# Grid Operating Systems/Middlewares and New Virtualization Techniques

Matías Zabaljáuregui
matiasz@info.unlp.edu.ar

## 0. Introduction

*“As grid technology gets absorbed into enterprise fabrics, it could become inseparable from technologies such as* virtualization *and service-oriented architectures* (SOA) *and the creation of enterprise utilities. Virtualization could deliver the dream of grid computing...”*
(http://www.gridcomputingplanet.com/news/article.php/3651536)

This is what analysts say now a days. What we see is an inevitable convergence between these technologies. To run a job on a grid today, a user has to identify a set of platforms capable of running that job, with the right operating system, libraries and so on. Virtualization introduces a layer of abstraction, which means that instead of having to snoop out what resources are available and try to adapt a problem to use them, a user can describe a resource environment — or workspace — and expect it to be deployed on the grid on demand ("*infrastructure as a service"* is the new term for this capability).

Virtual machines and virtual appliances — together with distributed storage facilities and network overlays — look as though they will be able to map this kind of virtual workspace onto physical resources. Moreover, the promise is that they will be easy to define, test, install, transport and adjust on demand. Putting them together into 'virtual grids' should enable users to test them before the actual allocation of virtual resources is made.

Furthermore, if we do some quick research of the new Cloud Computing paradigm, we will find that it is a consequence of the underlying services provided by virtualization techniques. A group of virtualization and cloud-computing experts gathered at MIT's Emerging Technology conference (September 2008). Here is what they said:

*"The good news is that virtualization will become a critical part of an even larger part of most IT infrastructures as time goes on. The bad news is that it will do so as part of a larger movement toward cloud computing and will, in large part, disappear as a separate discipline. "*

In fact, virtualization, along with few other technological advances, is the one that actually unlocks cloud computing and changes the way IT is done in the industry. It has also shaped the way consumers consume computing power. This makes virtualization part and parcel of cloud computing paradigm and any attempts to consider it as a separate discipline seems to be unnecessary.

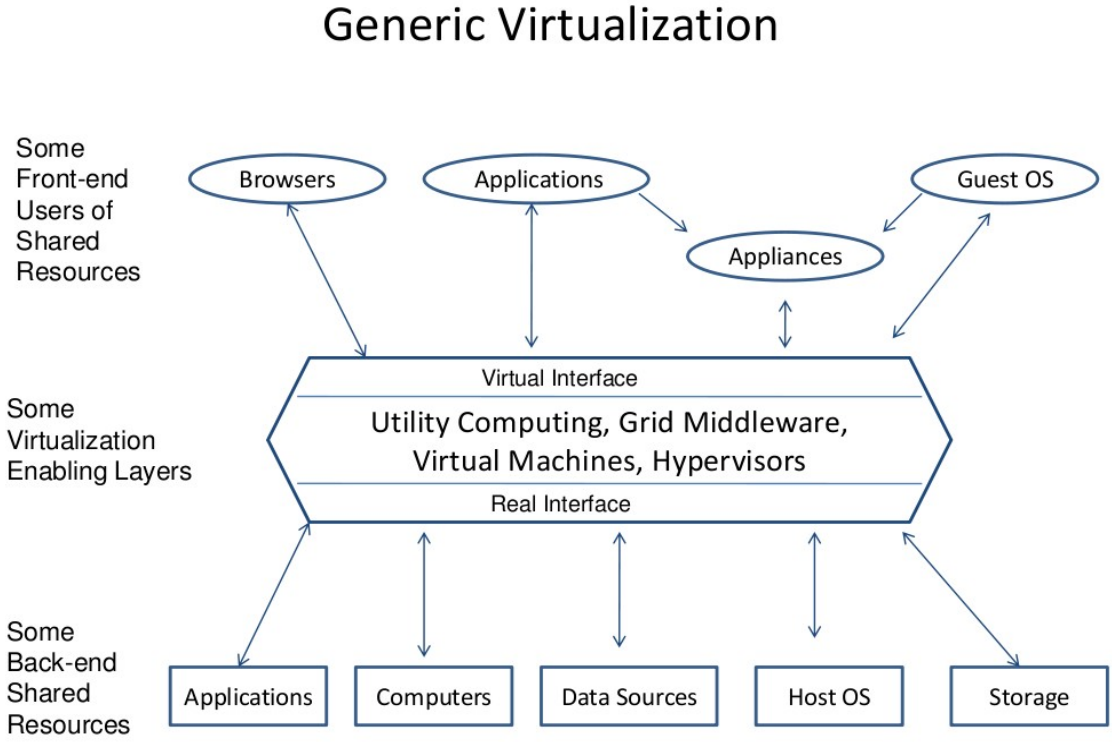
My PhD research work is related with a new virtualization technique called hybrid virtualization. I pretend to explore this novel approach combined with the recently mainstream adopted multicore CPU architectures. Since this report is intended to be the final report for a Grid related postgraduate course, I will talk about virtualization techniques from a Grid/Cloud Computing perspective. In other words, I will describe virtualization taking it as a way to allow the implementation of these new computation paradigms.

## 1. Motivation: grid and virtualization relationship

### 1.1. Understanding what “Virtualization” really means

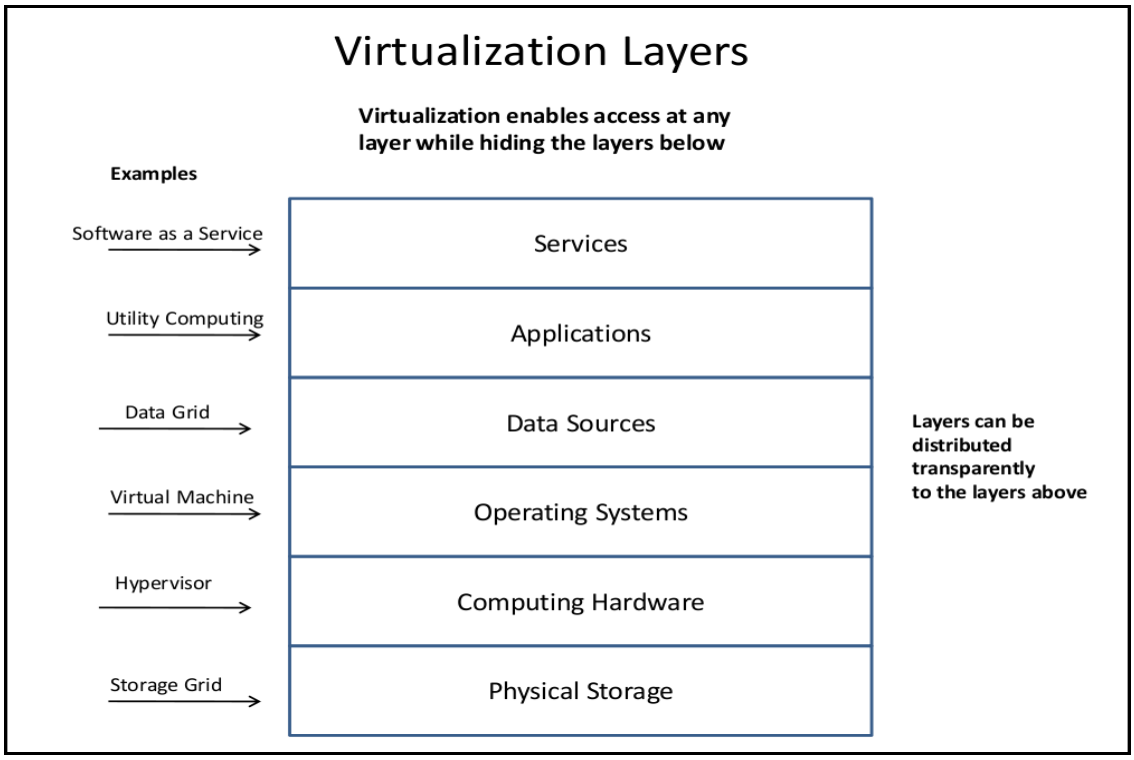
This section briefly introduces some virtualization concepts. I also make a review of a novel stacked approach to the virtualization environment, identifying the abstraction layers involved in these new techniques. Finally, I introduce a relatively new way of understanding virtualization through two dimensions, vertical and horizontal virtualization. None of these taxonomies are “standards” since these are very new ways of classifying these new concepts.

First of all, I'll introduce the actors of any virtualized scenario: the users, the shared resources and the virtualization layer that maps users and resources, as we can see in the diagram:




Evidently, this is an oversimplified way of seeing it, but this generic concept can be found in several layers of the architecture of any data center built on the present age. During the lasts years, all new kind of particular uses cases for this generic idea have appeared: Utility Computing, Grid Middleware, Virtual Machines, Hypervisors, etc. **The main goal of this section is to show how Grid computing and all other forms of virtualization belong to the same family.**

The first method we use to understand the variants of virtualization techniques is going through the data center architectural stack. The next diagram shows the virtualization layers existing today. Layers can be distributed transparently to the layers above. To the left of the picture you can see one example for every layer proposed (I will explain them later).



Next, I will introduce another way of classifying virtualization techniques: **Horizontal and Vertical Virtualization**. Horizontal Virtualization is virtualization across distributed back-end resources. Software as a Service , Utility Computing and Grids are examples of this dimension. Vertical Virtualization is virtualization across architectural layers. Examples of this are Virtual Machines, Hypervisors, Virtual Appliances.

For the sake of completeness, I will briefly explain the examples I mentioned before. This can also help to understand the difference between vertical and horizontal virtualization.

**Horizontal Virtualization Alternatives**

**a. Software as a Service (SaaS):** Software as a service makes applications available in a remote data center through a service-based interfaces available to multiple external organizations. Benefits: Reduced cost for software and infrastructure. Issues: Security across multiple uses.

**b. Utility Computing (Cloud):** Utility computing means that resources that are managed by a single organization are made available to multiple external organizations as necessary (on demand). Benefits: Reduced infrastructure cost. Issues: Accounting, Resource management.

**c. Computational Grids:** Transparent sharing of computational server resources among multiple groups across or within an enterprise. Benefits: Reduced cost of infrastructure. Issues: Cross-organization management

**d. Transactional Grids and Utilities:** Sharing distributed hardware and software platform resources to support high performance transactional applications. Benefits – Reduced cost for transactional capabilities. Issues: Lack of standards.

**e. Data Grid and Utilities:** Transparent sharing of data servers among multiple applications across multiple groups or within an enterprise. Benefits: Easier access to distributed data. Issues: Maintenance of metadata and data consistency.

**f. Storage Grids and Utilities:** Transparent sharing of distributed physical storage devices by multiple clients. Benefits: Reduced infrastructure costs. Issues: Performance

**Vertical Virtualization Alternatives**

**a. Virtual Machine Monitor within Host OS:** Virtual machine capabilities built on top of a specific operating system. Can be used to partition resources or to host guest operating systems. Benefits: Better utilization for resources. Issues: Performance.

**b. Virtual Machine Monitor (Hypervisor) on CPU:** Virtual machine capabilities built on top of a CPU not requiring a host operating system. Benefits: Better utilization of resources. Issues: Functionality.

**c. Application Virtualization:** Platform (CPU, OS) independent and controlled environment for running applications. Benefits: Portability. Issues: Performance.

**d. Virtual Appliances:** Pre-configured bundling of application and operating system capabilities into a module that can run on a virtual machine. Provides a means of rapidly deploying applications using OVF standard. Benefits: Ease of deployment. Managing evolving interdependencies across multiple appliances and physical environments.

The next diagram shows the Users, Shared Resources and Virtualization Layer (remember the actors from the first diagram) for every virtualization alternative mentioned before:

# Virtualization Alternatives

| Alternative | Shared Resource | Resource Users | Enabling Layer |
|---|---|---|---|
| Software as a Service | Application accessed as Web Services | Web clients paying per use | Multi-tenant architectures |
| Utility Computing (On Demand, Cloud ) | Distributed data center software and hardware | Multiple clients renting resources | Distributed resource and workload managers |
| Computational Grids | Computers across locations or organizations | Multiple groups sharing computing resources | Grid middleware |
| Transaction Grids (Fabrics) | Hardware and software within an organization | Enterprise applications | Fabric middleware |
| Data Grids | Data sources across locations or organizations | Multiple groups creating a shared data capability | Data source resource broker |
| Storage Grids or Utilities | Storage hardware | Multiple applications and databases | Storage broker |
| Application Virtualization | Execution environment | Processes | Run-time support |
| Virtual Server | OS and CPU | Applications running in multiple partitions | Virtual server support |
| Virtual Machine Monitor | CPU | Multiple OS running on CPU | Hypervisor possibly with CPU support |
| Virtual Appliance | CPU | Bundled application, OS, database | Application virtualization Packaging and run-time |

Until now, there has been a need for additional standards in both horizontal and vertical virtualization. A key question is the **integration of horizontal and vertical virtualization capabilities** (e.g. virtual appliances and grids). This combination, sometimes called **Distributed Virtualization,** enables the secure rapid deployment of applications in a distributed heterogeneous environment for net-centric operations. Experts say that the next few years will see extensive development of distributed virtualization architectures.

### 1.2. How is reacting the *grid community* to this infrastructure evolution

This section is intended to analyze the most recent publications regarding the tendency in grid computing and virtualization techniques to converge to some unique utility computing research field.

#### 1.2.1. What is the Open Grid Forum "doing"?

I will start with the main ideas from the new working group (WG) of the Open Grid Forum called **Grid & Virtualization Working Group.** This WG has the following goals:

1. Verification that within existing Grid standards the specifications are neutral to virtualized systems and resources . The request for "resources" should be satisfied either by "virtualized resources" or "physical resources" . The key questionis : **Is virtualization "transparent to the current Grid standards"?**

2. Explore how virtualization technologies can be exploited to better support Grid use cases. **Define the use cases / scenarios** wherein the Grid infrastructure is seen interacting with system virtualization platforms and making use of its capabilities .

3. Define the requirements to the Grid architecture for integration with system virtualization platforms.

They assume a classic interpretation of the virtualization word. More interesting for this work is the way they relate the key capabilities of virtualization and some grid common scenarios. For example, they remark the following, so-mentioned, virtualization functionalities:

- Creation of virtual systems on-demand: Specify the environment the application / jobs needs to run . The environment of the allocating can be pre-configured an persisted as images that can be activated on creation (multiple times if necessary) .
- Dynamic resizing: Change the configuration of virtual system .
- Isolation: Applications / jobs can run isolated from each other .
- Snapshotting: Suspending the virtual system and persisting the state which can be reactivated again.
- Migration: Movement of virtual system among host systems (physical systems) .

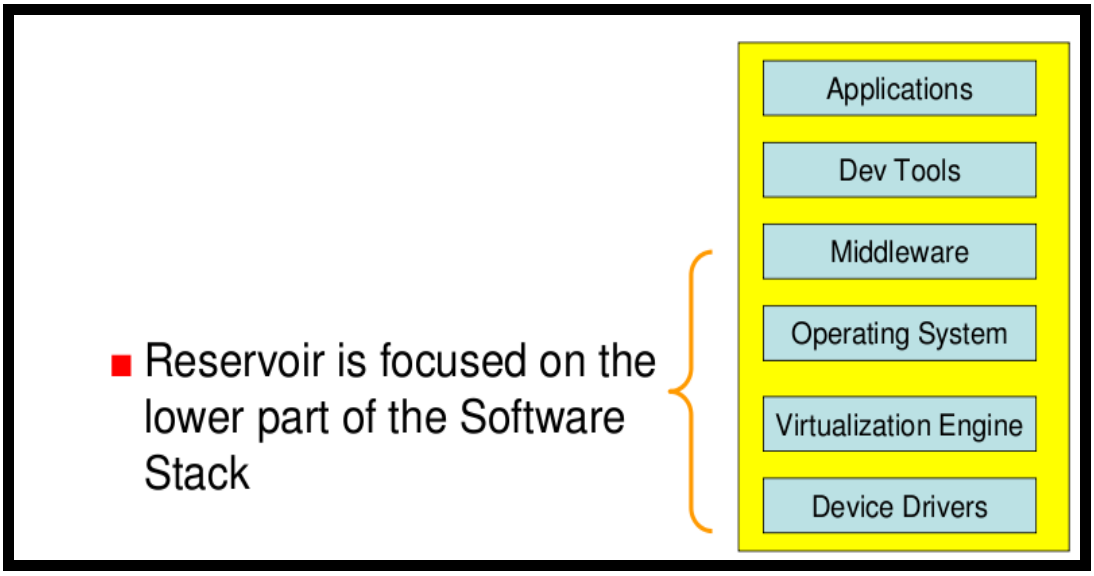
And they offer these selected Grid Use Cases, so they can show how virtualization could apply:

**Dynamically changing capacity requirements** : During runtime the job may require additional capacity (e.g. CPU capacity, Memory capacity, I/O bandwidth etc.). If the underlying physical system is able to serve the requirements, more capacity for the job / virtual system can be provided locally on the same physical system. If the requirements can be better fulfilled on another physical system the virtual system might be migrated. **It could be used the "dynamic system resizing" capability.**

**Dynamically changing capacity offering / availability :** Capacity availability may change in the physical system (e.g. CPU capacity, Memory capacity, I/O bandwidth etc.) because of recently freed resources by the completed jobs. In these situations available capacity can be utilized for the running jobs. Additional resources might become available on another physical systems which can be utilized. **It could be used "Live migration" of virtual system during runtime .**

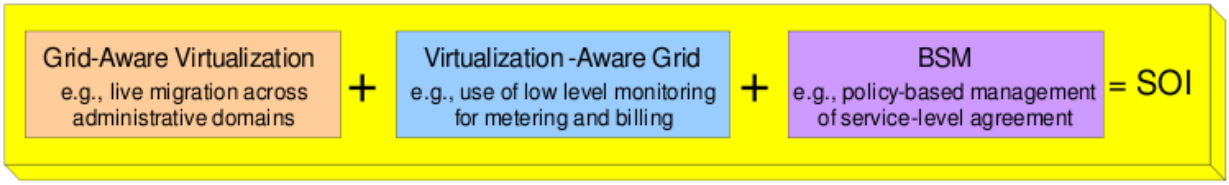
### 1.2.2. NESSI, the Networked European Software and Services Initiative.

This huge European project is talking about **Service-Oriented Economy**. They say that Service-Oriented Economy requires **Service-Oriented Infrastructure** (SOI) . On December 2007, they presented **Reservoir, the NESSI Strategic SOI Project .**




The main motivation is to deliver Services as Utilities . Reservoir is intended to be a Next-Generation Infrastructure for Service Delivery where Resources and services can be transparently and dynamically managed, provisioned and relocated virtually “without borders” .

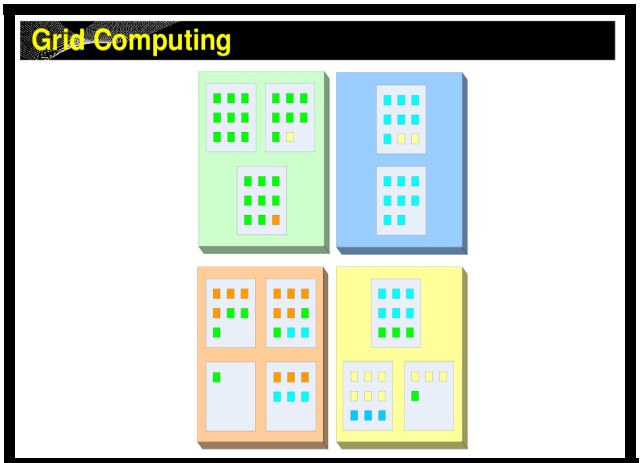
“*To accomplish the vision, we will pursue **deep integration of virtualization technologies with grid computing** . Integrating these key technologies - with supporting new techniques for business service management – underlies the vision of ubiquitous utility computing aimed to create basis for future service products* ”.



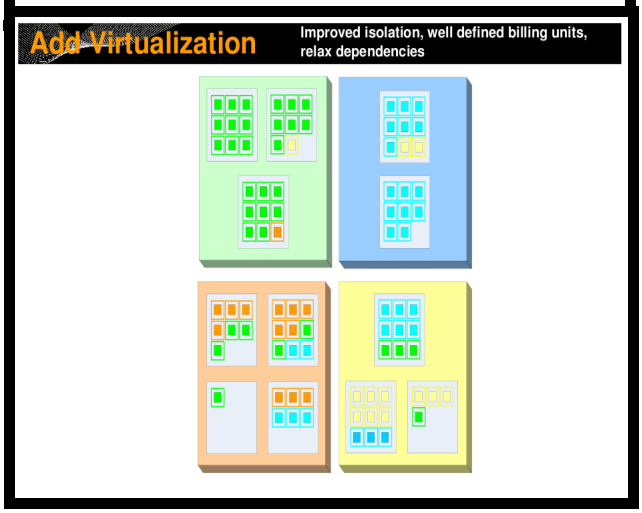

The main goals are:

- Definition of reference architecture for SOI
- Building upon the notion of **Virtual Execution Environment (VEE)** to provide support for the dynamic deployment and re-allocation of Virtual Machines and Java Service Containers

Here we can see some diagrams where they show the benefits of using this approach. The first one shows a classical grid scenario, where every site (big boxes with different colors) execute some processes (little colored boxes) on physical machines (gray boxes). Note that processes share the same color that the site that owns them.

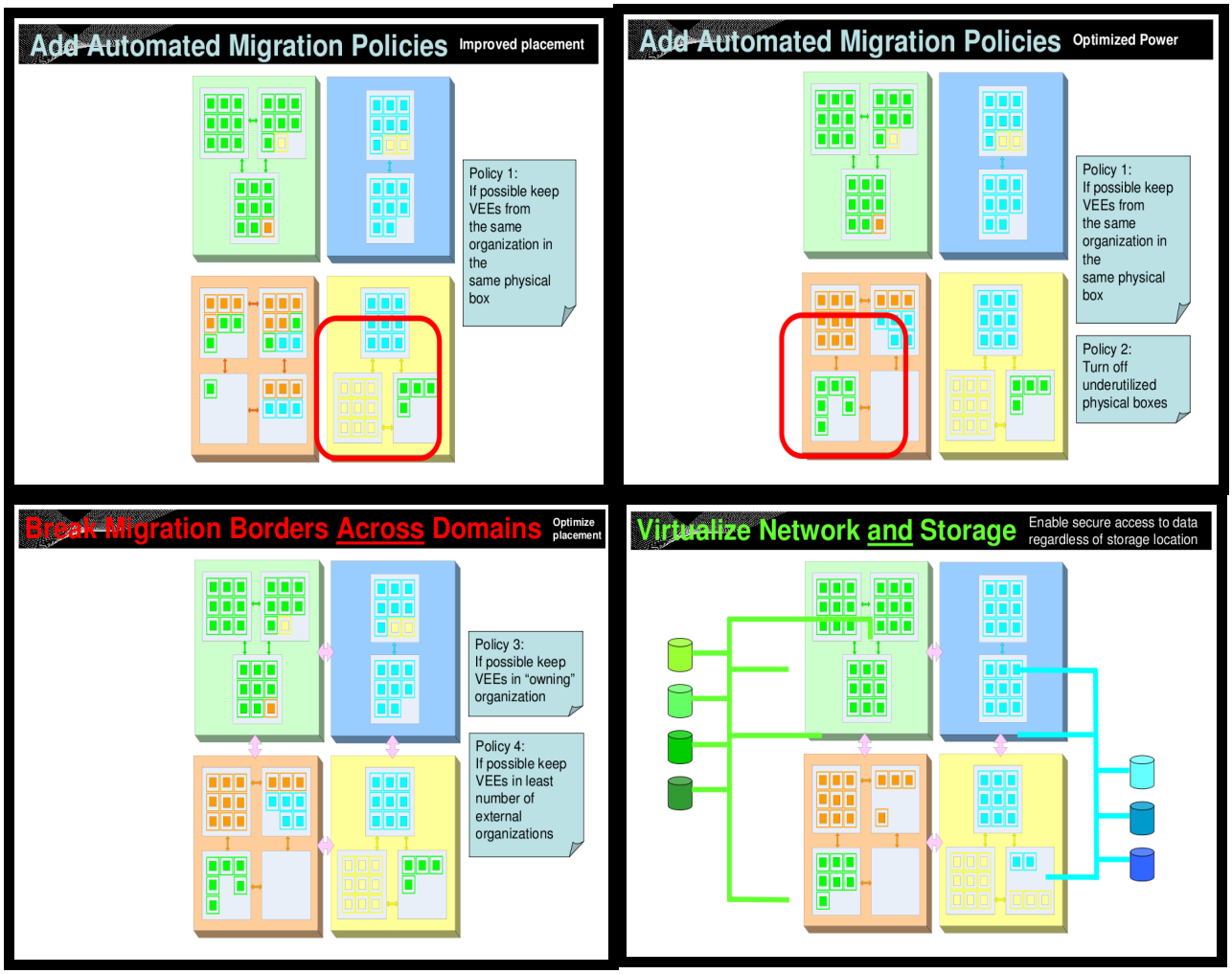



So, the first step is to add a virtualization layer, and start thinking in VEEs instead of simple processes. Some of the benefits are "improved isolation, well defined billing units, relax dependencies".




Then we can think, for example, in migration policies, automatically looking for the maximization of efficiency, security, power saving, etc. Since we can move not only the process itself, but the whole execution environment, we can freely distribute VEEs among physical machines, disregarding the specific requirement of the process.

The next four diagrams show some of the posibilities:




### 1.2.3. XtreemOS : The European Union Grid OS.

The third project I will comment about is XtreemOS, another huge collaboration project from the European Community working directly on the Operating System level to add Grid support.

*"The XtreemOS Grid Operating system will offer a native support for Virtual Organizations (VOs) in Linux. Linux will be modified (with kernel patches), extended (with kernel modules) and configured (enabling optional components and/or exploiting the framework provided by the Linux system) according to the need to support virtual organizations, and in order to provide the features and API needed to implement the XtreemOS services."*

As the state on the last technical report, "*An envision of virtual machine (VM) based approach for grid computing was justified in, followed by numerous VM-based grid research prototypes including the virtual workspace of Globus Toolkit. VMs provide outstanding isolation properties by instantiating independent guest environments on a host.* ***Besides this, VMs bring additional benefits such as full state checkpointing and migration, which facilitates the dynamic deployment of execution environments on demand.*** *Current VM-based grid computing approaches are almost based on utilizing open source or commercial VM software techniques. These techniques adopt* ***different levels of virtualization including emulation, paravirtualization and operating system-level extensions."***

Their technical documentation is very detailed, but I will focus the attention on the most interesting feature for this report. The policies specified by a VO, such as security, resource limitations, scheduling priorities and whatever attributes imposing regulations on how shared resources could be used by members of a VO, will be finally checked and enforced at resource nodes (sites). The main challenge here is the isolation among different users' accesses to the same node and also multiplexing usage of the same node from different VOs.

At the same time, the autonomy nature of a node must allow domain administrators to have the final control of resources which precedes any VO policies. This problem is also known as workspace management. Generally there are two approaches for protection and isolation of VO accesses in local operating systems: account mapping and virtual machines.

The account mapping approach is a simple but efficient way to isolate accesses to local nodes by different grid users. However, in this approach, it is complicate to maintain the mapping table of grid credentials to local Linux credentials and VO policies to local Linux capabilities, especially when VOs are dynamically changed. Their initial thoughts on improving the account mapping approach are:

*"Leveraging operating system-level virtualization techniques in Linux. Operating system-level virtualization has its advantages over emulation or paravirtualization based approaches in terms of low overhead for accommodating hundreds of virtual servers and low setup costs for instantiating virtual servers on one physical machine. It provides another way for separation and isolation of different VO accesses to the same local node. The challenge here is to prevent misbehaving virtual servers crudely exhausting resources from impacting good ones."*

### 1.3. Conclusions to this section

As I stated on the introduction section, and proved with with the previously mentioned projects, if we care about Grid Computing evolution, we should be aware of how virtualization techniques can leverage the actual grid middlewares and operating systems. This is the motivation of the second part of this report, which explain the techniques that are being used to implement x86 architecture virtualization support.

## 2. Introduction to Virtualization Techniques

### 2.1. x86 virtualization introduction

With x86 computer virtualization, a virtualization layer is added between the hardware and operating system as seen in this figure. This virtualization layer allows multiple operating system instances to run concurrently within virtual machines on a single computer, dynamically partitioning and sharing the available physical resources such as CPU, storage, memory and I/O devices.

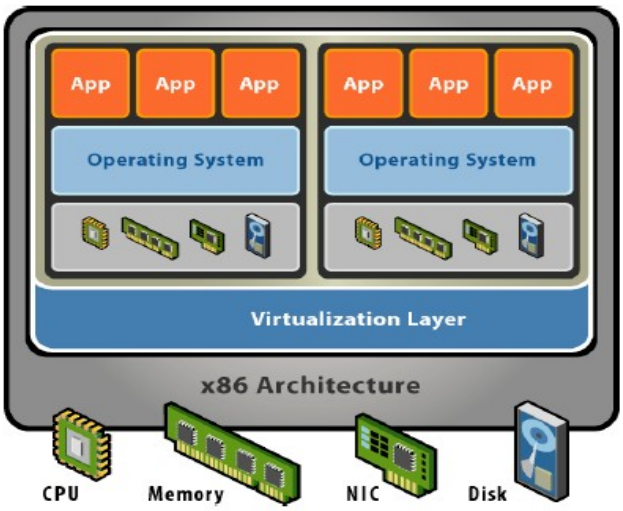



Figure 2 – x86 virtualization layer

For industry standard x86 systems, virtualization approaches use either a **hosted or a hypervisor architecture**. A hosted architecture installs and runs the virtualization layer as an application on top of an operating system and supports the broadest range of hardware configurations. In contrast, a hypervisor (bare-metal) architecture installs the virtualization layer directly on a clean x86-based system. Since it has direct access to the hardware resources rather than going through an operating system, a hypervisor is more efficient than a hosted architecture and delivers greater scalability, robustness and performance.

With a hypervisor approach, running directly on the hardware, the virtualization layer is the software responsible for hosting and managing all virtual machines on virtual machine monitors (VMMs), as depicted in the next figure. The functionality of the hypervisor varies greatly based on architecture and implementation. Each VMM running on the hypervisor implements the virtual machine hardware abstraction and is responsible for running a guest OS, which means that it **has to partition and share the CPU, memory and I/O devices to successfully virtualize the system.**

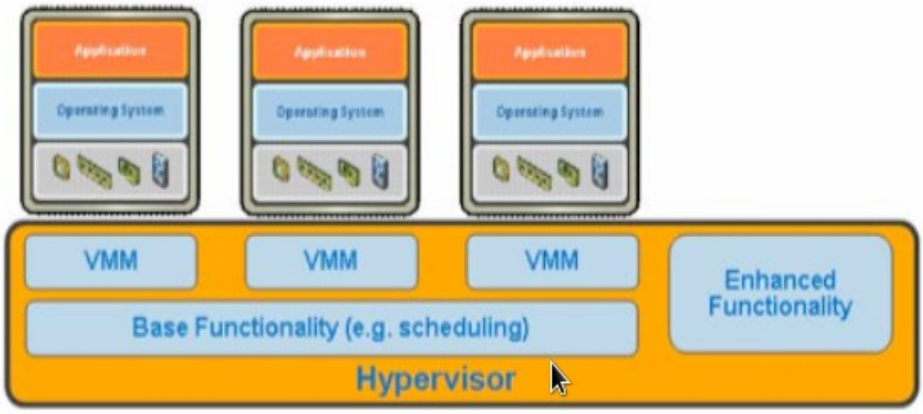



Figure 3 – The hypervisor manages virtual machine monitors that host virtual machines

### 2.2. Classical virtualization

The **Popek and Goldberg virtualization requirements** are a set of sufficient conditions for a computer architecture to efficiently support system virtualization. They were introduced by Gerald J. Popek and Robert P. Goldberg in their 1974 article *"Formal Requirements for Virtualizable Third Generation Architectures"*. Even though the requirements are derived under simplifying assumptions, they still represent a convenient way of determining whether a computer architecture supports efficient virtualization and provide guidelines for the design of virtualized computer architectures. They establishe three essential characteristics for system software to be considered a VMM:

> **1. fidelity**. Software on the VMM executes identically to its execution on hardware, barring timing effects.
>
> **2. Performance**. An overwhelming majority of guest instructions are executed by the hardware without the intervention of the VMM.
>
> **3. Safety.** The VMM manages all hardware resources.

In 1974, a particular VMM implementation style, trap-and-emulate, was so prevalent as to be considered the only practical method for virtualization. I will use the term **classically virtualizable** to describe an architecture that can be virtualized purely with trap-and-emulate. In this sense, **x86 is not classically virtualizable**, but it is virtualizable by Popek and Goldberg's criteria, using the new techniques described in next sections. In this section, I'll review the most important ideas from classical VMM implementations: **de-privileging, shadow structures and traces.**

**De-privileging**
In a classically virtualizable architecture, all instructions that read or write privileged state can be made to trap when executed in an unprivileged context. Sometimes the traps result from the instruction type itself (e.g., an out instruction), and sometimes the traps result from the VMM protecting structures that the instructions access (e.g., the address range of a memory-mapped I/O device).
A classical VMM executes guest operating systems directly, but at a reduced privilege level. The VMM intercepts traps from the de-privileged guest, and emulates the trapping instruction against the virtual machine state. This technique has been extensively described in the literature, and it is easily verified that the resulting VMM meets the Popek and Goldberg criteria.

**Primary and shadow structures**
By definition, the privileged state of a virtual system differs from that of the underlying hardware. The VMM's basic function is to provide an execution environment that meets the guest's expectations in spite of this difference. To accomplish this, the VMM derives shadow structures from guest-level primary structures. On-CPU privileged state, such as the page table pointer register or processor status register, is handled trivially: the VMM maintains an image of the guest register, and refers to that image in instruction emulation as guest operations trap.

However, off-CPU privileged data, such as page tables, may reside in memory. In this case, guest accesses to the privileged state may not naturally coincide with trapping instructions. For example, guest page table entries (PTEs) are privileged state due to their encoding of mappings and permissions. Dependencies on this privileged state are not accompanied by traps: every guest virtual memory reference depends on the permissions and mappings encoded in the corresponding PTE.

Such in-memory privileged state can be modified by any store in the guest instruction stream, or even implicitly modified as a side effect of a DMA I/O operation. Memory-mapped I/O devices present a similar difficulty: reads and writes to this privileged data can originate from almost any memory operation in the guest instruction stream.

For example, a critical component is memory virtualization. This involves sharing the physical system memory and dynamically allocating it to virtual machines. Virtual machine memory virtualization is very similar to the virtual memory support provided by modern operating systems. Applications see a contiguous address space that is not necessarily tied to the underlying physical memory in the system. The operating system keeps mappings of virtual page numbers to physical page numbers stored in page tables. All modern x86 CPUs include a memory management unit (MMU) and a translation lookaside buffer (TLB) to optimize virtual memory performance.

To run multiple virtual machines on a single system, another level of memory virtualization is required. In other words, one has to virtualize the MMU to support the guest OS. The guest OS continues to control the mapping of virtual addresses to the guest memory physical addresses, but the guest OS cannot have direct access to the actual machine memory. The VMM is responsible for mapping guest physical memory to the actual machine memory, and it uses shadow page tables to accelerate the mappings. As depicted by the red line in the next figure, the VMM uses TLB hardware to map the virtual memory directly to the machine memory to avoid the two levels of translation on every access. When the guest OS changes the virtual memory to physical memory mapping, the VMM updates the shadow page tables to enable a direct lookup.

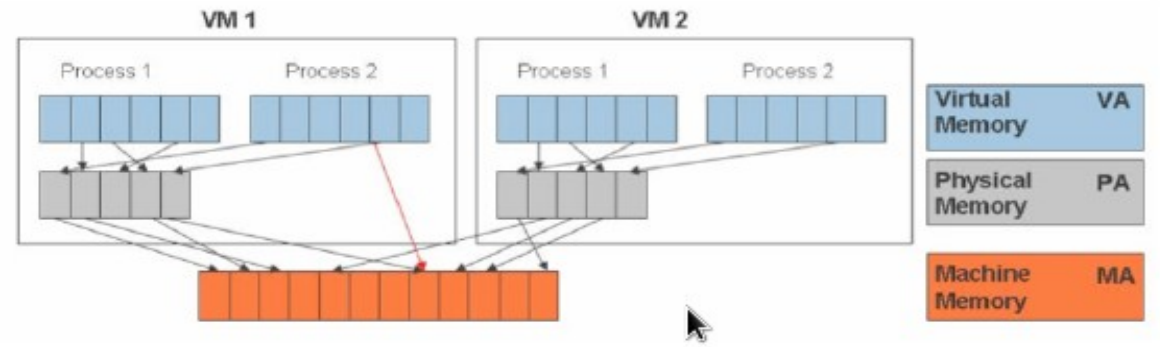



Figure 8 – Memory Virtualization

To maintain coherency of shadow structures, VMMs typically use hardware page protection mechanisms to trap accesses to in-memory primary structures. For example, guest PTEs for which shadow PTEs have been constructed may be write-protected. Memory-mapped devices must generally be protected for both reading and writing. This page-protection technique is known as tracing. Classical VMMs handle a trace fault similarly to a privileged instruction fault: by decoding the faulting guest instruction, emulating its effect in the primary structure, and propagating the change to the shadow structure.

The VMM manages its **shadow page tables as a cache of the guest page tables**. As the guest accesses previously untouched regions of its virtual address space, hardware page faults give control to the VMM. The VMM distinguishes true page faults, caused by violations of the protection policy encoded in the guest PTEs, from hidden page faults, caused by misses in the shadow page table. True faults are forwarded to the guest; hidden faults cause the VMM to construct an appropriate shadow PTE, and resume guest execution. The fault is “hidden” because it has no guest-visible effect.

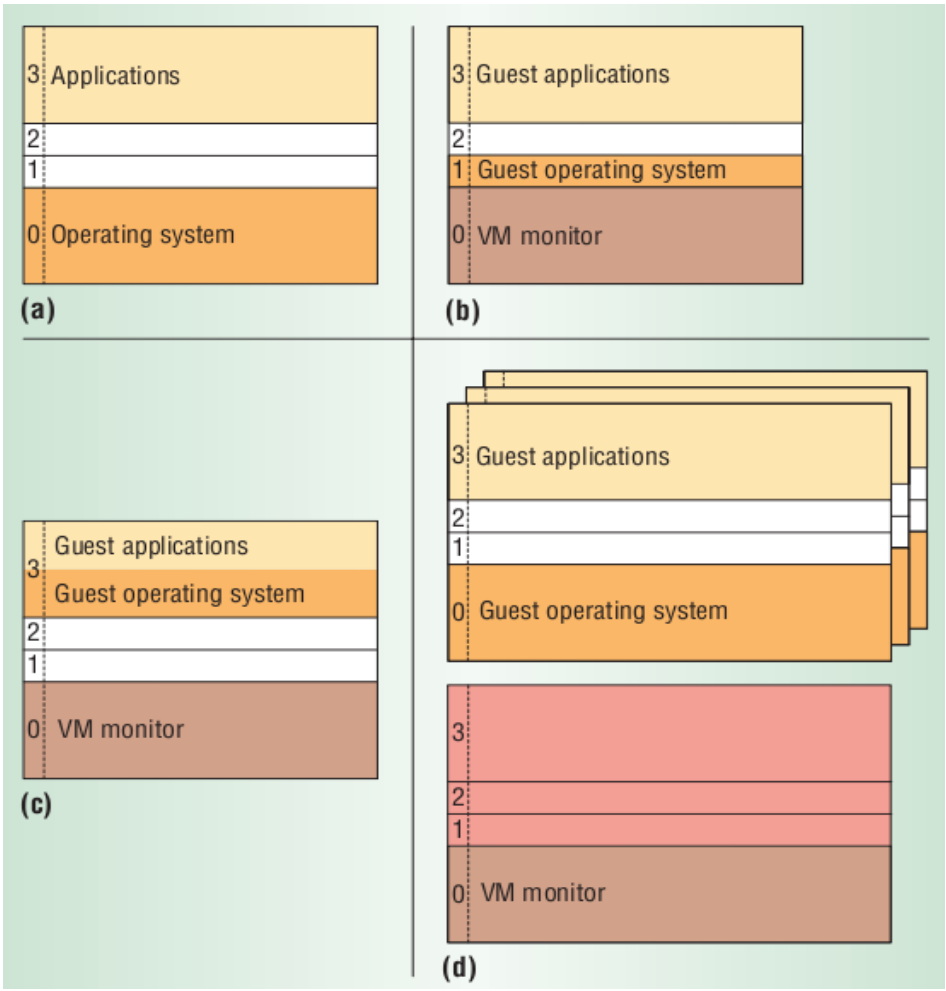
MMU virtualization creates some overhead for all virtualization approaches, but **this is the area where second generation hardware assisted virtualization will offer efficiency gains**, as we mention later.

### 2.3. The Challenges of x86 Hardware Virtualization

Intel microprocessors provide protection based on the concept of a 2-bit privilege level, using 0 for most privileged software and 3 for the least privileged. The privilege level determines whether privileged instructions, which control basic CPU functionality, can execute without fault; it also controls address space accessibility based on the configuration of the processor's page tables and, for IA-32, segment registers.

Most IA software uses only privilege levels 0 and 3, as figure a illustrates. For an OS to control the CPU, some of its components must run with privilege level 0. Because a VMM cannot allow a guest OS such control, a guest OS cannot execute at privilege level 0. Thus, IA-based VMMs must use ring deprivileging, a technique that runs all guest software at a privilege level greater than 0. A VM could deprivilege a guest OS by running it either at privilege level 1 (the 0/1/3 model) or at privilege level 3 (the 0/3/3 model). figures 1b and 1c illustrate these choices.

**Ring aliasing**
Ring aliasing refers to problems that arise when software is run at a privilege level other than the level for which it was written. An example in IA-32 is the PUSH instruction (which pushes its operand on the stack) when executed with the CS register (part of which is the current privilege level). A guest OS could easily determine that it is not running at privilege level 0.




**Address-space compression**
Operating systems expect to have access to the processor's full virtual address space, known as the linear-address space in IA-32. A VMM must reserve for itself some portion of the guest's virtual-address space. The VMM could run entirely within the guest's virtual-address space, which allows it easy access to guest data, although the VMM's instructions and data structures might use a substantial amount of the guest's virtual address space. Alternatively, the VMM could run in a separate address space, but even in that case the VMM must use a minimal amount of the guest's virtual-address space for the control structures that manage transitions between guest software and the VMM (For IA-32, these structures include the IDT and the GDT, which reside in the linear-address space.) The VMM must prevent guest access to those portions of the guest's virtual-address space that the VMM is using. Otherwise, the VMM's integrity could be compromised if the guest can write to those portions, or the guest could detect that it is running in a virtual machine if it can read them. Guest attempts to access these portions of the address space must generate transitions to the VMM, which can emulate or otherwise support them. The term address space compression refers to the challenges of protecting these portions of the virtual address space and supporting guest accesses to them.

**Nonfaulting access to privileged state**
Privilege-based protection prevents unprivileged software from accessing certain components of CPU state. In most cases, attempted accesses result in faults, allowing a VMM to emulate the desired guest instruction. However, the IA-32 architecture includes instructions that access privileged state and do not fault when executed with insucient privilege. For example, the IA-32 registers GDTR, IDTR, LDTR, and TR contain pointers to data structures that control CPU operation. Software can execute the instructions that write to, or load, these registers (LGDT, LIDT, LLDT, and LTR) only at privilege level 0. However, software can execute the instructions that read, or store, from these registers (SGDT, SIDT, SLDT, and STR) at any privilege level. If the VMM maintains these registers with unexpected values, a guest OS using the latter instructions could determine that it does not have full control of the CPU.

**Adverse impacts on guest transitions**
Ring deprivileging can interfere with the effectiveness of facilities in the IA-32 architecture that accelerate the delivery and handling of transitions to OS software. The IA-32 SYSENTER and SYSEXIT instructions support low-latency system calls. SYSENTER always effects a transition to privilege level 0, and SYSEXIT will fault if executed outside that privilege level. Ring deprivileging thus has the following implications: Executions of SYSENTER by a guest application will cause a transition to the VMM and not to the guest OS. The VMM must thus emulate every guest execution of SYSENTER. Execution of SYSEXIT by a guest OS will cause a fault to the VMM. Thus, the VMM must emulate every guest execution of SYSEXIT.

**Interrupt virtualization**
Providing support for external interrupts, especially regarding interrupt masking, presents some specific challenges to VMM design. The IA-32 architecture provides mechanisms for masking external interrupts, preventing their delivery when the OS is not ready for them. IA-32 uses the interrupt flag (IF) in the EflAGS register to control interrupt masking. A VMM will likely manage external interrupts and deny guest software the ability to control interrupt masking. Existing protection mechanisms allow such denial of control by ensuring that guest attempts to control interrupt masking will fault in the context of ring deprivileging. Such faulting can cause problems because some operating systems frequently mask and unmask interrupts. Intercepting every guest attempt to do so could significantly affect system performance.

Even if it were possible to prevent guest modifications of interrupt masking without intercepting each attempt, challenges would remain when a VMM has a \virtual interrupt" to deliver to a guest. A virtual interrupt should be delivered only when the guest has unmasked interrupts. To deliver virtual interrupts in a timely way, a VMM should intercept some, but not all, attempts by a guest to modify interrupt masking. Doing so could significantly complicate the design of a VMM.

**Ring compression**
Ring deprivileging uses translation privilege-based mechanisms to protect the VMM from guest software. IA-32 includes two such mechanisms: segment limits and paging. Because segment limits do not apply in 64-bit mode, paging must be used in this mode. Because IA-32 paging does not distinguish privilege levels 0-2, the guest OS must run at privilege level 3. Thus, the guest OS will run at the same privilege level as guest applications and will not be protected from them. This problem is called ring compression.

**Access to hidden state**
Some components of IA-32 CPU state are not represented in any software accessible register. Examples include the hidden descriptor caches for the segment registers. A segment-register load copies a referenced descriptor (from the GDT or LDT) into this cache, which is not modified if software later writes to the descriptor tables. IA-32 does not provide mechanisms for saving and restoring these hidden components of a guest context when changing VMs or for preserving them while the VMM is running.

### 2.4. Refinements to classical virtualization and "new" virtualization techniques

The type of workload significantly impacts the performance of the classical virtualization approach. During the first virtual machine boom, it was common for the VMM, the hardware, and all guest operating systems to be produced by a single company. These vertically integrated companies enabled researchers and practitioners to **refine classical virtualization using two orthogonal approaches**.

One approach exploited **flexibility in the VMM/guest OS interface**. Implementors taking this approach modified guest operating systems to provide higher-level information to the VMM. This approach relaxes Popek and Goldberg's fidelity requirement to provide gains in performance, and optionally to provide features beyond the bare baseline definition of virtualization, such as controlled VM-to-VM communication.

The other approach for refining classical VMMs **exploited flexibility in the hardware/VMM interface**. IBM's System 370 architecture introduced interpretive execution, a hardware execution mode for running guest operating systems. The VMM encodes much of the guest privileged state in a hardware-defined format, then executes the SIE instruction to "start interpretive execution." Many guest operations which would trap in a de-privileged environment directly access shadow fields in interpretive execution. While the VMM must still handle some traps, SIE was successful in reducing the frequency of traps relative to an unassisted trap-and-emulate VMM.
Both of these approaches have intellectual heirs in the present virtualization boom. **The attempt to exploit flexibility in the OS/VMM layer has been revived under the name paravirtualization. Meanwhile, x86 vendors are introducing hardware facilities inspired by interpretive execution.**

As clarified below, three alternative techniques now exist for handling sensitive and privileged instructions to virtualize the CPU on the x86 architecture:

- Full virtualization using binary translation
- OS assisted virtualization or paravirtualization
- Hardware assisted virtualization

**Technique 1 – Full Virtualization using Binary Translation**

The semantic obstacles to x86 virtualization can be overcome if the guest executes on an interpreter instead of directly on a physical CPU. The interpreter can prevent leakage of privileged state, such as the CPL, from the physical CPU into the guest computation and it can correctly implement non-trapping instructions like popf by referencing the virtual CPL regardless of the physical CPL. In essence, the interpreter separates virtual state (the VCPU) from physical state (the CPU).

However, while interpretation ensures fidelity and Safety, it fails to meet Popek and Goldberg's Performance bar: the fetch-decode-execute cycle of the interpreter may burn hundreds of physical instructions per guest instruction. Binary translation, however, can combine the semantic precision of interpretation with high performance, yielding an execution engine that meets all of Popek and Goldberg's criteria. A VMM built around a suitable binary translator can virtualize the x86 architecture and it is a VMM according to Popek and Goldberg.

Binary translation techniques allow the VMM to run in Ring 0 for isolation and performance, while moving the operating system to a user level ring with greater privilege than applications in Ring 3 but less privilege than the virtual machine monitor in Ring 0.

This approach translates kernel code to replace nonvirtualizable instructions with new sequences of instructions that have the intended effect on the virtual hardware. Meanwhile, user level code is directly executed on the processor for high performance virtualization. Each virtual machine monitor provides each Virtual Machine with all the services of the physical system, including a virtual BIOS, virtual devices and virtualized memory management.

This combination of binary translation and direct execution provides Full Virtualization as the guest OS is fully abstracted (completely decoupled) from the underlying hardware by the virtualization layer. The guest OS is not aware it is being virtualized and requires no modification. Full virtualization is the only option that requires no hardware assist or operating system assist to virtualize sensitive and privileged instructions. The hypervisor translates all operating system instructions on the fly and caches the results for future use, while user level instructions run unmodified at native speed.

**Technique 2 – OS Assisted Virtualization or Paravirtualization**

Paravirtualization refers to communication between the guest OS and the hypervisor to improve performance and efficiency. It involves modifying the OS kernel to replace nonvirtualizable instructions with *hypercalls* that communicate directly with the virtualization layer hypervisor. The hypervisor also provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping.

Paravirtualization is different from full virtualization, where the unmodified OS does not know it is virtualized and sensitive OS calls are trapped using binary translation. The value proposition of paravirtualization is in **lower virtualization overhead**. While it is very difficult to build the more sophisticated binary translation support necessary for full virtualization, modifying the guest OS to enable paravirtualization is relatively easy.

**Technique 3 – Hardware Assisted Virtualization**

In this section, we discuss recent architectural changes that permit classical virtualization of the x86. The discussion applies to both AMD's SVM and Intel's VT; the similarity between the two architectures is obvious from their respective manuals.
Hardware vendors are rapidly embracing virtualization and developing new features to simplify virtualization techniques. First generation enhancements include Intel Virtualization Technology (VT-x) and AMD's AMD-V which both target privileged instructions with a new CPU execution mode feature that allows the VMM to run in a new root mode below ring 0. Privileged and sensitive calls are set to automatically trap to the hypervisor,

removing the need for either binary translation or paravirtualization. Processors with Intel VT and AMD-V became available in 2006, so only newer systems contain these hardware assist features.

The hardware exports a number of new primitives to support a classical VMM for the x86. An in-memory data structure, which we will refer to as the virtual machine control block, or VMCB, combines control state with a subset of the state of a guest virtual CPU. A new, less privileged execution mode, **guest mode**, supports direct execution of guest code, including privileged code. We refer to the previously architected x86 execution environment as **host mode**. A new instruction, vmrun, transfers from host to guest mode. Upon execution of vmrun, the hardware loads guest state from the VMCB and continues execution in guest mode.

Guest execution proceeds until some condition, expressed by the VMM using control bits of the VMCB, is reached. At this point, the hardware performs an exit operation, which is the inverse of a vmrun operation. On exit, the hardware saves guest state to the VMCB, loads VMM-supplied state into the hardware, and resumes in host mode, now executing the VMM. Diagnostic fields in the VMCB aid the VMM in handling the exit; e.g., exits due to guest I/O provide the port, width, and direction of I/O operation. After emulating the effect of the exiting operation in the VMCB, the VMM again executes vmrun, returning to guest mode.

The VMCB control bits provide some flexibility in the level of trust placed in the guest. For instance, a VMM behaving as a hypervisor for a general-purpose OS might allow that OS to drive system peripherals, handle interrupts, or build page tables. However, when applying hardware assistance to pure virtualization, the guest must run on a shorter leash. The hardware VMM programs the VMCB to exit on guest page faults, TLB flushes, and address-space switches in order to maintain the shadow page tables; on I/O instructions to run emulated models of guest peripherals; and on accesses to privileged data structures such as page tables and memory-mapped devices.

**Hardware VMM implementation**
The hardware extensions provide a complete virtualization solution, essentially prescribing the structure of our hardware VMM (or indeed any VMM using the extensions). When running a protected mode guest, the VMM fills in a VMCB with the current guest state and executes vmrun. On guest exits, the VMM reads the VMCB fields describing the conditions for the exit, and vectors to appropriate emulation code. Most of this emulation code is shared with the software VMM. It includes peripheral device models, code for delivery of guest interrupts, and many infrastructure tasks such as logging, synchronization and interaction with the host OS. Since current virtualization hardware does not include explicit support for MMU virtualization, the hardware VMM also inherits the software VMM's implementation of the shadowing technique described previously.

**Example operation: process creation**
Consider a UNIX-like operating system running in guest mode on the hardware VMM, about to create a process using the fork(2) system call.

**a)** A user-level process invokes fork(). The system call changes the CPL from 3 to 0. Since the guest's trap and system call vectors are loaded onto the hardware, the transition happens without VMM intervention.

**b)** In implementing fork, the guest uses the "copy-on-write" approach of write-protecting both parent and child address spaces. Our VMM's software MMU has already created shadow page tables for the parent address space, using traces to maintain their coherency. Thus, each guest page table write causes an exit. The VMM decodes the exiting instruction to emulate its effect on the traced page and to reflect this effect into the shadow page table. By updating the shadow page table, the VMM write-protects the parent address space in the hardware MMU.

**c)** The guest scheduler discovers that the child process is runnable and context switches to it. It loads the child's page table pointer, causing an exit. The VMM's software MMU constructs a new shadow page table and points the VMCB's page table register at it.

**d)** As the child runs, it touches pieces of its address space that are not yet mapped in its shadow page tables. This causes hidden page fault exits. The VMM intercepts the page faults, updates its shadow page table, and resumes guest execution.

**e)** As both the parent and child run, they write to memory locations, again causing page faults. These faults are true page faults that reflect protection constraints imposed by the guest. The VMM must still intercept them before forwarding them to the guest, to ascertain that they are not an artifact of the shadowing algorithm.

**2.5. Conclusions to this section**

**The VT and SVM extensions make classical virtualization possible on x86**. The resulting performance depends primarily on the frequency of exits. A guest that never exits runs at native speed, incurring near zero overhead. However, this guest would not be very useful since it can perform no I/O. If, on the other hand, every instruction in the guest triggers an exit, execution time will be dominated by hardware transitions between guest and host modes. Reducing the frequency of exits is the most important optimization for classical VMMs.

To help avoid the most frequent exits, x86 hardware assistance includes ideas similar to the s/370 interpretive execution facility discussed above. Where possible, privileged instructions affect state within the virtual CPU as represented within the VMCB, rather than unconditionally trapping. Consider again popf. A naive extension of x86 to support classical virtualization would trigger exits on all guest mode executions of popf to allow the VMM to update the virtual “interrupts enabled” bit. However, guests may execute popf very frequently, leading to an unacceptable exit rate. Instead, the VMCB includes a hardware-maintained shadow of the guest %eflags register. When running in guest mode, instructions operating on %eflags operate on the shadow, removing the need for exits.

The exit rate is a function of guest behavior, hardware design, and VMM software design: a guest that only computes never needs to exit; hardware provides means for throttling some exit types; and VMM design choices, particularly the use of traces and hidden page faults, directly impact the exit rate as shown with the fork example above.

The remaining performance gap is due to the “stateless” nature of the hardware VMM: after resuming a guest in direct hardware-assisted execution, the VMM has little idea what state the guest is in when execution returns to the VMM. So the VMM incurs software overheads reconstructing guest state by reading VMCB fields (handling a typical exit requires ten vmreads) and in some cases decoding the exiting instruction. While improvements on this state reconstruction software are certainly possible, a complete elimination of it is unlikely. “Stateless” VMM operation is characteristic of hardware-assisted direct execution. Thus, the opportunities for making exits faster, in both hardware and software, are limited.

But most of the difficult cases for the hardware VMM examined before surround MMU virtualization. Second generation hardware assist technologies are in development that will have a greater impact on virtualization performance while reducing memory overhead. Both AMD and Intel have announced future development roadmaps, including hardware support for memory virtualization **(AMD Nested Page Tables [NPT] and Intel Extended Page Tables [EPT]) as well as hardware support for device and I/O virtualization (Intel VT-d, AMD IOMMU).** NPT/EPT will provide noticeable performance improvements for memory-remapping intensive workloads by removing the need for shadow page tables that consume system memory.

In both schemes, the VMM maintains a hardware-walked “nested page table” that translates guest physical addresses to host physical addresses. This mapping allows the hardware to dynamically handle guest MMU operations, eliminating the need for VMM interposition. The operation of this scheme is illustrated in figure 6. While running in hardware-assisted guest execution, the TLB contains entries mapping guest virtual addresses all the way to host physical addresses. The process of filling the TLB in case of a miss is somewhat more complicated than that of typical virtual memory systems. Consider the case of a guest reference to virtual address V that misses in the hardware TLB:

1. The hardware uses the guest page table pointer (%cr3) to locate the top level of the guest’s hierarchical page table.

2. %cr3 contains a guest physical address, which must be translated to a host physical address before dereferencing. The hardware walks the nested page table for the guest’s %cr3 value to obtain a host physical pointer to the top level of the guest’s page table hierarchy.

3. The hardware reads the guest page directory entry corresponding to guest virtual address V .

4. The PDE read in step 3 also yields a guest physical address, which must also be translated via the nested page table before proceeding.

5. Having discovered the host physical address of the final level of the guest page table hierarchy, the hardware reads the guest page table entry corresponding to V . In our example, this PTE points to guest physical address X, which is translated via a third walk of the nested page table, e.g. to host physical address Y.

6. The translation is complete: virtual address V maps to host physical address Y . The page walk hardware can now fill the TLB with an appropriate entry (V, Y ) and resume guest execution, all without software intervention. For an M -level guest page table on an N -level nested page table, a worst-case TLB miss requires M N memory accesses to satisfy. We are, however, optimistic about this potential problem. The same microarchitectural implementation techniques that make virtual memory perform acceptably (highly associative, large, multilevel TLBs, caching) should apply at least as well to the nested page table. Thus, nested paging holds the promise of eliminating trace overheads and allowing guest context switches without VMM intervention. By resolving the most important sources of overhead in current VMMs, nested paging hardware should easily repay the costs of (slightly) slower TLB misses.

**3. References**


Distributed Virtualization for Net-Centric Operations Draft
Bob Marcus
Approved for Public Release Distribution Unlimited
NCOIC-STKOUTRCH-PLEN-2007-DEC

OPEN GRID FORUM
Grid & Virtualization Working Group
OGF21 gridvirt-wg
Erol Bozak, Chair SAP, Development Architect
Wolfgang Reichert, Co-Chair, IBM, Senior Technical Staff Member

NETWORKED EUROPEAN SOFTWARE & SERVICES INITIATIVE
Reservoir – Resource and Service Virtualisation w/out Borders
Dr. Yaron Wolfsthal, IBM Haifa Research Lab
General Assembly 2007
Brussels – 11th & 12th December 200 7

XtreemOS Integrated Project
BUILDING AND P ROMOTING A LINUX-BASED O PERATING S YSTEM
TO S UPPORT V IRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS
Linux XOS Specification

A Comparison of Software and Hardware Techniques for x86 Virtualization
Keith Adams, VMware kma@vmware.com
Ole Agesen, VMware agesen@vmware.com

http://www.vmware.com/
http://chucksblog.typepad.com/chucks_blog/2008/06/post.html
http://www.serverwatch.com/news/article.php/3651786
http://chucksblog.typepad.com/chucks_blog/2006/12/grid_and_virtua.html
http://www.gridvm.org/
http://www.globusconsortium.org/journal/20060208/crosby.html
http://weblog.infoworld.com/gridmeter/archives/2006/03/grid_and_virtua.html
http://www.reservoir-fp7.eu/
http://www.techworld.com/news/index.cfm?RSS&NewsID=11332
http://imllorente.dsa-research.org/?p=14
http://xtreemos.org/
http://www.cio.com/article/print/451469