DITEN

Dipartimento di Ingegneria Navale, Elettrica, Elettronica e delle Telecomunicazioni

Università degli Studi di Genova

Corso di Dottorato di Ricerca in Scienze e Tecnologie per l'Informazione e la Conoscenza

XXVII ciclo

Ingegneria Elettronica, Informatica, della Robotica e delle Telecomunicazioni

Ph.D. Thesis

Submitted by Danilo Tigano

March, 2018

DITEN, Universita' di Genova

danilo.tigano@unige.it

Title: An innovative approach to performance metrics calculus in cloud computing environments: a guest-to-host oriented perspective.

Advisor: Professor Giulio Barabino

DITEN, Universita' di Genova

giulio.barabino@unige.it

Abstract

In virtualized systems, the task of profiling and resource monitoring is not straight-forward. Many datacenters perform CPU overcommittment using hypervisors, running multiple virtual machines on a single computer where the total number of virtual CPUs exceeds the total number of physical CPUs available.

From a customer point of view, it could be indeed interesting to know if the purchased service levels are effectively respected by the cloud provider.

The innovative approach to performance profiling described in this work is based on the use of virtual performance counters, only recently made available by some hypervisors to their virtual machines, to implement guest-wide profiling.

Although it isn't possible for the virtual machine to access Virtual Machine Monitor, with this method it is able to gather interesting informations to deduce the state of resource overcommittment of the virtualization host where it is executed.

Tests have been carried out inside the compute nodes of FIWARE Genoa Node, an instance of a widely distributed federated community cloud, based on OpenStack and KVM. AgiLab-DITEN, the laboratory I belonged to and where I conducted my studies, together with TnT-Lab–DITEN and CNIT-GE-Unit designed, installed and configured the whole Genoa Node, that was hosted on DITEN-UniGE equipment rooms.

All the software measuring instruments, operating systems and programs used in this research are publicly available and free, and can be easily installed in a micro instance of virtual machine, rapidly deployable also in public clouds.

To my beloved Mom and Dad, living forever in my memories

Acknowledgments

I wish to thank:
Giulio Barabino, master of science and life, as well as my friend, without whom this long journey would never have begun;
Mario Marchese, for his constant and careful encouragement to complete my doctorate;
Giorgio Robino and Giancarlo Portomauro, for their friendship, effective help and cooperation in realizing the Fiware Genoa Node.

Table of Contents

1 Introduction

Virtualization is not a new technology. In the 1960s computing systems were as large as a room and very expensive to operate. In those days only one application could be executed on one piece of hardware at a particular time. Then time-sharing had been introduced to execute several applications simultaneously. One major drawback of this approach was the lack of isolation of the running applications. If application A caused a hardware error all other running applications were affected.

To isolate these, virtualization provided several isolated environments to ran them into [1].

In the 1970s hardware architectures became virtualization aware. IBM mainframes allowed the administrators to partition the real hardware and provide isolated environments for each application.

In the 1980s, as the x86 architecture arose and the prices of hardware felt, it became affordable to run one computer per application. Also operating systems supported multi tasking and there was no need for time-sharing any more. As a consequence virtualization became history.

In the last couple of years virtualization experienced a comeback. Intel and AMD extended the IA32 instructon set of x86 processors to support virtualization. Since these are the big players on the CPU market, nearly any recent PC and server supports virtualization.

Today, virtualization is mainly used for *consolidation*: an interesting statistic reported by the U.S. Environmental Protection Agency (EPA) stood out. The EPA

study on server and data center energy efficiency found that only around 5% of server capacity was actually used. The rest of the time, the server was dormant. Virtualizing platforms on a single server can improve server utilization, but the benefits of reducing server count are a force multiplier. With reduced servers comes reduced real estate, power consumption, cooling (less energy costs), and management costs. Less hardware also means improved reliability. All in all, virtualization brings not only technical advantages but cost and energy advantages, as well.

There are many types of consolidation and the following examples should give a basic idea about it.

A lot of servers are running at a very low load but still consuming a huge amount of energy. Server consolidation means workload optimization of these servers by running each of them as a Virtual Machine (VM) on virtualization hosts. When contention is low these VMs are dynamically migrated to fewer virtualization hosts and shut down the others to reduce energy consumption and lower costs. If the load increases and more hosts are needed to fulfill server level objectives, these are started again and some VMs are migrated onto them.

Another example is application consolidation, where virtualization is used to replace the old hardware of a legacy system. It helps to provide an environment which mimics the old hardware and runs the legacy system.

Sandboxing is another purpose of virtualization. It is mainly used to increase security by running potentially insecure applications inside a VM. So an application runs in its isolated environment, while specialists can observe its behaviour. Thus malware and other malicious software could be found before it's deployed on a machine with access to the network of a company.

There are various techniques to provide and operate VMs, one of those are Virtual Machine Monitors (VMM). Such a VMM represents a software layer of indirection, running on top of the hardware. It operates all VMs running upon it.

2 System Virtualization

2.1 Definition

Since virtualization is a settled topic, there are several definitions on it. The following is a general definition of virtualization given by Chiueh and Brook [2]:

> "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"

This means, that virtualization uses techniques to abstract from the real hardware and provides isolated environments, so called Virtual Machines. These are capable to run various applications or even a whole operating system. A goal not mentioned in the definition is to have nearly to native performance for running VMs.
This is a very important point, because the users always want to get the most out

of their hardware. Most of them are not willing to introduce virtualization technology, if a huge amount of CPU power is wasted by managing Vms.
As well as virtualization in general, system virtualization is well defined too:

> "A system VM provides a complete environment in which an operating system and many processes, possibly belonging to multiple users, can coexist." [3]

The complete environment, in this case, means an environment that provides usual hardware like ethernet controllers, CPUs or hard disk drives to an operating system (OS) which runs inside of it. A server with real hardware attached to it commonly runs several VM's. Such a server is called virtualization *host* and the VM's running on top of it are called *guests*. The OS that runs inside a guest is called guest OS.
For this work, by virtualization we mean system virtualization.

2.2 Virtual Machine Monitor

Virtualization, in the context of this work, is the process of hiding the underlying physical hardware in a way that makes it transparently usable and shareable by multiple operating systems. This architecture is also known as platform virtualization other than system virtualization. In a typical layered architecture, the layer that provides for the system virtualization is called the hypervisor, also known as virtual machine monitor, or VMM. Each instance of a guest operating

system is called a virtual machine (VM), because to these VMs the hardware is virtualized to appear as dedicated to them. A simple illustration of this layered architecture is shown in Figure 1.

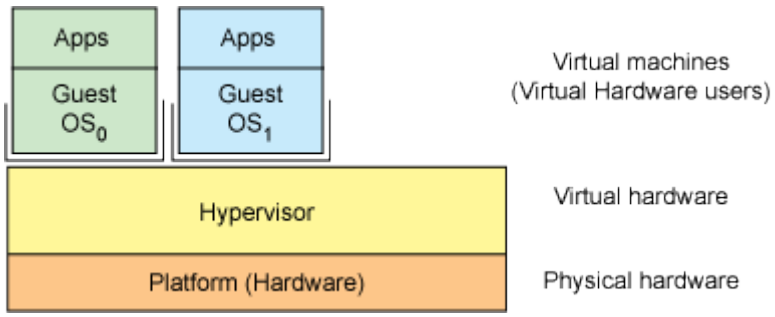

Figure 1. Simple layered architecture showing the virtualization of common hardware

Hypervisors do for operating systems what operating systems roughly do for processes. They provide isolated virtual hardware platforms for execution that in turn provide the illusion of full access to the underlying machine.

Operating systems virtualize access to the underlying resources of the machine to processes. Hypervisors do the same thing, but instead of processes, they accomplish this task for entire guest operating systems.

2.2.1 Hypervisor classifications

Hypervisors can be classified into two distinct types. The first, *type 1 hypervisors*, are those that natively run on the bare-metal hardware. The second, *type 2*, are hypervisors that execute in the context of another operating system (that runs on the bare metal). Examples of type 1 hypervisors include Kernel-

based Virtual Machine (KVM), Vmware ESXi, MS Hyper-V, Xen.

Examples of type 2 hypervisors include QEMU, WINE, Virtual Box, Vmware Workstation and Player.

So a hypervisor (regardless of the type) is just a layered application that abstracts the machine hardware from its guests. In this way, each guest sees a VM instead of the real hardware.

All guests are controlled and monitored by the VMM. It provides tools to the users to manage them. These tools allow to do several operations like starting or stopping a guest or migrating VMs between hosts.

At a high level, the hypervisor requires a small number of items to boot a guest operating system: a kernel image to boot, a configuration, such as IP addresses and quantity of memory to use, a disk, and a network device. The disk and network device commonly map into the machine's physical disk and network device, as shown in Figure 2. Finally, a set of guest tools is necessary to launch a guest and subsequently manage it.

A VM usually has at least one virtual CPU. The VMM maps the virtual CPU(s) of all actually running VMs to the physical CPU(s) of the host. Hence, there are usually more VMs running on a host than physical CPUs are attached to it, causes the need of some kind of scheduling. Therefore a VMM uses a scheduling mechanism to assign a certain share of the physical CPUs to each virtual CPU.

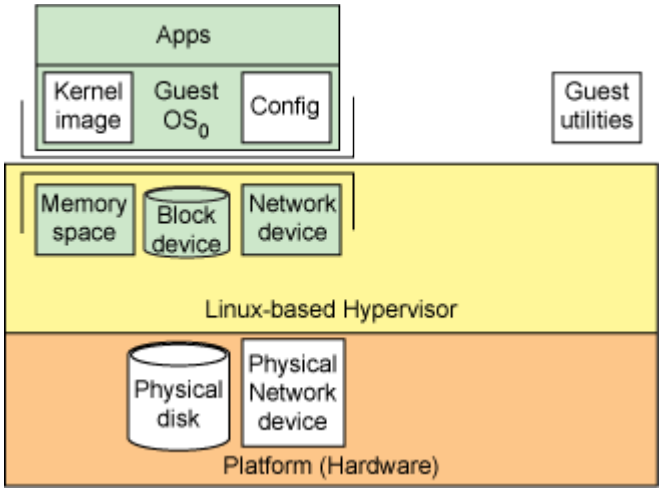

Figure 2. Minimal mapping of resources in a hypothetical hypervisor

A VMM has to deal with memory management, also. It maps an amount of physical memory into the VMs address space and also has to handle fragmentation of memory and swapping. Since some VMs need more memory than others, the amount of assigned memory is defined and often dynamically adjusted by using the management tools.

Usually, the VMs don't have access to the physical hardware and don't even know about it either. Only if direct access is desired, devices may be passed through directly. For running legacy software this may be a point. But in more common scenarios the VMM provides virtual I/O devices like network cards, hard disks and cd drives. Since a VMM provides different VMs mostly with same hardware, it is much easier to migrate them between hosts running the same VMM. The drivers for the virtual I/O devices need to be installed only once in this case.

2.2.2 Hardware Support

To implement a Virtual Machine Monitor on a x86 architecture, hardware assistance is needed. The privilege levels implemented by the CPU to restrict tasks that processes can do, are one aspect. Another one is the memory management that is emulated by the VMM which tends to be inefficient. Hardware support could lead to an increased performance of the virtual machines by supporting a VMM.

2.2.3 Privilege Levels

The most modern operating systems don't allow applications to execute certain operations. Only the OS may load drivers or access the hardware directly, for example. To restrict all running applications to only a subset of the resources, the OS and the CPU conspire using privilege levels.

As described in [4] a x86 CPU runs in a specific privileged level at any given time.

Figure 3 shows these levels as rings. Ring 0 is the most privileged and ring 3 is the least privileged.

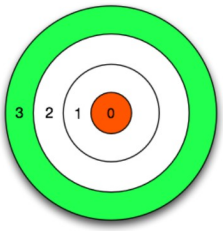

Figure 3. CPU privilege levels

The resources that are protected through the rings are: memory, I/O ports and CPU instructions. The operating system typically runs in ring 0. It needs the most privileged level to do resource management and provide access to the hardware. All the applications run in ring 3. Ring 1 and 2 are widely unused. From a OSs point of view ring 0 is called kernel-mode and ring 3 user-mode.
As mentioned in section 2.2.1 the VMM needs to access the memory, CPU and I/O devices of the host. Since only code running in ring 0 is allowed to perform these operations, it needs to run in the most privileged ring, next to the kernel.
An operating system installed in a VM also expects to access all the resources and in order of that running in ring 0 like the VMM does. Due to the fact that only one kernel can run in ring 0 at the same time, the guest OSs have to run in another ring with less privileges or have to be modified to run in user-mode.
Intel and AMD realized that this is a major challenge of virtualization on the x86 architecture. So they introduced Intel VT and AMD SVM as an extension of the IA-32 instruction set for better support of virtualization. These extensions allow the VMM to run a guest OS that expects to run in kernel-mode, in a lower privileged ring.

2.2.4 Memory Management

In order to run several VMs on top of a server, a multiple of the amount of memory that is attached to a common server is needed. Since each VM runs an entire operating system and applications on that, it is recommended to assign as much memory to a VM as a comparable physical machine would have. The

VMM splits the physical memory of the host into contiguous blocks of fixed length and maps it into the address space provided to a VM.
Most modern systems are using virtual memory management. This technique allows to provide the previously mentioned contiguous blocks of memory to a VM, although it is fragmented all over the physical memory or even partially stored on the hard disk. In this case it has to be copied back to memory by the virtual memory management first, when accessed. Since a VM is unaware of the physical address of its address space, it can't figure out whether parts of its virtual memory has to be copied or not. To achieve that, the VMM holds a so called shadow page table that stores the physical location of the virtual memory of all VMs. Thus, any time a VM writes to its memory, the operation has to be intercepted to keep the shadow pages up to date. When a swapped address is accessed the VMM first uses the virtual memory management to restore it.
With the introduction of Intel's Extended Paging Tables (EPT) and AMD's Nested Paging Tables (NPT) a VMM can use hardware support for the translation between virtual and physical memory. This reduces the overhead of holding shadow pages and increases the performance of a VMM [5].

2.3 Virtualization techniques

I introduce now two techniques to realize system virtualization:
paravirtualization and *full virtualization*.

2.3.1 Paravirtualization

The paravirtualization approach allows each guest to run a full operating system. But these do not run in ring 0. Due to that all the privileged instructions can't be executed by a guest. In order of that, modifications to the guest operating systems are required to implement an interface. This is used by the VMM to take over control and handle the restricted instructions for the VM. The paravirtualization approach promises nearly to native performance but lacks in the support for closed source operating systems [6]. To apply the mentioned modifications, the source code of the kernel of an operating system has to be patched. Thus, running Microsoft Windows in a VM is impossible using paravirtualization.

2.3.2 Full virtualization

This approach allows to operate several operating systems on top of a hosting system, each running into its own isolated VM. The VMM uses hardware support as described in section 2.3.1 to operate these, which allows to run the guest operating systems without modifications. The VMM provides I/O devices for each VM, which is commonly done by emulating older hardware. This ensures that a guest OS has driver support for these devices. Because of the emulated parts fullvirtualization is not as fast as paravirtualization. But if one needs to run a closed source OSs, it is the only viable technique to do so.

2.4 Linux as an hypervisor

One of the most important modern innovations of Linux is its transformation into a hypervisor or, in other terms, an operating system for hosted operating systems. A number of hypervisor solutions have appeared that use Linux as the core, but in this document I will introduce KVM, that is one of the emerging hypervisor technologies, completely open source and used to realize the experiments in our laboratory, for the purposes of my PhD research work.

2.4.1 Generic linux-based hypervisor

A simplified hypervisor architecture then implements the glue that allows a guest operating system to be run concurrently with the host operating system. This functionality requires a few specific elements, shown in Figure 4.
First, similar to system calls that bridge user-space applications with kernel functions, a hypercall layer is commonly available that allows guests to make requests of the host operating system. Input/output (I/O) can be virtualized in the kernel or assisted by code in the guest operating system. Interrupts must be handled uniquely by the hypervisor to deal with real interrupts or to route interrupts for virtual devices to the guest operating system. The hypervisor must also handle traps or exceptions that occur within a guest: after all, a fault in a guest should halt the guest but not the hypervisor or other guests. A core element of the hypervisor is a page mapper, which points the hardware to the pages for the particular operating system (guest or hypervisor). Finally, a high-level

scheduler is necessary to transfer control between the hypervisor and guest operating systems and back.

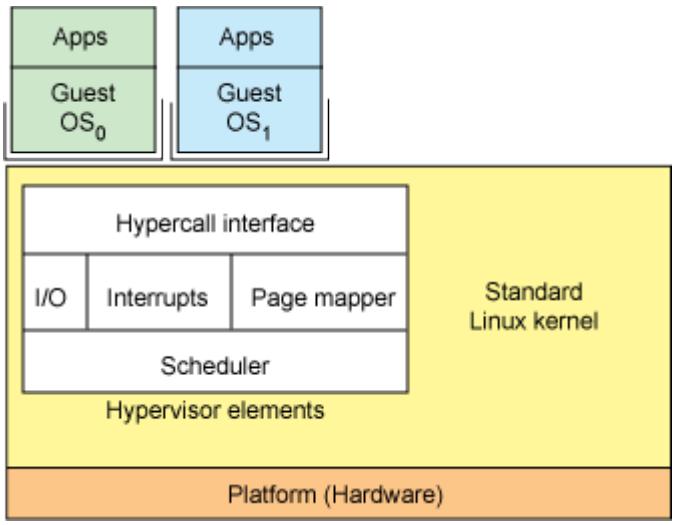

Figure 4. Simplified view of a Linux-based hypervisor

2.4.2 KVM Kernel-based Virtual Machine

KVM has been initially developed by Qumranet, a small company located in Israel. Redhat acquired Qumranet in september 2008, when KVM became more production ready. They see KVM as the next generation of virtualization technology. Nowadays it is used as the default VMM in Redhat Enterprise Linux (RHEL) since version 5.4 and the Redhat Enterprise Virtualization for Servers. Qumranet released the code of KVM to the open source community. Today, well known companies like IBM, Intel and AMD count to the list of contributors of the project. Since version 2.6.20 KVM is part of the vanilla linux kernel and thus

available on the most linux-based operating systems with a newer kernel.
Further more it benefits from the world class development of the open source operating system, because if linux gains better performance through new algorithms, drivers or whatsoever KVM also performs better.
KVM is a system virtualization solution that uses full virtualization to run Vms [7]. It has a small code base, since it was designed to leverage the facilities provided by hardware support for virtualization. KVM runs mainly on the x86 architecture, but IA64 and PowerPC support was added.
Additionally, KVM has added support for symmetrical multiprocessing (SMP) hosts and guests, and supports enterprise-level features such as live migration, to allow guest operating systems to migrate between physical servers.
KVM is implemented as a kernel module, allowing Linux to become a hypervisor simply by loading a module. KVM provides full virtualization on hardware platforms that provide hypervisor instruction support, such as the Intel® Virtualization Technology [Intel VT] or AMD Virtualization [AMD-V] offerings. KVM also supports paravirtualized guests, including Linux and Windows®.
This technology is implemented as two components. The first is the KVM-loadable module that, when installed in the Linux kernel, provides management of the virtualization hardware, exposing its capabilities through the /proc file system (see Figure 5). The second component provides for PC platform emulation, which is provided by a modified version of QEMU. QEMU executes as a user-space process, coordinating with the kernel for guest operating system requests.

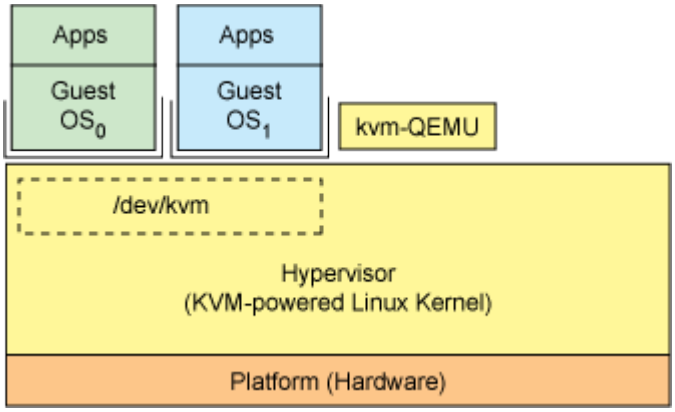

Figure 5. High-level view of the KVM hypervisor

2.4.3 KVM architecture

Linux has all the mechanisms a VMM needs to operate several VMs. So the developers didn't reinvent the wheel and added only few components to support virtualization. KVM is implemented as a kernel module that can be loaded to extend linux by these capabilities.

In a normal linux environment each process runs either in *user-mode* or in *kernel-mode*. KVM introduces a third mode, the *guest-mode*. Therefore it relies on a virtualization capable CPU with either Intel VT or AMD SVM extensions.

A process in guest-mode has its own kernel-mode and user-mode. Thus, it is able to run an operating system. Such processes are representing the VMs running on a KVM host. In [8] the author states what the modes are used for from a hosts point of view:

• user-mode: I/O when guest needs to access devices

• kernel-mode: switch into guest-mode and handle exits due to I/O operations

• guest-mode: execute guest code, which is the guest OS except I/O

2.4.4 Resource management

The KVM developers aimed to reuse as much code as possible. Due to that they mainly modified the linux memory management, to allow mapping physical memory into the VMs address space. Therefore they added shadow page tables, that were needed in the early days of x86 virtualization, when Intel and AMD had not released EPT respectively NPT yet. On May 2008 support for these technologies has been introduced.

In modern operating systems there are many more processes than CPUs available to run them. The scheduler of an operating system computes an order in that each process is assigned to one of the available CPUs. In this way, all running processes are share the computing time. Since the KVM developers wanted to reuse most of the mechanisms of linux, they simply implemented each VM as a process, relying on its scheduler to assign computing power to the VMs.

2.4.5 The KVM control interface

Once the KVM kernel module has been loaded, the */dev/kvm* device node appears in the filesystem. This is a special device node that represents the interface of

KVM. It allows to control the hypervisor through a set of ioctls. These are commonly used in certain operating systems as an interface for processes running in user-mode to communicate with a driver. The ioctl() system call allows to execute several operations to create new virtual machines, assign memory to a virtual machine, assign and start virtual CPUs.

2.4.6 Emulation of hardware

To provide hardware like hard disks, cd drives or network cards to the Vms, KVM uses a highly modified QEMU. This is a so called platform virtualization tool, which allows to emulate an entire pc platform including graphics, networking, disk drives and many more. For each VM a QEMU process is started in user-mode and certain emulated devices are virtually attached to these. When a VM performs I/O operations, these are intercepted by KVM and redirected to the QEMU process regarding to the guest.

2.4.7 Execution Model

Figure 6 depicts the execution model of KVM. This is a loop of actions used to operate the VMs. These actions are separated by the three modes we mentioned earlier in section 3.1.1.

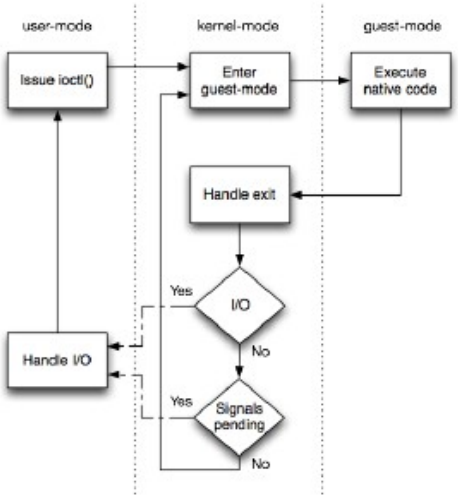

Figure 6. KVM execution model

In [9] Kivity et al. described the KVM execution model and stated which tasks are done in which mode:

• *user-mode*: The KVM module is called using ioclt() to execute guest code until I/O operations initiated by the guest or an external event occurs. Such an event may be the arrival of a network package, which could be the reply of a network package sent by the host earlier. Such events are expressed as signals that leads to an interruption of guest code execution.

• *kernel-mode*: The kernel causes the hardware to execute guest code natively.

If the processor exits the guest due to pending memory or I/O operations, the kernel performs the necessary tasks and resumes the flow of execution. If external events such as signals or I/O operations initiated by the guest

exists, it exits to the user-mode.

• *guest-mode*: This is on the hardware level, where the extended instruction set of a virtualization capable CPU is used to execute the native code, until an instruction is called that needs assistance by KVM, a fault or an external interrupt.

While a VM runs, there are plenty of switches between these modes. From kernel-mode to guest-mode switches and viceversa are very fast, because there is only native code that is executed on the underlying hardware. When I/O operations occur and the flow of execution switches to the user-mode, emulation of the virtual I/O devices comes into play. Thus, a lot of I/O exits and switches to user-mode are expected. Imagine an emulated hard disk and a guest reading certain blocks from it. Then QEMU emulates the operations by simulating the behaviour of the hard disk and the controller it is connected to. To perform the guests read operation, it reads the corresponding blocks from a large file and returns the data to the guest. Thus, user-mode emulated I/O tends to be a bottleneck which slows down the execution of a VM.

2.4.8 Paravirtual device drivers

With the support for the virtio [10] paravirtual device model, KVM addresses the performance limitations by using QEMU emulated devices. Virtio is common framework to write VMM independent drivers promising bare-metal speed for these, since paravirtual devices attached to a VM are not emulated any more.

Instead, a backend for the paravirtual drivers is used to perform I/O operations either directly or through a user-mode backend. KVM uses QEMU as such a backend which handles I/O operations directly. Thus, the overhead to mimic the behaviour of a IDE hard disk is tremendously decreased to simply using kernel drivers for performing certain operations and responding.

2.4.9 Linux hypervisor benefits

Developing hypervisors using Linux as the core has real, tangible benefits. Most obviously, basing a hypervisor on Linux benefits from the steady progression of Linux and the large amount of work that goes into it. From the typical optimizations and bug fixes, scheduling, and memory-management innovations to support for different processor architectures, Linux is a platform that continues to advance.

KVM proved not long ago that through the addition of a kernel module, one could transform the Linux kernel into a hypervisor. KVM operates in the context of Linux as the host but supports a large number of guest operating systems, given underlying hardware virtualization support, such as Linux itself, MS Windows (almost all versions), FreeBSD, Apple Mac OS X.

Another intriguing benefit of using Linux as the platform is that you can take advantage of that platform as an operating system in addition to a hypervisor. Therefore, in addition to running multiple guest operating systems on a Linux hypervisor, you can run your other traditional applications at that level. So instead of worrying about a new platform with new application programming

interfaces (APIs), you have your standard Linux platform for application development. The standard communication protocols and other useful applications are available alongside the guests.

3 Cloud Computing

Cloud computing is actually nothing more than the provisioning of computing resources (computers and storage) as a service. Beside that comes the flexibility to dynamically scale the service to further computers and storage in an easy and clear approach [11]. All this is similar to the ideas behind *utility computing,* in which computing resources were viewed as a metered service, as is the case for more traditional utilities (such as electricity or water). What's different is not the goal behind these ideas but the existing technologies that have come together to make them a reality.

One of the most important ideas behind cloud computing is scalability, and the key technology that makes that possible is virtualization. Virtualization allows better use of a server by aggregating multiple operating systems and applications on a single shared computer. Virtualization also permits online migration so that if a server becomes overloaded, an instance of an operating system (and its applications) can be migrated to a new, less cluttered server.

From an external view, cloud computing is simply the migration of computing and storage outside an enterprise and into the cloud. The user defines the resource requirements (such as computing and wide area network, or WAN, bandwidth needs), and the cloud provider virtually assembles these components within its infrastructure.

But why would you willingly relinquish control over your resources and allow them to virtually exist in the cloud? There are many reasons, but two that I

believe are most important are cost and scalability. The goal of cloud computing is to make these resources less expensive than what you can provide for and manage yourself. Along with this reduction in cost comes greater flexibility and scaling. A cloud computing provider can easily scale your virtual environment for greater bandwidth or computing resources with the provider's virtual infrastructure.

The green advantage to cloud computing is the ability to virtualize and share resources among different applications for better server utilization. Figure 7 shows an example. Here, three independent platforms existed for different applications, each running on its own server. In the cloud, servers can be shared (virtualized) for operating systems and applications to better use the servers, resulting in fewer servers. Fewer servers means less required space (minimizing the data center footprint) and less power for cooling (minimizing the carbon footprint).

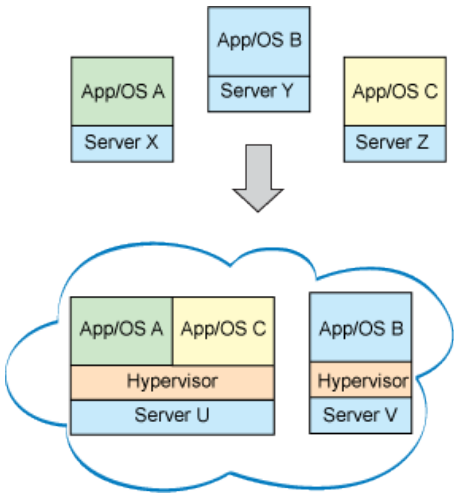

Figure 7. Virtualization and resource use

3.1 Anatomy of cloud computing

As you peer inside the cloud, you find that it's actually not just a single service but a collection of services, as shown in Figure 8. These layers define the level of service provided.

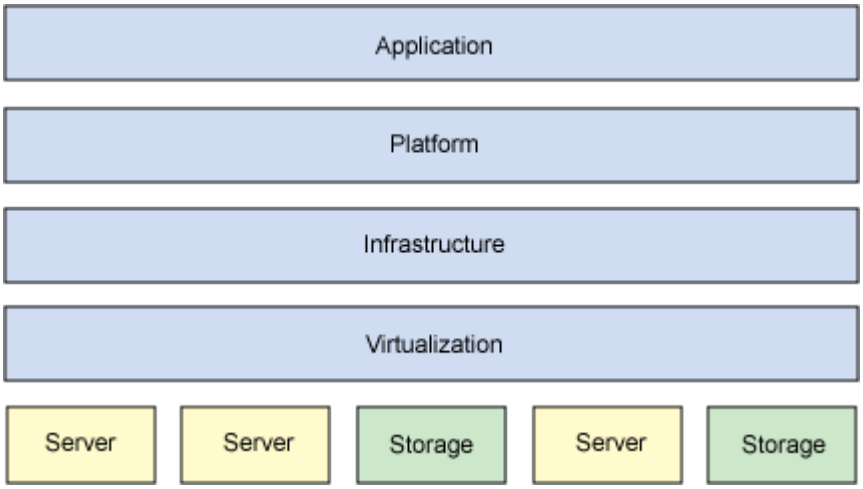

Figure 8. The layers of cloud computing

Let's start at the lowest level of service provided, which is the infrastructure (*Infrastructure-as-a-Service,* or IaaS). IaaS is the leasing of an infrastructure (computing resources and storage) as a service. This means not only virtualized computers with guaranteed processing power but reserved bandwidth for storage and Internet access. In essence, it's the capability of leasing a computer or data center with specific quality-of-service constraints that has the ability to execute an arbitrary operating system and software.

Besides reducing the management cost associated with cloud computing

resources, there are other advantages. For example, when you separate yourself from your resources by the Internet, it doesn't really matter where those resources reside. They could be, for example, in a climate that offers ambient (natural) cooling and therefore minimizes energy usage.

Moving up the stack, the next level of service is the platform (*Platform-as-a-Service,* or PaaS). PaaS is similar to IaaS but includes operating systems and required services that focus on a particular application. For example, a PaaS in addition to virtualized servers and storage provides a particular operating system and application set along with access to necessary services such as a MySQL database or other, specialized local resources. In other words, PaaS is IaaS with a custom software stack for the given application.

Finally, at the top of Figure 8 is the simplest service that can be provided: the application. This layer is called *Software-as-a-Service* (SaaS), and it is the model of deploying software from a centralized system to run on a local computer (or remotely from the cloud). As a metered service, SaaS allows you to lease an application and pay only for the time used.

That's only an overview of cloud computing. This view ignores some of the other aspects of the cloud, such as *data-Storage-as-a-Service* (dSaaS), which provides storage as a metered service in which the consumer is billed based on used capacity (the amount of storage used) and utilization (bandwidth requirements for the storage). Cloud services have also emerged, which provide internal mechanisms for interoperability as well as external application program interfaces (APIs), such as Web services.

3.2 The cloud computing landscape

Over recent years, there's been an explosion of investment into cloud computing and related infrastructure. This massive investment indicates that there is demand for virtualization of resources inside the cloud. There is a plenty of new services, some of which are shown in Figure 9.

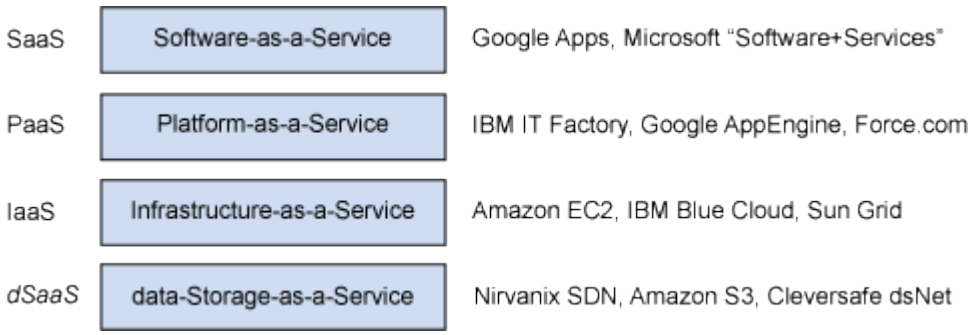

Figure 9. Cloud computing layers with offerings

This is by no means an exhaustive list of offerings, as it changes quite frequently. However, it does provide an overview of some of the offerings and how they are differentiated.

3.2.1 Software-as-a-Service

SaaS is the ability to access software over the Internet as a service. An early approach to SaaS was the Application Service Provider (ASP). ASPs provide

subscriptions to software that is hosted or delivered over the Internet. The ASP delivers the software and charges fees based on its use. In this way, you don't purchase the software but simply lease it on an as-needed basis.

A growing number of software companies offer their products using the traditional model, where customers host the application suite within their enterprise, or as SaaS, where customers host the application suite and make it available over the Internet. Microsoft itself provides its office automation suite, Office 365, both locally installed on your pc or remotely available as executed in its datacenters, and usable in the online version through your web browser.

In fact the use of software over the Internet that executes remotely can be in the form of services used by a local application (defined as *Web services*) or a remote application observed through a Web browser. One example of a remote application service is Google Apps, which provides several enterprise applications through a standard Web browser. Remotely executing applications commonly rely on an application server to expose needed services. An *application server* is a software framework that exposes APIs for software services (such as transaction management or database access). Examples include Red Hat JBoss Application Server, Apache Geronimo, and IBM® WebSphere® Application Server.

3.2.2 Platform-as-a-Service

PaaS can be described as an entire virtualized platform that includes one or more

servers (virtualized over the set of physical servers), operating systems, and specific applications (such as Apache and MySQL for Web-based applications). In some cases, these platforms can be predefined and selected; in others, you can provide a VM image that contains all the necessary user-specific applications.

One interesting example of a PaaS is Google App Engine. App Engine is a service that allows you to deploy your Web applications on Google's very scalable architecture. App Engine provides you with a sandbox for your Python, Java, Php, Go, Ruby, C#, Node.js applications that can be referenced over the Internet. App Engine provides APIs for persistently storing and managing data (using the Google Query Language, or GQL) in addition to support for authenticating users, manipulating images, and sending e-mail. The sandbox in which the Web application runs restricts access to the underlying operating system. Although App Engine limits the functionality available to your application, it supports the construction of useful Web services.

Another example of a PaaS is MongoDB, which is both a cloud platform and a downloadable open-source document database. Among the many services offered there is also an hosted platform, called MongoDB Cloud Manager, for managing your database on the infrastructure of your choice.

3.2.3 Infrastructure-as-a-Service

IaaS is the delivery of computer infrastructure as a service. This layer differs from PaaS in that the virtual hardware is provided without a software stack.

Instead, the consumer provides a VM image that is invoked on one or more

virtualized servers. IaaS is the rawest form of computing as a service, beside of access to the physical infrastructure. The most well-known commercial IaaS provider is Amazon Elastic Compute Cloud (EC2). In EC2, you can specify a particular VM (operating system and application set), and then deploy your applications on it or provide your own VM image to execute on the servers. You're then billed simply for compute time, storage, and network bandwidth.

3.3 Linux and open source in the cloud

Linux and open source technologies play a huge role into the world of cloud computing.

Owing to the new dynamic nature of virtualization and the new capabilities it provides, new management schemes are needed [12]. This management is best done in layers, considering local management at the server, as well as higher-level infrastructure management, providing the overall orchestration of the virtual environment.

Placing multiple virtualization nodes on a physical network with shared storage, orchestrating management over the entire infrastructure, then providing front-end load balancing of incoming connections (whether in a private or a public setting) with caching and filtering, you will realize a virtual infrastructure called a "*cloud*".

In such an infrastructure (Fig.10) dormant servers can be powered down until needed for additional compute capacity, providing better power efficiency, with VMs balanced (even dynamically) across the nodes depending upon their

collective loads.

In the next paragraphs I will show where open source software is being applied to build out a dynamic cloud infrastructure.

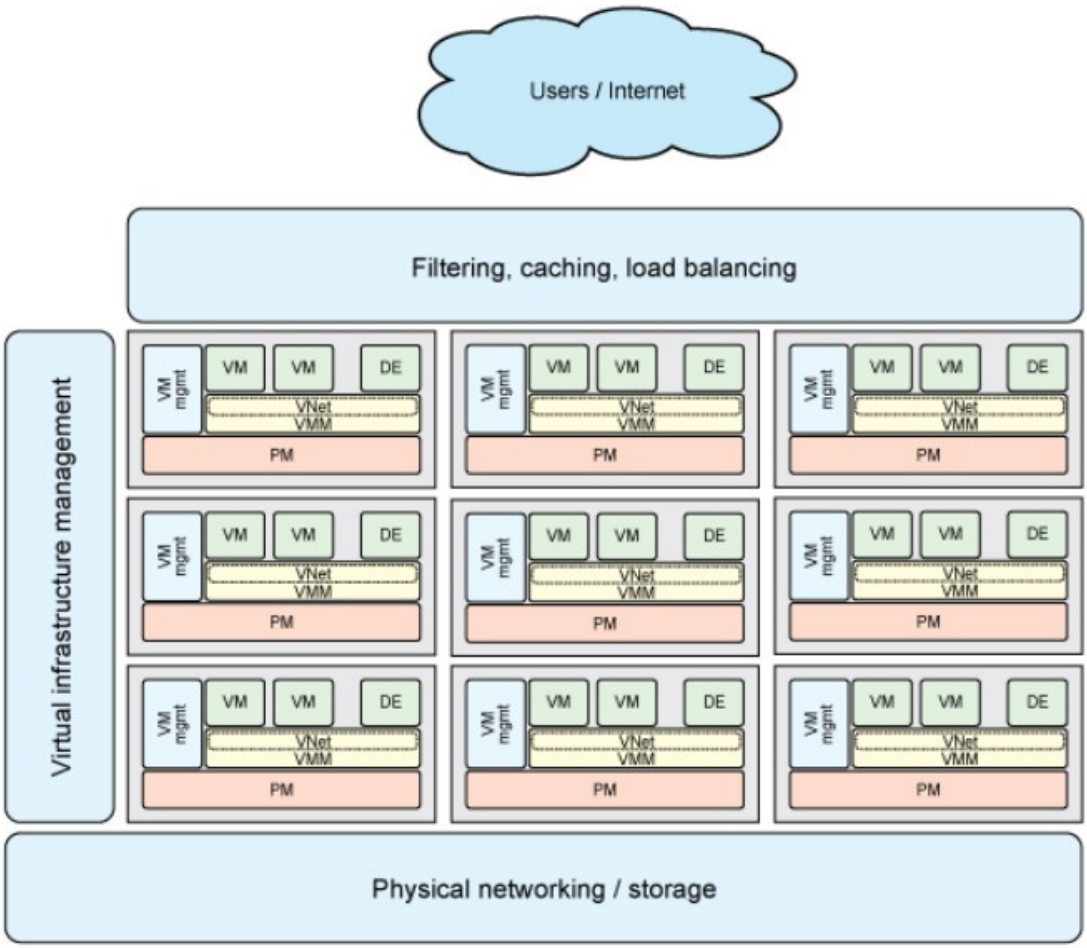

Figure 10. Cloud Computing Infrastructure

3.3.1 Core virtual computing open source technologies

The Linux landscape is seeing a wave of development focused on virtualized infrastructures for virtualization, management, and larger-scale integration of cloud software packages. Let's start with a view of open source at the individual node level, then step up to the infrastructure to see what's happening there.

3.3.1.1 Hypervisors

The base of the cloud at the node level is the hypervisor. Although virtualization is not a requirement, it provides undisputed capabilities for scalable and power-efficient architectures. A good number of open source virtualization solutions exists: among these we have seen the Linux Kernel Virtual Machine (KVM), but I would mention also LXC, an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel. The Xen hypervisor is also widely used within public and private IaaS solutions due to its performance advantages.

Outside of converting Linux to a hypervisor, there are other solutions that take a guest VM-focused approach. User-Mode Linux (UML) is another approach that modifies a guest Linux kernel to run on top of another Linux operating system, without hypervisor extensions. Because many users want to run an unmodified kernel, full virtualization solutions (such as KVM) are preferred.

3.3.1.2 Device emulation

The hypervisor provides the means to share the CPU with multiple operating systems (CPU virtualization), but to provide full virtualization, the entire environment must be virtualized for the VMs. Machine or platform emulation can be performed in a number of ways, but a popular open source package that supports a number of hypervisors is called QEMU. QEMU is a complete emulator and hypervisor. But KVM makes use of QEMU for device emulation as a separate process in the user space. One interesting feature of QEMU is that because it provides disk emulation (through the QCOW format), QEMU provides other advanced features such as snapshots and live VM migration.

KVM, since kernel 2.6.25, uses virtio as a means of optimizing I/O virtualization performance. It does this by introducing paravirtualized drivers into the hypervisor with hooks from the guest to bring performance to near-native levels. This works only when the operating system can be modified for this purpose, but finds use in Linux guest on Linux hypervisor scenarios.

Today, virtio and QEMU work together so emulated device transactions can be optimized between the Linux guest and QEMU emulator in the user space.

3.3.1.3 Virtual networking

As VMs consolidate onto physical servers, the networking needs of the platform intensify. But rather than force all of the VM's networking to the physical layer of the platform, local communication could instead be virtualized itself. To optimize

network communication among Vms, there is the introduction of the virtual switch. The vSwitch behaves like a physical switch, but is virtualized into the platform. In figure 11, virtualized interfaces (VIFs) associated with the VMs communicate through the virtual switch to the physical interfaces (PIFs).

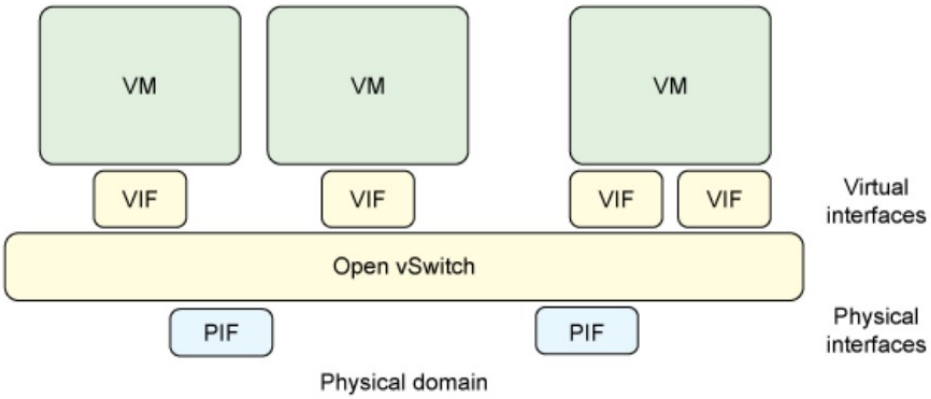

Figure 11 High-level view of Open vSwitch with virtual and physical interfaces

Open source is addressing this problem as well, with one very interesting solution called the Open vSwitch. In addition to providing a virtual switch for virtual environments, the vSwitch can also integrate across physical platforms and provide enterprise-level features like virtual local area networks (VLANs), priority-based Quality of Service (QoS), trunking, and support for hardware acceleration (such as single-root I/O virtualization [IOV] network adapters). The Open vSwitch is currently available for Linux kernels and supports the range of Linux-based virtualization solutions (Xen, KVM, VirtualBox) and management standards like Remote Switched Port Analyzer (RSPAN), NetFlow, etc.

3.3.1.4 VM tools and technologies

As VMs are an aggregation of operating system, root file system, and configuration, the space is ripe for tool development. But to realize the full potential of VMs and tools, there must be a portable way to assemble them. The current approach, called the Open Virtualization Format (OVF) is a VM construction that is flexible, efficient, and portable. OVF wraps a virtual disk image in an XML wrapper that defines the configuration of the VM, including networking configuration, processor and memory requirements, and a variety of extensible metadata to further define the image and its platform needs. The key capability provided by OVF is the portability to distribute VMs in a hypervisor-agnostic manner.

A number of utilities exist to manage VM images (VMIs) as well as convert them to and from other formats. The ovftool from VMware is a useful tool that you can use for VMI conversion, for example to convert from the VMware Virtual Disk (VMDK) format into OVF. This tool and others are useful once you have a VMI, but what if you have a physical server you'd like to convert into a VMI? You can employ a useful tool called Clonezilla for this purpose. Although it was originally developed as a disk-cloning tool for disaster recovery, you can use it to convert a physical server instance into a VM for easy deployment into a virtualized infrastructure. Numerous other tools exist (such as utilities built upon libvirt) or are in development for conversion and management as the OVF format gains adoption.

3.3.1.5 Local management

Red Hat introduced the libvirt library as an API for managing platform virtualization (hypervisors and VMs). What makes libvirt interesting is that it supports a number of hypervisor solutions (KVM and Xen for example) and provides API bindings for a number of languages (such as C, Python, and Ruby). It provides the "last mile" of management, interfacing directly with the platform hypervisor and extending APIs out to larger infrastructure-management solutions. With libvirt, it's simple to start and stop VMs, and it provides APIs for more advanced operations, such as migration of Vms between platforms. Using libvirt, it's also possible to use its shell (built on top of libvirt), called virsh.

3.3.2 Infrastructure open source technologies

Other open source applications support session management and infrastructure management.

3.3.2.1 Session management

Building a scalable and balanced web architecture depends upon the ability to balance web traffic across the servers that implement the back-end functionality. A number of load-balancing solutions exist; one of the most powerful was open sourced by Yahoo and donated to Apache Foundation, and it is known by the name of Traffic Server.

Traffic Server represents an interesting project, because it encapsulates a large

number of capabilities in one package for cloud infrastructures, including session management, authentication, filtering, load balancing, and routing. Yahoo! initially acquired this product from Inktomi, but the code is now publicly available to the community.

3.3.2.2 Infrastructure management

Larger-scale infrastructure management (managing many hypervisors and even more VMs) can be accomplished in a number of ways. Two of the more common solutions are each built from the same platform (libvirt).

The *oVirt* package is an open VM management tool that scales from a small number of VMs to thousands of VMs running on hundreds of hosts. The oVirt package, developed by Red Hat, is a web-based management console that, in addition to traditional management, supports the automation of clustering and load balancing. The oVirt tool is written in the Python language.

VirtManager, also based on libvirt and developed by Red Hat, is an application with a GTK+ UI (instead of being web-based like oVirt). VirtManager presents a much more graphically rich display (for live performance and resource utilization) and includes a VNC client viewer for a full graphical console to remote Vms.

Puppet is another open source package designed for data center infrastructure: although not designed solely for virtualized infrastructures, it simplifies the management of large infrastructures by abstracting the details of the peer operating system. It does this through the use of the Puppet language. Puppet is

ideal for automating administrative tasks over large numbers of servers and is widely used today.

3.3.2.3 Integrated IaaS solutions

The following open source packages take a more holistic approach by integrating all of the necessary functionality into a single package (including virtualization, management, interfaces, and security). When added to a network of servers and storage, these packages produce flexible cloud computing and storage infrastructures (IaaS). For details about these platforms, see Resources.

Eucalyptus

One of the most popular open source packages for building cloud computing infrastructures is Eucalyptus (for Elastic Utility Computing Architecture for Linking Your Programs to Useful Systems). What makes it unique is that its interface is compatible with Amazon Elastic Compute Cloud (Amazon EC2 Amazon's cloud computing interface). Additionally, Eucalyptus includes Walrus, which is a cloud storage application compatible with Amazon Simple Storage Service (Amazon S3 Amazon's cloud storage interface).

Eucalyptus supports KVM/Linux and Xen for hypervisors and includes the Rocks cluster distribution for cluster management.

OpenNebula

OpenNebula is another interesting open source application (under the Apache license) developed at the Universidad Complutense de Madrid. In addition to supporting private cloud construction, OpenNebula supports the idea of hybrid

clouds. A hybrid cloud permits combining a private cloud infrastructure with a public cloud infrastructure (such as Amazon) to enable even higher degrees of scaling.

OpenNebula supports Xen, KVM/Linux, and VMware and relies on elements like libvirt for management and introspection.

Nimbus

Nimbus is another IaaS solution focused on scientific computing. With Nimbus, you can lease remote resources (such as those provided by Amazon EC2) and manage them locally (configure, deploy VMs, monitor, etc.). Nimbus morphed from the Workspace Service project (part of Globus.org). Being dependent on Amazon EC2, Nimbus supports Xen and KVM/Linux.

Xen Cloud Platform

Citrix has integrated Xen into an IaaS platform, using Xen as the hypervisor while incorporating other open source capabilities such as the Open vSwitch. An interesting advantage to the Xen solution is the focus on standards-based management (including OVF, Distributed Management Task Force [DTMF], the Common Information Model [CIM], and Virtualization Management Initiative [VMAN]) from the project Kensho. The Xen management stack supports SLA guarantees, along with detailed metrics for charge-back.

OpenQRM

OpenQRM is categorized as a data center management platform. OpenQRM provides a single console to manage an entire virtualized data center that is architecturally pluggable to permit integration of third-party tools. OpenQRM

integrates support for high availability (through redundancy) and supports a variety of hypervisors, including KVM/Linux, Xen and Vmware.

OpenStack

Today, the leading IaaS solution is called OpenStack [13]. OpenStack was released in July 2010, and has quickly become the standard open-source IaaS solution. OpenStack is a combination of two cloud initiatives from RackSpace Hosting (Cloud Files) and NASA's Nebula platform. OpenStack is being developed in the Python language, and is under active development under the Apache license.

Each OpenStack deployment embraces a wide variety of technologies, spanning Linux distributions, database systems, messaging queues, OpenStack components themselves, access control policies, logging services, security monitoring tools, and much more.
I briefly introduce the kinds of clouds (private, public, and hybrid) before presenting an overview of the OpenStack components, referring briefly also to the most common security issues related to the use of such components.

3.3.2.4 Cloud types

OpenStack is a key enabler in the adoption of cloud technology and has several common deployment use cases. These are commonly known as Public, Private, and Hybrid models. The following sections use the National Institute of Standards and Technology (NIST) definition of cloud to introduce these different types of cloud as they apply to OpenStack.

Public cloud

According to NIST, a public cloud is one in which the infrastructure is open to the general public for consumption. OpenStack public clouds are typically run by a service provider and can be consumed by individuals, corporations, or any paying customer. A public-cloud provider might expose a full set of features such as software-defined networking or block storage, in addition to multiple instance types.

By their nature, public clouds are exposed to a higher degree of risk. As a consumer of a public cloud, you should validate that your selected provider has the necessary certifications, attestations, and other regulatory considerations. As a public cloud provider, depending on your target customers, you might be subject to one or more regulations. Additionally, even if not required to meet regulatory requirements, a provider should ensure tenant isolation as well as protecting management infrastructure from external attacks.

Private cloud

At the opposite end of the spectrum is the private cloud. As NIST defines it, a private cloud is provisioned for exclusive use by a single organization comprising multiple consumers, such as business units. The cloud may be owned, managed, and operated by the organization, a third-party, or some combination of them, and it may exist on or off premises. Private-cloud use cases are diverse and, as such, their individual security concerns vary.

Community cloud

NIST defines a community cloud as one whose infrastructure is provisioned for

the exclusive use by a specific community of consumers from organizations that have shared concerns (for example, mission, security requirements, policy, or compliance considerations). The cloud might be owned, managed, and operated by one or more of organizations in the community, a third-party, or some combination of them, and it may exist on or off premises. This is, on my own opinion, the type of cloud that best fits the characteristics of our lab installation, at least for the first stage of the project..

Hybrid cloud

A hybrid cloud is defined by NIST as a composition of two or more distinct cloud infrastructures, such as private, community, or public, that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability, such as cloud bursting for load balancing between clouds. For example, an online retailer might present their advertising and catalogue on a public cloud that allows for elastic provisioning. This would enable them to handle seasonal loads in a flexible, cost-effective fashion. Once a customer begins to process their order, they are transferred to a more secure private cloud that is PCI DSS compliant (Payment Card Industry Data Security Standard). Your security measures depend where your deployment falls upon the private public continuum.

3.3.3 OpenStack service overview

OpenStack embraces a modular architecture to provide a set of core services that

facilitates scalability and elasticity as core design tenets. In the following paragraphs I briefly review OpenStack components, their use cases and security considerations.

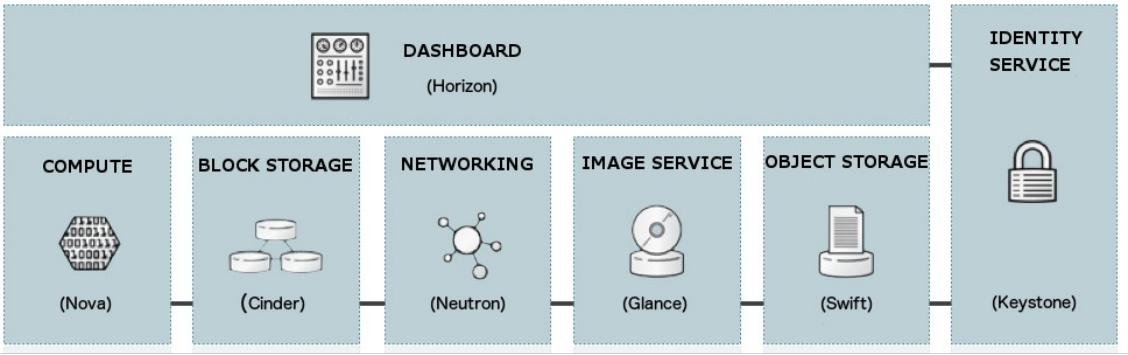

Figure 12. Openstack main components

3.3.3.1 Compute

The *OpenStack Compute Service* (*nova*) provides services to support the management of virtual machine instances at scale, instances that host multi-tiered applications, dev or test environments, "Big Data" crunching Hadoop clusters, or high-performance computing.

The Compute Service facilitates this management through an abstraction layer that interfaces with supported hypervisors, particularly KVM, chosen for all the nodes of our lab installation.

Compute security is critical for an OpenStack deployment. Hardening techniques should include support for strong instance isolation, secure communication

between Compute sub-components, and resiliency of public-facing API endpoints.

3.3.3.2 Object Storage

The *OpenStack Object Storage Service* (*swift*) provides support for storing and retrieving arbitrary data in the cloud. The Object Storage service provides both a native API and an Amazon Web Services S3-compatible API. The service provides a high degree of resiliency through data replication and can handle petabytes of data. It is important to understand that object storage differs from traditional file system storage. Object storage is best used for static data such as media files (MP3s, images, or videos), virtual machine images, and backup files. Object security should focus on access control and encryption of data in transit and at rest. Other concerns might relate to system abuse, illegal or malicious content storage, and cross-authentication attack vectors.

3.3.3.3 Block Storage

The *OpenStack Block Storage Service* (*cinder*) provides persistent block storage for compute instances. The Block Storage service is responsible for managing the life-cycle of block devices, from the creation and attachment of volumes to instances, to their release. Security considerations for block storage are similar to that of object storage.

3.3.3.4 Shared File Systems

The *Shared File Systems Service* (*manila*) provides a set of services for managing shared file systems in a multi-tenant cloud environment, similar to how OpenStack provides for block-based storage management through the OpenStack Block Storage service project. With the Shared File Systems service, you can create a remote file system, mount the file system on your instances, and then read and write data from your instances to and from your file system.

3.3.3.5 Networking

The *OpenStack Networking Service* (*neutron*) provides various networking services to cloud users (tenants) such as IP address management, DNS, DHCP, load balancing, and security groups (network access rules, like firewall policies). This service provides a framework for software defined networking (SDN) that allows for pluggable integration with various networking solutions.
OpenStack Networking allows cloud tenants to manage their guest network configurations. Security concerns with the networking service include network traffic isolation, availability, integrity, and confidentiality.

3.3.3.6 Dashboard

The *OpenStack Dashboard* (*horizon*) provides a web-based interface for both cloud administrators and cloud tenants. Using this interface, administrators and

tenants can provision, manage, and monitor cloud resources. The dashboard is commonly deployed in a public-facing manner with all the usual security concerns of public web portals.

3.3.3.7 Identity service

The *OpenStack Identity Service* (*keystone*) is a shared service that provides authentication and authorization services throughout the entire cloud infrastructure. The Identity service has pluggable support for multiple forms of authentication. Security concerns with the Identity service include trust in authentication, the management of authorization tokens, and secure communication.

3.3.3.8 Image service

The OpenStack Image service (glance) provides disk-image management services, including image discovery, registration, and delivery services to the Compute service, as needed. Trusted processes for managing the life cycle of disk images are required, as are all the previously mentioned issues with respect to data security.

3.3.3.9 Messaging and databases

Messaging is used for internal communication between several OpenStack

services. By default, OpenStack uses message queues based on the standard AMQP (Advanced Message Queuing Protocol). Like most OpenStack services, AMQP supports pluggable components. Today the implementation back end could be RabbitMQ, Qpid, or ZeroMQ. Because most management commands flow through the message queuing system, message-queue security is a primary security concern for any OpenStack deployment.

Several components use databases as backend, in particular MariaDB, though it is not explicitly called out. Securing database access is yet another security concern.

4 Performance monitoring

Virtual machines are becoming commonplace as a stable and flexible platform to run many workloads [14]. As developers continue to move more workloads into virtual environments, they need ways to analyze the performance characteristics of those workloads. Some standard profiling tools like Vtune and the Linux Performance Counter Subsystem rely on CPUs' hardware performance counters, which were exposed to the guests by most hypervisors only recently: in the case of KVM the support for virtual counters has only been officially made available starting from 2012.

4.1 Hardware-Based Monitoring

Performance monitoring means collecting information related to how an application or system performs. This information can be obtained either through software-based means or from the CPU or chipset.

Many modern processors contain a performance monitoring unit (PMU). Intel[15] and AMD[16] provide similar interfaces to their performance counting hardware. Each CPU has its own set of performance counters and performance event select registers. The event select register is used to specify which microarchitectural event is to be counted, and contains bits to enable, filter the count results, and raise interrupts if the counter overflows from negative to positive.

The Performance Monitoring Unit of processors supporting Intel® 64 and IA-32

architectures, generally consists of collections of MSRs (Model Specific Registers). The collection of MSRs include counter registers, event programming MSRs, global event control MSRs. PMUs of older processors are model-specific; PMU interfaces in more recent processors are evolving towards higher degrees of architectural stability.

MSR registers are accessed via the RDMSR and WRMSR instruction. Certain counter registers can be accessed via the RDPMC instruction at any privilege level while RDMSR and WRMSR are available only to software running at ring 0.

Events common across many architectures include cycle counts (relative to core cycles and to constant-rate cycles), TLB accesses and misses, last-level cache accesses and misses, and instruction and branch retired counts. In addition to these common events, each CPU generation has its own assortment of architecture-specific events including store-to-load forwarding failure counts and functional unit stall events. AMD and Intel each have mechanisms to enable and disable individual counters during the state transition between the hypervisor's own code and running the guest.

A typical usage of the performance counters could include configuring Event Select 0 to count Last-Level Cache misses in all privilege levels with the overflow interrupt disabled and configuring Event Select 1 to count Last-Level Cache accesses with identical privilege and interrupt settings. A profiler then samples the Event Counts 0 and 1 and calculates per-sample period differences to track the ratio of cache misses to accesses. In addition, the interrupt facility of the hardware counters can be enabled to cause interrupts after a set number of

events.

For example, setting the Event Select's interrupt-enable bit and setting the corresponding counter to -10,000 would cause the hardware to raise an interrupt after the 10,000th cache miss.

Another special counting mode used by Intel is PEBS (Precise Event Based Sampling), in which counters can be configured to overflow, interrupt the processor, and capture machine state at that point.

Uncore or Northbridge counters are shared among multiple physical CPUs and thus are less amenable to time-multiplexing.

Compared to software profilers, hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches and main memory etc. Another benefit of using them is that no source code modifications are needed in general.

Many resource and performance monitoring tools are available for non-virtualized systems. The type of tool to be used depends on the granularity of information to be extracted and frequency of profiling.

4.2 Perf: a profiling tool for linux based systems

For the purpose of my research, I have chosen as principal measuring tool *perf*. Perf subsystem [17] has been evolved in mainline kernel to unify performance measurement across the system for processor PMU, software and trace point

events. This uniform framework enables user to understand various system performance bottlenecks in a holistic manner. Also perf has been improved to accommodate performance measurement capabilities in a virtualized environment.

As we can see from Figure 13, 14 and 15, perf_events, the kernel counterpart of perf user mode program, instruments "events", which are a unified interface for different kernel instrumentation frameworks.

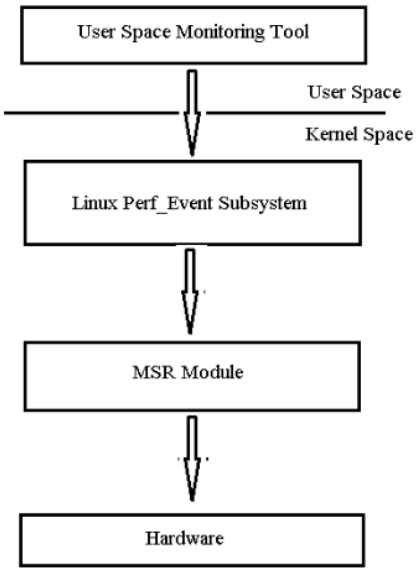

Figure 13. Perf_event stack overview

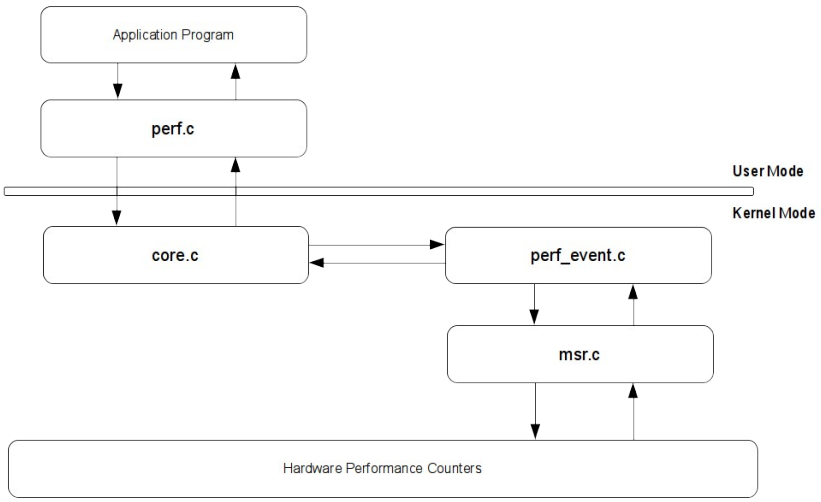

Figure 14. Perf_event software modules detailed

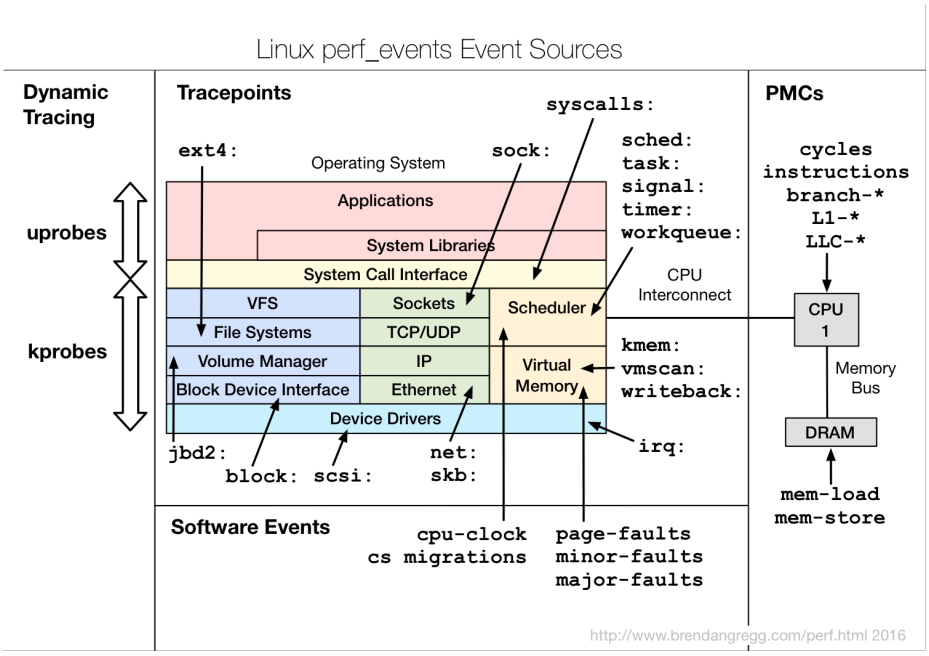

Figure 15. Linux perf_event event sources

The types of events are:

- **Hardware Events**: CPU performance monitoring counters.
- **Software Events**: These are low level events based on kernel counters. For example, CPU migrations, minor faults, major faults, etc.
- **Kernel Tracepoint Events**: This are static kernel-level instrumentation points that are hardcoded in interesting and logical places in the kernel.
- **User Statically-Defined Tracing (USDT)**: These are static tracepoints for user-level programs and applications.
- **Dynamic Tracing**: Software can be dynamically instrumented, creating events in any location. For kernel software, this uses the kprobes framework. For user-level software, uprobes.
- **Timed Profiling**: Snapshots can be collected at an arbitrary frequency, using `perf record -F*Hz*`. This is commonly used for CPU usage profiling, and works by creating custom timed interrupt events.

Details about the events can be collected, including timestamps, the code path that led to it, and other specific details.

Currently available events can be listed using the *`list`* subcommand, as in the following example:

```
# perf list
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  cache-references                                   [Hardware event]
  cache-misses                                       [Hardware event]
  branch-instructions OR branches                    [Hardware event]
```

```
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  ref-cycles                                         [Hardware event]
  cpu-clock                                          [Software event]
  task-clock                                         [Software event]
  page-faults OR faults                              [Software event]
  context-switches OR cs                             [Software event]
  cpu-migrations OR migrations                       [Software event]
  minor-faults                                       [Software event]
  major-faults                                       [Software event]
  alignment-faults                                   [Software event]
  emulation-faults                                   [Software event]
  L1-dcache-loads                                    [Hardware cache
event]
  L1-dcache-load-misses                              [Hardware cache
event]
  L1-dcache-stores                                   [Hardware cache
event]
[...]
  rNNN                                               [Raw hardware
event descriptor]
  cpu/t1=v1[,t2=v2,t3 ...]/modifier                  [Raw hardware
event descriptor]
   (see 'man perf-list' on how to encode it)
  mem:<addr>[:access]                                [Hardware
breakpoint]
  probe:tcp_sendmsg                                  [Tracepoint
event]
[...]
  sched:sched_process_exec                           [Tracepoint
event]
  sched:sched_process_fork                           [Tracepoint
event]
  sched:sched_process_wait                           [Tracepoint
event]
  sched:sched_wait_task                              [Tracepoint
event]
  sched:sched_process_exit                           [Tracepoint
event]
[...]
```

4.2.1 Hardware Events

For my experiments, I focused attention on hardware events coming from performance monitoring counters (PMCs).

Perf_events began life as a tool for instrumenting the processor's performance monitoring unit (PMU) hardware counters, also called performance monitoring counters, or performance instrumentation counters (PICs). These instrument low-level processor activity, for example, CPU cycles, instructions retired, memory stall cycles, level 2 cache misses, etc. Part of them will be listed as Hardware Events, others as Hardware Cache Events.

PMCs are documented in the Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2 and the BIOS and Kernel Developer's Guide (BKDG) For AMD Family Processors. There are thousands of different PMCs available.

A typical processor will implement PMCs in the following way: only a few or several can be recorded at the same time, from the many thousands that are available. This is because they are a fixed hardware resource on the processor (a limited number of registers), and are programmed to begin counting the selected events.

4.2.1.1 CPU Statistics

The `perf stat` command instruments and summarizes key CPU counters (PMCs).

```
# perf stat gzip file1

 Performance counter stats for 'gzip file1':

       1920.159821 task-clock                #    0.991 CPUs utilized
                13 context-switches          #    0.007 K/sec
                 0 CPU-migrations            #    0.000 K/sec
               258 page-faults               #    0.134 K/sec
     5,649,595,479 cycles                    #    2.942 GHz
[83.43%]
     1,808,339,931 stalled-cycles-frontend   #   32.01% frontend
cycles idle    [83.54%]
     1,171,884,577 stalled-cycles-backend    #   20.74% backend
cycles idle    [66.77%]
     8,625,207,199 instructions              #    1.53  insns per
cycle
                                             #    0.21  stalled cycles
per insn [83.51%]
     1,488,797,176 branches                  #  775.351 M/sec
[82.58%]
        53,395,139 branch-misses             #    3.59% of all
branches         [83.78%]

       1.936842598 seconds time elapsed
```

This includes instructions per cycle (IPC), labeled "insns per cycle", or in earlier versions, "IPC". This is a commonly examined metric, either IPC or its invert, CPI. Higher IPC values mean higher instruction throughput, and lower values indicate more stall cycles. I'd generally interpret high IPC values (eg, over 1.0) as good, indicating optimal processing of work. However, I'd want to double check what the instructions are, in case this is due to a spin loop: a high rate of instructions, but a low rate of actual work completed.

There are some advanced metrics now included in `perf stat`:

frontend cycles idle, backend cycles idle, and stalled cycles per instruction. The frontend and backend metrics refer to the CPU pipeline, and are also based on

stall counts. The frontend processes CPU instructions, in order. It involves instruction fetch, along with branch prediction, and decode. The decoded instructions become micro-operations (uops) which the backend processes, and it may do so out of order.

The backend can also process multiple uops in parallel; for modern processors, three or four. Along with pipelining, this is how IPC can become greater than one, as more than one instruction can be completed ("retired") per CPU cycle.

Stalled cycles per instruction is similar to IPC (inverted), however, only counting stalled cycles, which will be caused by memory or resource bus access. This makes it easy to interpret: stalls are latency, so we should reduce stalls to increase performance.

There is also a "detailed" mode for `perf stat`:

```
# perf stat -d gzip file1

 Performance counter stats for 'gzip file1':

       1610.719530 task-clock                #    0.998 CPUs utilized
                20 context-switches          #    0.012 K/sec
                 0 CPU-migrations            #    0.000 K/sec
               258 page-faults               #    0.160 K/sec
     5,491,605,997 cycles                    #    3.409 GHz
[40.18%]
     1,654,551,151 stalled-cycles-frontend   #   30.13% frontend
cycles idle    [40.80%]
     1,025,280,350 stalled-cycles-backend    #   18.67% backend
cycles idle    [40.34%]
     8,644,643,951 instructions              #    1.57  insns per
cycle
                                             #    0.19  stalled cycles
per insn [50.89%]
     1,492,911,665 branches                  #  926.860 M/sec
[50.69%]
        53,471,580 branch-misses             #    3.58% of all
branches         [51.21%]
     1,938,889,736 L1-dcache-loads           # 1203.741 M/sec
```

```
[49.68%]
       154,380,395 L1-dcache-load-misses     #    7.96% of all L1-
dcache hits   [49.66%]
                  0 LLC-loads                #    0.000 K/sec
[39.27%]
                  0 LLC-load-misses          #    0.00% of all LL-
cache hits    [39.61%]

       1.614165346 seconds time elapsed
```

This example includes additional counters for Level 1 data cache events, and last level cache (LLC) events.

I can instrument specific counters, seen in perf list, using the following example, referred particularly to cache events:

```
# perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-
dcache-stores gzip file1
```

```
 Performance counter stats for 'gzip file1':

     1,947,551,657 L1-dcache-loads

       153,829,652 L1-dcache-misses
         #    7.90% of all L1-dcache hits
     1,171,475,286 L1-dcache-stores


       1.538038091 seconds time elapsed
```

The percentage printed is a convenient calculation that perf_events has included, based on the counters I specified. If you include the "cycles" and "instructions" counters, it will include an IPC calculation in the output.

4.2.2 Performance monitoring in KVM virtualized environments

In virtualized systems, the task of profiling and resource monitoring is not straight-forward.

Many datacenters perform CPU overcommittment using hypervisors, running multiple VMs on a single computer where the total VCPU count exceeds the total number of PCPUs. The hypervisor must share PCPUs among all the VCPUs, giving each VCPU a fraction of the total runtime of the system. The sharing of hardware resources requires the hypervisor to apply heuristics to enable guest operating systems to accurately keep track of absolute time, often called wall-clock time. The guest operating system wall-clock should track absolute time over the long term. To achieve this, while the VM is descheduled, some hypervisors, like Vmware ESXi, make available a virtual timer device that is used by the guest operating system for timekeeping. This device is allowed to fall behind real time and later catch up faster than real time when the VM is rescheduled. This way, over the longer term, these devices track absolute real time. Profilers, on the other hand, are more concerned with relative time differences over the short term, and want to count only the time that the VCPU is scheduled on a PCPU.

This tension over the desired semantics of a timer device requires the hypervisor to carefully trade off keeping a guest's notion of wall-clock time correct and giving a notion of time appropriate for profilers' use. Both Intel and AMD CPUs provide an event called *core cycles not halted*, which tracks the CPU cycle count independently of wall-clock time. CPU frequency can increase or decrease due to

power saving modes, and CPU cycles can stop entirely if the OS has executed the HLT instruction. The notion of core cycles not halted is thus a convenient hardware interface that can be extended for profiling in a virtual environment. The hypervisor can define core cycles not halted to count only core clock cycles when the VCPU is in context on a PCPU, including time spent in the hypervisor on that VCPU's behalf.

Sharing hardware leads to other, less direct effects. Just as multiple processes may compete for cache and other resources, multiple VCPUs and other unrelated hypervisor threads that share a physical core can pollute each other's caches, branch predictors, TLBs, and other microarchitectural state.

4.2.2.1 Perf kvm: the host perspective

Guest virtual machine's individual performance and its impact on the host machine can be measured from various directions with the help of perf tool. Performance data collected from all these methods help us monitor and detect a situation of performance degradation [18].

Perf tool has been improved to have capabilities to profile KVM virtualized environment. A new subcommand *kvm* has been added to that effect. Perf kvm understands how qemu process address space encapsulates the entire guest kernel and guest user space and how to resolve addresses inside that into guest kernel symbols.

Perf kvm requires access to guest /proc/modules and /proc/kallsyms file sets to be able to resolve all the captured event instruction addresses into respective

guest kernel symbols. Users can either transfer these files from guests to the host and then provide them explicitly while invoking perf kvm session or they can mount the guest root file system (typically with sshfs) in the host, so that perf kvm can extract required files from the designated mount point.

There are two different methods available in perf kvm to profile either the host or guest virtual machines. One is `perf kvm top` and the other one being `perf kvm record` followed by `perf kvm report`.

Any perf session is always initiated from the host machine which can subsequently profile either the host or a guest or both. The profiling methods mentioned above are sampled counter based which associates event captured instruction addresses with respective symbols and sorts the symbols according to their relative percentage across the workload. During the session, perf captures the event's sample data from the kernel and stores them in a file named `perf.data.kvm` (`perf.data.guest` or `perf.data.host` if they are profiled individually). Though in case of `perf top`, this file is created and analysed on the run and the results are refreshed periodically.

The host perf exclusively configures, initiates and terminates the PMU access for any process requesting PMU events. The direct control over the PMU cannot be granted to the guest virtual machine as they are not aware of other guests who might be requesting PMU events at the same point of time. Host is always required to arbiter access to the PMU. Because of these reasons, for host linux kernels before version 3.3 and qemu-kvm before version 1.2.5, perf inside guest does not support hardware PMU events.

A typical perf kvm command is represented by the following:

```
# perf kvm --host --guest --guestkallsyms=guest-kallsyms \
--guestmodules=guest-modules record -a -o perf.data
```

gathering data both from host and guest obtained files “guest-kallsyms” and “guest-modules”, recording and then reporting events to “perf.data.kvm” file.

4.2.2.2 vPMU: the guest perspective

Performance counter virtualization for the hardware-assisted KVM virtual machine monitor is included in recent versions of the Linux kernel.
Users of a public cloud service are normally not granted the privilege to run a profiler in the VMM, which is necessary for conducting system-wide profiling [19]. To achieve guest-wide profiling, the VMM needs to provide PMU multiplexing, i.e. saving and restoring PMU registers.
The hypervisor context switches all relevant CPU state when each VCPU is scheduled and descheduled. To virtualize performance counters, the hypervisor must context switch the active performance counter state, in addition to the context switching of general purpose registers and control state. This serves to time-multiplex the CPU and performance counter hardware resources and guarantee that virtual counters do not advance while that VCPU is out of context. The context switching of the counter state satisfies previous definition of unhalted core-cycles.

The virtualization extensions augment x86 with two new operation modes: host

mode and guest mode. KVM runs in host mode, and its guests run in guest mode. Host mode is compatible with conventional x86, while guest mode is very similar to it but deprivileged in certain ways. Guest mode supports all four privilege levels and allows direct execution of the guest code. A virtual machine control structure (VMCS) is introduced to control various behaviors of a virtual machine. Two transitions are also defined: a transition from host mode to guest mode called a VM-entry, and a transition from guest mode to host mode called a VM-exit. Regarding performance profiling, if a performance counter overflows when the CPU is in guest mode, the currently running guest is forced to exit, i.e., the CPU switches from guest mode to host mode. The VM-exit information filed in the VMCS indicates that the current VM-exit is caused by a non-maskable interrupt (NMI). By checking this field, KVM is able to decide whether a counter overflow is contributed by a guest. This approach assumes all NMIs are caused by counter overflows in a profiling session. To be more precise, KVM could also check the content of all performance counters to make sure that NMIs are really caused by counter overflows.

A good guest-wide profiling implementation requires no modifications to the guest and its profiler. The guest profiler reads and writes the physical PMU registers directly as it does in native profiling. KVM is responsible for virtualizing the PMU hardware and forwarding NMIs due to performance counter overflows to the guest. *A user can launch the profiler from the guest and do performance profiling exactly as in a native environment.*

When CPU switch is enabled, KVM saves all the relevant MSRs when a VM-exit happens and restores them when the corresponding VM-resume occurs. By

configuring certain fields in the VMCS, this is done automatically in hardware. When domain switch is enabled, all Linux kernel threads belonging to a guest are tagged and grouped into one domain. When the Linux kernel switches to a thread not belonging to the current domain, it saves and restores the relevant registers.

To enable the vPMU in a KVM hardware assisted guest VM, I must pass the parameter <cpu mode='host-passthrough'> during virtual machine launch. By the way this was a default option in our lab Openstack installation.

With these assumptions, after the vPMU is enabled, I can display a virtual machine's performance statistics by simply running the `perf` command from the guest virtual machine.

The hardware events available inside the VM are the same as those listed for the virtualization host, while the hardware cache event types available for the virtual machine are fewer. In my experiments and next discussion I will use only the hardware events.

Follows an output of `perf list` command inside virtual machine, limited to harware events:

```
branch-instructions OR branches  [Hardware event]
branch-misses                    [Hardware event]
bus-cycles                       [Hardware event]
cache-misses                     [Hardware event]
cache-references                 [Hardware event]
cpu-cycles OR cycles             [Hardware event]
instructions                     [Hardware event]
ref-cycles                       [Hardware event]
```

5 The evolution of the testbed

5.1 General characteristics

The very early tests in my work were conducted on a simple dual-core computer, based on i7-3517U, equipped with 8GB ram and a solid state drive with 512 GB storing capacity. The operating system was initially based on Ubuntu Linux 12.04, kernel 3.2, and to act as an hypervisor the qemu-kvm module was loaded while the libvirtd daemon was installed and loaded for vm management purposes. Afterwards, to access the newly supported feature of virtual performance counters inside virtual machines, I upgraded the operating system to the next long term support release 14.04, supporting kernel 3.13 and qemu-kvm 2.0. After first measurement sessions, I focused my attention on some inconsistencies and missing values returned by virtual counters, caused by a bug in qemu-kvm that was quickly corrected by next module release.

Basing upon a more reliable base system I could extract some important informations to narrow the search of main performance indicators, useful for the purpose of the study, focusing on branch misprediction, cache misses, virtual cpu cycles and number of instructions.

At the same time of the early stages of my research work “InsideOutCC”, that would have resulted in this doctoral dissertation, I have been party to an important european project, within the wider Future Internet Project, named FIWARE.

FIWARE is an open source computing platform, sponsored and financed by the

European Union, with the specific mission *"to build an open sustainable ecosystem around public, royalty-free and implementation-driven software platform standards that will ease the development of new Smart Applications in multiple sectors"*.

I will describe in more detail aims and finality of FIWARE Project in Annex A, but now I shall focus on the computing infrastructure that was used in my experiments. In particular FIWARE Genoa Node is an instance of a widely distributed federated community cloud, based on OpenStack.

AgiLab - DITEN, the laboratory I belonged to and where I conducted my studies, together with TnT-Lab – DITEN and CNIT-GE Unit designed, installed and configured the whole Genoa Node, that was hosted on DITEN-UniGE equipment rooms, expressly dedicated to the dawning data center.

5.2 Focus on compute nodes

5.2.1 Hypervisors in OpenStack

Whether OpenStack is deployed within private data centers or as a public cloud service, the underlying virtualization technology provides enterprise-level capabilities in the realms of scalability, resource efficiency, and uptime. While such high-level benefits are generally available across many OpenStack-

supported hypervisor technologies, there are significant differences in the security architecture and features for each hypervisor, particularly when considering the security threat vectors which are unique to elastic OpenStack environments. As applications consolidate into single Infrastructure-as-a-Service (IaaS) platforms, instance isolation at the hypervisor level becomes paramount. The requirement for secure isolation holds true across commercial, government, and military communities.

Within the OpenStack framework, you can choose among many hypervisor platforms and corresponding OpenStack plug-ins to optimize your cloud environment.

FIWARE choice fell on KVM for two main reasons: product maturity and certification.

One of the biggest indicators of a hypervisor's maturity is the size and vibrancy of the community that surrounds it. As this concerns security, the quality of the community affects the availability of expertise if you need additional cloud operators. It is also a sign of how widely deployed the hypervisor is, in turn leading to the battle readiness of any reference architectures and best practices.

Further, the quality of community, as it surrounds an open source hypervisor like KVM, has a direct impact on the timeliness of bug fixes and security updates. When investigating both commercial and open source hypervisors, you must look into their release and support cycles as well as the time delta between the announcement of a bug or security issue and a patch or response.

One additional consideration when selecting a hypervisor is the availability of various formal certifications and attestations. While they may not

be requirements for your specific organization, these certifications and attestations speak to the maturity, production readiness, and thoroughness of the testing a particular hypervisor platform has been subjected to.

Common Criteria [20] is an internationally standardized software evaluation process, used by governments and commercial companies to validate software technologies perform as advertised. In the government sector, NSTISSP No. 11 mandates that U.S. Government agencies only procure software which has been Common Criteria certified, a policy which has been in place since July 2002.

In addition to validating a technologies capabilities, the Common Criteria process evaluates how technologies are developed, verifying

- how is source code management performed?
- how are users granted access to build systems?
- is the technology cryptographically signed before distribution?

The *KVM hypervisor has been Common Criteria certified* through the U.S. Government and commercial distributions. These have been validated to separate the runtime environment of virtual machines from each other, providing foundational technology to enforce instance isolation. In addition to virtual machine isolation, KVM has been Common Criteria certified to:

"...provide system-inherent separation mechanisms to the resources of virtual machines. This separation ensures that large software component used for virtualizing and simulating devices executing for each virtual machine cannot interfere with each other. Using the SELinux multi-category mechanism, the virtualization and simulation software instances are isolated. The virtual machine management framework configures SELinux multi-category settings

transparently to the administrator."
I should also point out that KVM is the only hypervisor included in “group A” of fully supported Compute Drivers for Nova Openstack Compute Service; other hypervisors like Hyper-V, Vmware and Xen lack on some functional tests.

Considering the context in which our cloud instance should have been installed, that is a little data center but with full cooling capacities and uninterrupted power availability, we chose an high density dual rack solution, preferring sled servers for compute and controller nodes, while using more traditional 2U servers for ceph storage nodes, hosting a plenty of hard disk drives.
Sled servers are rack-mounted servers that support multiple independent servers in a single 2U or 3U enclosure. These deliver higher density as compared to typical 1U or 2U rack-mounted servers. Our Intel sled servers offer four independent dual-socket nodes in 2U for a total of eight CPU sockets in 2U, sharing one cooling system and double power supply unit for each server.
The type of CPUs we choose had to support virtualization by way of Intel VT-x, in particular Intel Xeon E5-2630V4 and E5-2660V4 where used with hyper-threading capabilities. Hyper-threading is Intel’s proprietary simultaneous multithreading implementation used to improve parallelization on their CPUs. Enabling hyper-threading may improve the performance of multi-threaded applications. Whether you should enable Hyper-Threading on your CPUs depends upon your use case. For example, disabling Hyper-Threading can be beneficial in intense computing environments, but in our tests did not make the difference.
In most cases, hyper-threading CPUs can provide a 1.3x to 2.0x performance benefit over non-hyper-threaded CPUs depending on types of workload.

Dynamic memory available on average per socket was 64GB, while storage totalized 24TB on three ceph storage nodes.

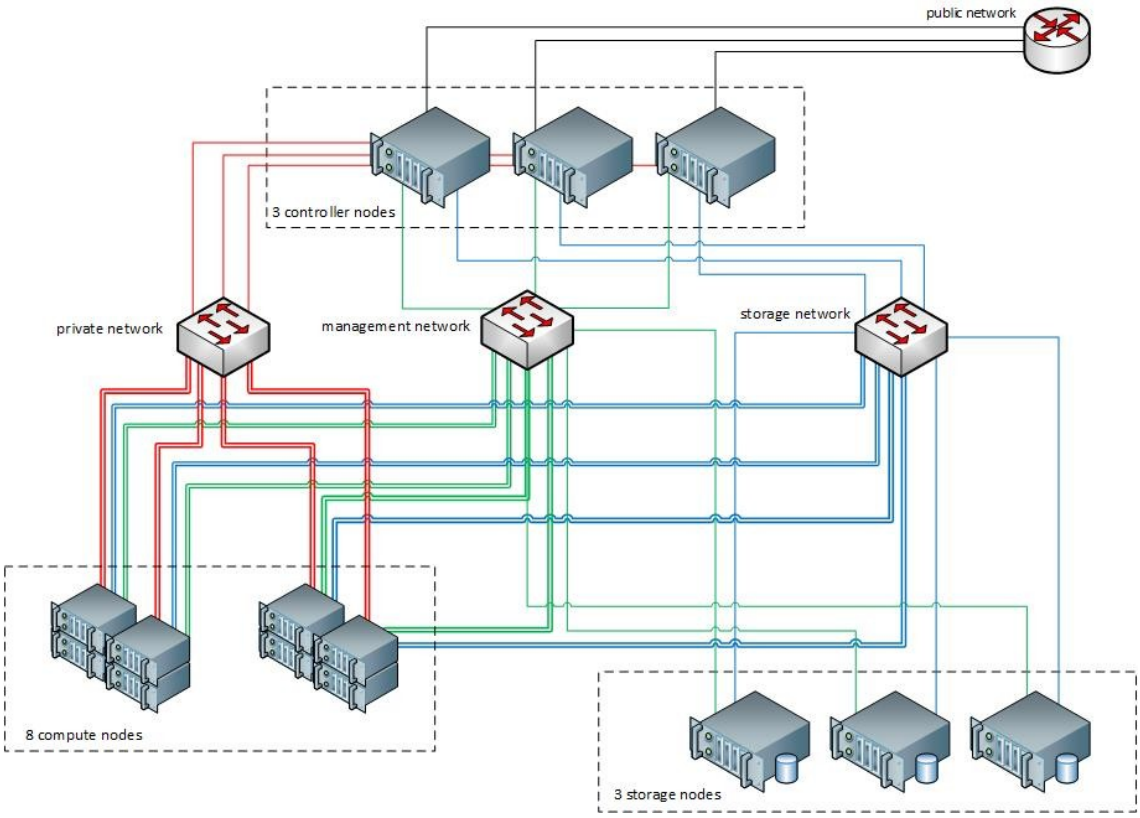

Figure 16. FIWARE Genoa Node infrastructure

5.2.2 CPU and RAM overcommitting

OpenStack allows you to overcommit CPU and RAM on compute nodes. This allows you to increase the number of instances running on your cloud at the cost

of reducing the performance of the instances. The Compute service uses the following ratios by default:

CPU allocation ratio: 16:1

RAM allocation ratio: 1.5:1

The default CPU allocation ratio of 16:1 means that the scheduler allocates up to 16 virtual cores per physical core. For example, if a physical node has 10 cores, the scheduler sees 160 available virtual cores. With typical flavor definitions of 4 virtual cores per instance, this ratio would provide 40 instances on a physical node.

The formula for the number of virtual instances on a compute node is **(OR*PC)/VC**, where:

OR represents CPU overcommit ratio (virtual cores per physical core)

PC number of physical cores

VC number of virtual cores per instance

Similarly, the default RAM allocation ratio of 1.5:1 means that the scheduler allocates instances to a physical node as long as the total amount of RAM associated with the instances is less than 1.5 times the amount of RAM available on the physical node.

For example, if a physical node has 128 GB of RAM, the scheduler allocates instances to that node until the sum of the RAM associated with the instances reaches 192 GB (such as twenty four instances, in the case where each instance has 8 GB of RAM).

Obviously regardless of the overcommit ratio, an instance of virtual machine can not be placed on any physical node with fewer raw (pre-overcommit) resources

than the instance flavor requires.
For my research purposes, owing to an intensive computing pressure I would push inside virtual machines, and a consequently supposed early warning signal, I fixed overcommit ratio to 2:1 both for CPU and RAM.

5.2.3 Bulk and privileged workers

I realized a template for "*bulk worker*" virtual machines, which consists of a single vCPU without pinning, 3 GB RAM and a little virtual disk sizing 3 GB.
The operating system was based on Linux Ubuntu 14.04 server edition, with the only ssh server enabled and listening on network, awaiting for commands. I was particularly careful to disable any underlying background service that could alter measures (i.e. cron, udev, etc.) and to install the test suites "stress" and "stress-ng".
I used this template to instantiate 80 virtual machines, each with its private static ip address, to reach the 2:1 overcommit ratio of a compute node equipped with 40 cores and 128 GB ram.
From the same template I derived a single virtual machine, named *vm-micro*, which is a sort of "*privileged worker*" and represents my particular point of view to probe the degree of saturation of host's computing resources. On this "privileged worker" I installed also the software measuring instrument "perf", previously included in perf-tools package and nowadays included in a linux-tools package specific for each version of the kernel. I used a third type of virtual machine, named *vm-director*, also based on the same template, but not executed

on the same compute node hosting bulk and privileged workers. Vm-director was realized with the express purpose of sending the necessary commands to workers to perform the experiments.
Particularly, to allow the execution of scripts in parallel on more virtual machines, I used a versatile tool like the program *parallel-ssh,* that opens different connections to several hosts via ssh protocol, allowing to pass commands to the virtual machines, all at the same time.
To avoid the interactive authentication steps, ssh public-key from vm-director's root user was exchanged with all controlled virtual machines.

```
parallel-ssh -t 0 -h hosts_25.txt -p 10 /root/test_script.sh
parallel-ssh -t 0 -h hosts_50.txt -p 20 /root/test_script.sh
parallel-ssh -t 0 -h hosts_75.txt -p 30 /root/test_script.sh
parallel-ssh -t 0 -h hosts_100.txt -p 40 /root/test_script.sh
parallel-ssh -t 0 -h hosts_125.txt -p 50 /root/test_script.sh
parallel-ssh -t 0 -h hosts_150.txt -p 60 /root/test_script.sh
parallel-ssh -t 0 -h hosts_175.txt -p 70 /root/test_script.sh
parallel-ssh -t 0 -h hosts_200.txt -p 80 /root/test_script.sh
```

The bulk workers was divided in 8 groups of 10 virtual machines each, to graduate computational load for the host in 25% steps, from 0 to 200%.
I noticed that the extra load caused by the boot phase of each virtual machine could excessively disturb the measures, so I decided to bring all the virtual machines in a stable booted state, without extra computational load, that would

represent the steady basis for all following tests.

I deployed testing scripts on all worker virtual machines and started to step up efforts on 0, 10, 20, and so on till 80 vms, with a delay between each step of about 10 minutes, to stabilize the situation between one test and the other.

```
deploy_script.sh (from vm-director):
#!/bin/bash
        for i in `seq 11 90`;
        do
#               echo $i
            #ssh  -f  172.27.27.$i  "stress  -c  1  -m  2  -t  600s  >
/dev/null 2>&1"
            scp /root/test_script.sh 172.27.27.$i:/root/
        done

test_script.sh (deployed towards each worker vm):
#!/bin/bash
stress -c 1 -m 1 -i 4 -t 600s > /dev/null 2>&1
```

At about 8 minutes from the start of each step, on vm-micro was run the perf tests of hardware PMU indicators visible from inside guest operating system, for a duration of 60 seconds, saving locally a comma separated file with results for further processing of data.

```
(from vm-micro)
```

```
perf stat -x, -I 1000 -e cycles,instructions,branch-misses,cache-
misses stress -c 1 -m 1 -i 4 -t 60s 2>&1| tee Exp0X_DDMM_hhmm.csv
```

to obtain for each launch a comma separated file containing performance counters values, every 1000 ms, for a duration of 60 seconds.
From several measuring sessions, I realized that two particular software mechanisms introduced by KVM could partially alter results:
Memory Ballooning and Kernel Same-page Merging.

5.2.3.1 Memory Ballooning

Through memory ballooning [21], a host server can reclaim unused memory from other less busy virtual machines and reassign it to ones that require it more. Theoretically, a server with 32GB of memory might be able to support a combined virtual machine memory capacity allocation of 64GB simply because all of those virtual machines will not be using the maximum amount of memory they have been assigned at the same time.

The balloon driver in each guest operating system keeps track of the excess memory of each VM and when the hypervisor calls for a memory reclamation through ballooning, the balloon driver in the VM pins down a specific amount of memory so that the VM cannot consume it, and then the hypervisor reclaims that pinned memory for reallocation. If there is a scarcity of unused memory then a memory swap might be initiated in order to fulfill the balloon quota. If this

happens too much, there would be a lot of I/O overhead between the various VMs that are doing memory swapping with the disk and might adversely affect overall performance of the virtual system.

The obvious benefit is that a host can support more VMs provided that most of them will not consume their memory allocation most of the time. But in a system where most of the VMs are busy and consume most of their allocated memory, then ballooning might cause performance degradation. This just highlights the importance of memory capacity for any computer system.

5.2.3.2 Kernel Same-page Merging

Kernel Same-page Merging (KSM) [22], used by the KVM hypervisor, allows KVM guests to share identical memory pages. These shared pages are usually common libraries or other identical, high-use data. KSM allows for greater guest density of identical or similar guest operating systems by avoiding memory duplication.

The concept of shared memory is common in modern operating systems. For example, when a program is first started, it shares all of its memory with the parent program. When either the child or parent program tries to modify this memory, the kernel allocates a new memory region, copies the original contents and allows the program to modify this new region. This is known as copy on write.

KSM is a Linux feature which uses this concept in reverse. KSM enables the

kernel to examine two or more already running programs and compare their memory. If any memory regions or pages are identical, KSM reduces multiple identical memory pages to a single page. This page is then marked copy on write. If the contents of the page is modified by a guest virtual machine, a new page is created for that guest.

This is useful for virtualization with KVM. When a guest virtual machine is started, it only inherits the memory from the host `qemu-kvm` process. Once the guest is running, the contents of the guest operating system image can be shared when guests are running the same operating system or applications. KSM allows KVM to request that these identical guest memory regions be shared.

KSM provides enhanced memory speed and utilization. With KSM, common process data is stored in cache or in main memory. This reduces cache misses for the KVM guests, which can improve performance for some applications and operating systems. Secondly, sharing memory reduces the overall memory usage of guests, which allows for higher densities and greater utilization of resources.

In recent KVM versions, KSM is NUMA aware. This allows it to take NUMA locality into account while coalescing pages, thus preventing performance drops related to pages being moved to a remote node. It's highly recommended avoiding cross-node memory merging when KSM is in use. If KSM is in use, you should change the `/sys/kernel/mm/ksm/merge_across_nodes` tunable to `0` to avoid merging pages across NUMA nodes. This can be done with the command

```
virsh node-memory-tune --shm-merge-across-nodes 0
```

Kernel memory accounting statistics can eventually contradict each other after large amounts of cross-node merging. As such, numad can become confused after the KSM daemon merges large amounts of memory. If your system has a large amount of free memory, you may achieve higher performance by turning off and disabling the KSM daemon.

Two separate methods are normally used for controlling KSM:

- The `ksm` service, that starts and stops the KSM kernel thread.
- The `ksmtuned` service that controls and tunes the `ksm` service, dynamically managing same-page merging. `ksmtuned` starts the `ksm` service and stops the `ksm` service if memory sharing is not necessary. When new guests are created or destroyed, `ksmtuned` must be instructed with the `retune` parameter to run.

Both of these services are controlled with the standard service management tools.

The KSM Service

- The `ksm` service is included in the qemu-kvm package.
- When the `ksm` service is not started, Kernel same-page merging (KSM) shares only 2000 pages. This default value provides limited memory-saving benefits.
- When the `ksm` service is started, KSM will share up to half of the host system's main memory. Start the `ksm` service to enable KSM to share more memory.

```
# systemctl start ksm
Starting ksm:                                              [  OK  ]
```

The ksm service can be added to the default startup sequence. Make the ksm service persistent with the systemctl command.

```
# systemctl enable ksm
```

The KSM Tuning Service

The ksmtuned service fine-tunes the kernel same-page merging (KSM) configuration by looping and adjusting ksm. In addition, the ksmtuned service is notified by libvirt when a guest virtual machine is created or destroyed. The ksmtuned service has no options.

```
# systemctl start ksmtuned
Starting ksmtuned:                                         [  OK  ]
```

The ksmtuned service can be tuned with the retune parameter, which instructs ksmtuned to run tuning functions manually.

KSM Variables and Monitoring

Kernel same-page merging (KSM) stores monitoring data in the /sys/kernel/mm/ksm/ directory. Files in this directory are updated by the kernel and are an accurate record of KSM usage and statistics.

The variables in the list below are also configurable variables in the /etc/ksmtuned.conf file, as noted above.

Files in /sys/kernel/mm/ksm/:

full_scans

Full scans run.

merge_across_nodes

Whether pages from different NUMA nodes can be merged.

pages_shared

Total pages shared.

pages_sharing

Pages currently shared.

pages_to_scan

Pages not scanned.

pages_unshared

Pages no longer shared.

pages_volatile

Number of volatile pages.

run

Whether the KSM process is running.

sleep_millisecs

Sleep milliseconds.

These variables can be manually tuned using the `virsh node-memory-tune` command. For example, the following specifies the number of pages to scan before the shared memory service goes to sleep:

```
# virsh node-memory-tune --shm-pages-to-scan number
```

Deactivating KSM

Kernel same-page merging (KSM) has a performance overhead which may be too large for certain environments or host systems. KSM may also introduce side channels that could be potentially used to leak information across guests. If this

is a concern, KSM can be disabled on per-guest basis.

KSM can be deactivated by stopping the `ksmtuned` and the `ksm` services. However, this action does not persist after restarting. To deactivate KSM, run the following in a terminal as root:

```
# systemctl stop ksmtuned
Stopping ksmtuned:                                         [  OK  ]
# systemctl stop ksm
Stopping ksm:                                              [  OK  ]
```

Stopping the `ksmtuned` and the `ksm` deactivates KSM, but this action does not persist after restarting. Persistently deactivate KSM with the `systemctl` commands:

```
# systemctl disable ksm
# systemctl disable ksmtuned
```

When KSM is disabled, any memory pages that were shared prior to deactivating KSM are still shared. To delete all of the PageKSM in the system, use the following command:

```
# echo 2 >/sys/kernel/mm/ksm/run
```

After this is performed, the `khugepaged` daemon can rebuild transparent hugepages on the KVM guest physical memory. Using

`echo 0 >/sys/kernel/mm/ksm/run`

stops KSM, but does not unshare all the previously created KSM pages (this is the same as the # `systemctl stop ksmtuned` command).

Fortunately, as we have been able to ascertain, both mechanisms can be disabled by software, to obtain cleaner results.
Anyway, I report graphs for both conditions, with and without memory ballooning and kernel share page merging.

5.3 Collected data rendering

I present a synthesis of the measurements collected during this research, in particular I have organized 3D graphs to report on each axis respectively:

- time in range 0 to 60 seconds;
- virtual to physical resources ratio, referring to the percentage of overcommitting previously explained and graduated in 25% steps;
- branch miss to vcpu cycles ratio.

Values are derived from tests conducted on a compute node equipped with 40 cores and 128 GB ram.

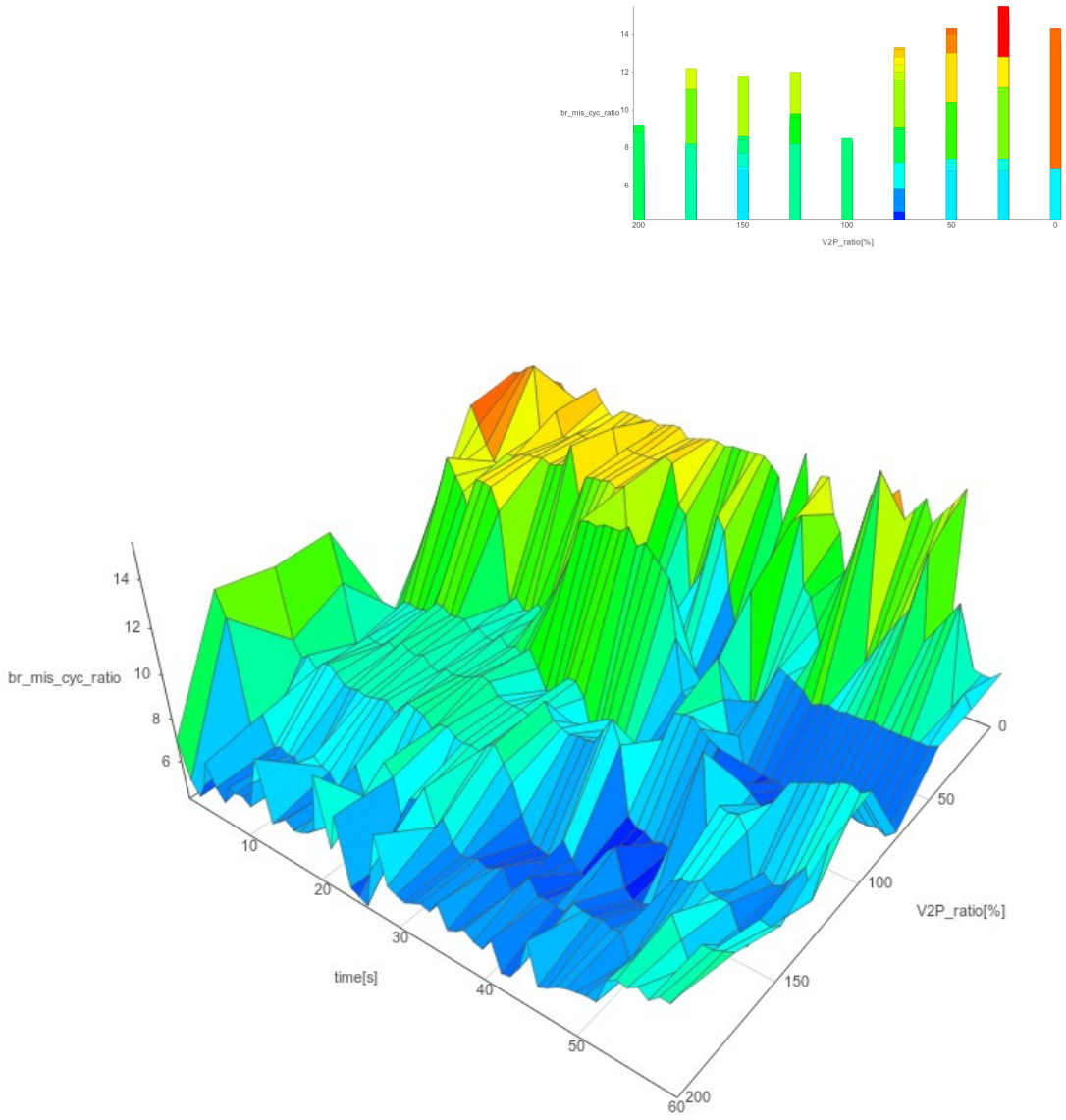

Figure 17. Host performance seen from vm-micro perspective: the noisy way

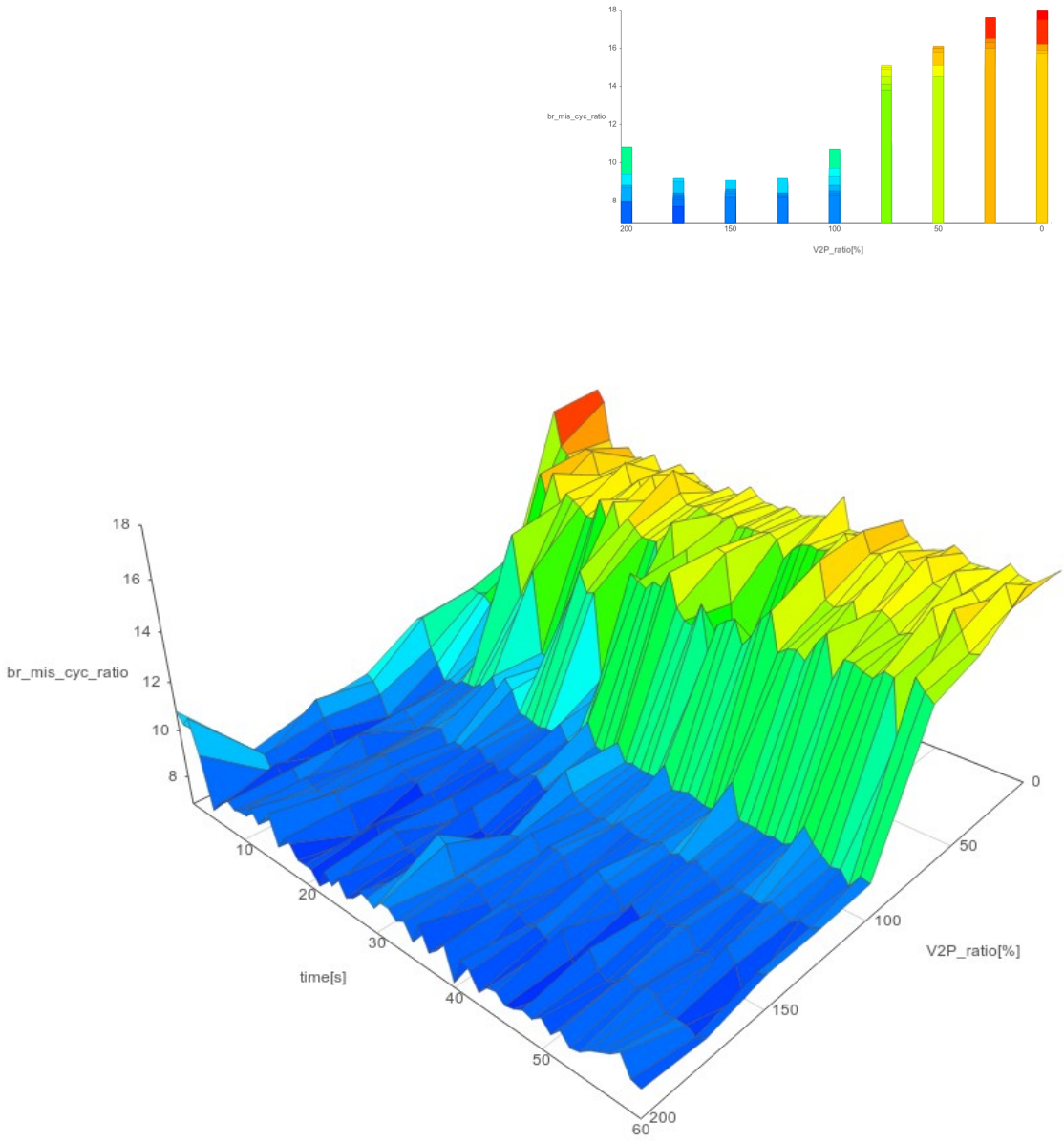

Figure 18. Host performance seen from vm-micro perspective without ballooning and KSM

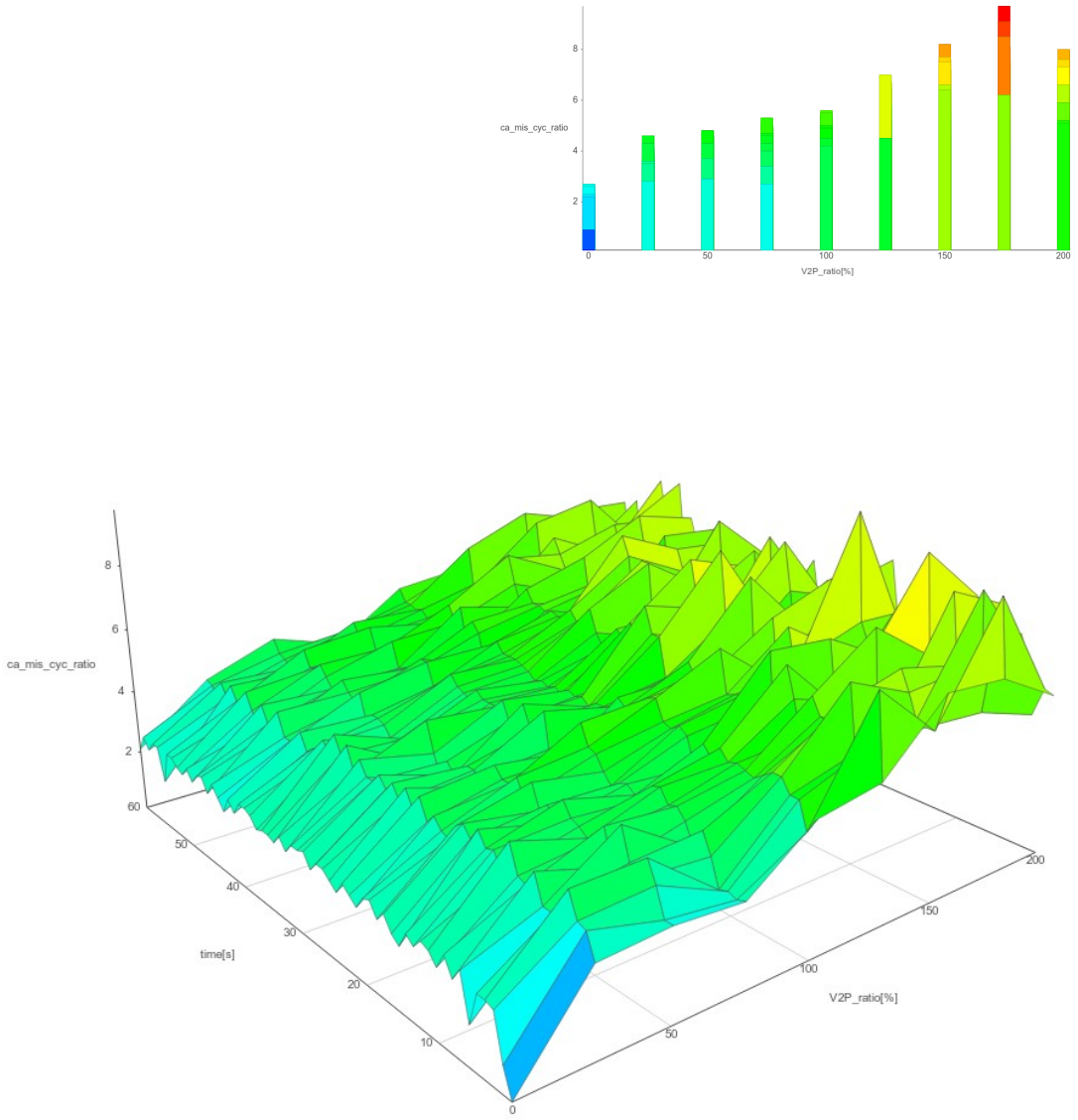

Figure 19. Cache_misses to cycle ratio seen from vm-micro perspective w/o ballooning and KSM

5.4 Data interpretation

Modern processors use pipelining to exploit parallelism and improve performance. Conditional branches in the instruction stream degrade performance by causing pipeline flushes. Branch prediction mechanisms [23] can overcome this limitation by predicting the outcome of the branch before its condition is resolved. As a result, instruction fetch is not interrupted as often and the window of instructions over which ILP (Instruction-level Parallelism) can be exposed increases. In fact, accurate branch predictors can eliminate over 90% of these pipeline stalls and are thus critical to realizing the performance potential of a processor.

Branch prediction accuracy is important because the new generation of processors have deeper pipelines, which result in larger misprediction penalties.

Most processors use dynamic branch prediction to predict branch directions. Dynamic predictors record and utilize information from previous runs of a static branch instruction to predict its outcome in the future. This requires additional hardware to store the branch history. These predictors dynamically adjust their prediction to match the changing behavior of a branch instruction as the program executes.

One aspect of branch prediction that has largely been ignored is the effect of context switches. In typical systems, several processes are in the active queue at any given time and they share the branch predictor structure. Each process runs for its allotted time slice and then yields the processor to allow another waiting process to execute. Unless steps are taken to change the state of the predictor

structure, it will contain stale information from the run of the previous process when the new process commences execution. Since different processes generally have completely different branch behaviors, reusing the stale information will increase the misprediction rate.
Several papers on branch prediction acknowledge the effects of context switching on branch prediction accuracy and on system performance [24] [25].

In the same way, for a virtualization host a context switch is the switching of the CPU from one process or thread to another. A guest operating system running in a virtual machine is executed by the host just as any other processes or threads running on the host are executed. When the host operating system receives a hardware interrupt, it generally suspends the progression of the current process on the CPU and starts servicing the interrupt. Once the interrupt has been serviced, either the current process or some other process (as decided by the scheduler of the host operating system) continues with its execution.
The guest operating system is scheduled in the same manner as any other process on the host. Context switches can occur during program execution for several reasons such as I/O requests, system calls, page faults, expiration of time slice etc. The frequency of these context switches depends on factors like the number of virtual machines active on a system, the types of applications executed, the operating system used and the scheduling scheme.

I have found that a good performance indicator in this context is represented by branch miss to vcpu cycles ratio (Fig.17 and 18), referring to branch miss as the number of branch misprediction events in a second and to vcpu cycles as the number of execution cycles the virtual cpu executed in a second. I would like to

remind that both quantities are measured from inside virtual machine vm-micro, using virtual performance counters, and so without any perception of host's intensity of saturation. The value of this ratio reported in graphs has been multiplied by 10^4 only for ranging and scaling purposes.

Even in the noisy mode with ballooning and KSM active, the behavioral pattern of the virtual system shows a trend that identifies with good approximation the state of over-committment of host's resources, since values exceeding 100% virtual to physical ratio.

The same thing cannot be said about cache_misses performance indicator (Fig. 19): in this case the cache_miss to cycles ratio presents a behavior proportional to P2V overcommitment ratio, but with a smoother profile that doesn't point out a threshold effect.

6 Summary and conclusion

In this work I analyzed the performance of a virtualization host in particular conditions of resource over-committing, from an innovative point of view as that of a virtual machine executed on the same host.

Normally this guest virtual machine isn't aware of its host's condition, in fact hypervisors are specifically designed and realized to isolate the execution environment of each hosted virtual machine, so that none of them could interfere with any other one in a crowded computing environment like a cloud node.

From a customer point of view, it could be indeed interesting to know if the purchased service levels are effectively respected by the cloud provider.

All the software measuring instruments used in this research are publicly available and free, and can be easily installed in a micro instance of virtual machine, rapidly deployable also in public clouds.

Actually the method described in this work has been applied only to KVM hosts and guests, and the results show a trend that identifies with good approximation the state of over-committment of host's resources.

Further studies should be conducted on other type of hypervisors, such as Xen or Vmware ESXi, principally to investigate the accessibility of virtual counters, even if I suppose the results should be comparable.

References

[1] Anatomy of a Linux hypervisor - Tim Jones - IBM developerWorks®, 2009

[2] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A Survey on Virtualization Technologies. RPE Report, pages 1–42, 2005

[3] Smith, James E., and Ravi Nair. "The architecture of virtual machines." *Computer* 38.5 (2005): 32-38.

[4] Gustavo Duarte. CPU Rings, Privilege, and Protection, 2008

[5] [Bha09] Nikhil Bhatia. Performance Evaluation of Intel EPT Hardware Assist, 2009

[6] Gil Neiger, Santoni Amy, Felix Leing, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. Intel Technology Journal, 10(03), 2006

[7] KVM – The kernel-based virtual machine – Timo Hirt – Red Hat Inc, 2010

[8] Qumranet. KVM - Kernel-based Virtualiztion Machine White paper, 2006

[9] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual machine monitor. In Proceedings of the Linux Symposium, Ottawa, 2007

[10] Rusty Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. SIGOPS Oper. Syst. Rev., 42(5):95–103, 2008

[11] Tim Jones, M. "Cloud computing with Linux." *2008-09-10]. http..//www, ibm. com/developerworks/library/I-cloudcomputing* (2009)

[12] Anatomy of an open source cloud - Tim Jones - IBM developerWorks®, 2012

[13] OpenStack - https://docs.openstack.org

[14] Serebrin, Benjamin, Daniel Hecht. "Virtualizing performance counters." *European Conference on Parallel Processing*. Springer, Berlin, Heidelberg, 2011

[15] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2

https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html

[16] AMD Bios and Kernel Developer Guide.
http://support.amd.com/us/Processor_TechDocs/31116.pdf

[17] Linux performance - http://www.brendangregg.com/perf.html

[18] Khandual, Anshuman. "Performance monitoring in linux kvm cloud environment." *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*. IEEE, 2012.

[19] Du, Jiaqing, Nipun Sehrawat, and Willy Zwaenepoel. "Performance profiling of virtual machines." *Acm Sigplan Notices* 46.7 (2011): 3-14

[20] Common Criteria for Information Technology Security Evaluation https://www.commoncriteriaportal.org/

[21] https://www.techopedia.com/definition/30466/memory-ballooning

[22] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/chap-ksm

[23] S. Pasricha and A. Veidenbaum, "Improving branch prediction accuracy in embedded processors in the presence of context switches," in *Computer Design, 2003. Proceedings. 21st International Conference on*, 2003, pp. 526–531

[24] Tse-Yu Yeh, Yale N. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction” Nineteenth ISCA, 1992
[25] Nicolas Gloy, Cliff Young, J. Bradley Chen, Michael D. Smith, “An Analysis of Dynamic Branch Prediction Schemes on System Workloads”, Proc. 23rd Annual ISCA, 1996