the beginning of each round. The results for the scalability tests using the ZFG with buffer dirtying benchmark with three different physical cores allocation schemes executed for 10 rounds are shown in figures 6.20, 6.21 and 6.22.

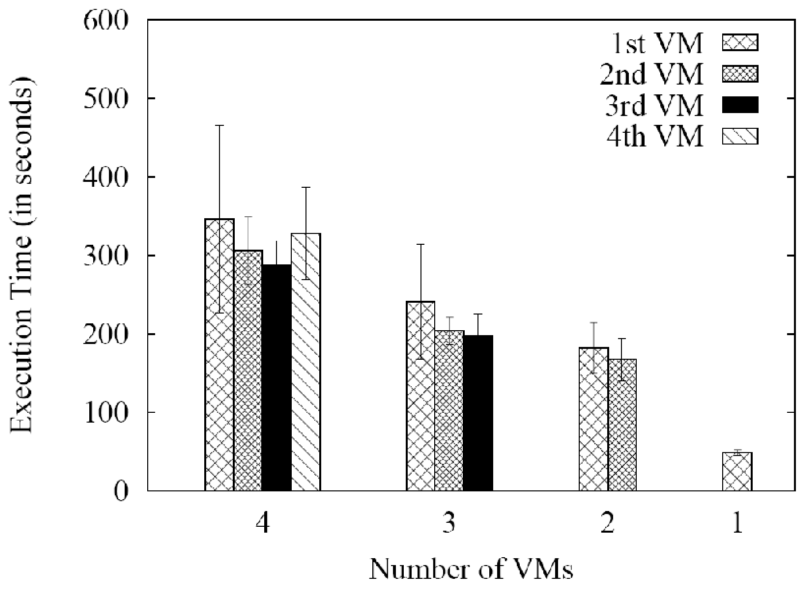

Figure 6.20: ZFG with buffer dirtying benchmark with multiple-VM using one physical core for all the vCPUs.

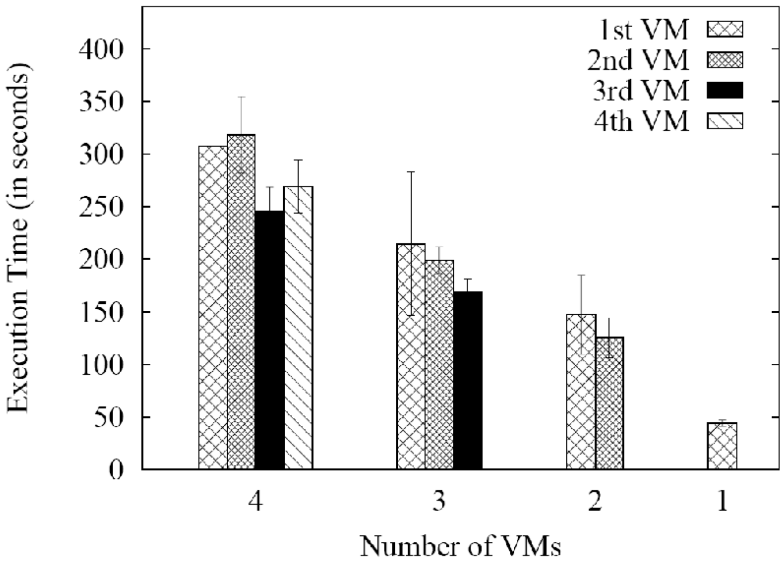

Figure 6.21: ZFG with buffer dirtying benchmark with multiple-VM using one exclusive physical core for each VM vCPU and for each Dom0 vCPU.

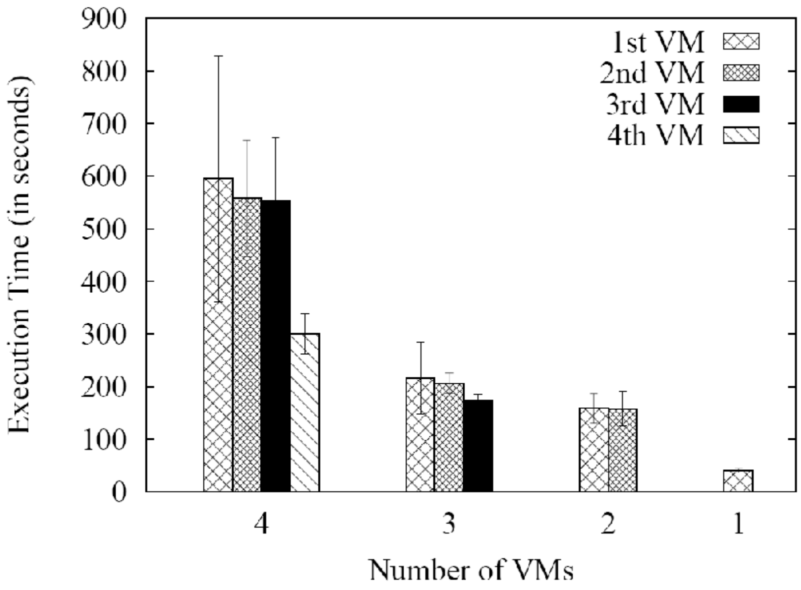

Figure 6.22: ZFG with buffer dirtying benchmark with multiple-VM using Xen Default physical core allocation scheme.

The results show that all the three CPU schemes present much greater variance as the number of VMs is increased showing that for this task Xen had difficulty to schedule the VMs in a fair way. The scheme that presented better performance was the one with one exclusive physical core for each vCPU. The scheme with one physical core serving all the vCPUs presented small overhead when compared to the one exclusive physical core for each vCPU scheme, which can be explained by the fact that in this task the bottleneck is the hard drive even with only one core for all the vCPUs. Surprisingly, Xen Default scheme presented low performance when the task was executed with four VMs showing instability. Further investigation is required to understand why this instability was observed.

6.2.4 Iperf: UDP data flow from VM to Network Tests Computer

The results for the scalability tests using the Iperf benchmark generating UDP data flow from VM to the Network Tests Computer with three different physical cores allocation schemes executed for 10 rounds are shown in figures 6.23, 6.24 and 6.25.

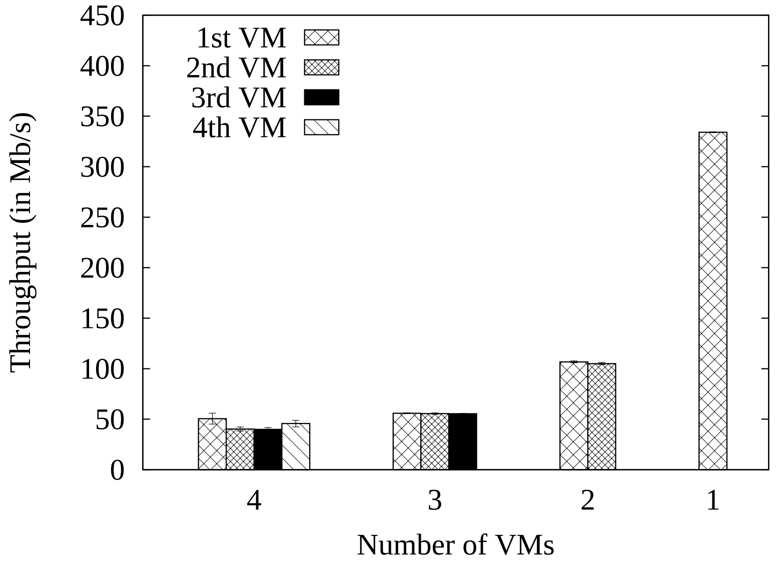

Figure 6.23: Iperf: UDP data flow from VM to Network Tests Computer with multiple-VM using one physical core for all the vCPUs.

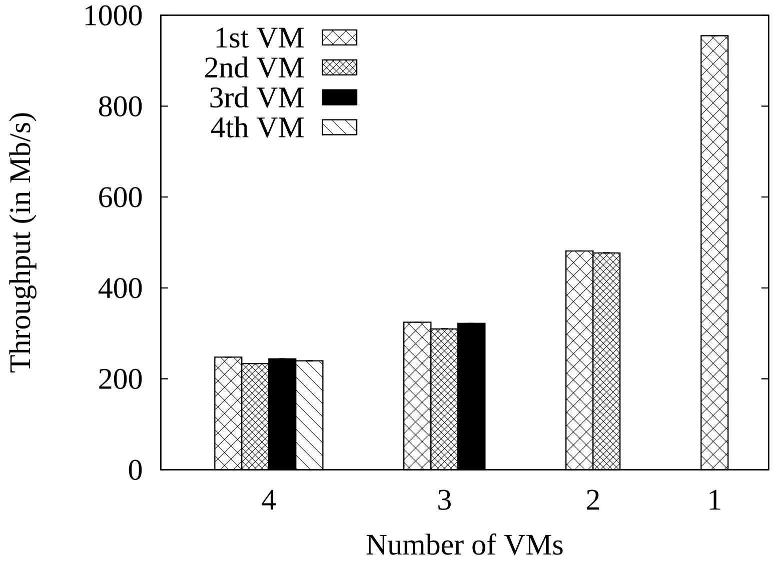

Figure 6.24: Iperf: UDP data flow from VM to Network Tests Computer with multiple-VM using one exclusive physical core for each VM vCPU and for each Dom0 vCPU.

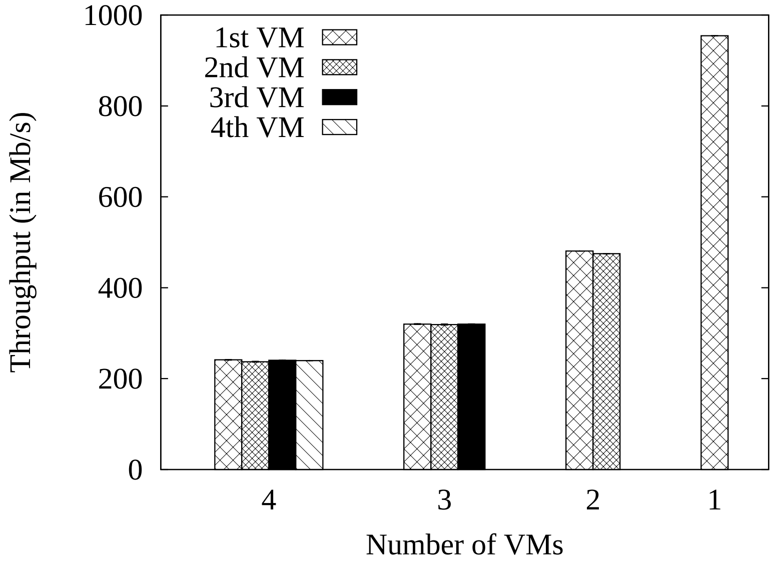

Figure 6.25: Iperf: UDP data flow from VM to Network Tests Computer with multiple-VM using Xen Default physical core allocation scheme.

The results show that both Xen Default physical core allocation scheme and one exclusive physical core for each VM vCPU scheme had similar results achieving total throughput near the theoretical limit of the network adapter. For the one physical core for all the vCPUs scheme, the bottleneck was not the network adapter but the CPU since the VMs were not able to generate enough packets to fill the network adapter capacity. In all schemes Xen has proven to be fair in the resource sharing.

6.2.5 Linux Kernel Compilation

The results for the scalability tests using the Linux Kernel Compilation with three different physical cores allocation schemes executed for 10 rounds are shown in figures 6.26, 6.27 and 6.28.

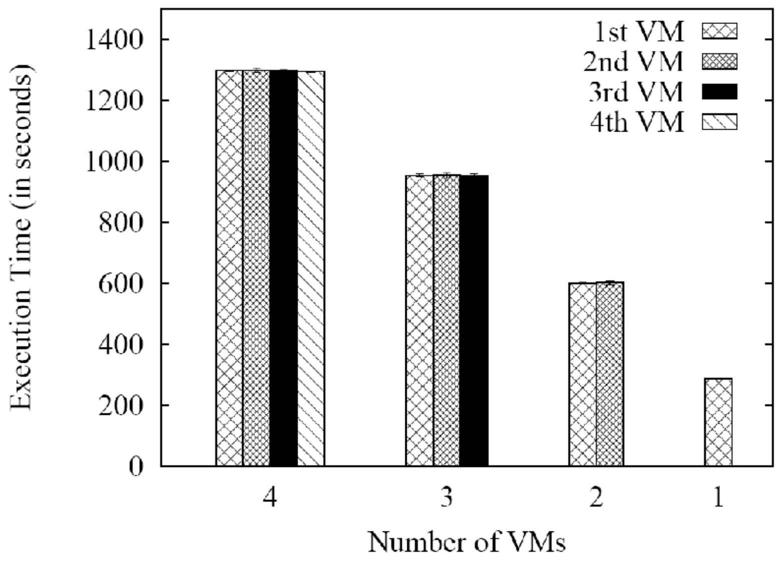

Figure 6.26: Linux Kernel Compilation with multiple-VM using one physical core for all the vCPUs.

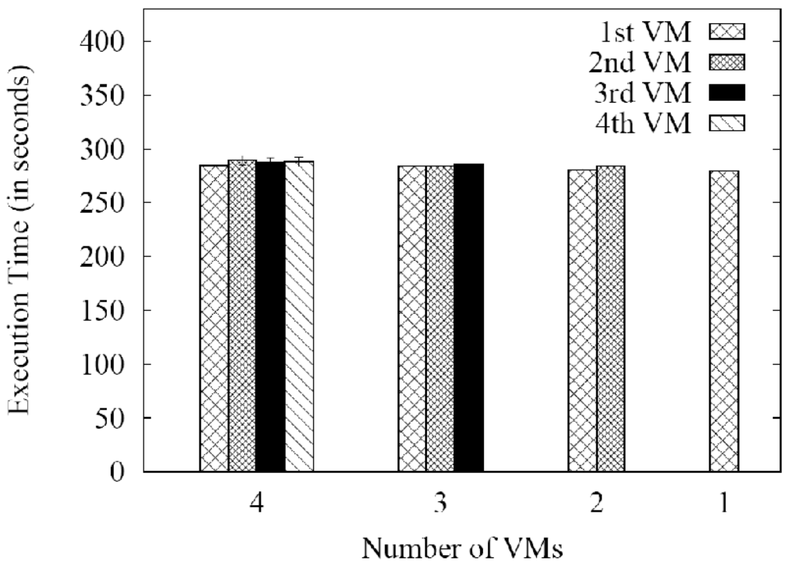

Figure 6.27: Linux Kernel Compilation with multiple-VM using one exclusive physical core for each VM vCPU and for each Dom0 vCPU.

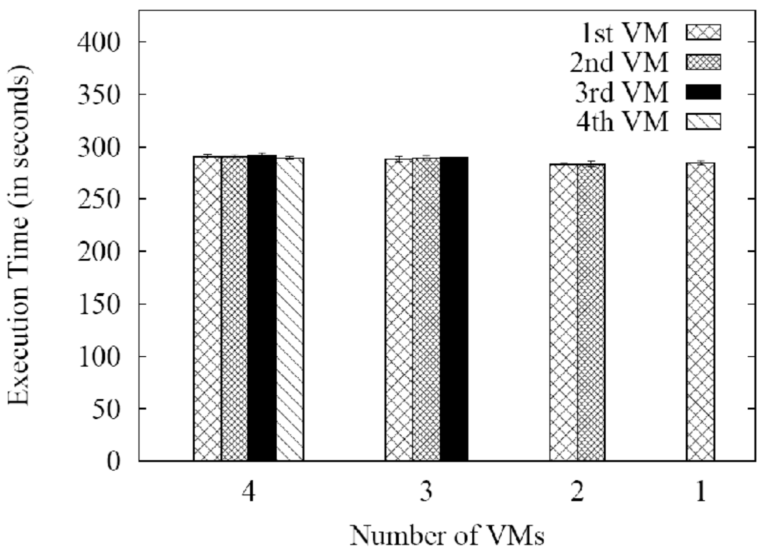

Figure 6.28: Linux Kernel Compilation with multiple-VM using Xen Default physical core allocation scheme.

The results show that both Xen Default scheme and the scheme with one exclusive physical core for each VM and for each Dom0 vCPU scaled well from one to four VMs, showing no degradation of performance. The scheme using one physical core for all the VMs and Dom0 has shown that for this test CPU was a bottleneck and that Xen Hypervisor was fair regarding the division of the CPU among the VMs.

6.2.6 Conclusions

In general, Xen has proven to be fair when scaling to multiple VMs. For some tests, scaling multiple VMs has affected fairness, which was shown by the increasing variance in the results as the number of VMs would increase. It is also noticeable that except for the hard drive test, concentrating all the vCPUs in one physical core has considerably decreased VMs performance. Except for the memory tests, there were no noticeable differences between the performance of Xen Default scheme and the one physical core for each VM vCPU scheme showing that CPU cache preservation was not a major issue for the tasks we executed. An unexpected result that would need extra investigation is Xen Default scheme poor performance in the hard drive test when 4 VMs were executing the test.

Chapter 7

Conclusions

Due to the “ossification” of the Internet described in chapter 1 the scientific community decided that a new Internet should be developed. The efforts to propose a new Internet architecture can be classified in two groups: (i) the purists, in which the proposed architecture is composed of a single network sufficiently general to support any service requirement, and (ii) the pluralists, in which the proposed architecture is composed of multiple networks, each one built to cope to a kind of service requirements. Horizon project has the objective of proposing a new Internet architecture using the pluralist approach. For supporting multiple networks with independent network stacks, we propose using network virtualization.

In this work we compared VMware, Xen and OpenVZ as candidates for achieving network virtualization in Horizon project architecture. In chapter 2, we presented virtualization concepts, challenges and techniques used by the virtualization tools we considered. In chapter 3, we described the inner workings of the virtualization tools we tested. Chapter 4 presented the tests scenario and the methodology we have used. Chapter 5 presented the tests we conducted and the results we obtained in single-VM and multiple-VM tests.

As discussed in section 6.1.10, we conclude that VMware is not a good option for Horizon project, because it achieved the worst results in the tests and is a proprietary solution that does not allow modifications since there is no access to its source code. We also concluded that, although OpenVZ had great performance results in most of the tests, it is also not the best choice for Horizon project since it achieved low network performance, which is one of the main concerns for building virtual routers, and also provides low flexibility for the construction of virtual routers, because all the virtual environments must share not only the same OS but also the same kernel. Xen presented itself as the best virtualization tool for our project since it offers the flexibility we need by allowing any OS inside the VMs and for allowing customizations, since it is an open-source tool. Another positive point is that Xen obtained satisfactory performance virtualizing CPU, memory and network, which are the most important resources for virtual routers forwarding operation. As discussed in section 6.2.6 Xen provides fair resource sharing between the VMs and has scaled from one to four VMs with no performance degradation caused by the hypervisor.

Bibliography

[1] A. Feldmann, “Internet clean-slate design: what and why?,” *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 3, pp. 59–64, July 2007.

[2] T. Spyropoulos, S. Fdida, and S. Kirkpatrick, “Future Internet: Fundamentals and measurement,” *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 101–106, Apr. 2007.

[3] L. H. M. K. C. e. O. C. M. B. D. Marcelo D. D. Moreira, Natalia C. Fernandes, “Internet do Futuro: Um Novo Horizonte,” *Minicursos do Simpósio Brasileiro de Redes de Computadores - SBRC’2009*, pp. 1–59, 2009.

[4] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, “Internet indirection infrastructure,” *IEEE/ACM Trans. Netw.*, 2004.

[5] J. He, R. Zhang-Shen, Y. Li, C.-Y. Lee, J. Rexford, and M. Chiang, “Davinci: Dynamically adaptive virtual networks for a customized internet,” in *CoNext*, 2008.

[6] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, “Towards high performance virtual routers on commodity hardware,” *ACM CoNEXT*, Dec. 2008.

[7] N. Feamster, L. Gao, and J. Rexford, “How to lease the Internet in your spare time,” *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 61–64, Jan. 2007.

[8] D. L. Tennenhouse and D. J. Wetherall, “Towards an active network architecture,” *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 2, no. 2, pp. 5–17, 1996.

[9] R. Braden, T. Faber, and M. Handley, “From protocol stack to protocol heap: role-based architecture,” *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, no. 1, pp. 17–22, 2003.

[10] T. Anderson, L. Peterson, S. Shenker, and J. Turner, “Overcoming the Internet impasse through virtualization,” *IEEE Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

[11] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield, “Plutarch: an Argument for Network Pluralism,” in *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, pp. 258–266, Sept. 2003.

[12] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Commun. ACM*, vol. 17, no. 7, no. 7, pp. 412–421, 1974.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles - SOSP03*, Oct. 2003.

[14] D. Chisnall, *The Definitive Guide To The Xen Hypervisor*. Prentice Hall, 2008.

[15] VMWare Inc, *The Architecture of VMware ESX Server 3i*, 2007.

[16] K. Kolyshkin, *Virtualization in Linux*, 2006.

[17] VMWare Inc, *Timekeeping in VMware Virtual Machines*, 2008.

[18] XenSource, Inc., *A Performance Comparison of Commercial Hypervisors*, 2007.

[19] VMWare Inc, *A Performance Comparison of Hypervisors*, 2007.

[20] K. Adams and O. Agesen, “A comparison of software and hardware techniques for x86 virtualization,” in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[21] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley, “Evaluating Xen for router virtualization,” in *International Conference on Computer Communications and Networks - ICCCN*, pp. 1256–1261, Aug. 2007.

[22] A. Menon, A. L. Cox, and W. Zwaenepoel, “Optimizing network virtualization in Xen,” in *USENIX Annual Technical Conference*, pp. 15–28, May 2006.

[23] VMWare Inc, *VMware ESX Server 2 Architecture and Performance Implications*, 2005.

[24] SWsoft Inc, *OpenVZ User's Guide*, 2005.

[25] W. C., C. C., M. J., S. S., and K.-H. G., "Linux security modules: General security support for the linux kernel," *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.